Fast Fourier Transform (FFT)

On this page…

Introduction

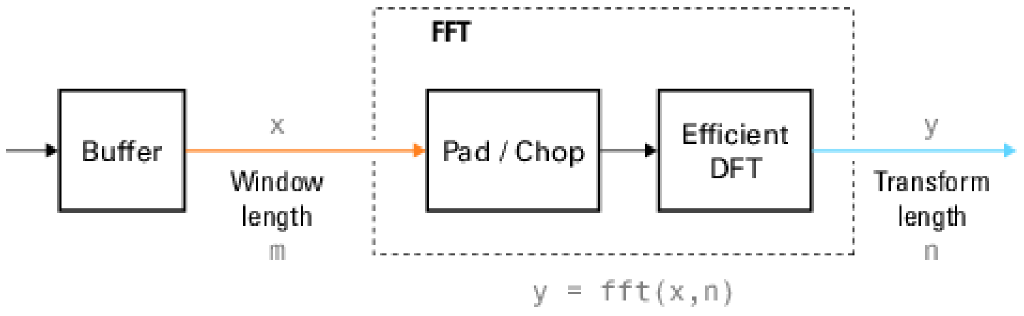
DFTs with a million points are common in many applications. Modern signal and image processing applications would be impossible without an efficient method for computing the DFT.

Direct application of the definition of the DFT (see **Discrete Fourier Transform (DFT)**) to a data vector of length n requires n multiplications and n additions—a total of $2n^2$ floating-point operations. This does not include the generation of the powers of the complex nth root of unity ω. To compute a million-point DFT, a computer capable of doing one multiplication and addition every microsecond requires a million seconds, or about 11.5 days.

Fast Fourier Transform (FFT) algorithms have computational complexity $O(n \log n)$ instead of $O(n^2)$. If n is a power of 2, a one-dimensional FFT of length n requires less than $3n \log_2 n$ floating-point operations (times a proportionality constant). For n = 220, that is a factor of almost 35,000 faster than $2n^2$.

The MATLAB® functions **`fft`**, **`fft2`**, and **`fftn`** (and their inverses **`ifft`**, **`ifft2`**, and **`ifftn`**, respectively) all use fast Fourier transform algorithms to compute the DFT.

When using FFT algorithms, a distinction is made between the *window length* and the *transform length*. The window length is the length of the input data vector. It is determined by, for example, the size of an external buffer. The transform length is the length of the output, the computed DFT. An FFT algorithm pads or chops the input to achieve the desired transform length. The following figure illustrates the two lengths.



The execution time of an FFT algorithm depends on the transform length. It is fastest when the transform length is a power of two, and almost as fast when the transform length has only small prime factors. It is typically slower for transform lengths that are prime or have large prime factors. Time differences, however, are reduced to insignificance by modern FFT algorithms such as those used in MATLAB. Adjusting the transform length for efficiency is usually unnecessary in practice.

The FFT in One Dimension

Introduction

The MATLAB **`fft`** function returns the DFT `y` of an input vector `x` using a fast Fourier transform algorithm:

```
y = fft(x);
```

In this call to `fft`, the window length `m = length(x)` and the transform length `n = length(y)` are the same.

The transform length is specified by an optional second argument:

```
y = fft(x,n);
```

In this call to `fft`, the transform length is `n`. If the length of `x` is less than `n`, `x` is padded with trailing zeros to increase its length to `n` before computing the DFT. If the length of `x` is greater than `n`, only the first `n` elements of `x` are used to compute the transform.

Basic Spectral Analysis

The FFT allows you to efficiently estimate component frequencies in data from a discrete set of values sampled at a fixed rate. Relevant quantities in a spectral analysis are listed in the following table. For space-based data, replace references to time with references to space.

Quantity	**Description**
`x`	Sampled data
`m = length(x)`	Window length (number of samples)
`fs`	Samples/unit time
`dt = 1/fs`	Time increment per sample
`t = (0:m-1)/fs`	Time range for data
`y = fft(x,n)`	Discrete Fourier transform (DFT)
`abs(y)`	Amplitude of the DFT
`(abs(y).^2)/n`	Power of the DFT
`fs/n`	Frequency increment
`f = (0:n-1)*(fs/n)`	Frequency range
`fs/2`	Nyquist frequency

Consider the following data `x` with two component frequencies of differing amplitude and phase buried in noise.

```
fs = 100;                                % Sample frequency
```

```
(Hz)
t = 0:1/fs:10-1/fs;                      % 10 sec sample
x = (1.3)*sin(2*pi*15*t) ...             % 15 Hz component
  + (1.7)*sin(2*pi*40*(t-2)) ...         % 40 Hz component
  + 2.5*gallery('normaldata',size(t),4); % Gaussian noise;
```

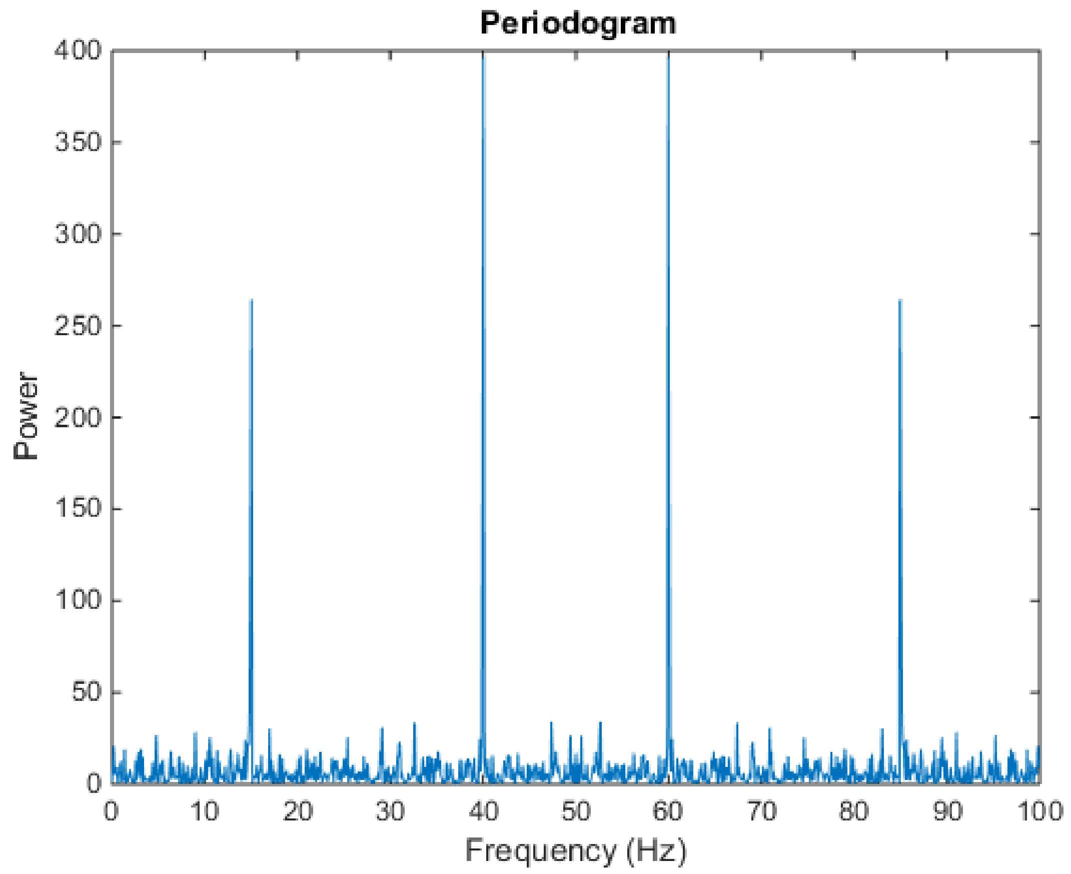
Use `fft` to compute the DFT `y` and its power.

```
m = length(x);          % Window length
n = pow2(nextpow2(m));  % Transform length
y = fft(x,n);           % DFT
f = (0:n-1)*(fs/n);     % Frequency range
power = y.*conj(y)/n;   % Power of the DFT
```

`nextpow2` finds the exponent of the next power of two greater than or equal to the window length (`ceil(log2(m))`) and `pow2` computes the power. Using a power of two for the transform length optimizes the FFT algorithm, though in practice there is usually little difference in execution time from using `n = m`.

To visualize the DFT, plots of `abs(y)`, `abs(y).^2`, and `log(abs(y))` are all common. A plot of power versus frequency is called a *periodogram*.

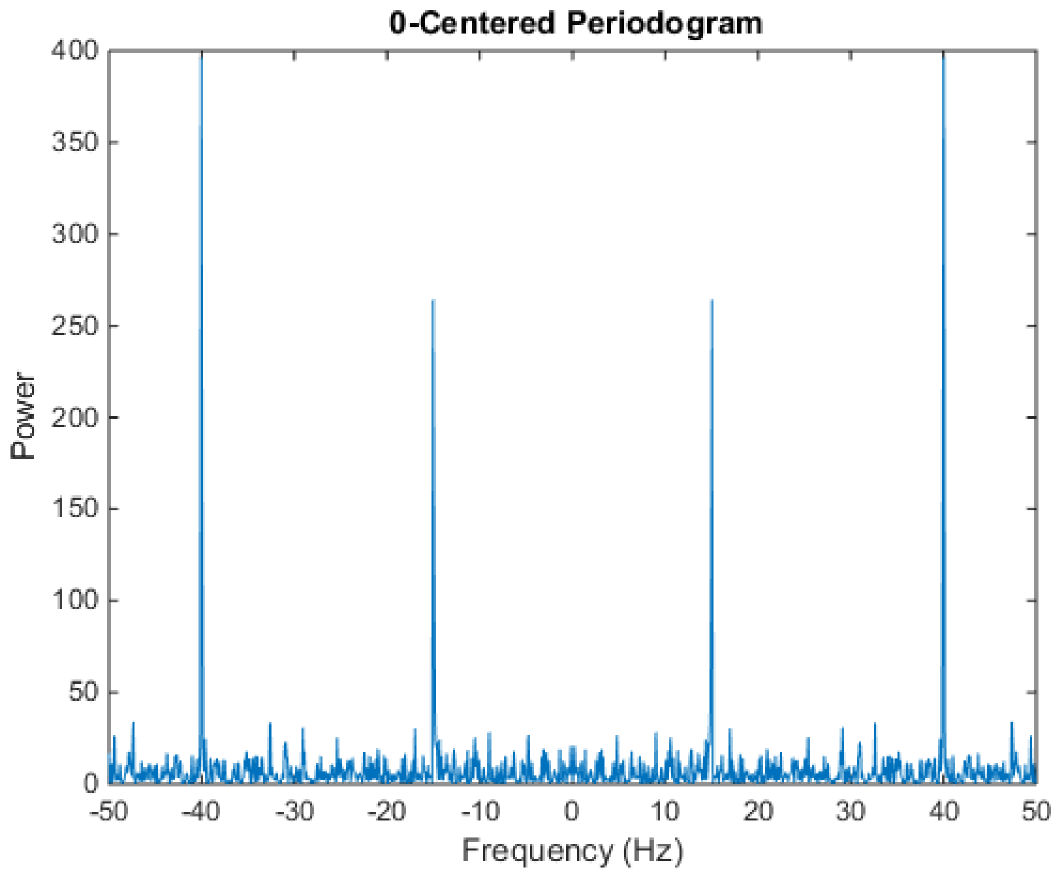
```
plot(f,power)
xlabel('Frequency (Hz)')
ylabel('Power')
title('{\bf Periodogram}')
```



The first half of the frequency range (from 0 to the Nyquist frequency `fs/2`) is sufficient to identify the component frequencies in the data, since the second half is just a reflection of the first half.

In many applications it is traditional to center the periodogram at 0. The `fftshift` function rearranges the output from `fft` with a circular shift to produce a 0-centered periodogram.

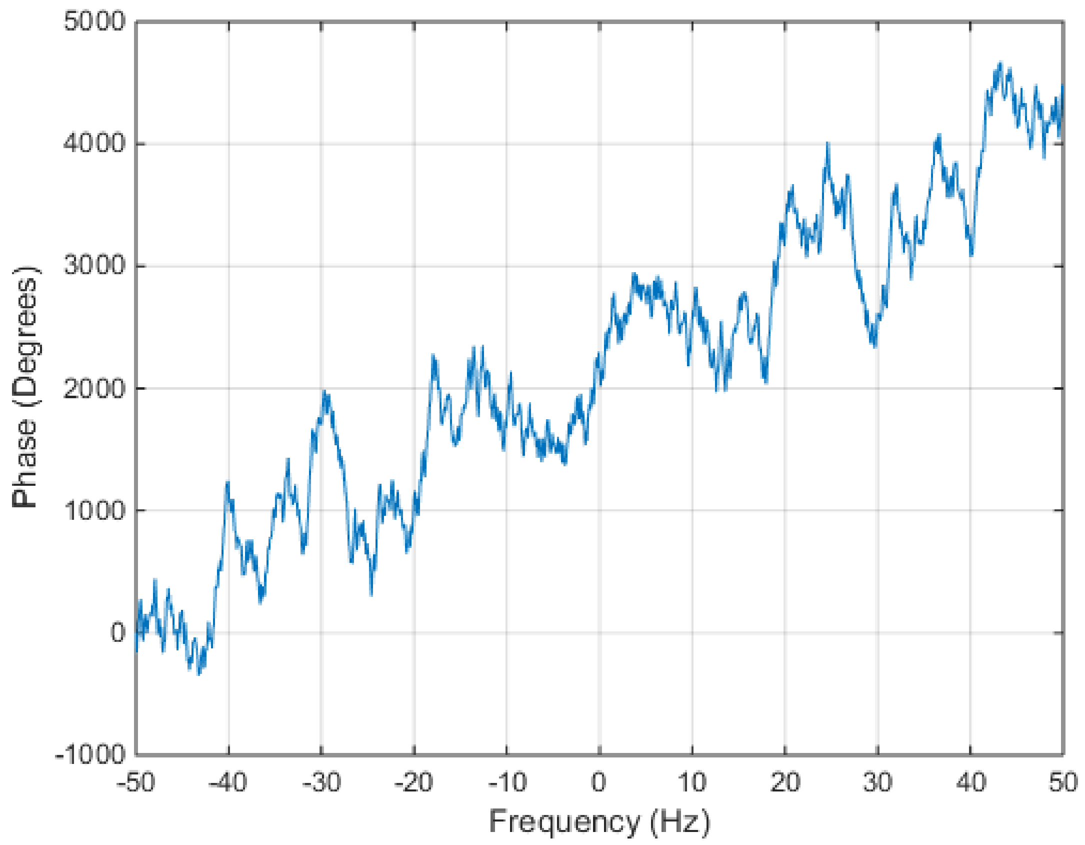
```
y0 = fftshift(y);          % Rearrange y values
f0 = (-n/2:n/2-1)*(fs/n);  % 0-centered frequency range
power0 = y0.*conj(y0)/n;   % 0-centered power

plot(f0,power0)
xlabel('Frequency (Hz)')
ylabel('Power')
title('{\bf 0-Centered Periodogram}')
```



The rearrangement makes use of the periodicity in the definition of the DFT.

Use the MATLAB `angle` and `unwrap` functions to create a phase plot of the DFT.

```
phase = unwrap(angle(y0));

plot(f0,phase*180/pi)
xlabel('Frequency (Hz)')
ylabel('Phase (Degrees)')
grid on
```



Component frequencies are mostly hidden by the randomness in phase at adjacent values. The upward trend in the plot is due to the `unwrap` function, which in this case adds 2π to the phase more often than it subtracts it.

Spectral Analysis of a Whale Call

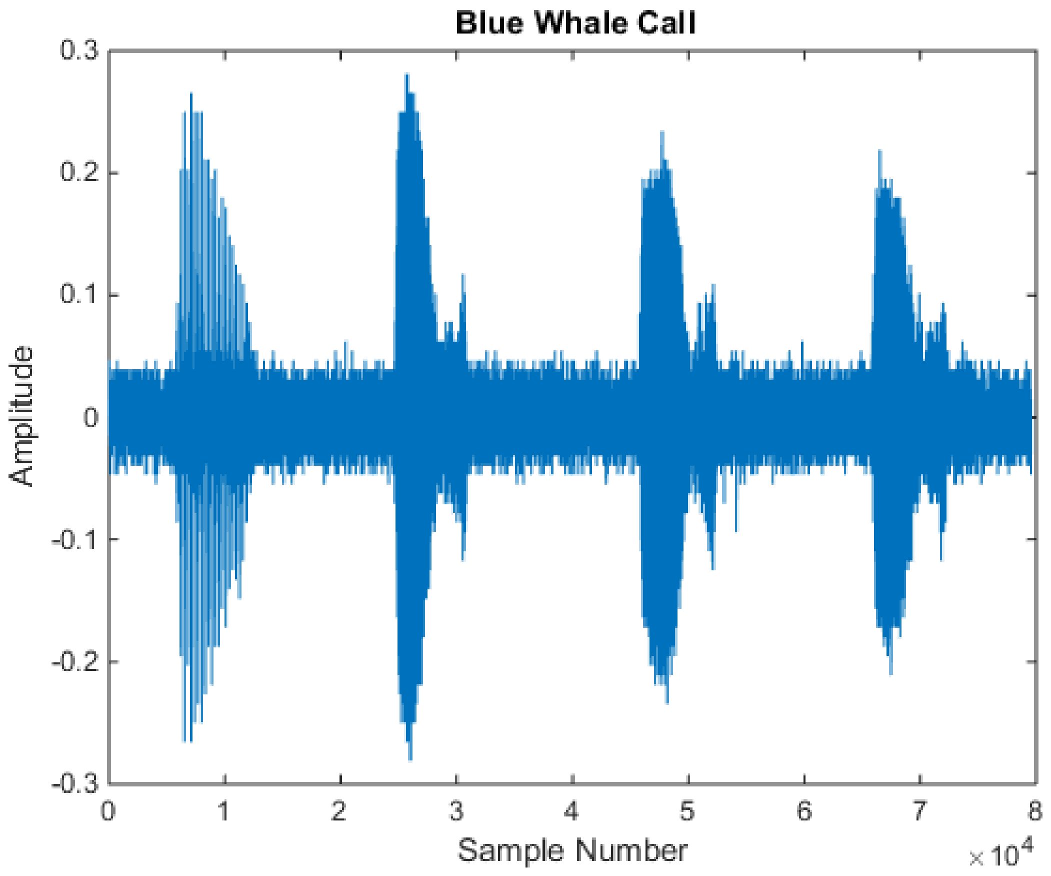
This example shows how to perform spectral analysis of audio data.

The documentation example file `bluewhale.au` contains audio data from a Pacific blue whale vocalization recorded by underwater microphones off the coast of California. The file is from the library of animal vocalizations maintained by the **Cornell University Bioacoustics Research Program (http://www.birds.cornell.edu/brp/)** .

Because blue whale calls are so low, they are barely audible to humans. The time scale in the data is compressed by a factor of 10 to raise the pitch and make the call more clearly audible. The following reads and plots the data.

```
whaleFile =
fullfile(matlabroot,'examples','matlab','bluewhale.au');
[x,fs] = audioread(whaleFile);

plot(x)

xlabel('Sample Number')
ylabel('Amplitude')
```

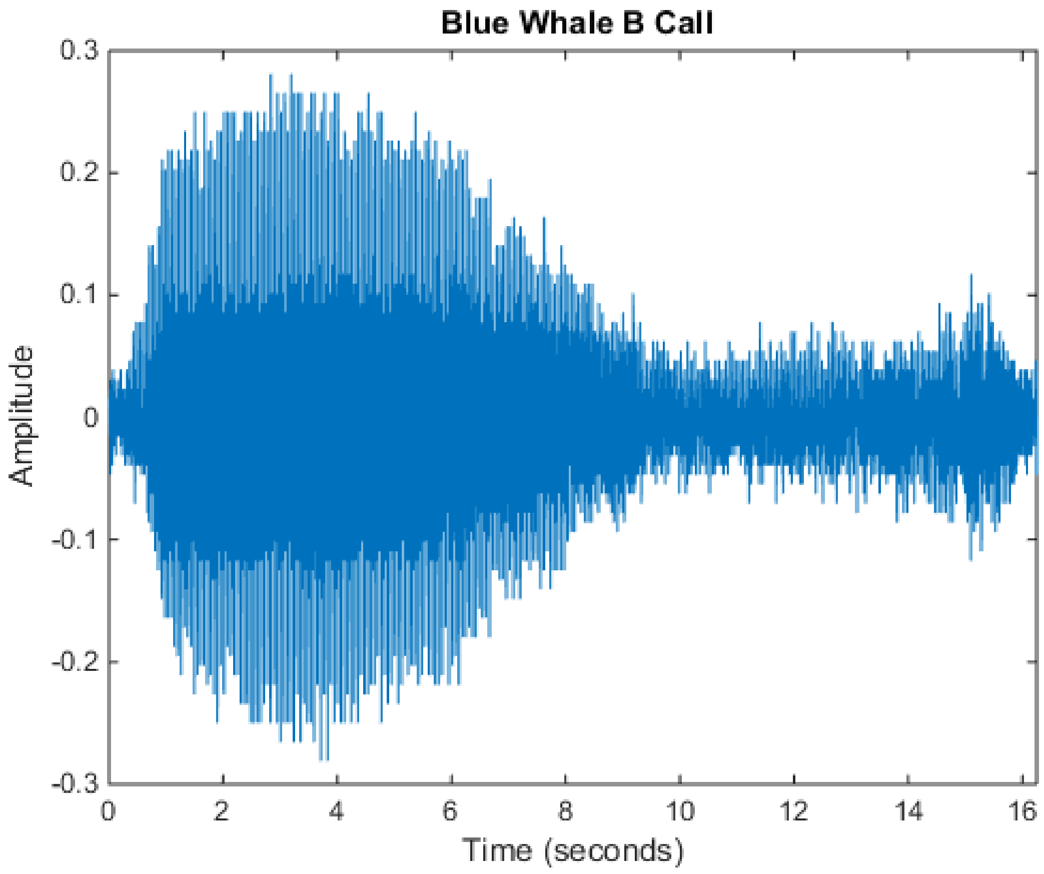
```
title('{\bf Blue Whale Call}')
```



An A "trill" is followed by a series of B "moans." You can use `sound(x,fs)` to play the data.

The B call is simpler and easier to analyze. Use the previous plot to determine approximate indices for the beginning and end of the first B call. Correct the time base for the factor of 10 speed-up in the data.

```
bCall = x(2.45e4:3.10e4);
tb = 10*(0:1/fs:(length(bCall)-1)/fs); % Time base

plot(tb,bCall)

xlim([0 tb(end)])

xlabel('Time (seconds)')
ylabel('Amplitude')

title('{\bf Blue Whale B Call}')
```

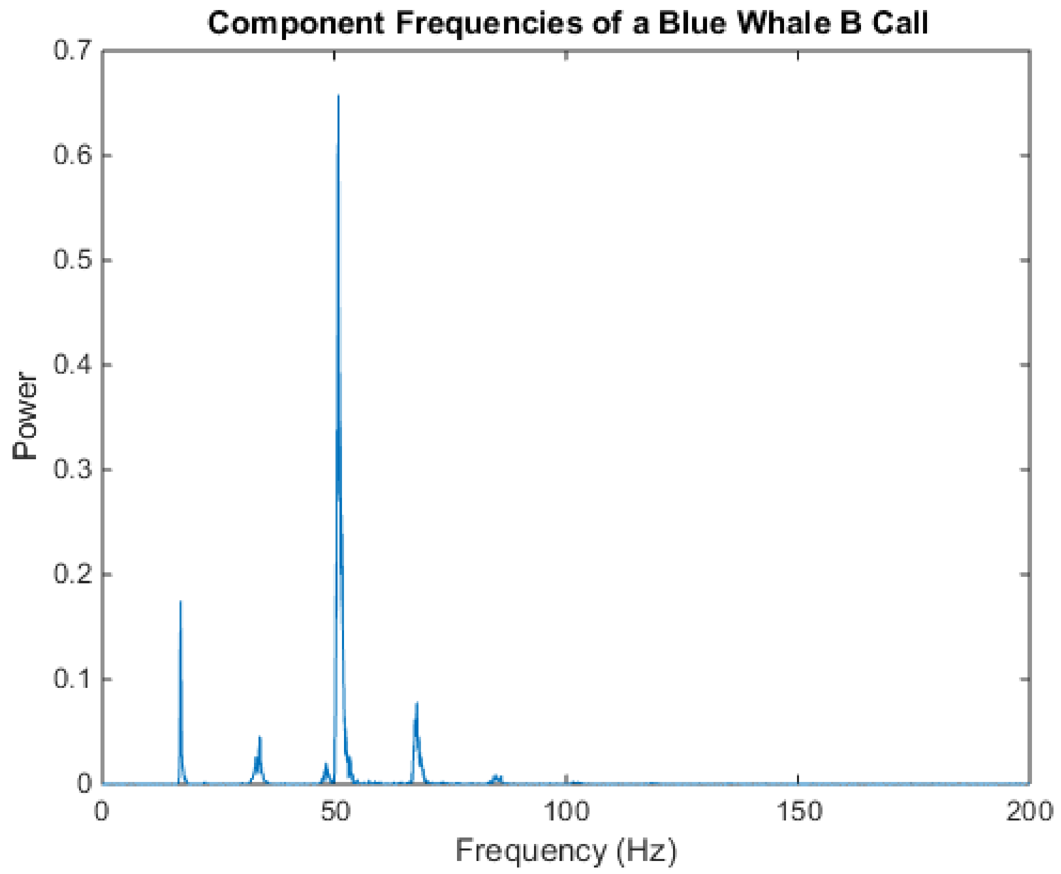


Use `fft` to compute the DFT of the signal. Correct the frequency range for the factor of 10 speed-up in the data.

```
m = length(bCall);       % Window length
n = pow2(nextpow2(m));   % Transform length
y = fft(bCall,n);        % DFT of signal
f = (0:n-1)*(fs/n)/10;   % Frequency range
p = y.*conj(y)/n;        % Power of the DFT
```

Plot the first half of the periodogram, up to the Nyquist frequency.

```
plot(f(1:floor(n/2)),p(1:floor(n/2)))

xlabel('Frequency (Hz)')
ylabel('Power')

h =gca;
h.XTick = [0 50 100 150 200];

title('{\bf Component Frequencies of a Blue Whale B Call}')
```



The B call is composed of a fundamental frequency around 17 Hz and a sequence of harmonics, with the second harmonic emphasized.

Data Interpolation Using FFT

This example shows how to use FFT in a context other than spectral analysis—estimating coefficients of a trigonometric polynomial that interpolates a set of regularly-spaced data. This approach to data interpolation is described in **[1]**.

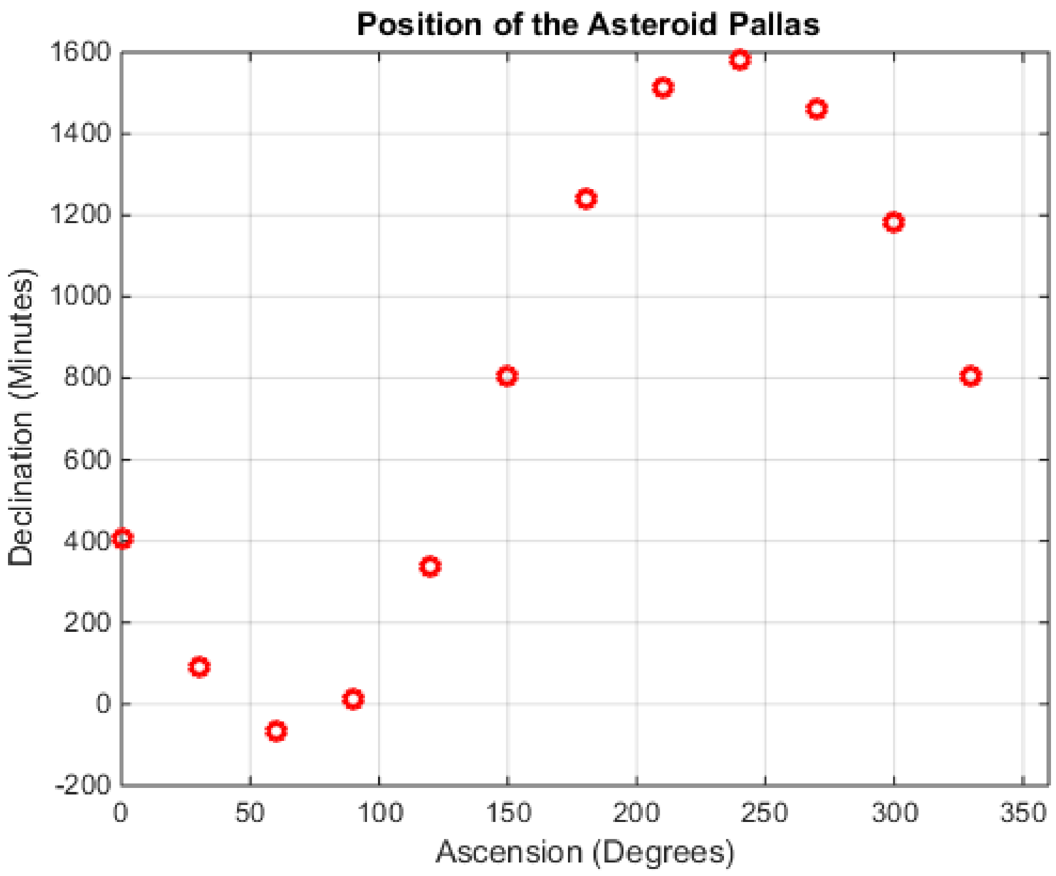
Several people discovered fast DFT algorithms independently, and many people have contributed to their development. A 1965 paper by John Tukey and John Cooley **[2]** is generally credited as the starting point for modern usage of the FFT. However, a paper by Gauss published posthumously in 1866 **[3]** (and dated to 1805) contains indisputable use of the splitting technique that forms the basis of modern FFT algorithms.

Gauss was interested in the problem of computing accurate asteroid orbits from observations of their positions. His paper contains 12 data points on the position of the asteroid Pallas, through which he wished to interpolate a trigonometric polynomial with 12 coefficients. Instead of solving the resulting 12-by-12 system of linear equations by hand, Gauss looked for a shortcut. He discovered how to separate the equations into three subproblems that were much easier to solve, and then how to recombine the solutions to obtain the desired result. The solution is equivalent to estimating the DFT of the data with an FFT algorithm.

Here is the data that appears in Gauss' paper.

```
asc = 0:30:330;
dec = [408 89 -66 10 338 807 1238 1511 1583 1462 1183 804];

plot(asc,dec,'ro','Linewidth',2)
```

```
xlim([0 360])
xlabel('Ascension (Degrees)')
ylabel('Declination (Minutes)')
title('{\bf Position of the Asteroid Pallas}')
grid on
```



Gauss wished to interpolate a trigonometric polynomial of the form

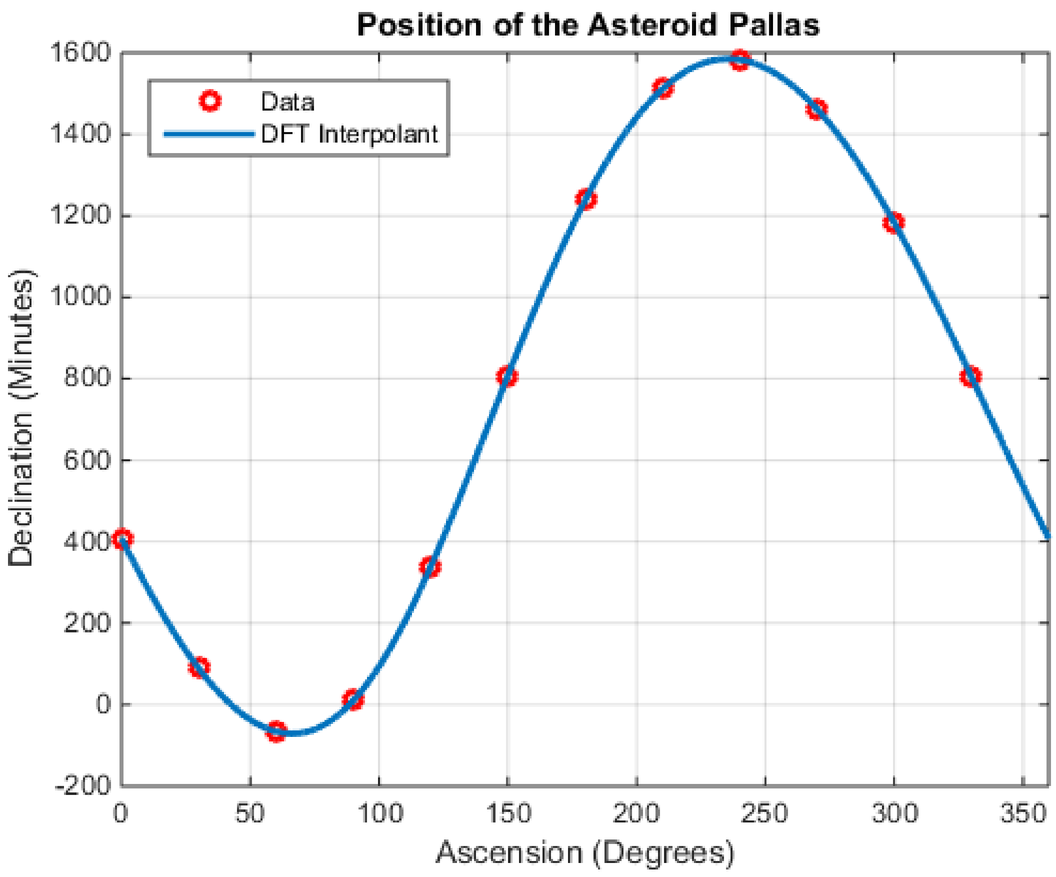
$$\begin{aligned} y = a_0 \quad + \quad & a_1 \cos(2\pi(x/360)) + b_1 \sin(2\pi(x/360)) \\ & a_2 \cos(2\pi(2x/360)) + b_2 \sin(2\pi(2x/360)) \\ & \cdots \\ & a_5 \cos(2\pi(5x/360)) + b_5 \sin(2\pi(5x/360)) \\ & a_6 \cos(2\pi(6x/360)) \end{aligned}$$

Use `fft` to perform an equivalent of Gauss' calculation.

```
d = fft(dec);
m = length(dec);
M = floor((m+1)/2);

a0 = d(1)/m;
an = 2*real(d(2:M))/m;
a6 = d(M+1)/m;
bn = -2*imag(d(2:M))/m;
```

Plot the interpolant with the data.

```
hold on

x = 0:0.01:360;
n = 1:length(an);
y = a0 + an*cos(2*pi*n'*x/360) ...
       + bn*sin(2*pi*n'*x/360) ...
       + a6*cos(2*pi*6*x/360);

plot(x,y,'Linewidth',2)
legend('Data','DFT Interpolant','Location','NW')
```



References.

[1] Briggs, W. and V.E. Henson. *The DFT: An Owner's Manual for the Discrete Fourier Transform*. Philadelphia: SIAM, 1995.

[2] Cooley, J.W. and J.W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation*. Vol. 19. 1965, pp. 297–301.

[3] Gauss, C. F. "Theoria interpolationis methodo nova tractata." *Carl Friedrich Gauss Werke*. Band 3. Göttingen: Königlichen Gesellschaft der Wissenschaften, 1866.

[4] Heideman M., D. Johnson, and C. Burrus. "Gauss and the History of the Fast Fourier Transform." *Arch. Hist. Exact Sciences*. Vol. 34. 1985, pp. 265–277.

[5] Goldstine, H. H. *A History of Numerical Analysis from the 16th through the 19th Century*. Berlin: Springer-Verlag, 1977.

The FFT in Multiple Dimensions

- **Introduction**
- **Diffraction Patterns**

Introduction

This section discusses generalizations of the DFT in one dimension (see **Discrete Fourier Transform (DFT)**).

In two dimensions, the DFT of an m-by-n array X is another m-by-n array Y:

$$Y_{p+1,q+1} = \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} \omega_m^{jp} \omega_n^{kq} X_{j+1,k+1}$$

where ω_m and ω_n are complex roots of unity:

$$\omega_m = e^{-2\pi i/m}$$

$$\omega_n = e^{-2\pi i/n}$$

This notation uses i for the imaginary unit, p and j for indices that run from 0 to m–1, and q and k for indices that run from 0 to n–1. The indices p+1 and j+1 run from 1 to m and the indices q+1 and k+1 run from 1 to n, corresponding to ranges associated with MATLAB arrays.

The MATLAB function **`fft2`** computes two-dimensional DFTs using a fast Fourier transform algorithm. `Y = fft2(X)` is equivalent to `Y = fft(fft(X).').'`, that is, to computing the one-dimensional DFT of each column `X` followed by the one-dimensional DFT of each row of the result. The inverse transform of the two-dimensional DFT is computed by **`ifft2`**.

The MATLAB function **`fftn`** generalizes `fft2` to N-dimensional arrays. `Y = fftn(X)` is equivalent to:

```
Y = X;
for p = 1:length(size(X))
    Y = fft(Y,[],p);
end
```

That is, to computing in place the one-dimensional DFT along each dimension of `X`. The inverse transform of the N-dimensional DFT is computed by **`ifftn`**.

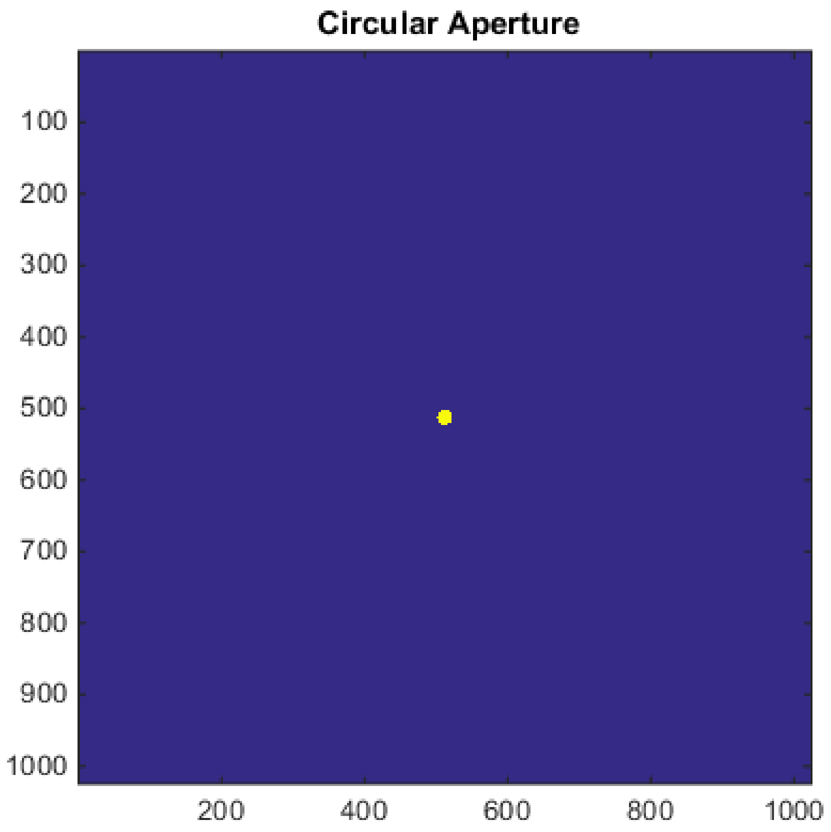
Diffraction Patterns

The theory of optics predicts that the diffraction pattern produced by a plane wave incident on an optical mask with a small aperture is described, at a distance, by the Fourier transform of the mask. See, for example, **[1]**.

Create a logical array describing an optical mask `M` with a circular aperture `A` of radius `R`.

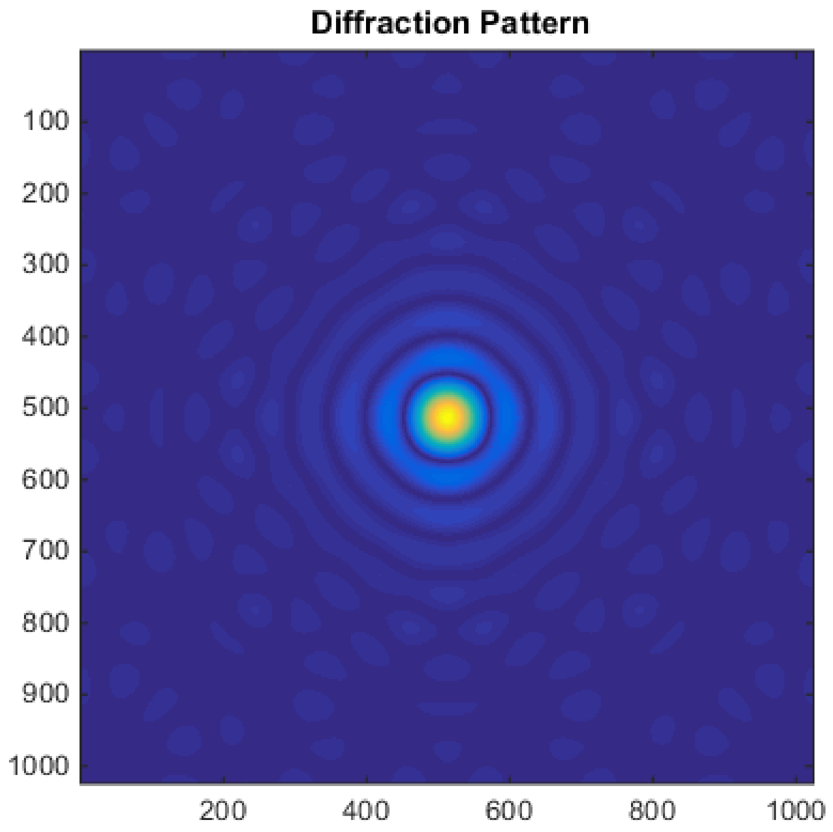
```
n = 2^10;
M = zeros(n);

I = 1:n;
```

```
x = I-n/2;
y = n/2-I;
[X,Y] = meshgrid(x,y);
R = 10;
A = (X.^2 + Y.^2 <= R^2);
M(A) = 1;

imagesc(M)
axis image
title('{\bf Circular Aperture}')
```



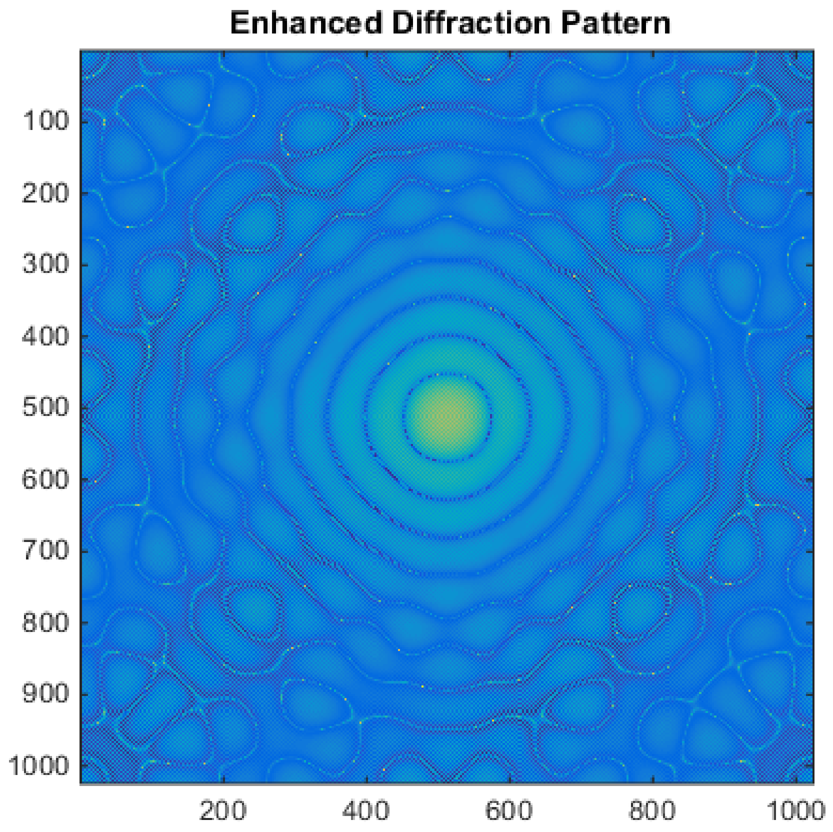
Use `fft2` to compute the two-dimensional DFT of the mask and `fftshift` to rearrange the output so that the zero-frequency component is at the center.

```
D1 = fft2(M);
D2 = fftshift(D1);

imagesc(abs(D2))
axis image
title('{\bf Diffraction Pattern}')
```



The logarithm helps to bring out details of the DFT in regions where the amplitude is small.

```
D3 = log2(D2);

imagesc(abs(D3))
axis image
title('{\bf Enhanced Diffraction Pattern}')
```



Very small amplitudes are affected by numerical round-off. The lack of radial symmetry is an artifact of the rectangular arrangement of data.

Reference.

[1] Fowles, G. R. *Introduction to Modern Optics*. New York: Dover, 1989.

See Also

`angle` | **`fft`** | **`fft2`** | **`fftn`** | **`fftshift`** | **`ifft`** | **`ifft2`** | **`ifftn`** | **`nextpow2`** | **`pow2`** | **`unwrap`**

More About

- **Discrete Fourier Transform (DFT)**