ELEC 421

Digital Signal and Image Processing

Siamak Najarian, Ph.D., P.Eng.,

Professor of Biomedical Engineering (retired),

Electrical and Computer Engineering Department,

University of British Columbia

Course Roadmap for DSP

Lecture	Title
Lecture 0	Introduction to DSP and DIP
Lecture 1	Signals
Lecture 2	Linear Time-Invariant System
Lecture 3	Convolution and its Properties
Lecture 4	The Fourier Series
Lecture 5	The Fourier Transform
Lecture 6	Frequency Response
Lecture 7	Discrete-Time Fourier Transform
Lecture 8	Introduction to the z-Transform
Lecture 9	Inverse z-Transform; Poles and Zeros
Lecture 10	The Discrete Fourier Transform
Lecture 11	Radix-2 Fast Fourier Transforms
Lecture 12	The Cooley-Tukey and Good-Thomas FFTs
Lecture 13	The Sampling Theorem
Lecture 14	Continuous-Time Filtering with Digital Systems; Upsampling and Downsampling
Lecture 15	MATLAB Implementation of Filter Design

Lecture 12:
The Cooley-Tukey and Good-Thomas FFTs

Table of Contents

Recap of radix-2 FFTs

- Last time we started to talk about how we can make the DFT more efficient. The DFT is what is under the hood in MATLAB. It is DFT that MATLAB and other digital signal processors use to make efficient Fourier transform calculations. We talked about what we can do when the DFT is a power of **2**. We spent the most time on the decimation in time algorithms, and at the end of the previous lecture, we covered the decimation in frequency algorithm.
- All those things basically mean is that, at every stage of the algorithm, we are decomposing a long DFT into **2** DFT's that are half the length. We keep on doing that until we get down to DFT's that are only **2** units and those turn out to be just adds and subtracts. If we look at the time savings (computational cost), we have $\mathbf{log_2}$ of **N** stages and **N** multiplies, which is $\mathbf{N.log_2N}$. This $\mathbf{N.log_2N}$ gives us a much more efficient algorithm than doing the DFT normally, which is an **N**-squared operation.
- What we want to talk about in this lecture is **what we can do when our DFT is non-power-of-2 DFT**. That is, if our DFT length is not **2** to the power of something. We can always zero-pad the DFT to be longer, and that is not actually a big problem for short **N**. For instance, if our original signal has a length of **11** (not a power of **2**), we can pad it with **5** zeros to create a new length of **16** (which ***is*** a power of **2**).
- If we have an **N** around **1,000**, it is not so hard to just wait for the next power of **2**. But if we want to do a length like **5** million DFT, we may need to up the lengths substantially to get to the next power of **2**. So, what we will talk about is what we can do to make that process similarly a fast process when we do not have a power of **2**.

The Cooley-Tukey FFT; Factoring N into two smaller lengths

THE COOLEY-TUKEY FFT

DFT OF LENGTH N

(1) $$X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{kn}$$ $n=0,1,\ldots N-1$, $k=0,1,\ldots N-1$, $W_N = e^{-\frac{2\pi}{N}j}$

ASSUME $N = n_1 n_2$ (2)

WRITE $n = n_1 i + j$ $\quad i = 0,1,\ldots n_2-1$, $j = 0,1,\ldots n_1-1$

$N=12,\ n_1=3,\ n_2=4$

$n = 3i + j$ $\quad i=0,1,\ldots 3$, $j=0,1,2$

$$\begin{array}{c|ccc} & j=0 & j=1 & j=2 \\ \hline i=0 & 0 & 1 & 2 \\ \cdot & 3 & 4 & 5 \\ \cdot & 6 & 7 & 8 \\ i=3 & 9 & 10 & 11 \end{array}$$

(3)

- The FFT algorithm we will be using is called the **Cooley-Tukey FFT**. Let us remember our original set up, **(1)**. Here, we have a DFT of length **N**. Our only assumption for this method is that **N** can be factored into two numbers. For example, if **N = 15**, we could choose those two numbers as **3** and **5**.
- The idea is that we are going to split up this length-**N** DFT into a whole bunch of DFT's that are of smaller length, i.e., smaller factors that make it up. So, we are going to basically assume that **N is made up of the product of two factors, N = n_1n_2**, **(2)**.
- Next, we think about how we can represent **x[n]** and **X[k]**. Instead of treating them like they are long skinny vectors, we are going to **rearrange them into 2D arrays**. We are going to write **n = n_1i + j**. Here, **i** ranges from **0** to **n_2 - 1**, and **j** ranges from **0** to **n_1 - 1**. Note that **i** and **j** are just **indices**.
- Let us suppose that we have **N = 12**. We can write that as say **n_1 = 3** and **n_2 = 4**. Then for any number **n**, we could write **n** as **n = 3i + j**, where **i** ranges from **0** to **3** and **j** ranges from **0** to **2**.
- All we are doing is basically rearranging the numbers **0** to **11**. In other words, we are rearranging the **12** numbers in a **4** by **3** grid, instead of writing them like a **12** by **1** vector. That is, instead of vector **[0 1 2 3 4 5 6 7 8 9 10 11]**, we put them in a 2D form, a **4** by **3** grid, like **(3)**. Next, we will write both the input **x[n]**, and the output **X[k]** using the same method.

Switching between 1D and 2D indexing; 2D indexing of the input and output DFT maps

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \qquad \begin{matrix} n=0,1,\dots N-1 \\ k=0,1,\dots N-1 \\ W_N = e^{-\frac{2\pi}{N}j} \end{matrix} \tag{1}$$

Input Index:

WRITE $n = n_1 i + j$ $\qquad i = 0,1,\dots n_2-1$, $\; j = 0,1,\dots n_1-1$

$N=12,\; n_1 = 3,\; n_2 = 4$

$n = 3i + j \qquad i = 0,1,..3 \quad j = 0,1,2$

$$\begin{array}{c|ccc} & j=0 & j=1 & j=2 \\ i=0 & 0 & 1 & 2 \\ \cdot & 3 & 4 & 5 \\ \cdot & 6 & 7 & 8 \\ i=3 & 9 & 10 & 11 \end{array} \tag{2}$$

Output Index:

WRITE $k = n_2 a + b$ $\qquad a = 0,1,\dots n_1-1$, $\; b = 0,1,\dots n_2-1$

$N=12,\; n_1 = 3,\; n_2 = 4$

$k = 4a + b$

$$\begin{array}{c|cccc} & b=0 & \cdot & \cdot & b=3 \\ a=0 & 0 & 1 & 2 & 3 \\ a=1 & 4 & 5 & 6 & 7 \\ a=2 & 8 & 9 & 10 & 11 \end{array} \tag{3}$$

$$X[k] = \sum_{i=0}^{n_2-1} \sum_{j=0}^{n_1-1} x[n_1 i + j] W_N^{k(n_1 i + j)} \tag{4}$$

$$X[n_2 a + b] = \sum_{i=0}^{n_2-1} \sum_{j=0}^{n_1-1} x[n_1 i + j] W_N^{(n_2 a + b)(n_1 i + j)} \tag{5}$$

- We start by our old DFT formula, **(1)**. We write **k** as **$n_2a + b$**, and we substitute for **n** and **k**, in **x[n]** and **X[k]**, using **$n = n_1i + j$** and **$k = n_2a + b$**, in **(2)** and **(3)**, respectively. Here, **$n_1i + j$** and **$n_2a + b$** are called ***double array***.
- After substituting for **n** in **(1)**, we will arrive at **(4)**, which is now a double summation. Once we substitute for **k** in **(4)**, we will have **(5)**. This whole process is called **2D indexing of the input and output DFT maps**.
- So far, all we have done is we have ***rewritten the DFT formula*** just in terms of thinking about the input as a 2D array and also thinking about the output as a 2D array.

Simplifying the formula

(*)

(1) $$X[n_2a+b] = \sum_{i=0}^{n_2-1}\sum_{j=0}^{n_1-1} x[n_1 i+j]\,\boxed{W_N^{(n_2a+b)(n_1 i+j)}}$$

(A)

(*) $$W_N^{(n_2a+b)(n_1 i+j)} = \underbrace{\boxed{W_N^{n_1 n_2 a i}}}_{1}\, W_N^{n_1 i b}\, W_N^{n_2 a j}\, W_N^{bj}$$

Complex number j

$$W_N = e^{-\frac{2\pi}{N}j} \Rightarrow W_N^{n_1} = W_{n_1 n_2}^{n_1} \longrightarrow$$

$$W_N^{n_1} = e^{-\frac{2\pi}{\cancel{n_1} n_2}j\cancel{n_1}} = e^{-\frac{2\pi}{n_2}j} = W_{n_2} \longrightarrow W_N^{n_1} = W_{n_2}$$ (2)

Index number j

$$\hookrightarrow = W_{n_2}^{ib}\, W_{n_1}^{aj}\, W_N^{bj}$$

$$\longrightarrow W_N^{(n_2a+b)(n_1 i+j)} = W_{n_2}^{ib}\, W_{n_1}^{aj}\, W_N^{bj}$$ (3)

- Let us just focus on term (*). The whole term of (A) is will be equal to **1**, since $\mathbf{W_N^N = 1}$.
- Equation (2) shows that $\mathbf{W_N^{n1} = W_{n2}}$. That is like saying that if we see $\mathbf{W_N}$ raised to one of the factors, what we get is the **W** for some other smaller number. Also, we will have $\mathbf{W_N^{n2} = W_{n1}}$. Now, we substitute $\mathbf{W_N^{n1} = W_{n2}}$ and $\mathbf{W_N^{n2} = W_{n1}}$ in (*), and we get (3).

The final formula: decomposition into smaller DFTs

(1)

$$X[n_2 a+b] = \underbrace{\left[\sum_{j=0}^{n_1-1} \underbrace{\left[\sum_{i=0}^{n_2-1} x[n_1 i+j]\, W_{n_2}^{ib}\right]}_{\substack{(*)\\ \text{FOR } j \text{ FIXED,}\\ n_2\text{-LENGTH DFT}}} \overset{(**)}{\underset{\substack{\uparrow\\ \text{TWIDDLE}\\ \text{FACTOR}}}{W_N^{bj}}}\; W_{n_1}^{aj}\right]}_{\substack{(***)\\ n_1\text{-LENGTH DFT.}}}$$

WE NEED n_1 LENGTH n_2 DFTS

n_2 LENGTH n_1 DFTS

N MULTIPLICATIONS BY TWIDDLE FACTORS.

- In **(1)**, we see that the sum **(*)**, for a fixed number **j**, is a length-$\mathbf{n_2}$ DFT. Also, **(**)** is like a twiddle factor. The outer sum, **(***)**, is like a length-$\mathbf{n_1}$ DFT.
- This is where the savings comes in. The idea is that what we are doing ***on the inside*** is that for every fixed **j**, we are doing a smaller DFT, **(*)**. For example, we are doing $\mathbf{n_1}$ length-$\mathbf{n_2}$ DFT's. Then ***on the outside*** it is like saying we are doing $\mathbf{n_2}$ length-$\mathbf{n_1}$ DFT's.
- If we put it all together, we need to do an $\mathbf{n_1}$ length-$\mathbf{n_2}$ DFT's, an $\mathbf{n_2}$ length-$\mathbf{n_1}$ DFT's, and finally, an **N** multiplication by twiddle factors. That is going to save us some time.

Computational cost

$N = n_1 n_2$

ASSUMING NAIVE SMALLER-LENGTH DFTS:

$$n_1(n_2^2) + n_2(n_1^2) + N \quad \textbf{(1)}$$

$$= N(n_1 + n_2 + 1)^{\textbf{(2)}} \quad \text{vs.} \quad N^2$$

IN GENERAL, IF $N = \prod_{i=1}^{\ell} n_i$ **(3)**

$\#$ MULTS $= N\left(\sum_{i=1}^{\ell} n_i\right)$

- Let us compute the time savings. Assuming naive small-lengths DFT's, ***we can count up what we need***. Typically, if we do the DFT in the naiveness way possible, it requires this many operations: $\mathbf{n_1.n_2^2}$-operations **+** $\mathbf{n_2.n_1^2}$-operations **+ N**, as shown in **(1)**. By factoring out **N**, we will arrive at **(2)**. Equation **(2)** is actually just an upper bound since a lot of these multiplications can be made smaller.
- Equation **(2)** is saying that instead of **N**-squared operations (**N^2**-operations), we have **N** times sum of the factors. And, that turns out to be a lot smaller.
- So, in general, if **N** is the product of a whole bunch of factors, shown by **(3)**, then the number of multiplications, **# MULTS**, is equal to **N** times the sum of the factors. This is a general form of what we talked about last time.

Computational cost

LAST TIME: IF $N = 2^{V}$

WE SHOWED RADIX-2 FFT HAD

$O(N \log_2 N)$ MULTIPLICATIONS.

$\log_2 N = V \rightarrow N \times \log_2 N = N \times v = \mathbf{O(N \times v)}$ **(1)**

THE C-T FFT IS LIKE A GENERALIZATION WITH THE SAME KIND OF EFFICIENCY (BUT N CAN BE ANYTHING)

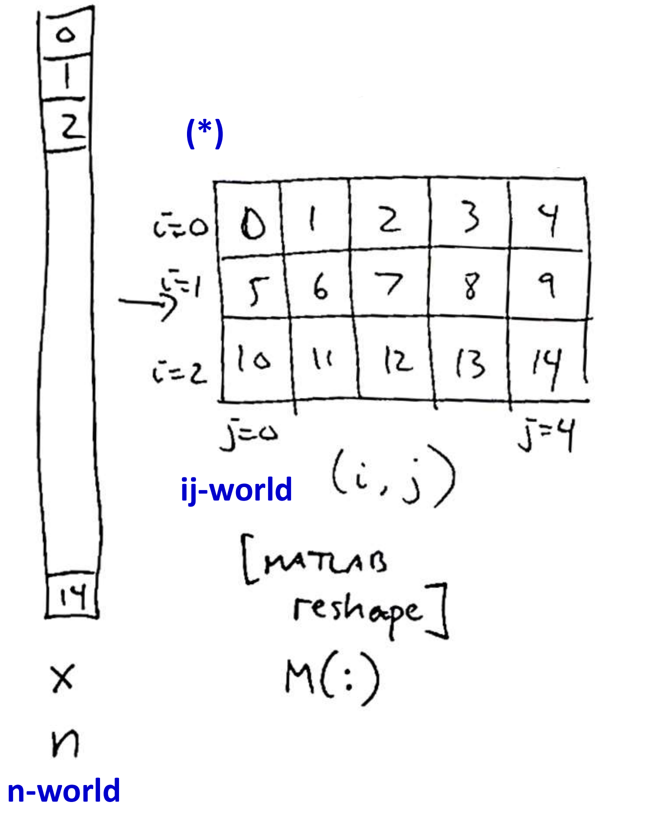
- Let us find another form of what we talked about last time. We mentioned that if **N** was **2** to some power, like **v**, or **N = 2^v^**, then we showed that the Radix-**2** FFT had approximately **$N.\log_2 N$** multiplications.
- Here, this is the same idea. In this case, **$\log_2 N$** equals **v**. This means **$N.\log_2 N = N \times v = O(N \times v)$**, as shown in **(1)**.
- **Conclusion:** We can show that the Cooley-Tukey FFT (**C-T FFT**) is like a generalization with the same kind of efficiency, but here, **N** can be anything. That is, **N** does not have to be a power of **2**.

N = 15 example, showing steps and input/output maps

$n = n_1 i + j$ $\quad i = 0, 1, .. \, n_2 - 1$
$\quad\quad\quad\quad\quad\; j = 0, 1, .. \, n_1 - 1$

$k = n_2 a + b$ $\quad a = 0, 1, .. \, n_1 - 1$
$\quad\quad\quad\quad\quad\; b = 0, 1, .. \, n_2 - 1$

$N = 15 \quad n_1 = 5, \quad n_2 = 3$



- Let us use this sketchy picture to show how it works. Suppose, we are going to have **N = 15**. Next, we factor **15** as **5×3**. We start out with the original vector **x[n]**. The first thing we do is we sort it into a **3** by **5** array. Here, what we are doing is like going from the **n-world** to the **ij-world**. In **(*)**, for the first row, **i = 0**, and for the first column, **j = 0**.
- **Now, how do we put the elements into the array?** What we do is we put them in left-to-right order. When we come to do this in MATLAB, we find out that there are some commands that are very helpful for this kind of process. Let us check out, for example, the MATLAB command **reshape**. Also, the **M(:)** command in MATLAB is used to reshape a multidimensional array **M** into a single column vector. It essentially "stacks" the elements of **M** column-wise into a long vector.
- Let us go through various steps, one at a time.

N = 15 example, showing steps and input/output maps

What does matlab command reshape do?

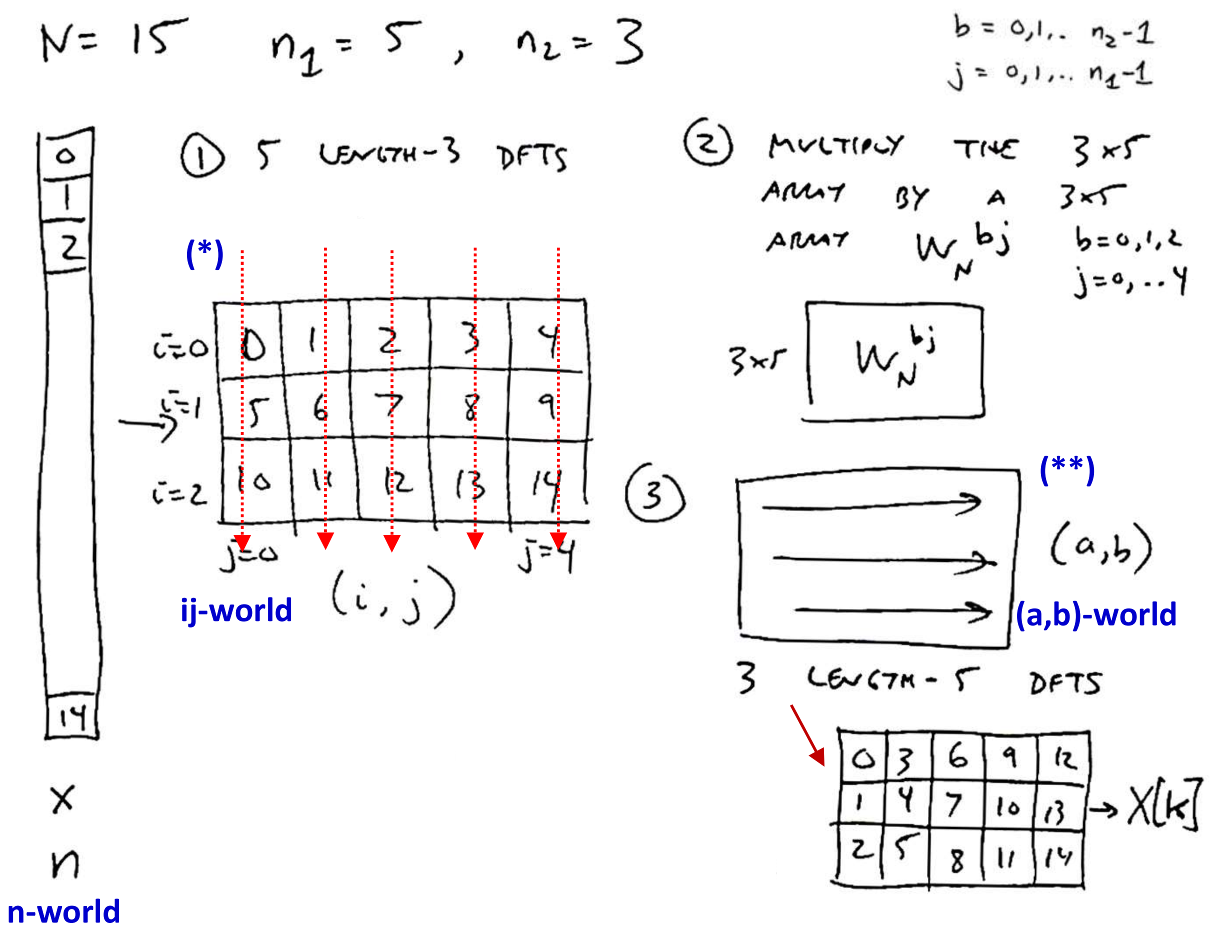
- The reshape command in MATLAB is used to **reorganize the elements of an existing array into a new shape (dimensions)**. It does not change the total number of elements in the array, but it rearranges them in memory to create a new array with a different size. Here is a breakdown of how reshape works:
- **Syntax: B = reshape(A, sz)**
  - **A:** This is the input array we want to reshape.
  - **sz:** This is a vector that specifies the new dimensions of the output array (**B**).
- **Example:**
  - **% Original array A = [1 2 3; 4 5 6]; % Reshape into a row vector B = reshape(A, 1, 6); % Reshape into a 3x2 matrix C = reshape(A, 3, 2);**
- In this example:
  - **A** is a 2x3 matrix.
  - **B** is a 1x6 row vector created by reshaping **A**.
  - **C** is a 3x2 matrix created by reshaping **A**.
- **Key Points:**
  - Reshape can be used to convert rows to columns, columns to rows, or create multi-dimensional arrays from one-dimensional arrays (and vice versa).
  - Reshape is a convenient tool for manipulating data and preparing it for various operations in MATLAB.

N = 15 example, showing steps and input/output maps

What does M(:) command do in matlab?

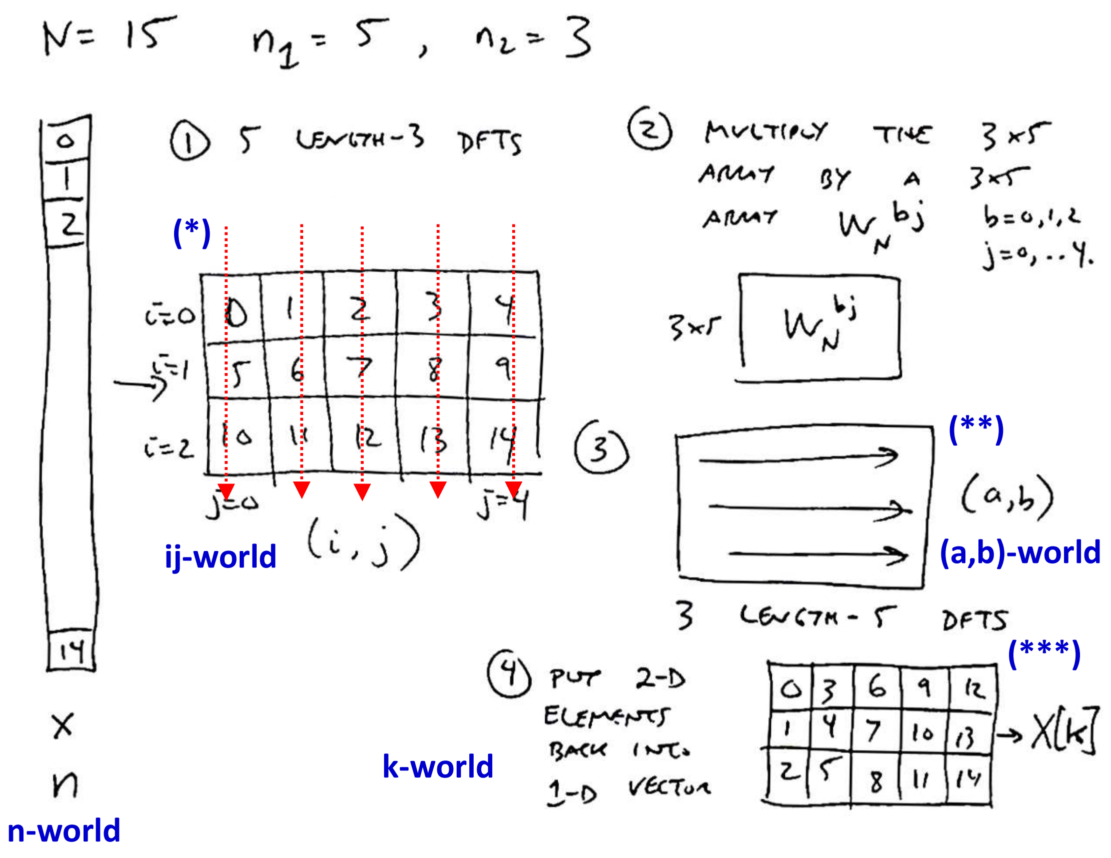
- The **M(:)** command in MATLAB is used to **reshape a multidimensional array M into a single column vector**. It essentially "stacks" the elements of **M** column-wise into a long vector. Here is a breakdown of how it works:
  - **Functionality: M(:)** treats the entire array **M** as a linear collection of elements and arranges them one after the other in a column vector.
- **Example:**
  - % Define a matrix M = [1 2 3; 4 5 6]; % Reshape into a column vector v = **M(:)**;
- In this example:
  - **M** is a 2x3 matrix.
  - **v** is a 6x1 column vector containing all the elements from **M** stacked on top of each other (following the order of columns in each row).

N = 15 example, showing steps and input/output maps



- **Step 1:** The first step is to do a whole bunch of length-**3** DFT's along the **columns**. We are going to do a DFT down the first column in **(*)**, shown by **red** dotted arrow, and then we move on to the next column and we do a DFT for that column, and so on. So, basically, step **1** is to do **5** length-**3** DFT's.
- **Step 2:** The second step is to multiply the **3** by **5** array by another **3** by **5** array given by the **twiddle factors**, $\mathbf{W_N^{bj}}$. That is like saying we take another twiddle factor matrix, that is **3** by **5**, and ***element-by-element***, we multiply the **(*)** by this twiddle factor matrix. The element-by-element multiplication in MATLAB uses the notation of "**.***".
- **Step 3:** In the third step, we are going to do **3** length-**5** DFT's along the **rows** of the result of the previous step.

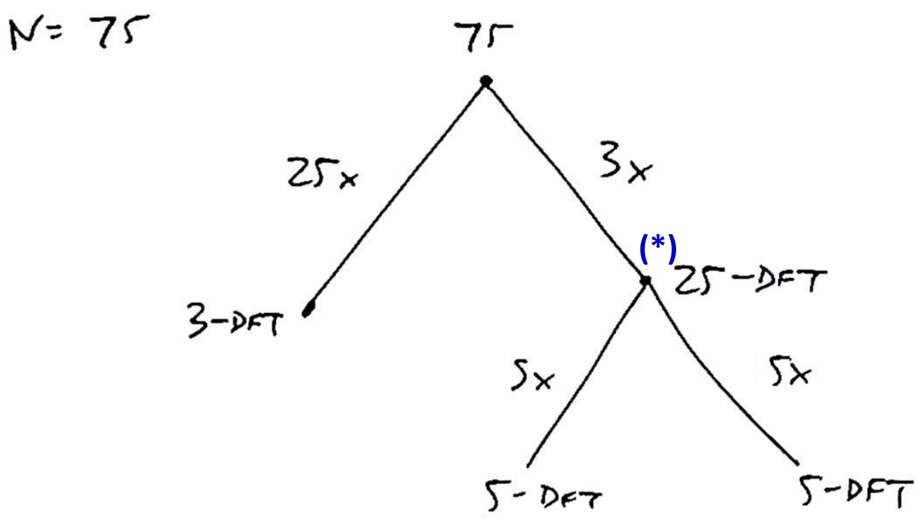
N = 15 example, showing steps and input/output maps



- **Step 4:** The last thing is to put the 2D elements back into a 1D vector. This is because **(**)** is in **(a,b)-world**. What we want is to go back into **k-world** and get **X[k]**. How do we do that? We take them out in a different order than we put them in. So, we take them out as shown in **(***)**. And, that gives us **X[k]**'s.
- **Recap:** We take the original signal, **x[n]**, we make a 2D array **(*)**, we put the elements into the 2D array in row work, we take a bunch of DFT's down the columns of **(*)**, we multiply by some special matrix, $\mathbf{W_N^{bj}}$, we take the DFT's across the rows, and we remove the elements from matrix **(***)** to get back our long vector, **X[k]**. What we just discussed is a pretty slick idea, when we look at the mechanics of it. We take the original signal, we make a matrix out of it, we do various DFT's, and finally, we take the elements out. It turns out that this process is very efficient. So, again, the idea would be that we do not have to stop here!

- **Recap for MATLAB Commands:** A handy MATLAB command for a matrix is **M(:)**, **M of column**. Here, if we have a matrix and we do **M(:)** of column, that is like saying we take the elements out of the matrix and put them back into a long skinny vector. Both **reshape** and **M(:)** operators are very useful in this process.

Choices in factoring composite N



```
>> factor(75)

ans =

     3     5     5
```

- As an example, let us say we have **N = 75**. We want to do a **75**-point DFT. We could start by decomposing that into **25 3** DFT's, and **3 25** DFT's. But every time we have to do that **25** DFT, **(*)**, we could further decompose that into **5** length-**5** DFT's and another **5** length-**5** DFT's. So, what happens is eventually we break the DFT down into just its prime factors.
- Computationally, this is quite neat. There has been a lot research in this area, and so, we do not have to figure out how to make super efficient short DFT's. The most efficient length-**3** DFT (or length-**2**) are all hardwired into MATLAB and into a DSP chips. In MALTAB, **factor(75)** will give us the prime factors of **75**. Note that a **Digital Signal Processor Chip**, also known as a **Digital Signal Processing Unit**, is a specialized microprocessor designed to efficiently handle and manipulate digital signals.
- **Conclusion:** When we get down to these really short DFT's, we are getting the most efficient implementation of the DFT that is possible. The benefit comes from the fact that the longer DFT's have been broken down into those factors.

The Good-Thomas FFT; The greatest common divisor

HOW TO GET RID OF TWIDDLE FACTORS?

GOOD-THOMAS FFT.

IF a AND b ARE INTEGERS, NOT BOTH 0, THEN THE GREATEST COMMON DIVISOR $(a,b) = \gcd(a,b)$ EXISTS.

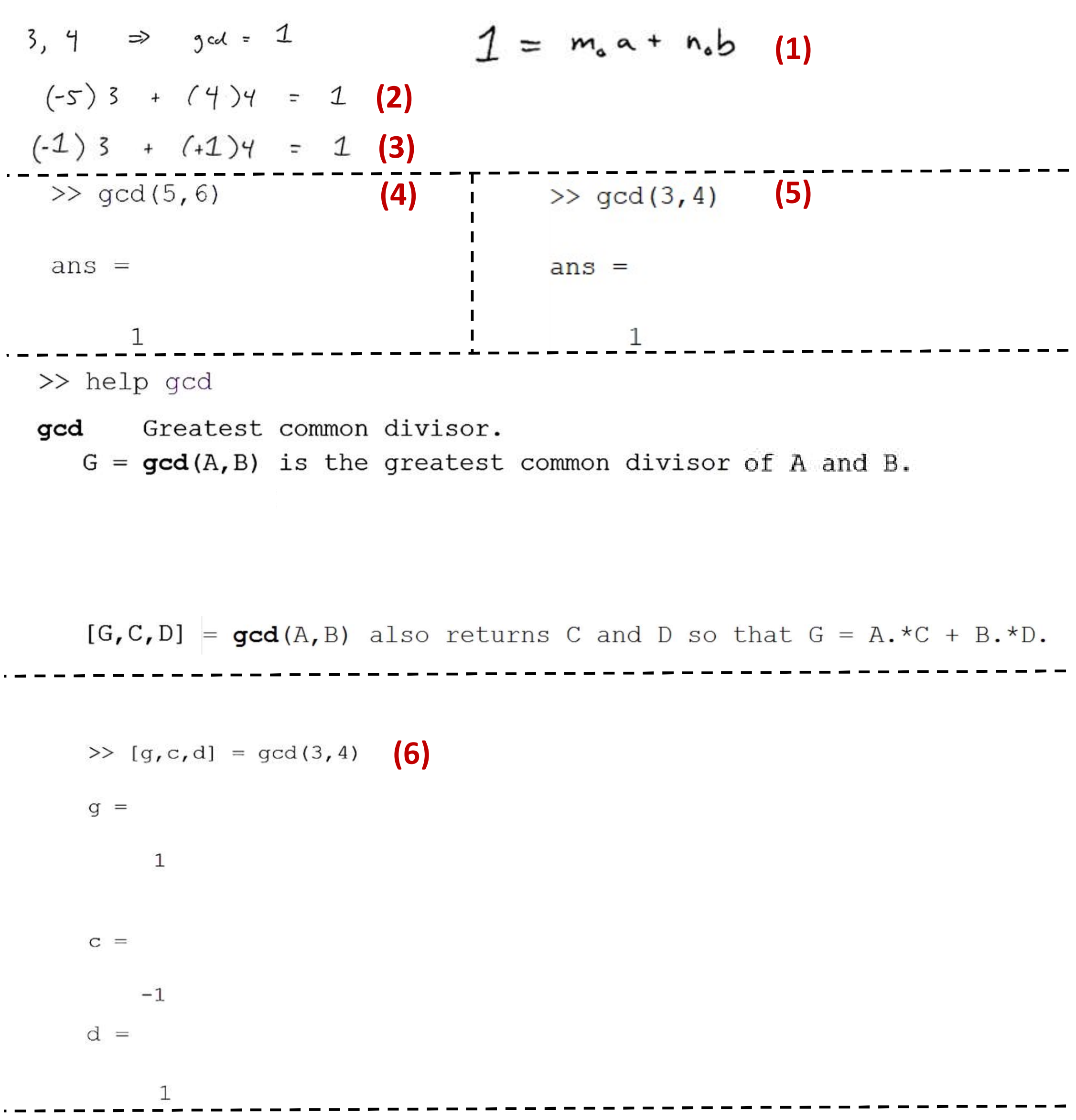
THERE ARE INTEGERS m_0 AND n_0 SO THAT $(a,b) = m_0 a + n_0 b$.

IF $(a,b) = 1$ a AND b ARE RELATIVELY PRIME, AND THERE EXIT m_0, n_0 SO THAT

$$1 = m_0 a + n_0 b$$

- Next, we want to talk about a neat ***extension of the Cooley-Tukey FFT***. We noticed that the **twiddle factors** did not add a large amount of computation to the process. But still would not it be nice if we could just take the input, put it into this matrix form, do the DFT's, and take the result out? That is, no twiddling, just straight out DFT along every dimension. It would be very satisfying to be able to do that!
- Actually, there is a way to do that. That is what we are going to talk about next. So, this **advanced DFT** is called the **Good-Thomas FFT**. So, **how to get rid of the twiddle factors?** This relies on some results from abstract math. The relevant topics are **Number Theory** and **Abstract Algebra**.
- Here, if **a** and **b** are integers, not both **0**, then we have something that is called the **Greatest Common Devisor (GCD)**. That is, if both numbers are non-zero integers, then GCD exists. The notation for GCD is "parentheses **a** comma **b**", **(a,b)**, **gcd(a,b)**, or just **GCD**. Also, there are some integers, let us call them **m_0** and **n_0**, so that the greatest common divisor can be written as **GCD = (a,b) = m_0a + n_0b**. If **(a,b) = 1**, then **a** and **b** are ***relatively prime***, and there exit **m_0** and **n_0**, so that **1 = m_0a + n_0b**.

Relatively prime factors; MATLAB's gcd function



3, 4 $\Rightarrow$ gcd = 1

$1 = m_0 a + n_0 b$ (1)

$(-5)3 + (4)4 = 1$ (2)

$(-1)3 + (+1)4 = 1$ (3)

```
>> gcd(5,6)

ans =

     1
```
(4)

```
>> gcd(3,4)

ans =

     1
```
(5)

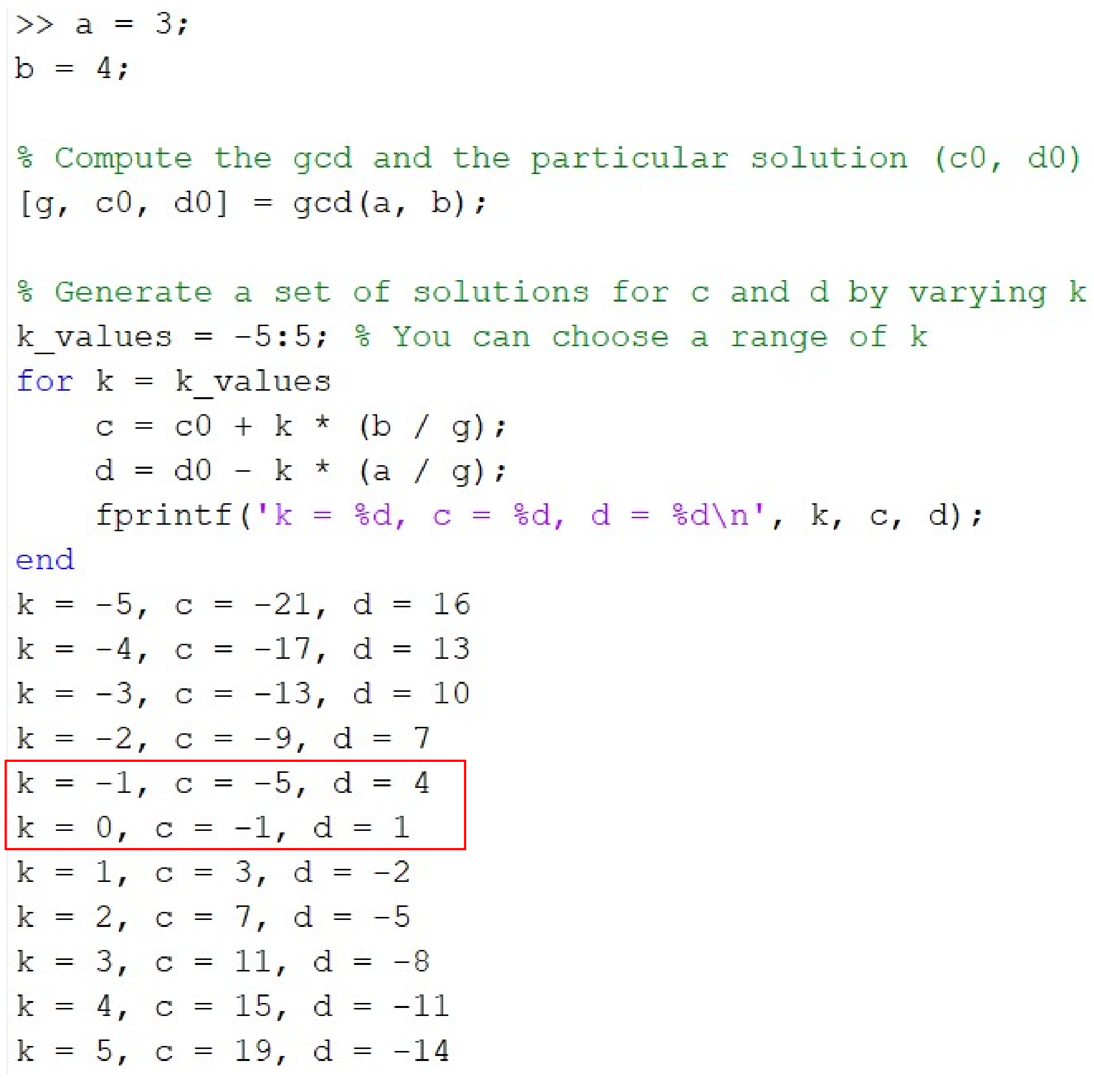
```
>> help gcd
gcd    Greatest common divisor.
   G = gcd(A,B) is the greatest common divisor of A and B.



   [G,C,D] = gcd(A,B) also returns C and D so that G = A.*C + B.*D.
```

```
>> [g,c,d] = gcd(3,4)
```
(6)

```
g =

     1


c =

    -1

d =

     1
```

- For example, let us take **3** and **4**. We know that **3** is prime but **4** is not prime itself. It is **2^2**. But **3** and **4** are **relatively prime** (or **coprime**) because they do not have any common factors. So, **3** and **4** have **GCD = 1**. The other part of the theorem is telling us that we should be able to find some other integers, and these integers are such that equation **(1)** is true. By eyeballing it, we can say that we want something, such as **m_0**, times **3** plus something else, such as **n_0**, times **4** to be equal to **1**. If we take **(-5)×(3) + (4)×(4)**, we get **1**, as shown in **(2)**. So, we found the missing integers.
- If we look in MATLAB, there is a GCD function that takes care of this operation. It is **[G,C,D] = gcd(A,B)**. If we do GCD of **5** and **6**, for example, it tells us that these two do not have any factors in common, i.e., **GCD = 1**, **(4)**. In MATLAB, if we look at **3** and **4**, the **GCD = 1**, **(5)**. Here, we get **(-1)(3) + (+1)(4) = 1** by eyeballing, as shown in **(3)**, or **(6)** using MATLAB.
- **Conclusion:** The multiple integers are not unique. So, there is an infinite set of these **m_0** and **n_0** that would make this work.

Relatively prime factors; MATLAB's gcd function



```
>> a = 3;
b = 4;

% Compute the gcd and the particular solution (c0, d0)
[g, c0, d0] = gcd(a, b);

% Generate a set of solutions for c and d by varying k
k_values = -5:5; % You can choose a range of k
for k = k_values
    c = c0 + k * (b / g);
    d = d0 - k * (a / g);
    fprintf('k = %d, c = %d, d = %d\n', k, c, d);
end
k = -5, c = -21, d = 16
k = -4, c = -17, d = 13
k = -3, c = -13, d = 10
k = -2, c = -9, d = 7
k = -1, c = -5, d = 4
k = 0, c = -1, d = 1
k = 1, c = 3, d = -2
k = 2, c = 7, d = -5
k = 3, c = 11, d = -8
k = 4, c = 15, d = -11
k = 5, c = 19, d = -14
```

- In MATLAB, the function **gcd(a,b)** returns the greatest common divisor **g** of two numbers **a** and **b**, along with the coefficients **c** and **d** such that:

$$g = a \cdot c + b \cdot d$$

- These coefficients **c** and **d** are particular solutions to the equation, but the general solution for **c** and **d** is given by:

$$c = c_0 + k \cdot \frac{b}{g}$$

$$d = d_0 - k \cdot \frac{a}{g}$$

- Where $\mathbf{c_0}$ and $\mathbf{d_0}$ are the coefficients returned by MATLAB's **gcd** function, **g** is the greatest common divisor, and **k** is any integer.

- To find all possible values of **c** and **d**, we can use the following approach:
  - ➢ Use **gcd** to get the initial values of **c** and **d**.
  - ➢ Generate new pairs of **c** and **d** by using the formulae above for different values of **k**.

2D indexing for Good-Thomas

$$X[k_1][k_2] = \underbrace{\sum_{j=0}^{n_2-1}\Bigg(\underbrace{\sum_{i=0}^{n_1-1} x[i][j]\, W_{n_1}^{ik_1}}_{\text{LENGTH } n_1\text{- DFTS}}\Bigg)}_{\text{LENGTH } n_2\text{-DFTS}} W_{n_2}^{jk_2} \quad (1)$$

NO "TWIDDLE FACTORS"

Good-Thomas formula

- Although beyond the scope of this course, after some lengthy substitution, and by using the theory of GCD and mod, along with further mathematical manipulations, we arrive at **(1)**.
- Now, in the inside, we have length-$\mathbf{n_1}$ DFT'S and on the outside, we have length-$\mathbf{n_2}$ DFT's. The final equation is clean, in the sense that, now, there are no twiddle factors to deal with.

End of Lecture 12