EECS 16B Designing Information Devices and Systems II
Fall 2019

Note: DFT

Overview

We have seen how polynomial interpolation can be used to fit an $(m-1)$-degree polynomial to any set of m points with distinct x-coordinates. Often, however, we are not given these x-coordinates explicitly. For instance, we may be given a sequence of observations $y_1, y_2, \ldots, y_m$, told that they are values sampled at a uniform rate from some sensor, and asked to interpolate between them. The meta-data from our sensor is always present in terms of when each sample was actually taken, but we have considerable flexibility in how we use it.

In this note, we will look at various natural ways of doing this interpolation and see how each one falls short. More important than the specific ways that each falls short is how confronting each of these obstacles helps us open our mind more truly to the possibilities and helps us design ways that overcome the obstacles that we faced. This will ultimately lead us to a family of ideas called the Discrete Fourier Transform via this interpolation context.

1 Problems with real-valued polynomials

One natural way of doing interpolation is to write our observations as

$$(0, y_1), (\Delta, y_2), (2\Delta, y_3), \ldots, ((m-1)\Delta, y_{m-1}),$$

where Δ is the time between observations, and then use polynomial interpolation to express y as a continuous function of time.

We know that polynomial interpolation uses a parametric function

$$g(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_{m-1} x^{m-1},$$

that is a linear combination of the monomials $x^0, x^1, x^2, \ldots, x^{m-1}$. The α_i are the parameters that define the function. We proved last time that polynomial interpolation will always work, meaning that $g(x)$ will pass through all our sample points exactly.

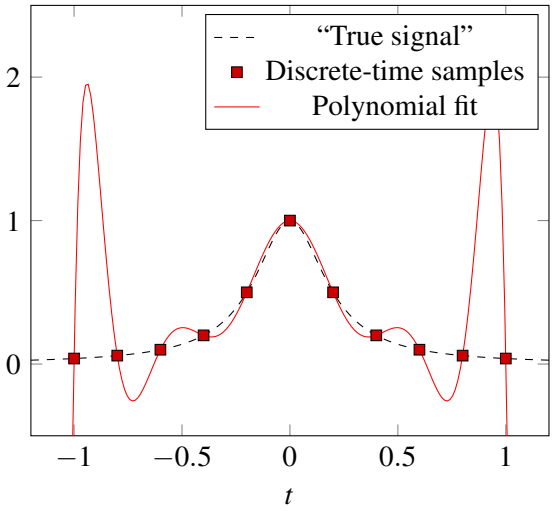
So what's more to do? As it turns out, although polynomial interpolation always produces *an* interpolation, it is not always a particularly desirable one. Let's try to see why. Imagine that we had a reasonably large number of samples - say, $m = 100$. Then our monomial basis will consist of the functions

$$x^0, x^1, x^2, \ldots, x^{99}.$$

The problem here is that x^{99} is really not a very nice function to work with over the reals. When x is just slightly less than 1, x^{99} quickly drops to zero. Try $(0.9)^{99} \approx 0.00003$ for example. Similarly, when x is just slightly greater than 1, the x^{99} starts blowing up rapidly. Try $(1.1)^{99} \approx 12500$. That is a huge dynamic range over a span of just 0.2. The consequence of this is that our coefficients α_i similarly become very small or very large in order to control these high-degree monomials, which is undesirable. The effect is that our

polynomial interpolations quickly become unreasonable - they may *technically* pass through all the sample points, but act "unreasonably" everywhere else.

Numbers don't have to get big before you start seeing this effect. For example, consider the following polynomial fit of $m = 11$ points:



Eleven is not that big of a number, and this is already starting to be pretty nasty in terms of how it is behaving. The fit turned out to be[1] $-220.94174208145742 * x^{10} + 494.90950226246065 * x^8 + -381.43382352942115 * x^6 + 123.35972850678912 * x^4 + -16.85520361990966 * x^2 + 1$ and the magnitude of those coefficients are getting big for the leading terms. In turn, these large coefficients are inducing oscillations present that did not exist in the original dataset.

Clearly, a better method of interpolation is needed for global approximation.

2 Escaping the desert of the reals: polynomials over the unit circle

Fundamentally, the problems we have faced with polynomial interpolation come down to the poor behavior of high-degree monomials away from $|x| = 1$, since their magnitudes vary enormously. We actually understand this very well since we have already seen the concept of stability for discrete-time systems. The reason that eigenvalues with magnitude less than 1 are stable is that raising them to high powers results in values close to zero, and hence the influence of disturbances a long time ago decays away. Similarly, eigenvalues with magnitudes greater than 1 cause violent instability because raising them to high powers blows up, and hence even small disturbances compound to overwhelm us. However, there are only two real numbers with absolute value equal to 1, so, with a large number of samples, we clearly cannot all place them such that $|x_i| = 1$.

Or can we? If we allow ourselves to use complex numbers, then we have infinitely many values z such that $|z| = 1$. We know that any complex number of the form $e^{j\theta}$, where θ is real, lies on the unit circle on the complex plane. We know from our studies of stability that the unit circle is special[2] — large powers neither

[1] Why are there only even terms here? Because all the odd terms are zero because the function we are interpolating is symmetric about the origin $f(x) = f(-x)$.

[2] It turns out that this property of not blowing up or shrinking to zero also occurs elsewhere — in finite fields. You will learn about finite fields in our successor course 70, and can learn even more about them in abstract algebra courses like Math 113 and Math 114. The fact that one can take large powers in finite fields without problems of instability turns out to be very useful

blow up nor go to zero. So, given a sequence $y_1, y_2, \ldots, y_m$, we may choose our x-coordinates to be such that our points become

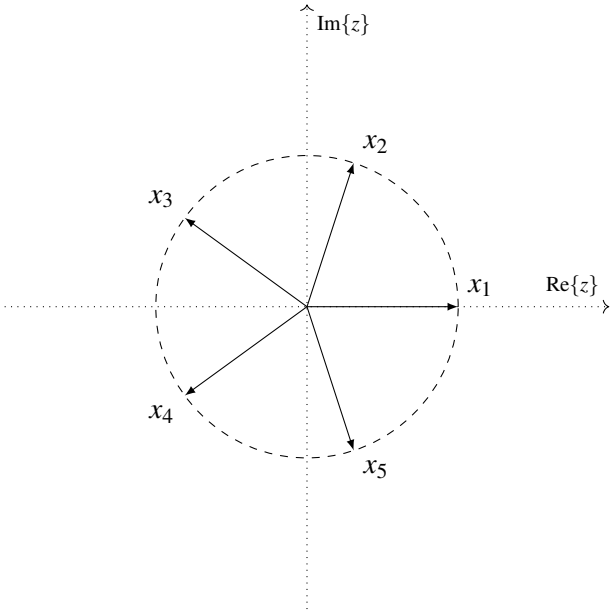
$$(1, y_1), (e^{2\pi j/m}, y_2), (e^{2\cdot 2\pi j/m}, y_3), \ldots, (e^{(m-1)\cdot 2\pi j/m}, y_{m-1}).$$

In other words, we let

$$x_i = (e^{j\frac{2\pi}{m}})^{i-1} = \omega_m^{i-1}, \quad \forall i \in \{1, 2, 3, \ldots, m\}$$

where $\omega_m = e^{j\frac{2\pi}{m}}$ is the mth primitive root of unity. ⇒ Unit circle separated into m sectors

We plot the x_i below on the complex plane, in the case of $m = 5$. Notice that they are all distinct, of magnitude 1, and equally spaced around the complex unit circle.



We can see clearly that the k-th power of $e^{j\theta}$ is $e^{jk\theta} = \cos(k\theta) + j\sin(k\theta)$ which always has a real part and imaginary part bounded by 1. Notice also the qualitative connection with what we traditionally associate with high degree polynomials — higher degrees are more "wiggly" than lower degrees. We get all these nice features without any of the pain of either blowing up or shrinking to zero. So this problem related to stability has been entirely exorcised.

3 Interpolating using complex numbers

Now, we can perform polynomial interpolation as usual, using these new, complex, values of x_i. Our previous results on the linear independence of the columns of a Vandermonde matrix still apply, since we never assumed there that our values were real. Thus, if our parametric function looks like

$$g(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_{m-1} x^{m-1}$$

for communication-type applications like error-correcting codes and cryptography. This is also why Fourier-type analysis is also practiced in those areas.

with the coefficients/parameters stacked into a vector $\vec{\alpha}$, then we have

$$B\vec{\alpha} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{m-1} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_m & x_m^2 & \cdots & x_m^{m-1} \end{bmatrix} \vec{\alpha} = \vec{y}.$$

Note that the matrix B is the same as the matrix B in the interpolation note, and the $\vec{y}$ are the samples.

Now, substituting in our chosen values for the x_i, our equation becomes

$$\begin{bmatrix} \omega_m^{0\cdot 0} & \omega_m^{0\cdot 1} & \omega_m^{0\cdot 2} & \cdots & \omega_m^{0\cdot(m-1)} \\ \omega_m^{1\cdot 0} & \omega_m^{1\cdot 1} & \omega_m^{1\cdot 2} & \cdots & \omega_m^{1\cdot(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_m^{(m-1)\cdot 0} & \omega_m^{(m-1)\cdot 1} & \omega_m^{(m-1)\cdot 2} & \cdots & \omega_m^{(m-1)\cdot(m-1)} \end{bmatrix} \vec{\alpha} = \vec{y}.$$

When working with real numbers, the k-th column of our matrix B was composed of the stacked evaluations of the monomials x^k at $x_1, x_2, \ldots, x_m$. The same is true here, where now, because of our choice of x_i, they are the stacked evaluations of the monomials x^k at $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m-1}$. As they are linearly independent, they continue to form a basis, so we can make any $\vec{y}$ as a linear combination of the columns of B.

It turns out that this particular $B = [\vec{b}_0, \vec{b}_1, \cdots, \vec{b}_{m-1}]$ matrix has a couple of nice properties that will be convenient for us later. First, notice that it is symmetric, by construction. Furthermore, we can compute the (squared) norm of each column (or row!) to be

$$\begin{aligned} \|\vec{b}_i\|^2 &= \langle \vec{b}_k, \vec{b}_k \rangle \\ &= \vec{b}_k^* \vec{b}_k \\ &= \sum_{i=0}^{m-1} \overline{\omega_m^{k\cdot i}} \cdot \omega_m^{k\cdot i} \\ &= \sum_{i=0}^{m-1} \omega_m^{-k\cdot i} \cdot \omega_m^{k\cdot i} \\ &= \sum_{i=0}^{m-1} 1 \\ &= m. \end{aligned}$$

Notice that, since $\vec{b}_k$ is a complex vector, we must take care to use the complex inner product when computing its norm. This involves a conjugate transpose instead of a transpose as in the real inner product. This result tells us that all the rows and columns of our B matrix have the same norm $\sqrt{m}$.

Furthermore, consider the inner product of any two *different* columns, $\vec{b}_a$ and $\vec{b}_b$. We see that

$$\begin{aligned}\langle \vec{b}_a, \vec{b}_b \rangle &= \vec{b}_b^* \vec{b}_a \\ &= \sum_{i=0}^{m-1} \overline{\omega_m^{b \cdot i}} \omega_m^{a \cdot i} \\ &= \sum_{i=0}^{m-1} \omega_m^{(a-b) \cdot i}.\end{aligned}$$

This is a finite geometric series with m terms and common ratio ω_m^{a-b}. Since this common ratio is not equal to 1, we can use the known formula for the sum of finite geometric series, to see that

$$\begin{aligned}\langle \vec{b}_a, \vec{b}_b \rangle &= \frac{\left(\omega_m^{a-b}\right)^m - 1}{\omega_m^{a-b} - 1} \\ &= \frac{\left(\omega_m^m\right)^{a-b} - 1}{\omega_m^{a-b} - 1}\end{aligned}$$

But since ω_m is an mth root of unity, $\omega_m^m = 1$. Thus,

$$\langle \vec{b}_a, \vec{b}_b \rangle = \frac{1^{a-b} - 1}{\omega_m^{a-b} - 1} = 0.$$

In other words, all the columns of B are mutually orthogonal. We can go a step further! Since all the columns have the same norm $\sqrt{m}$, we can normalize each column to obtain

$$U = \frac{1}{\sqrt{m}} B,$$

which has orthogonal columns of unit norm, and so is a complex orthonormal matrix!

Recall that we started with the equation $B\vec{\alpha} = \vec{y}$. We were searching for the coefficients α_i. Since B is known to be a square matrix of full rank, we can pre-multiply by its inverse to obtain

$$\vec{\alpha} = B^{-1} \vec{y}.$$

But now, since U is a square orthonormal matrix, we know that its inverse equals its conjugate transpose, $U^{-1} = U^*$. Thus, we see that the inverse of B is

$$B^{-1} = (\sqrt{m}U)^{-1} = (\sqrt{m})^{-1} U^{-1} = \frac{1}{\sqrt{m}} U^* = \frac{1}{\sqrt{m}} \left(\frac{1}{\sqrt{m}} B \right)^* = \frac{1}{m} B^*.$$

Substituting back, we find that the coefficients of our desired polynomial interpolation are

$$\vec{\alpha} = \frac{1}{m} B^* \vec{y}.$$

The $\vec{\alpha}$ are called the (polynomial-style) DFT coefficients that correspond to the vector $\vec{y}$. Often, an upper-case letter is used for them so $\vec{Y} = \vec{\alpha}$ and satisfies $\vec{y} = B\vec{Y}$. Equivalently, $\vec{Y} = \frac{1}{m} B^* \vec{y}$ and hence $Y[i] = \frac{1}{m} \vec{b}_i^* \vec{y}$.

Notice that B^* is a matrix with terms all of the same magnitude (1), and so (assuming that all the elements of $\vec{y}$ are about the same size) it is intuitive to expect that the DFT coefficients $(1/m)B^*\vec{y}$ is a vector whose

elements are within the same order of magnitude. Thus, by switching to samples taken on the unit circle over the complex plane, it seems that we have resolved the problem we had with our real polynomial interpolations, where the coefficients varied hugely in their magnitudes because the basis functions were poorly behaved over the points of interest.

4 Problems with our interpolation

Now that we have all this machinery available to us, let's try a simple example. Let our observations be

$$\vec{y} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\top,$$

so $m = 3$.

Substituting, our B matrix becomes

$$B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & e^{j2\pi/3} & e^{j4\pi/3} \\ 1 & e^{j4\pi/3} & e^{j2\pi/3} \end{bmatrix}.$$

Thus, we can solve for the coefficients of our polynomial interpolation, to obtain

$$\begin{aligned} \vec{\alpha} &= \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & e^{2j\pi/3} & e^{4j\pi/3} \\ 1 & e^{4j\pi/3} & e^{2j\pi/3} \end{bmatrix}^* \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\ &= \frac{1}{3} \begin{bmatrix} 1 \\ e^{-2j\pi/3} \\ e^{-4j\pi/3} \end{bmatrix}. \end{aligned}$$

So our polynomial interpolation is

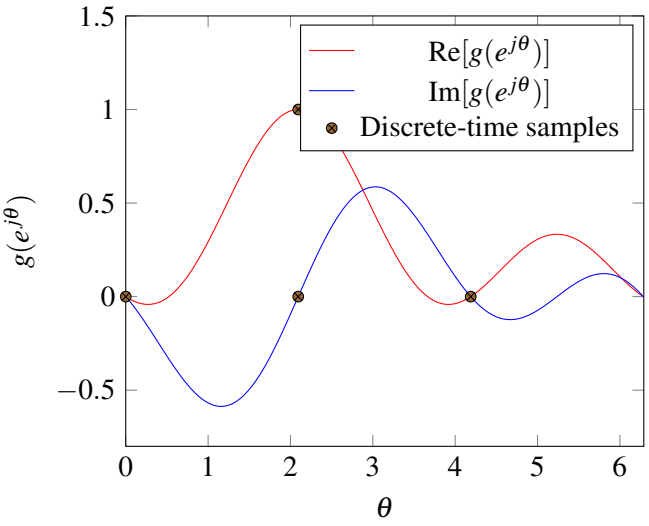
$$g(x) = \frac{1}{3} + \frac{e^{-2j\pi/3}}{3} x + \frac{e^{-4j\pi/3}}{3} x^2.$$

We can verify that $g(x)$ passes through our three observations. So are we done now?

Well, we really want to interpolate between our samples. Since our samples are located at $\omega_3^i = e^{j\frac{2\pi}{3}i}$ for $i \in \{0,1,2\}$, it makes sense to look at how $g(x)$ behaves at $e^{j\theta}$ for all $\theta \in [0, 2\pi)$, in order to obtain an equivalent to our previous real-valued polynomial interpolations. Here, our samples are interpreted as being at $\theta = 0, \frac{2\pi}{3}, \frac{4\pi}{3}$. Doing so, we obtain

$$\begin{aligned} g(e^{j\theta}) &= \frac{1}{3}\left(1 + e^{-2j\pi/3} e^{j\theta} + e^{-4j\pi/3} e^{j2\theta}\right) \\ &= \frac{1}{3}\left(1 + \cos(\theta - 2\pi/3) + \cos(2\theta - 4\pi/3)\right) + \frac{j}{3}\left(\sin(\theta - 2\pi/3) + \sin(2\theta - 4\pi/3)\right). \end{aligned}$$

Plotting the real and imaginary parts of this interpolation, we obtain:



At $\theta \in \{0, e^{j\frac{2\pi}{3}}, e^{j\frac{2\pi}{3}2}\}$, we can see that the imaginary component of our interpolation is 0 and the real component matches the sampled y_i, as expected. Unfortunately, everywhere in between, our interpolation has a nonzero imaginary component!

This is probably not what we wanted. Starting with only real sample points, we'd probably want our interpolated values between the samples to be real as well. So this is a problem. What can we do about it? Do we have any freedom?

5 Changing the basis functions without changing the matrix B

To resolve this, we will try to do as little work as possible. We have already got a pretty good method for finding parameter values — a unique solution always exists, the coefficients are all of similar orders of magnitude, and the B matrix has lots of nice properties. It is easy to invert. Because of the orthogonality properties inherent in B, it is also very easy to project onto subspaces that are defined by a subset of the columns — we can just keep those coefficients and zero out the others. This means that it is also very compatible with least-squares fitting. The only issue is that our interpolation has a nonzero imaginary component even when given purely real points to interpolate.

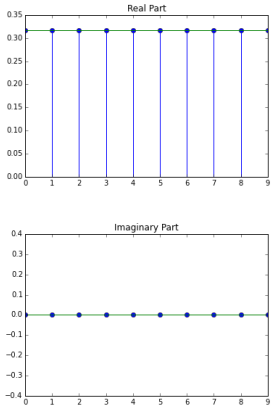
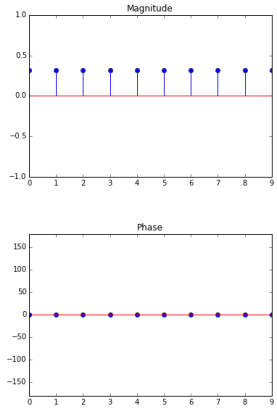
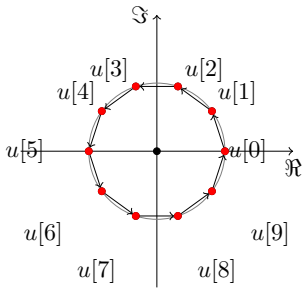
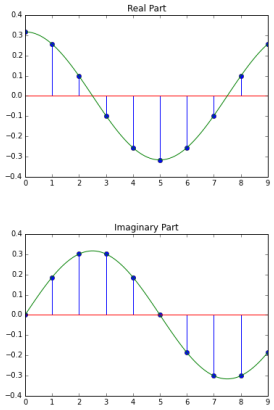
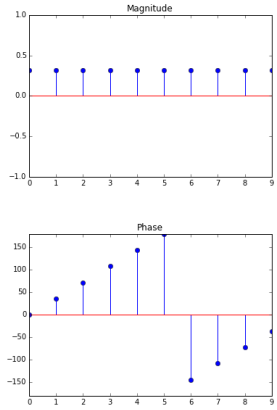
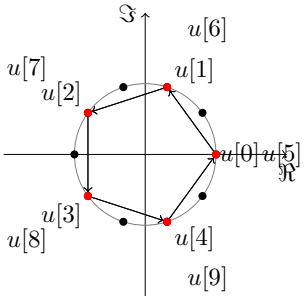
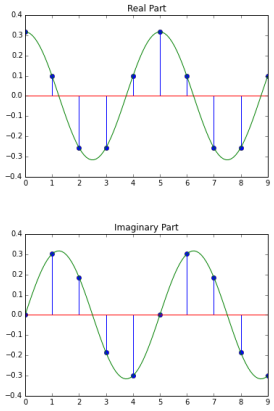
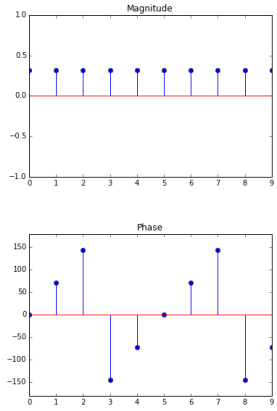
We want to make the smallest change possible to our approach such that, assuming we start with real samples y_i, we can be assured that our interpolation $h(e^{j\theta})$ is real for all real θ. In particular, we want to keep the same B matrix and α_is, but just change the parametric function we use to interpolate. In other words, we want to change the basis functions.

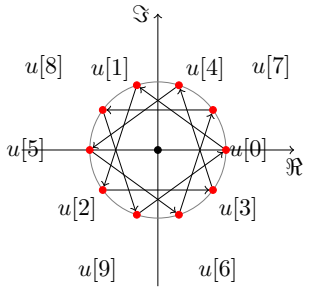
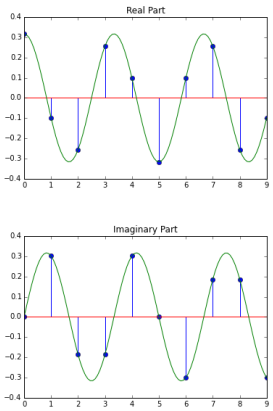
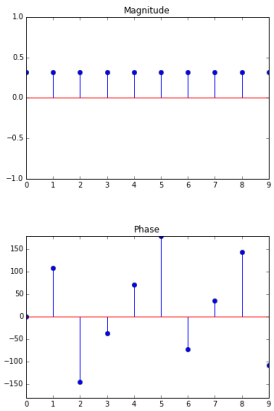
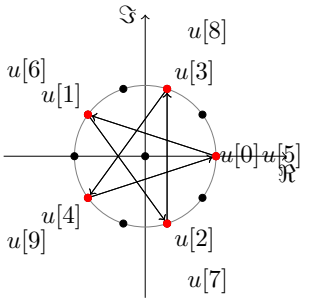
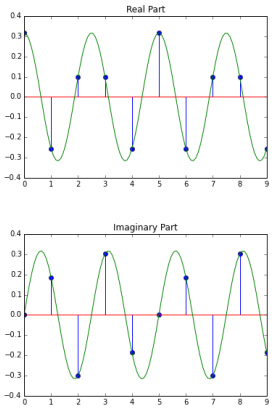
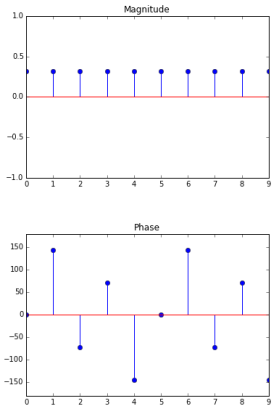
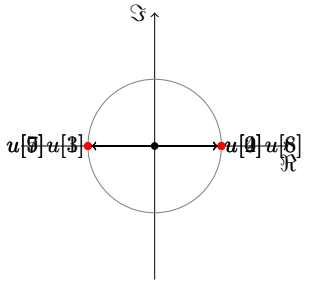
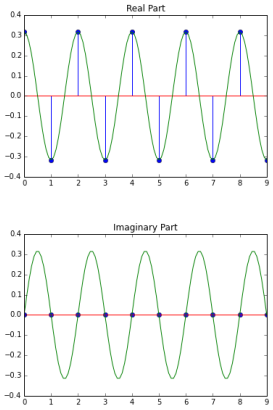
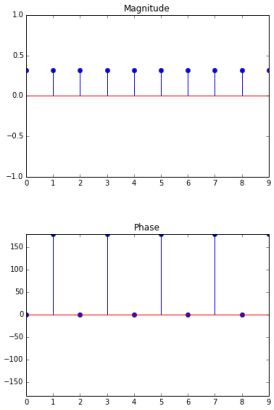
5.1 Taking a closer look at what is going on

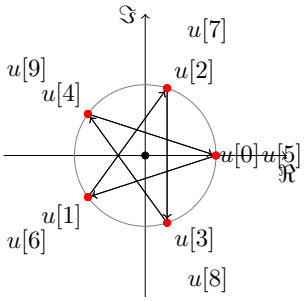
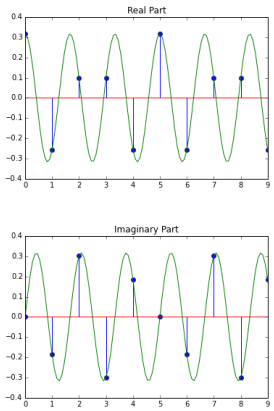
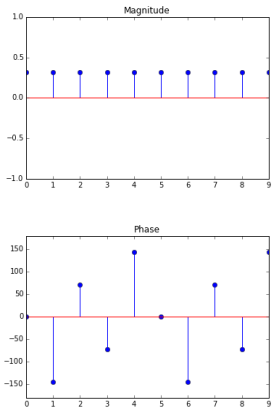
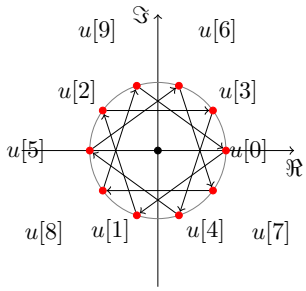
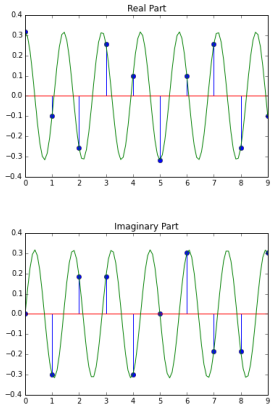
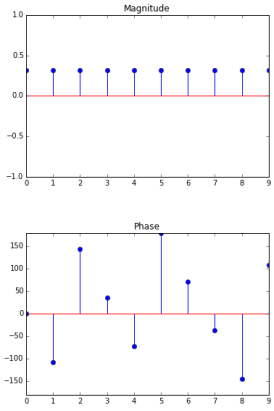
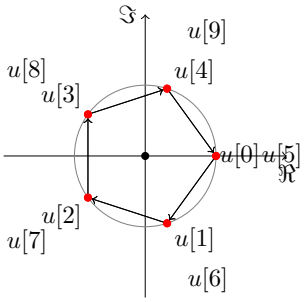
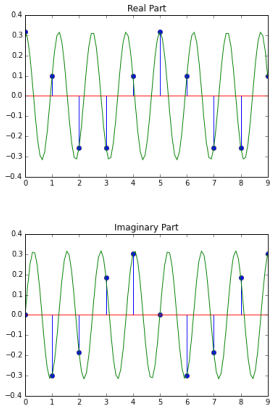
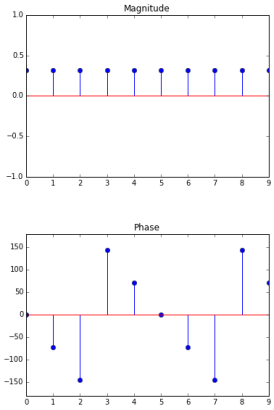
It is worth actually looking at an example of what the DFT basis vectors $\vec{b}_k$ or $\vec{u}_k$ look like. These are essentially complex exponentials, and as the k increases, they "wiggle" more and more. This is why k is often referred to as the frequency associated with that vector. They always have an integer number of periods between in the total duration m.

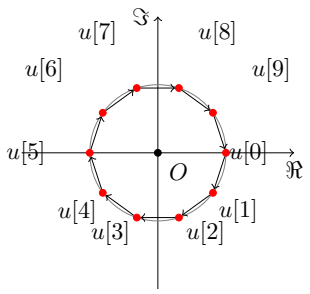
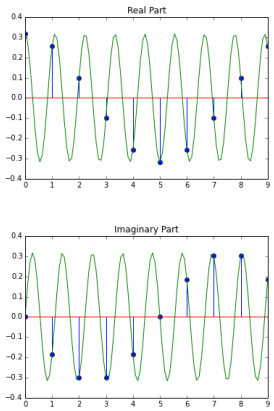
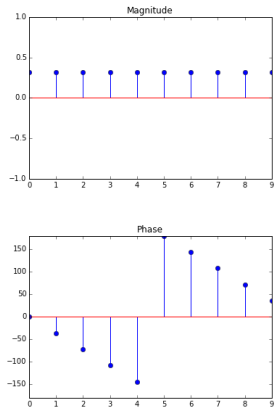
Here, we will illustrate $\vec{u}_k$ for the $m = 10$. The only reason we have chosen to illustrate $\vec{u}_k$ instead of $\vec{b}_k$ is so that the magnitude plot in polar coordinates of what the values inside the vector is will be slightly less boring. For $\vec{b}_k$, the magnitude of every entry in the vector is 1. Here, for the Cartesian Coordinates plots, we

have also drawn the underlying complex exponential that is being sampled to create the specific DFT basis vector being illustrated.

k	$u_k[t]$	Unit circle plots	Cartesian coordinates plots	Polar coordinates plots
0	1			
1	$e^{j\frac{2\pi t}{10}}$			
2	$e^{j\frac{2\pi 2t}{10}}$			

3	$e^{j\frac{2\pi 3t}{10}}$			
4	$e^{j\frac{2\pi 4t}{10}}$			
5	$e^{j\frac{2\pi 5t}{10}}$			

6	$e^{j\frac{2\pi 6t}{10}}$			
7	$e^{j\frac{2\pi 7t}{10}}$			
8	$e^{j\frac{2\pi 8t}{10}}$			

9	$e^{j\frac{2\pi 9t}{10}}$			

Looking at these vectors, we see that there are lots of symmetries in them. We also notice that there is a curious deja-vu feeling when looking at the circle diagrams — we go decagon to pentagon to lotus to pentagram to line and then walk back in the reverse order: back to pentagram (but reversed arrows), to lotus (also reversed arrows), to pentagon (reversed arrows again), to decagon (also reversed). The labels are also mirrored across the real axis, which tells us that there is a complex conjugation that is happening.

5.2 Returning to interpolation

Currently, our interpolation is of the form

$$g(e^{j\theta}) = \alpha_0 + \alpha_1 e^{j\theta} + \alpha_2 e^{j2\theta} + \cdots + \alpha_{m-1} e^{j(m-1)\theta}.$$

We would like to suppress the imaginary components of our interpolation. One way of doing so would be to add the complex conjugates of each term, so only the real terms remain. But there is no obvious way of getting complex conjugates to appear in our above equation without changing the coefficients α_i.

To do so, we need to take a closer look at our coefficients. Looking at one row of our equation for $\vec{\alpha}$, we see that

$$\begin{aligned}\alpha_i &= \frac{1}{m}\sum_{k=0}^{m-1} B^*[i][k]y_k \\ &= \frac{1}{m}\sum_{k=0}^{m-1} \omega_m^{-i\cdot k} y_k.\end{aligned}$$

Thus, since the y_j are real, its conjugate must equal

$$\overline{\alpha_i} = \frac{1}{m}\sum_{k=0}^{m-1} \omega_m^{i\cdot k} y_k.$$

But as ω_m is an mth root of unity,

$$\omega_m^{i\cdot k} = \omega_m^{i\cdot k - k\cdot m} = \omega_m^{k(i-m)}.$$

This is the mirroring across the real axis that we were seeing in the circle diagrams of the labels! This

means:

$$\overline{\alpha_i} = \frac{1}{m}\sum_{k=0}^{m-1} \omega^{k(i-m)} y_k = \frac{1}{m}\sum_{k=0}^{m-1} \omega^{-k(m-i)} y_k = \alpha_{m-i}.$$

In other words, the conjugates of the coefficients α_i are simply the coefficients of the "reversed" coefficient vector $\vec{\alpha}$.

So now we have some hope for leveraging complex conjugates! We remember from Phasor Analysis of circuits that by using a complex exponential with a negative sign together with every use with a positive sign, we will get real valued functions as long the coefficients are complex conjugates.

So can we do this? We originally generated the k-th column of the B matrix by viewing them as samples from the monomial "x^k" evaluated at the m-th roots of unity. How can we just change the function?

5.3 Aliases

This is where we get to a matter that is at the heart of all machine learning. Trying to learn a function from a finite amount of data is fundamentally an underspecified problem because there are always infinitely many functions that pass through any finite amount of data. If we view the actual data as a thing or object, we can view the abstract function that generated the data as its "name." The issue is that the same data has many possible names[3]. This issue is called "aliasing" in the literature. When we want to interpolate at some point where hadn't collected data, we need to summon[4] forth an evaluation by calling a specific name. This disambiguation is something that we are always doing, either implicitly or explicitly.

In this case, the symmetries inherent in the roots of unity let us see lots of the different names/functions that correspond to our columns of the B matrix. In particular, if we look at the k-th column $\vec{b}_k$, we see that it could have come from any formal monomial "x^{k+qm}" for any integer q. This is because[5] we can look at the i-th entry in $\vec{b}_k$ and notice that $(\omega_m^i)^{k+qm} = \omega_m^{ik}\omega_m^{qmi} = \omega_m^{ki}(\omega_m^m)^{qi} = \omega_m^{ki}(1)^{qi} = \omega_m^{ki}$.

This means we are free to summon forth any of these or even any convex combination (a linear combination where the weights sum to 1) of these "x^{k+qm}" and these are all valid choices. The resulting functions will interpolate all the data points.

6 "Natural" interpolation using the DFT coefficients

The reality of aliasing tells us that we need to make a choice for which functions to use to interpolate. We know what we want in this case. We want to be able to get real values for our interpolation if our original data was real, and this means that we need to leverage complex conjugates. This means that since we have $\alpha_1 = \overline{\alpha_{m-1}}$ we would prefer to summon forth x^{-1} instead of x^{m-1} associated with the parameter α_{m-1}. This is valid because $-1 = m-1+(-1)m$ and hence x^{-1} and x^{m-1} are both aliases of the last column of the B matrix. The same applies for $\vec{b}_{m-2}$ — we can view this as coming from x^{-2} and so on. This allows us to pair up the terms and cancel out all the imaginary parts.

[3] Notice how this classical terminology is somewhat curiously sample-centric. It views the sampled vectors as being the objects and the functions as being "names" — and so the same vector has many aliases. You might argue that reality is actually reversed and the phenomenon is actually one of Dopplegangers. The functions are the actual objects of interest. And many different functions can look the same if you only sample them at those points. The name of the game here is in choosing among these Dopplegangers.

[4] It is this deep cultural connection to the idea of "true name magic" that explains the classical terminology. When programming, we "invoke" and "call" functions — by saying their names. We do this to make them manifest. You saw this more deeply in 61A when you were doing your interpreter project. As a result, as engineers, names have power for us. Hence this curious terminology. Anyway, you'll get used to it.

[5] Because we are dealing with integer powers, we are allowed to do these manipulations.

6.1 The case of m odd

The story becomes easiest to understand if we stick to m being an odd number. Then, we have the 0-th coefficient for the constant term and the rest of the coefficients divide evenly into pairs. Imagine that we could replace the second half of the monomial terms with their conjugates, to obtain the new interpolation

$$h(e^{j\theta}) = \alpha_0 + \sum_{i=1}^{(m-1)/2} \left(\alpha_i e^{ji\theta} + \alpha_{m-i} e^{-ji\theta}\right) = \alpha_0 + \sum_{i=1}^{(m-1)/2} \left(\alpha_i e^{ji\theta} + \overline{\alpha_i e^{ji\theta}}\right).$$

The second equality is where we used the realness of the underlying $\vec{y}$ that we are interpolating.

Clearly, this interpolation is real, since each of the terms in the summation are real (by the property that adding a complex number to its conjugate gives twice the real part). But does it still pass through all of our sample points? It should by construction, but let's verify this. Since our sample points are of the form (ω_m^{i-1}, y_i), we can set up the system of equations to check as:

$$\begin{bmatrix} h(\omega_m^0) \\ h(\omega_m^1) \\ \vdots \\ h(\omega_m^{m-1}) \end{bmatrix} = \vec{y}.$$

Now, we can simplify the sum step by step. We start by splitting up the summation that defines our interpolation $h(\cdot)$:

$$\begin{aligned}
h(\omega_m^k) &= \alpha_0 + \sum_{i=1}^{(m-1)/2} \alpha_i (\omega_m^k)^i + \sum_{i=1}^{(m-1)/2} \overline{\alpha_i (\omega_m^k)^i} \\
&= \alpha_0 + \sum_{i=1}^{(m-1)/2} \alpha_i (\omega_m^k)^i + \sum_{i=m-1}^{(m+1)/2} \overline{\alpha_{m-i} (\omega_m^k)^{m-i}} \quad \text{(looking at the second sum backwards)} \\
&= \alpha_0 + \sum_{i=1}^{(m-1)/2} \alpha_i (\omega_m^k)^i + \sum_{i=(m+1)/2}^{m-1} \alpha_i (\omega_m^k)^{i-m} \quad \text{(conjugacy properties)} \\
&= \alpha_0 + \sum_{i=1}^{(m-1)/2} \alpha_i (\omega_m^k)^i + \sum_{i=(m+1)/2}^{m-1} \alpha_i (\omega_m^k)^i \quad \text{(root of unity property)} \\
&= \alpha_0 + \sum_{i=1}^{m-1} \alpha_i (\omega_m^k)^i.
\end{aligned}$$

So despite the different function being used to interpolate, on the sample points ω_m^k, it returns exactly the same thing as the traditional polynomial interpretation. Recall that we originally chose our $\vec{\alpha}$ to satisfy the equation $B\vec{\alpha} = \vec{y}$. Recall how we constructed the columns $\vec{b}_i$. Each of them were the evaluation of the ith basis function on the x-coordinates of our sample points. After changing our basis functions, the key observation is that although the functions are changed, their *evaluations* on these coordinates remain the same. Since the α_i were chosen to satisfy polynomial interpolation, this new function also interpolates perfectly.

So even after changing the basis functions used for the linearly parameterized function that we want to learn, the same B matrix remains valid, so the equation $B\vec{\alpha} = \vec{y}$ remains a necessary and sufficient condition in order for the interpolation to pass through all our sample points. Thus, what we can do is compute $\vec{\alpha}$ as

before, then plug into the new interpolation $h(\cdot)$, to get a purely real interpolation that passes through all of our sample points. So we have found exactly the interpolation that we want.

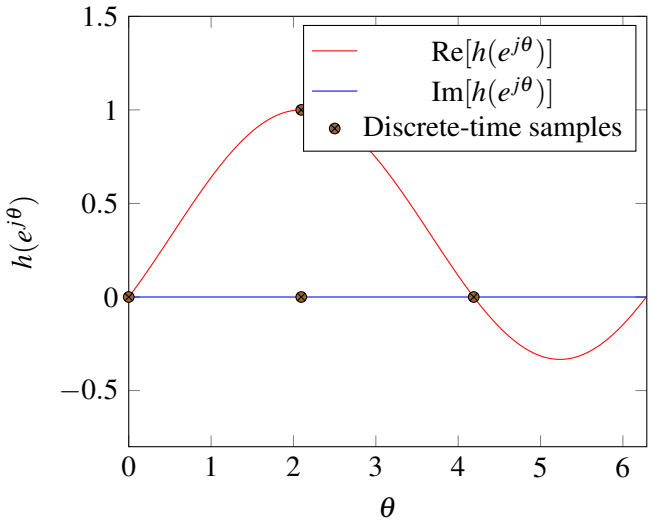
Let's verify that it works by returning to our example of interpolating $\vec{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$. Recall that we got $\vec{Y} = \frac{1}{3}\begin{bmatrix} 1 \\ e^{-2j\pi/3} \\ e^{-4j\pi/3} \end{bmatrix}$ for the DFT coefficients. Putting them into our interpolation we get

$$h(x) = \frac{1}{3} + \frac{e^{-2j\pi/3}}{3}x + \frac{e^{-4j\pi/3}}{3}x^{-1}$$

and expanding this out for arbitrary $x = e^{j\theta}$ on the unit circle, we get

$$h(e^{j\theta}) = \frac{1}{3} + \frac{e^{-j2\pi/3}}{3}e^{j\theta} + \frac{e^{+j2\pi/3}}{3}e^{-j\theta} = \frac{1}{3}(1 + 2\cos\left(\theta - \frac{2\pi}{3}\right))$$

Plotting the real and imaginary parts of this interpolation, we obtain:



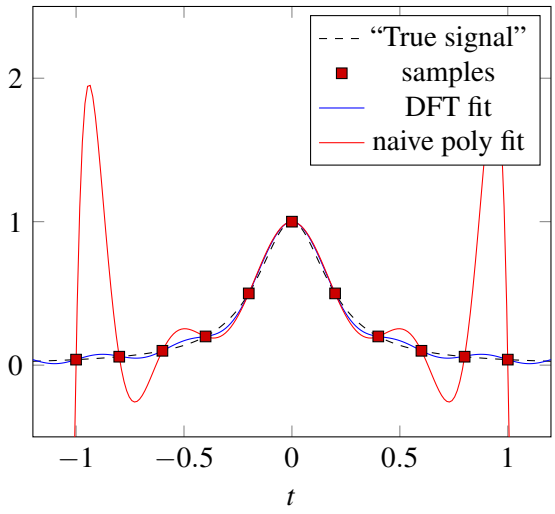
This clearly feels like a qualitatively "simpler" interpolation and thus satisfies our intuition for what Occam's razor demands.

Returning to our example of 11 points. The values were $\vec{y} = [\frac{1}{26}, \frac{1}{16}, \frac{1}{10}, \frac{1}{5}, \frac{1}{2}, 1, \frac{1}{2}, \frac{1}{5}, \frac{1}{10}, \frac{1}{16}, \frac{1}{26}]^\top$. The DFT coefficients $\vec{Y} = \frac{1}{11}B^*\vec{y}$ and these can be used to get an interpolation $Y[0] + \sum_{k=1}^{5} Y[k]e^{jk\theta} + \overline{Y[k]}e^{-ji\theta} = Y[0] + 2\sum_{k=1}^{5} |Y[k]|\cos\big(k\theta + \angle Y[k]\big)$. gives us a DFT-based interpolation of $0.255 + (-0.159 - j0.047)e^{j\theta} + (-0.159 + j0.047)e^{-j\theta} + (0.077 + j0.050)e^{j2\theta} + (0.077 - j0.050)e^{-j2\theta} + (-0.036 - j0.042)e^{j3\theta} + (-0.036 + j0.042)e^{-j3\theta} + (0.014 + j0.030)e^{j4\theta} + (0.014 - j0.030)e^{-j4\theta} + (-0.004 - j0.026)e^{j5\theta} + (-0.004 + j0.026)e^{-j5\theta}$ This interpolation has the sampled points being viewed as being regularly spaced starting at 0 and heading towards 2π.

However, the meta-data tells us that the samples were taken at $-1, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1$. If we want to use the meta-data to line up the points and get an interpolation function in terms of the original points, we need to find the affine translation between the original x and θ. Here -1 for x translates to $\theta_0 = 0$. While $+1$ for x translates to $\theta_{10} = \frac{2\pi 10}{11}$. This means that $\theta(x) = \frac{2\pi 5}{11} + \frac{2\pi 5}{11}x$ is the affine map from

original x inputs to the corresponding θ. We see that $\theta(-1) = 0, \theta(-0.8) = \frac{2\pi}{11}, \theta(-0.6) = \frac{2\pi 2}{11}, \theta(-0.4) = \frac{2\pi 3}{11}, \theta(-0.2) = \frac{2\pi 4}{11}, \theta(0) = \frac{2\pi 5}{11}, \theta(0.2) = \frac{2\pi 6}{11}, \theta(0.4) = \frac{2\pi 7}{11}, \theta(0.6) = \frac{2\pi 8}{11}, \theta(0.8) = \frac{2\pi 9}{11}, \theta(1) = \frac{2\pi 10}{11}$.

Using this we can plot:



Notice how much better and more natural the DFT-based interpolation is for this data. This naturalness is why we use DFT-based interpolations or analogous DFT-based projections to get global approximations for data. Notice that this is indeed an approximation — the interpolation doesn't get our "true signal" perfectly right, but getting it perfectly right is in general too much to ask for in practice. The only way that would happen is if the true signal was indeed something that was perfectly representable in the basis that we had decided to use for fitting. What we are going for with our DFT-based interpolation is a good general-purpose approximation for signals that are smooth and not too wiggly over a finite domain of interest.

A further thing to notice is that by choosing to map the finite domain of interest to the unit circle, we are indeed making another implicit choice. After all, a circle has no natural beginning and no natural end — it is a loop. This means that the functions we are going to get are always going to be implicitly periodic. It turns out that there are perfectly natural ways to eliminate that assumption. For example, we can put all our sample points from 0 to π instead of from 0 to 2π, and just mirroring them on the other side of the unit circle. All sequences are naturally periodic if you walk through them up and then down — you are guaranteed to end up where you started. This goes by the name of the DCT and is something you'll see more about in 123. There are similar natural ways to adapt to nonuniformly spaced samples, etc. Orthogonality gets lost and the computations slow down, but it is possible to proceed.

6.2 The even m case

All that remains is the case of m even. Here, there is a slight twist because the DFT coefficients don't naturally come in complex conjugate pairs. As before, there is the special 0 coefficient α_0 which is real because the first column of the B matrix is real and we are assuming that the y_i values we are interpolating are real. But for the even m case, there is another special case of $\alpha_{\frac{m}{2}}$. This is also real because $\omega_m^{\frac{m}{2}} = -1$ and hence the middle column $\vec{b}_{\frac{m}{2}}$ of B consists entirely of alternating $-1, +1$s. Because of this, since the inverse of B has $\frac{1}{m}\vec{b}_k^*$ for its k-th row, we know that $\alpha_{\frac{m}{2}}$ is always real if the y_i values are all real.

For the rest of the coefficients, they come in complex conjugate pairs. So what we did for the odd m case will work for interpolation. But what about the $\frac{m}{2}$ coefficient? What function should we summon forth

here? $x^{\frac{m}{2}}$ isn't the right choice on its own — it will be complex. For the same reason, $x^{-\frac{m}{2}}$ also isn't the right choice on its own — it too will be complex. But the average of two aliases is also an alias, so we can use $\frac{1}{2}(x^{\frac{m}{2}}+x^{-\frac{m}{2}})$. Evaluated on $x=e^{j\theta}$ this is $\frac{1}{2}(e^{j\frac{m}{2}\theta}+e^{-j\frac{m}{2}\theta})=\cos\left(\frac{m}{2}\theta\right)$ which is definitely real.

Putting everything together, we get for our interpolation (when the original data was real):

$$h(e^{j\theta})=\alpha_0+\alpha_{\frac{m}{2}}\frac{1}{2}(e^{j\frac{m}{2}\theta}+e^{-j\frac{m}{2}\theta})+\sum_{i=1}^{\frac{m}{2}-1}\left(\alpha_i e^{ji\theta}+\overline{\alpha_i e^{ji\theta}}\right)$$

which is definitely real.

A final comment can be made on what should we do if the original samples $\vec{y}$ were complex to begin with. (This is not an even vs odd thing, but the even case is what brings it to mind more strongly.) The above expressions continue to be valid interpolations as long as we replace $\overline{\alpha_i}$ with what it actually comes from: α_{m-i}. It was the realness of $\vec{y}$ that gave the complex conjugacy relationship between the pairs of α_i, α_{m-i} coefficients. The fact that these are interpolations comes from the fact that $\vec{b}_k$ has many names, and we have chosen to invoke one of them. That didn't care about what $\vec{y}$ was like. That said, when we have $\vec{y}$ complex, we need to ask where they are coming from and how we want to decide on the best strategy for interpolating them. Sometimes the answer is as above, and sometimes the nature of the problem will want to have us use a different strategy. You'll learn more about that in 120 and 123. It turns out to be particularly relevant in the context of making communication systems.

7 Other DFTs

The DFT is an amazingly versatile tool. Interpolation and global approximations of functions is just one of its uses. The "Fourier style" of thinking turns out to be helpful in other ways as well.

As a result, there are actually multiple definitions that you will encounter in the literature and when using codebases or libraries that exist. It is helpful to always think about the DFT as a coordinate transformation. As we saw when we illustrated them, because of the interesting properties of exponentials and powers, the kth DFT basis vector represents a counterclockwise path through the m-th roots of unity starting at 1 and then moving by k roots at a time. So the 0th DFT basis vector just stays at 1. The 1st DFT basis vector goes through the n roots of unity one at a time clockwise starting with 1. The 2nd DFT basis vector takes two trips around the unit circle starting at 1 and moving two at a time. And so on.

There are several different ways to normalize the DFT basis vectors, each coming from different motivations. Below is a table of three such normalizations in widespread use. They don't have standardized names[6], and so we will give them names that we like.

Variant	Normalizing Factor	Notation	entries
Polynomial DFT basis	1	$\vec{b}_k$	$b_k[i]=e^{j\frac{2\pi ik}{m}}$
Orthonormal DFT basis	$\frac{1}{\sqrt{m}}$	$\vec{u}_k$	$u_k[i]=\frac{1}{\sqrt{n}}e^{j\frac{2\pi ik}{m}}$
Traditional/Classic DFT basis	$\frac{1}{m}$	$\vec{d}_k$	$d_k[i]=\frac{1}{n}e^{j\frac{2\pi ik}{m}}$

[6] To add to the confusion, inspired by certain discrete settings, some folks use the conjugate of the classic DFT with the basis vectors effectively going clockwise to start. For example, without any attempt to tell it to do otherwise, Wolfram Alpha and Mathematica default to this alternative. The moral is, you have to be careful and check to see if any software package is indeed giving you what you are asking for.

The table above defines the relevant matrices $B = [\vec{b}_0, \vec{b}_1, \cdots, \vec{b}_{m-1}]$ or $U = [\vec{u}_0, \vec{u}_1, \cdots, \vec{u}_{m-1}]$ or $D = [\vec{d}_0, \vec{d}_1, \cdots, \vec{d}_{m-1}]$ and correspondingly result in relationships (e.g. $B\vec{F} = \vec{f}$) between the learned parameters $\vec{F}$ and the original data $\vec{f}$. They clearly all differ only by scaling[7].

Their inverses are what we use to go from observations $\vec{y}$ to the corresponding DFT coefficients $\vec{Y}$. These are $B^{-1} = \frac{1}{m}B^*$ for the polynomial-style DFT, $U^{-1} = U^* = \frac{1}{\sqrt{m}}B^*$ for the orthonormal DFT, and $D^{-1} = B^*$ for the classic/traditional DFT.

You will learn more about the traditional/classic DFT in 120/123, and see why it is defined the way that it is. The reason has to do with eigenvalues and eigenvectors, and is intimately connected to Phasors and Transfer functions. Essentially, the coefficients obtained using the classic DFT represent the eigenvalues of a particular matrix, and they play the role of a Transfer function. You will see in 120/123 how you can take limits to explicitly connect these two.

Contributors:

- Rahul Arya.
- Anant Sahai.

[7] FYI: Numpy given normalization "none" will compute the Traditional/Classic map and solve the equation $D\vec{F} = \vec{f}$ to return $\vec{F}$. If you give Numpy the normalization "ortho", then it will compute the Orthonormal DFT basis version and solve $U\vec{F} = \vec{f}$ and return $\vec{F}$. Nobody bothered to define a separate normalization for the Polynomial version because it turns out that it can be computed by calling the inverse classic DFT on a "flipped" vector $\vec{f}$. Can you see why? This has to do with the complex conjugacy relationships between the columns of B. By appropriately "flipping" the vector, you can effectively multiply by the conjugate of B. Here, flipping involves reversing the order of almost all the entries, but keeping the 0-th entry the same.

Anyway, since this is an EECS course and the FFT (Fast Fourier Transform) is literally considered one of the most important algorithms of all time, we felt that it was important to connect to numpy here. The FFT itself is a topic for 170 and 120/123. It tells us that we can compute the DFT coefficients far faster than having to do a full matrix multiplication.

