# Fall 2019 EECS 16B: Big Picture Story, Key Ideas

Anant Sahai*

## 1 Module 1

### 1.1 The Initial Story

Dear 16B students,

This pinned post will serve as the master thread for conversations pertaining to the first section of lectures in Module 1. We know that sometimes, it is possible to lose the forest for the trees, and hopefully, having the short form of the big picture story in a running post will be helpful.

Module 1 began with a motivating question — **can we understand what is limiting the ability of modern computers to run fast?**

Modern computers are based on the idea of digital logic that is based in binary — information is represented with abstract 0s and 1s that are represented electrically using high and low voltages. While the details of how digital logic can be used to realize computation will have to wait until 61C (where that is one of the major topics), we did introduce the "transistor" as playing a very important role since it is basically an electrically controlled switch. The NMOS and PMOS transistors were introduced as a complementary pair of transistors – with the NMOS turning on (closing the switch between source and drain) when the gate to source voltage is a large enough positive number, and the PMOS turning on when the gate to source voltage is a negative enough negative number. In both cases, the idea of "large enough" or "negative enough" is captured by a threshold voltage, which in practice is around $0.4$V in magnitude these days.

The actual realistic physics of a transistor is out of scope for 16B since Physics courses are not a prerequisite. However, an extremely caricatured "conducting puddle" model was introduced just to help students remember. The idea of this model is that the gate of the transistor is like one terminal of a capacitor with the other part being in the silicon between the source and the drain. When the voltage on the capacitor is high enough in the right direction, a "puddle" of charge carriers forms in the silicon to balance out the charge being put on the gate. When the puddle is large enough, it connects the source and the drain, allowing current to flow between them. (Aside: this article on threshold voltage has an interesting animation that literally shows the "puddle" growing as the gate voltage changes.) The "source" is the terminal that can be viewed as where the relevant charge carriers spill from to form the puddle. For NMOS, these carriers are electrons having a negative charge. For PMOS, these carriers are what are called "holes" and they have a positive charge. Actually understanding this requires more physics and so none of this is in scope for an exam, homework, etc... However, we tell you

*proofread and converted to LaTeX by Neelesh R. in Sp21. Reach out (email: neelesh.r@berkeley.edu) if there are any issues. Note that the initial publication of this text was on Piazza for Fall 2019 EE 16B, hence the references to posts and specific worksheets/lecture context.

this just because it might help some of you remember the difference between PMOS and NMOS.

The important things about transistors are two-fold (at this point in the course): We can build logic gates out of them. We showed how a PMOS connected to an NMOS could form the simplest logic gate: an inverter. When the input is high, the output is low. When the input is low, the output is high. This could be understood by looking at which transistors have turned their relevant switches on vs off. We can model the transistor in more detail as having a capacitor at the gate (connected to the source) and having a switch that either connects the source and the drain together through a resistor, or leaves them disconnected from each other.

This then let us understand the intuitive physical reason why computers can't run infinitely fast. To build sophisticated computations, logical gates must drive other logical gates. The states ("0" vs "1") of the inputs and outputs need to change. But these states are represented by finite amounts of charge sitting on these transistor gate capacitances. These charges must be put there (to get to a logical "1"), and removed to ground (to change into a logical "0"). This placing and removing of charge must happen through the finite resistance that occurs between the source and drain of the relevant transistor that has turned on. The current through the resistor will be finite because the voltages are finite, and thus there is a finite rate of change for the charge on the gate. Consequently, the voltage can't change too fast.

From this intuitive understanding, we sought out a more quantitative understanding. How exactly does the voltage behave as a function of time? In 16A, we always assumed that change happened "fast enough" when switches got toggled. But here, we care about the speed and so the function of time matters. To understand this, we wrote out the system of equations that governed a simple circuit (two inverters — one feeding the other) as far as the output voltage of the first inverter was concerned. We assumed that before the change it was high $V_{DD}$, and wanted to see what happens as it decays to zero after the input change.

Unlike in 16A, despite the fact that we had two elements (a resistor and a capacitor) in a loop and used KCL, we didn't get a solution immediately. (As we would have gotten had we connected a voltage source to a resistor in such a loop.) Instead, we got an equation that related the time derivative of the voltage to the voltage itself. $\frac{d}{dt}V = -\frac{1}{2RC}V$. Here, the "2" in the "$2RC$" comes from the fact that there are two capacitors that need to be discharged: the PMOS gate and the NMOS gate. In general, the "$2C$" will be the total capacitance that the logical gate driving the output needs to deal with. The more logical gates this output is driving, the bigger that term will be.

We mentioned that such things are called differential equations but represent something new — we haven't seen these things before.

Because the key objective in the 16 series is to better equip you with the tools needed to systematically think about new and previously unseen problems, we thought about how one could approach this. In the absence of certainty, one has to be willing to speculate and reason by analogy — even when there is not as yet any true justification for this analogy. In this case, we noticed that the voltage V is a function of time and hence can be thought of as a vector of sorts — we index into it by time. The $-\frac{1}{2RC}$ is a constant and so the $\frac{d}{dt}$ operation is playing the role of a matrix — it returns a function of time when given a function of time. After a quick reality check verifying that this Matrix analogy is plausible (it maps $0$ to $0$, distributes over addition, and commutes with scalar multiplication — the same properties of linearity that we saw in 16A matrices satisfied), we ventured the guess that it feels like we are looking for the relevant eigenvector of the $\frac{d}{dt}$ operator.

At this point — still in speculation mode mind you — we remembered that $\frac{d}{dt}e^t = e^t$ and in a way, this is the very *raison-d'etre* for the exponential function. It's the function whose derivative is itself. From an eigen-

value/eigenvector point of view, it's like an eigenvector for $\frac{d}{dt}$ corresponding to the eigenvalue $1$. Once we had grasped this thread, it was a straight shot to noticing that the chain rule in calculus tells us that $\frac{d}{dt}e^{st} = se^{st}$ for any constant $s$ and so $e^{-\frac{1}{2RC}t}$ has the property of also being an eigenvector of $\frac{d}{dt}$ that corresponds to eigenvalue $-\frac{1}{2RC}$. At this point, we could see that $e^{-\frac{1}{2RC}t}$ is a possible solution, but because any constant multiple of an eigenvector is also an eigenvector, here we can see that any constant multiple $Ke^{-\frac{1}{2RC}t}$ would also be a candidate solution to the differential equation $\frac{d}{dt}V = -\frac{1}{2RC}V$.

So which $K$ should we choose? For this, we need to see if we have any other piece of information that we have not used yet. And indeed we do. We know that at time $t = 0$, the voltage in question was $V_{DD}$. This means that $K = V_{DD}$ is our guess and we think that $V(t) = V_{DD}e^{-\frac{1}{2RC}t}$ for $t \geq 0$.

Now, the status of all the previous reasoning is that of a kind of speculation. Systematic speculation, yes. But speculation nonetheless. We can check that our proposed solution satisfies the equations and everything that we know about the problem. But it still doesn't let us make a firm prediction for the actual behavior of the circuit.

This is because we don't know if there are other possible solutions to the same differential equation with the same initial condition. We saw in 16A that in general, just because we have a system of equations and a candidate solution — that doesn't meant that the solution necessarily reflects the underlying reality. The problem might actually be ambiguous as written (just as systems of linearly dependent equations can be ambiguous) even though it seems firm. We need a proof of uniqueness.

To try and show uniqueness, we need to think about some hypothetical $y(t)$ that also satisfies the same differential equation. For ease of notation, we just looked at $\frac{d}{dt}x(t) = \lambda x(t)$ with initial condition $x(0) = x_0$. We know that $x_d(t) = x_0e^{\lambda t}$ for $t \geq 0$ solves this. It is a solution. But how do we know it is the only one? We have to consider a hypothetical $y(t)$ that also solves this. Such a $y(t)$ would have $y(0) = x_0$ and $\frac{d}{dt}y(t) = \lambda y(t)$. How do we show that $y(t)$ must be the same as $x_d(t)$? We have many different things we could try. (And these different proof pathways and patterns will be exercised at different points in the course) The path we followed was inspired by the ratio path — to show two things are the same, you can show that their ratio is always $1$.

We just considered $y(t)e^{-\lambda t}$. This is something that we want to be able to show must equal the constant $x_0$ for $t \geq 0$. How can we show its constancy? We can look at the derivative. The derivative, by the product rule and chain rule, is $\frac{d}{dt}y(t)e^{-\lambda t} = (\frac{d}{dt}y(t))e^{-\lambda t} - \lambda e^{-\lambda t}y(t) = \lambda y(t)e^{-\lambda t} - \lambda e^{-\lambda t}y(t) = 0$. Since the derivative is always zero, it must be a constant. Which constant? We can just evaluate it at anywhere, so we pick $0$ since that is where we know how to evaluate it. $y(0)e^{-\lambda 0} = y(0) = x_0$. So indeed, this is the constant $x_0$.

This proof told us that we have uniqueness of solutions for such differential equations. This means that we are free to guess the solutions by whatever means we want, and as long as they turn out to satisfy the differential equation and its initial conditions, they are guaranteed to be the unique solutions. Having a uniqueness theorem is liberating — because it means that we can use nonrigorous heuristics and other ways to guess, and get confidence in the results by simply checking. (This is different from how Gaussian elimination was derived — in Gaussian elimination, every step was valid by construction and so the process of solving was itself a proof.)

Using this, we are now certain that $V(t) = V_{DD}e^{-\frac{1}{2RC}t}$ does predict the behavior for $t \geq 0$. Here, $\tau = 2RC$ defines the natural time-constant for this situation. That means $\tau$ is the natural unit of time that governs the behavior. After one time constant, the output has dropped to $e^{-1} \approx 0.37$ times its initial behavior. That means that if $V_{DD} = 1\text{V}$, and $V_{th} = 0.4\text{V}$, it means that after one time constant, the output voltage has dropped enough to be able to switch the behavior of the follow-on inverter. (Or more generally, the NMOS transistors at the bottom of the subsequent logical gate's inputs.)

Having seen this for the output of the inverter going from high to low, it is useful to see it for the other direction: going from low to high.

Setting up the circuit, we see that after the input to the inverter has gone low, the PMOS turns on while the NMOS turns off. This means that as far as the gate capacitances on the input to the next gate are concerned, they are connected through a resistor to the high voltage $V_{DD}$. The resulting differential equation for their voltage is now $\frac{d}{dt}V(t) = \frac{1}{2RC}(V_{DD} - V(t))$ with initial condition $V(0) = 0\text{V}$. This is very close to the form that we know how to solve and for which we've proved a uniqueness theorem. The only problem is that we have this annoying $V_{DD}-$ term. At this point, we can use the classic technique you know from your calculus courses — change of variables. We can define $\widetilde{V}(t) = V_{DD} - V(t)$ and then write our differential equation in terms of it. (In other word, make a definition that swallows the annoying term and hope that it goes away.) This gives us $\frac{d}{dt}\widetilde{V}(t) = -\frac{1}{2RC}\widetilde{V}(t)$ with initial condition $\widetilde{V}(0) = V_{DD}$. Notice that these steps of changing variables and reducing the problem to this form do not have the conceptual status of guesses — these are rigorous reversible steps. But now, we know how to solve this and are guaranteed uniqueness of the solution: $\widetilde{V}(t) = V_{DD}e^{-\frac{1}{2RC}t}$ for $t \geq 0$, which gives us $V(t) = V_{DD}(1 - e^{-\frac{1}{2RC}t})$ for $t \geq 0$. A rising curve that converges to $V_{DD}$ as expected.

At this point, we know how to solve any differential equation of the form: $\frac{d}{dt}x(t) = \lambda x(t) + u$ with initial condition $x(0) = x_0$, for any constants $u$ and $\lambda$. But this actually means that we can "stitch together" solutions to understand such differential equations even if the $\lambda$ and $u$ aren't strictly constant forever, but instead are constant for any specific durations. The initial condition of the next segment is wherever the differential equation's $x(t)$ ended up at the end of the previous segment. We can grind out the solutions for such piecewise-constant problems.

In discussion, you saw how we can use this, together with what you know about limits and Riemann Integration, to guess a the solution where $u$ is a nice-enough function of time. There were a few key steps. First, you decided to focus on what happened from one segment's endpoint to the endpoint of the following segment. Because this was a discrete chain, you could fold it into a sum that was comprehensible. Then, you looked at the terms in the sum and noticed that they were almost like a Riemann integral, except that you seemed to be missing the "width of the rectangles" term. By using a simple Taylor approximation of the only candidate term for this "missing width" term, you saw that indeed behaved exactly like the width when that width was small. Then, taking a limit, you got a Riemann integral to give a guess for a solution to $\frac{d}{dt}x(t) = \lambda x(t) + u(t)$ with initial condition $x(0) = x_0$. Namely: $x(t) = x_0e^{\lambda t} + \int_0^t u(\tau)e^{\lambda(t-\tau)}d\tau$ for $t \geq 0$.

With this expression in hand, we could advance to the next phase of Module 1 (below).

## 1.2 The Story Concludes...

Dear 16B students,

This pinned post reflects the continuation of the story from above.

Once we had a basic understanding of scalar differential equations, at this point, we introduced the more holistic motivation and organizational principle behind all of 16B, including the rest of Module 1. We want to understand what is required to make cyborgs — the seamless blending of humans and machines to restore function and augment the abilities of people. This is a civilization-level undertaking, and one that we are far from being able to complete. However, it promises a lot of potential betterment of the lives of people with disabilities, injuries, etc. We used the example of a hypothetical brain-controlled prosthetic robot limb as a an

example. We need to understand how to think about this problem. Namely, how we can extract signals from the brain, figure out a person's intentions, and then how to control the robot limb to do what we know the person wants. Ideally, we'd like to do this without having to have bulky and/or unsightly wires connecting the brain interface to the robot limb. Module 2 will deal with the understanding required to control the robot. Module 3 will deal with figuring out the intentions of the person. For the rest of Module 1, we will deal with getting signals from the brain. The robot car in lab is a caricature of a cyborg — sound signals play the role of neural signals and the car is like the robot arm. Figuring out what the person said is like figuring out intention.

In particular, we are concerned about the problem of interference — the electrical signals that we pick up from the brain will be contaminated with all the other stray electrical signals that are present in the environment — the 60Hz oscillation of the power mains in the wall, WiFi and cellular transmissions, etc. We need to *filter* out the undesired part. In doing this, analog circuits play an important role.

After all, we have just seen that a simple RC circuit is actually implicitly computing an integral just by its very nature. The question is whether this integral is good for anything. We studied the response to a cosine to see that we could indeed interpret the integral as doing something useful — the voltage on the capacitor seemed to be filtering for slower lower-frequency cosine waves while severely attenuating/shrinking any higher-frequency cosine that might be at the input voltage. It was behaving like a filter, that cared about how fastly or slowly varying a signal was.

But there is only so much that we can do with a single capacitor. So, to be able to build more interesting filters, we need to be able to understand circuits with more than one capacitor. So we setup a simple two capacitor circuit and saw that this gives rise to a nontrivial system of differential equations in which the rate of change of the voltage on each capacitor doesn't just depend on itself, but also on the the voltage on the other capacitor.

We saw that if we were somehow magically given a change of coordinates for our example, we could get the problem to decompose into two simpler problems — a purely scalar problem that we know how to solve, and a second scalar problem that has the solution of the first scalar problem as an input. This we also know how to solve.

In discussion, you started from the other side. Assume that you have a collection of independent scalar differential equations, and then see how a change of variables could make this very simple situation look more complicated. The change of variables made things look like you had a system of linear differential equations.

Then in lecture, the story continued from there. We asked how we could possible come up with the right change of variables ourselves. We talked about the idea of a coordinate changes, and drew one of the most important diagrams in this course:

$$\begin{array}{lccc} \text{"Original"} & \vec{x} & \xrightarrow{A} & \frac{d}{dt}\vec{x} \\ \text{Transformations} & V^{-1}\downarrow\uparrow V & & V^{-1}\downarrow\uparrow V \\ \text{"Nice"} & \widetilde{\vec{x}} & \xrightarrow{\widetilde{A}=V^{-1}AV} & \frac{d}{dt}\widetilde{\vec{x}} \end{array}$$

This showed how the "nice" coordinate system was in terms of the $V$-basis — namely that if $V = [\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n]$, then to get back into original coordinates, you just needed to multiply $\widetilde{\vec{v}}$ by $V$. The $i$-th coordinate of $\widetilde{\vec{v}}$, namely $\widetilde{\vec{v}}[i]$ is what multiplies $\vec{v}_i$ and so on. The above diagram applies no matter what $V$ we choose.

To get the $\widetilde{A}$ to have a nice form, we saw that it would be great to use an eigenbasis for $V$. Namely, suppose

that each of the $\vec{v}_i$ was a nontrivial eigenvector of $A$ with eigenvalue $\lambda_i$ — that is to say, $A\vec{v}_i = \lambda_i \vec{v}_i$ and moreover, that all of the $\vec{v}_i$ are linearly independent (so we can define $V^{-1}$). Then $\widetilde{A} = V^{-1}AV = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \lambda_n \end{bmatrix} = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$.

At that point, if we can find $n$ linearly independent eigenvectors, then we have found a change of variables that reduces the problem to an independent collection of scalar differential equations. These we know how to solve!

In discussion, this technique was demonstrated on the example from the previous discussion. We saw how it is possible to discover an underlying "nice" basis using this eigenspace-finding technique.

From there, we continued our discussion by introducing the concept of an inductor. Initially, we simply motivated this by a mathematical/physical curiosity and desire for symmetry — we know that the capacitor is something that connects $\frac{d}{dt}V(t)$ to the current $I(t)$ through a proportionality constant $C$. Is there anything that does the reverse: have a changing current $\frac{d}{dt}I(t)$ induce a voltage $V(t)$ via a proportionality constant $L$? Indeed there is and it is called an inductor. We handwaved that this is basically an electromagnet that stores energy in a magnetic field as compared to the electric field of a capacitor. Whereas a capacitor behaves like an open circuit when nothing is changing, an inductor behaves like a short when nothing is changing. No current vs no voltage.

Our differential equation solving tools allow us to handle circuits involving inductors in pretty much the same fashion as those involving capacitors — the only difference is that the derivative operation will be landing on a current and so that current will be a natural state to keep track of, just as the voltages on capacitors were natural state. As we explored this new circuit element, we saw something interesting emerge. When we looked at a circuit that consisted of just a capacitor and an inductor, the relevant system of differential equations had imaginary eigenvalues and complex eigenvectors! We just pushed forward anyway, and leveraged our understanding of change-of-coordinates. We used Euler's identity $e^{j\theta} = \cos\theta + j\sin\theta$ to understand what the complex exponential solutions were doing, and thankfully, when we converted back to the original coordinate system, all of the imaginary parts cancelled away. We were left with simple sinusoidal oscillatory behavior. This oscillatory behavior was what is new — at a physical level, this emerges from energy moving from the electric field of the capacitor to the magnetic field of the inductor and back again, forever.

In discussion, we focussed on reviewing complex numbers and making sure that everyone understood that there is nothing scary about them, even in the context of differential equations and changing variables. The discussion also started students on the HW.

With our bag of tools now seemingly rich enough to analyze every kind of linear system of differential equations that we can construct out of circuits, we then faced a quandary. While we could in principle analyze any circuit's response to any input, we have no way of helping us do design. In 16A, we had seen that to do design, it is useful to have useful building blocks and to be able to understand them individually, as well as what happens when we compose them in ways that facilitate understanding of the composition. We have all the building blocks from 16A at our disposal (we can do voltage dividers, can use op-amps to buffer and to apply gain, as well as use op-amps to add voltages — basically we can do all the linear operations we want), but what we want is a way to have filtering-type behavior where certain signals of interest are treated differently than others. And we need this filtering to be comprehensible. With the differential-equation perspective, we just

know that we will get integrals that we can evaluate. Trial and error alone is too painful, so we needed a way to understand that ideally, would better let us leverage everything that we already know from 16A to do design.

To get something that we can understand, we are willing to accept limitations. After all our goal is to be able to do design. Once we have a candidate design, we can always get the computer to grind out the analysis of the differential equations and integrals with the kinds of inputs that we are actually dealing with, and then can fiddle/tweak it to be closer to what we want. Ease of use of the understanding is very important. We noticed that in the homework, we saw that when we applied an input $u(t) = \tilde{u}e^{st}$ for $t \geq 0$ to the (paradigmatic) scalar differential equation $\frac{d}{dt}x(t) = \lambda x(t) + u(t)$, when $s \neq \lambda$, we got a solution that looked like $x(t) = Ke^{\lambda t} + \frac{\tilde{u}}{s-\lambda}e^{st}$ for $t \geq 0$, where the constant $K$ depended on the initial condition for $x(0)$ as well as $\tilde{u}, s, \lambda$. But in the filtering context, we care about what happens as time goes on, so we saw that since $e^{\lambda t} = e^{Re(\lambda)t}e^{jIm(\lambda)t} = e^{Re(\lambda)t}(\cos(Im(\lambda)t) + j\sin(Im(\lambda)t))$, that as long as $Re(\lambda) < 0$, this term will decay away in the long run. So, it seems that exponentials, when input into differential equations, end up coming out as the same exponentials.

To test this idea before developing it further, we decided to try this approach on an entire system of differential equations $\frac{d}{dt}\vec{x}(t) = A\vec{x}(t) + \vec{u}(t)$ where we assumed that the time-varying $\vec{u}(t) = \vec{\tilde{u}}e^{st}$ was a constant vector $\vec{\tilde{u}}$ times a single exponential function of $t$. Then we guessed that the solution would be of the form $\vec{x}(t) = \vec{\tilde{x}}e^{st}$ where $\vec{\tilde{x}}$ is a constant vector. Plugging all this in gave us $s\vec{\tilde{x}}e^{st} = A\vec{\tilde{x}}e^{st} + \vec{\tilde{u}}e^{st}$ where it is clear that we can just cancel all the $e^{st}$ to get a system of linear equations $s\vec{\tilde{x}} = A\vec{\tilde{x}} + \vec{\tilde{u}}$ which by collecting terms can be solved by $\vec{\tilde{x}} = (sI - A)^{-1}\vec{\tilde{u}}$. We noticed that this was going to work as long as the constant $s$ was not an eigenvalue of $A$.

So we can reduce a system of linear differential equations driven by constant multiples of a single exponential function to simply solving a system of linear equations. But where do these systems of linear differential equations come from in the context of circuits? They come from writing out all the algebraic (KCL equations, resistor equations, op-amp equations, etc.) equations and element/branch differential equations corresponding to capacitors and inductors. Before now, we just treated the $\frac{d}{dt}I_L(t)$ for inductors and the $\frac{d}{dt}V_C(t)$ for capacitors as additional unknowns and got the system of differential equations by solving for these unknowns in terms of their non-differentiated counterparts. This eliminated most of the algebraic linear equations. (You can view this as doing the downward pass of Gaussian Elimination using an ordering of the variables so that the $\frac{d}{dt}\cdot$ variables come second to last and their non-differentiated counterparts come last. Because there will be more unknowns than equations, Gaussian elimination will stop on the last $\frac{d}{dt}$-ed variable. Then, we can do a limited upward-pass (back-substitution pass) of Gaussian elimination to purge any dependence of one $\frac{d}{dt}$-ed variable on any other $\frac{d}{dt}$-ed variable. This gives rise to the system of differential equations in a systematic way — it is the lower block that remains after doing this back-substitution.) So why do we have to wait for the elimination before doing the guessed substitution? We can do it at the level of the individual element/branch equations themselves.

If we do that, we see that for an inductor, we start with $V_+(t) - V_-(t) = L\frac{d}{dt}I(t)$ where $V_+(t)$ is the voltage on the $+$ node of the inductor, $I(t)$ is the current flowing into that node (following passive sign convention), and $V_-(t)$ is the voltage on the $-$ node of the inductor. Substituting in $V_+(t) = \tilde{V}_+e^{st}$, $V_-(t) = \tilde{V}_-e^{st}$, $I(t) = \tilde{I}e^{st}$, we get $\tilde{V}_+ - \tilde{V}_- = Ls\tilde{I}$. This is exactly like a resistor, except with the $s-$dependent term $Ls$ in place of the resistance $R$. This $Ls$ is called the $s$-impedance of the inductor. Doing the same thing for a capacitor gives us $\tilde{V}_+ - \tilde{V}_- = \frac{1}{Cs}\tilde{I}$. This $\frac{1}{Cs}$ is the $s$-impedance of the capacitor. This means that we can treat a general linear circuit driven by exponential inputs as an $s$-dependent resistive circuit, and analyze the whole thing using 16A-style techniques.

This is useful because any sinusoidal input can be viewed as a sum of complex exponentials, and thus using

superposition, we can in principle understand the behavior of any linear circuit in response to sinusoidal inputs. For a sinusoidal function $u(t) = A\cos(\omega t + \phi) = (\frac{A}{2}e^{j\phi})e^{j\omega t} + (\frac{A}{2}e^{-j\phi})e^{-j\omega t}$, this can further be simplified to $u(t) = (\frac{A}{2}e^{j\phi})e^{j\omega t} + \overline{(\frac{A}{2}e^{j\phi})} \cdot \overline{e^{j\omega t}} = (\frac{A}{2}e^{j\phi})e^{j\omega t} + \overline{(\frac{A}{2}e^{j\phi})e^{j\omega t}}$. We noticed that the two terms are complex conjugates of each other and this makes sense since the result must be real. Furthermore, the two coefficients multiplying $e^{j\omega t}$ and $e^{-j\omega t}$ are also complex conjugates of each other. This latter fact turns out to save us considerable work — we don't have to actually do superposition in practice. Because of the properties of complex conjugacy in relationship to phasors and impedances, itl turns out that simply understanding the response to the $e^{j\omega t}$ part will tell us the whole story.

In discussion, we practiced some properties of complex conjugacy and did practice to understands the idea of impedances, as well as how these can be used to solve circuits. One of the key properties of complex conjugacy was the fact that the inverse $(\bar{M})^{-1}$ of the conjugate $\bar{M}$ of a matrix $M$ was the conjugate $\overline{M^{-1}}$ of its inverse $M^{-1}$. This thread was picked up in lecture to establish the main result. It suffices to define the phasors $\widetilde{A}$ corresponding to real sinusoidal inputs as being the coefficients of $e^{+j\omega t}$ in the complex exponential representations of those sinusoids. Then, by defining phasor variables for every circuit quantity, we can solve for them by solving a single system of linear equations. Because the impedances of capacitors and resistances are purely imaginary, this means that the system matrix corresponding to the coefficients of $e^{-j\omega t}$ is just the conjugate of the system matrix corresponding to $e^{+j\omega t}$.

Lecture then continued with the introduction of transfer functions — looking at the (complex) ratio of output quantity phasors to input quantity phasors. This is generally a frequency-dependent quantity since the capacitors and inductors have frequency-dependent impedances, and hence the $M$ matrix will have frequency dependent entries. The ratio is useful because it allows us to build more complicated filters out of simpler ones by simply cascading them together, as long as we put the appropriate buffers in between to prevent loading effects. The ratios will then multiply together. This desire to support composition by multiplication also led to the natural way to visualize transfer functions using Bode plots — log plots of the magnitude of the transfer function and linear plots of the phase (because phase by its nature is already logarithmic — it add up when you multiply complex numbers). The horizontal axis is typically also in log scale just to be able to encompass the many orders of magnitude of frequencies that are usually relevant.

We also saw and analyzed a simple RC lowpass filter. This was the circuit that inspired us to think about sinusoids as potentially a convenient set of inputs to look at in the first place. It is called a lowpass filter because the voltage across the capacitor tends to faithfully preserve low-frequency sinusoids while being much smaller for higher-frequency ones. The specific frequency which qualitatively seems to demarcate the boundary between low and high frequencies is $\frac{1}{RC}$ radians/sec. This is the frequency at which the magnitude response of the filter is $\frac{1}{\sqrt{2}}$ and is also where qualitatively, the Bode plot seems to begin curving significantly on the classic log-log Bode plot. This is also the frequency at which the magnitude of the (purely imaginary) impedance of the capacitor is the same as the magnitude of the (purely real) resistance $R$.

In discussion, we then practiced doing phasor analysis and saw how the log-scale of Bode plots facilitated composition.

In the final lecture, we did an example of filter design for a concrete problem —- removing 60Hz noise and 60kHz noise while preserving a desired signal that was around 600Hz. By cascading simple lowpass and highpass filters (with buffers in between), our goals could be achieved. Here, we talked about the simple highpass filter that considered the transfer function from the input to the voltage across the resistor in an RC circuit rather

than the voltage across a capacitor. We saw that the transfer function for the lowpass filter was $H_L(j\omega) = \frac{1}{1+j\frac{\omega}{\omega_0}}$ where $\omega_0 = \frac{1}{RC}$ is the cutoff frequency. Similarly, for the highpass filter, it was $H_H(j\omega) = \frac{1}{1-j\frac{\omega_0}{\omega}}$.

We also introduced a basic design paradigm for filters. First, convert all the relevant frequencies into radians per second. Second, figure out where you want to put the cutoff frequencies. These should be in-between the desired and undesired signals. When things are spaced closely together in frequency, the geometric means of frequencies are a reasonable choice. When they are far apart, then staying a factor of 10 away from the desired signals, but toward the undesired signals is usually a safe choice. For our example, things were close and so we stuck to the geometric means.

We saw that we couldn't get our desired rejection of noise (100 times for the 60Hz and 50 times for the 60kHz) without cascading filters. And ended up cascading two lowpass filters (cutoffs at 6kHz) and four highpass filters (cutoffs at 191Hz — between 60Hz and 600Hz). This worked for the case, but the need to cascade four highpass filters in this case already indicated that this methodology wouldn't easily support cases where the desired signals were somewhere close in frequency to undesirable interference.

For those cases, we needed to leverage the power of inductors. We saw that an inductor and a capacitor in series have a favored natural resonant frequency at which the series combination has zero impedance! This is because the impedances combine to be $j(L\omega - \frac{1}{C\omega})$. So at $\omega = \sqrt{\frac{1}{LC}}$, the series impedance is zero. That allowed us to use an RLC circuit to notch out an undesirable signal — say at 60Hz. Meanwhile, further away in frequency, the impedance is not zero for the LC combination and so by choosing a resistance $R$ that is low enough, the frequency selectivity around the natural frequency can be made as tight as desired.

In addition to notching out undesireable signals, such resonant filters can be used to select for certain desired ones as well. This is accomplished by taking the transfer function to be to the voltage drop across the resistor. When the LC are resonating, all the input voltage drops across the resistor. But once you are far-enough away from that natural frequency, the resistance $R$ will be dwarfed by the LC series impedance and consequently, most of the voltage drop will not be across the resistor.

In discussion, we will explore the resonant RLC filter in more detail.

At this point however, we have learned enough to be able to fulfill our Module 1 design objective — we can make circuits that will filter our signals for us and take us one step closer on the road to cyborgs!

# 2 Module 2

## 2.1 The Story So Far...

In Module 1: we learned how to analyze systems of differential equations with constant coefficients, with and without inputs. We saw how the eigenvalues (once we placed the system into vector matrix form) determined all the key behaviors. To solve, we basically took the system into a coordinate system that was as close to diagonal as possible and then reduced to a collection of scalar problems that could essentially be solved one at a time. For circuits with capacitors and inductors and other linear elements, this quickly grows unwieldy by hand and uninsightful (which eigenvalues matter most? where are they coming from?) — so we developed transfer function techniques by means of phasor domain. Phasor domain allowed us to sidestep the differential-equation nature of circuits and just deal with linear systems of equations. Using transfer functions and basic

building blocks like buffers, RC lowpass filters, CR highpass filters, LR highpass filters, RL lowpass filters, and LCR bandpass as well as RLC notch filters; we could filter out signals based on their frequencies. Here, the log-scale plots of magnitude and phase response help us understand the composition of building blocks.

In Module 2: we are studying what it takes to control systems of differential equations using their inputs to make them behave the way we want them to. In lab, you'll be making a robot car go straight. We won't be using phasor domain or transfer functions, but we will be building very strongly upon the differential equations, eigenvector, and coordinate-change philosophy that we have developed in Module 1. Mathematically, one of the new themes that will start to recur is recursion and arguments by induction. This is another piece of mathematical maturity that we want to build along the way.

We saw that because we wanted to be able to use computers and microcontrollers to control such systems, it is useful to be able to sample them every $\Delta$ seconds using piecewise-constant inputs and view everything as evolving in discrete time. Here, the next realization of the state that we will see is a function of the current state, the inputs, and some potential disturbance $\vec{w}(t)$. In this module, the disturbance becomes important because we don't necessarily fully trust our models to infinite decimal digits of accuracy, but still want to more or less make the real world systems do what we want. The resulting discrete-time model of the form: $\vec{x}_d(t+1) = A_d\vec{x}_d(t) + B_d\vec{u}(t) + \vec{w}(t)$ is often called a difference equation or a recurrence relation with discrete time $t$ that moves in increments of 1 time unit (corresponding to a real-world time-step of $\Delta$ seconds.).

We can then ask the kinds of basic questions that we asked at the beginning of 16A. When is it even possible to get the system we want to do what we want? Do we have enough control inputs? We saw that it was definitely possible for there to exist dimensions of the system that we were unable to influence through the choice of control inputs. And more intriguingly, by allowing ourselves a little bit of extra flexibility in time, we could sometimes get the state to go where we want even though there were more dimensions of state than we had dimensions of control input.

In discussion, we talked more about the sampling process that maps a continuous-time differential equation to a model that represents the discrete-time system that the digital-computer-based controller is effectively interacting with. In particular, we went through the vector matrix case where we need to change coordinates to get to a nice eigenbasis. This allowed us to convert to a discrete-time system. We also saw more about how we can control such discrete-time systems by means of an example where it is clear that multiple time steps allow us to reach any state that we want.

We developed the concept of controllability (grows out of the Segway problem from 16A — in a way this whole module grows out of that single problem) that told us that we don't necessarily need to have $n$ immediate degrees of freedom that we can independently actuate in order to eventually control an $n$-dimensional state. Instead, the known time-evolution dynamics of the system allows us to plan and use the past controls to help us. This could be checked by looking at the rank of a particular matrix: $[B, AB, A^2B, \ldots, A^{n-1}B]$ where we used induction arguments to establish this test, and systematic thinking to discover it. Induction arguments are an important part of mathematical maturity, and in Module 2, we will build up some facility with induction thinking only assuming the base that students should have from high-school or community college (see here and here for our view).

Our goal remains being able to get real world systems to reliably do what we want — but we face three obstacles:

1. How do we actually figure out the model for our system assuming it is linear?
2. How do we get it to do what we want without being overwhelmed by the disturbances?
3. How do we get and use a linear model for something if reality is actually nonlinear?

The System ID section below covers how we dealt with (1). Basically, we use least-squares to do system identification if the data is all trustworthy and clean, and can use OMP to help us discard outliers if the data is corrupted in some way.

We know that anything we estimate using least-squares isn't going to be exactly correct. So disturbances are a fact of life. All we can hope for is that they are small. So what matters? This is where we developed the concept of Bounded-Input Bounded-Output stability and began connecting it to the nature of the dynamics. The stability section below is where this story is built out more. We also figured out how to come up with a plan of controls that get us from where we start to where we want to end up in the required amount of time. As well as how we use the idea of linear approximation of nonlinear functions to figure out how to deal with systems that have nonlinear differential equations.

Throughout Module 2, Murat Arcak's reader from previous semester is an excellent reference. We will be releasing updated notes, but if students want to read ahead, that reader is a great place to start.

## 2.2 The System-ID Story

To keep things organized, recall from above that the first goal we need to achieve to do control is to actually learn a model from experimental data.

We showed that it was possible to use the least-squares techniques we learned in 16A more or less directly when we can directly see the traces of the states and know what inputs we applied. The unknowns we want to estimate are the parameters that sit inside the $A, B$ matrices. The approximate equations are obtained from the state and input traces. We just fit them with least squares. We discussed the scalar case first, and then saw that everything generalized to the case of vector states and vector inputs. (The discussion helped us understand by dwelling a bit on the seemingly odd case of a scalar state and a vector input.) The idea was to treat each observation of a scalar component of the state as an equation. This equation depends linearly on the past $n$-dimensional state $\vec{x}(t-1)$ and the past $k$-dimensional input $\vec{u}(t-1)$. We noticed that all these states could be collected into a giant $D = [X^T U^T]$ matrix where the $X$ matrix has one column for each time index going from $0$ to $m-1$ and similarly for the $U$ matrix. This $D$ matrix has $m$ rows corresponding to the length of the trace and has $n+k$ columns corresponding to the size $n$ of the state and the size $k$ of the input. Each individual dimension $\vec{x}[\ell](t)$ of the state $\vec{x}(t)$ has its own private row of $A$ and row of $B$ that constitutes the parameters $\vec{p}_\ell$ that govern its behavior. Similarly, we have state observations $\vec{s}_\ell = \begin{bmatrix} \vec{x}[\ell](1) \\ \vec{x}[\ell](2) \\ \vdots \\ \vec{x}[\ell](m) \end{bmatrix}$. So the approximate equation $D\vec{p}_\ell \approx \vec{s}_\ell$ is what we need to solve. We can put all the parameters together into a matrix $P = [\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_n]$ and similarly for the observations $S = [\vec{s}_1, \vec{s}_2, \cdots, \vec{s}_n]$. Then they can be estimated in one fell swoop using least-squares as $\widehat{P} = (D^T D)^{-1} D^T S$. We could also vectorize the $P$ matrix into a giant $\vec{p}$ parameter vector, do the same for the state observations $S$ into $\vec{s}$, and construct the relevant giant $D$ matrix to have a more standard $\widehat{p} = (D^T D)^{-1} D^T \vec{s}$.

Least-squares works to get estimates. However, least-squares is sensitive to "outliers" — observations that have been corrupted in a way that isn't just a small disturbance. In the real world, outliers are a fact of life. There are some reasonable heuristics that people can use to reject outliers like "if a point is just too big then it is probably an outlier." But what is "too big" and how can we turn this into an algorithm that doesn't require eyeballing the data by a person. Humans can eyeball data in two dimensions, maybe (using tricks) up a few dimensions more than that. For higher dimensions, we need a more systematic way to seek out and eliminate outliers from our data to avoid contaminating our estimates. To proceed systematically, we introduced variables $f_i$ that represented whether the equation $i$ was "fake" or not. By "fake," we meant that we deem it an outlier. We observed that by adding the $i$-th standard basis element as a column in our $D$ matrix — corresponding to the variable $f_i$ — as far as least-squares was concerned, it was as though the $i$-th equation no longer existed. So, for all the $f_i$, this just means that we augment the $D$ matrix to be $[D, I]$ where the $I$ is an $m \times m$ identity. What did we want to model about outliers? That there are hopefully only a few of them.

So, how do we solve a problem in which we want to find the best fit using the fewest variables? This is exactly what we had seen in 16A with OMP! Orthogonal matching pursuit is an intuitive algorithm for doing this functions in a greedy manner. It picks variables one at a time and adds their corresponding feature columns to a working set of selected columns. Then, it does a least-squares fit using just those selected columns. It computes the residual from that fit, and uses that to pick the next variable/column. Explicitly, when facing a problem " $A\vec{x} \approx \vec{y}$" where $A = [\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_d]$,

Initialize the residual $\vec{r} = \vec{y}$, initialize the selected columns $A_{select} = []$ to be an empty matrix, and set the number of selected columns to be zero $\ell = 0$. Greedily pick the column most correlated with the residual. i.e. $i^* = \arg\max_i |\vec{a}_i^T \vec{r}|$ Add this column to our selected set: $A_{select} = [A_{select}, \vec{a}_{i^*}]$, and increment $\ell = \ell + 1$. Compute the least square fit and the new residual: $\hat{\vec{y}} = A_{select}(A_{select}^T A_{select})^{-1} A_{select}^T \vec{y}$ and so $\vec{r} = \vec{y} - \hat{\vec{y}}$. Stop the loop if a stopping condition is met, otherwise goto 2 Compute the final estimate using the selected columns.

The kinds of stopping conditions we had seen in 16A were to stop if the residual $\vec{r}$ got small enough or if $\ell$ had reached our budget for the number of columns to select.

In the context of outlier removal, the step of looking at $|\vec{a}_i^T \vec{r}|$ has a particularly simple and intuitive meaning when the $\vec{a}_i$ corresponds to one of the columns of the identity — it just means look at $|\vec{r}[i]|$. In other words, find the equation that is the least happy and declare it to be an outlier. When we have plenty of data and expect only a small fraction of outliers, it is not that bad to throw out some good points along with all the outlier. The rest of our points will still provide least squares with enough information to get a good estimate. So a stopping condition setting a generous limit on $\ell$ together with an estimate of what clean data should look like in terms of the residual size will work just fine. There are other approaches that can be more adaptive to the fraction of outliers that actually exist in the data, but we didn't talk about them. The $A\vec{x} \approx \vec{y}$ approach to using OMP with augmentation with "identity features" assumes that are dealing with all the parameters in one big vector, but it can be naturally extended to when we have the parallel structure represented by when $\widehat{P} = (D^T D)^{-1} D^T S$ is the least-square solution. This can be treated like $n$ parallel OMPs where if any one of the states votes for a particular equation to be an outlier, the relevant feature gets added to the sibling OMPs as well.

The OMP approach above is fine, but it has a computational issue in terms of running time — as you all saw in 16A. When the $A$ matrix is very big, it starts taking a long time to run. Looking at the algorithm above, we quickly zoomed in on line 4 and in particular, the need to compute $A_{select}^T A_{select})^{-1}$. This is an $\ell \times \ell$ matrix and so it is getting bigger and bigger as the algorithm runs. Taking an inverse involves running Gaussian elimination

and so generically takes about $O(\ell^3)$ time where the constant in front of $\ell^3$ depends on the details. Why this? Because we make $\ell$ steps downward, and each involves looking at almost the whole matrix which is $\ell^2$ in size. To make this faster, we wished that somehow it was just taking the inverse of the identity matrix, which takes no work at all.

So, how could we make a counterpart of $A_{select}^T A_{select}$ be the identity? We also have to preserve the point of $A_{select}$ — which is the span of the columns of $A_{select}$. After all, what line 4 is doing is projecting $\vec{y}$ onto that span. We realized that we wanted a matrix $Q_{select} = [\vec{q}_1, \vec{q}_2, \cdots, \vec{q}_\ell]$ to have the property that span$(Q_{select})$ = span$(A_{select})$ and $Q_{select}^T Q_{select} = I$. This second condition means that the vectors $\vec{q}_i$ need to be orthonormal. That is: $\vec{q}_i^T \vec{q}_i = 1$ (i.e. $\|\vec{q}_i\| = 1$) and $\vec{q}_i^T \vec{q}_j = 0$ if $i \neq j$. How can we do this?

Well, we could just start at the beginning and proceed systematically. $\vec{q}_1 = \frac{\vec{a}_1}{\|\vec{a}_1\|}$. And then, we could leverage what we know about projections and least-squares from 16A to find orthogonal vectors. We know that the residual after a projection is always orthogonal to the subspace being projected upon. Consequently, we can recursively define $\vec{q}_i = \frac{\vec{a}_i - \sum_{j=1}^{i-1} \vec{q}_j (\vec{q}_j^T \vec{a}_i)}{\|\vec{a}_i - \sum_{j=1}^{i-1} \vec{q}_j (\vec{q}_j^T \vec{a}_i)\|}$. This has norm 1 by construction and it is orthogonal to all the previous $\vec{q}_j$ because it is proportional to the residue that remains after projecting $\vec{a}_i$ onto the subspace spanned by them. This collection also preserves the same span because every column is just a linear combination of the original vectors and this map is clearly invertible. It turns out that this very natural process that we "discovered for ourselves" has a name: Gram-Schmidt Orthonormalization.

With this, we could write a much faster OMP algorithm that avoids having to take a big inverse.

Initialize the residual $\vec{r} = \vec{y}$, initialize the selected subspace $Q_{select} = []$ to be an empty matrix, and set the number of selected columns to be zero $\ell = 0$. Greedily pick the column most correlated with the residual. i.e. $i^* = \arg\max_i |\vec{a}_i^T \vec{r}|$ Add the orthonormalized column to our selected set: $\vec{q}_{\ell+1} = \frac{\vec{a}_{i^*} - \sum_{j=1}^{\ell} \vec{q}_j (\vec{q}_j^T \vec{a}_{i^*})}{\|\vec{a}_{i^*} - \sum_{j=1}^{\ell} \vec{q}_j (\vec{q}_j^T \vec{a}_{i^*})\|}$ $Q_{select} = [Q_{select}, \vec{q}_{\ell+1}]$, and increment $\ell = \ell + 1$. Compute the new residual: $\vec{r} = \vec{r} - \vec{q}_\ell (\vec{q}_\ell^T \vec{r})$. Stop the loop if a stopping condition is met, otherwise goto 2 Compute the final estimate using the selected columns.

Here, we got a happy bonus of also being able to speed up our computation of the new residual. Because the selected $Q_{select}$ is always orthonormal, projecting onto it just becomes easy and decomposes along the different directions within it. Since we are just adding one more direction, the update to the residual is easy to do.

We know that anything we estimate using least-squares isn't going to be exactly correct. So disturbances are a fact of life. All we can hope for is that they are small. So what matters? This is where we developed the concept of Bounded-Input Bounded-Output stability and began connecting it to the nature of the dynamics. That story will be continued in another post.

## 2.3 The Stability and Feedback Control Story

Having a way to identify a system from data, we know that the model is going to be imperfect — we can't trust it to infinite decimal digits of accuracy. We group all the unknown parts and inaccuracies into the disturbance term $\vec{w}(t)$ in our discrete-time models $\vec{x}(t+1) = A\vec{x}(t) + B\vec{u}(t) + \vec{w}(t)$ or our continuous-time differential-equation models $\frac{d}{dt}\vec{x}(t) = A\vec{x}(t) + B\vec{u}(t) + \vec{w}(t)$. The hope is that these disturbances are small. What we want to make sure of is that these disturbances don't make our system behave in a way that is drastically different from how we want it to behave.

How do we want our system to behave? One most basic condition is stability. We don't want our system to

explode. Literally. When aspects of real-world systems get to extreme values, bad things tend to happen. We are particularly concerned with runaway chain reactions in which even without any disturbances, the system does bad things. This is what the concept of stability captures.

Consider a scalar model in discrete time $x(t+1) = \lambda x(t) + w(t)$. Even without a $w(t)$, this will blow up from a nonzero initial condition if $|\lambda| > 1$. Meanwhile when $|\lambda| < 1$, such a system would exponentially approach zero from any nonzero initial condition. Similarly in a scalar continuous-time differential-equation model $\frac{d}{dt}x(t) = \lambda x(t) + w(t)$, if the real part of $\lambda$ was strictly positive, a nonzero initial condition would make the system blow up. If the real part of $\lambda$ is strictly negative, then a nonzero initial condition would be driven exponentially to zero in time. The case of $|\lambda| = 1$ in discrete time and $Re(\lambda) = 0$ in continuous time are more ambiguous. They don't seem to blow up, but they aren't going to zero either.

This is where we developed the concept of Bounded-Input Bounded-Output stability and connected it to the nature of the dynamics. If the dynamics of the system tends to blow up, then even small disturbances will eventually have a big effect. If the dynamics of the system tends to go to zero, then even the net effect of a constant stream of small disturbances will also be small. It turns out that this definition clearly says that $|\lambda| = 1$ in the discrete-time case corresponds to an unstable system because a bounded disturbance can have an unbounded cumulative effect as time goes on. A steady push can move the system state quite a bit because the effect of the disturbance is cumulative. Similarly for the continuous time case of $\lambda = 0$.

For vector systems, BIBO stability is connected to the eigenvalues of the system $A$. We claimed that if all the eigenvalues of $A$ are stable, then the system is stable. Meanwhile, if even one of them is unstable, the system is unstable. For the case of an $A$ with an eigenbasis, this result can be shown by changing coordinates to one in which we have parallel uncoupled differential equations. However, we know that $A$ need not always have an eigenbasis. We will come back to that case.

So, what could we do if the eigenvalues were such that the system would be unstable? Is all hope lost?

It turned out that we could actually choose controls using feedback so that the entire system (including the effect of the feedback) behaves differently. We called this the closed-loop system behavior and saw that it was determined not by the eigenvalues of $A$ but by the eigenvalues of $(A + BF)$ where $F$ was the linear feedback law's dependence on the current state — i.e. $\vec{u}(t) = F\vec{x}(t)$. When $B$ is invertible, it is clear that by choosing $F$ we can make the closed-loop dynamics whatever we'd like. The interesting thing is when $B$ is not invertible — the most dramatic case of which is when $B$ is just a single column $\vec{b}$. When we have fewer degrees of freedom of control than we have state variables within the state, that is called the "underactuated case." We looked at the 2x2 case and saw by example that when the system had $A, B$ such that they were controllable, it turned out that it was always possible to place the closed-loop eigenvalues to be wherever we wanted. This allows us to reject disturbances in the sense that small disturbances will only slightly impact the state.

Such feedback control is incredibly example in practice because it allows us to work with imperfect models and still be successful. Later in the course, we will prove the statement for the case of general matrices $A$ and controllable $\vec{b}$.

In discussion, you saw more details about eigenvalue placement.

In lecture, we returned to the question of stability and recognized that the argument that we have so far only really works for diagonalizable matrices. This also reminded us of a loose end from Module 1 — if we have a system of differential equations that doesn't have a full eigenbasis, how do we solve it? The critically-damped

case of an RLC circuit pointed the way. That was a 2x2 case, but does the same thing happen in general? Can we find such coordinate systems for general $A$ matrices — coordinate systems in which the matrix becomes upper-triangular. To find this coordinate system, we proceeded systematically. We found an eigenvector, extended it using Gram-Schmidt to get an orthonormal basis, and then recursed into a smaller submatrix. This allowed us to find an orthonormal coordinate system (for matrices that have real eigenvalues) that takes the matrix into an upper-triangular form with the eigenvalues along the diagonal, and zeros below it.

Discussion was postponed due to the PGE outage.

We finished up the story of upper-triangularization in three ways. (1) We showed that a coordinate transformation does not change the characteristic polynomial and hence the eigenvalues of a matrix, and that an upper-triangular matrix indeed has the eigenvalues along the diagonal. (2) We explicitly wrote out the inductive proof that the for an upper-triangular system, it is BIBO stable if all the eigenvalues are stable. Here, we used strong induction in a natural way. (3) We considered the case of symmetric matrices and showed that if a symmetric real matrix has all real eigenvalues, then when we upper-triangularize it by an orthonormal coordinate change, we are in fact diagonalizing it.

This led us to naturally wonder if real symmetric matrices always have real eigenvalues and we found that indeed they do — all eigenvalues must be real because the symmetry implies that they must equal their own complex conjugates. This concluded the proof of what is called the spectral theorem for real symmetric matrices.

From there, we began considering the problem of planning for control. Here, we saw that we have many choices and began introducing the "Principle of Least Action" to try and find natural plans by minimizing the size of the control inputs that we apply.

In discussion, you went through the upper-triangularization proof to make sure that you understand it. Because this idea of upper-triangularization really is the differential-equation (or difference-equation) counterpart to Gaussian Elimination for systems of linear equations. It allows you to conceptually solve the problem by upper-triangularizing, and then doing back-substitution. There's a homework about this as well.

The next lecture used the problem of finding the minimum energy control to derive the SVD. In this summary, I will give you the same story slightly differently to emphasize the SVD derivation.

In general, for any matrix $M$, you know from 16A that if you understand what $M$ does to basis vectors, you know what it does for anything. Writing this fact out in mathematical notation, we see that if $\vec{x} = \sum_i \beta_i \vec{b}_i$ and the collection $B = [\vec{b}_1, \ldots, \vec{b}_\ell]$ is a basis for the $\ell$-dimensional space that $\vec{x}$ lives in, then indeed $M\vec{x} = \sum_{i=1}^{\ell} \beta_i (M\vec{b}_i)$. Here, we can also express things as $\vec{x} = B\vec{\beta}$ if we collect $\vec{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_\ell \end{bmatrix}$ using vector notation. This is what matrix-vector multiplication means, as you know from 16A.

We know that since $\vec{x} = B\vec{\beta}$, that $\vec{\beta} = B^{-1}\vec{x}$. This holds even without orthonormality. When the basis $B$ is orthonormal, then we have an additional nice property. Orthonormality tells us further that since $B^T B = I$, that

$B^{-1} = B^T$. Consequently, $\vec{\beta} = B^T\vec{x}$ and since $B^T = \begin{bmatrix} \vec{b}_1^T \\ \vec{b}_2^T \\ \vdots \\ \vec{b}_i^T \\ \vdots \\ \vec{b}_\ell^T \end{bmatrix}$, this means that $\beta_i$ (the $i$-th coordinate in $\vec{\beta}$) can be directly evaluated as $\beta_i = \vec{b}_i^T\vec{x}$.

For this reason, we can further say that $M\vec{x} = \sum_{i=1}^{\ell}(M\vec{b}_i)\beta_i = \sum_i(M\vec{b}_i)\vec{b}_i^T\vec{x}$. Drawing parentheses, we can see that $M\vec{x} = (\sum_{i=1}^{\ell}(M\vec{b}_i)\vec{b}_i^T)\vec{x}$, which tells us that the matrix $M = \sum_{i=1}^{\ell}(M\vec{b}_i)\vec{b}_i^T$.

So far, none of this has relied upon anything about the basis $B$ except its orthonormality. When we further know that each of the $\vec{b}_i$ from $i = k+1, \ldots, \ell$ are in the nullspace of $M$, then we know that $M\vec{b}_i = \vec{0}$ for those. This lets us simplify the sum by dropping all these zero terms.

So, the further fact that the basis $B$ has a basis for the nullspace of $M$ as its last $\ell - k$ vectors tells us that in fact, $M = \sum_{i=1}^{k}(M\vec{b}_i)\vec{b}_i^T$.

Up to this point, we have only used the orthonormality of $B$ and the fact that the nullspace of $M$ has its basis as the last $(\ell - k)$ vectors in $B$.

Because $B$ was the eigenbasis of $M^TM$ with the eigenvectors arranged in decreasing order according to the corresponding eigenvalue, the spectral theorem for real symmetric matrices told us that $B$ is orthonormal. We further knew that the smallest eigenvalue of $M^TM$ has to be $0$ and so the entire nullspace of $M^TM$ was spanned by the last eigenvectors in $B$. We had separately established that the nullspace of $M^TM$ is the same as the nullspace of $M$. So, this told us that the $B$ that we found in this way satisfied the properties above. However, it also gave us an additional property.

Namely that $M\vec{b}_i$ was orthogonal to $M\vec{b}_j$ whenever $i \neq j$. This was seen because $(M\vec{b}_i)^T(M\vec{b}_j) = \vec{b}_i^TM^TM\vec{b}_j = \vec{b}_i^T(M^TM\vec{b}_j) = \vec{b}_i^T\lambda_j\vec{b}_j = \lambda_j\vec{b}_i^T\vec{b}_j = \begin{cases} \lambda_i \text{ if } i = j \\ 0 \text{ otherwise} \end{cases}$. Here, the $\lambda_i$ is the $i$-th eigenvalue (in descending order) of $M^TM$. Because of the nature of $M^TM$, it must be the case that $\lambda_i = \|M\vec{b}_i\|^2$. This is why we are free to take the square-root of $\lambda_i$.

Because of this property, we were able to define $\vec{w}_i = \frac{M\vec{b}_i}{\sqrt{\lambda_i}} = \frac{M\vec{b}_i}{\|M\vec{b}_i\|}$ and know that $\|\vec{w}_i\| = 1$ and furthermore $\vec{w}_i^T\vec{w}_j = 0$ whenever $i \neq j$. In other words, the $\{\vec{w}_i\}$ are orthonormal, and $M\vec{b}_i = \sqrt{\lambda_i}\vec{w}_i$.

Plugging this back in, we get that $M = \sum_{i=1}^{k}(M\vec{b}_i)\vec{b}_i^T = \sum_{i=1}^{k}\sqrt{\lambda_i}\vec{w}_i\vec{b}_i^T$. This is precisely the outer-product form of the SVD. The $\vec{w}_i\vec{b}_i^T$ are the outer-products, and the $\sqrt{\lambda_i}$ are the non-zero singular values.

From here, it is a straight shot to see that if we define $W = [\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_k]$ as a matrix, that $W\begin{bmatrix} \sqrt{\lambda_1} & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \sqrt{\lambda_k} \end{bmatrix} = [\sqrt{\lambda_1}\vec{w}_1, \sqrt{\lambda_2}\vec{w}_2, \ldots, \sqrt{\lambda_k}\vec{w}_k]$. This means that $M = W\begin{bmatrix} \sqrt{\lambda_1} & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \sqrt{\lambda_k} \end{bmatrix}\begin{bmatrix} \vec{b}_1^T \\ \vec{b}_2^T \\ \vdots \\ \vec{b}_k^T \end{bmatrix}$. This is what is called the

"compact form" of the SVD.

To get the full form of the SVD, we just want the full $B^T$ matrix at the end. So, we want to extend the matrix in the middle to let us do this with no negative consequences. That can be done by padding it with zeros. So all the extra nullspace of $M$ basis vectors just get mapped to zero. In other words:

$$M = W \left[\begin{array}{cccc|ccc} \sqrt{\lambda_1} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \sqrt{\lambda_k} & 0 & \cdots & 0 \end{array}\right] \left[\begin{array}{c} \vec{b}_1^T \\ \vec{b}_2^T \\ \vdots \\ \vec{b}_k^T \\ \vec{b}_{k+1}^T \\ \vdots \\ \vec{b}_\ell^T \end{array}\right] = W \left[\begin{array}{cccc|ccc} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \sigma_k & 0 & \cdots & 0 \end{array}\right] B^T = W\Sigma B^T.$$

That's the full SVD. $W$ is orthonormal. $B$ is orthonormal. The $\sigma_i = \sqrt{\lambda_i}$ are the singular values and $\Sigma$ is the diagonal matrix that has the same shape as $M$ and has the $\sigma_i$ on the diagonal.

In lecture, we did all this and we used the matrix $C_{100} = [\vec{b}, A\vec{b}, A^2\vec{b}, \cdots, A^{\ell-1}\vec{b}]$ for the $M$ matrix above. $\ell = 100$ and $k = 10$ because we had a 10-dimensional state governed by $\vec{x}(t+1) = A\vec{x}(t) + \vec{b}u(t)$. In lecture, we used the notation $V$ to denote the eigenbasis $B$ above. The least-effort sequence of controls that will take us from state $\vec{x}_0$ to $\vec{x}_d$ is the $\vec{u} = \begin{bmatrix} u(99) \\ u(98) \\ \vdots \\ u(1) \\ u(0) \end{bmatrix}$ is the solution to $\arg\min_{C_{100}\vec{u} = \vec{x}_d - A^{100}\vec{x}_0} \|\vec{u}\|$. We realized that the optimal $\vec{u}^*$ cannot have any component that was in the nullspace of $C_{100}$. That is what motivated finding an orthonormal coordinate system $V$ that had a basis for the nullspace of $C_{100}$ as the last 90 vectors in $V$. To find such an orthonormal basis, we considered the symmetric matrix $C_{100}^T C_{100}$ that had the same nullspace as $C_{100}$. (Why is the nullspace the same? Because if $C_{100}\vec{v} = \vec{0}$, certainly we know that $C_{100}^T C_{100}\vec{v} = \vec{0}$. And more interestingly, if $C_{100}^T C_{100}\vec{v} = \vec{0}$, then we know that $\vec{v}^T C_{100}^T C_{100}\vec{v} = 0$ and thus $(C_{100}\vec{v})^T(C_{100}\vec{v}) = 0$ which means that $\|C_{100}\vec{v}\|^2 = 0$ which can only happen if $C_{100}\vec{v} = \vec{0}$ since the sum of squares can be zero only if all the real numbers being squared are themselves zero. So, the nullspaces are the same.) Once we did all this, we could find the optimal solution $\vec{u}^* = \sum_{i=1}^{k} \vec{v}_i \frac{\vec{w}_i^T}{\sqrt{\lambda_i}}(\vec{x}_d - A^{100}\vec{x}_0)$. The more detailed description above is designed to help students who might have gotten a bit lost about which properties we are using where.

Next lecture, we will continue. We will mention that while we did this for a wide matrix, the same decomposition is possible for a tall matrix. Just take the transpose to get a wide matrix. Repeat the process above to get the SVD. And then transpose the result. The end-result is the same. We have written the matrix as a product of an orthonormal matrix times a diagonal matrix of the same shape as the original matrix times another orthonormal matrix.

After an interlude into leveraging the SVD to do PCA and to help support classification, we returned to the control story to tie up a couple of loose ends: how do we do feedback control of systems with state dimension greater than 2; and how can we deal with nonlinear systems? For the first, when the system had $A, B$ such that they were controllable, it turned out that it was always possible to place the closed-loop eigenvalues wherever we wanted. This allows us to reject disturbances in the sense that small disturbances will only slightly impact the state. We could also do this systematically by choosing coordinates so that systems were in control-

lable canonical form — a form in which the scalar-input system basically behaves like a glorified higher-order recurrence relation in discrete time (and a higher-order differential equation in continuous time). The conceptual keys to doing this were: (a) recognizing that this "higher-order" relationship plays the role of the simplest nontrivial case of something be obviously controllable; (b) expressing this higher-order type relationships into vector-valued states so that we could see the desired matrix structure; (c) seeing that these matrices clearly had eigenvectors that must be $\begin{bmatrix} 1 \\ \lambda \\ \lambda^2 \\ \vdots \\ \lambda^{n-2} \\ \lambda^{n-1} \end{bmatrix}$; (d) seeing that the coefficients of the recurrence relation must also essentially be the coefficients of a polynomial satisfied by the eigenvalues; (e) seeing that using the natural basis from the controllability matrix must in fact map us to the transpose of the form that we want; (f) leveraging the pull of this from both sides to find the desired transformation.

After gaining a pretty solid understanding of the key ideas for linear system modeling, and having seen the idea of approximation introduced via our discussion of PCA and keeping the most important subspace, we decided to explore an aspect of approximation that didn't involve learning from data. Namely, how to deal with nonlinear differential equations by pretending that they are linear ones.

Given $\frac{d}{dt}\vec{x}(t) = \vec{f}(\vec{x}(t), \vec{u}(t))$ as our nominal model for the system with a potentially nonlinear vector-valued function $\vec{f}(\vec{x}, \vec{u})$ of vector state $\vec{x}$ and vector controls $\vec{u}$, we want to approximate this using a linear model.

Here, we were inspired by what we had learning in basic calculus courses in the concept of Taylor series — the goal is to approximate the nonlinear function $\vec{f}$ in a local neighborhood by a linear function. Here, there are a couple of key questions: what kind of neighborhood do we want to look in and what does the approximation look like. We tackled the second issue first. Here, across lecture and discussion, we established that the approximation of $\vec{f}$ around any particular nominal point $(\vec{x}_n, \vec{u}_n)$ looks like: $\vec{f}(\vec{x}_n + \vec{\delta x}, \vec{u}_n + \vec{\delta u}) = \vec{f}(\vec{x}_n, \vec{u}_n) + [\frac{\partial}{\partial \vec{x}}\vec{f}|_{(\vec{x}_n, \vec{u}_n)}]\vec{\delta x} + [\frac{\partial}{\partial \vec{u}}\vec{f}|_{(\vec{x}_n, \vec{u}_n)}]\vec{\delta u} + \vec{w}$ where $\vec{w}$ is the approximation error thought of as a disturbance. Here, the $[\frac{\partial}{\partial \vec{x}}\vec{f}|_{(\vec{x}_n, \vec{u}_n)}]$ is a matrix filled with the partial derivatives $\frac{\partial}{\partial x[j]} f[i](\vec{x}, \vec{u})$ evaluated at $(\vec{x}_n, \vec{u}_n)$ and similarly $[\frac{\partial}{\partial \vec{u}}\vec{f}|_{(\vec{x}_n, \vec{u}_n)}]$ is a matrix filled with the partial derivatives $\frac{\partial}{\partial u[j]} f[i](\vec{x}, \vec{u})$ evaluated at $(\vec{x}_n, \vec{u}_n)$.

For the nonlinear differential equation $\frac{d}{dt}\vec{x}(t) = \vec{f}(\vec{x}(t), \vec{u}(t))$, if we apply this kind of expansion around a nominal controlled trajectory $(\vec{x}_n(t), \vec{u}_n(t))$ — meaning that this pair is a valid solution to the differential equation (namely $\frac{d}{dt}\vec{x}_n(t) = \vec{f}(\vec{x}_n(t), \vec{u}_n(t))$ for all $t$) — then we get a linear approximate differential equation by changing coordinates to $\vec{x}(t) = \vec{x}_n(t) + \vec{\delta x}(t)$ and $\vec{u}(t) = \vec{u}_n(t) + \vec{\delta u}(t)$. The deviations $\vec{\delta x}(t)$ are governed by the differential equation: $\frac{d}{dt}\vec{\delta x}(t) = [\frac{\partial}{\partial \vec{x}}\vec{f}|_{(\vec{x}_n(t), \vec{u}_n(t))}]\vec{\delta x}(t) + [\frac{\partial}{\partial \vec{u}}\vec{f}|_{(\vec{x}_n(t), \vec{u}_n(t))}]\vec{\delta u}(t) + \vec{w}(t)$. Here, the $[\frac{\partial}{\partial \vec{x}}\vec{f}|_{(\vec{x}_n(t), \vec{u}_n(t))}]$ is playing the role of the traditional square $A$ matrix in a linear control problem and $[\frac{\partial}{\partial \vec{u}}\vec{f}|_{(\vec{x}_n(t), \vec{u}_n(t))}]$ is playing the role of the traditional $B$ matrix. If the nominal trajectory $(\vec{x}_n(t), \vec{u}_n(t))$ does not stay in one place, then these matrices can be time-varying. We didn't deal with that case in lecture, but we can always approximate them again as being piecewise-constant and fold the resulting approximation error into the $\vec{w}(t)$ term. This will be fine as long as they vary slowly enough.

The case that we did focus on was where the trajectory $(\vec{x}_n(t), \vec{u}_n(t))$ was constant so $\vec{x}_n(t) = \vec{x}_0$ and $\vec{u}_n(t) = \vec{u}_0$ for all time. These kinds of solutions are referred to as DC operating points, stationary points, or equilibria. They have the advantage of allowing us to linearize around them without having to worry about any time-

variation. Based on the eigenvalues of the resulting $A$ matrices, such equilibria can be stable or unstable. In either case, we can use the feedback control techniques we know to adjust the behavior to make such points more stable. To find such DC operating points, we "merely" have to solve the nonlinear system of equations $\vec{f}(\vec{x}_n, \vec{u}_n) = \vec{0}$. Usually we do this by first picking a desired $\vec{x}_n$ or a desired $\vec{u}_n$ and then solving. There can be multiple solutions or no solutions or a unique solution. Graphical methods (as seen in discussion) or iterative methods (like the Newton's method approach you will see on the homework — which is interesting because you again use the principle of linearization iteratively) can be used to find such solutions — and in some cases they can be found directly based on our knowledge of the nonlinear functions involved (you will see such an example in the homework).

As mentioned in lecture, linearization into small neighborhoods can also be used for planning trajectories. However to do this properly requires us to use ideas from our sister course 61B and so it is out of scope. Essentially, we cover the space (of $\vec{x}, \vec{u}$ so this can quickly get intractable if the dimensions are high) with many small overlapping balls such that linearization is a pretty good model within that ball. Then, we pick some points within the overlaps between balls as well as points that lie along the "zero-control" (just stick to the nominal $\vec{u}$ for that ball) trajectories and land in the overlap with another ball. Then, we use sampling of time, piecewise-constant inputs, and standard linear-control minimum-norm ideas to plan trajectories within that ball that get from one such point to another. In these balls, there is a non-zero "drift" term that comes from the constant part of the $\vec{f}$ evaluated around the center of the ball, and so sometimes the cost required to go somewhere else is quite large, but we can always at least follow the "zero-control" trajectories. We then make a big graph where the vertices are these points and the edges correspond to when we can get from one point to another within a single ball. Each edge is labeled with a cost that corresponds to how much it costs actual in control cost to make that little hop. At this point, graph-shortest path can be used to figure out how to get from one ball to another ball, even if those balls are far apart from each other. If desired, the graph-path can then be used to initialize an iterative process of linearization and global min-norm planning that further refines the nominal trajectory to be more natural. This overall approach decomposes dealing with the essential nonlinearity of the real world problem into three levels:

1. The highest-level of point sampling and graphs deals with the global picture for planning
2. The intermediate level of linearization around the balls deals with the nominal local details for local planning
3. The lowest level of feedback control around the resulting nominal path deals with the impact of the approximations used in the above levels.

This is the general spirit of how engineering approximations are used — an approximation is used because we know that another part of our overall system will be able to deal with the imperfections that our approximation has introduced. In modern systems, we also allow for constant replanning using the graph-level and the intermediate level. Everything operates on distinct time-scales — the closed-loop feedback system at level 3 operates very fast; and the higher-level replannings occur less frequently but can still be many times per second given the advances in modern computational capabilities.

# 3 Module 3

## 3.1 Learning and Interpolation

Dear students,

This module is a bit different in that we will be dithering in and out of it instead of doing a smooth or sharp transition. (Remember that by contrast, the transition to learning in 16A was more of a clean pivot that could be accomplished the instant 16A's module 2 had built enough problem-solving and design maturity.) The reason is twofold. First, learning is fundamentally (as well as historically) an intellectual outgrowth of control. Consequently, going back and forth is very natural. Second, it is natural to build technical tools in an order that respects the growth of maturity that is needed to conceive of those tools.

We'll keep to the standard 16B motivational story: we want to be able to make cyborgs or cybernetic implants. There are a couple of natural points where learning from data enters the story: we need to learn the actual dynamics of the actual electromechanical components of our cyborg (like our robot arms, etc...). Here, the system-id story is very natural. Not unsurprisingly, we leaned heavily upon the learning module (Module 3) of 16A where you learned about projections, least-squares, and orthogonal matching pursuit. The second natural entry point for learning is in processing the signals recorded from the brain to figure out what goal we want to achieve with our robot arm. It turns out that to do this in an intelligent way, we will need the same tools (namely the SVD in particular) that helped us do planning for control. However, we will need to use these in support of what is referred to as "classification" — going from continuous measurements to a discrete choice.

Notice that you have already seen the idea of making a discrete choice in 16A when you were using the idea of maximizing correlation to figure out which discrete delay best represents the received microphone signal from a given beacon. Discrete choices were also what were involved in OMP where you used maximization of inner products to pick which signals were hypothesized to be present in the real world. In the HW, we first expose you to a classification approach that is based on that.

On the Nov 5th lecture, we introduced the big picture of the steps needed to process the signals recorded in the brain. After whatever analog circuit filtering is needed to remove interference, the signal is sampled and enters the computer. Because each electrode can pick up neural activity from several neurons in the vicinity, the first thing to do is to figure out which neuron is doing what. For biological reasons involving the electrochemistry of the neurons themselves, they seem to have spiking behavior: any particular neuron might decide to "fire" and when it does, it will tend to have its own somewhat idiosyncratic waveform shape for its spike. (This is from a combination of things including its physical relationship to the electrode, as well as its own nature.) The hope is that by looking at the shape of the waveform that is picked up, we can figure out which neuron is firing. To do this, the sampled waveform is first processed with a "spike detector" that cuts the waveform into standard sized pieces (say a contiguous block of 100 samples) that are aligned to the shape of the spikes. Before classifying the waveform into which neuron it comes from, however, we introduce one more step of processing — dimensionality reduction aka feature extraction. This uses relatively simple processing to reduce the 100 dimensional vectors into something that is much smaller — say 2-4 dimensional. This can be viewed as an informative summary that distills as much of the relevant information as possible. These lower-dimensional feature vectors are then used to classify the segment in terms of which neuron it corresponds to. The resulting pattern of neuron firings is then used to infer the person's goals, etc.

When a brain-machine-interface is in operation, these operations are happening in real time. But how do we decide how to do feature extraction? How do we do classification? For classification, you already know a basic approach from 16A: pick the class for whom its template looks the most like the received vector we want to classify. This is also what you did in the HW on classification of sinusoids. The harder question is how to do dimensionality reduction or feature extraction. Here, we will follow the same overall paradigm that we established for system identification: we will record a bunch of traces of actual data and use them to figure out how to do feature extraction. How are we going to do feature extraction? We will project onto an appropriate low-dimensional subspace. So the goal is to extract the best subspace to project onto given a bunch of data.

This goal of finding a subspace is what Principal Component Analysis (PCA) is useful for. The SVD plays a critical role here. In brief, the approach is as follows. Collect your data into a matrix. How you do so depends on the interpretation of the data itself. If the data comes in as a bag of vectors, you are free to just collect the vectors into a matrix, column by column. Once you have a matrix $M$ of vectors whose related important subspaces you want to find, you take the SVD $M = \sum_i \sigma_i \vec{u}_i \vec{v}_i^T$. Because each of the $\vec{u}_i$ and $\vec{v}_i$ are unit length, it is the $\sigma_i$ that tell you the relative sizes of these different matrices. The intuitive idea is that whenever you want to do an approximation and have a sum, the natural thing to do is just to keep the biggest terms in the sum. This is what you have done when you studied Taylor series, etc., and that is what we are doing in PCA. We just approximate our data $M$ by the $k$ biggest terms: $\widehat{M} = \sum_{i=1}^{k} \sigma_i \vec{u}_i \vec{v}_i^T$. This is because we chose to order the $\sigma_i$ in decreasing order. (Recall that the $\sigma_i = \sqrt{\lambda_i}$ are the square roots of the eigenvalues of the symmetric matrix $M^T M$.)

So, now that we've approximated the matrix $M$, what is our choice of the subspace we want to potentially project our data vectors into? Here, the answer depends on whether the vectors of interest are like the columns of $M$ or the rows of $M$. If the vectors of interest are like the columns of $M$, then it is the first $k$ of the $\vec{u}_i$ that give us an orthonormal basis for the subspace of interest. If the vectors of interest are like the rows of $M$, then it is the first $k$ of the $\vec{w}_i^T$ that give us an orthonormal basis for the subspace (of rows) of interest. This can be seen by simply looking at the dimensionalities involved. If the matrix $M$ is rectangular, only one of these will even be the right size. The first principal component is the most important direction (the one corresponding to the biggest singular value), the second principal component is the second most direction (the one corresponding to the second biggest singular value), and so on.

The above might seem a little too slick and abstract. Why is this a reasonable approach? And how does this apply to the case of the brain-machine interface where we have 100 dimensional segments that we want to project into a 3 dimensional space? How to apply it is pretty easy to answer. Suppose we collected 1000 traces of recordings of neural spikes on the electrode we are interested in. So we have a data set that has a thousand 100-dimensional vectors in it. We can arrange these into a matrix $M$ with a thousand columns, each of which is one of these 100-dimensional vectors. Then, we can use the first three $\vec{u}_i$ from the SVD as the basis of the space that we project into, and because the $\vec{u}_i$ are orthonormal, the projection can be calculated pretty easily. Given a new spike recording $\vec{x}$ that is a 100-dimensional vector, we summarize it by computing the vector $\begin{bmatrix} \vec{u}_1^T \vec{x} \\ \vec{u}_2^T \vec{x} \\ \vec{u}_3^T \vec{x} \end{bmatrix}$. This 3-dimensional vector should contain what we believe to be the important information in $\vec{x}$.

Now, let's move on to why is any of this a reasonable approach at all. Before we do a proof, it is good to understand things in an example that is more intuitive than the practical motivation of classifying neural recordings. To explain this, we considered the problem of film ratings. (Think about Netflix. They need to predict which movie you will like before even you know whether you like that movie.) Consider a collection of

$n$ people who have rated $m$ films — giving each film a real number rating. Suppose we arrange this data into a matrix $R$ with $n$ rows corresponding to different individuals' ratings of the films and $m$ columns corresponding to different films' ratings by the people. There are a total of $nm$ numbers in this matrix and if we knew nothing about the underlying structure, there is nothing more we could say. To be able to predict anything, there has to be an underlying structure. One simple underlying structure could be that every film is associated with an intrinsic "goodness" attribute and every person's rating of a film is the product of their personal sensitivity with that film's goodness. So, there is a vector $\vec{s} = \begin{bmatrix} s[1] \\ s[2] \\ \vdots \\ s[n] \end{bmatrix}$ that contains the sensitivities of the different people. And similarly a vector $\vec{g} = \begin{bmatrix} g[1] \\ g[2] \\ \vdots \\ g[m] \end{bmatrix}$ that contains the goodnesses of the different films. If there was nothing else to the ratings, then $R = \vec{s}\vec{g}^T$. Here, we definitely see that there is a discoverable structure that can be learned from data — however from rating data alone you can't figure these underlying vectors out exactly, only up to scaling. Films might actually be a bit more complicated in their nature. Suppose that every film is accurately summarized by three intrinsic attributes: romance, comedy, and action. Similarly, every individual person has a particular personal weighted combination of romance, comedy, and action that is used to determine their rating. Then, we can imagine that $R = \vec{s}_r\vec{r}^T + \vec{s}_c\vec{c}^T + \vec{s}_a\vec{a}^T$ in outer product form. If this model was perfect, then the $R$ matrix would actually be rank 3 — that is it would have a column span and a row span both of dimension 3. However, reality is never perfect and there is going to be some "disturbance" that corresponds to unmodeled terms or random fluctuations in the ratings. That means that the matrix $R$ won't actually be rank 3, it will just be "almost" rank 3. This softer sense of rank is what we need to capture, and that is what the SVD is intuitively helping us to do.

Anyway, for the movie example, it is useful to think about how we would actually use our discovered $\vec{s}_r, \vec{s}_c, \vec{s}_a$ and $\vec{r}, \vec{c}, \vec{a}$. After all, these seem to be about the people and the films for which you already have ratings. From an engineering perspective, what we care about is understanding new people or new films. First, let's think about a new film. What would we want to know about it? We have decided that what is important to summarize a film is its romance factor, comedy factor, and action factor. These three numbers are what we would want. So, how would we get these? In the context of this story, we would extract them from the film's rating vector $\vec{f} = \begin{bmatrix} f[1] \\ f[2] \\ \vdots \\ f[n] \end{bmatrix}$ that corresponds to how our already known people rated that new film. How would we get these factors? We would do a least-squares projection of $\vec{f}$ onto the subspace spanned by $\vec{s}_r, \vec{s}_c, \vec{s}_a$ and read off the solution to $[\vec{s}_r, \vec{s}_c, \vec{s}_a]\vec{w} \approx \vec{f}$. The first component of $\vec{w}$ would tell us this film's estimated romance factor, the second would tell us this film's estimated comedy factor, and the third would tell us this film's estimated action factor.

We could also do the same thing for a new person. What do we want to know about them? We decided that what was important was their sensitivities to romance, comedy, and action. How do we get these? We look at

that person's ratings of all the known films $\vec{p} = \begin{bmatrix} p[1] \\ p[2] \\ \vdots \\ p[m] \end{bmatrix}$ and do a least squares projection of $\vec{p}$ onto the subspace spanned by $\vec{r}, \vec{c}, \vec{a}$. In other words, solve $[\vec{r}, \vec{c}, \vec{a}]\vec{w} \approx \vec{p}$. The first component of $\vec{w}$ would tell us this person's estimated romance sensitivity, the second would tell us this person's estimated comedy sensitivity, and the third would tell us this person's estimated action sensitivity.

When PCA helps us discover a subspace of interest (namely, the subspace that corresponds to the largest singular values), it actually hands us an orthonormal basis for the subspace. The orthonormality makes projection easy, but it may not correspond to values that are as "physically" significant as say romance, comedy, and action. For example, it might be that in our library of films, there are lots of romantic comedies in which romance and comedy co-occur more often than not. Then, the vectors we get won't correspond to pure romance or pure comedy, but rather to some mixtures. We'd have a romantic comedy vector and a decidedly unromantic comedy vector instead. But for our engineering purposes at the moment, this is not that important. It is more important to be able to summarize films using three numbers, and PCA will give us a way to do that. Recall that we know that when we do these kind of "rank-1" decompositions into a sum of outer products, we are free to scale the two vectors as we want as long as the product works out correctly. The SVD perspective just keeps the scaling factor out separately so we can see it. If you want, you can consider it a kind of "unit" that we need to keep track of. In the film example, we might want to fold it into the people's sensitivities for example. Thinking about this is important if we ever want to extract both kinds of summary features and then use them — for example, we estimate the sensitivities of new people on the basis of films that they've rated and estimate the content factors of new films on the basis of how people rate them. Then, if we want use both of these estimates to predict a new rating, we have to be careful not to double count. For example, consider the following trivial rating matrix:

$R = \begin{bmatrix} 4, 4, 4, 4 \end{bmatrix} = [1][8][\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$. This is a universe with one person and four films. Now, suppose that we see a new person, and this new person rates these same films as $[2, 2, 2, 2]$. Then, if we project this person's ratings into the SVD given coordinate system, we would say that this new person has a sensitivity of $4$. (As compared to the original person's $8$ — which makes sense in a relative sense, this person is clearly half the sensitivity of the original person.) If a new film is rated by the first person as a $3$, then projecting this film's rating using the SVD-given coordinate system would give us a goodness factor of $3$. (As compared to the original films all having a goodness factor of $4$ — which makes sense in a relative sense, this film is clearly $\frac{3}{4}$ as good as the original films.) But how do we predict how the new person would rate the new fim? We know what the answer should be: the new film should be rated $\frac{3}{2}$. But what happens when we just naively multiply together $3$ (the new film's goodness factor) times $4$ (the new person's sensitivity)? We get $12$ which is obviously wrong. We are off by a factor of $8$. This happened because we were essentially double counting. The $8$ from the SVD implicitly got folded into both estimates. We need to avoid this if we're going to use double-sided dimensionality reduction to make predictions.

In discussion, we have focused on making sure that you understand what is going on with the SVD and provided a bit more of the geometry of what it is doing.

Next, we set out to prove that the approach to PCA that we are taking indeed picks out the best subspace. Here, we had to first confront the issue of how do we express "best" mathematically? For a matrix $M$, we wanted to find $k$ orthonormal columns (arranged into a tall matrix $B$) so that the projection of $M$ onto the

subspace spanned by $B$ is as good as possible. This projection is $BB^TM$ and so the residual matrix is $M - BB^TM$. We would like this residual to be small, but what does small mean for matrices? We need a natural norm for matrices that treats them like a big array of numbers. It turns out that the Frobenius norm does this: $\|M\|_F = \sum_i \sum_j |m[i][j]|^2$. We showed that $\|M\|_F = \|M^T\|_F$ and furthermore, if $U$ had orthonormal columns, that $\|UM\|_F = \|M\|_F$, paralleling the story for vectors. Similarly, if $V$ is square and has orthonormal columns, then $\|MV^T\|_F = \|M\|_F$ as well.

Once we had linearization of nonlinear functions in our toolkit, we became empowered to revisit the question of classification — taking data and returning a discrete label for it. For example, taking audio samples and returning which word was being said (as you will do in lab) or in the spike-sorting example above, Here, we used simple examples to illustrate how we can conceptually think about classification as being done at different levels of complexity:

Classification by means — this is what you do in lab. You just choose the average of all the examples of a particular category as the paradigmatic example of that category, and then classify any new point based on which of these centers it is closest to. In the binary case (where there are two categories), this turns into finding the perpendicular bisector of the two means and then seeing which side of that hyperplane we are on. Classification by least squares: Here, we never compute the means of each category. Instead, we just directly look for the appropriate "perpendicular bisector" of the data points themselves. This can be done by giving each of one category the label $y_i = +1$ and each of the other category the label $y_i = -1$. At that point, we can ask least squares to fit these labels using the measurements $(1, \vec{x}_i)$. Here, the 1 is added to allow the perpendicular bisector to go through something other than the origin. Classification by least-???: The least-squares approach is nice to implement, but it has a bizarre behavior from an engineering point of view. If there are points that are very far from the (correct) boundary but are well within their home territory, least-squares will try to move the boundary to make these points closer to it. This is because least squares doesn't like deviations from +1 or -1 in either direction, whereas we as engineers would be fine if some points that were nominally marked as $y_i = +1$ actually were getting $\vec{w}^T\vec{x}_i$ that were much bigger than 1. Positive is positive — we're not making any mistakes because of this. This requires using a penalty function that is not squared-loss, and it turns out that we can do this and still keep essentially the same computational simplicity as least-squares because we can use linearization ideas repeatedly.

The key tool that we developed for dealing with (3) above was the concept of the second derivative of a scalar-valued function with respect to a vector-valued argument. This allowed us to approximate a function locally as a quadratic: $f(\vec{w}_0 + \vec{\delta w}) \approx f(\vec{w}_0) + \underbrace{\frac{\partial f}{\partial \vec{w}}|_{\vec{w}_0}}_{\text{row vector}} \vec{\delta w} + \frac{1}{2}\vec{\delta w}^{\top} \underbrace{\frac{\partial^2 f}{\partial \vec{w}\partial \vec{w}}|_{\vec{w}_0}}_{\text{square matrix}} \vec{\delta w}$. Here the matrix $\frac{\partial^2 f}{\partial \vec{w}\partial \vec{w}}|_{\vec{w}_0}$ is filled with the second partial derivatives $\frac{\partial^2 f}{\partial w[h]\partial w[g]}|_{\vec{w}_0}$ with respect to the components of $\vec{w}$. By doing this local expansion in the context of function $f(\vec{w}) = \sum_{i=1}^{m} c^{\ell_i}(\vec{x}_i^{\top}\vec{w})$, we massaged the problem into a least-squares type problem that we know how to solve as long as the second derivatives of the $c^{\ell}$ functions were positive. This allowed us to iteratively solve.

The natural non-quadratic loss functions that we considered were exponential loss where $c^+(p) = e^{-p}$ for points that had a positive label and $c^-(p) = e^p$ for points that had a negative label. This loss successfully ignored correctly classified points that were well within their designated region, but got overly obsessed over points that were incorrectly classified because their losses were huge. To mellow out the loss, we were inspired by the behavior of Bode plots on a log-log scale to consider $c^+(p) = \ln(1 + e^{-p})$ and $c^-(p) = \ln(1 + e^p)$. The logarithm

doesn't interfere with the cost function's ignoring of well classified points while it does mellow out the obsession over incorrectly classified points because $\ln(1 + e^p) \approx p$ when $p$ is large. This essentially linear-type behavior makes the second-derivative small instead of being explodingly huge.

These iterative least-squares algorithms can be reconceptualized as having an implicit approximation for the entire function $f(\vec{w})$ as a patchwork of local quadratic approximations. However, this global approximation for $f$ is never fully computed, and instead, the algorithms proceed in the style that you saw in 61A when you learned about iterators and streams — the approximation was evaluated as needed where it was needed. It turns out that there is also value in having explicit global approximations for functions of interest.

We were motivated by the desire to reduce the rate at which we take samples in our brain-machine interface. A similar issue arises in the GPS application that you saw in 16A — we want to compute our position to within a few meters, but the speed of light is so fast that we don't want to have to take samples every ten nanoseconds so that we can maximize the correlation at that resolution. We would prefer to take samples more infrequently and be able to figure out what was happening in between the samples we took, even though we didn't take those samples. In the case of the brain-machine interface, we know that we are going to be projecting down to a lower-dimensional subspace anyway. So why not take fewer samples? But how? What can we count on? Our wish is to globally approximate the time-waveform that we are sampling under the assumption that it is nicely behaved. What do we mean by nicely behaved? Clearly, we don't want it to have lots of interesting things happening in-between the samples that we do take — if that were happening, we would definitely be missing something. We want it to be predictable, and one way of understanding that at an intuitive level is that we hope that the waveform isn't too "wiggly" — if there were lots of surprising twists and movements that happened at a very fine time-scale, it would probably be hopeless. At the same time, we want to have an approximation that we know how to do — in terms of familiar things.

Fortunately, polynomials are something that we are quite familiar with from our understanding of local approximation by Taylor series, and just generally from years of exposure in high school, etc. Polynomials also have a connection to "wiggliness" in our intuition — a quadratic can have one turn, a third-degree polynomial can have two turns, a quartic can have three turns, a quintic can have four turns, and so on. So we decided to start by considering polynomials.

We can immediately set up the problem of global approximation by a degree-at-most $k$ univariate polynomial as a least squares problem that we allows us to learn from $m > k$ data points $(x_i, y_i)$. We have seen this before in 16A when it was demonstrated to us how the orbit of Ceres was reconstructed from measurements. However, there is a technical condition — how do we know that the resulting feature matrix $D = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_2^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^k \end{bmatrix}$ actually has full column rank — i.e. linearly-independent columns? We need that to compute $(D^\top D)^{-1}$ along the way to getting $\widehat{a} = (D^\top D)^{-1} D^\top \vec{y}$. Here, we used the easy way to show this — we followed the spirit of changing coordinates. The easiest matrix to verify linear-independence of columns is the identity matrix. So, we want to rethink our representation of a generic degree-at-most-$k$ polynomial instead of being $\sum_j a[j]x^j$ as instead being $\sum \widetilde{a}[j]\ell_j(x)$ where the $\ell_j(x)$ are some different set of degree-$k$ polynomials. For computing $\widetilde{a}$ we

would instead have to compute $\widetilde{D} = \begin{bmatrix} \ell_1(x_1) & \ell_2(x_1) & \ell_3(x_1) & \cdots & \ell_{k+1}(x_1) \\ \ell_1(x_2) & \ell_2(x_2) & \ell_3(x_2) & \cdots & \ell_{k+1}(x_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \ell_1(x_m) & \ell_2(x_m) & \ell_3(x_m) & \cdots & \ell_{k+1}(x_m) \end{bmatrix}$. Let's truncate this to the first $k+1$ rows and just consider $m = k+1$. In that case, the matrix $\widetilde{D}$ is square and we want it to be the identity. This means that we want $\ell_i(x_j) = 0$ if $i \neq j$ and $\ell_i(x_1) = 1$. Setting a polynomial to have zeros where we want is easy, we just consider $\prod_{j \neq i}(x - x_j)$. Getting it to be 1 where we want is also easy since that is just normalization, giving us the Lagrange polynomials $\ell_i(x) = \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$. These are clearly degree $k$ since $x^k$ is the leading term when one multiplies together $k$ terms of the form $(x - x_j)$. Writing out the Lagrange polynomials in terms of the original monomials $\ell_i(x) = \sum_{j=0}^{k} \beta_{i,j} x^j$ tells us that the columns of $\widetilde{D}$ are a linear combination of the columns of $D$. In particular $DB = \widetilde{D} = I$ where $B = \begin{bmatrix} \beta_{1,0} & \beta_{2,0} & \cdots & \beta_{k+1,0} \\ \beta_{1,1} & \beta_{2,1} & \cdots & \beta_{k+1,1} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{1,k+1} & \beta_{2,k+1} & \cdots & \beta_{k+1,k+1} \end{bmatrix}$. Thus $B = D^{-1}$ and so $D$ is invertible and hence has linearly independent columns.

This is great. It not only tells us that we can run least-squares in principle, it also gives an explicit way to interpolate a degree-at-most-$k$ polynomial given $k+1$ data points. We just use $\widehat{f}(x) = \sum_{i=1}^{k+1} y_i \ell_i(x)$ as our approximation. So what more needs to be said? In the context of recording samples from a continuous-time waveform, there is one issue. We need to pick the $x_i$ that we use for polynomial interpolation. If we sample regularly every $\Delta$ seconds, one choice for $x_i$ could be the actual time $i\Delta$ that the sample was taken. Alternatively, we could also normalize the entire block of $m$ samples to correspond to $x_i$ between 0 and 1 by using $x_i = \frac{i}{m}$ and use zero-indexing for $i$. However, it is here that we encounter a devastating practical roadblock — this simply doesn't work in practice when $k$ gets even moderately large (like 30-ish). This is because monomials like $x^20$ are very bad behaved on the real axis. Adopting the metaphor from the story of Goldilocks and the Three Bears: the region between $(-1, +1)$ is "too cold" — raising a number in there to a high power makes the result very close to zero. The region $|x| > 1$ is "too hot" — raising a number there to a high power makes the result enormous. This just doesn't work out well with the finite precision arithmetic that our computers do. The problem is that the "Goldilocks Zone" of "just right" is too small on the real axis — only the two numbers +1 and -1 have $|x| = 1$.

To resolve this issue, we must free our mind and abandon the desert of the reals. Polynomials are perfectly well defined and well behaved over the complex plane, so why not take advantage of the plentiful room on the unit circle! If we restrict attention to $x = e^{j\theta}$, the monomials are all very well behaved. $x^k = (e^{j\theta})^k = e^{jk\theta} = \cos(k\theta) + j\sin(k\theta)$. Over the natural domain $[0, 2\pi]$, these functions still have our desired control of "wiggliness" — the higher $k$ gets, the more wiggles can happen. But nothing blows up to infinity or decays to close to zero. This suggests that we should instead pick $x_i = e^{j\frac{2\pi i}{m}}$ — $m$ evenly spaced points on the unit circle. Polynomial interpolation will work here and we will get a global approximation of the function that we want. Least-squares is what we want to work, but here, we need to understand how to do projections of complex vectors, and this requires figuring out how to define the complex inner product.

It turns out that it is natural to define $< \vec{a}, \vec{b} >= \sum_i a[i]\overline{b[i]} = \vec{b}^* \vec{a}$ where $\vec{b}^*$ is the conjugate transpose — take the transpose, and then complex conjugate everything. This way, the projection of $\vec{a}$ onto the vector $\vec{b}$ remains $\frac{<\vec{a},\vec{b}>}{<\vec{b},\vec{b}>}\vec{b}$ as it was in 16A. Discussion section walked through this in more detail.

With the complex inner-product in hand, we could handle projections onto a standard monomial basis of

degree $k$ polynomials represented by their evaluations on the $m$-th roots of unity — corresponding to our $m$ data samples. If we look at $B = [\vec{b}_0, \vec{b}_1, \cdots, \vec{b}_{m-1}]$ where $\vec{b}_k[i] = (x_i)^k = e^{j\frac{2\pi ik}{m}}$, this is a basis for the entire space of $m$-dimensional complex vectors because it inherits that property from its basic polynomial nature since all the evaluation points are distinct. However, a direct calculation shows that $< \vec{b}_k, \vec{b}_\ell >= 0$ if $\ell \neq k$ and $\|\vec{b}_k\| = \sqrt{m}$. So they are all orthogonal to each other, and if we defined $U = [\vec{u}_0, \vec{u}_1, \cdots, \vec{u}_{m-1}]$ where $\vec{u}_k = \frac{1}{\sqrt{m}}\vec{b}_k$, then this would be an orthonormal basis. The $U$ basis is called the orthonormal Discrete Fourier Transform (DFT) basis and the $B$ basis is called the polynomial-style DFT basis. Another normalization $\frac{1}{m}B$ is called the classic/traditional DFT basis, but we aren't going to cover that one. The important thing is that the orthonormality of $U$ also lets us easily compute the inverse of $B$ as $\frac{1}{m}B^*$. At this point, it is easy to take a sequence of samples $\vec{f} = \begin{bmatrix} f[0] \\ f[1] \\ \vdots \\ f[m-1] \end{bmatrix}$ and find a sequence of coefficients $\vec{F} == \begin{bmatrix} F[0] \\ F[1] \\ \vdots \\ F[m-1] \end{bmatrix} = \frac{1}{m}B^*\vec{f}$ for it.

The question is just, how do we use this sequence $\vec{F}$ to interpolate the samples $\vec{f}$. If we identify the $i$-th point in the vector $\vec{f}$ with the real number $\theta_i = \frac{2\pi i}{m}$, then we can think of $f[i] = \widehat{f}(e^{j\theta_i})$ where $\widehat{f}$ is the function that we have learned, parameterized by the $\vec{F}$. Then, we could query this function for any desired place in between (where we don't necessarily have an actually measured sample) by just evaluating $\widehat{f}(e^{j\theta})$.

There is a natural choice that comes from our motivation, let's use $\widehat{f}(x) = \sum_{k=0}^{m-1} F[k]x^k$ — a straight-up degree $m-1$ polynomial. This seems eminently reasonable, and since $x^k$ over the $m$-th roots of unity evaluates to $\vec{b}_k$, and we used $\vec{b}_k$ to build $B$, the polynomial will definitely interpolate all the data. There is just one practical oddity — when we use this approach for a real-valued set of samples (say we were measuring voltages in the brain), we will generally always get truly complex interpolations in between.

This behavior causes us interpretational difficulties — why is the interpolation hallucinating nontrivial imaginary components despite learning from data in which the imaginary components were always zero. These seems like a blatant violation of Occam's razor of always trying to learn the simplest possible pattern that is compatible with the data. So, we need to do something about this. If we are committed to the $\vec{F}$, our only option is to change the $\widehat{f}$ somehow. But what freedom do we have? It is here that we notice an interesting ambiguity of sorts. The vector $\vec{b}_k$ actually supports multiple back-stories. It turns out that because of the behavior of the $m$-th roots of unity — namely that raising any one of them to the power $m$ gives us 1 — that $\vec{b}_k = \vec{b}_{k+qm}$ for any integer (positive or negative) $q$. In the context of the DFT basis vectors, we can get a clean handle on this phenomenon, but it is actually universal across all machine learning settings — there are generally always infinitely many patterns that are compatible with any finite set of data. The fact that many functions can give rise to the same samples is referred to as "aliasing" — in an interested choice of terminology where we view the functions as the "names" that we invoke and the data samples that we have taken as being the object in front of it. Each object has many "aliases" and we want to summon forth the correct name for our purposes.

One interesting consequence is that $\vec{b}_{m-1} = \vec{b}_{-1}$ which means that $\vec{b}_1 = \overline{\vec{b}_{-1}}$ — they are complex conjugates of each other. So the $\vec{F}$ that we learn from real data will always have complex conjugacy relationships built in. We can exploit this to get real interpolations. How? By snapping our fingers and changing the backstory for half

of the DFT vectors. We change the latter half of them to all correspond to negative small frequencies instead of big positive ones. For odd $m$, this means that we use $\widehat{f}(x) = F[0] + (\sum_{k=1}^{\lfloor \frac{m}{2} \rfloor} F[k]x^k) + (\sum_{k=1}^{\lfloor \frac{m}{2} \rfloor} F[m-k]x^{-k}) = F[0] + \sum_{k=1}^{\lfloor \frac{m}{2} \rfloor} F[k]x^k + F[m-k]x^{-k}$. Evaluated at $e^{j\theta}$, this always gives real answers and these answers turn out to be smooth and natural.

The case of even $m$ was explored in discussion, where the single coefficient at $F[\frac{m}{2}]$ has to be interpreted somehow. This is always real for real $\vec{f}$ because $\vec{b}_{\frac{m}{2}}$ has entries that alternate between $+1, -1$, so its complex conjugate is itself. We choose to split the difference for this coefficient and summon forth the symmetric name $\frac{1}{2}(x^{\frac{m}{2}} + x^{-\frac{m}{2}})$ for this term.

In later courses, you will learn more about these things, as well as how the DFT basis can also be viewed as an eigenbasis. But this concludes the story in 16B.