Computational Perception & Scene Analysis
15/85-485/785

A tutorial on filtering, convolution, and the Fourier domain using MATLAB

1 Frequency domain, Fourier transforms

Sounds can broken up into components of varying frequency. Simple tones are sine waves, and it turns out that any sound (in fact, any function) can be written as a sum of pure tones of different frequencies and amplitudes

$$f(t) = a(0) + \sum_{k=1}^{N/2} a(k)\cos(2\pi kt/N + \theta(k)) \tag{1}$$

where N is the number of samples in the sound $f(t)$.

So a sound can be completely described by its *Fourier coefficients* $a(k)$, which measure the volumes of tones of frequency $k/(N\delta)$ in the sound (where δ is the interval between samples), together with information about the relative offsets of these sinusoids, $\theta(k)$ (we'll pretty much ignore these phases from now on). Adding up all these different tones weighted by $a(k)$ and offset by $\theta(k)$ reproduces the original sound $f(t)$.

This means that the function $a(k)$ (together with the phases $\theta(k)$) is an equivalent alternative representation of all the information in the function $f(t)$. The original $f(t)$ was a function of time; the alternative representation $a(k)$ is a function of *frequency*. For some operations — like localizing sounds — it can be convenient to work with representations like $a(k)$; this is known as working in *frequency domain* instead of *time domain*.

Given any function $f(t)$, an operation known as a *Fourier transform* (written $\mathcal{F}(f(t))$) will return its frequency domain representation. In MATLAB, the function `fft(t)` (for "Fast Fourier Transform") returns the frequency domain coefficients a. However, the MATLAB version uses a more general representation of sinusoids than the cosines of equation 1, based on complex exponentials using Euler's famous formula, $e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta)$. We've provided the functions `e2c` and `c2e` to translate between the exponential version and the cosine amplitudes and phases we're using here. You can download these commands from course Blackboard page under Assignments.

We'll be giving examples in MATLAB. If you're unfamiliar with MATLAB, you can get help about a function by typing `help` followed by the function name; or `helpdesk` to bring up a web-based help system.

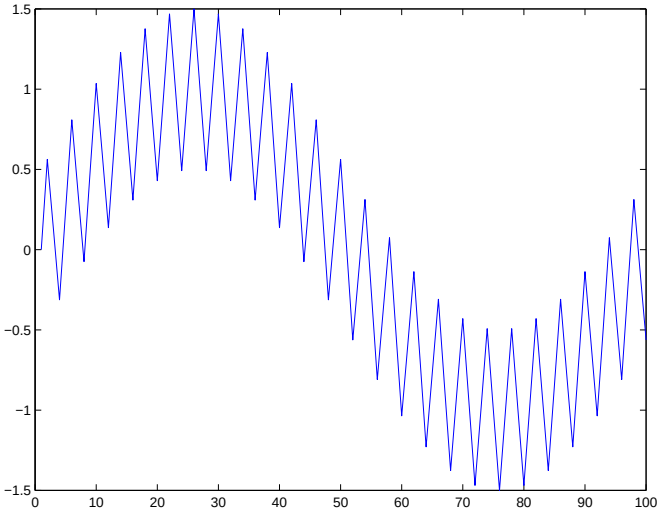
Now, for instance, we can define:

```
>> t = [0:.01:.99];
```

Variables in MATLAB can represent vectors or matrices. Here we're defining `t` as a vector containing the numbers between 0 to 0.99, staggered by .01. If you leave off the semicolon at the end of any statement, MATLAB will print the result. MATLAB can manipulate vectors or matrices in a single step; in the next statement every point in the vector `t` is multiplied by 2π and has the `sin` function applied to it to produce a new vector.

```
>> f = sin(t * 2 * pi) + .5 * sin(25 * t * 2 * pi);
>> plot(f)
```

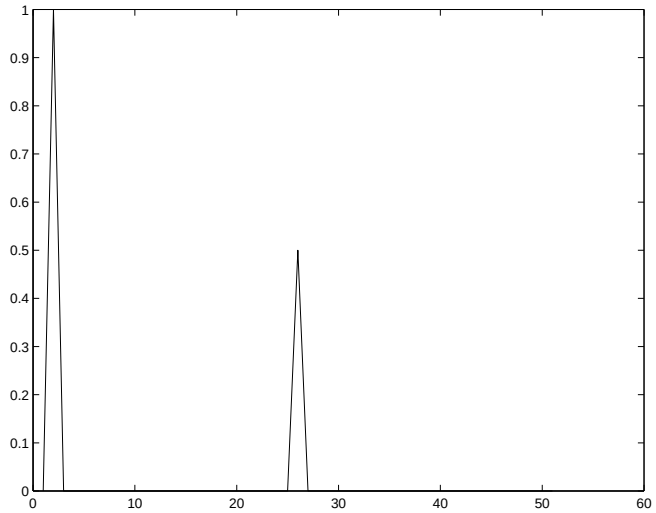
which gives us a small, high-frequency sinusoid riding on top of a low-frequency one:



Plotting the frequency coefficients from the Fourier transform (known as the *spectrum*):

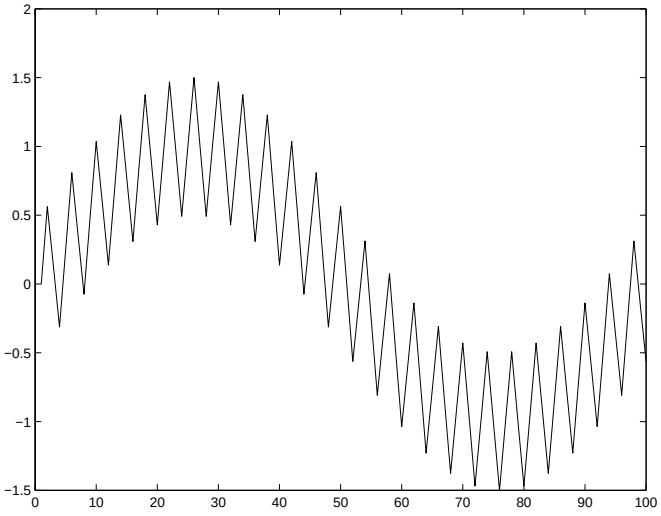
```
>> [a,theta] = e2c(fft(f));
>> plot(a)
```

reveals two "peaks" of frequency, corresponding to the two sinusoids; the second of half amplitude:



Again, all sounds can be broken up into their component frequencies this way, not just simple sinusoids. Once you are done working in frequency domain, you can reconstitute the original function *f(t)* with an *inverse Fourier transform* $\mathcal{F}^{-1}$: that is, just add up all the sinusoids weighted according to equation 1. In MATLAB, the inverse Fourier transform is `ifft`:

```
>> f = ifft(c2e(a, theta));
>> plot(real(f)) % real necessary to avoid a polar plot
```



Finally, we'll show how the `e2c` function is implemented to give an example of how you can implement functions in MATLAB. The function is in a file called `e2c.m`, and it contains the following:

```
function [coeffs,phases] = e2c(F)
```

This defines a function taking the argument `F` and returning the results `coeffs` and `phases`.

```
N = length(F);         % the length of the input vector
n = floor(N / 2) + 1;  % the length of the output vectors
```

Now we set the values of the results:

```
coeffs = 2 * abs(F(1:n)) / N;
phases = angle(F(1:n));
```

We extract information about sinusoid magnitudes with `abs` and phases with `angle`. Note that we ignore the second half of the Fourier transform vector `F`. The exponential sinusoid representation used by MATLAB's `fft` includes information about sinusoids of *negative* frequency there. Assuming the input to the Fourier transform was real-valued, this information is redundant with the positive frequency information in the first half of the vector. The function continues with code to handle some special cases.

2 Filtering

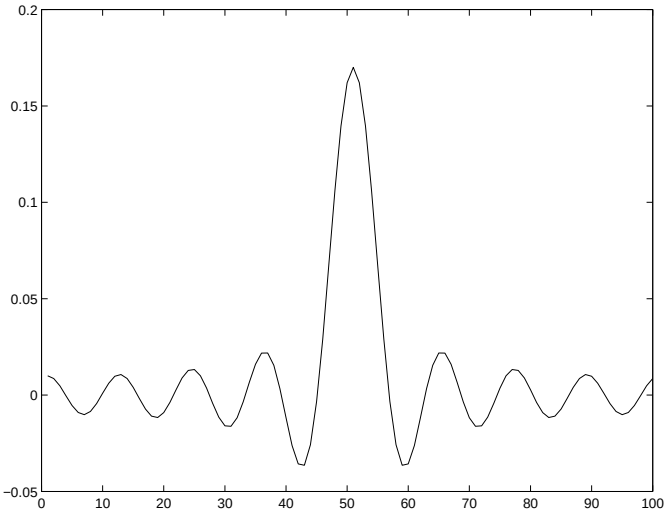
One application for which it is useful to work in the frequency domain is filtering. Suppose you have a sound which is corrupted with high frequency noise. One way to remove this noise would be to take the Fourier transform of the sound, remove all high frequencies by dampening the amplitudes $a(k)$ for all k greater than some cutoff c, and then reconstitute what remains of the sound in time domain by taking the *i*nverse Fourier transform.

This operation is known as applying a *low-pass filter*, since it allows only low frequencies to survive the process. In frequency domain, we can describe a filter as a function indicating how much each frequency is dampened or amplified. In the case of our lowpass filter, we have:

$$H(k) = \begin{array}{ll} 1; & k < c \\ 0; & k > c \end{array}$$

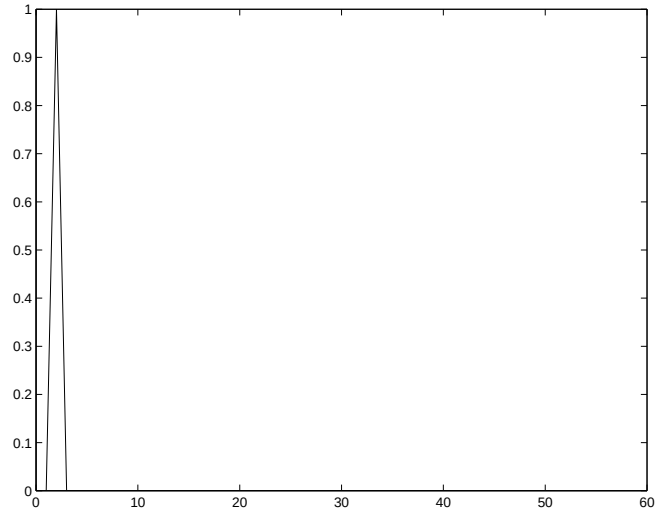
The filter looks like a step function in frequency domain:

```
>> H = [1:51 < 10];
>> plot(H); axis([0 50 -.5 1.5])
```



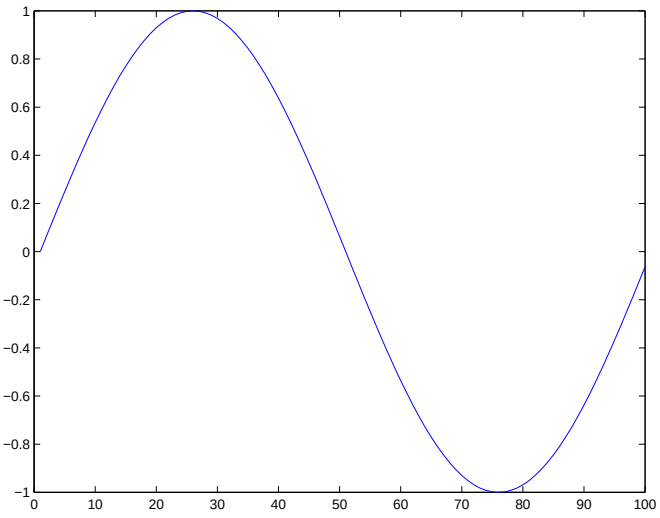
To apply the filter to a sound in frequency domain, we simply multiply each Fourier amplitude $a(k)$ by its corresponding filter amplitude $H(k)$. (This corresponds to the MATLAB operation .* for componentwise vector multiplication, instead of *, which does a dot product.) Applying a filter like this affects the spectrum of the sound by increasing or decreasing the amplitudes of different frequencies while leaving their phases unchanged. (More general filters can affect the phases as well.) Applying it to our double sine wave flattens one of the two peaks in its spectrum:

```
>> a = a .* H;                    % note use of .* instead of *
>> plot(a)
```



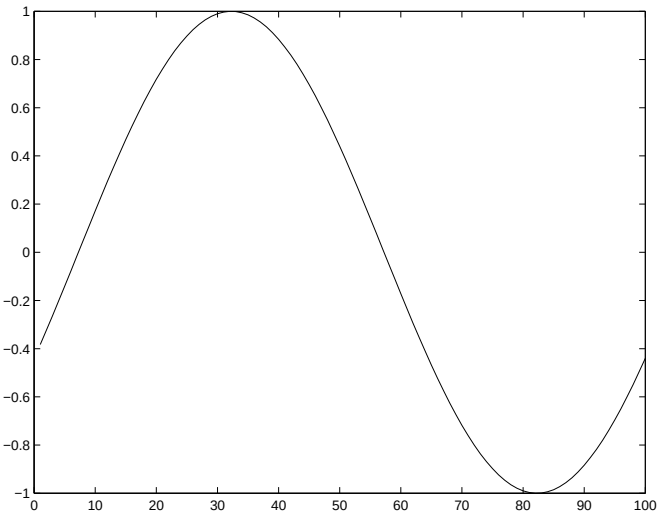
And, viewed in time domain, the filter has removed the high frequency sine wave while leaving the lower one intact:

```
>> filteredf = real(ifft(c2e(a, theta)));
>> plot(filteredf);
```



It's also possible, of course, to modify the phases at various frequencies to change their relative timings. For instance, we can introduce a delay in the low frequency sinusoid by subtracting $\pi/8$ from its phase:

```
>> theta(1) = theta(1) - pi / 8;
>> filteredf = real(ifft(c2e(a, theta)));
>> plot(filteredf);
```



Now the sinusoid is slightly set back — a little slower to peak — than it was before. One advantage of the exponential sinusoid representation we're not using here is that you can modify the phases this way just by multiplying, so filters can affect both sinusoid amplitudes and phases simultaneously.

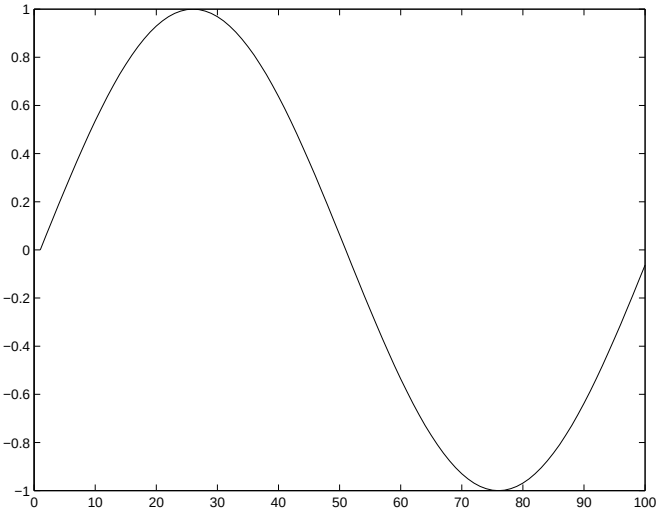
3 Convolution

In order to understand how the filter works in the time domain, we need to use the concept of *convolution*. The convolution of two vectors $\mathbf{x}$ and $\mathbf{y}$, written $\mathbf{x} \star \mathbf{y}$ (or `conv(x,y)` in MATLAB), is a vector $\mathbf{z}$ defined as:

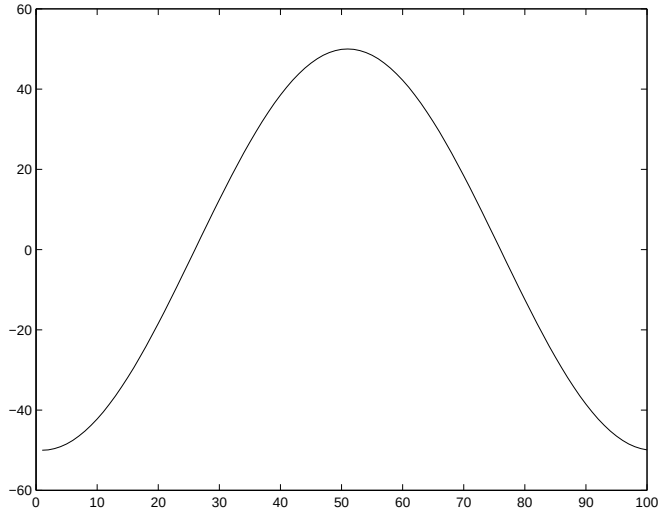
$$z(i) = \sum_{j=1}^{i} x(i-j)y(j)$$

where $\mathbf{x}$ and $\mathbf{y}$ are zero-padded as necessary. This operation boils down to reversing $\mathbf{x}$, then "sliding" it along $\mathbf{y}$, taking the dot product of the overlapping parts at all possible offsets. Since the dot product of two vectors is a measure of similarity, the convolution in a sense measures how much, at different offsets, the vector $\mathbf{x}$ (reversed) looks like the vector $\mathbf{y}$. In particular, convolving two sinusoids of the same frequency together will produce another sinusoid of the same frequency, since the sinusoids have a high dot product when they are in phase and a low one when they are out of phase.

```
>> f1 = sin(t * 2 * pi);
>> plot(f1);
```

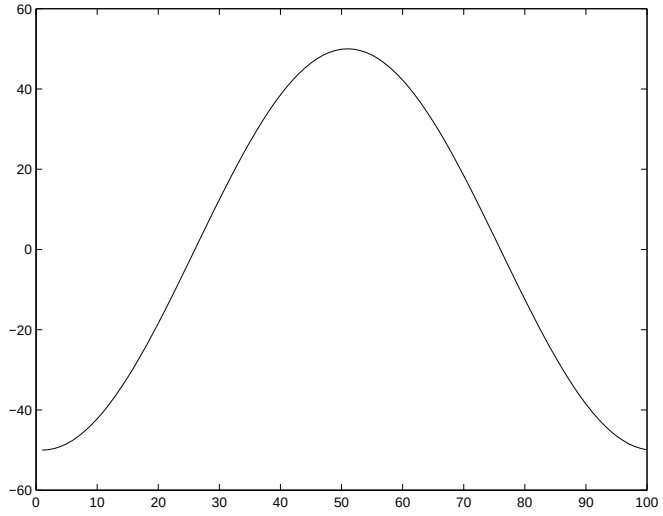



```
>> f2 = conv(f1,[f1,f1]); % double size to avoid boundary effects
>>                        % from zero-padding
>> plot(f2(100:199));     % view the part which was not zero-padded.
```



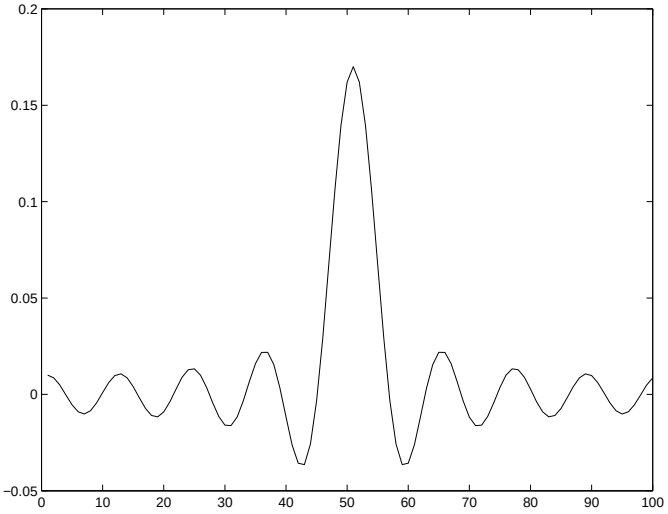
It turns out that applying a filter $H(k)$ in frequency domain corresponds to *convolving* by some corresponding vector $h(t)$ in time domain, with $h(t) = \mathcal{F}^{-1}(H(k))$. That is, *convolution* in time domain is equivalent to *multiplication* in frequency domain: $h(t) \star f(t) = H(k) \cdot a(k)$. (This relationship, the *convolution theorem,* strictly only holds true if the vectors are continuous and infinite, or if you play some games with boundary conditions and zero padding of the vectors, which we do throughout our examples here.) Here, for example, we obtain the same result we got from convolving `f1` with itself above, but by using multiplication in frequency domain:

```
>> plot(real(ifft(fft(f1) .* fft(f1))))
```

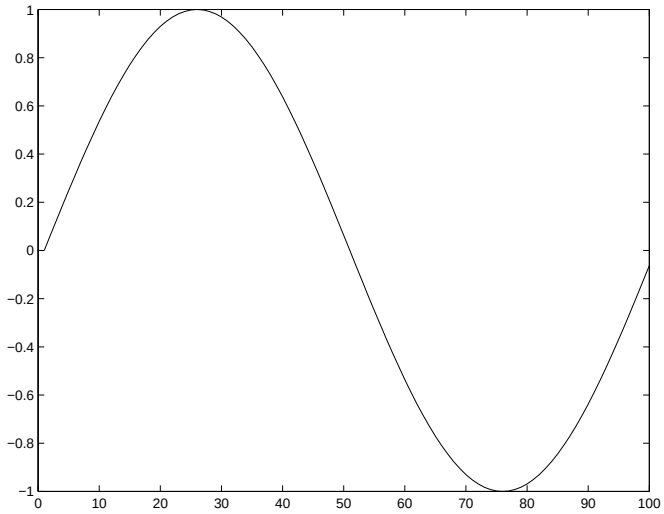


And here, we get the same result as applying our lowpass filter in frequency domain, by convolving by an inverse Fourier transformed version:

```
>> h = real(ifft(c2e(H) / 50)); % / 50 to fix scaling of cosine representation
>> h = [h,h];                   % double size to avoid boundary issues
>> plot(h(51:150))              % view middle part of filter
```



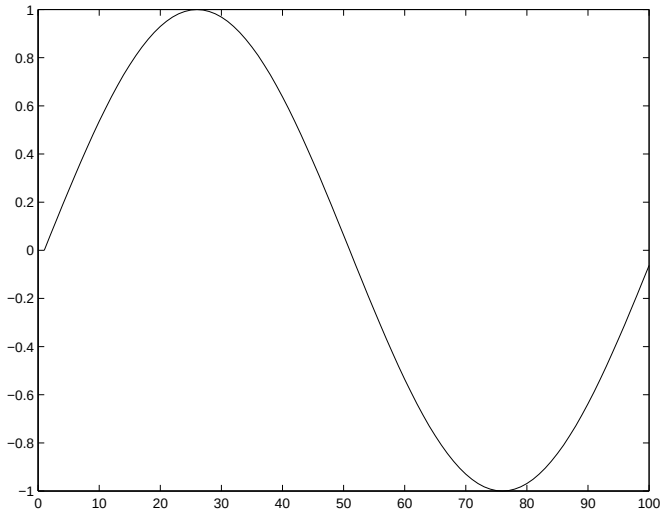

```
>> filteredf = conv(f,h);
>> plot(filteredf(101:200))      % view the part which didn't involve
>>                               % zero padding
```



The idea is that if the frequency domain filter $H(k)$ passes a particular frequency, then its time domain version $h(t)$ will contain a sinusoid of that frequency (remember, the inverse Fourier transform is just the sum of sinusoids). Then convolving by $h(t)$ will *pick out* that frequency and amplify or suppress it depending on the amplitude of the sinusoid in $h(t)$. You can see this in the inverse Fourier transformed version of our lowpass filter, which contains only low-frequency oscillations. Convolving by it matches low frequency sinusoids in the sound $f(t)$, preserving them while dampening oscillations at other frequencies.

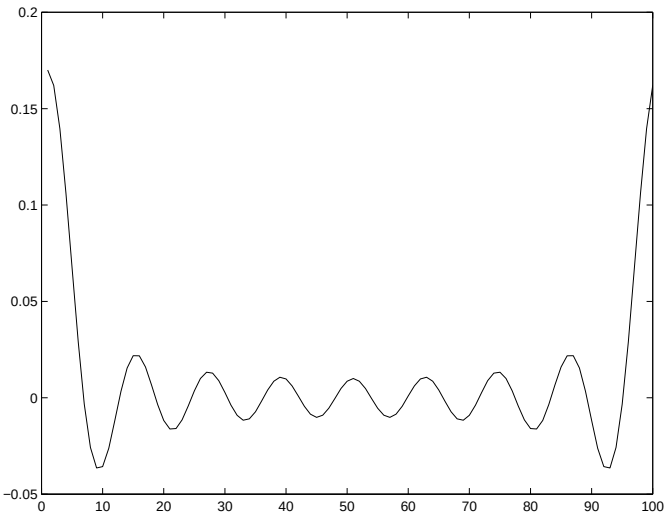
The vector $h(t)$ is also known as the *impulse response* function because it is the vector that would result if a sound consisting of a single momentary impulse of unit amplitude were passed through the filter. That is, if $\delta(t)$ is one at $t = 0$ and zero elsewhere, $f(t) \star d(t) = f(t)$: convolving a function by an impulse returns the same function.

```
>> d = [1,zeros(1,99)];
>> f2 = conv(f1,d);
>> plot(f2(1:100))
```



And another way to recover the time domain filter $h(t)$ (here phase-shifted from how we previously viewed it) is to pass an impulse through it and look at the response:

```
>> plot(real(ifft(c2e(e2c(fft(d)) .* H))))
```



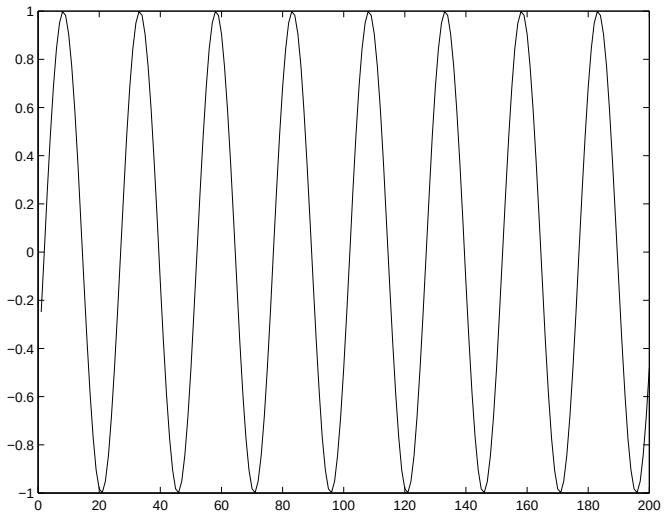
4 Correlation

A function related to convolution is *cross correlation*. We may want to see if some vector $\mathbf{x}$ of length n resembles a time-shifted version of another vector $\mathbf{y}$, for instance to calculate interaural time delays. Then we can perform the same operation as convolution, only without reversing $\mathbf{x}$, to define a vector $\mathbf{z}$ as:

$$z(i) = \sum_{j=1}^{i} x(n - i + j)z(j)$$

The vector $\mathbf{z}$ should peak at some offset i at which $\mathbf{y}$ best matches $\mathbf{x}$. (Note the way we have defined this: if $\mathbf{x}$ and $\mathbf{y}$ are the same vector then they should peak at $i = n$. If $\mathbf{x}$ is ahead of $\mathbf{y}$ then $i > n$ and vice versa.) If some vector $\mathbf{x}$ is periodic, then its *autocorrelation* — its cross correlation with itself — will have periodic peaks spaced by the period of $\mathbf{x}$. The MATLAB function for a cross correlation is `xcorr(x,y)`:

```
>> t = [0:.001:.99];
>> x = sin(2*pi*40*t);
>> plot(x(400:599))
```




```
>> xx = xcorr(x,x);
>> plot(xx(900:1099))
```