Data Analysis in Geophysics
ESCI 7205

Bob Smalley
Room 103 in 3892 (long building), x-4929

Tu/Th - 13:00-14:30
CERI MAC (or STUDENT) LAB

Lab – 19, 10/31/13

Optimization

MATLAB

For the most part we can quickly write sloppy code and get away with it.

Sooner or later, however, you will attempt a problem that takes more time than you have available.

A simple way to get into this situation is trying to do something in "real-time".

E.G. – you want to process GPS orbits for every day, within X days, so the community of GPS researchers can process their GPS data.

If it takes you longer than one day to do the calculation, the situation is hopeless.

The hurrier you go the behinder you get.

How to fix this?

Buy a bigger, faster, computer.

The HPC is this idea on steroids.

Buy 1,000 compters!

If you are lucky (e.g. Blaine) your problem will be amenable to having more computers.

You have to do lots of independent calculations (you have to do the same thing lots of times, and they don't depend on one another. Send one calculation to each computer.)

Blaine will talk more about this on Tuesday.

Today we are going to look at what you can do in terms of programming to speed up your calculations.

As the size of the problems you want to do tends to grow faster than you can buy computers, a good algorithm can make-or-break whether or not you can do it.

A good example of this is Fourier Analysis – calculating the Fourier or inverse Fourier transform.

(actually on the computer one is calculating the "Discrete Fourier Transform", involving discrete frequencies [Fourier Series, periodic], discrete time samples [, and finite precision arithmetic]).

The basic DFT formula is

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-i2\pi kn/N}$$

Where x_n are terms in the N point long time domain time series and X_k is the k^{th} term of N terms in the frequency domain representation.

What "is" the exponential?

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-i 2\pi k n / N}$$

Basically it is $(\cos(\theta) + i \sin(\theta))$.

"straightforward" calculation of the DFT

To see what is actually going on, don't be so fancy and go back to first presentation of Fourier Series

(which actually goes the "other" way, what the inverse Fourier transform does, making the signal in the real [time, space, etc.] domain from the frequency domain representation).

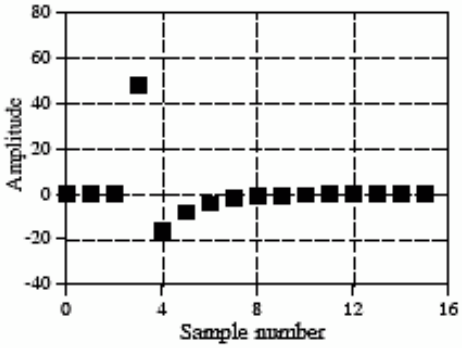
$$x_n = a_0 + \sum_{k=1}^{N-1} \left(a_k \sin\left(k \frac{2\pi}{N} n \right) + b_k \cos\left(k \frac{2\pi}{N} n \right) \right)$$

So assume for now we have the a's and b's.

Note the indices.

For each element n of our real space sequence, we are summing over index k, which represents a weighted sum of the sin or cos at k different frequencies.

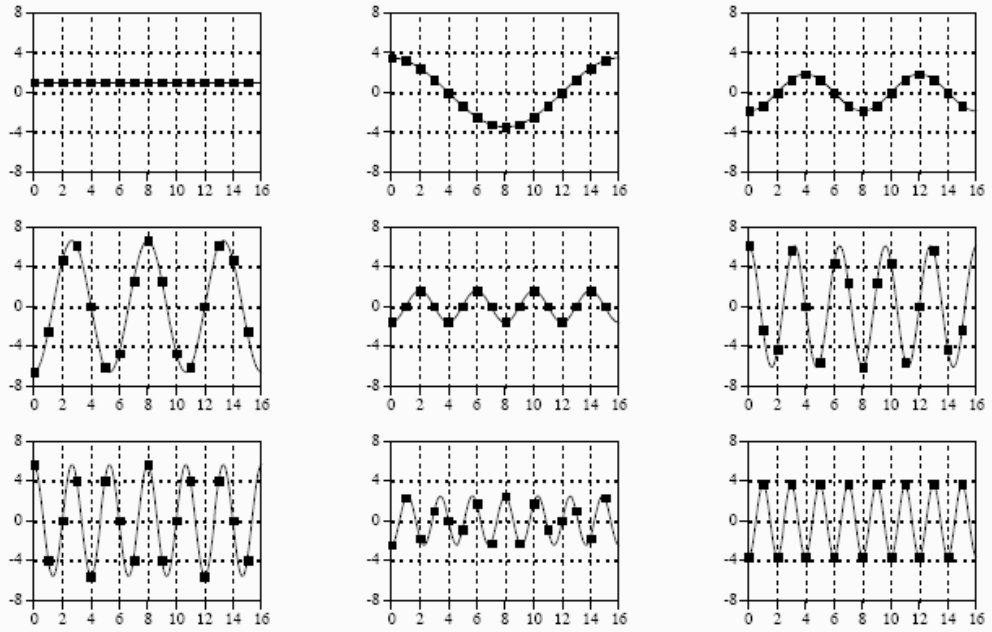
$$x_n = a_0 + \sum_{k=1}^{N-1} \left(a_k \sin\left(k \frac{2\pi}{N} n \right) + b_k \cos\left(k \frac{2\pi}{N} n \right) \right)$$



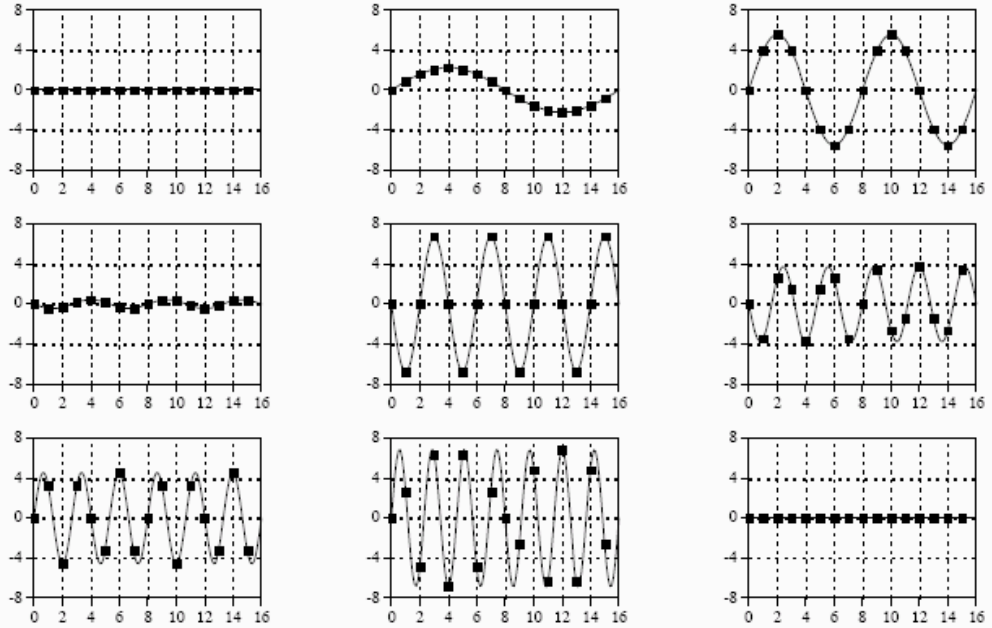
DECOMPOSE

SYNTHESIZE

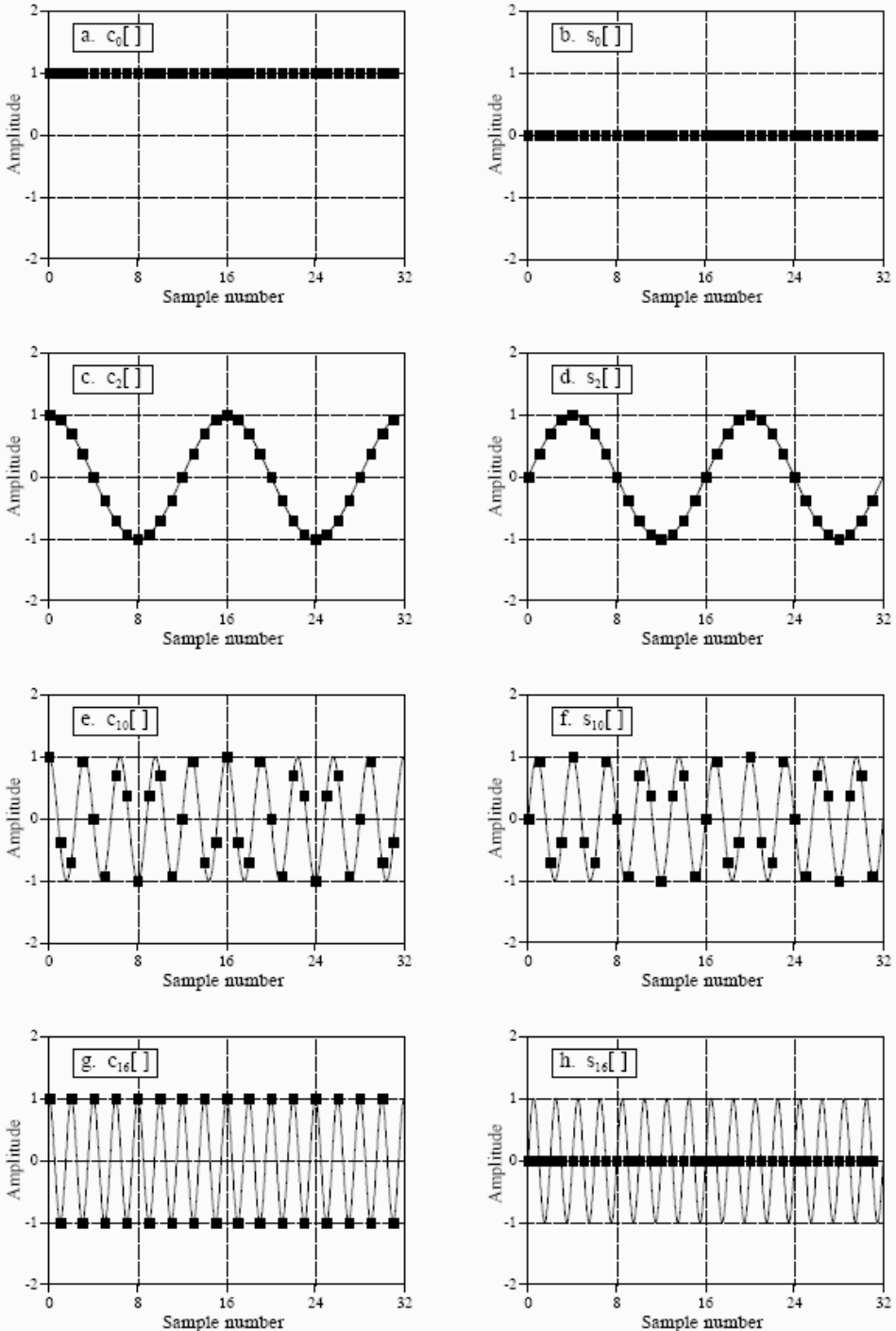
Cosine Waves



Sine Waves



The set of cos and sine terms form a set of basis functions that can be used to represent any other 16 point long sequence (function).



Returning to the equation – note that each of the trig terms in the sum is basically the dot/inner product of the sequence of the weights with a sequence of trig terms at a fixed n and varying frequency (k2π).

$$x_n = a_0 + \sum_{k=1}^{N-1} \left(a_k \sin\left(k \frac{2\pi}{N} n \right) + b_k \cos\left(k \frac{2\pi}{N} n \right) \right)$$

Next remember that the dot/inner product can be written as the multiplication of a 1xN vector times a Nx1 vector.

So if we make a matrix where each column has the k^{th} sin or cos basis vector, and multiply that by a vector with the weights, we can generate the whole sequence of the x_n.

Draw on board.

The problem with this straightforward, naïve, implementation is that as N gets larger the amount of time it takes gets even larger.

The time goes as N^2 the number of points in the sequence.

This is generally considered to be non-computable.

This "problem" can be "solved" by "parallelizing" the operations by

doing each dot product (they are independent) simultaneously on its own processor.

This is the buy a bigger computer method.

As N goes up you need N^2 dollars to do this.

Lots of effort therefore went into finding an algorithm to significantly reduce the number of calculations.

In 1965, Cooley and Tukey of Bell Labs published such an algorithm and Fourier processing has become a fundamental element of numerical methods.

The method published by C&T was originally invented by C.F. Gauss around 1805 but without computers it was not very useful.

(Same with linear algebra, which was a theoretical lagoon of math until computers came along and could evaluate 1000x1000 matrices – and as we have seen the DFT can be cast in a linear algebra form.
We now beat problems into forms amenable to linear algebra and Foruier analysis to do them efficiently on the comptuer.)

The C&T algorithm was reinvented a number of times between Gauss and C&T, but computer technology was either not ready when it was published or the inventors kept it a trade secret (read – oil companies, big commercial advantage being able to process seismic reflection data).

C&T let the cat out of the bag and changed the world.

We are not going to derive it completely, but give the outline.

The C&T algorithm, one of many types of algorithm now known to speed up the discrete Fourier transform, is of the "divide-and-conquer" type.

So how does it work?

Return to our original definition of the DFT and assume we have an even number or points (if not add a point!).

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-i2\pi kn/N}$$

Now notice (pull our of your you-know-what) that we can break this into two sums, each using every other point.

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \, e^{-i2\pi k(2n)/N} + \sum_{n=0}^{N/2-1} x_{2n+1} \, e^{-i2\pi k(2n+1)/N}$$

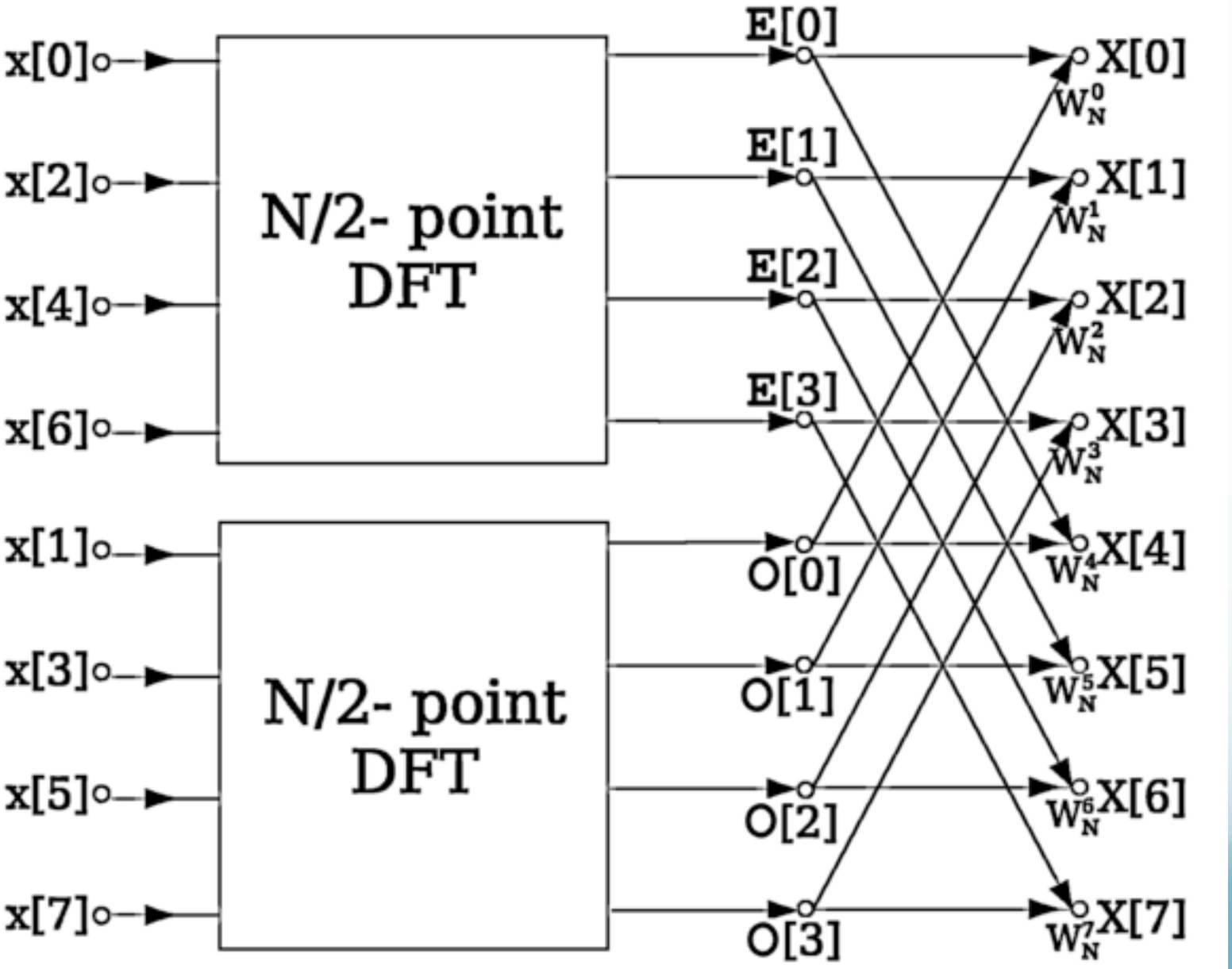
$$X_k = \sum_{n=0}^{N/2-1} x_{2n}\, e^{-i2\pi k(2n)/N} + \sum_{n=0}^{N/2-1} x_{2n+1}\, e^{-i2\pi k(2n+1)/N}$$

And factoring out the common term in the second sum (and rearranging the stuff in the exponent) we have

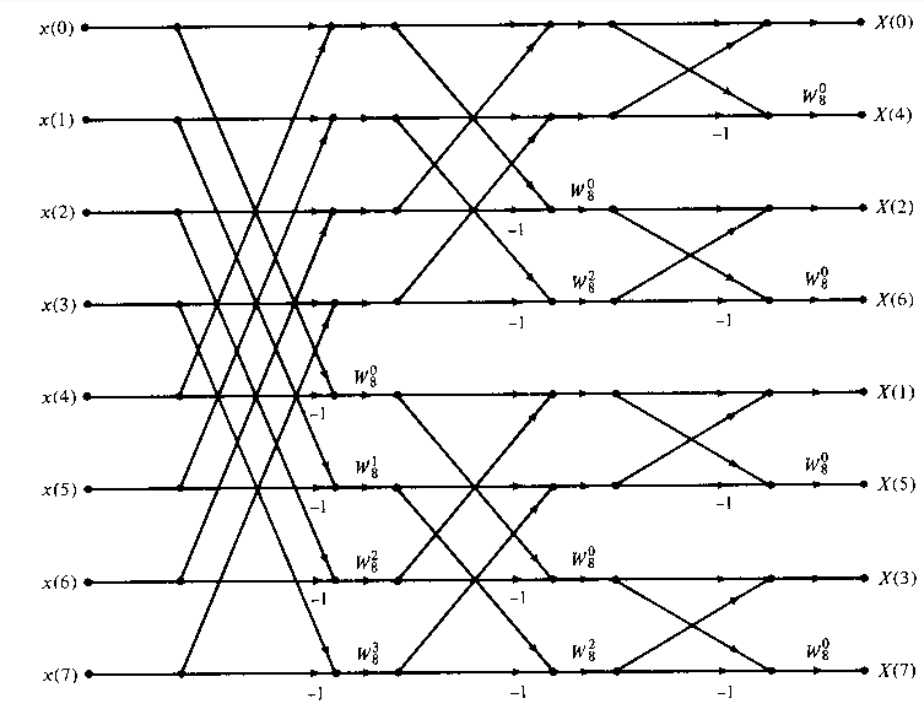
$$X_k = \sum_{n=0}^{N/2-1} x_{2n}\, e^{-ik\frac{2\pi}{N}(2n)} + e^{-ik\frac{2\pi}{N}} \sum_{n=0}^{N/2-1} x_{2n+1}\, e^{-ik\frac{2\pi}{N}(2n)}$$

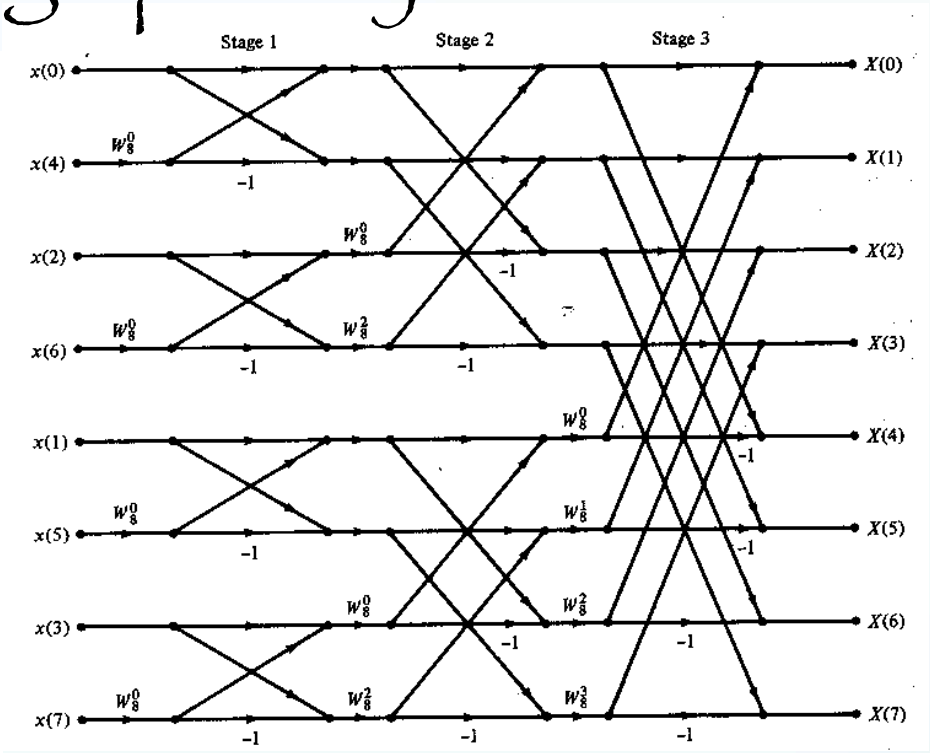
Skipping all the details, if we start out with a sequence that is a power of two long (if not – pad with zeros till it is) we can continue the above process until each sum (of which there will now be N/2) has only 1 element.

Pictorially the first step looks like

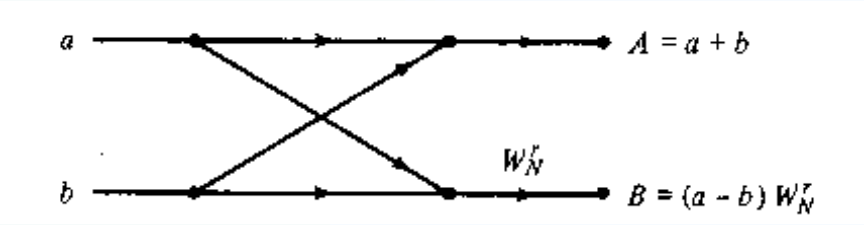


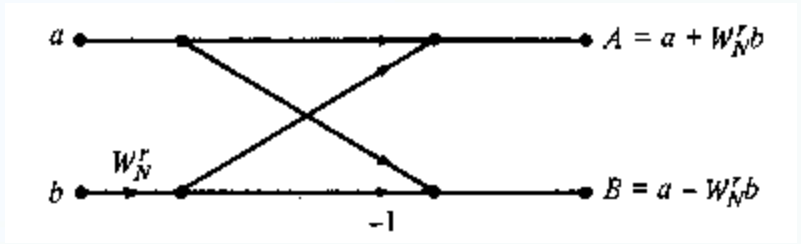
So now we just have to go backwards putting it all together – graphically





These two figures show the same set of operations, on the left the input is in order and the output is scrambled (decimation in frequency), on right the input is scrambled and the output is in order (decimation in time).





The basic operation is called a "butterfly"

Analyzing this algorithm it is found to require

$O(N \log_2 N)$ multiplies

(the arithmetic operation that counts – it is SLOW).

This is not very significant for small N, but is very important for large N.

We will now check this out by comparing the two methods.

http://blogs.mathworks.com/steve/2012/05/01/the-dft-matrix-and-computation-time/

There are lots of other methods/algorithms but the C&T one is the most widely used.

If you need to go faster, the lesson here is that you need to some thing similar – find an algorithm that will significantly reduce the number of calculations (mostly multiplies).

You also need to know how the computer language you are using "does things".

Matlab for example is optimized for matrix manipulations – so it behooves you to beat your algorithm into something that uses matrices.

This process is called vectorization, and like algorithm development and the C&T development is NOT ALGORITHMIC and requires lots of practice to develop.

General advice for speeding up your code.

http://www.matlabtips.com/optimizing-your-code/

"Singleton expansion"

e.g. when you need to apply elements of a 1xN vector to the N columns of an MxN matrix.

Example – subtract mean of each column from each column.

bsxfun: `AMRb=bsxfun(@minus,A,mean(A));`

Repmat: `AMRr=A-repmat(mean(A),size(A,1),1);`

http://stackoverflow.com/questions/12951453/in-matlab-when-is-it-optimal-to-use-bsxfun

http://blogs.mathworks.com/loren/2008/08/04/comparing-repmat-and-bsxfun-performance/