# DFT and FFT

C. Kankelborg

Rev. February 3, 2014

## 1 Introduction

The Fourier transform is a powerful tool in the solution of linear systems, including:

- Inhomogeneous ODEs (e.g. frequency response, impulse response)
- Inhomogeneous PDEs (e.g. scattering, diffraction, diffusion)
- Linear integral equations (e.g. deconvolution, tomography)

All of these applications can be realized numerically on coordinate grids in 1 or more dimensions using the *discrete Fourier transform* (DFT), typically implemented as a *fast fourier transform* (FFT). In addition to the above applications, the FFT offers a computationally efficient approach to a wide range of signal and image processing tasks:

- Power spectral estimation and peak bagging
- Ideal interpolation of time series and images
- Digital filtering
- Compression algorithms

This paper summarizes some important practical aspects of using the FFT for newcomers or old hands who have grown rusty. For those seeking a more comprehensive treatment of continuous or discrete Fourier transforms, I recommend Ronald Bracewell's classic *The Fourier Transform and Its Applications*.

# 2 Definition and Properties

## 2.1 DFT and its Inverse

Given a time series $f_n$ for uniformly sampled times $t_n = (n-1)\,\Delta t$, where $1 \le n \le N$, the DFT is defined as follows:

$$\tilde{f}_k = \sum_{n=1}^{N} f_n \, e^{-2\pi i (k-1)(n-1)/N}. \tag{1}$$

I have reluctantly used the awkward $n-1$ and $k-1$ instead of just $n$ and $k$ because MATLAB/Octave uses unit-offset arrays (indices $1..N$). The inverse transform is:

$$f_n = \frac{1}{N} \sum_{k=1}^{N} \tilde{f}_k \, e^{2\pi i (k-1)(n-1)/N}. \tag{2}$$

The frequency index $k$ takes on $N$ discrete values. Therefore arrays $f$ and $\tilde{f}$ contain an equal number of (in general, complex) elements. *In principle, there is no loss of information with the transform or its inverse.* In practice, there is roundoff error, as the following example demonstrates. The example is also a good illustration of the DFT normalization (table 1).

```
octave:16> foo = [1, 2, 1, 0, -1, 0, -1, 3]
foo =

   1   2   1   0  -1   0  -1   3

octave:17> foof = fft(foo)
foof =

 Columns 1 through 3:

   5.00000 + 0.00000i   5.53553 - 1.29289i   0.00000 + 1.00000i

 Columns 4 through 6:

  -1.53553 + 2.70711i  -5.00000 + 0.00000i  -1.53553 - 2.70711i

 Columns 7 and 8:

   0.00000 - 1.00000i   5.53553 + 1.29289i

octave:18> foo2 = ifft(foof)
```

```
foo2 =

 Columns 1 through 5:

   1.0000e+00   2.0000e+00   1.0000e+00  -2.2204e-16  -1.0000e+00

 Columns 6 through 8:

  -2.2204e-16  -1.0000e+00   3.0000e+00
```

## 2.2 DFT Properties

The DFT is useful because it possesses all the useful properties of the Fourier transform. Some examples are listed in table 1. There also exist analogues to the Fourier scaling, translation, differentiation and integration theorems. Like the inverse, all of these hold exactly in the absence of roundoff error.

Table 1: Some properties of the DFT.

| | |
|---|---|
| **Normalization** | $\tilde{f}_0 = \sum_n f_n$ |
| **Symmetry** | real, even $\iff$ real, even |
| of transform | imaginary, even $\iff$ imaginary, even |
| pairs | real, odd $\iff$ imaginary, odd |
| **Convolution** | $\mathrm{DFT}[f * g] = \tilde{f}\,\tilde{g}$ |
| | $\mathrm{DFT}[fg] = \left(\frac{1}{N}\right) \tilde{f} * \tilde{g}$ |
| **Correlation** | $\mathrm{DFT}[f \otimes g] = \tilde{f}^{*}\tilde{g}$ |
| | $\mathrm{DFT}\left[f^{*} g\right] = \left(\frac{1}{N}\right) \tilde{f} \otimes \tilde{g}$ |
| **Parseval's theorem** | $\sum_n {f_n}^{*} f_n = \left(\frac{1}{N}\right) \sum_k {\tilde{f}_k}^{*} \tilde{f}_k$ |

# 3 Being Discrete

Though the DFT works much like the Fourier transform, there are some subtleties intrinsic to working with discrete data over a finite interval.

## 3.1 Finite Domain

Since the time domain is of finite extent, and the basis functions all satisfy periodic boundary conditions over the domain, the DFT is really more like a

Fourier series than a Fourier transform. The time series $f_n$ is therefore treated as if it corresponds to a periodic function. Consequently, a discontinuity (in $f$ or its derivatives) from $f_N$ to $f_1$ can produce artifacts in $\tilde{f}$. For this reason, *detrending* and *windowing* are commonly used. See comments in §§ 4.2, **??**, and 5.

## 3.2 Nyquist Frequency

The shortest period that can be meaningfully represented in our time series is $2\,\Delta t$. This corresponds to $k = \frac{N}{2} + 1$. The corresponding highest frequency is called the Nyquist frequency,

$$\nu_c = \frac{1}{2\,\Delta t}. \tag{3}$$

## 3.3 Arrangement of Frequencies

First-time users of the DFT may be surprised by the order in which the frequencies are stored (figure 1). Where do the so-called "negative frequencies" come from? We will see that this is a natural consequence of the definition of the DFT.

The left half of the frequency diagram, from DC (zero frequency) to the Nyquist frequency, looks reasonable. In physical units, the frequencies are evidently

$$\nu = \frac{k-1}{N\,\Delta t}. \tag{4}$$

The Nyquist frequency (as we saw in the last section) occurs only about halfway though the progression of frequencies. How can we physically interpret a frequency "higher" than the Nyquist frequency? The range of interest is

$$\frac{N}{2} + 1 < k \leq N.$$

On the above range, let us use a new variable $k'$:

$$(k-1) = N - (k'-1), \quad 2 \leq k' < \frac{N}{2} + 1$$

In terms of the new variable, the DFT is

$$\tilde{f}_k = \sum_{n=1}^{N} f_n e^{-2\pi i[N-(k'-1)](n-1)/N} = \sum_{n=1}^{N} f_n e^{-2\pi i[-(k'-1)](n-1)/N}.$$

It is now evident that $k'$ represents a negative frequency, in direct analogy to equation 4:

$$\nu = \frac{-(k'-1)}{N\,\Delta t}, \quad -\nu_c < \nu < 0.$$

Note that the function $e^{-i\omega t}$ is orthogonal to $e^{i\omega t}$. Therefore the negative frequencies cannot be ignored in general. However, if the signal $f$ is known to be real-valued, then the information in the negative frequencies is redundant. Questions for study:

1. For a positive frequency of index $k$, what is the index of the corresponding negative frequency?

2. If $f$ is real, and the Fourier coefficient $\tilde{f}_k$ is known, what is the Fourier coefficient for the corresponding negative frequency?

3. Sketch the waveforms of the sine and cosine at the Nyquist frequency. When these are sampled, can they both be measured?

4. Why is there only one coefficient in the DFT at the Nyquist frequency? Does it correspond to a positive frequency, or a negative one?

Following is another simple example using Octave. In the first case, `v` is a signal at the Nyquist frequency, but with a DC offset (mean value) of $\frac{1}{2}$. The transform therefore contains two Kronecker delta functions, one in the first element and one in element 5. In the second part of the example, the frequency is half the Nyquist frequency. Since the original signal is real, the values in the positive frequency bins are complex conjugates of the corresponding negative frequency bins. The same symmetry is evident in the example in § 2.1.

```
octave:1> v = [1,0,1,0,1,0,1,0]
v =

   1   0   1   0   1   0   1   0

octave:2> f = fft(v)
f =

   4 + 0i   0 + 0i   0 + 0i   0 + 0i   4 + 0i   0 - 0i   0 - 0i   0 - 0i

octave:3> v = [1,1,0,0,1,1,0,0]
```

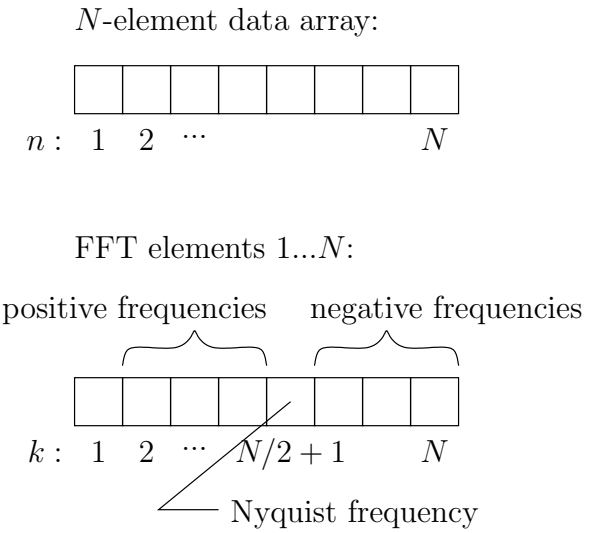


Figure 1: Arrangement of frequencies in the DFT of a time series with an even number of elements. If there are an odd number of elements, the Nyquist frequency is omitted.

```
v =

   1   1   0   0   1   1   0   0

octave:4> f = fft(v)
f =

   4 + 0i   0 + 0i   2 - 2i   0 + 0i   0 + 0i   0 - 0i   2 + 2i   0 - 0i

octave:5> v2 = ifft(f)
v2 =

   1   1   0   0   1   1   0   0
```

## 3.4 Aliasing

A given sampling rate has limited bandwidth — a definite range of frequencies that can be represented by the sampled data: $|\nu| < \nu_c$. Unfortunately, this does not mean that a signal with a frequency outside this range will remain undetected. In § 3.3, we saw that there is a correspondence between negative

frequencies and frequencies in the range $\nu_c < \nu < 2\nu_c$. Figure 2 shows explicitly how a frequency above the Nyquist cutoff will be *aliased* back into the sampling passband.

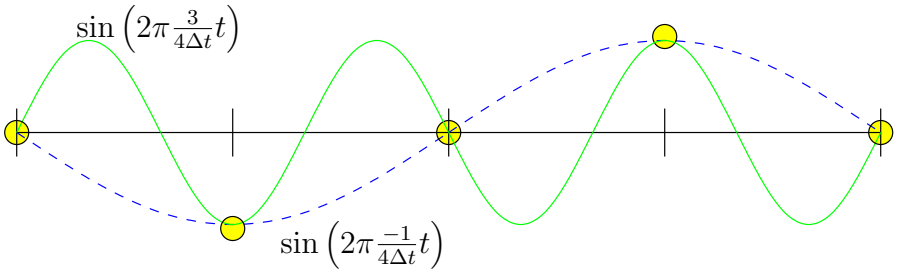


Figure 2: Two signals sampled at interval $\Delta t$. The sampled data, which are identical, are circled by yellow-shaded circles. A frequency of $\nu = \frac{3}{4\Delta t}$ has the same observational signature as $\nu' = \frac{-1}{4\Delta t}$.

It turns out that every frequency outside the passband gets aliased back into the passband. The program given in Appendix B.1 is a numerical experiment used to explore this mapping (figure 3). If you really understand the example, then you will be able to predict how it would look if the complex exponential in `aliastest.m` were replaced by a sine or a cosine.

## 3.5 Shannon's Sampling Theorem

Any periodic function $f(t)$ (or equivalently, a function defined only on the interval $0 < t < T$) can be represented as a Fourier series,

$$f(t) = \sum_{k=1}^{\infty} a_k e^{2\pi i(k-1)t/T}.$$

Let us suppose that all we know of $f(t)$ is its value at $N$ evenly spaced points:

$$f(t_n) = f_n, \quad t_n = \frac{n-1}{N}T, \quad 1 \le n \le N.$$

Let us further suppose that

$$a_k = 0 \quad \text{for} \quad k > N.$$

In other words, there are no frequencies in $f$ above the Nyquist frequency. This is the definition of a *band-limited* signal. Under this condition, there are

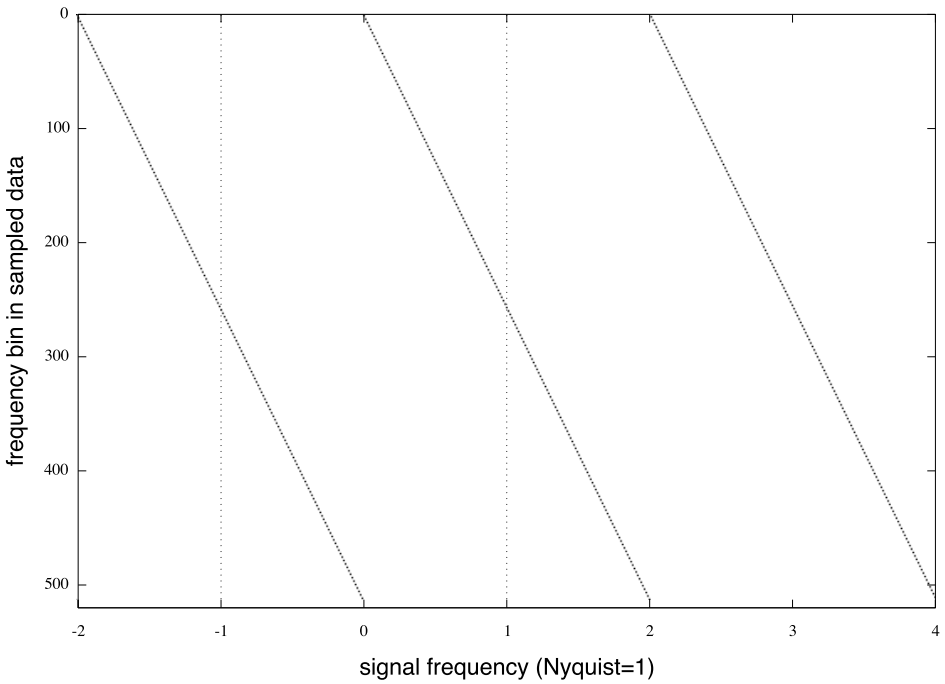


Figure 3: Aliasing of any frequency into the range $|\nu| < \nu_c$, as marked by the dotted lines. The input signal is $e^{2\pi i\nu t}$. The source code for this example is in Appendix B.

only as many nonzero Fourier coefficients as there are data points. Indeed, we note by comparison with equation 2 that $a_k = f_k/N$. It follows that we can use the DFT (equation 1) to construct the exact Fourier series for $f(t)$. This demonstrates the validity of Shannon's theorem:

> If a function $f(t)$ contains no frequencies higher than $\nu$ cycles per second, it is completely determined by giving its values $f_n$ at a series of points spaced $\Delta t = 1/(2\nu)$ seconds apart.

The application of Shannon's sampling theorem to interpolation is investigated in § 5.

# 4 Fast Fourier Transform (FFT)

The DFT as defined in equation 1 uses a sum of $N$ terms to find $f_k$ for each value of $k$. The whole transform ($1 \leq k \leq N$) therefore requires $O(N^2)$ terms, each involving the evaluation of either a complex exponential or trig functions. This is computationally daunting for large data sets.

## 4.1 The "Fast" Part

Fortunately, there exists a fast Fourier transform (FFT) algorithm that can be completed in $O(N \log N)$ operations. Since the FFT is widely available in every programming language, there is little point in dwelling on the details (see *Numerical Recipes*, Ch. 12). The key points are

1. Factoring of $N$ to split $f_n$ into smaller sub-arrays.
2. Combining the DFTs of the sub-arrays to form the DFT of the whole array.
3. Clever use of multiple-angle formulae to minimize calls to trig functions.

This all works most efficiently if $N$ is a power of 2.

## 4.2 FFT implementations

The only important qualities in an FFT implementation are speed, speed, and speed. Well, OK, speed, versatility, and minimizing *Roundoff error*. Appendix A briefly considers roundoff error because it can afflict practically any sort of numerical computation.

There are many implementations of the FFT. Here are several that you might encounter:

- **Numerical Recipes** Ch. 12 contains a simple FFT and several variations. The *NR* FFT is mainly a pedagogical tool, uses single precision arithmetic, and is not especially versatile ($N$ must be a power of 2). Nevertheless, it would presumably work well enough for many applications.

- **FFTW** (the "Fastest Fourier Transform in the West," see `http://www.fftw.org`) was developed, somewhat ironically, at MIT.[1] FFTW is a highly optimized, open source package that is widely used.

- **GNU Octave** uses FFTW. Relevant functions include `fft`, `ifft`, and `fftw`. Detrending and windowing are provided through functions such as `detrend`, `hanning`, and `hamming`.

- **MATLAB** has its own proprietary FFT code that is well-optimized.

Many FFT implementations such as FFTW and *Numerical Recipes* leave out the $1/N$ in the inverse, so that it is not quite the inverse. Although the FFT requires $O(N \log N)$ operations, multiplication is more computationally expensive than addition; therefore elimintating $N$ multiplication operations results in non-negligible time savings for some high performance tasks. Since many FFT applications involve multiplying the transform by a filter function, it is most efficient to include the normalization as part of the filter function. Although Octave uses FFTW, they have included the $1/N$ factor (presumably for compatibility with MATLAB).

# 5 Application: Fourier Interpolation

Shannon's theorem implies that a band-limited signal $f(t)$ for which we have $N_1$ samples, $f_n$, can be reconstructed exactly. All we need to do is supply the higher frequency Fourier coefficients associated with measuring $f(t)$ at some higher resolution, $N_2 > N_1$. Since the signal is band-limited, this is a trivial exercise: all the new Fourier coefficients will be zero! The algorithm is simply:

1. Transform the time series $f_n$ to obtain $\tilde{f}_k$.

2. Form an expanded transform, $\tilde{f}'_k$, by snipping the array $\tilde{f}_k$ at the Nyquist frequency, and inserting $N_2 - N_1$ zeros in the middle.

3. Inverse transform $\tilde{f}'_k$ to form the interpolated time series $f'_n$, which will now have $N_2$ elements.

[1] The FFTW website points out the MIT is west of Italy, the home of the spaghetti western.

As a first example, look at figure 4. Recall that the DFT treats the time series as periodic. The large change from $f_N$ to $f_1$ is impossible to accommodate in a band-limited signal without ringing. What we must do is *detrend* the data. The simplest way is to remove a linear trend with the right slope so that the end data ponts are brought to the same level. After calculating the interpolants, we will then add the linear trend that was removed. For this purpose, the `detrend` function in Octave/MATLAB is unfortunately useless. I have instead built a detrend algorithm into `fstretch`, my general purpose FFT interpolation routine (§ B.2). A better result using `fstretch` with detrending is shown in figure 5. The example script `fstretchExample.m` is in Appendix B.3.

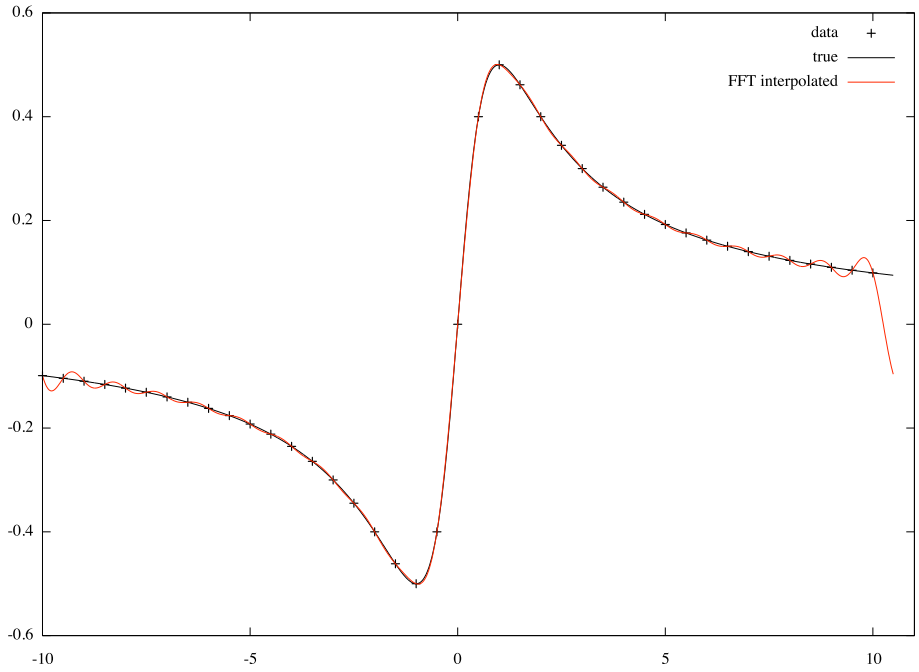


Figure 4: Fourier interpolation of data. The discontinuity between $f_N$ and $f_1$ causes ringing.

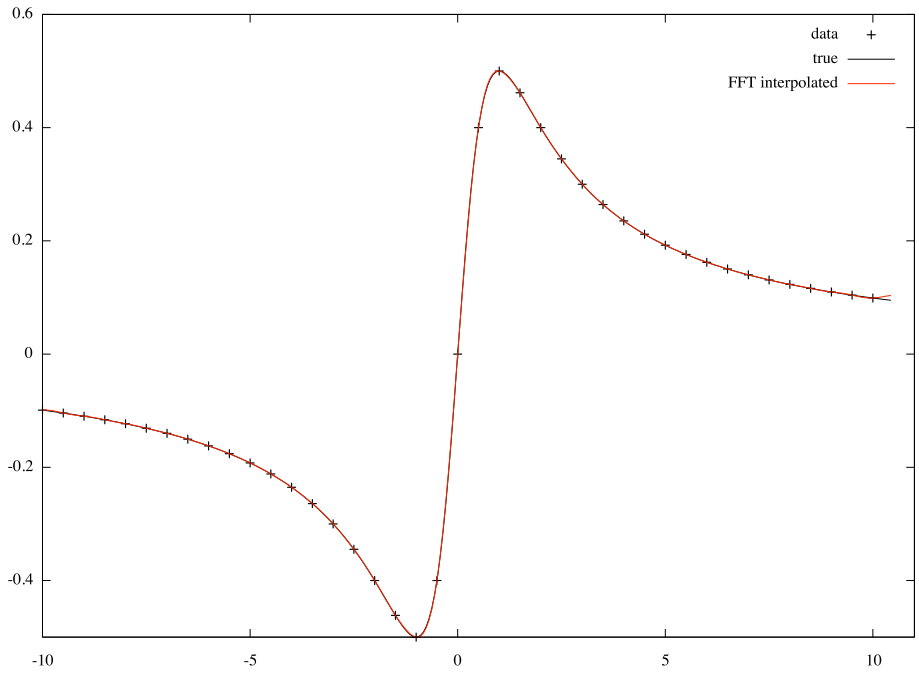


Figure 5: Fourier interpolation of detrended data.

# A Roundoff Error

Roundoff error results from approximating real numbers with a finite number of digits. As operations are performed, the roundoff error gradually accumulates. Using our discrete Fourier transform as an example, suppose that we have $N = 10^8$ samples, the signal is merely a constant, $f_n = 1$, and the internal representation of the machine is good to about 7 significant figures (this is roughly true of single precision floating point numbers; numerical codes nearly always employ double precision—except *Numerical Recipes*!). When we evaluate the first (zero frequency, $k = 1$) Fourier coefficient, it is simply

$$\tilde{f}_1 = \sum_{n=1}^{N} 1.$$

After the first 10 million terms, we will be adding 1 to approximately 10 million. Because of the limited precision however, $10^7 + 1$ rounds to $10^7$, and all the remaining terms are similarly ignored. We end up adding $10^8$ ones

to obtain only $10^7$! A more realistic example would show a modest loss of precision even for a smaller array at double precision. Fortunately, the FFT tends to minimize roundoff error because it subdivides the data into small segments and combines them hierarchically.

# B Source Code

## B.1 aliastest.m

```
N = 512; % Number of times, and of frequencies
minf = -2; % Minimum frequency (Nyquist=1)
maxf = 4; % Maximum frequency (Nyquist=1)
df = (maxf-minf)/N; % Frequency interval
window = hanning(N)*ones(1,N); % Apodization

freq = (0:N-1).*df + minf; % Frequency row vector (Nyquist=1)
t = (0:N-1)'; % Time column vector

freq_mat = ones(N,1) * freq;
   % N x N  array of frequencies, varying along the horizontal.

t_mat = t * ones(1,N);
   % N x N  array of times, varying along the vertical.

signals = exp(i * pi .* freq_mat .* t_mat);
   % N x N  array of signals, each column being a sinusoid
   % at a particular frequency given by freq_mat.

signals_f = fft( window .* signals ); % FFT of each column
signals_power = abs(signals_f).^2 ; % Power spectrum of each column

% Display the power spectra as an image.
hold off
imagesc(freq, (1:N), -signals_power); % minus sign inverts the image.
colormap(gray); % replaces the ugly default color table.
xlabel('signal frequency (Nyquist=1)','fontsize',20);
ylabel('frequency bin in sampled data','fontsize',20);

% Mark out the range of frequencies |f| < Nyquist
hold on
plot(ones(1,100), (1:(N-1)/99:N), "k.")
plot(-1.*ones(1,100), (1:(N-1)/99:N), "k.")
```

## B.2 fstretch.m

```
% fstretch --- FFT-based signal interpolation
function [x2,y2] = fstretch(x1,y1,N2)
% Original data are (x1, y1). x1 is assumed to be uniformly spaced.
%     (but note that only x1(1) and x1(n) are used).
% N2 is the size of the interpolated grid.
% y2 is the new (interpolated) y values.
% x2 is corresponding the new set of x values.

N1 = numel(y1);
if (N2 <= N1)  % We can make the signal bigger, not smaller.
   error('Require N2 > N1.')
endif

% Construct the new x-axis, x2
dx1 = (x1(N1) - x1(1))/(N1-1);
period = N1*dx1; % Note that this is larger than x1(N1)-x1(1).
dx2 = period/N2; % New sampling interval.
x2 = x1(1) + (0:N2-1)*dx2; % New x-axis

% Detrend the data
slope = (y1(N1) - y1(1))/((N1-1)*dx1);
trend1 = slope * dx1 * (0:N1-1);
y1d = y1 - trend1;

y1f = fft(y1d); % FFT of y1, which will supply parts for FFT of y2.

% Construct the FFT of the interpolated signal, y2
y2f = zeros(1,N2);  % initialize blank FFT for y2
y2f(1:floor(N1/2+1)) = y1f(1:floor(N1/2+1));
   % Frequencies 0 through Nyquist go to the first part of y2f.
y2f(N2-floor(N1/2-1):N2) = y1f(ceil(N1/2+1):N1);
   % Frequencies from Nyquist to N go to the last part of y2f.
% In the case where N1 is odd, there is no element at Nyquist
% frequency; this case is handled above by floor() and ceil().

y2 = ifft((N2/N1)*y2f); % create y2 by the inverse transform.

% Reconstruct the original trend, rebinned for the newly interpolated
% data, and add it back in.
trend2 = slope * dx2 * (0:N2-1);
y2 = y2 + trend2; % Add the trend back in
```

## B.3 fstretchExample.m

```
% A script to demonstrate Fourier interpolation with fstretch()
% Create coarsely sampled dataset
x1 = -10:0.5:10; % x data
y1 = x1./(1+x1.^2); % y data, using function y = x/(1+x^2).

N1 = numel(x1);
N2 = 7*N1+19; % Note that N2/N1 doesn't have to be an integer.

[x2,y2] = fstretch(x1,y1,N2); % FFT interpolation
y2a = x2./(1+x2.^2); % True analytic function, for comparison

plot(x1,y1,'+k', x2,y2a,'k', x2,y2,'r');
legend('data','true','FFT interpolated');
```