CS170 – Spring 2007 – Lecture 8 – Feb 8

1 Fast Fourier Transform, or FFT

The FFT is a basic algorithm underlying much of signal processing, image processing, and data compression. When we all start inferfacing with our computers by talking to them (not too long from now), the first phase of any speech recognition algorithm will be to digitize our speech into a vector of numbers, and then to take an FFT of the resulting vector. So one could argue that the FFT might become one of the most commonly executed algorithms on most computers.

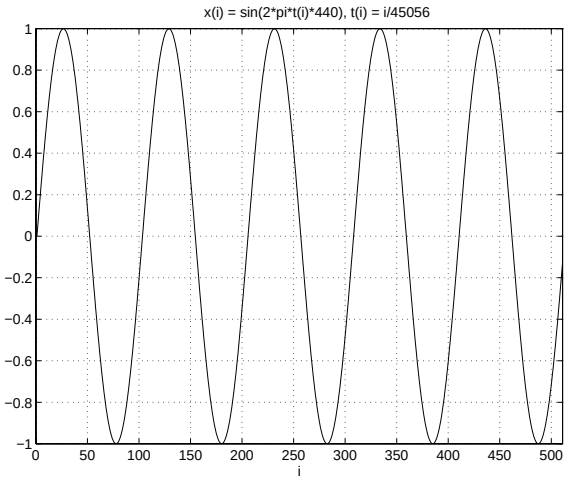
There are a number of ways to understand what the FFT is doing, and eventually we will use all of them:

- The FFT can be described as multiplying an input vector x of n numbers by a particular n-by-n matrix F_n, called the DFT matrix (Discrete Fourier Transform), to get an output vector y of n numbers: $y = F_n \cdot x$. This is the simplest way to describe the FFT, and shows that a straightforward implementation with 2 nested loops would cost $2n^2$ operations. The importance of the FFT is that it performs this matrix-vector in just $O(n \log n)$ steps using divide-and-conquer. Furthermore, it is possible to compute x from y, i.e. compute $x = F_n^{-1} y$, using nearly the same algorithm, and just as fast. Practical uses of the FFT require both multiplying by F_n and F_n^{-1}.

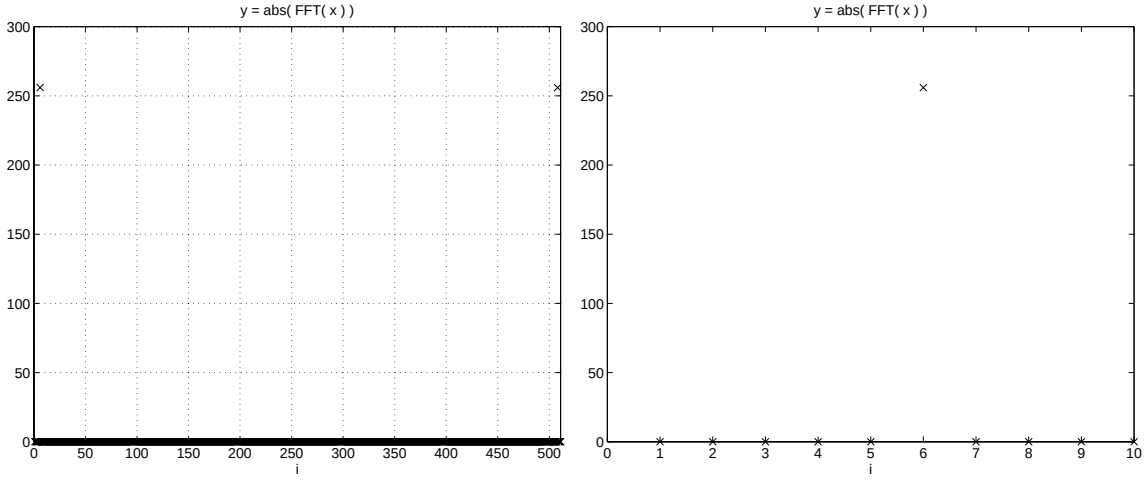
- The FFT can also be described as evaluating a polynomial with coefficients in x at a special set of n points, to get n polynomial values in y. We will use this polynomial-evaluation-interpretation to derive our $O(n \log n)$ algorithm. The inverse operation in called *interpolation*: given the values of the polynomial y, find its coefficients x.

- To pursue the signal processing interpretation mentioned above, imagine sitting down at a piano and playing a chord, a set of notes. Each note has a characteristic frequency (middle A is 440 cycles per second, for example). A microphone digitizing this sound will produce a sequence of numbers that represent this set of notes, by measuring the air pressure on the microphone at evenly spaced *sampling times* t_1, t_2, ... , t_i, where $t_i = i \cdot \Delta t$. Δt is the *interval* between consecutive samples, and $1/\Delta t$ is called the *sampling frequency*. If there were only the single, pure middle A frequency, then the sequence of numbers representing air pressure would form a sine curve, $x_i = d \cdot \sin(2 \cdot \pi \cdot t_i \cdot 440)$. To be concrete, suppose $1/\Delta t = 45056$ per second (or 45056 Hertz), which is a reasonable

sampling frequency for sound[1]. The scalar d is the *maximum amplitude* of the curve, which depends on how loud the sound is; for the pictures below we take $d = 1$. Then a plot of x_i would look like the following (the horizontal axis goes from $i = 0$ ($t_0 = 0$) to $i = 511$ ($t_i \approx .011$ seconds) so that $2 \cdot \pi \cdot t_i \cdot 440$ goes from 0 to $\approx 10\pi$, and the sine curve has about 5 full periodic cycles:



The next plots show the absolute values of the entries of $y = F_n \cdot x$. In the plot on the left, you see that nearly all the components are about zero, except for two near i=0 and i=512. To see better, the plot on the right blows up the picture near i=0 (the other end of the graph is symmetric):

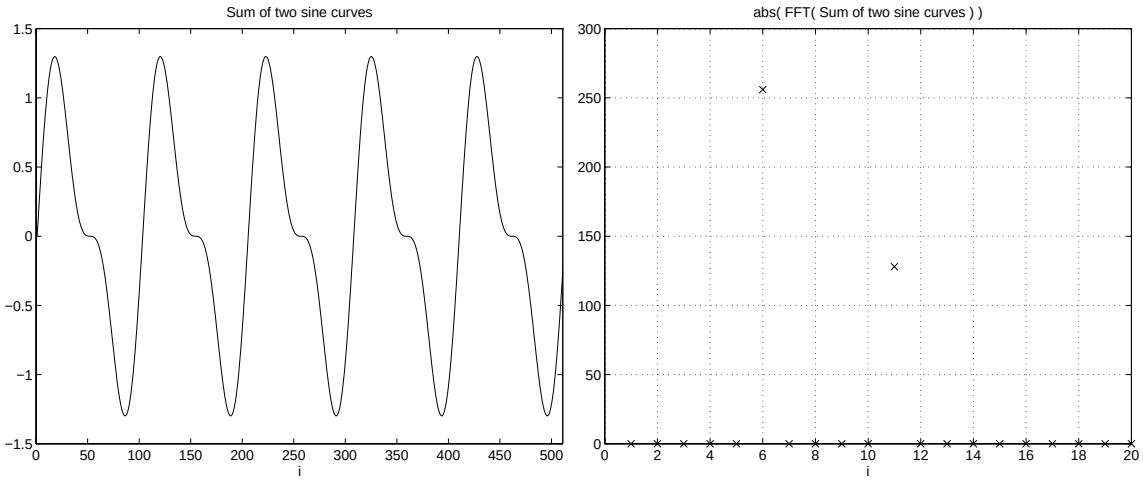


We will see that the fact that only one value of y_i is large for i between 0 and 256 means that x consists of a sine wave of a single frequency, and since the large value is y_6 then that frequency is one that consists of 5=6-1 full periodic cycles in x_i as i runs from 0 to 512.

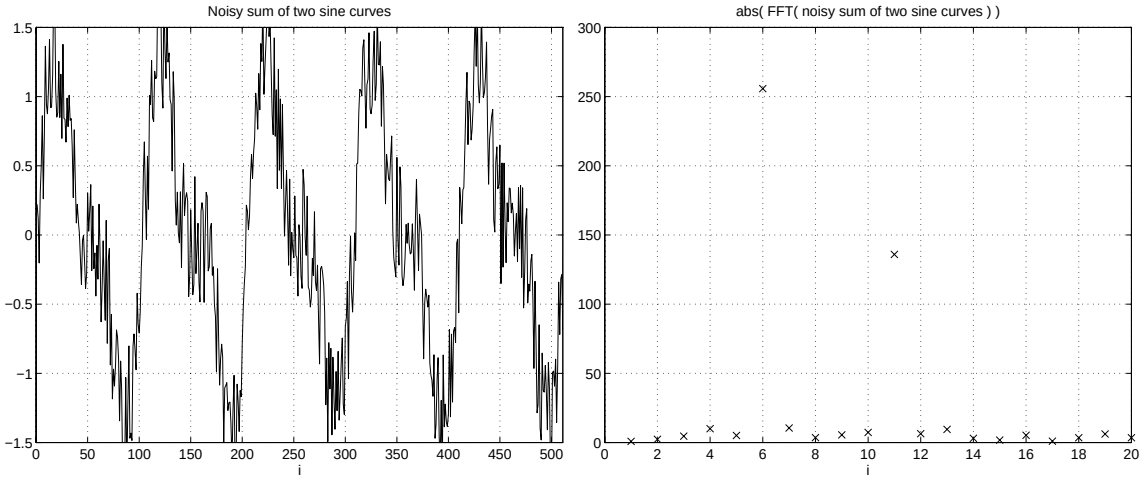
To see how the FFT might be used in practice, suppose that the signal consists not just

[1]I have chosen 45056 to make the pictures below look particularly nice, but all the ideas work for different sampling frequencies.

of the pure note A (frequency 440), but also an A one octave higher (i.e. with twice the frequency) and half as loud: $x_i = \sin(2\pi t_i 440) + .5\sin(2\pi t_i 880)$. Both x_i and its FFT (in the range $i = 0$ to 20) are shown below. The larger nonzero component of the FFT corresponds to our original signal $\sin(2\pi t_i 440)$, and the second one corresponds to the sine curve of twice the frequency and half the amplitude.



But suppose that there is "noise" in the room where the microphone is recording the piano, which makes the signal x_i "noisy" as shown below to the left (call it x'_i). (In this example we added a random number between $-.5$ and .5 to each x_i to get x'_i.) x and x' appear very different, so that it would seem difficult to recover the true x from noisy x'. But by examining the FFT of x'_i below, it is clear that the signal still mostly consists of two sine waves. By simply zeroing out small components of the FFT of x' (those below the "noise level") we can recover the FFT of our two pure sine curves, and then get nearly the original signal $x(i)$ back. This is a very simple example of *filtering*, and is discussed further in other EE courses.

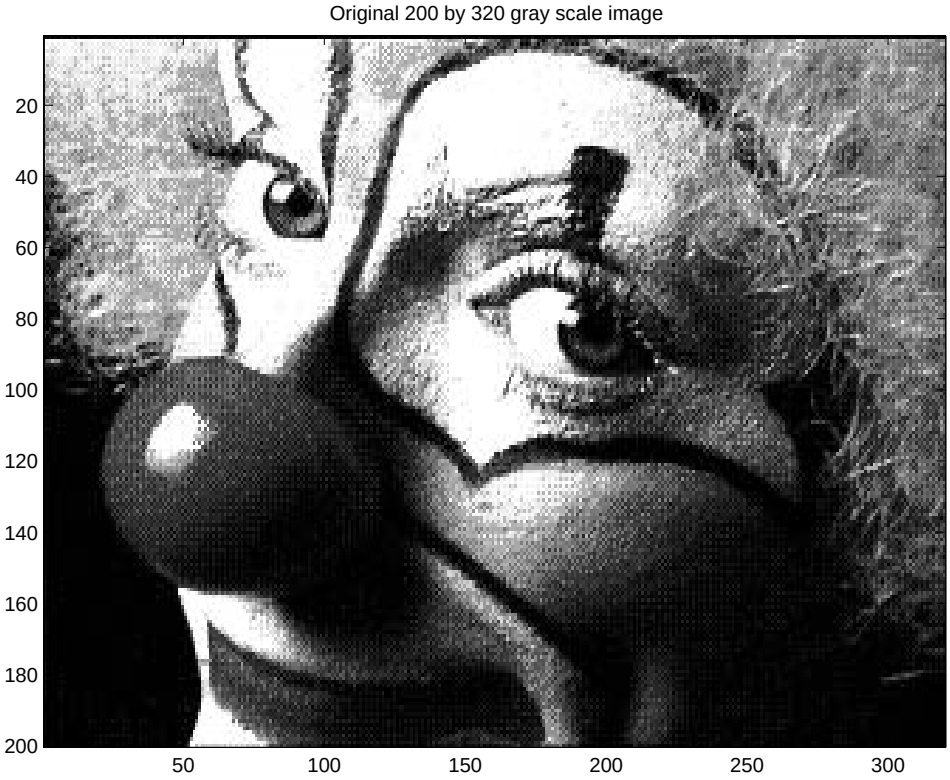


For a Java applet that plots x and $y = F_n \cdot x$ at the same time, and lets you move the x_i (or y_i) with your mouse, and automatically updates y_i (or x_i), see

http://sepwww.stanford.edu/oldsep/hale/FftLab.html. Playing with this will help your intuition about what the FFT does.

Another interactive tool for exploring the FFT is Matlab, for which there is a campus-wide site liense. All the above graphs were produced using Matlab.

Here is one more example, using the FFT for image compression. An image is just a two dimension array of numbers, or a matrix, where each matrix entry represents the brightness of a pixel. For example, 0 might mean black, 1 might mean pure white, and numbers in between are various shades of gray. Below is shown such a 200-by-320 image.



What does it mean to use the FFT to compress this image? After all, an image is 2-dimensional, and so far we have talked about FFTs of 1-dimensional arrays. It turns out that the right thing to do is the following. Let X be the m-by-n array, or image. Then the easiest way to describe the algorithm is to compute $ffX = F_m \cdot X \cdot F_n$. The way these two matrix-multiplications are actually implemented is as follows:

1. For each column of X, compute its FFT. Call the m-by-n array of column FFTs fX. In other words, column i of fX is the FFT of column i of X.

2. For each row of fX, compute its FFT. Call the m-by-n array of row FFTs ffX. In other words, row i of ffX is the FFT of row i of fX.

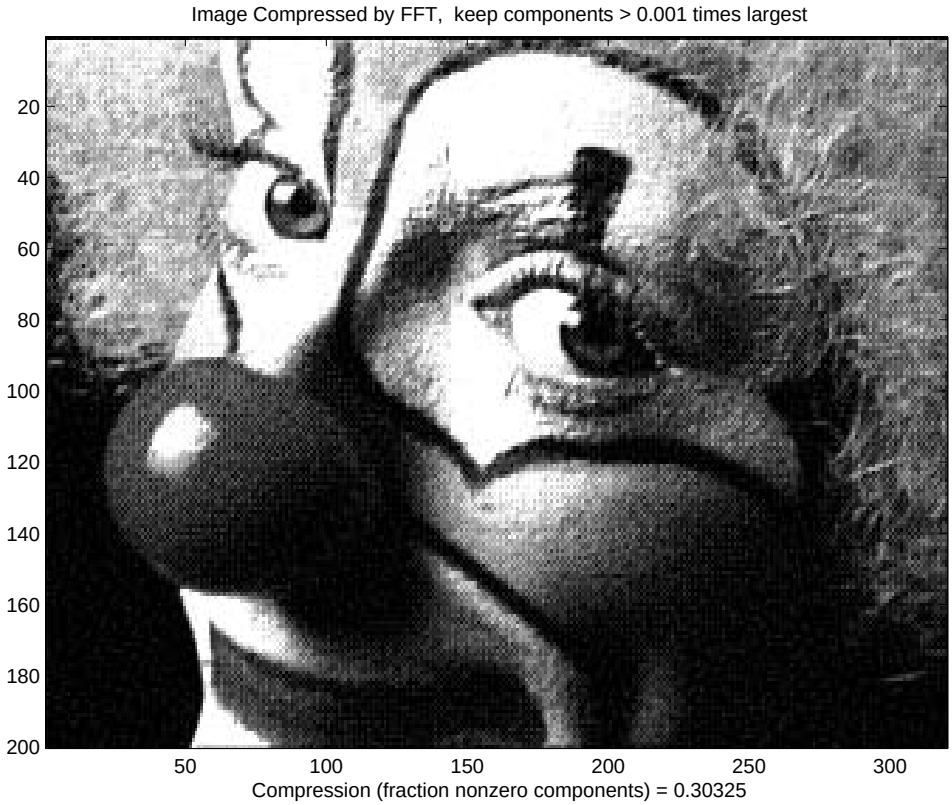
ffX is called the *2-dimensional FFT* of X.

We use ffX for compression as follows. The motivation is similar to what we saw before,

that the FFT of a pure sine curve has only two nonzero entries. Rather than store all the zeros, we could simply store the nonzero entries, and their locations. To compress, we will not only "throw away" the zero entries, but all entries whose absolute values are less than some threshold t. If we choose t to be small, we keep most entries, and do not compress very much, but the compressed version will look very similar to the original image. If we choose t to be large, we will compress a lot, but perhaps get a worse image. For example, when ffX is 200-by-320, and throwing away all the entries of ffX greater than .002 times the largest entry of ffX leave only 8.6% of the original entries, then the storage drops from 200*320 = 64000 words to 3*.086*64000 = 16512 words. (The factor 3 comes from having to store each nonzero entry, along with its row index and column index. More clever storage schemes for ffX use even less space.)

This is called "lossy compression", as opposed to the "lossless compression" of a scheme like Lempel-Ziv (used in gzip). The idea is that we can compress more if we are willing to actually lose information. If all we care about is whether an image "looks good", which is subjective, then it is ok to lose some information.

Here are some examples of compressed clowns. The tradeoff between compression and loss of accuracy is evident.



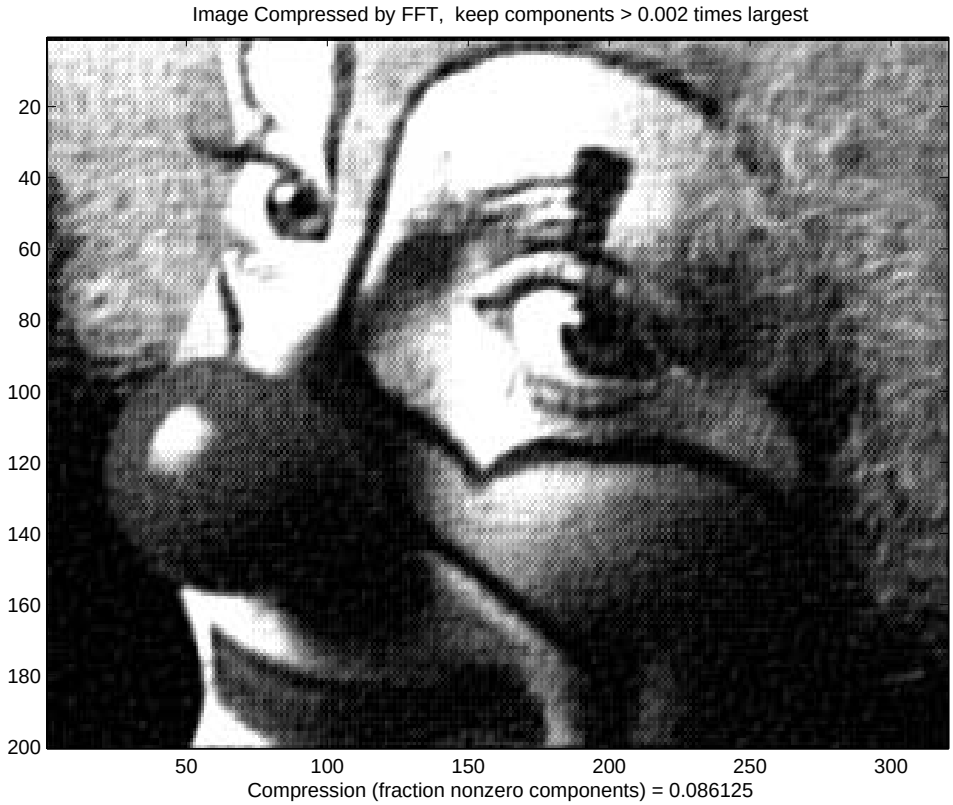


Image Compressed by FFT, keep components > 0.002 times largest
20
40
60
80
100
120
140
160
180
200
50
100
150
200
250
300
Compression (fraction nonzero components) = 0.086125

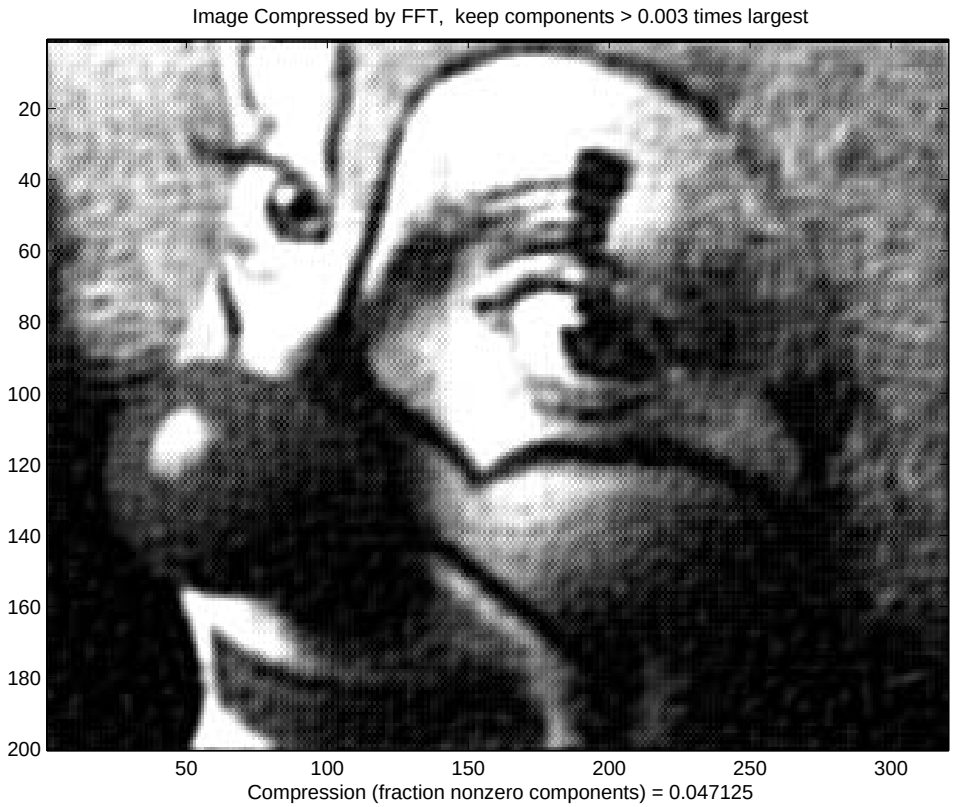
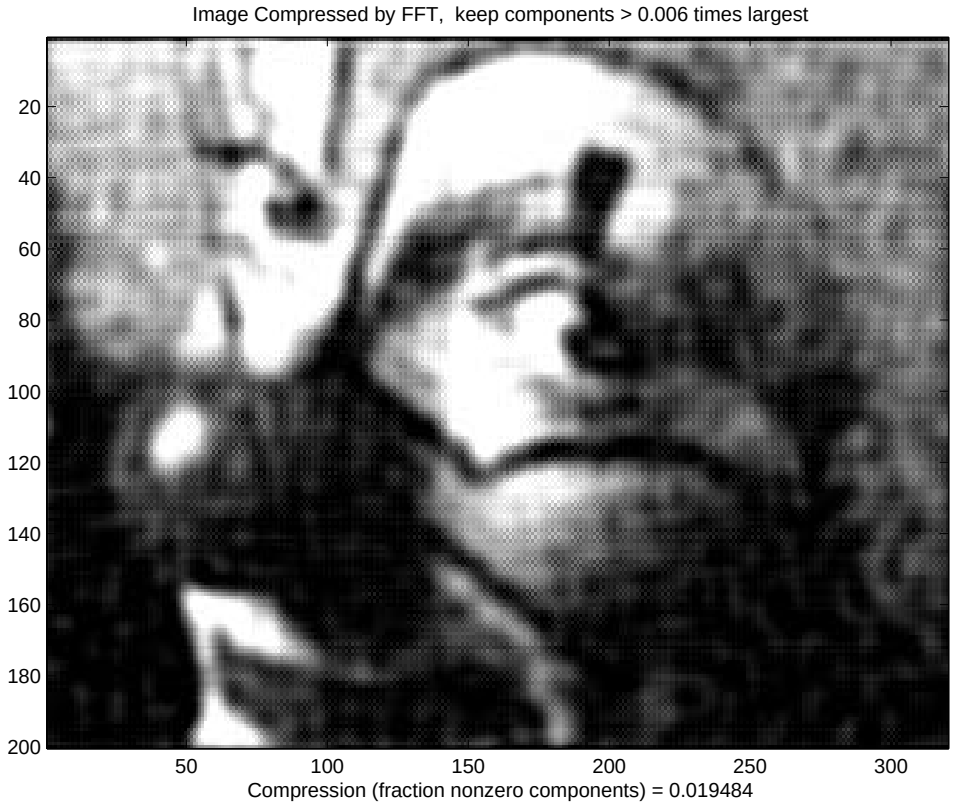
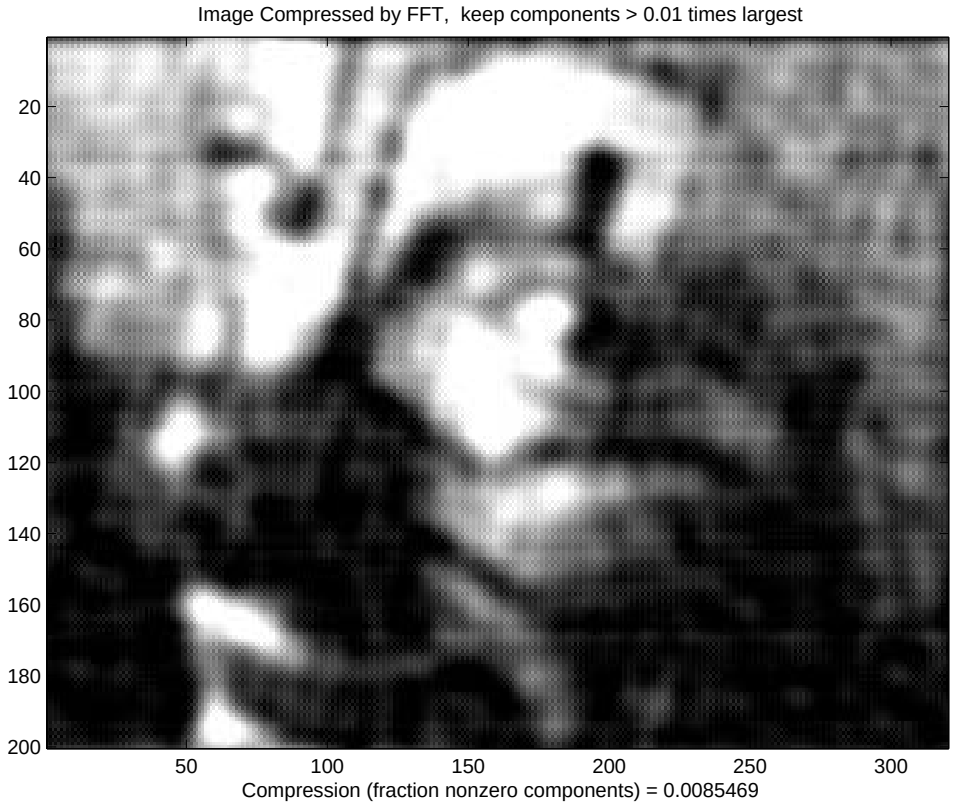


Image Compressed by FFT, keep components > 0.003 times largest
20
40
60
80
100
120
140
160
180
200
50
100
150
200
250
300
Compression (fraction nonzero components) = 0.047125





The complete Matlab program that produced the above images is available on the class home page. MPEG uses the FFT to compress images, although in a more sophisticated way than described here.

2 Review of Complex Arithmetic

To define the DFT matrix F_n, we will first review *complex numbers*. We let i denote $\sqrt{-1}$, that is $i^2 = -1$. i is called the *pure imaginary unit*. Then complex numbers are of the form $z = a + i \cdot b$, where a and b are any real numbers. a is the *real part of* z, sometimes written $a = \mathrm{Re}z$, and b is the *imaginary part of* z, sometimes written $a = \mathrm{Im}z$. $|z| = \sqrt{a^2 + b^2}$ is called the *absolute value* or *magnitude* of z. $\bar{z} = a - i \cdot b$ is called the *complex conjugate* of $z = a + i \cdot b$. We can think about complex number geometrically by thinking of $z = a + i \cdot b$ as a point in a 2-dimensional plane, called the *complex plane*, where a is the x-coordinate and b is the y-coordinate (you will explore this further on your homework).
We can do addition, subtraction, multiplication and division with complex numbers analogously to real numbers:

$$\begin{aligned}
(a + i \cdot b) + (c + i \cdot d) &= (a + c) + i \cdot (b + d) \\
(a + i \cdot b) \cdot (c + i \cdot d) &= [a \cdot c] + [a \cdot (i \cdot d)] + [(i \cdot b) \cdot c] + [(i \cdot b) \cdot (i \cdot d)] \\
&= [a \cdot c] + [a \cdot (i \cdot d)] + [(i \cdot b) \cdot c] + [b \cdot d \cdot i^2] \\
&= [a \cdot c] + [a \cdot (i \cdot d)] + [(i \cdot b) \cdot c] - [b \cdot d] \\
&= [a \cdot c - b \cdot d] + i \cdot [a \cdot d + b \cdot c]
\end{aligned}$$

All the usual arithmetic rules (commutativity, associativity and distributivity) can be shown to hold for complex arithmetic. For example, the reader can confirm that $z \cdot \bar{z} = |z|^2$ for any complex number z. Two complex numbers $z_1 = a + i \cdot b$ and $z_2 = c + i \cdot d$ are the same if and only if $a = c$ and $b = d$, i.e. if $z_1 - z_2 = 0 + i \cdot 0 = 0$. We leave it to the reader to show that

$$(a + i \cdot b) \cdot (\frac{a}{a^2 + b^2} - i \cdot \frac{b}{a^2 + b^2}) = 1 + i \cdot 0 = 1$$

so that

$$\frac{1}{z} = \frac{1}{a + i \cdot b} = \frac{a}{a^2 + b^2} - i \cdot \frac{b}{a^2 + b^2} = \frac{\overline{z}}{|z|^2}$$

Another fact about complex numbers that we need is that for any number θ

$$e^{i \cdot \theta} = \cos\theta + i \cdot \sin\theta$$

To illustrate this fact note that by the law of exponents and above definition of $e^{i \cdot \theta}$

$$\begin{aligned}
e^{i \cdot a} \cdot e^{i \cdot b} &= e^{i \cdot (a+b)} \\
&= \cos(a + b) + i \cdot \sin(a + b) \qquad (1)
\end{aligned}$$

But we can also write

$$\begin{aligned}
e^{i \cdot a} \cdot e^{i \cdot b} &= (\cos a + i \cdot \sin a) \cdot (\cos b + i \cdot \sin b) \\
&= (\cos a \cdot \cos b - \sin a \cdot \sin b) + i \cdot (\sin a \cdot \cos b + \cos a \cdot \sin b) \qquad (2)
\end{aligned}$$

Comparing equations (1) and (2), we get

$$\begin{aligned}\cos(a+b) &= \cos a \cdot \cos b - \sin a \cdot \sin b \\ \sin(a+b) &= \sin a \cdot \cos b + \cos a \cdot \sin b\end{aligned}$$

which are standard trigonometric identities (remembering that $e^{i \cdot a} \cdot e^{i \cdot b} = e^{i \cdot (a+b)}$ and $e^{i \cdot \theta} = \cos\theta + i \cdot \sin\theta$ is probably the easiest way to remember them!).
Another simple fact we will need is that

$$\overline{e^{i \cdot \theta}} = \overline{\cos\theta + i \cdot \sin\theta} = \cos\theta - i \cdot \sin\theta = \cos(-\theta) + i \cdot \sin(-\theta) = e^{-i \cdot \theta}$$

3 Definition of the DFT Matrix

We next need to define the *primitive n-th root of unity*

$$\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i \cdot \sin\frac{2\pi}{n}$$

It is called an n-th root of unity because

$$\omega^n = e^{(2\pi i/n)n} = e^{2\pi i} = \cos 2\pi + i \cdot \sin 2\pi = 1 + i \cdot 0 = 1$$

Note that all other integer powers ω^j are also n-th roots of unity; since they are all powers of ω, ω is called *primitive*.
Finally, we can define the n-by-n DFT matrix F_n. We will denote its entries by $f_{jk}^{(n)}$ or just f_{jk} if n is understood from context. To make notation easier later we will let the subscripts j and k run from 0 to $n-1$ instead of from 1 to n. Then

$$f_{jk}^{(n)} = \omega^{jk} = e^{\frac{2\pi ijk}{n}}$$

Here is an important formula that shows that the *inverse* of the DFT matrix is very simple:
Lemma. Let $G_n = \frac{1}{n}\overline{F_n}$, that is

$$g_{jk} = \frac{\overline{f}_{jk}}{n} = \frac{\overline{\omega^{jk}}}{n} = \frac{\omega^{-jk}}{n}$$

Then G_n is the inverse matrix of F_n. In other words, if $y = F_n \cdot x$, then $x = G_n \cdot y$.
Proof: We compute the j,k-th component of the matrix product $G_n \cdot F_n$, and confirm that it equals 1 if $j = k$ and 0 otherwise.

$$\begin{aligned}(G_n \cdot F_n)_{j,k} &= \sum_{r=0}^{n-1} g_{jr} f_{rk} \\ &= \frac{1}{n}\sum_{r=0}^{n-1} \bar{f}_{jr} f_{rk}\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{n}\sum_{r=0}^{n-1}\omega^{-jr}\omega^{rk} \\
&= \frac{1}{n}\sum_{r=0}^{n-1}\omega^{r(k-j)}
\end{aligned}
$$

Now if $j = k$, then $\omega^{r(k-j)} = \omega^0 = 1$, and the expression equals 1 as desired. Otherwise, if $j \neq k$, we have a geometric sum which equals

$$
\begin{aligned}
(G_n \cdot F_n)_{j,k} &= \frac{1}{n}\sum_{r=0}^{n-1}(\omega^{k-j})^r \\
&= \frac{1}{n}\frac{\omega^{(k-j)n}-1}{\omega^{j-k}-1} \\
&= \frac{1}{n}\frac{(\omega^n)^{k-j}-1}{\omega^{j-k}-1} \\
&= \frac{1}{n}\frac{1^{k-j}-1}{\omega^{j-k}-1} \\
&= 0
\end{aligned}
$$

as desired.

Note that evaluating $y = F_n \cdot x$ by standard matrix-vector multiplication would cost $2n^2$ arithmetic operations. The FFT is a way to perform matrix-multiplication by the special matrix F_n in just $O(n \log n)$ operations.

4 Interpreting $y = F_n \cdot x$ as polynomial evaluation

By examining the formula for y_j in $y = F_n \cdot x$ we see

$$y_j = \sum_{k=0}^{n-1} f_{jk} \cdot x_k = \sum_{k=0}^{n-1} (\omega^j)^k \cdot x_k$$

so that y_j is just the value of the polynomial $p(z)$ at $z = \omega^j$, where

$$p(z) = \sum_{k=0}^{n-1} z^k \cdot x_k$$

In other words, the x_k are just the coefficients of the polynomial $p(z)$, from the constant term x_0 to the highest order term x_{n-1}. Note that the *degree* of $p(z)$, the highest power of z that appears in $p(z)$, is at most $n-1$ (if $x_{n-1} \neq 0$).

By our Lemma in the last section $x = \frac{1}{n}\overline{F_n} \cdot y$ gives a formula for finding the coefficients x_j of the polynomial $p(z)$ given its values at the n-th roots of unity ω^k, for $0 \leq k \leq n-1$. Computing polynomial coefficients given polynomial values is called *interpolation*.

5 Deriving the FFT

We will use this polynomial evaluation interpretation to derive an $O(n \log n)$ algorithm using divide and conquer. More precisely, the problem that we will divide-and-conquer is "evaluate a polynomial of degree $n-1$ at n points". Write the polynomial p that we just defined as follows, putting the "even terms" into one group and the "odd terms" into another:

$$\begin{aligned} p(z) &= x_0 + x_1 \cdot z + x_2 \cdot z^2 + \cdots + x_{n-1} \cdot z^{n-1} \\ &= (x_0 + x_2 \cdot z^2 + x_4 \cdot z^4 + \cdots) + (x_1 \cdot z + x_3 \cdot z^3 + x_5 \cdot z^5 + \cdots) \\ &= (x_0 + x_2 \cdot z^2 + x_4 \cdot z^4 + \cdots) + z \cdot (x_1 + x_3 \cdot z^2 + x_5 \cdot z^4 + \cdots) \\ &= p_{\text{even}}(z^2) + z \cdot p_{\text{odd}}(z^2) \end{aligned}$$

Here we have defined two new polynomials with the even-numbered coefficients ($p_{\text{even}}(z') = \sum_{i=0}^{n/2-1} x_{2i} z'^i$) and odd-numbered coefficients ($p_{\text{odd}}(z') = \sum_{i=0}^{n/2-1} x_{2i+1} z'^i$) as coefficients. So we have reduced the problem of computing the FFT, or evaluating p at n points ω^j ($0 \leq j \leq n-1$), to evaluating the two polynomials p_{even} and p_{odd} at $z' = z^2 = \omega^{2j}$. To get a divide-and-conquer algorithm, we have to show that evaluating p_{even} and p_{odd} are "half as big problems" as evaluating p:

- Both p_{even} and p_{odd} have degree at most $n/2 - 1$, whereas p has degree at most $n-1$. So the "degree + 1" has been cut in half.
- We have to show that we only have to evaluate p_{even} and p_{odd} at $n/2$ points instead of n. It appears that we have to evaluate them at the n points ω^{2j}, $0 \leq j \leq n-1$. But it turns out that only $n/2 - 1$ of these numbers are distinct, because

  $$\omega^{2j} = e^{2\pi i \frac{2j}{n}} = e^{2\pi i \left(\frac{2j}{n} + 1\right)} = e^{2\pi i \frac{2(j+\frac{n}{2})}{n}} = \omega^{2(j+\frac{n}{2})}$$

  i.e. the $\frac{n}{2}$ numbers $\omega^{2 \cdot 0}$, $\omega^{2 \cdot 1}$, $\omega^{2 \cdot 2}$, ... $\omega^{2 \cdot (\frac{n}{2}-1)}$ are the same as the $\frac{n}{2}$ numbers $\omega^{2 \cdot \frac{n}{2}}$, $\omega^{2 \cdot (\frac{n}{2}+1)}$, $\omega^{2 \cdot (\frac{n}{2}+2)}$, ... $\omega^{2 \cdot (\frac{n}{2}+(\frac{n}{2}-1))}$.

This lets us write down our first, recursive version of the FFT:

function FFT(x) ... $n = \text{length}(x)$
  if $n = 1$
    return x ... 1-by-1 FFT is trivial
  else
    p_{even} =FFT($(x_0, x_2, x_4, ...x_{n-2})$)
    p_{odd} =FFT($(x_1, x_3, x_5, ...x_{n-1})$)
    $\omega = e^{2\pi i/n}$
    for $j = 0$ to $n/2 - 1$
      $p_j = p_{\text{even},j} + \omega^j \cdot p_{\text{odd},j}$
      $p_{j+n/2} = p_{\text{even},j} + \omega^{j+n/2} \cdot p_{\text{odd},j}$
    endfor
  end if

We analyze the complexity of this divide-and-conquer scheme in the usual fashion, by writing down a recurrence: $T(n) = 2T(n/2) + \Theta(n)$, or $T(n) = O(n \log n)$ by our general solution to such recurrences.

6 Making the FFT Efficient

In practice the FFT may not be implemented recursively, but with two simple, nested loops, that we get by "unwinding" the recurrence. The complexity is the same in the $O(.)$ sense, but may be more efficient.

Like matrix multiplication, the FFT is so important that an automatic tuning system to search for the fastest implementation on any particular computer and for any value of n has been written; see www.fftw.org.

First we can eliminate some arithmetic in the inner loop by noting that

$$\omega^{j+n/2} = e^{2\pi i(j+n/2)} = e^{2\pi ij+\pi i} = e^{2\pi i} \cdot e^{\pi i} = -e^{2\pi i} = -\omega^j$$

so the loop

    for $j = 0$ to $n/2 - 1$
      $p_j = p_{\text{even},j} + \omega^j \cdot p_{\text{odd},j}$
      $p_{j+n/2} = p_{\text{even},j} + \omega^{j+n/2} \cdot p_{\text{odd},j}$
    endfor

reduces to

for $j = 0$ to $n/2 - 1$

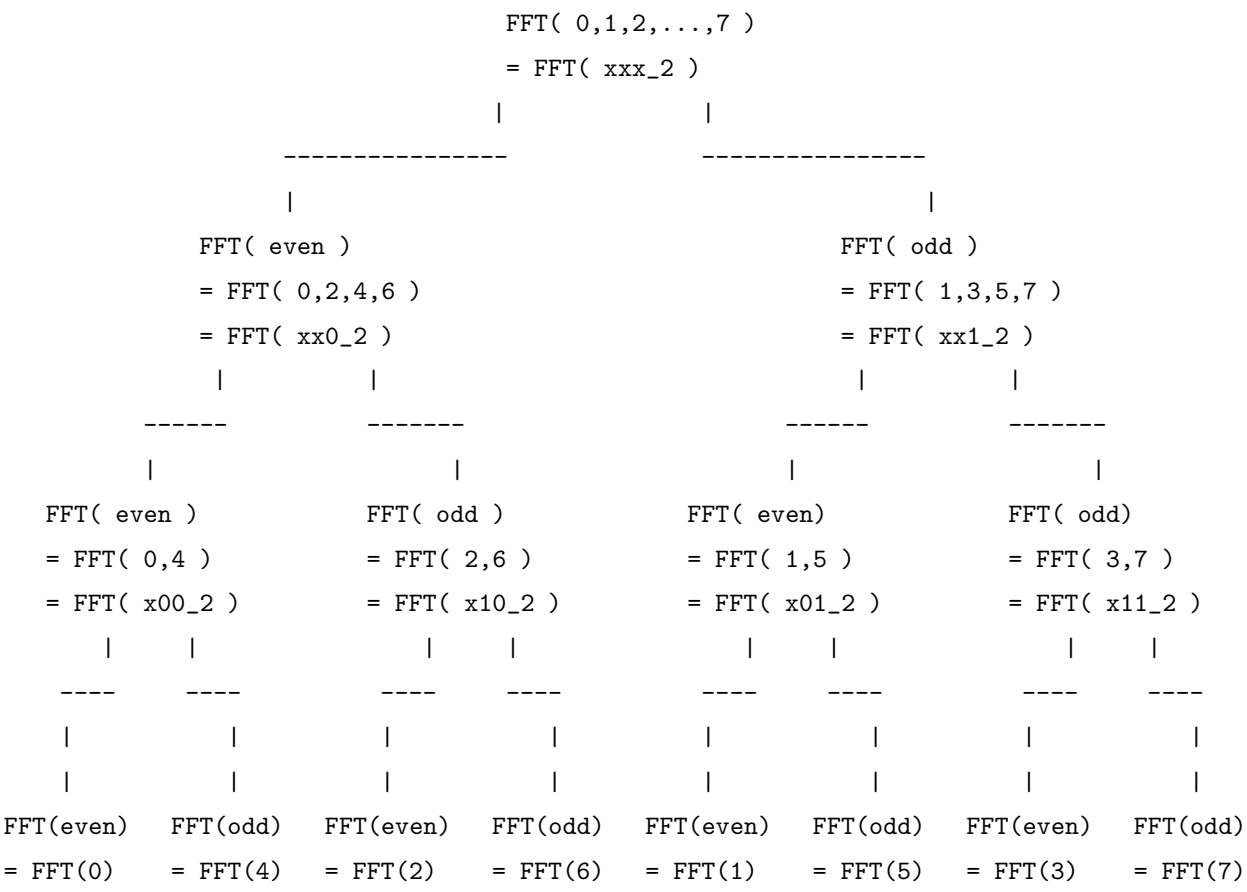
$tmp = \omega^j \cdot p_{\text{odd},j}$

$p_j = p_{\text{even},j} + tmp$

$p_{j+n/2} = p_{\text{even},j} - tmp$

endfor

The next figure unwinds the recursion for $n = 8$. Note there are $4 = \log_2 n + 1$ levels in the tree; the bottom level would consist of recursive calls to FFTs of vectors of length 1, and so this is just the initial data. Thus there are really $\log_2 n$ levels of recursion.



We have indicated which components of x are passed as arguments to each recursive call. To keep track of the "evens" and "odds", we have both written out the subscripts explicitly in the figure above, as well as shown their bit patterns. For example, in the root xxx_2 means that the subscripts of the arguments consist of all 8 binary numbers with 3 bits (each x could be a 0 or 1). In the right child of the root, corresponding to the recursive call on the even subscripts 0,2,4,6, the bit pattern of these subscripts is shown as xx0_2, meaning all 4 binary numbers with a 0 rightmost bit. Similarly, the right child of the root handles odd subscripts 1,3,5,7 with bit patterns xx1_2.

What is important to note is that this pattern repeats at each level, the even subscripts are gotten by taking the rightmost x in the bit pattern of the parent, and changing it to 0. The odd subscripts are gotten by taking the same bit and changing it to 1.
The nonrecursive FFT evaluates the above tree by doing all the leaf nodes first, then all their parents, then all their parents, etc., up to the root. In contrast, the recursive FFT amounts to doing postorder traversal on the above tree. Any order is legal as long as the children are completed before the parents, but our nonrecursive FFT (see below) will require just two nested loops: the outer one from tree level $\log_2 n$ down to 1, and the inner loop from 0 to n, to do all the arithmetic in the body of the FFT. Let $s = \log_2 n$.

function FFT(x) ... overwrite x with FFT(x)
  for $j = 0$ to $s - 1$ ... for each level of the tree from bottom to top
    for $k = 0$ to $n/2 - 1$ ... for each pair of even/odd values to combine
      let $k = (k_0 k_1 \cdots k_{s-3} k_{s-2})_2$ be the binary representation of k
        ... i.e. k_0 is the leading bit of k, k_1 is the next bit, etc.
        ... note that k only has $s - 1$ bits
      let $k_{\text{even}} = (k_0 \cdots k_{j-1} 0 k_j \cdots k_{s-2})_2$ ... subscript of even coefficient
      ...i.e. take the bits of k, and insert a 0 before bit k_j and/or after bit k_{j-1}
      let $k_{\text{odd}} = (k_0 \cdots k_{j-1} 1 k_j \cdots k_{s-2})_2$ ... subscript of odd coefficient
      ...i.e. take the bits of k, and insert a 1 before bit k_j and/or after bit k_{j-1}
      $x\_even = x_{k_{\text{even}}}$
      $x\_odd = x_{k_{\text{odd}}}$
      $w = e^{2\pi i e(j,k)/n}$ ... power of ω stored in a table
      $tmp = w \cdot x\_odd$
      $x_{k_{\text{even}}} = x\_even + tmp$
      $x_{k_{\text{odd}}} = x\_even - tmp$
    endfor
  endfor

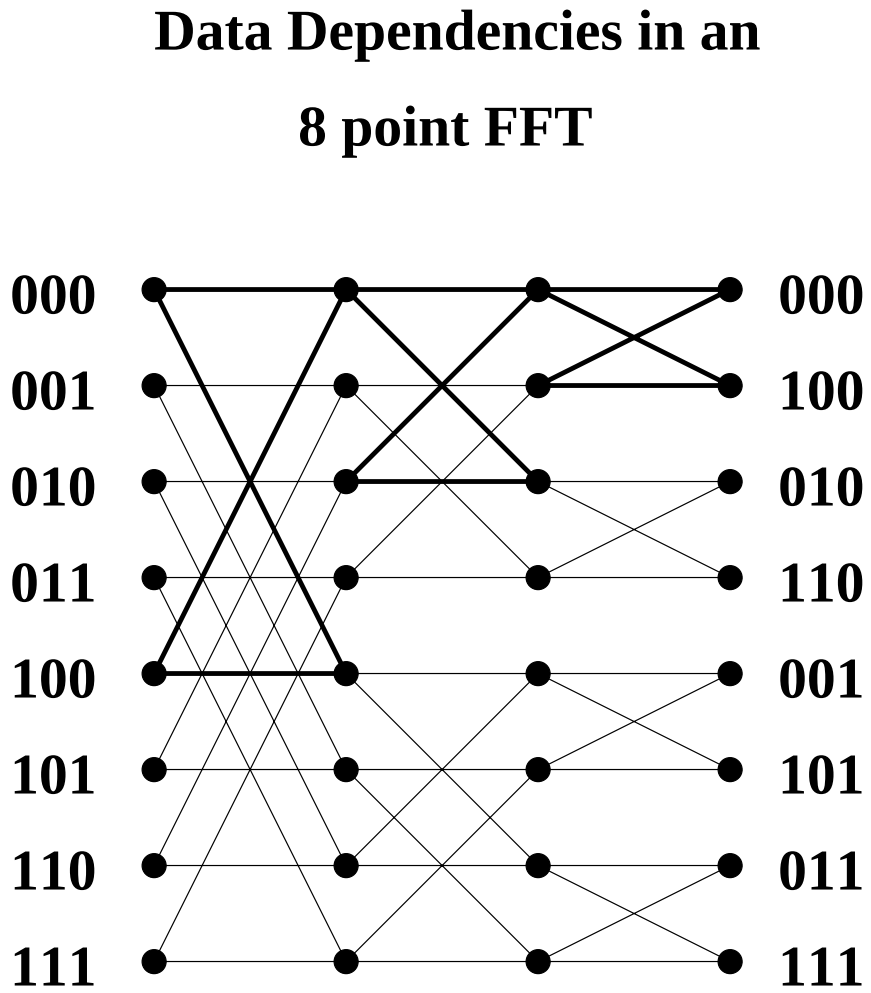
In this algorithm, as bits $k_0 \cdot k_{j-1}$ take on their 2^j possible values for a fixed value of bits $k_j \cdots k_{s-2}$, the inner loop executes one 2^{j+1} point FFT at level j of the tree ($j = s - 1$ is the root). Different values of bits $k_j \cdots k_{s-2}$ correspond to different FFT calls at the same level of the tree.
This implementation makes it easy to see that the complexity is $s \cdot (n/2) \cdot 3 = (3/2)n \log_2 n$, where we have counted the number of complex arithmetic operations in the inner loop, and assumed that w is looked up in a table depending on the integer $e(j, k)$. If we take into

account that complex addition costs 2 real additions, and complex multiplications take 4 real multiplications and 2 real additions, the *real* operation count is $5n \log_2 n$.

Let us examine the pattern of subscripts produced by the algorithm. The values of k_{even} and k_{odd} produced by this can be seen to be

		$j=0$	$j=1$	$j=2$
$k=0$	k_{even}	0	0	0
	k_{odd}	4	2	1
$k=1$	k_{even}	1	1	2
	k_{odd}	5	3	3
$k=2$	k_{even}	2	4	4
	k_{odd}	6	6	5
$k=3$	k_{even}	3	5	6
	k_{odd}	7	7	7

A graphical way to see the pattern of subscripts being combined in the inner loop is as follows.



There is one column of dots for each level in the tree. Each dot represents the value of an

entry of the vector x at a level in the tree. So the leftmost column of dots represents the initial data, and the rightmost column represents the final answers. There is an edge connecting a dot to another dot to its right if the value of the right dot is computed using the value of the left dot. For example, consider the top, leftmost, heavy black "butterfly". These 4 edges connecting $x_0 = x_{000_2}$ and $x_4 = x_{100_2}$ in the leftmost column to $x_0 = x_{000_2}$ and $x_4 = x_{100_2}$ in the next column mean that in the first pass through the inner loop ($j = k = 0$), $x_{k_{\rm even}} = x_0$ and $x_{k_{\rm odd}} = x_4$ are combined to get new values of x_0 and x_4. Thus each "butterfly" represents one pass through the inner loop.

The numbers at the left of the picture are the subscripts of the initial data. The numbers at the right are the subscripts of the final data, the computed FFT. It turns out that they are *not* returned in order, but in *bit reverse order*. In other words when the computation is completed $x_1 = x_{001_2}$ does not contain component $1 = 001_2$ of the FFT, but component $4 = 100_2$, because 100_2 is the *bit reversal* of 001_2.

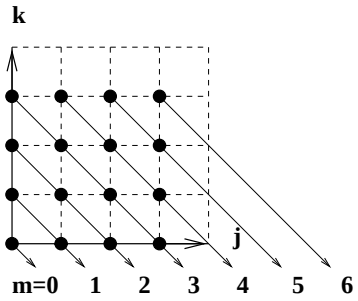
7 Multiplying Polynomials

At the beginning of our discussion of FFTs, we motivated them by a signal processing application, namely *filtering* noise from a signal (sound received by a microphone and digitized). We will now talk about to do *polynomial multiplication* using the FFT, and then how to use this to do filtering. Another common term for this process is *convolution*.

Let $p(z) = \sum_{k=0}^{n-1} p_k z^k$ and $q(z) = \sum_{j=0}^{n-1} q_j z^j$ be two polynomials of degree at most $n-1$. Then their *product* $r(z) = p(z) \cdot q(z)$ is the polynomial of degree at most $2(n-1)$:

$$\begin{aligned}
r(z) &= \sum_{m=0}^{2(n-1)} r_m z^m \\
&= p(z) \cdot q(z) \\
&= (\sum_{k=0}^{n-1} p_k z^k) \cdot (\sum_{j=0}^{n-1} q_j z^j) \\
&= \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} p_k q_j z^{k+j} \\
&= \sum_{m=0}^{2(n-1)} \sum_{l=0}^{m} p_l q_{m-l} z^m \qquad \text{where } m = k + j \\
&= \sum_{m=0}^{2(n-1)} z^m \sum_{l=0}^{m} p_k q_{m-l}
\end{aligned}$$

so $r_m = \sum_{l=0}^{m} p_k q_{m-l}$. The following picture explain how we transformed the sum in line 4 above to the sum in line 5 (each grid point represents a (j, k) value when $n = 4$, and m is constant along diagonals):



We also say that the vector of coefficients of $r(z)$ is the *convolution* of the vector of coefficients of $p(z)$ and $q(z)$.

If we evaluated the r_m using the above formula straightforwardly, it would cost $\sum_{m=0}^{2(n-1)} m = O(n^2)$ operations. The FFT lets us compute all the r_m in $O(n \log n)$ time instead. Here is the algorithm described in terms of polynomials:

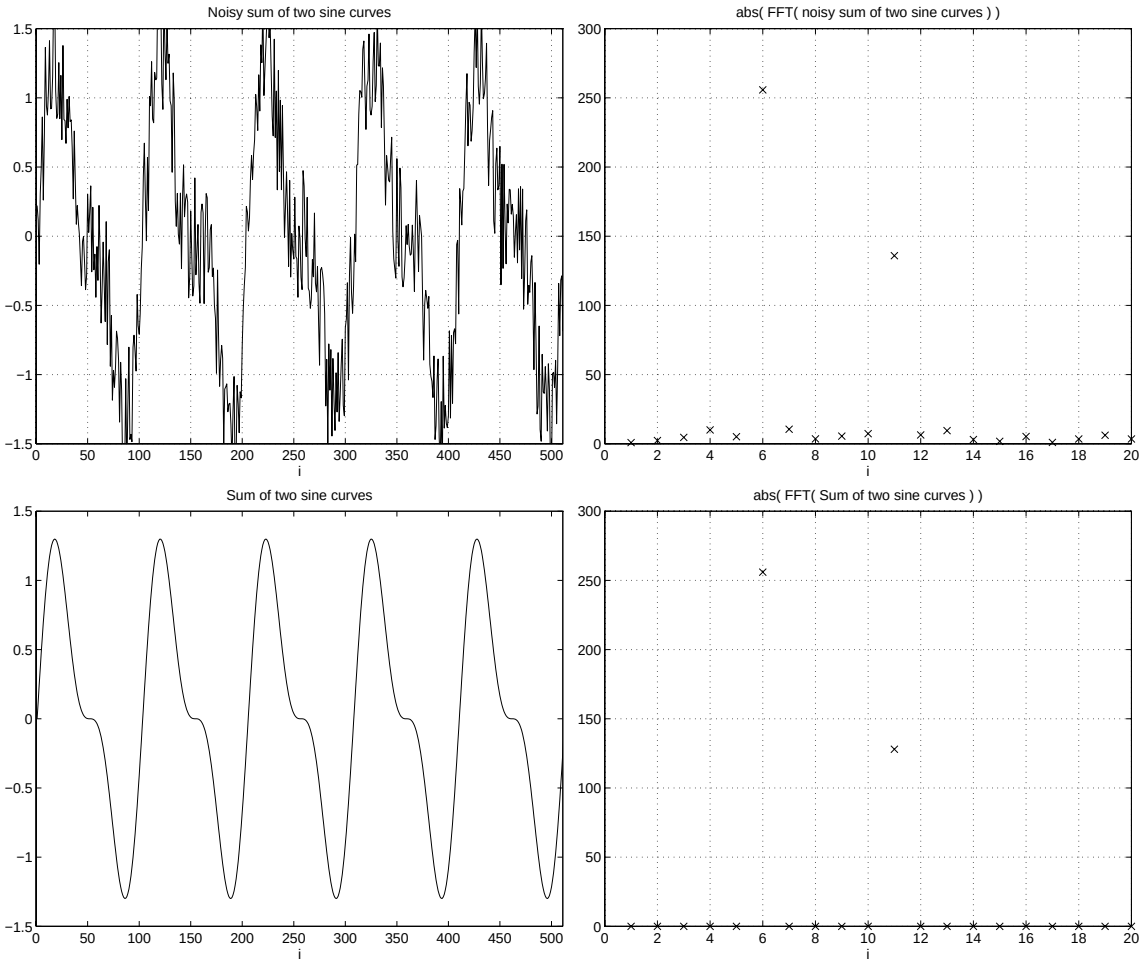
1) Evaluate $p(z)$ at $2n$ points (ω^j, $0 \le j \le 2n-1$)
2) Evaluate $q(z)$ at $2n$ points (ω^j, $0 \le j \le 2n-1$)
3) Multiply $r(\omega^j) = p(\omega^j) \cdot q(\omega^j)$, $0 \le j \le 2n$
4) Interpolate to get the coefficients of $r(z)$ from the values of $r(\omega^j)$

Using the FFT, this becomes

1) Let $p = [p_0, p_1, ..., p_{n-1}, 0, 0, ...0]$ be a vector of length $2n$
   (the n coefficients of p followed by n zeros)
   p' =FFT(p)
2) Let $q = [q_0, q_1, ..., q_{n-1}, 0, 0, ...0]$ be a vector of length $2n$
   (the n coefficients of p followed by q zeros)
   q' =FFT(q)
3) for $m = 0$ to $2n-1$
   $r'_m = p'_m \cdot q'_m$
   endfor
4) r =inverseFFT(r')

The cost of steps 1), 2) and 4) is $O(n \log n)$, the cost of 3 FFTs. The cost of step 3) is n multiplications.

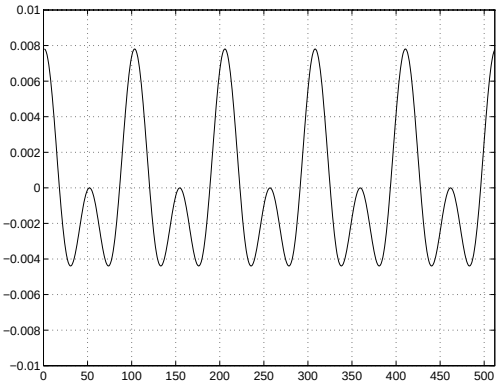
Here is how to use the FFT to do filtering. Recall our example from Lecture 13: We start with the noisy signal x' at the top left below, and wish to recover the filtered signal x below it. We do this by taking the FFT of the noisy signal $y' = F_n \cdot x'$ (top right plot), setting the tiny ("noise") components of y' to zero to get y (bottom right plot), and taking the inverse FFT to get the filtered signal $x = F_n^{-1} \cdot y$ (recall that $F_n^{-1} = \frac{1}{n}\overline{F_n}$ is the inverse FFT).



To relate y to y', let the array f' (called the *filter*) be defined by

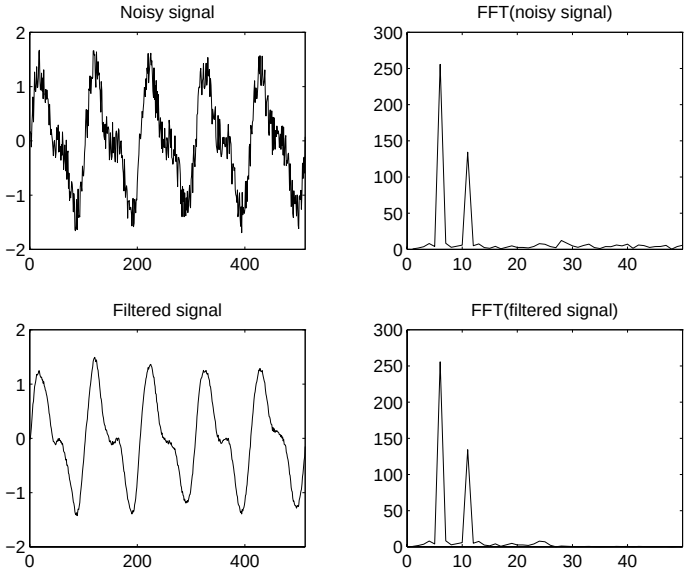
$$f_j = \begin{cases} 1 \text{ if } j \in \{6, 11, 503, 508\} \\ 0 \text{ otherwise} \end{cases}$$

Then $y_j = f_j \cdot y'_j$ for all j; this just sets all components of y to zero except 6, 11, 503 and 508 (the last two are not shown). In other words, if we let $g = F_n^{-1} \cdot f$, we see that filtering the noise out of y' is equivalent to taking the convolution of y' and g to get y. Below is a plot of g (which you do not need to form in practice):



As another example, consider the "bass" button on your stereo, which you push to make the

bass louder. This is equivalent to multiplying the FFT by a filter function which is large for low frequencies (j near 0 - and because of symmetry - j near 512) and small for high frequencies. For example, let x' be a noisy signal similar to the one above (2 pure frequencies at 440Hz and 880Hz with noise added) and f be a (very strong!) filter equal to one up to $j = 25$ (and for $j \geq 489$), equal to .1 for $27 \leq j \leq 487$, and equal to $(1 + .1)/2 = .55$ at $j = 26$ and $j = 488$ (these two values are set so that $g = F_n^{-1} \cdot f$ is real). Then the following are pictures of the noisy signal x', the FFT of the noisy signal $y' = F_n \cdot x'$ (shown up to $j = 50$), the FFT of the filtered signal y where $y_j = f_j \cdot y'_j$ (shown up to $j = 50$), and the filtered signal $x = F_n^{-1} \cdot y$.



Here we show a picture of the filter f and its inverse FFT $g = F_n^{-1} \cdot f$ (shown up to $j = 256$; the graphs are symmetric about this point):