slide1:
Hello, everyone.#Let’s take a few minutes before we dive into today’s MATLAB tutorial to connect what we’ve learned so far.
Over the past lectures, we’ve covered some very important foundations — linear systems, convolution, the delta function, Fourier series, Fourier transforms, and the sampling theorem. These ideas are deeply connected, and if you understand them well, everything that follows will be much easier. If you are still unsure about any of these topics, now is a great time to go back and review.

slide2:
Anyway, so this is our schedule. We are on schedule, so no problem.
Let me briefly walk you through the big picture again.
A linear system is one where we can predict the output for any input by combining the system’s response to simple building blocks. Convolution is the tool that allows us to do exactly that — it takes the input signal and the system’s impulse response, and combines them to give the output.
The Fourier series is another powerful idea. It tells us that any periodic signal can be written as a sum of sine and cosine waves. If we make the period infinitely long, this naturally becomes the Fourier transform, which works for non-periodic signals.
But computers don’t deal with continuous signals. We must first discretize them — that is, represent them as sequences of numbers. This involves sampling in time, and sometimes quantizing amplitudes. The sampling theorem assures us that, if we sample fast enough — at least twice the highest frequency in the signal — we can reconstruct the original waveform perfectly under ideal conditions.
Once we sample in time, something interesting happens in the frequency domain: the spectrum becomes periodic. This leads us to the discrete Fourier transform. And if we compute it efficiently, we use the fast Fourier transform, or FFT.
These tools are not just theory. They are powerful methods for denoising signals, extracting features, and understanding system behavior. But please remember — typing “fft” in MATLAB is easy; the real value comes from understanding what that result means and why it looks the way it does.
Today, I will guide you through some hands-on MATLAB examples so you can see how Fourier analysis works on discrete data, and how these ideas connect back to the theory we’ve been building. This is a great chance to make sure the concepts are clear in your mind.
Alright, let’s get started.

slide3:
Let me give you a quick roadmap for today’s session.
We’ll begin with discrete convolution. This is simply the convolution operation applied to signals that are already in discrete form — sequences of numbers rather than continuous waveforms. We’ll explore how to compute it, and why sometimes we add extra zeros at the ends of our signals — a technique called zero-padding. We’ll also see the idea of circular convolution, and how it relates to regular, or linear, convolution.
Next, we’ll move into spectral analysis. This is one of the most common applications of Fourier analysis. Here again, zero-padding plays an important role — but now it’s in the frequency domain. We’ll also talk about refined spectral bins, which allow us to see frequency details more clearly.
Finally, we’ll take what we’ve learned and apply it to two-dimensional filtering for images. Using the Fourier transform, we’ll perform noise removal to clean up an image, and edge enhancement to bring out important structures.
So, in short — discrete convolution, spectral analysis, and 2D image filtering. We’ll work through these step by step, connecting the math to hands-on MATLAB demonstrations.

slide4:
Now, let’s take a moment to visualize what the discrete Fourier transform — or DFT — really means.
On this slide, you see two columns. On the left, we have signals in the time domain. On the right, we have their corresponding representations in the frequency domain. The arrows between them remind us that the Fourier transform

takes us from one domain to the other, and the inverse Fourier transform brings us back.
Each row here is a different example. In some cases, we’re multiplying two signals in the time domain. In others, we’re convolving them. And each time, there’s a matching operation in the frequency domain — because multiplication in one domain corresponds to convolution in the other, and vice versa. This is one of the most important dual relationships in Fourier theory.
You can also see how different shapes in the time domain create specific patterns in the frequency domain. For example:
A narrow pulse in time spreads out in frequency.
A broad, smooth shape in time becomes more concentrated in frequency.
Sampling a continuous signal in time produces repeated copies — or “aliases” — in the frequency domain.
By moving row by row, you can start to see the bigger picture: every change we make to a signal in one domain has a predictable effect in the other. This is exactly why Fourier analysis is so powerful — it gives us two different but connected ways of looking at the same signal.
As we go through the rest of today’s examples, keep these relationships in mind. They’ll help you understand not just the MATLAB results, but also the deeper reason why those results look the way they do.

slide5:
Here’s a simple, visual way to think about convolution.#Imagine you have two hand shapes. One is fixed, and the other is flipped and shifted.
In convolution, we take one function — or one signal — and flip it around, just like turning a right hand into a left hand. Then, we slide it along the other signal. At each position, we calculate how much the two shapes overlap. The amount of overlap becomes the value of the convolution at that point.
The picture here shows this idea using two hands. The vertical axis represents flipping, and the horizontal axis represents shifting. You can imagine that at certain positions, the hands align perfectly, giving a large overlap, while at other positions they hardly touch at all, giving a small or zero overlap.
This is exactly what happens in math: convolution measures the similarity between a signal and a shifted, flipped version of another signal. This idea works whether we are dealing with shapes like these hands, continuous waveforms, or sequences of numbers.
By keeping this visual in mind, convolution will feel much less abstract. It’s just sliding one pattern over another and measuring how well they match at each shift.

slide6:
Here’s our first step in the convolution process — calculating the value at n equals zero.
On the top left, the red arrows show f of k, our first discrete signal. On the bottom left, the green arrows show g of negative k, which means we have flipped the second signal in time.
The formula in the middle tells us what convolution is in mathematical terms: we take the sum over all k of x of k multiplied by h of n minus k.
Here, x is our input, and h is our system’s impulse response.
At n equals zero, we line up the flipped signal so that its zero index matches the zero index of the original signal. You can think of this like the hand illustration on the right: one hand flipped, the other fixed, and their starting points aligned at zero.
Now, at each matching position of the red and green arrows, we multiply the values together and then sum all those products. That sum is the value of the convolution at n equals zero.
This is the key step — convolution is nothing more than multiplying corresponding values from two aligned sequences, and then summing the results. We repeat this process for each shift of n to get the full convolution.

slide7:
Now let’s see what happens when we compute the convolution at n equals eight.
Just like before, the red arrows on the top represent f of k, and the green arrows on the bottom show g of eight minus k — meaning the second signal has

been flipped and shifted so that its index aligns with n equals eight.
In our hand analogy, you can imagine sliding the flipped hand far enough to the right so that only the outer fingers — the “end fingers” — are touching. This is the opposite alignment from what we saw at n equals zero.
At this position, only a few of the red and green arrows overlap. We multiply those overlapping values and then sum them up. The result gives us the convolution value at n equals eight.
By repeating this shifting process for every value of n, we build up the entire convolution sequence, one position at a time.

slide8:
Let’s walk through this functional example, which will connect directly to what we’ll do in MATLAB later.
We start with h of n, shown in the first plot — a simple rectangular pulse. Then we have x of n in the second plot — think of it as three rectangular pulses in a row, one shown in red and the other two in blue.
If we take these two signals and perform a linear convolution — shown in the third plot — we get a series of triangular shapes. This makes sense: convolving two rectangular pulses produces a triangle. Here we have three such triangles because x of n contains three rectangles.
Now, in the fourth plot, something changes. We create x N of n — this means x of n is now periodically extended with N samples per period. In other words, we’ve made it repeat over and over, which is why we call it circularly extended.
In the fifth plot, we see the linear convolution of this periodic x N of n with h of n. Notice what happens: instead of three clean, separate triangles, the edges wrap around, and the red and blue parts start overlapping. These are edge effects caused by the circular extension.
Finally, the last plot shows the composite output. The green section in the middle is the portion unaffected by edge effects. This part matches the original linear convolution result and is what we actually want.
The lesson here is important: to avoid these unwanted overlaps in circular convolution, we often use zero-padding — adding extra zeros to the signals before performing the convolution. This ensures that the result matches the true linear convolution without distortion at the edges.
We’ll explore this zero-padding step in detail in a few slides, and you’ll see how MATLAB handles it.

slide9:
Here’s a simple MATLAB example you can try for yourself.
We start by defining two discrete signals — or vectors in MATLAB — x and h. In this case:#x equals 5,4,3,2,1#h equals 1,2,3,4,5
Next, we perform a convolution using MATLAB’s built-in conv function:#y equals conv x, h;
Finally, we plot the result:#plot y; ylim 0 to 100;
What do we see? A clean triangular shape. This is exactly what we expect when we linearly convolve two finite sequences that look like ramps in opposite directions. The convolution produces a sequence that starts small, rises to a peak — here at the center — and then falls symmetrically.
This simple example is a good reminder that convolution is not mysterious. It’s just a systematic way of multiplying and summing overlapping parts of two signals. And MATLAB makes it easy to visualize.

slide10:
Now let’s think about circular convolution.
Earlier, with linear convolution, we had just one pair of signals sliding past each other. In this hand analogy, that meant one top hand and one bottom hand.
Here, with circular convolution, the situation is different. The signal is treated as if it repeats periodically — just like in our earlier example from Wikipedia. So instead of one top hand, we now have several copies in a row, as shown here with three hands.
When we slide the lower signal, we can’t “run out” of data on either end, because the pattern simply wraps around. That means parts of the signal from the end overlap with parts from the beginning.
In the picture, you can see that the fingers from the lower hands are

overlapping with the fingers from the upper hands — not just in the middle, but also across the boundaries. This wrapping-around effect is exactly what creates the edge overlaps we saw earlier.
If we don’t want those overlaps to distort our result, we need to use zero-padding before performing the convolution. That’s the key difference between the clean linear convolution result and the wrapped-around circular convolution result.

slide11:
Now, let’s see how zero-padding helps us fix the edge effect problem we saw with circular convolution.
In the earlier example, the red and green lines — representing the start and end of the main region we care about — were very close together. This caused the signal to wrap around and overlap in places we didn’t want.
Zero-padding means we insert extra zeros at the ends of our signals before performing the convolution. In this hand analogy, adding zero-padding is like putting some extra space between the hands. The top and bottom hands are now farther apart before they start overlapping.
As a result, the main section we care about — here marked in red — remains untouched by unwanted overlaps. The green section, which previously had interference from wrapping, now stays clean.
This is exactly how we prevent the distortion seen in the Wikipedia example: by padding with enough zeros, the circular convolution result becomes identical to the true linear convolution result.

slide12:
Here’s a straightforward MATLAB example to show linear convolution in action.
We start with two sequences:#x equals 5,4,3,2,1,0#h equals 1,2,3,4,5,6
The first figure shows both sequences plotted together. The blue line is x — starting high and decreasing. The red line is h — starting low and increasing.
Now we use MATLAB’s built-in conv function to compute their linear convolution:#lconv equals conv x, h;
When we plot the result, we see the familiar triangular shape. It rises as the two signals increasingly overlap, reaches a peak when their centers align, and then falls as the overlap decreases.
If this feels familiar, it’s because we used the same function — conv — earlier when convolving the RC circuit’s impulse response with the unit step function in your homework. The process is exactly the same here; only the input sequences are different.
This example is a nice reminder that MATLAB makes it very easy to visualize convolution, and seeing the result in a plot helps reinforce what’s happening mathematically.

slide13:
Here’s where we can clearly see the difference between linear convolution and circular convolution without zero padding.
In MATLAB, we can perform circular convolution using the Fourier transform approach. First, we take the FFT of our two sequences, x and h. Then, we multiply their spectra point by point. Finally, we take the inverse FFT to bring the result back into the time domain.
Mathematically, multiplication in the frequency domain is equivalent to convolution in the time domain — but here’s the key point: if we don’t use zero-padding, the convolution we get is circular by nature.
In the plot below, the blue curve is the circular convolution result without zero-padding. The red dashed curve is the linear convolution result from our earlier example.
Notice how the blue curve is distorted — it doesn’t match the clean triangular shape of the red curve. That distortion comes from the wrapping-around effect we talked about earlier: parts of the signal from the end are interfering with parts from the beginning.
This is why zero-padding is essential when we want the frequency-domain method to give the same result as linear convolution. Without it, circular convolution will give you something different.

slide14:
Here we see why zero-padding is the key to making circular convolution match linear convolution.
We begin by setting N to be the length of x plus the length of h, minus 1. This value of N is exactly the length of the linear convolution result — something you’ve already seen in your RC circuit homework.
Next, we create padded versions of our two signals:
x pad starts with the original values of x, followed by enough zeros so that its length is N.
h pad is created in exactly the same way, starting with h and then adding zeros to reach length N.
Now, we take the FFT of these zero-padded signals, multiply them point by point in the frequency domain, and take the inverse FFT to return to the time domain.
When we plot the result, we see that the zero-padded circular convolution — shown in blue — lies exactly on top of the linear convolution — shown in red. This perfect match confirms that zero-padding has removed the unwanted wrap-around effects and given us the correct, undistorted result.
So the takeaway is clear: if you’re using the FFT method to compute convolution, always add enough zero-padding to get the true linear convolution.

slide15:
Now that we’ve explored convolution in detail — both in the time domain and through the FFT method — let’s move into one of the most common and powerful applications of the Fourier transform: spectral analysis.
Spectral analysis is about looking at a signal in the frequency domain to see what frequencies are present and how strong they are. This is incredibly useful in engineering, science, and even everyday technology — from analyzing audio signals, to detecting features in images, to identifying patterns in biomedical data.
In the upcoming slides, we’ll take an example signal, examine its spectrum, and then see how techniques like zero-padding and refining spectral bins help us get a clearer, more detailed view of its frequency content.
Let’s start by looking at our example signal and breaking it down step by step.

slide16:
Let’s start our spectral analysis with a simple but quite subtle example.
We’ll create a signal that contains two different pure tones — one at one hundred hertz and the other at two hundred two point five hertz.
In MATLAB, we first set the sampling frequency, F s, to one thousand hertz. That means we take one thousand samples every second.
Next, we create the time vector, t. It starts at zero seconds and increases in steps of zero, zero point zero zero one seconds — that’s one millisecond — until just before one second.
Now, our signal x is built by adding two parts:
First, a cosine wave with frequency one hundred hertz. In MATLAB, that’s written as: cosine of open parenthesis two times pi times one hundred times t close parenthesis.
Second, a sine wave with frequency two hundred two point five hertz. In MATLAB, that’s: sine of open parenthesis two times pi times two hundred two point five times t close parenthesis.
When we add these two waveforms together, we get x, which contains both frequencies.
If we plot only the first one hundred samples, we can see the combined oscillations. Sometimes the waves reinforce each other, producing larger peaks; other times they partially cancel out, making smaller peaks.
This is the time-domain view. Next, we’ll use the Fourier transform to find these frequencies in the frequency domain — and then we’ll explore how zero-padding can help us make those frequencies stand out more clearly. It takes some effort to appreciate the benefit of zero-padding!

slide17:
Now, let’s look at the spectrum of our synthetic signal without applying any zero-padding.
We perform the Fourier transform of our signal using the FFT function. This

gives us the discrete frequency components of the signal. Then, we plot the magnitude of the FFT against frequency, with frequency in hertz along the horizontal axis.
From the plot, we can clearly see a sharp spike at one hundred hertz. That spike corresponds exactly to the first or cosine component we built into the signal.
But notice something important — the second or sine component, at two hundred two point five hertz, does not appear as a clear spike. Instead, it’s blurred and not well-resolved. This happens because two hundred two point five hertz is not exactly aligned with one of the FFT’s default spectral bins, which are spaced one hertz apart.
So, without zero-padding, the FFT resolution is limited by the number of samples we have. The first frequency happens to align perfectly with a bin, so it looks sharp. The second one falls between bins, so its energy is spread out over multiple points in the plot, making it appear a bit blurry and less distinct.
We’ll see in the next slide how zero-padding helps us refine the spectral bins and make that second frequency stand out much more clearly.

slide18:
Now let’s see what happens when we add zero-padding before taking the Fourier transform.
This time, when we call the FFT function, we specify a length of two thousand. That means we take our original signal and append enough zeros to make it two thousand points long before computing the FFT.
By doing this, we change the spacing between frequency bins. Previously, without padding, the bins were one hertz apart. Now, the bin spacing is F s divided by two thousand, which equals zero point five hertz. This finer spacing allows us to more accurately pinpoint frequencies that fall between the old one-hertz bins.
When we plot the result, we still see the sharp spike at one hundred hertz.
Also, we now see a very clear spike at two hundred two point five hertz — the second frequency in our original signal.
The reason is simple: with zero-padding, we’ve increased the frequency resolution of the spectrum, making it possible to distinguish both components of our signal cleanly.
So, zero-padding doesn’t create new information — it simply lets us view the spectrum at a finer scale, revealing details that would otherwise be hidden between coarse frequency bins.

slide19:
Let’s look at a more complex example — a continuous signal that contains several frequency components.
We start with the same two frequencies from our earlier example: a cosine wave at one hundred hertz and a sine wave at two hundred two point five hertz.
Then, we add three more components:
A cosine wave at forty-five hertz
A cosine wave at four hundred seven hertz
And a sine wave at four hundred forty-five point eight hertz
When we add all these together, we get a signal that looks quite messy in the time domain — as you can see in the plot. The rapid oscillations and varying amplitudes come from the interaction of all these frequencies.
Looking at this time-domain plot alone, it’s not easy to tell exactly what frequencies are present or how strong they are. That’s where the Fourier transform becomes incredibly valuable — it lets us switch to the frequency domain, where each frequency component appears as a distinct peak, making the signal’s composition much clearer.
In the next step, we’ll apply the Fourier transform to this signal and see how it reveals the underlying structure that’s hidden in the time-domain view.

slide20:
Here’s the spectrum of our multi-frequency signal without zero-padding and without refining the frequency bins.
When we take the FFT and plot the magnitude against frequency in hertz, some frequencies stand out clearly, while others do not.
Look at forty-five hertz — this is a clean, discrete value, and it aligns

perfectly with an FFT bin. As a result, its spike reaches the maximum amplitude of one, just as we would expect.
But for the frequencies at two hundred two point five hertz and four hundred forty-five point eight hertz, things are different. Because these frequencies do not align exactly with the FFT's default bin spacing, their energy is spread across multiple bins. This makes their peaks appear lower than one and less sharply defined.
So, without zero-padding — and therefore without finer bin spacing — our frequency resolution is limited. Frequencies that happen to fall exactly on a bin are well-resolved, but others that fall in between bins are smeared out and not as clear.
Next, we'll see how refining the bins through zero-padding improves our ability to clearly detect those non-integer frequencies.

slide21:
Now, let's refine our frequency bins by a factor of two — the same idea we used earlier when going from one-hertz bins to zero-point-five-hertz bins.
We do this by setting a bin-refine factor equal to two, and then increasing the FFT length to be two times the number of samples in our original time vector. In practice, this means appending enough zeros to double the length of the signal before computing the FFT.
When we plot the result, the improvement is clear: the frequency at two hundred two point five hertz is now well resolved, showing a sharp and distinct peak.
However, notice that the frequency at four hundred forty-five point eight hertz is still not well resolved. Even with a bin spacing of zero-point-five hertz, this frequency does not fall exactly on one of the discrete FFT bins, so its energy is still spread across multiple bins. The peak is visible but not as sharp or as tall as the perfectly aligned frequencies.
The takeaway is that zero-padding and bin refinement improve resolution, but they cannot create a perfectly sharp spike unless the frequency exactly matches a discrete bin location.

slide22:
Now, let's refine our frequency bins even further — this time by a factor of four.
In MATLAB, that means setting the bin-refine factor to four and increasing the FFT length to four times the length of our original time vector. In other words, we append enough zeros so that the signal length is quadrupled before computing the FFT.
By doing this, our frequency bin spacing changes from one hertz to zero-point-two hertz. This very fine resolution means that even frequencies like four hundred forty-five point eight hertz — which were previously hard to resolve — now align closely with a bin and show a sharp, well-defined spike.
When we plot the result, every frequency in our multi-component signal — forty-five hertz, one hundred hertz, two hundred two point five hertz, four hundred seven hertz, and four hundred forty-five point eight hertz — is clearly visible, each with a maximum amplitude of one.
This example confirms that refining the bin spacing through zero-padding greatly improves our ability to detect non-integer frequencies. With enough refinement, even challenging frequencies can be resolved as sharply as those that naturally align with the original bin spacing.

slide23:
So far, we've looked at the Fourier transform in one dimension — signals that vary with time. But Fourier analysis is just as powerful in two dimensions, and that's exactly what we need for image processing.
In the context of images, the two dimensions are the horizontal and vertical spatial coordinates, rather than time. Just as a one-dimensional Fourier transform tells us what frequencies are present in a signal, a two-dimensional Fourier transform tells us what spatial frequencies are present in an image.
This is a critical tool in image analysis, because many image processing tasks — such as noise removal or edge enhancement — are easier to understand and perform in the frequency domain.
In the next slides, we'll see how 2D Fourier transforms can be applied to

filtering images, starting with noise removal and then moving on to edge enhancement.

slide24:
Here we see the concept of the two-dimensional Fourier transform.
The equations at the top describe two processes:
The forward 2D Fourier transform, which takes a spatial-domain function — something that varies in the x and y directions — and expresses it in terms of its spatial frequency components.
The inverse 2D Fourier transform takes that frequency-domain representation and reconstructs the original spatial-domain function.
In our example, the starting function is shown as a sideways rectangle in the spatial domain. When we perform the 2D Fourier transform, we obtain a frequency-domain representation — shown here in the rainbow-colored plot. The colors represent the amplitude of different spatial frequency components, and the values can be complex, with both real and imaginary parts.
Just as in the one-dimensional case, the Fourier transform lets us switch back and forth between two perspectives:
In the spatial domain, we see shapes and patterns directly.
In the frequency domain, we see how much of each spatial frequency is present in the image.
Understanding this relationship is essential for image processing, because many operations — such as filtering, noise removal, and edge detection — can be performed more effectively in the frequency domain.

slide25:
In image processing, noise can often hide important details. This is a common problem in real-world applications, especially when images are captured in low light, transmitted over noisy channels, or acquired using sensors in a challenging environment.
Here, we start with an image — the well-known “Lena” test image — that has been corrupted by noise. In the top left, you can see the noisy image in the spatial domain.
When we take the two-dimensional Fourier transform of this image, shown in the bottom left, we see its frequency-domain representation. The bright spot in the center corresponds to low-frequency components, which carry the main shapes and structures of the image. The scattered speckles throughout the spectrum represent higher-frequency noise.
If we want to reduce noise, we can selectively remove certain frequency components. One simple way is to set the outer regions of the spectrum to zero, while keeping only the frequencies near the center — this is essentially a low-pass filter in the frequency domain. In the bottom right, you can see this filtered spectrum.
Finally, applying the inverse Fourier transform gives us the restored image in the top right. The noise is greatly reduced, and the underlying features — such as the woman’s face and hat — become much clearer.
This kind of frequency-domain noise suppression is a powerful technique in imaging, allowing us to recover important details that would otherwise be lost.

slide26:
We’ve just seen how Fourier transforms can be used to remove noise. Now, let’s look at two related techniques — low-pass filtering and high-pass filtering — and how they affect an image.
Starting with the original image of the building, shown in the top left, we first take its two-dimensional Fourier transform. This gives us the frequency-domain representation, shown in the bottom left.
If we want to blur the image or remove fine details, we apply a low-pass filter. In the frequency domain, this means keeping only the low-frequency components near the center of the spectrum — as shown in the middle bottom image — and setting the high frequencies to zero. After applying the inverse Fourier transform, we get the middle top image: the overall structure is preserved, but the sharp edges are softened.
On the other hand, if we want to highlight edges and fine details, we use a high-pass filter. This is done by removing the low-frequency center and keeping

only the higher frequencies in the spectrum, as shown in the bottom right. The resulting image, in the top right, emphasizes edges and textures, but loses smooth gradients.
Low-pass and high-pass filters are fundamental tools in image processing — whether we want to smooth an image, detect edges, or prepare data for further analysis.

slide27:
In image processing, we often use specialized filters to enhance certain features or to suppress unwanted information. Here are three common examples, shown both in the spatial domain and in the frequency domain.
First, the Unsharp Filter. Despite the name, this filter is used to sharpen images. It works by subtracting a blurred version of the image from the original, which enhances the edges. In the spatial domain, you can see the filter kernel — dark around the edges, light in the center — designed to emphasize changes in intensity.
Next, the Gaussian Filter. This is a smoothing filter, which reduces noise and small details. In the spatial domain, it appears as a soft, bell-shaped pattern — bright in the middle, fading smoothly outward. Its frequency-domain representation shows that it preserves low frequencies while gradually suppressing high frequencies.
Finally, the Sobel Filter. This is an edge detection filter, which highlights regions of rapid intensity change. In the spatial domain, you can see its distinctive pattern — one side light, the other dark — which responds strongly to vertical or horizontal edges depending on its orientation. In the frequency domain, its pattern reflects the fact that it suppresses low frequencies and emphasizes specific directional components.
MATLAB provides built-in functions for these filters, including fspecial for 2D and fspecial3 for 3D filtering, as well as options to design your own custom kernels. By choosing the right filter and domain of application — spatial or frequency — we can control exactly what features of an image are enhanced or suppressed.

slide28:
Here we have an example showing the effects of different image filters, both in the spatial domain and in the frequency domain.
In the top left is our original image — a close-up of an eye. Next to it, in the top right, we see its two-dimensional Fourier transform, which shows the frequency content of the image.
On the second row, we have three filtered versions of the image in the spatial domain:
Unsharp Image — produced by applying an unsharp filter, which enhances edges and fine details by subtracting a blurred version of the image from the original.
Gaussian Blurred Image — generated using a Gaussian filter, which smooths the image by reducing high-frequency details, resulting in a softer appearance.
Sobel Filtered Image — created using the Sobel operator, which emphasizes edges, particularly strong intensity changes. In the eye image, you can clearly see the contours of the iris, eyelid, and surrounding features.
The third row shows the corresponding frequency-domain representations for each filter.
For the Gaussian blur, you can see that high-frequency components are greatly reduced, as indicated by the darker edges in the spectrum.
For the unsharp filter, higher frequencies are more pronounced, showing stronger edge components.
For the Sobel filter, the spectrum highlights frequencies associated with sharp transitions in the image, which is why it’s effective for edge detection.
By choosing different filters, we can highlight or suppress specific image features — whether we want to sharpen details, blur textures, or extract edges for further analysis.

slide29:
That brings us to the end of today’s lecture.
We’ve covered quite a lot — from discrete convolution and zero-padding, to spectral analysis, bin refinement, and practical applications in two-dimensional

image filtering. I hope you now have a clearer picture of how these concepts connect and how they are applied in real-world signal and image processing. Take some time to review the examples and try the MATLAB code on your own. Seeing the results firsthand will help solidify your understanding.

Thank you for your attention, and I look forward to seeing you in the next lecture.