slide1:
Welcome to this session, where we'll begin exploring MATLAB — one of the most widely used computational platforms in science and engineering, especially in biomedical imaging.
This will serve as your gateway to MATLAB: how it works, why we use it, and how it helps us handle real-world data. You don't need any prior experience — we'll start with the basics and build up step by step.
As we move forward, you'll use MATLAB to generate signals, reconstruct images, and analyze datasets — all key tasks in medical imaging. The goal is not just to learn syntax, but to develop intuition and confidence in working with computational tools that are directly relevant to biomedical research and clinical practice.

slide2:
We are right on schedule in our journey through this material.
If you've already previewed the reading materials related to today's lecture, excellent! That head start will help you connect concepts more easily. If not, no worries. I do encourage you to follow along closely and review key ideas afterward. This habit of reinforcing concepts as you progress will help you build a stronger, more intuitive understanding, especially as the ideas become more mathematically involved.

slide3:
Now let me address a common issue students often encounter early on — installation problems with MATLAB.
Here's a message I received from a student. He was unable to install MATLAB on his computer because of authentication issues with CAS — something that happens occasionally, especially at the beginning of the semester. The good news is that this is fixable, and you're not alone.
If you experience something similar, please reach out early. Visit the RPI Help Center, and they'll assist you with installing MATLAB. And of course, please feel free to contact me or the teaching assistant — we're here to help.

slide4:
If you're unable to install MATLAB on your own machine, don't worry — you still have options.
One solution is MATLAB Online, a browser-based version provided by MathWorks. You can access it from any modern browser without installing anything on your device. Just search "run MATLAB online," and you'll find the link right away.
Another option is Octave Online. It's a free, open-source alternative that supports many MATLAB commands. While it doesn't include all the advanced toolboxes, it's good enough for our assignments.
So remember, lack of installation shouldn't be a blocker preventing you from doing homework. There are various ways to stay engaged and complete your tasks — all you need is a browser and a willingness.

slide5:
Let's take a closer look at Octave.
Octave is a scientific programming language that mirrors MATLAB's syntax and functionality. It's free, open-source, and available for Windows, macOS, and Linux. You can run it in the browser using Octave Online.
The interface looks different, but under the hood, many of the same commands will work — especially those used for basic matrix operations, plotting, and simple scripts. This makes Octave a reliable backup if MATLAB access is delayed or limited.
While it might not support all specialized toolboxes or image processing features, it's more than enough for getting started and building foundational skills, at least for this course.

slide6:
Let me walk you through what we'll cover in this lab module.
We'll start by getting MATLAB installed on your machine. Once it's up and running, you'll go through the module MATLAB On Ramp — an interactive tutorial that introduces the core features of the platform in a hands-on way.

Next, we'll work through two examples that show MATLAB in action. First, we'll look at spectral shifts of starlight — a great illustration of how MATLAB can be used in astrophysics. Then, we'll switch gears to generate a 3D cone surface, which introduces MATLAB's powerful graphics.
And once you've the basics, we can take steps further. As good examples, we can explore several advanced MATLAB toolboxes — like image processing, instrument control, optimization, statistics and machine learning, and symbolic computation. These toolboxes allow MATLAB to go far beyond basic math and become a most powerful scientific computing platform.

slide7:
The very first step is to install and activate MATLAB.
If you're at RPI, you can download MATLAB directly from the campus software portal: dotcio.rpi.edu/services/software-labs. Make sure to follow the installation instructions that match your operating system — whether you're on Windows, macOS, or Linux.
During the installation, you'll be asked to enter a product key to activate the software. This key is provided by the institution, and it unlocks your licensed access.
If you run into any issues during the setup — like installation errors or activation trouble — don't hesitate to reach out. You can consult the help center, or message me directly. We're here to help you get started smoothly.
Having MATLAB set up and ready prepares us well before diving into hands-on examples.

slide8:
Now that you have MATLAB installed, let's talk about how to launch it.
If you're using Windows, just go to your start menu or applications folder and open MATLAB like any other program. On Linux, you'll typically launch it from the terminal by typing matlab.
Once MATLAB starts, you'll see what's called the MATLAB desktop — this is the main environment where everything happens. You'll notice a command window for typing code, a workspace that keeps track of your variables, a file browser, and an editor where you can write and save scripts.
Take a few minutes to get familiar with this layout. Learning your way around now will help you feel more confident and efficient as we go forward.

slide9:
As you begin using MATLAB, you might occasionally wonder, "What does this function do?" That's where the built-in help system comes in.
Let's say you want to understand how to create a plot. Just type help plot in the command window, and MATLAB will show you the syntax, arguments, and examples for that function.
If you don't know the name of the function you need, use the lookfor command. For example, typing lookfor matrix will return a list of functions that relate to matrices.
These built-in tools are powerful and often overlooked. Knowing how to access documentation and search for functions will make you more independent and effective as you learn.

slide10:
Once you start writing code in MATLAB, you'll be creating and manipulating variables — and the workspace is used to hold them.
The who and whos commands show you what variables are currently stored. who gives you a simple list, while whos gives more detail — like size, type, and memory usage.
You can save all your current variables to a file using the save command and later bring them back with load. If you want to start fresh, clear all wipes out everything in the workspace, clc clears the command window, and close all shuts any open figures.
These are simple but essential tools. Once you know how to manage your workspace, you'll find MATLAB easier and more enjoyable to use.

slide11:

Let’s take a moment to look at some fundamental syntax in MATLAB that you’ll see all the time.
The percent symbol — % — is used for writing comments. Anything after that symbol is ignored by MATLAB, so it’s a great way to document your code.
A semicolon — ; — tells MATLAB not to display the output of a command, which keeps your console cleaner.
If a command is getting too long, you can break it across multiple lines using three dots — .... This helps keep your code readable.
MATLAB also includes a few important constants: eps stands for machine precision, inf is infinity, and NaN — which stands for “Not a Number” — appears when a calculation isn’t mathematically defined, like dividing some number by zero.
To control how numbers appear on your screen, use formatting commands. For example, format long shows more decimal places, and format short shows fewer.
You also have access to a large library of built-in functions — like sqrt for square roots, exp for exponentials, and trigonometric functions such as sin and cos.
Basic arithmetic is intuitive — you’ll use plus, minus, times, and divide symbols just as you'd expect. And for constants, MATLAB gives you pi, and Euler’s number, which you can access with exp 1.
Together, these form the foundation of all numerical work in MATLAB.

slide12:
Suppose we define:#v as a row vector equals negative two, three, zero, four point five, and negative one point five.#To make it a column, we write: v equals v prime — using the transpose.
To access elements:#v of one gives the first,#v of two to four gives the second to fourth,#and v of three and five gives just those two.
You can also create sequences.#For example: v equals four to two, stepping by negative one — gives four, three, two.
And combining vectors is easy.#If a equals one to three, and b equals two to three,#then c equals a b — gives one, two, three, two, three.

slide13:
Let’s see how to create and work with matrices in MATLAB.
To get evenly spaced numbers, use linspace.#For example:#x equals linspace from minus pi to pi, with 10 points — gives 10 values across that range.#For logarithmic spacing, use logspace.
To define a matrix, write:#A equals square bracket one two three; four five six — that gives a two-by-three matrix.
To access elements:#A of one, two gives the value in the first row, second column.#A colon, two returns the second column.#A two, colon gives the second row.
You can add, subtract, or scale matrices like:#A plus B, or two times A.
Use A star B for matrix multiplication,#and A dot star B for element-by-element.
Transpose with A prime,#and get the determinant using det of A.

slide14:
MATLAB also makes it easy to generate special-purpose matrices, which are often useful for setting up or testing algorithms.
For example, diag of v creates a diagonal matrix from a vector. You can also reverse it: diag of A pulls the diagonal elements from a matrix.
If you need an identity matrix — that’s a square matrix with ones on the diagonal and zeros elsewhere — just write eye of n.
To create a matrix full of zeros, use zeros of m and n. If you need one filled with ones, use ones of m and n.
These kinds of matrices are especially useful when initializing arrays before filling them with computed values.

slide15:
Now let’s look at logical operations in MATLAB.
You can compare values using operators like:#double equals, less than, greater than, or not equals, which is written as tilde equals.
The tilde symbol means not.#For combining conditions, use ampersand for AND, and

vertical bar for OR.
To find specific values in an array, use the find function.#For example: find A equals three — gives you the positions where A is exactly three.
These logical tools are great for filtering data and writing conditional code.

slide16:
Alright, let’s shift gears and talk about solving systems of linear equations — something that comes up all the time in engineering, physics, and, yes, even in medical imaging.
Suppose we’re working with a system like A times x equals b. Now, you might think — okay, just take the inverse of A and multiply it by b. So, x equals inv of A times b.
And sure, that’s mathematically valid — but in practice, it’s a bad idea. It’s slow, it’s unstable, and it can lead to all kinds of numerical issues, especially with large or ill-conditioned matrices.
Instead, MATLAB gives us a much better tool: the backslash operator. So we write x equals A backslash b. This tells MATLAB to choose the most efficient way to solve the system behind the scenes — no explicit inverse needed.
Later on, we’ll learn how to implement our own solvers, but for now, this built-in method is what you should use. It’s clean, fast, and accurate — everything we want.

slide17:
Now that we’ve seen how to work with equations, let’s take a moment to explore how we can inspect the structure of the data we’re working with — vectors and matrices.
To find out how long a vector is, just use length — so length of v tells you how many elements it has.
If you’re working with a matrix, and you want to know its shape — how many rows and columns — just use size of A.
Want to check if a matrix is full rank? That’s easy — rank of A will do the job.
And when we want to understand how large a matrix or vector is in terms of magnitude, we use the norm function. Just norm of A gives the 2-norm by default — that’s the square root of the sum of squares. You can also ask for specific norms like the 1-norm using norm of A, one, or the infinity norm with norm of A, inf.
These commands give you insight into the structure and stability of your data — and they’re especially useful when you’re preparing data for algorithms or interpreting results.

slide18:
Let’s talk about control flow — specifically, loops.
Start with a for loop. Use it when you know how many times you want to repeat something.
For example:#Set x to zero.#Then, for i from 1 to 5 in steps of 2 — so, 1, 3, and 5 —#we add i to x.#At the end, x equals 9.
for loops are useful when working through a known sequence.
Now, a while loop runs as long as a condition is true.
In this example:#Start with x equal to 7.#While x is greater than or equal to zero, subtract 2 each time.#The loop stops when x becomes negative.
Just be careful — if the condition never becomes false, the loop won’t stop.
Finally, break lets you exit a loop early.#It’s useful if you find what you need before the loop finishes.#In nested loops, it only exits the innermost one.#Use it wisely — don’t skip steps that still matter.

slide19:
Let’s talk about how we make decisions in MATLAB.
First, the if statement.
You might say:#“If x is equal to 3, display: the value of x is 3.#If x is equal to 5, display: the value of x is 5.#Otherwise, display: the value of x is not 3 or 5.”
This allows the program to choose what to do based on the value of x.
Now, when you want to check one variable against several specific values, a switch statement is better.

For example:#“If face is 1, say: rolled a one.#If face is 2, say: rolled a two.#Otherwise, say: rolled a number greater than or equal to three.”
This is easier to read than writing multiple if statements. And in MATLAB, the switch ends automatically — no need for a break.

slide20:
Now let’s talk about vectorization — one of the most important habits in MATLAB programming.
Here’s the idea. Instead of looping through each element one by one, we apply an operation to the whole array at once.
For example:#Let’s say x is the vector — one, two, three.#Using a loop, we would add five to each element, one at a time.#But with vectorization, we just say: x equals x plus five — and MATLAB does it all at once.
Same result, much faster, much cleaner.
Whenever you can, avoid loops. Use vectorized operations. MATLAB is built for that.

slide21:
Let’s look at how MATLAB handles basic plots.
Start by generating x values from minus one to one using linspace. Then take the sine of x to get y.
Use plot x and y to create a basic line graph.
To change the appearance, you can specify style and color. For example, plot x, y, black line draws the line in black.
Use hold on if you want to overlay another plot on the same figure.#Use figure to open a new figure window for a separate plot.
Plotting helps visualize data — and it’s built into MATLAB as a core strength.

slide22:
Sometimes, you want to show several plots in one window.
Use the subplot command to divide the figure into a grid.#For example, subplot 2, 3, 1 creates two rows and three columns — and selects the first cell.
Each subplot acts like an independent plot. You can add titles, labels, and style just like a regular figure.
This is helpful when comparing outputs or organizing results cleanly on a single screen.

slide23:
Now let’s add a third dimension.
Use plot 3 of x, y, z to make a 3D line plot.
For surfaces, use the mesh command — it draws a grid over the surface defined by z values.
If you want a contour plot — use contour z matrix. That shows level curves of a surface.
You can also control the axis limits with axis, add a title with title, and label the axes using x label and y label.
To explain the meaning of different curves, use legend.

slide24:
Here are a few examples of what you can do with MATLAB plots.
You’ll see grouped bar charts, stacked bars, 2D lines, and 3D surfaces — all using basic commands.
Try different styles, colors, and layouts to get the look you want.
Good visualization tells a story — not just numbers, but patterns, relationships, and insights.
Use plots not just to check results, but to communicate them clearly.

slide25:
Now let’s talk about reading and writing data files.
To open a file, use f open, and pass the file name and read mode.#For example: f open in dot dat, r t.
This gives you a file ID — or fid.
Use f scanf to read numbers from the file.#When you're done, always close the file using f close.

If you need formatting help, use help textread or help f printf.
Below is an example of a data file — it has names, types, numbers, and yes or no labels.
You'll often need to work with mixed data — so learning how to read from and write to files is essential in MATLAB workflows.

slide26:
Let's now learn how to read data files in MATLAB — both fully and partially.
To read an entire structured dataset, you first open the file using fopen, then use the textread function to extract data from all columns.
Each data type in the file corresponds to a format symbol — like %s for strings, %f for floating-point numbers, and %d for integers. MATLAB assigns each column to a variable automatically.
This works well when you need to load the whole dataset into memory — like names, types, values, and so on.
Now, if you only want to read part of the data — say just the first column — you can still use textread, but with asterisks added to the format string to skip over the columns you don't need. MATLAB will still move through the file correctly, but it'll only save the data you asked for.
This selective reading is great for reducing memory usage or focusing on relevant features.

slide27:
Sometimes you don't need the full file — you just want a specific line. Maybe line two contains Joe's entry, and that's all you care about.
In this case, you can tell MATLAB to skip one line, and then read just one. You still use textread, but now with two extra options: the number of lines to read, and the number of header lines to skip.
This lets you directly target a record without loading everything — useful for indexed data, or when you're debugging or sampling from a large file.

slide28:
So far we've looked at reading files — now let's switch to writing.
To write data out, open a file in write mode using fopen. Then use fprintf to format the output line, inserting variables in the order you want them saved.
For example, you can save name, type, and numerical values all in one line. And as always, once writing is done, make sure to close the file with fclose.
This gives you full control over the format, spacing, and layout of your data.
Very useful when preparing results for review, or generating output for other software to read.

slide29:
Next, let's see how to keep a record of your work using the diary command.
Start with diary followed by a filename. From that point on, everything you type and everything MATLAB prints will be saved in a log file. Once you're done, stop the recording by typing diary off.
This is great for tracking your workflow, recording sessions, or saving what you've done for future reference or reporting.
Now, another useful tool is timing. If you want to measure how long your code takes to run, just put tic before the commands, and toc after them. MATLAB will show you the elapsed time in seconds.
This helps you compare different methods and optimize performance.

slide30:
One of the key ideas in MATLAB is that everything is a matrix.
A single number? That's just a one-by-one matrix.#A row of numbers is a row vector. A column is a column vector.#And an image? That's a two-dimensional matrix.#If it's in color, it becomes three-dimensional — width, height, and color channels.
This matrix mindset makes MATLAB powerful and consistent across tasks — whether you're doing math, image processing, or even machine learning.
Now, let's talk about indexing. If you have a matrix A:#A of i comma j gives the element in row i and column j.#A of i comma colon gives the entire i-th row.#A of colon comma j gives the whole j-th column.

You can reshape matrices, slice out parts, or use logical masks to filter data.
To modify the matrix, use expressions like A plus five — which adds five to every element.#Or A dot star B — for element-by-element multiplication.
This is the foundation of MATLAB — matrix thinking in everything you do

slide31:
In MATLAB, you can do both matrix and element-wise operations — and the difference is all in the dot.
“A times B” — with a single star — means matrix multiplication.#“A dot star B” — with a dot — means multiply element by element.
Same goes for powers and division.#“A caret 2” raises the matrix to a power.#“A dot caret 2” squares each element.#And for division, “A divided by B” solves equations, while “A dot slash B” divides element-wise.
These small differences are key when working with data, equations, or signals in MATLAB.

slide32:
Whenever you can, avoid loops — and write in matrix form.
Instead of looping to square elements, just write “A dot caret 2”.
Want row-wise sums? Use “sum of A, comma 2”.
Need probabilities per row? Divide each row by its total using matrix broadcasting — it’s cleaner and faster than nested loops.

slide33:
The peaks function creates a sample surface. Use mesh to visualize it.
To smooth it out, create a finer grid with meshgrid, and use interp2 to estimate new Z-values.
This technique is useful in graphics, simulations, and image processing — anytime you want smoother, higher-resolution data.

slide34:
Here’s how we morph faces using matrix tools.
We warp two face images using interpolation on each color channel. Then blend them using a weighted average — with alpha controlling the mix.
To detect unusual pixels, we reshape the image and compute the Mahalanobis distance from a reference color.
Next, we create a binary mask by thresholding the distance, and apply a median filter to clean it up.
It’s a compact example of how MATLAB combines image warping, statistics, and filtering — all with matrices.

slide35:
This example shows how plot resolution affects curve smoothness.
In a loop, we increase the number of x-values using linspace, then compute y equals x over one plus x squared.
We plot each result in a subplot grid, with labels and titles showing the resolution.
With each pause, you can see the graph getting smoother — a great way to visualize the impact of sampling density.

slide36:
Let’s talk about the two main types of files you’ll create in MATLAB — scripts and functions.
Scripts are simple. They run a series of commands and use variables already in your workspace. You don’t pass anything in, and they don’t return anything out.
They’re great for quick experiments or automation.
Functions, on the other hand, are more structured. You give them inputs, and they return outputs. Inside a function, all variables are local — meaning they don’t interfere with anything else in your workspace.
Knowing when to use a script or a function is key to writing clean, reliable code.

slide37:
Here’s a basic function example — one that calculates the area and circumference

of a circle.
We define a function called circle that takes one input: the radius. Inside, we calculate the area as pi times radius squared, and the circumference as two times pi times the radius.
Save this in a file named circle dot m.
Now, to use this function, we write a script. For example, if the radius is 7, we call the function with that value and display the result using disp.
You'd save this script as myscript dot m.
This is how functions and scripts work together. You build reusable tools with functions and run them from scripts.

slide38:
Let's try a simple numerical task: summing one over i squared, from i equals 1 to 10.
We can do this two ways — forward, starting from i equals 1, or backward, starting from i equals 10.
The code is nearly identical, just with the loop going in reverse.
Now, in theory, both results should be the same. But because of floating-point rounding, the order can make a small difference. This is a great example of how numerical stability matters — especially in scientific computing.

slide39:
Now let's write our own matrix multiplication function — no built-in functions allowed.
We create a function called matrix_multiply that takes two square matrices and a size n.
Using three nested loops, we compute the product row by row and column by column. For each entry in the result, we take a dot product between a row of A and a column of B.
This hands-on approach shows exactly how matrix multiplication works under the hood — which is important to understand before we talk about optimization.

slide40:
To test our new function, we write a quick script.
We choose a matrix size n, generate two random n-by-n matrices A and B, and call our matrix_multiply function to compute the result.
You can also compare it to MATLAB's built-in multiplication to check for accuracy.
And here's something to think about — this triple-loop method works, but it's not fast. Later on, we'll talk about how to rewrite this using vectorized operations to make it much more efficient.

slide41:
Now let's look at a real-world scientific task — measuring how stars move.
We start with spectral data from a star. It includes many wavelengths, evenly spaced. The key is to find a feature called the Hydrogen-alpha line. If that line shifts, the star is moving — either away or toward us.
With MATLAB, we'll identify that line, calculate how much it's shifted, and then use the Doppler formula to estimate the star's speed.

slide42:
Here's how we do that in MATLAB.
First, we compute the wavelength range using the start point, number of data points, and spacing. Then, we build a vector of wavelengths.
Next, we plot the spectrum and locate the lowest point — which marks the Hydrogen-alpha line. Its position gives us the shifted wavelength.
Using the Doppler shift formula, we compute the redshift, then multiply by 300,000 — the speed of light in kilometers per second — to get the star's velocity.
Just a few lines of code, and we're measuring motion across space.

slide43:
Let's switch gears to 3D graphics.
Here, we're creating a cone using MATLAB's cylinder function.

We define a shape with the vector: t equals zero, one, zero. This gives us a cone shape.
Then we call the cylinder function to get the X, Y, and Z coordinates, and use surf to draw the 3D surface.
This shows how easily you can build and visualize 3D shapes in MATLAB — helpful for both learning and engineering.

slide44:
Here’s the full example, all in one place.
We define t as a vector with values zero, one, zero. Then we call the cylinder function to generate the geometry, and use surf to plot it.
That’s it — three lines of code give us a smooth 3D cone you can rotate, zoom, and customize. You can even add lights, shadows, or change the viewing angle.
It’s a simple example, but it shows the power of MATLAB for 3D visualization.

slide45:
Now let’s explore the Image Processing Toolbox.
It includes powerful tools for loading, modifying, analyzing, and segmenting images.
You can sharpen edges, enhance contrast, or isolate bright regions. It even lets you analyze object shapes and intensities — which is essential in fields like medical imaging.
You can also use it to register images, align features, and prepare large datasets for analysis. And if needed, you can even accelerate the workflow with C or C++ code.
This toolbox makes MATLAB a great platform for both learning and real-world applications in imaging.

slide46:
Now let’s take a look at the Instrument Control Toolbox.
This toolbox allows MATLAB to talk directly to hardware — like oscilloscopes, sensors, or signal generators. That means you can send commands, receive data, and even control external devices — all from your MATLAB environment.
You can also create your own instrument drivers, manage device sessions, and work with supported protocols like VISA or TCP/IP.
So, if you're working in a lab with physical instruments, this toolbox helps you connect MATLAB to the real world.

slide47:
Next up — the Optimization Toolbox.
This one’s all about solving problems where you want to find the best outcome — like minimizing cost, or maximizing performance.
You can solve linear programs, quadratic programs, and nonlinear problems. It also supports multi-objective optimization.
It’s useful in many fields — from data fitting and machine learning to operations research. If your work involves tuning models or decision-making, this toolbox is essential.

slide48:
Now we move into data science with the Statistics and Machine Learning Toolbox.
This toolbox helps you analyze data, test hypotheses, and build predictive models.
You can use it for clustering, regression, dimensionality reduction, or training classifiers. It also supports parallel computing, which is great for handling large datasets.
And the best part? Many of these features work with just a few lines of code — so you can try powerful techniques without writing complex algorithms from scratch.

slide49:
The Symbolic Math Toolbox brings exact math into MATLAB.
Instead of working with numbers alone, you can define symbolic variables and compute exact derivatives, integrals, or solutions to equations.
For example, you can write a function like f of x equals x squared times sine x

and ask MATLAB to find its derivative. You’ll get the answer in symbolic form — not an approximation.
You can also plot these expressions or export them into Simulink for simulation. It’s a great toolbox for exploring math symbolically, especially in calculus and algebra-heavy problems.

slide50:
Finally, let’s talk about how to keep learning on your own.
There’s a helpful online tutorial by Dr. Azernikov that introduces MATLAB basics step by step. It’s clear and beginner-friendly — a great way to reinforce what we’ve learned here.
Also, don’t forget about MATLAB’s built-in help tools. If your cursor is on a command, just press F1. Or type help followed by the function name in the Command Window.
For more support, you can check the MathWorks website, Stack Overflow, or MATLAB Central. These are great places to ask questions and find real code examples.
In short — keep exploring, keep experimenting, and you'll keep getting better.