

Sure! Here's a simplified version of the video transcript using easy words:

[MUSIC]

Welcome to my course **Numerical Methods for Engineers**.

In this course, we'll use math from other courses—like matrix algebra, differential equations, and vector calculus—and apply it in **scientific computing** using **MATLAB**.

◆ **Week 1:**

We'll learn the basics:

- How computers store numbers
- How to write simple programs in MATLAB

◆ **Week 2:**

We'll learn **root finding**—this means solving equations like $f(x) = 0$ when we can't solve them by hand.

We'll learn the common methods, how fast they are, and how to use MATLAB to find roots.

◆ **Week 3:**

We'll study **matrix algebra**:

- How to solve linear systems
- How to find **eigenvalues** and **eigenvectors**
- How to solve nonlinear systems
This is important for solving **PDEs** later.

◆ **Week 4:**

We'll learn:

- **Quadrature** (numerical integration—finding the area under a curve)
- **Interpolation** (finding values between known data points)
We'll use MATLAB for both, including **cubic spline interpolation** (which works very well).

◆ **Week 5:**

We'll work with **ordinary differential equations (ODEs)**.

These are equations used in many science problems.

We'll learn easy methods and also **Runge-Kutta methods**, and use MATLAB's **ode45.m** tool.

We'll also briefly talk about **boundary value problems**, which use both ODE solving and root finding.

◆ **Week 6:**

We'll learn about **partial differential equations (PDEs)**.

This is the hardest part.

We'll need matrix algebra to solve big systems of equations.

We'll also learn **iterative methods** and some theory like **stability**.

✓ Each week includes:

- Short video lessons
 - Practice problems
 - A quiz and a small coding project
-

🎉 Thanks for joining me in learning **Numerical Methods for Engineers**. Let's get started!

Let me know if you want a PDF or notes version!

Here's a simplified version of your Week 1 introduction in easy words:

Welcome to Week 1 of Numerical Methods for Engineers!

This week, we'll start learning the basics of **scientific computing**.

◆ First, we'll learn:

- How **computers store numbers** using **binary** (just 0s and 1s).
- What **double precision numbers** are – these are the main type of numbers we'll use.

⚠ These numbers have **limits**:

- Some fractions (like $1/3$ or $1/5$) can't be stored exactly.
- Numbers can't be too big or too small, or they may **overflow** or **underflow**.
- This can cause small **errors** in our results.

◆ Then, we'll learn **MATLAB basics**:

- How to use it like a calculator
- How to make **vectors** and **matrices**
- How to **draw plots** (line graphs)
- How to write **scripts** and **functions**
- How to use **logic, if-else statements**, and **loops**

💡 If you already know how to program, this will be easier.
If not, you can learn the basics of MATLAB this week.

🔑 Your **first project** is to create a **bifurcation diagram** for something called the **logistic map**. It's a simple equation, but it creates very complex patterns.

📅 Join me in this exciting first week of learning numerical methods!

Here's a **simple version** of the lecture transcript using easy words:

[MUSIC]

Before we start using computers to do math, we need to learn **how computers store numbers**. We'll cover that in this and the next lecture.

Decimal Numbers (Base-10):

- We normally use **decimal numbers** (also called **base-10**).
- They use **digits from 0 to 9**.
- We likely use base-10 because we have **10 fingers** (digits = fingers!).

Example:

524.503

Each digit's **position** matters:

- $5 \times 10^2 = 500$
- $2 \times 10^1 = 20$
- $4 \times 10^0 = 4$
- $5 \times 10^{-1} = 0.5$
- $0 \times 10^{-2} = 0$
- $3 \times 10^{-3} = 0.003$

So, $524.503 = 500 + 20 + 4 + 0.5 + 0 + 0.003$

 This is called a **positional number system**.

Decimal numbers made math like adding and multiplying **easy**, compared to old systems like **Roman numerals**.

Binary Numbers (Base-2):

- **Computers use binary**, not decimals.
- Binary uses only **0s and 1s**. These are called **bits**.
- Binary is a **base-2** positional system.

Example:

101.011 (binary)

- $1 \times 2^2 = 4$
- $0 \times 2^1 = 0$
- $1 \times 2^0 = 1 \rightarrow$ Total before point = 5

- Binary point = like the decimal point
 - After the point:
 - $0 \times 2^{-1} = 0$
 - $1 \times 2^{-2} = 0.25$
 - $1 \times 2^{-3} = 0.125$
→ Total after point = 0.375
 - 👉 So, 101.011 (binary) = **5.375** in decimal
-

🚫 Some numbers can't be stored exactly:

In base-10:

- $1/2 = 0.5$ ✅
- $1/3 = 0.333...$ ❌ (repeats forever, **inexact**)
- $1/5 = 0.2$ ✅
- $1/7 = 0.142857142857...$ ❌ (repeats forever)

In binary:

- $1/2 = 0.1$ ✅
- $1/3 = 0.010101...$ ❌ (repeats)
- $1/4 = 0.01$ ✅
- $1/7 = 0.001001...$ ❌ (repeats)

These repeating numbers **can't be stored exactly** on a computer.
We call them **inexact numbers**.

⚠️ What does this mean for you?

- When numbers can't be stored exactly, **small errors** can happen.
 - These are called **round-off errors**.
 - Most of the time, they don't matter much.
 - But it's important to **understand** them when writing programs.
-

🧠 Summary:

- Decimal: base-10, digits 0–9
- Binary: base-2, digits 0 and 1 (bits)
- Some fractions can't be stored exactly in decimal or binary → **inexact numbers**

- This can lead to **rounding errors** when computers do math

 I'm Jeff Chesnoff. Thanks for watching, and I'll see you in the next video!

Here's a clear **summary** of the video lecture you just watched on **IEEE Floating Point Arithmetic (Double-Precision)** by Jeff Chasnov:

Key Concepts of IEEE 754 Double-Precision Floating Point Representation (64-bit)

1. Bit Structure

A double-precision number uses **64 bits**:

- **1 bit for Sign (S)**
 - 0 → Positive
 - 1 → Negative
- **11 bits for Exponent (E)**
 - Stored as **biased exponent**: actual exponent = $E - 1023$
- **52 bits for Fraction (F)** (also called the **significand** or **mantissa**)
 - Value is stored as 1.F in binary (with implicit leading 1)

Formula:

$$(-1)^S \times 1.F \times 2^{(E-1023)} \quad (-1)^S \times 1.F \times 2^{\{E - 1023\}}$$

2. Special/Reserved Values

- **Infinity ($\pm\infty$)**
 - E = all 1s (2047) and F = all 0s
 - Example: Division by 0 (e.g., $1 / 0$)
- **NaN (Not a Number)**
 - E = all 1s and F ≠ all 0s
 - Example: $0 / 0$, $\infty - \infty$
- **Denormal/Subnormal Numbers**
 - E = all 0s
 - Value is computed as:

$$(-1)^S \times 0.F \times 2^{-1022} \quad (-1)^S \times 0.F \times 2^{\{-1022\}}$$

- Used to represent numbers **very close to zero** (graceful underflow)

- **Zero**
 - All bits = 0 (positive zero)
 - Sign bit = 1 (negative zero)

3. Important Limits

- **realmax (Maximum Representable Number)**

$\approx 1.797 \times 10^{308}$

- **realmin (Smallest Normal Positive Number)**

$\approx 2.2251 \times 10^{-308}$

4. Machine Epsilon (ϵ)

- Smallest difference such that:

$1 + \epsilon \neq 1$

- For double precision:

$\epsilon \approx 2.22 \times 10^{-16}$

- **Used to detect roundoff errors**

5. Sources of Roundoff Error

- **Adding a very small number to a large one**
 - If $|y| \ll |x|$, then $x + y \approx x$
- **Subtracting nearly equal numbers**
 - Can lose significant digits ("catastrophic cancellation")

6. Why IEEE Standard is Powerful

- Represents:
 - Very large numbers (up to 10^{308})
 - Very small numbers (down to 10^{-308})
 - Precise values with up to 16 decimal digits of accuracy
 - Special values: ∞ , NaN, subnormals, etc.

Let me know if you want a **diagram**, **practice questions**, or help with **binary-to-decimal examples** using IEEE 754 format.

Here's a **summary** of the key points from the MATLAB video tutorial by **Jeff Chasnov** on **writing mathematical expressions in MATLAB**:

Basic Operators in MATLAB

| Symbol | Operation | Example |
|--------|--------------------------|-----------|
| + | Addition | $3 + 2$ |
| - | Subtraction | $5 - 1$ |
| * | Multiplication | $4 * 3$ |
| / | Division | $10 / 2$ |
| ^ | Exponentiation | $2^3 = 8$ |
| ' | Transpose (for matrices) | A' |

Common Functions and Constants

| Function/Constant | Meaning | Example |
|-----------------------------------|--|-----------------------|
| $\sin(x)$, $\cos(x)$, $\tan(x)$ | Trigonometric functions | $\sin(\pi/2) = 1$ |
| $\log(x)$ | Natural logarithm (ln) | $\log(\exp(1)) = 1$ |
| $\log_{10}(x)$ | Base-10 logarithm | $\log_{10}(1000) = 3$ |
| $\text{sqrt}(x)$ | Square root | $\text{sqrt}(16) = 4$ |
| $\text{abs}(x)$ | Absolute value or modulus of complex number | $\text{abs}(-5) = 5$ |
| π | $\pi = 3.1416\dots$ | |
| i or j | Imaginary unit | $1 + 2i$ |
| inf | Infinity | $1/0 = \text{Inf}$ |
| NaN | Not a Number (e.g. $0/0$, $\text{Inf} - \text{Inf}$) | |

Examples Demonstrated

- **Euler's Number:**
- $\exp(1) \approx 2.7183$

- $\log(\exp(1))$ % = 1
- **Golden Ratio (Φ):**
- $\Phi = (\sqrt{5} + 1) / 2$; % ≈ 1.618
- $\phi = (\sqrt{5} - 1) / 2$; % ≈ 0.618
- $\Phi * \phi$ % ≈ 1 (with a small roundoff error)
- **Roundoff Errors:**
 - Multiplying irrational numbers may give slight errors (e.g., not exactly 1).
 - Use format hex to view floating-point representation.

Special Numbers

| Operation | Result |
|--------------------|--------|
| 1 / 0 | Inf |
| -1 / 0 | -Inf |
| 0 / 0 or Inf - Inf | NaN |

Formatting and Output

Ηερεεσ α στρυχυτρεδ **συμμαρψ** οφ τηε τηιρδ ΜΑΤΛΑΒ πιδεδ βψ **θεδφ Χηασνω**, φοχυσεδ ον **πεχτορσ** ανδ **πεχτορσ** οπερατιονσ ιν **ΜΑΤΛΑΒ**:

☐ Βασικ Δατα Τυπε: Τηε Ματριξ

Ιν ΜΑΤΛΑΒ, **επερψτηιγγ ισ α ματριξ** ☐ ινχλυδιγγ **πεχτορσ**, ωηιχη αρε 1Δ ματριχεσ.

| Τυπε | Δεδφινιτιον | Εξαμπλε |
|----------------------|-------------------------|--|
| Ροω ξεχτορ | 1 ροω, μυλτιπλε χολυμνσ | $\xi = [1 \ 2 \ 3]$ ορ $\xi = 1 \ 2 \ 3$ |
| Χολυμν ξεχτορ | Μυλτιπλε ροωσ, 1 χολυμν | $\xi = [1; 2; 3]$ |

Ψου χαν **αχχεσσ ελεμεντσ** τηε σαμε ωαψ φορ βοτη:

$\xi(3)$ % ρετυρνσ τηε 3ρδ ελεμεντ

☐ Τρανσποσε ξεχτορσ

Υσε ε (αποστροπηε) το τρανσποσε:

$\xi = [1; 2; 3];$ % χολυμν πεχτορ

ξ' % βεχομεσ ροω πεχτορ $[1 \ 2 \ 3]$

□ **ζεχτορ Γενερατιον Φυνχτιονσ**

| Φυνχτιον | Πυρποσε | Εξαμπλε |
|-----------------|--|--|
| ζεροσ(μ,ν) | Ματριξ (ορ πεχτορ) οφ αλλ ζεροσ | $\xi = \text{ζεροσ}(1,3) \Rightarrow [0 \ 0 \ 0]$ |
| ονεσ(μ,ν) | Ματριξ (ορ πεχτορ) οφ αλλ ονεσ | $\xi = \text{ονεσ}(1,3) \Rightarrow [1 \ 1 \ 1]$ |
| λινσπαχε(α,β,ν) | ν εθυαλλψ σπαχεδ παλυεσ βετωεεν α α νδ β | $\text{λινσπαχε}(0,1,6) \Rightarrow [0 \ 0.2 \ 0.4 \ 0.6 \ 0.8 \ 1]$ |
| Χολον νοτατιον | Δεφινε σταρτ:στεπ:ενδ | $\xi = 0:0.2:1 \Rightarrow \text{σαμε ασ αβοπε}$ |

- λινσπαχε(0,1,6) □ νυμβερ οφ ποιιντσ
- 0:0.2:1 □ στεπ σιζε

□ **Ελεμεντ–Ωισε Οπερατιονσ**

Φορ χομπονεντ–ωισε οπερατιονσ ον πεχτορσ, υσε α **δοτ (.)** βεφορε τηε οπερατορ:

| Οπερατιον | Συνταξ | Εξαμπλε | Ρεσουλτ |
|----------------|---------------|---------------------|---------------|
| Μυλτιπλιχατιον | \cdot | $\xi \cdot \xi$ | $[1 \ 4 \ 9]$ |
| Ποωερ | $\cdot \perp$ | $\xi \cdot \perp 2$ | $[1 \ 4 \ 9]$ |
| Διωισιον | $\cdot /$ | $\xi \cdot / \xi$ | $[1 \ 1 \ 1]$ |

Εξαμπλε:

$\xi = [1 \ 2 \ 3];$

$\xi \cdot \xi \quad \% [1 \ 4 \ 9]$

$\xi \cdot \perp 2 \quad \% [1 \ 4 \ 9]$

$\xi \cdot / \xi \quad \% [1 \ 1 \ 1]$

□ **Φυνχτιονσ Αππλιεδ το ζεχτορσ**

Μοστ ΜΑΤΛΑΒ φυνχτιονσ **αχχεπτ πεχτορσ**:

$\xi = 0:\pi/2:2*\pi; \quad \% [0 \ \pi/2 \ \pi \ 3\pi/2 \ 2\pi]$

χοσ(ξ) $\quad \% \text{ρετυρνσ } [1 \ 0 \ -1 \ 0 \ 1] \text{ (αππροξ.)}$

 Τησεσ αρε νοτ **εξαχτ** δυε το **ρουνδ–οφφ ερρορσ** (ε.γ., χοσ(π/2) Η 6.1232ε–17 ινστεαδ οφ 0).

□ **ζεχτορσ ιν Πλοτινγ**

- Ιν νυμεριχαλ χομπυτινγ, φυνχτιονσ αρε **επαλυατεδ ασ πεχτορσ**.

- Φορ πλοττινγ:

Ψου τυπιχαλλψ πλοτ ψ παλυεσ (φυνχτιον παλυεσ) **αγαινστ** ξ παλυεσ (δομαιν ποιιντσ).

□ Χονχλυσιον

∀Μανψ νυμεριχαλ σολυτιονσ τακε τηε φορμ οφ πεχτορσ □ Ωεαλλ σεε ηοω το πλοτ φυνχτιονσ νεξτ . ▽

□ Θεφφ Χηασνοπ

Ωουλδ ψου λικε α **πραχτιχε σηεετ** ον πεχτορσ ορ σομε **μινι-εξερχισεσ** το αππλψ ωηατ ψουεπε λε αρνεδ?

Here's a structured **summary** of Jeff Chasnov's MATLAB video on **plotting graphs (specifically a circle)** using plot and customizing it:

Plotting Basics in MATLAB

- Most solutions are visualized using **line plots**.
- Commonly, we plot **y vs x**, i.e., a graph of $y = f(x)$.

● Example: Plotting a Circle

1. Set up polar coordinates

```
theta = linspace(0, 2*pi); % 100 points from 0 to 2π (default)
```

```
x = cos(theta);
```

```
y = sin(theta);
```

2. Plot the circle

```
plot(x, y)
```

▲ But this appears **oval**, because MATLAB **scales the x and y axes differently by default**.

Fixing Plot Appearance

3. Make axes equally scaled

```
axis equal
```

✓ Now it **looks like a circle**, but spacing might still be off.

4. Set axis limits manually

```
axis([-1.1 1.1 -1.1 1.1]) % [xmin xmax ymin ymax]
```

✅ Now both axes show the same range.

5. Customize Tick Marks

Use `gca` (Get Current Axis) and **dot notation** to access axis properties:

```
ax = gca;           % get current axis handle
ax.XTick = [-1 -0.5 0 0.5 1]; % set x-axis tick marks
ax.YTick = [-1 -0.5 0 0.5 1]; % set y-axis tick marks
```

📌 Add Labels and Title (Using LaTeX Style)

Use dollar signs to format as LaTeX math in labels:

```
xlabel('$x$', 'interpreter', 'latex', 'fontsize', 14)
ylabel('$y$', 'interpreter', 'latex', 'fontsize', 14)
title('Circle Plot', 'fontsize', 16)
```

💡 You can skip LaTeX formatting if unfamiliar, or just use simple strings.

📖 MATLAB Help and Documentation

- Use `help` or `doc` to learn about functions:
 - `help plot`
 - `doc axis`
 - `doc gca`
 - Most plotting customization requires **checking documentation or Google**.
-

✅ Final Code (Clean Circle Plot)

```
theta = linspace(0, 2*pi);
x = cos(theta);
y = sin(theta);

plot(x, y)
axis equal
axis([-1.1 1.1 -1.1 1.1])
```

```

ax = gca;
ax.XTick = [-1 -0.5 0 0.5 1];
ax.YTick = [-1 -0.5 0 0.5 1];

xlabel('$x$', 'interpreter', 'latex', 'fontsize', 14)
ylabel('$y$', 'interpreter', 'latex', 'fontsize', 14)
title('Circle Plot', 'fontsize', 16)

```

Would you like a **hands-on task** to practice plotting functions like $\sin(x)$, e^x , or even a spiral?

Here's a **structured summary** of Jeff Chasnov's video on **matrix operations in MATLAB**:

What is MATLAB?

- **MATLAB** stands for **Matrix Laboratory**.
 - **Matrices** are the **core data structure** in MATLAB.
 - MATLAB provides many built-in **functions** for creating and manipulating matrices.
-

Creating Matrices

| Function | Description | Example |
|-------------------------|---|---|
| <code>zeros(m,n)</code> | $m \times n$ matrix of all zeros | <code>zeros(3,2)</code> → 3×2 matrix |
| <code>ones(m,n)</code> | $m \times n$ matrix of all ones | <code>ones(2,3)</code> → 2×3 matrix |
| <code>eye(n)</code> | $n \times n$ identity matrix | <code>eye(2)</code> → 2×2 identity |
| <code>diag(v)</code> | Diagonal matrix with elements of vector v | <code>diag([1 2])</code> → 2×2 with 1, 2 on diagonal |

Accessing Matrix Elements

Example Matrix A:

```

A = [1 2 3;
     4 5 6;
     7 8 9];

```

Element Access:

| Action | Syntax | Result |
|----------------------------|--------------------------|------------------------------------|
| Access single element | <code>A(2,3)</code> | 6 |
| Submatrix | <code>A(1:2, 2:3)</code> | <code>[2 3; 5 6]</code> |
| First row | <code>A(1, :)</code> | <code>[1 2 3]</code> |
| First column | <code>A(:, 1)</code> | <code>[1; 4; 7]</code> |
| Single index (column-wise) | <code>A(5)</code> | 5 (fifth element down column-wise) |
| Modify element | <code>A(1,1) = 10</code> | A is updated |

Colon Operator :

- Used to select entire rows/columns or ranges.
 - `A(1,:)` % all elements in row 1
 - `A(:,3)` % all elements in column 3
 - `A(1, 2:end)` % elements from 2nd to last in row 1
-

Matrix Multiplication

`A = [1 2; 3 4];`

`B = [-3 -4; 4 5];`

`C = A * B;`

✓ Valid because inner dimensions match: $(2 \times 2) * (2 \times 2) \rightarrow 2 \times 2$

✗ Invalid: `x = [1 2 3]; x * x;` → Error! $(1 \times 3) * (1 \times 3)$ not valid.

✓ Valid scalar product: `x * x'` → $(1 \times 3) * (3 \times 1) = \text{scalar } 14$

Special Matrix: Banded Matrix

Create banded matrix (main diagonal and off-diagonals):

`n = 4;`

`A = diag(2*ones(1,n)) + diag(-1*ones(1,n-1), 1) + diag(-1*ones(1,n-1), -1);`

✓ Produces a 4×4 matrix with:

- 2 on main diagonal
- -1 above and below the diagonal

Applications

- Matrix operations are **fundamental** in:
 - Solving **nonlinear systems**
 - Solving **PDEs**
 - Performing **numerical methods**
-

Summary Table

| Operation | Syntax |
|--------------------|---|
| Create zero matrix | <code>zeros(m,n)</code> |
| Create ones matrix | <code>ones(m,n)</code> |
| Identity matrix | <code>eye(n)</code> |
| Diagonal matrix | <code>diag([a b c])</code> |
| Access row/column | <code>A(1,:)</code> , <code>A(:,2)</code> |
| Multiply matrices | <code>A * B</code> |
| Transpose | <code>x'</code> |
| Dot/scalar product | <code>x * x'</code> |

Would you like a **practice task** or code quiz based on matrix manipulation in MATLAB?

Here's a structured summary of Jeff Chasnov's video on **Logical Variables and Operators in MATLAB**:

Logical Variables in MATLAB

- Logical variables can take only **two values**:
 - 1 → **true**
 - 0 → **false**
-

Logical Operators

| Operator | Meaning | Example | Result |
|----------|--------------------------|----------------|--------|
| > | Greater than | $3 > 2$ | 1 |
| < | Less than | $2 < 3$ | 1 |
| >= | Greater than or equal to | $3 \geq 3$ | 1 |
| <= | Less than or equal to | $2 \leq 1$ | 0 |
| == | Equal to | $4 == 4$ | 1 |
| ~= | Not equal to | $4 \neq 5$ | 1 |
| ~ | NOT | $\sim(1 == 1)$ | 0 |
| & | AND | $1 \& 0$ | 0 |
| | OR | $1 0$ | 1 |

⚠ Note: == is for comparison, = is for assignment.

📌 Examples in MATLAB

`p = 0 < 1; % p = 1 (true)`

`q = 0 > 1; % q = 0 (false)`

- These are **logical variables** and can be used in computations:

`p * 5 % Result: 5, because 1 * 5 = 5`

`q * 5 % Result: 0, because 0 * 5 = 0`

🧠 Logical Arrays (Element-wise Comparison)

`x = [0 1 2 0];`

`y = [0 2 4 0];`

`z = (x == y); % Compare element-wise`

Result:

`z = [1 0 0 1]`

- $0 == 0 \rightarrow 1$
- $1 == 2 \rightarrow 0$
- $2 == 4 \rightarrow 0$
- $0 == 0 \rightarrow 1$

Operator Precedence: Logical vs Arithmetic

Example:

```
result = 10 / 2 == 5; % TRUE (1)
```

- 10 / 2 is done **first**, then compared to 5.
- To make it clear, use parentheses:

```
result = (10 / 2) == 5;
```

 **Recommended:** Use **extra parentheses** for clarity and safety.

Why Logical Variables Matter

- Logical expressions are used in **conditional statements**:
 - if, else, while, etc.
 - These control the **flow of your programs**.
-

Would you like a **mini MATLAB quiz** or **practice exercise** on logical operations?

Here's a clear summary of **Jeffrey Chasnov's video** on **Conditional Statements in MATLAB**:

What Are Conditional Statements?

- Conditional statements allow MATLAB to **decide which code to execute** based on whether a **logical expression is true or false**.
 - These are **core elements** of programming logic.
-

Types of Conditional Statements in MATLAB

1. if ... end

```
if condition
```

```
    % Code runs if condition is true
```

```
end
```

- If the condition is **true** → execute the code.
 - If the condition is **false** → **do nothing**.
-

2. if ... else ... end

if condition

 % Code runs if condition is true

else

 % Code runs if condition is false

end

- Either one **or** the other block of code will run, depending on the condition.
-

Example: sinc Function Fix

We want to define:

$\text{my_sinc}(x) = \begin{cases} \frac{\sin(x)}{x}, & x \neq 0 \\ 1, & x = 0 \end{cases}$

MATLAB Code:

```
function y = my_sinc(x)
```

```
    if x ~= 0
```

```
        y = sin(x) / x;
```

```
    else
```

```
        y = 1;
```

```
    end
```

```
end
```

- If $x \neq 0 \rightarrow$ normal formula
 - If $x = 0 \rightarrow$ return 1 (the correct mathematical limit)
-

Test Cases

```
my_sinc(pi/4) % ≈ 0.9036 (same as sin(pi/4)/pi/4)
```

```
my_sinc(0) % = 1
```

Simplified Version (Optional)

```
function y = my_sinc(x)
```

```
    if x
```

```
        y = sin(x) / x;
```

```
    else
```

```
y = 1;
end
end
```

- if x is **true** when $x \neq 0$
 - Still behaves the same way
 - More concise, but slightly less readable
-

Why Conditional Statements Matter

- Used to **control the program's flow**
 - Essential in **functions, loops, and decision-making logic**
-

Would you like a short **practice problem or quiz** using conditional statements in MATLAB?

Here is a structured summary of **Jeffrey Chasnov's video** on **Iteration in MATLAB**:

Iteration in MATLAB

Iteration is essential in mathematical algorithms for repeatedly performing computations.

 **MATLAB provides two main loop structures:**

1. for Loop (Fixed number of iterations)

Syntax:

```
for n = 1:N
    % Code to execute N times
end
```

- n takes values from 1 to N.
 - Commonly used for **predefined number of steps**.
-

Example: Logistic Map with for Loop

```
n_times = 10000;
x = 0.5;
r = 2.5;
```

```
for n = 1:n_times
    x = r * x * (1 - x);
end
```

```
disp(x)
```



- Change initial condition: $x = 0.1 \rightarrow$ still converges to 0.6
 - Increase n_times: still 0.6
 - Change r to 2.6 \rightarrow different result (shows sensitivity to parameters)
-

2. while Loop (Condition-based iteration)



```
while condition
    % Code to execute while condition is true
end
```

- Runs until a **condition becomes false**.
 - Often used for **convergence checks** or **unknown number of iterations**.
-

Example: Convergence with while Loop

```
x = 0.1;
r = 2.5;
tol = 1e-8;
error = 2 * tol;

while error > tol
    x_old = x;
    x = r * x * (1 - x);
    error = abs(x - x_old);
end

disp(x)
```

- Iterates until the change between x values is smaller than tol.
 - Also converges to 0.6 for $r = 2.5$
-

🛑 break and continue

- break: exits the loop immediately
- continue: skips to the next iteration

Rarely used, but useful in certain control flow conditions.

🧠 Key Insights

- Most computational code runs **inside loops**, often for thousands of iterations.
 - Loop code must be **efficient and correct**.
 - Prefer for loop for **fixed iterations**, and while loop for **convergence-based iteration**.
-

Would you like a **practice exercise** on loops in MATLAB to solidify your understanding?

Here's a **clear summary** of Professor Jeff Chasnov's lecture on the **bifurcation diagram for the logistic map** and the **theory behind it**:

📖 Project Overview: Bifurcation Diagram for the Logistic Map

🧩 The Logistic Map Equation

$$x_{n+1} = \mu x_n (1 - x_n) \quad x_{n+1} = \mu x_n (1 - x_n)$$

- A **nonlinear recurrence relation** (or iterative map)
 - μ is a **parameter**, typically varied from 0 to 4
 - Used to model population dynamics
-

🔄 Iteration Concept

- Start with an initial value x_0
- Apply the function repeatedly to get:

$$x_1 = f(x_0), x_2 = f(x_1), x_3 = f(x_2), \dots, x_{n+1} = f(x_n), \quad x_2 = f(x_1), \quad x_3 = f(x_2), \quad \dots$$

🎯 Fixed Points

A **fixed point** x^* satisfies:

$$x^* = f(x^*) = \mu x^* (1 - x^*) \Rightarrow x^* = \frac{1}{1 - \mu} (1 - x^*)$$

Solving gives:

- $x^* = 0$
- $x^* = 1 - \frac{1}{\mu}$ (when $\mu \neq 0$)

✓ Stability of Fixed Points (Linear Stability Analysis)

Let:

$$x_{n+1} = x_n + \epsilon_n$$

Apply Taylor expansion and linearize:

$$x_{n+1} - x^* \approx f'(x^*) (x_n - x^*) \Rightarrow \frac{|x_{n+1} - x^*|}{|x_n - x^*|} \approx |f'(x^*)|$$

- **Stable:** if $|f'(x^*)| < 1$
- **Unstable:** if $|f'(x^*)| > 1$

📌 Derivatives and Stability

$$f(x) = \mu x(1-x) \Rightarrow f'(x) = \mu(1-2x) \Rightarrow f'(0) = \mu$$

1. For $x^* = 0$:

$$f'(0) = \mu \Rightarrow \text{Stable if } \mu < 1$$

2. For $x^* = 1 - \frac{1}{\mu}$:

$$f'(x^*) = \mu(1-2(1-\frac{1}{\mu})) = 2-\mu \Rightarrow \text{Stable if } 1 < \mu < 2$$

🔗 Bifurcation and Chaos

When $\mu > 3$:

- The second fixed point becomes unstable.
- The system **bifurcates**: exhibits a **period-2 cycle**: alternates between two values.

Further increases in μ :

- Period-4 cycle
- Period-8 cycle
- Eventually, the system becomes **chaotic**: unpredictable and sensitive to initial conditions.

This phenomenon is called a **period-doubling route to chaos**.

Bifurcation Diagram

- Horizontal axis: μ (0 to 4)
 - Vertical axis: Long-term values of x_n
 - Shows the transition from:
 - Stability (fixed point)
 - to periodicity (cycles)
 - to chaos
-

Your Task

Compute and **plot** the bifurcation diagram of the logistic map:

1. Vary μ from 0 to 4 (e.g. in small steps like 0.001)
 2. For each μ , iterate the logistic map (e.g., 1000 iterations)
 3. Discard the first few hundred iterations (transients)
 4. Plot the remaining x_n values vs μ
-

Would you like a **sample MATLAB code** to start with for plotting the bifurcation diagram?

Thanks for sharing the project instructions. Here's a **clear summary** of what you're being asked to do and **how to structure your code** for computing the **bifurcation diagram of the logistic map**:

Objective:

Compute and visualize the **bifurcation diagram** of the logistic map:

$$x_{n+1} = \mu x_n (1 - x_n) \quad x_{n+1} = \mu x_n (1 - x_n)$$

For values of μ from **2.4 to 4**.

Three-Loop Structure:

Outer Loop (Loop 1): Loop over μ

- Range: $\mu \in [2.4, 4.0]$
- Use `linspace(2.4, 4.0, num_mu)` to define the resolution in μ

- Set $x = 0.5$ (initial condition) at the start of each μ

✓ Middle Loop (Loop 2): Transient iterations

- Iterate the logistic map for `num_transient` steps
- Purpose: discard early iterations to let system settle

✓ Inner Loop (Loop 3): Data collection

- Iterate `num_data` more steps
- Store x values for plotting (or binning if using density plot)

🔧 Parameters to Set:

1. `num_mu` — how many μ values you simulate (e.g. 1000)
2. `num_transient` — number of transient iterations to discard (e.g. 100)
3. `num_data` — number of iterations to record (e.g. 100)
4. `x0 = 0.5` — initial value of x
5. (Optional for pixel plotting) `num_bins_x` — number of bins along the x-axis

🖼️ Plotting (Two Options):

1. Simple Scatter Plot (Good for learning)

- Just plot x values for each μ
- In MATLAB:

```
plot(mu_val * ones(1, num_data), x_data, '.k', 'MarkerSize', 1)
```

2. Density Plot (Better visualization)

- Bin the values of x at each μ
- Count how many values fall in each bin
- Shade pixels according to density
- Use provided code to generate the density image

✓ Minimal MATLAB Code Skeleton:

```
mu_vals = linspace(2.4, 4.0, 1000);
```

```
x0 = 0.5;
```

```
num_transient = 100;
```

```
num_data = 100;
```

```

all_mu = [];
all_x = [];

for mu = mu_vals
    x = x0;

    % Loop 2: Transient iterations
    for i = 1:num_transient
        x = mu * x * (1 - x);
    end

    % Loop 3: Data collection
    for i = 1:num_data
        x = mu * x * (1 - x);
        all_mu(end+1) = mu; %#ok<SAGROW>
        all_x(end+1) = x; %#ok<SAGROW>
    end
end

plot(all_mu, all_x, '.k', 'MarkerSize', 1);
xlabel('\mu');
ylabel('x');
title('Bifurcation diagram of the logistic map');

```



Summary of What You Learned:

- The logistic map has two fixed points: $x=0$ and $x=1-\frac{1}{\mu}$
- Stability of fixed points is determined by $|f'(x^*)| < 1$
- As μ increases:
 - Stable fixed point \rightarrow Period 2 \rightarrow Period 4 \rightarrow Chaos
- Bifurcation diagram shows how the long-term behavior of x_n changes with μ

If you'd like, I can help you:

- Translate this to Python code instead of MATLAB
- Add the density plot version
- Add a color-coded image version for clarity

Let me know how you'd like to proceed!

Thanks for sharing the Week 2 overview of **Numerical Methods for Engineers** — this week focuses on **root-finding**, a fundamental part of computational mathematics and engineering. Here's a structured summary with key concepts, algorithms, and what to expect in your second project:

Root-Finding Overview



Root-finding is about solving equations of the form:

$$f(x) = 0$$


Many numerical problems (like optimization, nonlinear equations, or physical simulations) can be **reformulated** into root-finding problems.

Three Root-Finding Algorithms:

1. Bisection Method

-  **Guaranteed to converge**
-  **Slow (linear convergence)**
- Requires that $f(a)$ and $f(b)$ have **opposite signs** (i.e., the root is bracketed)
- Procedure:
 - Compute midpoint: $c = \frac{a + b}{2}$
 - Replace a or b based on the sign of $f(c)$


2. Newton's Method

-  **Fast (quadratic convergence)**
- Requires:
 - A good **initial guess** x_0
 - The **derivative** $f'(x)$
- Iteration formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- Can fail if derivative is 0 or initial guess is poor

3. Secant Method

-  Faster than bisection, almost as fast as Newton
- Doesn't require $f'(x)$
- Uses two initial guesses: x_0, x_1
- Iteration formula:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

MATLAB Root-Finding Tools:

◆ `roots(poly_coeffs)`

- Computes **all roots of a polynomial**
- Input: coefficient vector of a polynomial
- `roots([1 -3 2])` % Solves $x^2 - 3x + 2 = 0$

◆ `fzero(function_handle, initial_guess)`

- Finds a single root of a **nonlinear** function
 - `f = @(x) x^3 - x - 2;`
 - `root = fzero(f, 1.5);`
-

Week 2 Project: Compute the Feigenbaum Delta

- You'll use **Newton's method** to compute the **Feigenbaum delta**, a universal constant in chaos theory.
 - It's related to the geometric convergence of bifurcation points in the logistic map.
 - You'll likely:
 - Write Newton's method code in MATLAB
 - Apply it to a recurrence or fixed-point equation derived from the bifurcation points
-

Feigenbaum Delta (Quick Insight):

- Describes the rate at which period-doubling bifurcations accumulate in maps like:

$$x_{n+1} = \mu x_n (1 - x_n)$$

- Delta value:

$$\delta = \lim_{n \rightarrow \infty} \frac{\mu_n - \mu_{n-1}}{\mu_{n+1} - \mu_n} \approx 4.669 \quad \delta = \lim_{n \rightarrow \infty} \frac{\mu_n - \mu_{n-1}}{\mu_{n+1} - \mu_n} \approx 4.669$$

✅ Summary of Skills You'll Gain in Week 2:

- Implement bisection, Newton, and secant methods
 - Understand convergence rates and tradeoffs
 - Use MATLAB built-in functions for root-finding
 - Apply Newton's method to chaos theory and discover Feigenbaum constants
-

Would you like help implementing any of the methods (e.g., Newton's method) in MATLAB for your project?

Great! Here's a clear and structured summary of **the Bisection Method** as explained by Jeff Chasnov in the video:

📊 Bisection Method – Root Finding

🎯 Goal

Solve the equation:

$$f(x) = 0$$

Find the root $x = r$, where the function crosses the x-axis (i.e., where $y = f(x) = 0$).

📌 Key Idea

- You need **two initial guesses**: x_0 and x_1
- These two values must **bracket the root**, i.e.:

$$f(x_0) \cdot f(x_1) < 0$$

(i.e., the signs of $f(x_0)$ and $f(x_1)$ must be opposite)

🔄 Iteration Process

1. Compute the midpoint:

$$x_{\text{mid}} = \frac{x_0 + x_1}{2}$$

2. Evaluate $f(x_{\text{mid}})$

3. Replace either x_0 or x_1 with x_{mid} so that the root remains bracketed

4. Repeat until:

$$|x_1 - x_0| < \epsilon (\text{desired tolerance}) \quad |x_1 - x_0| < \epsilon \quad \text{\texttt{\text{(desired tolerance)}}$$

Example: Find $2\sqrt{2}$

Let $f(x) = x^2 - 2$, so the root is $2 \approx 1.41421\sqrt{2} \approx 1.41421$

- Initial guesses: $x_0 = 1, x_1 = 2$
- Midpoint $x_2 = (1+2)/2 = 1.5$, $f(1.5) = 0.25$ → root is between 1 and 1.5
- $x_3 = (1+1.5)/2 = 1.25$, $f(1.25) = -0.4375$ → root is between 1.25 and 1.5
- $x_4 = (1.25+1.5)/2 = 1.375$, $f(1.375) = -0.109$
- $x_5 = (1.375+1.5)/2 = 1.4375$, etc.

As you keep iterating, you converge slowly toward $2\sqrt{2}$.

Pros and Cons



Pros

Guaranteed to converge

Simple to implement

Doesn't require derivative



Cons

Slow convergence (linear rate)

Not ideal for high-speed applications

Summary

- Bisection method uses **interval halving** to converge to a root
 - Always **preserves bracketing**
 - Converges **linearly**: interval size halves each iteration
 - Very **robust**, though **slow**
-

Would you like a simple MATLAB or Python code example for the Bisection method?

Here's a clear and complete **summary of Newton's Method** for root finding, as explained by Jeff Chasnov:

Newton's Method – Root Finding Using Calculus

Goal

Solve:

$$f(x) = 0$$

Find a root $x = r$, where the function intersects the x-axis.

Key Idea

Use the **tangent line** at a current point x_n to approximate the function and find the next guess x_{n+1} .

This uses **derivatives** (calculus) to get rapid convergence.

Derivation

From the tangent line at x_n :

$$y - f(x_n) = f'(x_n)(x - x_n) \quad y - f(x_n) = f'(x_n)(x - x_n)$$

Set $y = 0$ (since we want the root), and solve for x :

$$0 - f(x_n) = f'(x_n)(x_{n+1} - x_n) \quad 0 - f(x_n) = f'(x_n)(x_{n+1} - x_n) \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton's Iteration Formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Requirements

- A **function** $f(x)$
 - Its **derivative** $f'(x)$
 - A **good initial guess** x_0 close to the root
(If too far, Newton's method can diverge.)
-

Example: Find $2\sqrt{2}$

Let:

$$f(x) = x^2 - 2, f'(x) = 2x \quad f(x) = x^2 - 2, \quad f'(x) = 2x$$

Iteration Formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{x_n^2 + 2}{2x_n}$$

Start with $x_0 = 1$:

- $x_1 = \frac{1^2 + 2}{2 \cdot 1} = \frac{3}{2} = 1.5$

- $x_2 = (3/2)^2 + 2 \cdot 3/2 = 17/12 \approx 1.416$
 $x_2 = \frac{(3/2)^2 + 2 \cdot 3/2}{1} = \frac{17}{12} \approx 1.416$
- $x_3 = (17/12)^2 + 2 \cdot 17/12 = 577/408 \approx 1.41426$
 $x_3 = \frac{(17/12)^2 + 2 \cdot 17/12}{1} = \frac{577}{408} \approx 1.41426$

✅ After only **3 iterations**, the approximation is very close to:

$$2 \approx 1.41421 \sqrt{2} \approx 1.41421$$

Pros and Cons

✅ Pros

Very fast convergence (quadratic)

Simple and elegant iteration

Ideal for problems needing speed

❌ Cons

Requires derivative $f'(x)$

Can diverge with bad initial guess

Doesn't always converge globally

Summary

- Newton's method uses the **slope of the tangent line** to find the next approximation
- Formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- **Fastest converging** method if the initial guess is good
- Widely used in numerical applications requiring high precision quickly

Would you like a MATLAB or Python implementation for Newton's method as well?

Here's a **clear summary of the Secant Method** for root finding, as explained by Jeff Chasnov:

Secant Method – Root Finding Without Derivatives

Goal

Solve:

$$f(x) = 0$$

Find a root (x-intercept), like with Newton's method, but **without needing the derivative** $f'(x)$.

When to Use It

Use the **Secant Method** when:

- You want fast convergence.
 - You **cannot compute the derivative** $f'(x)$ analytically.
 - You **can evaluate** $f(x)$ at any point.
-

Idea

Instead of the **tangent line** (Newton), use a **secant line** — a line passing through two previous points on the curve.

Approximate the derivative:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Plug into Newton's formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{f(x_n)}{\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}}$$

Simplifies to:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Example: Approximating $2\sqrt{2}$

Let:

- $f(x) = x^2 - 2$
- The root is $2 \approx 1.41421\sqrt{2} \approx 1.41421$

Start with:

- $x_0 = 1$
- $x_1 = 2$

Iterations:

1. x_2 :

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 2 - \frac{2^2 - 2}{2 \cdot 2} = 2 - \frac{2}{4} = 1.5$$

2. x_3 :

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} = 1.5 - \frac{(1.5)^2 - 2}{2 \cdot 1.5} = 1.5 - \frac{0.25}{3} = 1.41667$$

3. x_4 :

$$x_4 = (7/5) \cdot (4/3) + 27/5 + 4/3 = 58/41 \approx 1.41463$$

$$x_4 = \frac{(7/5) \cdot (4/3) + 2}{7/5 + 4/3} = \frac{58}{41} \approx 1.41463$$

✅ Very close to $2 \approx 1.41421 \sqrt{2} \approx 1.41421$ in just 3 steps!

⚖️ Comparison of Methods

| Method | Requires Derivative? | Initial Guesses | Convergence Speed | Always Converges? |
|-----------|----------------------|-------------------|-----------------------|-------------------|
| Bisection | ❌ No | 2 (bracketing) | Slow (linear) | ✅ Yes |
| Newton's | ✅ Yes | 1 | Very Fast (quadratic) | ❌ No |
| Secant | ❌ No | 2 (not bracketed) | Fast (super-linear) | ❌ No |

📄 Summary

- Secant Method is a **derivative-free alternative** to Newton's method.
- Uses two points to **approximate the slope** (secant line).
- Formula:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

- Converges **faster than bisection**, but **not as fast as Newton**.
- Needs **2 initial guesses**, but can be very effective in practice.

Would you like a side-by-side code implementation for Newton's vs Secant in Python or MATLAB?

Here's a **summary of the concept of Order of Convergence**, as explained by Jeff Chasnov:

📐 Order of Convergence – How Fast Do Root-Finding Methods Converge?

🎯 Goal

To **quantify how fast** a root-finding method converges to the actual root r of a function $f(x)$ where $f(r) = 0$.

📈 Error Definition

Let:

- x_n be the n th approximation
- r be the actual root

Then the **error** at step n is:

$$\epsilon_n = |x_n - r|$$

Order of Convergence (p)

We say a method has **order of convergence** p if:

$$\epsilon_{n+1} \approx C \cdot \epsilon_n^p \quad \epsilon_{n+1} \approx C \cdot \epsilon_n^p$$

Where:

- C is a constant,
- $\epsilon_n \rightarrow 0$ as $n \rightarrow \infty$

 **Interpretation:**

- If $p = 1$: Linear convergence (slow)
- If $p = 2$: Quadratic convergence (fast)
- If $p > 1$: Superlinear convergence (very good)

Orders of Common Methods

| Method | Convergence Order p | Description |
|-----------|-----------------------|----------------------------------|
| Bisection | $p = 1$ | Linear (slowest) |
| Secant | $p \approx 1.618$ | Superlinear (faster than linear) |
| Newton's | $p = 2$ | Quadratic (fastest) |

Example: Bisection Method

Each iteration **halves** the interval, so:

$$\epsilon_{n+1} = \frac{1}{2} \epsilon_n \Rightarrow p = 1$$

Key Takeaway

- Higher $p \rightarrow$ faster convergence.
- Bisection always converges, but slowly.

- **Newton's method** is fastest, but needs derivative and a good starting guess.
- **Secant** is a great compromise: faster than bisection, doesn't need derivative.

Would you like to see this visualized in a graph or demonstrated with a small code simulation comparing convergence rates?

Here's a **step-by-step summary** of how the **order of convergence of Newton's Method** is derived, as explained by Jeff Chasnov:

Newton's Method: Order of Convergence

We aim to prove that Newton's method has **quadratic convergence** — i.e., **order $p=2$** — under suitable conditions.

Step 1: Newton's Iteration

The iteration formula is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Let r be the **root**, such that $f(r) = 0$, and define the **error** at step n :

$$\epsilon_n = x_n - r$$

So we want to express ϵ_{n+1} in terms of ϵ_n .

Step 2: Error Form of the Iteration

Start from the Newton update:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Then:

$$\epsilon_{n+1} = x_{n+1} - r = x_n - \frac{f(x_n)}{f'(x_n)} - r = \epsilon_n - \frac{f(x_n)}{f'(x_n)}$$

Now express $f(x_n)$ and $f'(x_n)$ as **Taylor expansions around r** , because $\epsilon_n = x_n - r$ is small.

Step 3: Taylor Expansions

Since $f(r) = 0$, expand $f(x_n) = f(r + \epsilon_n)$:

Numerator:

$$f(x_n) = f(r + \epsilon_n) = f'(r)\epsilon_n + \frac{1}{2}f''(r)\epsilon_n^2 + \dots$$

Denominator:

$$f'(x_n) = f'(r + \epsilon_n) = f'(r) + f''(r)\epsilon_n + \dots f'(x_n) = f'(r + \epsilon_n) = f'(r) + f''(r)\epsilon_n + \dots$$

Step 4: Plug into Error Formula

$$\epsilon_{n+1} = \epsilon_n - f'(r)\epsilon_n + \frac{1}{2}f''(r)\epsilon_n^2 + \dots \epsilon_{n+1} = \epsilon_n - \frac{f''(r)}{f'(r)}\epsilon_n^2 + \dots$$

Divide numerator and denominator by $f'(r)$ to simplify:

$$\epsilon_{n+1} = \epsilon_n - \frac{f''(r)}{f'(r)}\epsilon_n^2 + \dots \epsilon_{n+1} = \epsilon_n - \frac{f''(r)}{f'(r)}\epsilon_n^2 + \dots$$

Now use the **Taylor expansion of $1 + \delta \approx 1 - \delta + \delta^2 - \dots$**

So:

$$\epsilon_{n+1} \approx \epsilon_n - \frac{f''(r)}{f'(r)}\epsilon_n^2 + \dots \epsilon_{n+1} \approx \epsilon_n - \frac{f''(r)}{f'(r)}\epsilon_n^2 + \dots$$

Step 5: Multiply and Simplify

After careful expansion and dropping higher-order terms:

$$\epsilon_{n+1} \approx \frac{1}{2} \frac{f''(r)}{f'(r)} \epsilon_n^2$$

Conclusion: Quadratic Convergence

$$|\epsilon_{n+1}| \approx K |\epsilon_n|^2 \quad K = \left| \frac{1}{2} \frac{f''(r)}{f'(r)} \right|$$

Where $K = \left| \frac{1}{2} \frac{f''(r)}{f'(r)} \right|$

Thus, **Newton's method has order of convergence $p=2$** when:

- The root r is **simple** (i.e., $f'(r) \neq 0$)
- The function is sufficiently smooth (differentiable)

Important Notes

- If $f'(r) = 0$, convergence is not quadratic — it may be slower or fail.
- Newton's method is very **sensitive** to the initial guess.

Would you like a numerical example or Python code to **visualize the convergence rate**?

Here's a concise summary of the second video explaining **how to generate a fractal using Newton's Method for complex cube roots of unity**:

Mathematical Setup

- The equation to solve is:

$$f(z) = z^3 - 1 = 0$$

- The solutions (cube roots of unity) are:
 - $r_1 = 1$
 - $r_2 = -\frac{1}{2} + i\frac{\sqrt{3}}{2}$
 - $r_3 = -\frac{1}{2} - i\frac{\sqrt{3}}{2}$

These lie on the **unit circle** in the complex plane, evenly spaced at 120° .

Newton's Method

To find roots of $f(z)$, use the iterative formula:

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)} = z_n - \frac{z_n^3 - 1}{3z_n^2} = z_n - \frac{z_n^3 - 1}{3z_n^2}$$

Generating the Fractal

1. **Grid the complex plane** — Treat each complex number z_0 as a pixel.
 2. For each pixel:
 - Apply Newton's method starting from z_0 .
 - See **which root** it converges to (i.e., whether final value is close to r_1 , r_2 , or r_3).
 - **Color the pixel**:
 - Red if it converges to r_1
 - Green for r_2
 - Blue for r_3
 3. **Borderline pixels**, where it's not clear which root they'll converge to, result in extremely intricate patterns — **this is where the fractal appears**.
-

Why It's a Fractal

- In regions near the **boundaries between basins of attraction**, Newton's method becomes sensitive to small changes in the initial value.
 - This sensitivity produces **self-similar structures** as you zoom in — a defining property of fractals.
-

What's Next

In the next video (mentioned by Jeff Chasnoff), he'll show how to implement this fractal visualization using **MATLAB**.

If you'd like, I can help you implement this in **Python (e.g., with NumPy and Matplotlib)** or explain the MATLAB version.

Sure! Here's the English translation of the video transcript that explains how to code the Newton Fractal step-by-step:

[MUSIC]

In this video, I'll show you how to code the Newton fractal.

We want to apply Newton's method to find the cube roots of unity.

So, we need to find the zeros of the function:

$$f(z) = z^3 - 1.$$

We scan over the complex plane and iterate each grid point using Newton's method:

$$z_{n+1} = z_n - f(z_n)/f'(z_n)$$

Here, $f'(z)$ is just the derivative of $f(z)$, which is $3z^2$.

At **0:42**:

Since Newton's method will be the core of our code, we'll start with it.

0:52:

Here is Newton's iteration. **Z** will be a matrix. The initial values of Z are points on the complex plane where we want to compute the fractal.

At the end of the iteration, each element of Z should have converged to one of the three cube roots of unity.

1:15:

We will iterate Newton's method 40 times — it's fast and that should be enough for all starting points to converge.

1:37:

Now let's begin coding from the start. We want to set all initial values in Z and define the necessary functions.

1:53:

Our function **f** is: $f(Z) = Z^3 - 1$. Since **Z** is a matrix, we need to use the dot operator to cube element-wise.

Its derivative, **f'**, is defined as $3 * Z^2$. So that defines the two functions we'll use in Newton's method.

2:31:

We might as well define the cube roots at the beginning of the code.

2:36:

The cube roots of unity are:

- **root1** = 1,
- **root2** = $-1/2 + i\sqrt{3}/2$,
- **root3** = $-1/2 - i\sqrt{3}/2$.

When I write the imaginary unit **i**, I prefer to use **1i** because I sometimes use **i** as a variable or loop index in code.

So using **1i** avoids bugs.

3:17:

Next, I want to set up the grid.

3:23:

We start with 1000 points in the **x-direction** (real axis), and 1000 in the **y-direction** (imaginary axis).

3:35:

We're visualizing a fractal from -2 to 2 on both the real and imaginary axes — this gives a nice picture.

3:58:

Use `linspace` to divide `xmin` to `xmax` into `nx` points (and similarly for `y`).

4:15:

Create the grid using `meshgrid(X, Y)` — it returns two matrices:

- **X** holds the x-values of the grid
 - **Y** holds the y-values of the grid
-

4:47:

What we need is the complex grid **Z = X + iY**. This gives the initial values for Newton's method.

5:29:

We now run Newton's method — it's very fast, as it processes all grid values simultaneously.

5:43:

Next, we determine which root each grid point converged to.

5:49:

We define a small threshold, **epsilon = 0.001**, and then define four logical matrices: **Z1, Z2, Z3, Z4**.

- **Z1** will be true (1) where $\text{abs}(Z - \text{root1}) < \text{epsilon}$
- So if Z converged to root1, this condition is true, else false (0)

6:24:

So **Z1** is a matrix of ones where the starting points converged to **root1**, and zeros elsewhere.

7:19:

Similarly:

- **Z2** = 1 where values converged to **root2**
- **Z3** = 1 where values converged to **root3**

7:57:

Z4 = not (Z1 + Z2 + Z3) — so it indicates **non-convergence**.

8:00+:

Now, we are ready to plot the result.

We open a figure and define a color map:

- First row: how to color value **1**
- Second: how to color **2**, and so on...
- $[1, 0, 0] = \text{red}$
- $[0, 1, 0] = \text{green}$
- $[0, 0, 1] = \text{blue}$
- $[0, 0, 0] = \text{black}$ (for non-convergence)

Then set the colormap of the figure.

We now create the final image data:

Z = Z1 + 2Z2 + 3Z3 + 4*Z4

This results in a matrix of 1s (red), 2s (green), 3s (blue), and 4s (black).

Finally, plot the image using the **image** command, setting axes and labels.

Note:

- Images are by default shown top-down (like in image processing)
- Numerical plots usually have Y increasing upward

- So, to match expectations, we use `set(gca,'YDir','normal')`
-

12:56:

When writing your code, use MATLAB's **debugging tools**.

Set a **smaller grid** (like 10x10) for easier debugging.

Place breakpoints after assigning Z, after Newton's method, and before image creation.

16:16:

Now Z should contain only 1, 2, 3, or 4 (depending on which root it converged to or if it didn't).

17:03:

MATLAB also has a **profiler** to measure how long each line of code takes.

If we use a larger grid (e.g., 8000x8000), we can see that most of the runtime is spent in Newton's iteration — as expected.

18:13:

For example, 33 out of 35 seconds were spent on the line:

`Z = Z - f(Z)./fp(Z);`

This confirms that Newton's method is the "engine" of the fractal code.

Let me know if you'd like the full MATLAB code version in English or explained line-by-line!

Here's the **full English transcript** of the video, cleaned up and organized for clarity:

Introduction

0:03

In this video, let's learn about the **root-finding functions** provided by **MATLAB**.

Most of the time, you don't need to write your own root-finding algorithm, like we did when drawing the Newton fractal.

Instead, you can use MATLAB's **built-in functions**.

There are two main root-finding functions in MATLAB:

1. `roots` — finds **all real and complex roots** of a **polynomial**.
 2. `fzero` — finds **one real root** of a **non-linear function**.
-

Using the roots Function

To use `roots`, you just give it the **coefficients of the polynomial**.

Suppose we have a polynomial of the form:

$$p_1x^n + p_2x^{n-1} + \dots + p_{n+1} = 0$$

This polynomial has $n+1$ coefficients.

You must write the coefficients in **descending order of powers**, and include **zeros** for any missing powers.

Syntax:

```
r = roots(p)
```

Here, p is a vector of coefficients, and r will be a column vector of **complex roots**.

Example

Let's solve the polynomial:

$$x^3 - 3x^2 + 4x - 2 = 0$$

We define the coefficients in MATLAB:

```
p = [1 -3 4 -2];
```

```
r = roots(p);
```

Run this in the Live Editor.

The output is:

```
r =
```

```
1.0000 + 0.0000i
```

```
0.0000 + 1.0000i
```

```
0.0000 - 1.0000i
```

So the roots are **1**, **i**, and **-i**.

Note: MATLAB may show very small imaginary parts due to round-off errors.

We can verify that **1 is a root**:

$$1^3 = 1$$

$$-3 \times 1 = -3$$

$$+4 \times 1 = 4$$

$$-2 = -2$$

$$\text{So: } 1 - 3 + 4 - 2 = 0 \quad \checkmark$$

Using the fzero Function

fzero finds a **single real root** of a non-linear function:

$$f(x) = 0$$

Function Definition

You need to define f(x) as either:

- An **anonymous function**: $f = @(x) \dots$
- Or a **separate subfunction**

If the function depends on a parameter (like a), you can define:

```
f = @(x,a) x - exp(-a*x);
```

To solve $x = e^{(-a \cdot x)}$, rewrite it as:

$f(x) = x - \exp(-a \cdot x) = 0$

Let's say $a = 0.5$, then:

```
a = 0.5;
```

```
f = @(x) x - exp(-a*x);
```

You need to give fzero:

- A function of one variable
- An **initial guess** or a **bracketing interval**

If your function has parameters, you can wrap it like this:

```
r = fzero(@(x) f(x, a), x0);
```

Example

```
a = 0.5;
```

```
f = @(x, a) x - exp(-a*x);
```

```
x0 = 0; % initial guess
```

```
r = fzero(@(x) f(x, a), x0);
```

Running this, we get:

```
r = 0.7035
```

This is the solution to $x = e^{(-0.5x)}$.

Summary

8:41

We introduced two root-finding functions in MATLAB:

- **roots.m** — finds all roots of a **polynomial**
- **fzero.m** — finds one root of a **non-linear function**

"I've used fzero many times in my research. It's powerful and simple to use. You'll get a chance to try it out in course exercises."

— Jeff [last name not heard clearly]

Let me know if you'd like the code from these examples or want a practice problem based on this!

Great! Here's a concise **summary of the first lecture** on the theoretical background for computing the **Feigenbaum delta**:

Topic: Computing the Feigenbaum Delta

Context: Based on the **logistic map**:

$$x_{i+1} = \mu x_i (1 - x_i)$$

This map exhibits **bifurcations** leading to **chaos** as the parameter μ increases.

Bifurcation Behavior

- At small μ : the map converges to a **fixed point** (period-1).
 - As μ increases:
 - **Period-2** cycle emerges at μ_0
 - **Period-4** at μ_1
 - **Period-8** at μ_2 , and so on...
 - Eventually, the system exhibits **chaotic behavior** after infinite period-doubling.
-

Feigenbaum Delta Definition

This constant captures how the bifurcation points cluster closer as the period doubles:

$$\delta = \lim_{n \rightarrow \infty} \frac{\mu_n - \mu_{n-1}}{\mu_{n+1} - \mu_n}$$

Problem with Direct Computation

- Near bifurcation points, the **transient** (settling time) is very long.
 - Requires many iterations to accurately compute bifurcation values μ_n .
-

Better Approach: Superstable Cycles

Instead of bifurcation points, we compute at **superstable cycles**, where:

- The orbit includes $x = \frac{1}{2}$
 - These values of μ are denoted μ_n
 - Superstable cycles **converge fast**, so they're more computationally efficient.
-

Examples & Definitions

- $m_0 = 2$: because $x = 0.5x = 0.5$ is a **fixed point** when $\mu = 2$
- m_1 : for period-2 superstable cycle
 - Derived from solving a **cubic**:

$$\mu^3 - 4\mu^2 + 8 = 0 \quad \mu^3 - 4\mu^2 + 8 = 0$$

- Solution:

$$m_1 = 1 + \sqrt{5}$$

Final Delta Definition (Practical Version)

$$\delta = \lim_{n \rightarrow \infty} \frac{m_{n-1} - m_{n-2}}{m_n - m_{n-1}}$$

This converges to the **same Feigenbaum delta**, but is faster to compute.

What's Next

- In future lectures:
 1. How to **compute** m_n using **Newton's method**
 2. How to implement this in **MATLAB**
-

Let me know if you'd like a visual diagram or notes formatted for printing or LaTeX!

Here's a concise summary and explanation of the **algorithm** described in the video for computing the **Feigenbaum delta (δ)**:

Goal

Compute the Feigenbaum delta:

$$\delta = \lim_{n \rightarrow \infty} \frac{m_n - m_{n-1}}{m_{n-1} - m_{n-2}}$$

Where m_n is the value of μ such that starting from $x_0 = \frac{1}{2}$, the logistic map returns to $\frac{1}{2}$ after 2^n iterations. That means $x_{2^n} = \frac{1}{2}$.

Logistic Map

$$x_{i+1} = \mu x_i (1 - x_i)$$

Step-by-Step Algorithm

1. Define Function for Root Finding

We want:

$$g(\mu) = x_N(\mu) - \frac{1}{2}$$

Where $N = 2^n$, and $x_0 = \frac{1}{2}$.

We seek roots of $g(\mu)$, i.e., values of μ such that $x_N = \frac{1}{2}$.

2. Use Newton's Method

Newton's iteration:

$$\mu_{j+1} = \mu_j - \frac{g(\mu_j)}{g'(\mu_j)} = \mu_j - \frac{x_N(\mu_j) - \frac{1}{2}}{x_N'(\mu_j)}$$

So we need both x_N and x_N' — the derivative of x_N w.r.t. μ .

3. Compute Derivative Recursively

Let:

- $x_0 = \frac{1}{2}$
- $x_0' = 0$

Then iterate:

- $x_{i+1} = \mu x_i (1 - x_i)$
- $x_{i+1}' = x_i (1 - x_i) + \mu x_i' (1 - 2x_i)$

These two coupled equations must be computed together (simultaneous iteration) up to $N = 2^n$.

4. Initial Guess for Newton's Method

We don't know m_n yet, so we **guess** it based on the **previous values** and an estimate of δ :

$$\text{Initial guess for } m_n = m_{n-1} + \frac{m_{n-1} - m_{n-2}}{\delta_{n-1}}$$

Where:

$$\delta_{n-1} = \frac{m_{n-1} - m_{n-2}}{m_{n-2} - m_{n-3}}$$

5. Bootstrapping

To start computing m_{2m_2} , you only have $m_0 = 2m_0 = 2$ and $m_1 = 1 + \sqrt{5}$. You don't yet have a δ value.

So:

- Pick an initial δ estimate (say, 5.0).
- Use it to guess m_{2m_2}
- Compute m_{2m_2} using Newton's method
- Then compute updated δ_1
- Use δ_1 to guess m_{3m_3} , and so on.

Limitations

Eventually, **round-off error** in floating point arithmetic will prevent accurate computation of further m_n 's — especially as $N = 2^n$ becomes large.

Final Summary

To compute Feigenbaum δ :

- Define a root-finding problem for $g(\mu) = x_N - \frac{1}{2}$
- Use Newton's method:

$$\mu_{j+1} = \mu_j - \frac{x_N - 0.5}{x'_N} \quad \mu_{j+1} = \mu_j - \frac{x_N - 0.5}{x'_N}$$

- Simultaneously compute both x_i and x'_i
- Use previous m 's and δ estimates to generate initial guesses
- Iterate to get new m_n 's and better approximations to δ .

If you'd like, I can help you implement this algorithm in Python step-by-step.

Thanks for sharing the transcript. This final explanation from Prof. Jeff Chasnoff summarizes the implementation strategy for computing the **Feigenbaum δ** numerically using **Newton's method** on the **logistic map**. Here's a breakdown of the **computational plan** based on the lecture:

What You're Computing

- The **Feigenbaum δ** $\delta \approx 4.66920160910299\dots$ — a universal constant characterizing the geometric rate of period-doubling bifurcations in certain dynamical systems.
- Specifically, you're computing a sequence of bifurcation points m_n , and using them to estimate:

$$\delta_n = \frac{m_{n-1} - m_{n-2}}{m_n - m_{n-1}}$$

Key Concepts

- **Logistic Map:**

$$x_{n+1} = \mu x_n (1 - x_n)$$

- **Bifurcation Point** m_n :

Value of μ at which the system undergoes a period-doubling to 2^n -period.

- **Newton's Method:**

$$\text{Solve } g(\mu) = x_{2^n}(\mu) - x_{2^{n-1}}(\mu) = 0$$

Newton step:

$$\mu_{\text{new}} = \mu_{\text{old}} - g(\mu) / g'(\mu)$$

Three Loops

1. Outer Loop (Loop 1): Iterate Over n

- For $n = 2$ to max_n (e.g., 14–15)
- Compute m_n
- Use prior m_n values and δ to guess next m

2. Middle Loop (Loop 2): Newton's Method Iteration

- Until convergence (e.g., relative change $< 10^{-12}$)
- Each iteration:
 - Call logistic map function to compute x_{2^n}
 - Also compute $dg/d\mu$ (requires chain rule through logistic iterations)

3. Inner Loop (Loop 3): Logistic Map Iteration

- Run the map 2^n times for given μ
- Compute both x and derivative x' via:

$$x_{i+1} = \mu x_i (1 - x_i) \quad x'_{i+1} = x_i (1 - x_i) + \mu (1 - 2x_i) x'_i$$

Code Structure Sketch (Pseudocode/Algorithm)

```
% Initialization
```

```
m = zeros(1, max_n); % store bifurcation points
```

```
delta = zeros(1, max_n); % store Feigenbaum delta estimates
```

```

m(1) = known_m1;
m(2) = known_m2;

for n = 2:max_n-1
    % Initial guess for m_n based on previous deltas
    if n == 2
        mu = m(n) + 1e-4; % crude guess
    else
        mu = m(n) + (m(n) - m(n-1)) / delta(n-1);
    end

    % Newton's method to refine mu
    for newton_iter = 1:max_iter
        [g, dg] = compute_g_and_derivative(mu, n); % g = f^{2^n}(x) - x
        mu_new = mu - g / dg;

        if abs(mu_new - mu) < 1e-12
            break;
        end
        mu = mu_new;
    end

    m(n+1) = mu;

    % Compute delta
    delta(n) = (m(n) - m(n-1)) / (m(n+1) - m(n));
end

```

What You Should Implement

- A `logistic_map(mu, n)` function that returns x_{2^n}
- A derivative computation: $\frac{dx_{2^n}}{d\mu}$

- Newton's method step using both
-

Final Advice

- Save **all computed mnm_n** values and **delta** estimates.
- Plot or print the convergence of delta — it should stabilize around 4.669.
- Double precision only gives you around **15–16 digits**, so don't expect more accuracy than that.

Would you like help implementing any specific part of this in MATLAB?

Here's a concise summary of what you'll learn in **Week 3** of *Numerical Methods for Engineers*, based on the transcript:

Week 3 Overview: Numerical Linear Algebra

Topics Covered:

1. **Systems of Linear Equations**
 - **Gaussian Elimination with Partial Pivoting**
 - **LU Decomposition**
 - Counting the number of floating-point operations (flops)
2. **Eigenvalues and Eigenvectors**
 - **Power Method** for dominant eigenvalue/eigenvector computation
3. **MATLAB Tools**
 - `\` (backslash operator): Solves $Ax=b$
 - `lu(A)`: Computes LU decomposition
 - `eig(A)`: Computes eigenvalues and eigenvectors
4. **Systems of Nonlinear Equations**
 - Use of **Newton's Method** with **matrix algebra**

Project 3:

- Apply **Newton's Method** to compute **fractals** from solutions of the **Lorenz equations**
 - Lorenz system is a classic example of a nonlinear system that exhibits **chaotic behavior**
-

Would you like help getting started with any specific part (e.g. implementing LU decomposition in MATLAB, or applying the power method)?


Here's a **summary and key takeaways** from this lecture on **Gaussian Elimination and Round-Off Error**:

Lecture Summary: Gaussian Elimination Without Pivoting

What is Gaussian Elimination?

- A method for solving systems of linear equations.
 - Involves transforming the system's matrix into an upper triangular form using row operations.
 - Solving then proceeds via **back substitution**.
-

The Issue with Large Systems

 **Gaussian elimination works well on small systems (e.g., 3×3), but breaks down for large systems (e.g., 1000×1000) due to round-off error.**

Understanding Round-Off Error

- **Machine epsilon (ϵ)** is the smallest number that distinguishes two floating-point numbers.
 - In double precision:

$$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$$

- Example:
 - $2 + \epsilon = 2$
 - $4 - \epsilon = 4$due to limits in floating-point representation.
-

Contrived 2×2 Example

System of Equations:

$$\begin{cases} x_1 + 2x_2 = 4 \\ x_1 - x_2 = 1 \end{cases}$$

Expected (approximate) solution:

- $x_2 \approx 2$
- $x_1 \approx 3$

Gaussian elimination result (due to round-off):

- $x_2 = 2$ (correct)
- $x_1 = 0$ (totally wrong)

✗ What went wrong?

- Elimination step produced an expression like:

$$\epsilon x_1 = 0 \quad \text{where } \epsilon \approx 0$$

- Should have been:

$$\epsilon x_1 = \text{very small number} \Rightarrow x_1 \approx \frac{\text{very small number}}{\epsilon} \rightarrow x_1 \approx 3$$

- But due to round-off, it became 0, giving a **catastrophically incorrect result**.
-

💡 Key Insight:

- If the **pivot element** (diagonal entry) is **very small but non-zero**, and we don't **swap rows**, we **lose significant digits**.
 - In large matrices, **small round-off errors** can **accumulate** → completely wrong answers.
-

🔧 Solution: Partial Pivoting

- **Swap rows** to bring the **largest available element** into the pivot position.
 - This avoids dividing by small numbers and **reduces round-off errors**.
-

🔄 Coming Up:

- The next video will explain **partial pivoting** in detail.
-

Let me know if you'd like help implementing **Gaussian elimination with pivoting** in MATLAB or Python.

Here's a **summary and key points** from the lecture on **Gaussian Elimination with Partial Pivoting**:

📌 Lecture Summary: Gaussian Elimination with Partial Pivoting

🧠 Recap:

- In the last video, we saw that applying Gaussian elimination **without pivoting** can lead to **catastrophic errors** due to **round-off error**—especially in large systems.
-

🔧 What is Partial Pivoting?

- **Partial Pivoting = Row Swapping Only**

- At each step of elimination:
 - **Look down the current column.**
 - **Identify the entry with the largest magnitude.**
 - **Swap that row to the top (pivot position).**
- **Full Pivoting** = Row and column interchanges
 - Rarely used in practice; partial pivoting is sufficient for stability in most cases.

Example: Solving the System with Pivoting

Given system:

$$\begin{cases} \epsilon x_1 + 2x_2 = 4 \\ x_1 - x_2 = 1 \end{cases}$$

Augmented matrix:

$$\begin{bmatrix} \epsilon & 2 & | & 4 \\ 1 & -1 & | & 1 \end{bmatrix}$$

Step 1: Pivoting

- Compare first column: ϵ vs. 1
- Swap rows: put **row with 1** at the top

$$\begin{bmatrix} 1 & -1 & | & 1 \\ \epsilon & 2 & | & 4 \end{bmatrix}$$

Step 2: Elimination

- Eliminate ϵ below the pivot (first column):
 - Multiply first row by $-\epsilon$, add to second row:

Second row becomes:

- $\epsilon - \epsilon = 0$
- $2 + \epsilon$
- $4 - \epsilon$

$$\Rightarrow \begin{bmatrix} 1 & -1 & | & 1 \\ 0 & 2 + \epsilon & | & 4 - \epsilon \end{bmatrix}$$

Due to round-off:

- $2 + \epsilon \approx 2$
- $4 - \epsilon \approx 4$

Back Substitution

1. **Solve second equation:**

$$2x_2 = 4 \Rightarrow x_2 = 2$$

2. **Plug into first equation:**

$$x_1 - x_2 = 1 \Rightarrow x_1 = 3$$

✅ **Correct result achieved**, despite round-off approximations.

Key Insights

- **Partial pivoting avoids dividing by very small numbers**, which reduces round-off errors.
 - Without it, even simple 2×2 problems can fail numerically.
 - With it, **large systems (1000+ equations)** can be solved **accurately and stably**.
 - This was a **breakthrough** in numerical methods, making large-scale computations practical.
-

Summary Formula:

At step k of elimination:

1. Find $i \geq k$ such that:

$$|A[i,k]| = \max_{j \geq k} |A[j,k]|$$

2. Swap row i and row k

3. Proceed with Gaussian elimination as normal

Let me know if you want a code implementation of **Gaussian elimination with partial pivoting** (in Python or MATLAB), or help with LU decomposition, which builds on this method.

Here's a clear and structured summary of the lecture on **LU Decomposition with Partial Pivoting**, as explained by Prof. Jeff Chasnov:

Lecture Summary: LU Decomposition with Partial Pivoting

What is LU Decomposition?

- LU Decomposition factors a matrix **A** into:

$$A = LU$$

where:

- **L** = Lower triangular matrix

- **U** = Upper triangular matrix
 - **Purpose:** Makes solving $Ax=b$ faster, especially when:
 - The same matrix **A** is used with multiple different right-hand sides **b**
 - Common in solving **PDEs** numerically
-

Why Use Partial Pivoting?

- To avoid division by small numbers (which causes round-off errors)
 - We perform **row interchanges** during Gaussian elimination
-

Step-by-Step: LU with Partial Pivoting

Step 1: Apply Partial Pivoting

- Start with a matrix **A**, scan the first column.
 - Find the entry with **largest absolute value** (e.g., 6)
 - Swap rows if necessary — record this as a **permutation matrix** P_{12}
-

Step 2: Gaussian Elimination

- Eliminate below the pivot using **elementary row operations**, represented as matrices:

M_1, M_2, \dots, M_{n-1}

- For example:
 - Use row 1 to eliminate entries in rows 2 and 3
 - New entries calculated by subtracting a multiple of row 1 from other rows
-

Step 3: Second Pivoting Step

- In the second column, apply **partial pivoting again**:
 - Find largest magnitude entry in remaining submatrix
 - Swap rows (e.g., P_{23}) to avoid zero pivot
-

Step 4: Record Final Matrix

- After full Gaussian elimination, the resulting matrix is:

$U = \text{Upper Triangular Matrix}$

✅ Step 5: Constructing the L Matrix

- Combine the inverse of elementary matrices and account for row swaps
- Trick: since PP matrices are **their own inverses** ($P^{-1} = P^{T} = P$):

$$A = P^{-1} L U A = P^{-1} L U$$

- Rearranged:

$$P A = L U \quad \text{(standard form)}$$

- Example LL matrix (with multipliers used during elimination):

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{3} & 0 & 1 \end{bmatrix}$$

- Example UU matrix from elimination:

$$U = \begin{bmatrix} 6 & -6 & 7 \\ 0 & -5 & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

⚙️ In MATLAB

- MATLAB uses a slightly different convention:
 - It writes:

$$A = P L U A = P L U$$

where PP is a **permutation matrix**

- The **L matrix is not exactly lower triangular** — due to permutations
 - MATLAB calls it "**psychologically lower triangular**"
 - Meaning: it behaves like a lower triangular matrix, even if rows are permuted

📄 Summary Formula

$$P A = L U P A = L U$$

- P**: product of all permutation matrices (row swaps)
- L**: lower triangular matrix (from elimination multipliers)
- U**: final upper triangular matrix

🚀 Why It Matters

- LU decomposition is **much faster** than Gaussian elimination if you're solving the same system with different right-hand sides.
- LU with pivoting ensures **numerical stability**.

- For large systems, **let MATLAB do the work!**
-

Would you like to see an actual example of LU decomposition with pivoting in Python or MATLAB? I can walk you through the code and outputs.

Here's a clear and concise **summary** of the video on **operation count and algorithm scaling** in matrix algebra by Prof. Jeff Chasnov:

Lecture Summary: Why LU is Faster – Operation Count and Scaling

Main Claim Recap

- LU decomposition helps **speed up** solving systems like $Ax=b$, especially when:
 - A is fixed
 - b changes frequently (e.g. in time-stepping PDEs)
 - Why? Because once $A=LU$ is computed, solving becomes cheaper via **forward and backward substitution** rather than full Gaussian elimination each time.
-

How Do We Justify LU's Efficiency?

We use **operation counting**:

- Count **multiplications and additions** an algorithm performs
 - Called the **operation count** or **computational complexity**
-

Big O Notation:

- Denotes how the number of operations **scales with problem size**
 - We care about the **leading-order term** only (e.g. n^3 , not constants or lower powers)
-

Example: Matrix Multiplication

Let's say A and B are both $n \times n$ matrices.

Operation Count:

- **For one element** of result matrix:
 - n multiplications
 - $n-1$ additions
- **Total number of elements** in output matrix = n^2

✅ Total Operations:

Multiplications = $n^2 \cdot n = n^3$ Additions $\approx n^3$ (since $n^2(n-1) \approx n^3$)
 $\text{Multiplications} = n^2 \cdot n = n^3$
 $\text{Additions} \approx n^3$ (since $n^2(n-1) \approx n^3$)

So, matrix multiplication scales as:

$$O(n^3) \quad \boxed{O(n^3)}$$

💻 Real-World Application (MATLAB)

- Multiply two random 4096×4096 matrices:
 - Takes about **1.1 seconds**
- Now multiply two 8192×8192 matrices:
 - Size is doubled $\rightarrow 2n$
 - Expected time:

$$T_{2n} \approx 8 \cdot T_n = 8.8 \text{ seconds} \quad T_{2n} \approx 8 \cdot T_n = 8.8 \text{ seconds}$$

- Actual result: **7.9 seconds**
 - Close enough \rightarrow scaling holds
-

🎯 Why This Matters

- Helps estimate compute time in **high-performance computing**
 - If a simulation takes 30 mins, you want to know:
 - Can I double resolution and finish it **overnight**?
 - If complexity is $O(n^3)$, doubling n makes time $\approx 8 \times$
-

➡ What's Next?

- Next videos:
 1. Operation count for **Gaussian elimination**
 2. Operation count for **forward/backward substitution**
 3. Final justification: **LU + substitution is faster** than repeated elimination
-

✅ Review Summary

| Task | Scaling | Operations |
|-----------------------|-------------|-------------|
| Matrix Multiplication | $O(n^3)$ | $\sim 3n^3$ |
| Gaussian Elimination | Coming soon | TBD |
| LU + Substitution | Coming soon | TBD |

Would you like a visual chart or Python demo showing how run-time changes with n in matrix multiplication?

Here's a **clear and structured summary** of the video on **Operation Count for Gaussian Elimination** by Prof. Jeff Chasnov:

Lecture Summary: Counting Operations in Gaussian Elimination

Goal

To compute the **number of operations (additions + multiplications)** required to perform **Gaussian elimination**, and understand how it **scales with matrix size n** .

Key Identities Used

To simplify the count, we use standard summation formulas:

- $\sum_{k=1}^n 1 = n$
- $\sum_{k=1}^n k = \frac{n(n+1)}{2}$
- $\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$

Step-by-Step Operation Count

Setup

- Consider an $n \times n$ matrix A
- We perform elimination **pivot-by-pivot** from row 1 to $n-1$

At Pivot Step i :

- Pivot is at position (i, i)
- To eliminate below this pivot:
 - Operate on $n - i$ rows below
 - Each row has $n - i$ elements to the right

- Each step needs:

$(n-i)^2$ multiplications and additions $(n-i)^2$ \text{ multiplications and additions}

Total Operations:

$$\sum_{i=1}^{n-1} (n-i)^2 = \sum_{k=1}^{n-1} k^2 = \frac{(n-1)n(2n-1)}{6}$$

This expression grows like $\frac{1}{3}n^3$ for large n

Scaling Result

Gaussian Elimination is $O(n^3)$

- Doubling matrix size $\rightarrow 8\times$ computation time
- This is **why LU decomposition** is expensive upfront

Conclusion

| Method | Operation Count | Scaling |
|----------------------|-----------------------|----------|
| Gaussian Elimination | $\sim \frac{1}{3}n^3$ | $O(n^3)$ |

- This confirms Gaussian elimination is **costly**, especially for large matrices
- Next: Show that **forward and backward substitution are only $O(n^2)$** , making **LU + substitution much faster** when solving with multiple right-hand sides

Would you like a visual animation or table comparing Gaussian Elimination vs LU + Substitution over different values of n ?

Here's a **concise summary** of this video on **operation count for forward and backward substitution**, which follows up the earlier discussion on Gaussian elimination:

Lecture Summary: Why LU Decomposition is Efficient for Multiple Right-Hand Sides

Objective

To **justify** why solving systems using **LU decomposition** (i.e., solving $A\vec{x} = \vec{b}$ via $LU\vec{x} = \vec{b}$) is **far more efficient** than performing **Gaussian elimination** each time for multiple right-hand sides \vec{b} .

Operation Count for Back Substitution

Scenario:

- You already have $U\vec{x} = \vec{y}$, where U is **upper triangular**
- Solving this via **back substitution**

At row ii:

- You compute:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^n a_{ij} x_j \right)$$

- Requires:
 - $n - i$ multiplications (for the sum)
 - 1 subtraction
 - 1 division (can count as a multiplication)
- Total: $(n - i + 1)(n - i + 1)$ operations per row

Total operations:

$$\sum_{i=1}^n (n - i + 1) = \sum_{k=1}^n k = \frac{n(n+1)}{2}$$

◆ **Back substitution** = $\frac{1}{2}n^2 + \frac{1}{2}n \sim O(n^2)$

🔄 Forward Substitution (for solving $L\vec{y} = \vec{b}$)

- Same structure and cost as back substitution
- So total operations for solving $LU\vec{x} = \vec{b}$:

$$O(n^2) \text{ (forward)} + O(n^2) \text{ (backward)} = O(n^2)$$

🧠 Comparison with Gaussian Elimination

| Step | Operation Count | Scaling |
|-----------------------------|------------------|----------|
| LU Decomposition | $\frac{2}{3}n^3$ | $O(n^3)$ |
| Forward + Back Substitution | n^2 | $O(n^2)$ |

⚖️ Why This Matters

- If solving **many systems** $A\vec{x} = \vec{b}_1, \vec{b}_2, \dots$, reuse LU !
- For example:
 - $n = 1000$: LU is **300×** slower than one substitution
 - $n = 10,000$: LU is **3000×** slower

✓ So **decompose once**, solve fast many times.

🔙 Conclusion

- **Gaussian elimination** = costly: $O(n^3)$
 - **LU + forward/backward substitution** = efficient: $O(n^2)$
 - For **multiple right-hand sides**, **LU decomposition is dramatically faster**
-

Would you like a comparison **chart** or **code example** in Python or MATLAB to see this efficiency in practice?

Here's a **summary of the Eigenvalue Power Method** explained in the lecture by Jeff Chasnov:

🔍 Topic: The Power Method for Finding the Dominant Eigenvalue

🎯 Goal

To compute the **largest eigenvalue** (in absolute value) and its associated **eigenvector** of a real matrix A , using the **Power Method**.

📌 Assumptions

- $A \in \mathbb{R}^{n \times n}$
 - All eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ are **real and distinct**
 - $|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$
 - Corresponding eigenvectors e_1, e_2, \dots, e_n are **linearly independent**
-

🔑 Key Idea

Start with an arbitrary initial vector x_0 (not orthogonal to e_1), and repeatedly multiply by A :

$$x_p = A^p x_0$$

Since any vector can be written as:

$$x_0 = c_1 e_1 + c_2 e_2 + \dots + c_n e_n$$

Then:

$$x_p = A^p x_0 = \sum_{i=1}^n c_i \lambda_i^p e_i$$

As $p \rightarrow \infty$, terms like $(\lambda_i / \lambda_1)^p$ for $i \geq 2$ go to 0, so:

$$x_p \approx c_1 \lambda_1^p e_1 \approx c_1 \lambda_1^p e_1$$

Thus, x_p tends toward the direction of e_1 , and its **magnitude** grows/shrinks by λ_1^p .

Estimating λ_1


You can estimate λ_1 using the **Rayleigh quotient**:

$$\lambda_1 \approx \frac{x_p^T A x_p}{x_p^T x_p} \approx \frac{x_p^T A x_p}{\|x_p\|^2}$$

This approximates the dominant eigenvalue once x_p is close to e_1 .

Numerical Stability: Overflow / Underflow

- If $|\lambda_1| > 1$, x_p becomes **too large** (overflow)
- If $|\lambda_1| < 1$, x_p becomes **too small** (underflow)

 **Fix:** Normalize at each iteration:

$$x_{k+1} = \frac{A x_k}{\|A x_k\|}$$

This keeps the vector at a manageable size.

Algorithm Summary: Power Method

```
x = random_vector()
```

```
for k in range(max_iters):
```

```
    x_new = A @ x
```

```
    x_new = x_new / norm(x_new) # Normalize to avoid overflow
```

```
    lambda_approx = (x.T @ A @ x) / (x.T @ x)
```

```
    x = x_new
```

Result

- x converges to the **eigenvector e_1** of the **dominant eigenvalue**
- λ_1 can be approximated by:

$$\lambda_1 \approx \frac{x^T A x}{x^T x}$$



Conclusion

- Use MATLAB's eig() for general problems.
- But the **Power Method** teaches:
 - Insight into matrix behavior
 - A fast way to find the **largest eigenvalue/eigenvector** for large, sparse, or structured matrices.

Would you like a simple worked-out numerical example in Python or MATLAB next?

Here is a clear and concise **summary of the example** Professor Jeff Chasnov worked through using the **Power Method** to find the **dominant eigenvalue and eigenvector**:



Example: Power Method with a 2×2 Matrix



Matrix A

$$A = \begin{bmatrix} 6 & 5 \\ 4 & 5 \end{bmatrix}$$

This matrix has:

- **Eigenvalues:** $\lambda_1 = 10$ (dominant), $\lambda_2 = 1$



Step-by-step Iteration

Initial vector:

$$x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Multiply repeatedly by A:

1.

$$x_1 = Ax_0 = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$$

2.

$$x_2 = Ax_1 = A \begin{bmatrix} 6 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \times 6 + 5 \times 4 \\ 4 \times 6 + 5 \times 4 \end{bmatrix} = \begin{bmatrix} 56 \\ 44 \end{bmatrix}$$

3.

$$x_3 = Ax_2 = A \begin{bmatrix} 56 \\ 44 \end{bmatrix} = \begin{bmatrix} 6 \times 56 + 5 \times 44 \\ 4 \times 56 + 5 \times 44 \end{bmatrix} = \begin{bmatrix} 556 \\ 444 \end{bmatrix}$$

4.

$$x_4 = Ax_3 = A[556444] = [55564444]x_4 = Ax_3 = A \begin{bmatrix} 556 \\ 444 \end{bmatrix} = \begin{bmatrix} 5556 \\ 4444 \end{bmatrix}$$

5.

$$x_5 = Ax_4 = A[55564444] = [5555644444]x_5 = Ax_4 = A \begin{bmatrix} 5556 \\ 4444 \end{bmatrix} = \begin{bmatrix} 55556 \\ 44444 \end{bmatrix}$$

Pattern: The digits clearly show the dominant eigenvalue effect.

Estimating the Dominant Eigenvalue (λ_1)

Using the Rayleigh Quotient:

$$\lambda_1 \approx x_4^T x_5 / x_4^T x_4 = 506178271506180272 / 9.99991 \approx 506178271506180272 \approx 9.99991 \lambda_1 \approx \frac{x_4^T x_5}{x_4^T x_4} = \frac{506178271}{506180272} \approx 9.99991$$

 **Approximation is very close to 10**

Approximating the Dominant Eigenvector

From:

$$x_5 = [5555644444] \Rightarrow e_1 \approx [55556444441] = [1.250021]x_5 = \begin{bmatrix} 55556 \\ 44444 \end{bmatrix} \rightarrow e_1 \approx \begin{bmatrix} \frac{55556}{44444} \\ 1 \end{bmatrix} = \begin{bmatrix} 1.25002 \\ 1 \end{bmatrix}$$

This is close to:

$$[541] \begin{bmatrix} \frac{5}{4} \\ 1 \end{bmatrix}$$

Summary

- Start with $x_0 = [10]x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- After just **5 iterations**, the approximation of the dominant eigenvalue converged to **~10**
- The eigenvector converged to approximately $[54, 1]^T \left[\frac{5}{4}, 1 \right]^T$
- Simple method, useful for educational purposes and understanding matrix behavior

Would you like a Python or MATLAB implementation of this example?

Here's a clear **summary of the key MATLAB concepts** demonstrated in the video by Professor Jeff Chasnov, organized into four major matrix algebra topics:

Matrix Algebra in MATLAB: Key Operations

1 Solving a System of Linear Equations

Goal: Solve $Ax=b$ $x = b$, where AA is a square invertible matrix.

MATLAB Code:

```
A = [-3 2 -1; 6 -6 7; 3 -4 4];
```

```
b = [-1; -7; -6];
```

```
x = A\b; % Uses Gaussian elimination internally
```

Output:

```
x =
```

```
2
```

```
2
```

```
-1
```

✓ The `\` operator automatically chooses the best method to solve the system.

2 Solving a Least-Squares Problem

Goal: Solve $Ax=b$ $x = b$ when AA is overdetermined (more rows than columns), no exact solution.

MATLAB Code:

```
A = [1 1; 1 2; 1 3];
```

```
b = [1; 3; 2];
```

```
x = A\b; % Finds the least-squares solution
```

Output:

```
x =
```

```
1.0000
```

```
0.5000
```

✓ Equivalent to solving the **normal equations**: $ATAx=ATb$ $A^T A x = A^T b$

3 LU Decomposition

Goal: Factor $A=LU$ $A = L U$, useful for solving many systems with the same matrix AA but different right-hand sides bb

MATLAB Code:

```
[L, U] = lu(A);
```

```
x = U \ (L \ b); % Forward and backward substitution
```

Notes:

- L is a **permuted lower triangular matrix**, not strictly lower triangular due to pivoting.
 - This approach is **faster** for large matrices and repeated solves.
-

4 Eigenvalues and Eigenvectors

Goal: Solve $Ax = \lambda x$ $x = \lambda x$

MATLAB Code:

```
A = [0 1; 1 0]; % Simple matrix
```

```
lambda = eig(A); % Just eigenvalues
```

```
[V, D] = eig(A); % V: eigenvectors, D: diagonal matrix of eigenvalues
```

Output:

lambda =

-1

1

V = % Columns are eigenvectors

-0.7071 0.7071

0.7071 0.7071


D = % Diagonal eigenvalues

-1 0




0 1

Check:

$A * V(:,1)$ % Should equal $-1 * V(:,1)$

 Confirms $AV = VD$ $V = V D$, which is matrix diagonalization.

Conclusion

-  MATLAB makes **matrix computations easy and efficient**.
-  The **backslash operator** \backslash is a powerful tool for both exact and approximate solutions.
-  Functions like `lu` and `eig` provide deeper insights into matrix structure.

Would you like this in a **PDF handout** or an interactive MATLAB Live Script?

This video introduces **Newton's method for solving a system of nonlinear equations**, generalizing from the scalar (single-equation) case. Here's a concise summary of the concepts covered:

Goal:

Solve a system of two nonlinear equations:

$$f(x,y)=0, g(x,y)=0 \iff f(x,y)=0 \text{ and } g(x,y)=0$$

This generalizes to N equations in N variables.

Newton's Iteration for Systems:

We seek a sequence (x_n, y_n) that converges to a root of the system.

Derivation via Taylor Expansion:

Use **first-order Taylor expansion** of $f(x_{n+1}, y_{n+1})$ and $g(x_{n+1}, y_{n+1})$ around (x_n, y_n) :

$$\begin{aligned} f(x_{n+1}, y_{n+1}) &\approx f(x_n, y_n) + \frac{\partial f}{\partial x} \Delta x_n + \frac{\partial f}{\partial y} \Delta y_n \\ g(x_{n+1}, y_{n+1}) &\approx g(x_n, y_n) + \frac{\partial g}{\partial x} \Delta x_n + \frac{\partial g}{\partial y} \Delta y_n \end{aligned}$$

where:

$$\Delta x_n = x_{n+1} - x_n, \Delta y_n = y_{n+1} - y_n$$

We want $f(x_{n+1}, y_{n+1})=0$ and $g(x_{n+1}, y_{n+1})=0$, so we get:

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix} \begin{bmatrix} \Delta x_n \\ \Delta y_n \end{bmatrix} = - \begin{bmatrix} f(x_n, y_n) \\ g(x_n, y_n) \end{bmatrix}$$

Jacobian Matrix:

The matrix of partial derivatives is called the **Jacobian**:

$$J(x_n, y_n) = \begin{bmatrix} f_x & f_y \\ g_x & g_y \end{bmatrix}$$

Iteration Update Rule:

1. Solve the linear system:

$$J(x_n, y_n) \begin{bmatrix} \Delta x_n \\ \Delta y_n \end{bmatrix} = - \begin{bmatrix} f(x_n, y_n) \\ g(x_n, y_n) \end{bmatrix}$$

2. Update:

$$x_{n+1} = x_n + \Delta x_n, y_{n+1} = y_n + \Delta y_n$$

3. Repeat until convergence.

Convergence Tip:

- Newton's method converges **fast** (quadratically), if the initial guess is **close** to a solution.
- You must **recompute the Jacobian and the function values** at each iteration.

Application:

You'll likely be asked to:

- Implement this method.
- Compute Jacobians manually or symbolically.
- Use matrix algebra techniques like Gaussian elimination to solve the update equations.

If you'd like, I can provide a **MATLAB** or **Python** implementation of this method or help you solve a specific system using it.

Here's a clear breakdown of the **Newton's method application to the Lorenz system** from the video:

Lorenz Equations (for chaotic dynamics)

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(r - z) - y \\ \dot{z} &= xy - \beta z \end{aligned}$$

We're looking for **fixed points**, so set the derivatives to **zero**:

$$\begin{aligned} f(x, y, z) &= \sigma(y - x) = 0 \\ g(x, y, z) &= x(r - z) - y = 0 \\ h(x, y, z) &= xy - \beta z = 0 \end{aligned}$$

Goal: Use Newton's method to find (x, y, z) such that:

$$f(x, y, z) = 0, g(x, y, z) = 0, h(x, y, z) = 0$$

Jacobian Matrix $J(x, y, z)$

Take partial derivatives of f, g, h, f, g, h with respect to x, y, z, x, y, z :

$$J = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} & \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} & \frac{\partial g}{\partial z} & \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} & \frac{\partial h}{\partial z} \end{bmatrix} = \begin{bmatrix} -\sigma & 0 & r - z - 1 - xy & -\beta & 0 & 0 & 0 & 0 & 0 \\ \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} & \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} & \frac{\partial g}{\partial z} & \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} & \frac{\partial h}{\partial z} \end{bmatrix}$$

All values are evaluated at the current guess (x_n, y_n, z_n) .

Newton's Iteration Step:

We solve for:

$$J(x_n, y_n, z_n) \cdot \begin{bmatrix} \Delta x_n \\ \Delta y_n \\ \Delta z_n \end{bmatrix} = - \begin{bmatrix} f(x_n, y_n, z_n) \\ g(x_n, y_n, z_n) \\ h(x_n, y_n, z_n) \end{bmatrix}$$

Where:

- $f(x_n, y_n, z_n) = \sigma(y_n - x_n)$
- $g(x_n, y_n, z_n) = x_n(r - z_n) - y_n$
- $h(x_n, y_n, z_n) = x_n y_n - \beta z_n$

Update:

$$\begin{aligned} x_{n+1} &= x_n + \Delta x_n \\ y_{n+1} &= y_n + \Delta y_n \\ z_{n+1} &= z_n + \Delta z_n \end{aligned}$$

Repeat this until the change $(\Delta x_n, \Delta y_n, \Delta z_n)$ is sufficiently small.

Suggested Homework Implementation (in MATLAB or Python):

- Choose initial guess (x_0, y_0, z_0) — for example: $(1, 1, 1)$
- Set parameters σ, r, β (e.g., $\sigma=10, r=28, \beta=8/3$)
- Iterate Newton's method:
 - Evaluate the Jacobian and the function values
 - Solve the matrix equation for deltas
 - Update x, y, z

Reminder:

- You can analytically solve for the fixed points (try it as an exercise!).
- Newton's method converges **fast**, but only if your **initial guess is close to a root**.

Let me know if you want help implementing this in MATLAB or Python, or if you'd like help solving for the **exact fixed points** analytically.

Here's a **complete summary and action plan** for your **Project 3: Fractals from Newton's Method on Lorenz Equations**.

Project Overview: Newton Fractals in 3D (Lorenz Fixed Points)

You're building a **3D Newton fractal** using the Lorenz system's **fixed points** and visualizing convergence behavior in the **xz-plane** for a **fixed y**.

The Lorenz Fixed Point Equations

Set time derivatives to 0 (fixed points):

$$\begin{aligned} f(x, y, z) &= \sigma(y - x) = 0 \\ g(x, y, z) &= x(r - z) - y = 0 \\ h(x, y, z) &= xy - \beta z = 0 \end{aligned}$$

You already have the Newton iteration set up:

$$J(x, y, z) \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = - \begin{bmatrix} f(x, y, z) \\ g(x, y, z) \\ h(x, y, z) \end{bmatrix}$$

Update:

$$\begin{aligned} x &\leftarrow x + \Delta x \\ y &\leftarrow y + \Delta y \\ z &\leftarrow z + \Delta z \end{aligned}$$

The 3 Fixed Points

1. Origin:

$$(0, 0, 0)$$

2. Positive branch:

$$x = y = \beta(r - 1), z = r - 1$$

3. Negative branch:

$$x = y = -\beta(r - 1), z = r - 1$$

Parameters (from Lorenz's original work)

- $\sigma=10$
- $r=28$
- $\beta=\frac{8}{3}$

Then the square root term becomes:

$$\beta(r-1)=83(27)=72=62\sqrt{\beta(r-1)} = \sqrt{\frac{8}{3}(27)} = \sqrt{72} = 6\sqrt{2}$$

So the two non-zero fixed points are:

- $x=y=\pm 62 \approx \pm 8.485$
- $z=27$

Fractal Mapping Plan

- You'll map a grid in the **xz-plane** (e.g., $-20 \leq x \leq 20$, $-20 \leq z \leq 40$)
- For each grid point (x_0, z_0) , fix:

$$y_0 = 32 \approx 4.24$$

- Use **Newton's method** to determine to which of the 3 fixed points (x, y, z) converges.
- Assign colors:
 - **Red** \rightarrow origin
 - **Green** $\rightarrow +62, +62, 27+6\sqrt{2}, 27$
 - **Blue** $\rightarrow -62, -62, 27-6\sqrt{2}, 27$

Speed Optimization Strategy

 **Don't use a fixed number of iterations. Use a while loop.**

At each grid point:

- Iterate until:

$$\|\Delta x\| + \|\Delta y\| + \|\Delta z\| < \epsilon \quad \|\Delta x\| + \|\Delta y\| + \|\Delta z\| < \epsilon$$

- Use a max iteration cap (e.g., 50) to avoid infinite loops.
- Convergence is **slower near fractal boundaries**, so most time is spent there.

Steps to Implement in MATLAB

1. Define constants: $\sigma = 10$; $\beta = 8/3$; $r = 28$;
2. Define grid for x_0 and z_0

3. Fix $y_0 = 3\sqrt{2}$
 4. For each grid point:
 - Use while loop for Newton's method
 - Compute Jacobian and function values
 - Solve linear system for $\Delta x, \Delta y, \Delta z$
 - Update values
 - Check convergence
 5. Once converged, check final (x, y, z) against fixed points
 - Use $\text{norm}(x - x_{\text{fixed}})$ to match to one of the 3 roots
 - Assign color index: 1 = red, 2 = green, 3 = blue
 6. Plot fractal with `imagesc`, `imshow`, or `scatter`
-

Tips

- Use $\text{norm}([dx, dy, dz]) < \text{tol}$ for convergence check.
 - Use a grid size of 200×200 or smaller if performance is an issue.
 - Store output in a 2D matrix where each cell stores the color (1, 2, 3).
 - Preallocate the matrix for speed.
 - For identifying convergence, use a tolerance (e.g., $1e-6$).
-

Example: MATLAB Color Mapping Snippet

```
colors = [1 0 0; 0 1 0; 0 0 1]; % RGB: Red, Green, Blue
```

```
image_data = zeros(Nx, Nz, 3);
```

```
for i = 1:Nx
```

```
    for j = 1:Nz
```

```
        % Newton iteration...
```

```
        if converged_to_root1
```

```
            image_data(i, j, :) = colors(1, :);
```

```
        elseif converged_to_root2
```

```
            image_data(i, j, :) = colors(2, :);
```

```
        else
```



```
        image_data(i, j, :) = colors(3, :);
    end
end
end
```

```
imshow(image_data);
```

Would you like a full **MATLAB script** to start with? I can provide a complete template including initialization, iteration, convergence test, and plotting.

format compact — Makes output cleaner with less spacing.

- CLC — Clears the **Command Window**.
- ; (semicolon) — Suppresses output when assigning values:
- x = 0; % No output shown
- y = 0; % No output shown
- 2*x % Output: 0

What's Next

- This video focused on **interacting with MATLAB via the command line**.
- In real-world projects, you will write **scripts and functions** for complex tasks.

Let me know if you want practice problems or a cheatsheet for MATLAB operators and functions.

Here's a structured **summary** of the second MATLAB video by **Jeff Chasnov**, focused on **scripts, functions, and programming in MATLAB**:

Scripts vs. Functions in MATLAB

| Feature | Script | Function |
|----------------|---------------------------------------|-------------------------------------|
| Use Case | Series of commands, like a calculator | Reusable code with input and output |
| Variables | Shared with the workspace | Local (isolated from the workspace) |
| Inputs/Outputs | None required | Can have both inputs and outputs |

| Feature | Script | Function |
|---------------|---------------------------------------|--|
| Typical Usage | Simple tasks, plotting, testing ideas | Repetitive or modular code (like defining equations) |

Creating a Script

1. How to Create:

- Click **New Script** (toolbar or File > New > Script)
- Save with a .m extension

2. Start with Clean Slate (Recommended Initial Line):

3. clear all; close all; clc;

4. Example – Golden Ratio:

5. clear all; close all; clc;

6.

7. Phi = (sqrt(5) + 1) / 2;

8. phi = (sqrt(5) - 1) / 2;

Run the script → MATLAB will ask to name the file (e.g., golden_ratio.m)

9. Command Window Check:

10. Phi * phi % ≈ 1 with small roundoff error

11. 1 - ans % Shows the tiny error: -2e-16

Creating a Function

1. How to Create:

- Click **New > Function**
- Template appears automatically

2. Function Example – Euler's Number e:

MATLAB doesn't have a built-in e constant like pi. So:

3. function e = e()

4. e = exp(1);

5. end

Save as: e.m

6. Using the Function:

- Call in the Command Window:
- `e` % returns 2.7183...
- `log(e)` % returns 1

MATLAB will check for a matching file (e.m) in the **current directory**.

Local Functions (Mentioned)

- A **local function** is a helper function written **inside a script**.
 - Common in solving **differential equations** or for organizing code.
-

Best Practices

- Use **scripts** for building programs step by step.
 - Use **functions** for modular, reusable code.
 - Always start fresh with:
 - `clear all; close all; clc;`
-

Conclusion

"You'll write many scripts in this course. By the end I hope you'll be a pretty good MATLAB programmer."

— Jeff Chasnov

Let me know if you'd like help **writing your first function**, practicing exercises, or creating a **template** for scripts or functions in MATLAB.

Here's a clear **summary and explanation** of what Professor Jeff Chasnov derived in this video: the **Midpoint Rule** for numerical integration using **Taylor series**.

Goal: Approximate the integral

We want to compute:

$$I_h = \int_0^h f(x) \, dx \quad I_h = \int_0^h f(x) \, dx$$

This is a small “building block” integral which can later be used in larger composite rules to integrate over intervals like $[a, b]$.

Step-by-step Derivation

1. Expand $f(x)$ as a Taylor Series around $x = h/2$ (the midpoint):

$$f(x) = f(h/2) + (x - h/2)f'(h/2) + \frac{(x - h/2)^2}{2!}f''(h/2) + \dots$$

He writes more terms than needed (to analyze error):

$$f(x) = f(h/2) + (x - h/2)f'(h/2) + \frac{(x - h/2)^2}{2!}f''(h/2) + \frac{(x - h/2)^3}{3!}f'''(h/2) + \frac{(x - h/2)^4}{4!}f^{(4)}(h/2) + \dots$$

2. Substitute this into the integral

Now plug the Taylor expansion into the integral:

$$I_h = \int_0^h f(x) dx = \int_0^h \left(f(h/2) + (x - h/2)f'(h/2) + \frac{(x - h/2)^2}{2!}f''(h/2) + \dots \right) dx$$

3. Change of Variables

Let:

$$y = x - h/2 \Rightarrow dx = dy$$

Limits change from:

- $x = 0 \Rightarrow y = -h/2$
- $x = h \Rightarrow y = +h/2$

Now the integral becomes symmetric:

$$I_h = \int_{-h/2}^{h/2} f(y) dy$$

4. Use symmetry to eliminate odd terms

Since we're integrating over $[-h/2, h/2]$, **odd powers of y** integrate to 0:

Only even terms contribute:

$$I_h = \int_{-h/2}^{h/2} \left(f(h/2) + \frac{y^2}{2!}f''(h/2) + \frac{y^4}{4!}f^{(4)}(h/2) + \dots \right) dy$$

5. Evaluate Integrals

You need:

$$\int_{-h/2}^{h/2} y^2 dy = \frac{h^3}{12}$$

-

$$\int_{-h/2}^{h/2} y^2, dy = \frac{h^3}{12}$$

-

$$\int_{-h/2}^{h/2} y^4, dy = \frac{h^5}{80}$$

Plug back into the formula:

$$I_h = hf(h/2) + h^3 \frac{f''(h/2)}{24} + h^5 \frac{f^{(4)}(h/2)}{1920} + \dots$$

$$I_h \approx h f\left(\frac{h}{2}\right) + \frac{h^3}{24} f''\left(\frac{h}{2}\right) + \frac{h^5}{1920} f^{(4)}\left(\frac{h}{2}\right) + \dots$$

✅ Final Result: The Midpoint Rule

$$\int_0^h f(x) dx \approx h \cdot f\left(\frac{h}{2}\right)$$

- **Error term:** $O(h^3)$

$$\text{Error} = \frac{h^3}{24} f''\left(\frac{h}{2}\right) + \dots$$

🧠 Interpretation

- This is a simple and powerful **numerical integration rule**.
- You're using **only one function evaluation** at the midpoint of the interval.
- Accuracy is better than Left or Right Riemann sums (which are $O(h)$).

Let me know if you'd like the **code implementation** of this in MATLAB or Python—or help deriving **Trapezoidal Rule** or **Simpson's Rule**, which are coming next in the course.

Here's a **clear summary and explanation** of how Professor Jeffrey Chasnov derives the **Trapezoidal Rule** for numerical integration from a **Taylor series expansion**—building on the Midpoint Rule.

📌 Goal

We want to compute:

$$I_h = \int_0^h f(x) dx$$

🔄 Step-by-Step Derivation

1. Recall the Taylor Series about the midpoint $x = h/2$

$$f(x) = f(h/2) + (x - h/2)f'(h/2) + \frac{(x - h/2)^2}{2!}f''(h/2) + \frac{(x - h/2)^3}{3!}f'''(h/2) + \frac{(x - h/2)^4}{4!}f^{(4)}(h/2) + \dots$$

$$\begin{aligned} f(x) &= f\left(\frac{h}{2}\right) + (x - \frac{h}{2})f'\left(\frac{h}{2}\right) + \frac{(x - \frac{h}{2})^2}{2!}f''\left(\frac{h}{2}\right) + \frac{(x - \frac{h}{2})^3}{3!}f'''\left(\frac{h}{2}\right) + \frac{(x - \frac{h}{2})^4}{4!}f^{(4)}\left(\frac{h}{2}\right) + \dots \end{aligned}$$

2. Evaluate the Taylor series at $x=0$ $x = 0$

$$f(0) = f(h/2) - h/2 f'(h/2) + \frac{h^2}{2!} f''(h/2) - \frac{h^3}{3!} f'''(h/2) + \frac{h^4}{4!} f^{(4)}(h/2) + \dots$$

$$\begin{aligned} f(0) &= f\left(\frac{h}{2}\right) - \frac{h}{2} f'\left(\frac{h}{2}\right) + \frac{h^2}{2!} f''\left(\frac{h}{2}\right) - \frac{h^3}{3!} f'''\left(\frac{h}{2}\right) + \frac{h^4}{4!} f^{(4)}\left(\frac{h}{2}\right) + \dots \end{aligned}$$

3. Evaluate the Taylor series at $x=h$ $x = h$

Same structure, but the signs alternate **positively**:

$$f(h) = f(h/2) + h/2 f'(h/2) + \frac{h^2}{2!} f''(h/2) + \frac{h^3}{3!} f'''(h/2) + \frac{h^4}{4!} f^{(4)}(h/2) + \dots$$

$$\begin{aligned} f(h) &= f\left(\frac{h}{2}\right) + \frac{h}{2} f'\left(\frac{h}{2}\right) + \frac{h^2}{2!} f''\left(\frac{h}{2}\right) + \frac{h^3}{3!} f'''\left(\frac{h}{2}\right) + \frac{h^4}{4!} f^{(4)}\left(\frac{h}{2}\right) + \dots \end{aligned}$$

4. Add $f(0)+f(h)$ $f(0) + f(h)$

$$f(0) + f(h) = 2f(h/2) + 2 \cdot \frac{h^2}{2!} f''(h/2) + 2 \cdot \frac{h^4}{4!} f^{(4)}(h/2) + \dots$$

$$\begin{aligned} f(0) + f(h) &= 2f\left(\frac{h}{2}\right) + 2 \cdot \frac{h^2}{2!} f''\left(\frac{h}{2}\right) + 2 \cdot \frac{h^4}{4!} f^{(4)}\left(\frac{h}{2}\right) + \dots \\ &= 2f\left(\frac{h}{2}\right) + \frac{h^2}{1!} f''\left(\frac{h}{2}\right) + \frac{h^4}{2!} f^{(4)}\left(\frac{h}{2}\right) + \dots \end{aligned}$$

Now multiply by $h/2$:

$$\frac{h}{2}(f(0) + f(h)) = hf(h/2) + \frac{h^3}{2!} f''(h/2) + \frac{h^5}{4!} f^{(4)}(h/2) + \dots$$

$$\frac{h}{2}(f(0) + f(h)) = h f\left(\frac{h}{2}\right) + \frac{h^3}{2!} f''\left(\frac{h}{2}\right) + \frac{h^5}{4!} f^{(4)}\left(\frac{h}{2}\right) + \dots$$

5. Use Midpoint Rule for Substitution

From the **Midpoint Rule**, recall:

$$I_h = hf(h/2) + \frac{h^3}{24} f''(h/2) + \frac{h^5}{1920} f^{(4)}(h/2) + \dots$$

$$I_h = h f\left(\frac{h}{2}\right) + \frac{h^3}{24} f''\left(\frac{h}{2}\right) + \frac{h^5}{1920} f^{(4)}\left(\frac{h}{2}\right) + \dots$$

Now, rearrange:

$$hf(h/2) = I_h - \frac{h^3}{24} f''(h/2) - \frac{h^5}{1920} f^{(4)}(h/2) - \dots$$

$$hf(h/2) = I_h - \frac{h^3}{24} f''\left(\frac{h}{2}\right) - \frac{h^5}{1920} f^{(4)}\left(\frac{h}{2}\right) - \dots$$

Plug into the expression for $\frac{h}{2}(f(0) + f(h))$:

$$h^2(f(0)+f(h))=I_h+(h^3-h^3/24)f''(h/2)+(h^5-h^5/1920)f^{(4)}(h/2)+\dots\frac{h^2}{2}(f(0)+f(h))=I_h+\left(\frac{h^3}{8}-\frac{h^3}{24}\right)f''\left(\frac{h}{2}\right)+\left(\frac{h^5}{384}-\frac{h^5}{1920}\right)f^{(4)}\left(\frac{h}{2}\right)+\dots$$

Simplify:

- $h^3-h^3/24=h^3(1-1/24)=\frac{h^3}{12}$
- $h^5-h^5/1920=h^5(1-1/1920)=\frac{h^5}{480}$

So:

$$h^2(f(0)+f(h))=I_h+h^3/12f''(h/2)+h^5/480f^{(4)}(h/2)+\dots\frac{h^2}{2}(f(0)+f(h))=I_h+\frac{h^3}{12}f''\left(\frac{h}{2}\right)+\frac{h^5}{480}f^{(4)}\left(\frac{h}{2}\right)+\dots$$

Solve for I_h :

$$I_h=h^2(f(0)+f(h))-h^3/12f''(h/2)-h^5/480f^{(4)}(h/2)+\dots\boxed{I_h=\frac{h^2}{2}(f(0)+f(h))-\frac{h^3}{12}f''\left(\frac{h}{2}\right)-\frac{h^5}{480}f^{(4)}\left(\frac{h}{2}\right)+\dots}$$

✓ Final Result: The Trapezoidal Rule

$$\int_0^h f(x) dx \approx \frac{h}{2} [f(0) + f(h)] \quad \boxed{\int_0^h f(x) dx \approx \frac{h}{2} [f(0) + f(h)]}$$

- **Error term:** $O(h^3)$

$$\text{Error} = -\frac{h^3}{12}f''(h/2) + \dots \quad \text{Error} = -\frac{h^3}{12}f''\left(\frac{h}{2}\right) + \dots$$

💡 Intuition

- Approximates area under the curve using a **trapezoid** instead of a rectangle.
- Uses **two evaluations**: one at each endpoint.
- Better than Left/Right Riemann sum, slightly worse than Midpoint Rule in error constant.

Let me know if you want to see the **Simpson's Rule** next (which uses three points and has error $O(h^5)$).

Here's a **clear summary and explanation** of how **Simpson's Rule** is derived by **combining the Midpoint Rule and the Trapezoidal Rule**, as explained by Professor Jeffrey Chasnov.

📌 Goal

We want a more **accurate numerical integration** rule for:

$$\int_0^h f(x) dx$$

🔄 Step-by-Step Derivation

1. Recall the Midpoint and Trapezoidal Rules

- **Midpoint Rule:**

$$I_h = h \left[f\left(\frac{h}{2}\right) - \frac{h^3}{24} f''\left(\frac{h}{2}\right) + \frac{h^5}{1920} f^{(4)}\left(\frac{h}{2}\right) + \dots \right] \\ f''\left(\frac{h}{2}\right) + \frac{h^5}{1920} f^{(4)}\left(\frac{h}{2}\right) + \dots$$

- **Trapezoidal Rule:**

$$I_h = \frac{h}{2} (f(0) + f(h)) - \frac{h^3}{12} f''\left(\frac{h}{2}\right) + \frac{h^5}{480} f^{(4)}\left(\frac{h}{2}\right) + \dots \\ f''\left(\frac{h}{2}\right) + \frac{h^5}{480} f^{(4)}\left(\frac{h}{2}\right) + \dots$$

2. Cancel the h^3 Error Term by Combining

We add twice the midpoint rule and once the trapezoidal rule:

$$\text{Let: } 2 \cdot (\text{Midpoint}) + 1 \cdot (\text{Trapezoidal}) = 3I_h + (\text{higher-order error}) \\ 2 \cdot (\text{Midpoint}) + 1 \cdot (\text{Trapezoidal}) = 3I_h + (\text{higher-order error})$$

Left-hand side becomes:

$$h^2 [f(0) + f(h)] + 2hf\left(\frac{h}{2}\right) - \frac{h^3}{2} f''\left(\frac{h}{2}\right) + 2hf\left(\frac{h}{2}\right) - \frac{h^3}{2} f''\left(\frac{h}{2}\right)$$

Right-hand side:

$$= 3I_h + O(h^5) = 3I_h + \mathcal{O}(h^5)$$

Now solve for I_h by dividing by 3:

3. Simpson's Rule Formula

$$I_h = \frac{h}{6} [f(0) + 4f\left(\frac{h}{2}\right) + f(h)] + O(h^5) \\ \mathcal{O}(h^5)$$

This is **Simpson's Rule over the interval $[0, h]$** .

4. Standard Form Over $[0, 2h]$

To make the formula look more standard, consider:

$$\int_0^{2h} f(x) dx$$

Now:

- Interval length = $2h$
- Midpoint = h

Apply Simpson's Rule:

$$\int_0^{2h} f(x) dx \approx \frac{2h}{6} [f(0) + 4f(h) + f(2h)] = \frac{h}{3} [f(0) + 4f(h) + f(2h)] \\ \frac{h}{3} [f(0) + 4f(h) + f(2h)]$$

- **Error:** $O(h^5)$

Summary of All Three Elementary Rules

| Rule | Formula | Error Term | Points Used |
|-------------|--|------------|----------------------|
| Midpoint | $h f\left(\frac{h}{2}\right)$ | $O(h^3)$ | Midpoint only |
| Trapezoidal | $\frac{h}{2}[f(0) + f(h)]$ | $O(h^3)$ | Endpoints only |
| Simpson's | $\frac{h^3}{6}[f(0) + 4f\left(\frac{h}{2}\right) + f(h)]$ (or $\frac{h^3}{3} \cdots$) | $O(h^5)$ | Endpoints + Midpoint |

Insight

- Simpson's Rule cleverly **eliminates the h^3 error** by combining the Midpoint and Trapezoidal Rules.
- It gives **much better accuracy** using only **three points**.
- Often used in **composite form** to integrate over longer intervals.

Let me know if you'd like to see **Composite Simpson's Rule** next!

Here's a **complete summary and explanation** of the video on **composite quadrature rules**, including how to construct and apply the **composite trapezoidal** and **composite Simpson's rule** for numerically integrating a function over an interval $[a, b]$.

Goal

We want to compute the integral:

$$\int_a^b f(x) dx$$

by breaking the interval $[a, b]$ into smaller subintervals and applying our **elementary integration formulas** (Trapezoidal and Simpson's rule) over each subinterval.

Step 1: Divide the Interval

- Divide $[a, b]$ into n subintervals:

$$a = x_0 < x_1 < x_2 < \cdots < x_n = b$$

- Each subinterval is $[x_i, x_{i+1}]$ with width:

$$h_i = x_{i+1} - x_i$$

- Function values:

$$f_i = f(x_i) \quad f_{i+1} = f(x_{i+1})$$

Then:

$$\int_a^b f(x) dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x) dx \quad \Delta x = \sum_{i=0}^{n-1} (x_{i+1} - x_i) = b - a$$

Apply elementary rules to each subinterval.

Composite Trapezoidal Rule

For arbitrary (unequal) subintervals:

$$\int_{x_i}^{x_{i+1}} f(x) dx \approx \frac{h_i}{2} [f(x_i) + f(x_{i+1})]$$

So the full integral:

$$\int_a^b f(x) dx \approx \sum_{i=0}^{n-1} \frac{h_i}{2} [f(x_i) + f(x_{i+1})]$$

 **Works well for experimental data with uneven spacing.**

For equally spaced points:

Let $h = \frac{b-a}{n}$, then:

$$\int_a^b f(x) dx \approx \frac{h}{2} [f_0 + 2f_1 + 2f_2 + \dots + 2f_{n-1} + f_n]$$

- Endpoints f_0 and f_n appear once
- All interior points f_1 to f_{n-1} appear **twice**

 **Error term:**

$$O(h^2)$$

(because error from each subinterval is $O(h^3)$, and we have $n = \frac{b-a}{h}$ terms, so total error is $O(h^2)$)

Composite Simpson's Rule

Apply the elementary Simpson's rule over **pairs of subintervals**, so:

- Use **even number of subintervals**: n must be **even**
- Interval width: $h = \frac{b-a}{n}$

Formula:

$$\int_a^b f(x) dx \approx \frac{h}{3} [f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \dots + 4f_{n-1} + f_n]$$

📌 Pattern:

- First and last: f_0 and f_n appear once
- Odd indices f_1, f_3, \dots, f_{n-1} : coefficient **4**
- Even indices f_2, f_4, \dots, f_{n-2} : coefficient **2**

🧠 Error term:

$$O(h^4)$$

Very accurate compared to trapezoidal.

✅ Summary Table

| Rule | Formula | Error | Requirements |
|------------------------------|---|----------|-------------------------|
| Composite Trapezoidal | $\frac{h}{2} [f_0 + 2f_1 + 2f_2 + \dots + 2f_{n-1} + f_n]$ | $O(h^2)$ | Any n , even or odd |
| Composite Simpson's | $\frac{h}{3} [f_0 + 4f_1 + 2f_2 + 4f_3 + \dots + 4f_{n-1} + f_n]$ | $O(h^4)$ | n must be even |

🧠 Practical Use

- **Composite trapezoidal**: Great for **experimental data** where x_i values are uneven.
- **Composite Simpson's**: Ideal when you **can choose function points**, and want **more accuracy**.

Let me know if you'd like to see a Python or MATLAB example applying these rules!

Here's a clear **summary** of the second part of the lecture video on **Gaussian Quadrature**, presented by Jeff Chesnoff:

📊 Gaussian Quadrature: Overview

Gaussian quadrature is a **numerical integration** technique that differs from the trapezoidal and Simpson's rules. It's **more accurate**, especially for polynomials, and widely used in computational tools like MATLAB (integral.m uses it internally).

✅ General Form of Gaussian Quadrature

It approximates the integral of a weighted function:

$$\int_a^b W(x) f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

- **$W(x)$** : Weight function (varies based on the type of Gaussian quadrature)

- **w_i**: Quadrature weights
- **x_i**: Quadrature nodes (points at which the function is evaluated)

Types of Gaussian Quadrature

Each type uses a specific **weight function** and **interval**:

| Type | Weight Function $W(x)$ | Interval |
|------------------------|--------------------------|---------------------|
| Legendre-Gauss | 1 | $[-1, 1]$ |
| Chebyshev-Gauss | $\frac{1}{\sqrt{1-x^2}}$ | $[-1, 1]$ |
| Laguerre-Gauss | e^{-x} | $[0, \infty)$ |
| Hermite-Gauss | e^{-x^2} | $(-\infty, \infty)$ |

Legendre-Gauss is the most general-purpose and often used in numerical integration.

Example: 2-Point Legendre-Gauss Quadrature

To integrate $\int_{-1}^1 f(x) dx$, Gaussian quadrature finds weights and nodes such that the formula is exact for polynomials up to **degree 3**.

$$\int_{-1}^1 f(x) dx \approx w_1 f(x_1) + w_2 f(x_2)$$

Solving the system for $f(x) = 1, x, x^2, x^3$ gives:

- **Weights:** $w_1 = w_2 = 1$
- **Nodes:** $x_1 = -\frac{1}{\sqrt{3}}, x_2 = \frac{1}{\sqrt{3}}$

 Final rule:

$$\int_{-1}^1 f(x) dx \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right)$$

Practical Notes

- Higher-order Gaussian quadrature uses more points and has precomputed weights and nodes (in tables or software).
- **MATLAB's integral.m function** uses adaptive Gaussian quadrature internally.
- You don't typically calculate nodes/weights manually—just understand the concept.

Conclusion

Gaussian quadrature:

- Is highly accurate for smooth functions.
- Uses specially chosen nodes and weights for efficient integration.
- Is widely implemented in software—users benefit from its accuracy without needing to manually compute anything.

Let me know if you want a comparison chart with error rates of Trapezoidal, Simpson's, and Gaussian methods, or examples in Python or MATLAB.

Here's a **clear summary** of the lecture on **Adaptive Quadrature** by Jeff Chesnoff:

🌟 Adaptive Quadrature: Concept and Motivation

- **Goal:** Automatically determine where to place function evaluation points (x_i) to achieve a desired **accuracy** without the user needing to specify grid spacing.
- This is what **MATLAB's integral.m** function does behind the scenes.
- It adapts the grid **based on how the function behaves**, refining where needed.

🔧 Method: Using the Trapezoidal Rule

To explain adaptive quadrature, the **trapezoidal rule** is used as an example.

◆ Step 1: Start with Level 1 (coarse)

- Integrate from a to b using just the endpoints:

$$S_1 = \frac{h}{2} [f(a) + f(b)], h = b - a$$

- **Error estimate:**

$$E_1 = -\frac{h^3}{12} f''(\xi) \quad \text{(for some } \xi \in (a, b))$$

◆ Step 2: Move to Level 2 (refined)

- Add a **midpoint** $c = \frac{a+b}{2}$ and compute:

$$S_2 = \frac{h}{4} [f(a) + 2f(c) + f(b)]$$

- Now you have two trapezoidal areas: from a to c , and c to b .
- **Assume** the second derivatives across both subintervals are roughly equal, giving:

$$E_2 \approx \frac{1}{4} E_1$$

📊 Estimating the Error

Since both integrals should approximate the same true value:

$$S_1 + E_1 = S_2 + E_2 \quad S_1 + E_1 = S_2 + E_2$$

Using the assumption $E_1 = 4E_2$, you get:

$$S_1 + 4E_2 = S_2 + E_2 \Rightarrow 3E_2 = S_2 - S_1 \quad S_1 + 4E_2 = S_2 + E_2 \Rightarrow 3E_2 = S_2 - S_1$$

 **Error Estimate:**

$$|E_2| = |S_2 - S_1| / 3 \quad |E_2| = \frac{|S_2 - S_1|}{3}$$

Recursive Refinement (Adaptive Loop)

If the estimated error E_2 exceeds your tolerance:

- **Subdivide** again: from a to c and from c to b .
- Apply the same logic **independently to each subinterval**.
- Keep going until the local error on each subinterval is **below a decreasing fraction of the total tolerance** (e.g., half the tolerance for two subintervals).

Key Points About Adaptive Quadrature

- It **automatically refines** where the function is difficult (wiggly).
- It uses **fewer points** where the function is smooth (almost constant).
- **Efficient and precise**, especially for irregular functions.
- **MATLAB's integral.m** function:
 - Uses adaptive quadrature.
 - Has a **default tolerance**, or you can specify your own.
 - **You don't need to choose a step size or grid spacing**—the algorithm handles it.

Summary

| Concept | Meaning |
|-------------------------|---|
| S_1, S_2 | Two estimates using different numbers of points |
| E_1, E_2 | Corresponding error estimates |
| Adaptive process | Keep subdividing intervals where error > tolerance |
| Efficiency | Saves time and resources by focusing on problem areas |
| Used in | MATLAB integral.m, SciPy quad(), and many other numerical libraries |

Let me know if you'd like a **visual flowchart**, Python code for adaptive quadrature, or MATLAB example!

Here's a **concise summary** of the video tutorial on how to compute **definite integrals (quadrature)** using MATLAB:

Goal

Evaluate the definite integral:

$$\int_0^{\infty} e^{-ax^2} (\log x)^2 dx \quad \int_0^{\infty} e^{-a x^2} (\log x)^2 \, dx$$

with the parameter $a = \pi$, using MATLAB.

Key Concepts

- MATLAB function: **integral**
 - Syntax:
 - `integral(@(x) f(x), lower_limit, upper_limit)`
 - The **@ symbol** defines an anonymous function.
-

Step-by-Step MATLAB Implementation

1. **Define the integrand** as a function of x and parameter a :
 2. `f = @(x, a) exp(-a * x.^2) .* (log(x)).^2;`
 - `.^2` → element-wise square for vectors
 - `.*` → element-wise multiplication
 3. **Set parameter a to π :**
 4. `a = pi;`
 5. **Define the integral using `integral` and fix a :**
 6. `I = integral(@(x) f(x, a), 0, Inf);`
 7. **Run it**, and you'll get the result:
 8. `I = 1.8245`
-

Key Takeaways

| Concept | Description |
|------------------------|---|
| @ (anonymous function) | Needed to pass functions to <code>integral</code> |

| Concept | Description |
|---------------------------|---|
| integral() | MATLAB's adaptive quadrature function |
| Handles Inf limits | No need to truncate manually |
| Supports parameters | Use extra arguments like a via anonymous function |
| Automatically adapts grid | Uses adaptive quadrature internally |

Use Case

- Useful for integrals involving **parameters**, **infinite limits**, or **complex functions** like **Bessel functions** (as mentioned for later in the course).

Let me know if you'd like to see a **visual plot**, a **parameter sweep**, or a **Python equivalent** of this code!

Here's a **summary** of the video on **Piecewise Linear Interpolation**:

What is Interpolation?

Interpolation is a numerical method where:

- You are **given $n+1$ points**:
 $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$
- These y_i values are from an unknown function $y = f(x)$, evaluated at known x_i points.
- The **goal** is to estimate y at some new value x **not in the list** x_0, x_1, \dots, x_n .

Types of Interpolation

1. Polynomial Interpolation (Global)

- Fit a **single polynomial** of degree n through all $n+1$ points.
- Accurate for **2–3 points**, but:
 - For many points, causes unwanted **oscillations** ("wiggles").
 - Not preferred for large datasets.

2. Piecewise Linear Interpolation (Local)

- Use a **straight line between each pair** of consecutive points.
- Simple, fast, and avoids wiggles.

- Used by MATLAB's `plot()` function.

3. Cubic Spline Interpolation (explained in next videos)

- Uses **piecewise cubic polynomials**.
- Results in a **smoother** curve (continuous derivatives).

Piecewise Linear Interpolation – How It Works

Define a piecewise linear function $g(x)$, such that:

$$g(x) = g_i(x) \text{ for } x \in [x_i, x_{i+1}] \quad g(x) = g_i(x) \quad \text{for } x \in [x_i, x_{i+1}]$$

Each **segment** $g_i(x)$ is a line:

$$g_i(x) = a_i(x - x_i) + b_i \quad g_i(x) = a_i(x - x_i) + b_i$$

How to Find a_i and b_i

You have 2 constraints for each line:

1. $g_i(x_i) = y_i$
2. $g_i(x_{i+1}) = y_{i+1}$

From these:

- $b_i = y_i$
- $a_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \leftarrow$ this is the **slope**

Graphical Understanding

- Points: $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$
- Connect each pair with a straight line \rightarrow forms a **piecewise linear** curve $g(x)$.
- Function is **continuous**, but the **slope (derivative) is not**.

Summary

| Concept | Explanation |
|------------|--------------------------------------|
| Given | $n+1$ data points (x_i, y_i) |
| Goal | Estimate y for x not in the data |
| Method | Piecewise linear functions $g_i(x)$ |
| Continuity | $g(x)$ is continuous |

| Concept | Explanation |
|----------------------|--|
| Derivatives | Discontinuous at data points |
| Form of Each Segment | $g_i(x) = a_i(x - x_i) + y_i$ |
| Used When | Simple, quick interpolation with few assumptions |

Let me know if you want a **MATLAB implementation**, **code to visualize it**, or a **comparison with cubic spline interpolation**.

Here's a **summary** of the main points from the video on **Cubic Spline Interpolation (Part A)** by Prof. Jeff Chasnov:

Objective:

To interpolate a function using **piecewise cubic polynomials** — a smoother alternative to piecewise linear interpolation.

Problem Setup:

- Given $n+1$ data points:

$(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$

- Goal: Find a smooth function $g(x)$ such that $g(x_i) = y_i$, and also get good estimates of $g(x)$ at other x -values.

Cubic Spline Interpolation:

- Break the x -intervals into **n subintervals**.
- On each interval $[x_i, x_{i+1}]$, define a **cubic polynomial**:

$$g_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

- Total unknowns: $4n$ (since each of the n intervals has 4 coefficients).

Constraints (to determine the unknowns):

1. Interpolation Constraints ($2n$):

- Pass through both endpoints of each interval:

$$g_i(x_i) = y_i, g_i(x_{i+1}) = y_{i+1}, g_{i+1}(x_{i+1}) = y_{i+1}, g_{i+1}(x_{i+2}) = y_{i+2}$$

2. First Derivative Continuity ($n-1$):

- Ensure slope continuity at internal nodes:

$$g'_i(x_{i+1}) = g_{i+1}'(x_{i+1}) \quad g'_i(x_{i+1}) = g'_{i+1}(x_{i+1})$$

3. Second Derivative Continuity (n-1):

- Ensure curvature continuity at internal nodes:

$$g''_i(x_{i+1}) = g_{i+1}''(x_{i+1}) \quad g''_i(x_{i+1}) = g''_{i+1}(x_{i+1})$$

Counting Constraints:

- Interpolation: $2n$
- First derivative continuity: $n-1$
- Second derivative continuity: $n-1$

Total so far: $2n + 2(n-1) = 4n - 2$

→ **Missing 2 constraints** to fully determine the spline.

Two Extra Constraints (needed):

- Will be discussed in the next video (e.g., "**not-a-knot**" condition or specifying second derivatives at the boundaries).

Defining Helpful Quantities:

- Interval width: $h_i = x_{i+1} - x_i$
- Output difference: $\eta_i = y_{i+1} - y_i$

Summary of Coefficient Determination:

1. $d_i = y_i$
2. $a_i h_i^3 + b_i h_i^2 + c_i h_i + d_i = y_{i+1}$
3. First derivative continuity at x_{i+1} :

$$3a_i h_i^2 + 2b_i h_i + c_i = c_{i+1}$$

4. Second derivative continuity at x_{i+1} :

$$6a_i h_i + 2b_i = 2b_{i+1}$$

Insight:

You don't usually compute the spline by hand — MATLAB or Python can do it using built-in functions — but understanding the derivation helps you understand how the interpolation is smooth and continuous.

Let me know if you'd like a simplified derivation or code implementation in Python or MATLAB.

Here's a **summary and explanation** of the full cubic spline interpolation analysis from both parts of Jeff Chasnov's lecture:

Goal

To construct a **smooth interpolating function** that passes through $n+1$ known points $(x_0, y_0), \dots, (x_n, y_n)$ using **piecewise cubic polynomials** (called cubic splines).

Structure of the Interpolant

We build n cubic polynomials:

$$g_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i, \text{ for } i = 0 \text{ to } n-1$$

Each has **4 coefficients**, leading to **$4n$ unknowns**.

Constraints (Total: $4n - 2$ before final fix)

1. Function passes through data points ($2n$ constraints):


- $g_i(x_i) = y_i \Rightarrow d_i = y_i \Rightarrow d_i = y_i$
- $g_i(x_{i+1}) = y_{i+1} \Rightarrow$ gives equation in $a_i, b_i, c_i, d_i \Rightarrow$ gives equation in a_i, b_i, c_i

2. First derivative continuity at internal knots ($n - 1$ constraints):

$$g'_i(x_{i+1}) = g'_{i+1}(x_{i+1}) \Rightarrow g'_i(x_{i+1}) = g'_{i+1}(x_{i+1})$$

3. Second derivative continuity at internal knots ($n - 1$ constraints):

$$g''_i(x_{i+1}) = g''_{i+1}(x_{i+1}) \Rightarrow g''_i(x_{i+1}) = g''_{i+1}(x_{i+1})$$

 **Total so far: $2n + (n-1) + (n-1) = 4n - 2$**

Still Need 2 More Constraints

To fully solve for the $4n$ coefficients, **2 more equations are needed**.

 **Not-a-Knot Condition (Default in MATLAB)**

This assumes the spline is "extra smooth" at the first and last interior points:

- Require the third derivatives to be equal at the second and second-to-last point:

$$g_0'''(x_1) = g_1'''(x_1) \Rightarrow a_0 = a_1 \quad g_0'''(x_1) = g_1'''(x_1) \Rightarrow a_0 = a_1$$

$$g_{n-2}'''(x_{n-1}) = g_{n-1}'''(x_{n-1}) \Rightarrow a_{n-2} = a_{n-1} \quad g_{n-2}'''(x_{n-1}) = g_{n-1}'''(x_{n-1}) \Rightarrow a_{n-2} = a_{n-1}$$

✓ Now total constraints = $4n$

Final Form: Matrix Equation

After substituting and eliminating a_i , c_i , and d_i , we get a **tridiagonal matrix equation** for the b_i coefficients:

$$13h_i b_i + 23(h_i + h_{i+1})b_{i+1} + 13h_{i+1}b_{i+2} = \eta_i + 13h_i + 13h_{i+1} - \eta_{i+1} \frac{1}{3} h_i b_i + \frac{2}{3} (h_i + h_{i+1}) b_{i+1} + \frac{1}{3} h_{i+1} b_{i+2} = \frac{\eta_{i+1}}{h_{i+1}} - \frac{\eta_i}{h_i}$$

Where:

- $h_i = x_{i+1} - x_i$
- $\eta_i = y_{i+1} - y_i$

This leads to a **linear system** of size $(n+1) \times (n+1)$.

Solving It

- This tridiagonal system is solved using matrix operations (e.g., LU decomposition).
- Once b_i 's are found, plug them back to find a_i , c_i , and d_i .

In Practice

- You don't solve this by hand.
- Use built-in software routines (e.g., `spline()` in MATLAB or `scipy.interpolate.CubicSpline()` in Python).
- MATLAB uses **not-a-knot** by default, but can be changed to **clamped**, **natural**, or other types of boundary conditions.

Key Advantages of Cubic Splines

- **Smoothness:** Continuous first and second derivatives.
- **Stability:** Less oscillatory than high-degree single polynomials.
- **Accuracy:** Great for smooth interpolation of real-world data.

Would you like me to generate the matrix form visually or in code (MATLAB/Python)?

Here's a **summary and breakdown** of Jeff Chasnov's MATLAB demonstration on **cubic spline interpolation using interp1**:

Goal

Use MATLAB's `interp1` function to interpolate known data points and evaluate how well **cubic spline interpolation** reconstructs a smooth function—in this case, the sine function.

Step-by-Step in MATLAB

1. Define Sample Data Points

Create x values from 0 to 2π , spaced by $\frac{\pi}{4}$, and compute their sine values.

```
x = 0:pi/4:2*pi;
```

```
y = sin(x);
```

2. Plot Original Points

```
plot(x, y, 'o'); % circle markers for original data
```

```
hold on;
```

You'll see widely spaced points lying on the sine curve.

3. Create a Fine Grid for Interpolation

```
xx = linspace(0, 2*pi, 1000); % finer x-grid
```

This gives a smoother appearance when interpolating.

4. Perform Interpolation Using Cubic Spline

```
yy = interp1(x, y, xx, 'spline');
```

- 'spline' uses **cubic spline interpolation** with **not-a-knot** condition.
 - You can also use 'linear' if you want piecewise linear interpolation.
-

5. Plot the Interpolated Curve

```
plot(xx, yy, 'r'); % red interpolated curve
```

6. Compare with Actual Sine Function

```
plot(xx, sin(xx), 'b'); % actual sine in blue
```

When you run this, you'll notice:

- The **red** spline-interpolated curve follows the **blue** actual sine curve very closely.
- With zooming in, small differences can be observed—especially at the edges.

Conclusion

- `interp1` with 'spline' gives a **very accurate and smooth interpolation**, especially useful for approximating smooth functions like $\sin(x)$.
- **MATLAB's default spline method is not-a-knot**, making the interpolated curve extra smooth at boundaries.

Useful Command Summary

```
x = 0:pi/4:2*pi;
```

```
y = sin(x);
```

```
xx = linspace(0, 2*pi, 1000);
```

```
yy = interp1(x, y, xx, 'spline');
```

```
plot(x, y, 'o'); hold on;
```

```
plot(xx, yy, 'r');
```

```
plot(xx, sin(xx), 'b');
```

Would you like a version of this in **Python** with `scipy.interpolate` as well?

Here's a clear **summary and guidance** for your project on **finding zeros of Bessel functions** based on Jeff Chasnov's video:

Project Objective

Find the first five zeros of the first six Bessel functions $J_n(x)$ for $n=0,1,\dots,5$.

What You Need to Do

1. **Understand the Bessel function definition:**

The n^{th} Bessel function of the first kind $J_n(x)$ is defined by the integral:

$$J_n(x) = \frac{1}{\pi} \int_0^\pi \cos(x \sin \theta - n\theta) d\theta. \quad J_n(x) = \frac{1}{\pi} \int_0^\pi \cos\left(x \sin \theta - n\theta\right) d\theta.$$

2. Plot the Bessel functions to understand their shape:

- For each n , plot $J_n(x)$ for a range of x .
- The zeros of $J_n(x)$ are the points where it crosses zero on the x -axis.

3. Find zeros (roots) of $J_n(x)$:

- You want to find the first five zeros for each J_n .
- This is a **root-finding problem** where you find x such that $J_n(x) = 0$.

4. Tools to use in MATLAB:

- `integral.m` to numerically evaluate the integral defining $J_n(x)$.
- `fzero.m` to find roots of $J_n(x)$.

5. Code Structure:

- Use **nested loops**: outer loop over $n=0, \dots, 5$ (which Bessel function), inner loop over zeros $j=1, \dots, 5$.
- Use initial guesses for the zeros to feed into `fzero`. The zeros can be estimated by plotting or taken from tables (provided initial guesses in your template).
- Refine each root with `fzero` to get accurate numerical values.

MATLAB Implementation Sketch

% Function to compute $J_n(x)$ via integral

```
function val = Jn(n, x)
```

```
    integrand = @(theta) cos(x .* sin(theta) - n .* theta);
```

```
    val = (1/pi) * integral(integrand, 0, pi);
```

```
end
```

% Number of zeros and functions

```
numZeros = 5;
```

```
maxN = 5;
```

% Initial guesses for zeros (can be from a table or approximate)

```
initialGuesses = [...]; % 6x5 matrix or cell array, each row for n, each col for zero index
```

```
for n = 0:maxN
```

```
    fprintf('Zeros for  $J_{%d}(x)$ : \n', n);
```



```

for k = 1:numZeros
    root = fzero(@(x) Jn(n, x), initialGuesses(n+1, k));
    fprintf(' Zero %d: %.6f\n', k, root);
end
end

```

Tips

- Plot $J_n(x)$ first for $n=0, \dots, 5$ and x in a reasonable range to visually identify where zeros lie.
 - Use `linspace` and a vectorized form of your integral function for plotting.
 - The function `fzero` requires an initial guess near the root.
 - The **not-a-knot** condition or interpolation is not used here; this project focuses on integral evaluation and root finding.
-

If you want, I can help you build a **complete MATLAB script** for this project or even translate it into **Python** with `scipy.integrate.quad` and `scipy.optimize.root_scalar`. Would that help?

Here's a clear summary and some guidance for your **Week 5** content in Numerical Methods for Engineers, focusing on **ODEs**:

Week 5 Overview: Solving Ordinary Differential Equations (ODEs)

◆ Why ODEs matter:

- Many engineering problems are modeled by ODEs.
- Example: Newton's Second Law $F=ma$ leads to a second-order ODE describing motion.

◆ Topics you will cover:

1. **Initial Value Problems (IVPs)** — solving ODEs given initial conditions.
2. **Euler's Method** — the simplest numerical integration method.
3. **Modified Euler Method** — improves accuracy by correcting the slope estimate.
4. **Runge-Kutta Methods** — more accurate, widely used methods (including adaptive versions).
5. **Adaptive Runge-Kutta Methods** — algorithms that automatically adjust the step size for efficiency and accuracy.
6. **MATLAB's ode45** — an implementation of an adaptive Runge-Kutta method you'll use.
7. **Higher-order ODEs and systems** — transforming them into first-order systems for numerical solving.

8. **Shooting Method** — to solve two-point boundary value problems by converting them to root-finding problems.
-

Your Week 5 Project: The Two-Body Problem

- Model the motion of two masses interacting gravitationally via Newton's Laws.
 - You will solve the system of ODEs describing their positions and velocities.
 - This will involve numerical integration of the equations of motion.
 - Likely you will use methods like Runge-Kutta (or MATLAB's ode45) to simulate their trajectories.
-

Tips for Success:

- Review how to convert second-order ODEs into first-order systems (by defining velocity and position variables).
 - Practice implementing Euler and Runge-Kutta methods on simpler problems.
 - Familiarize yourself with MATLAB's ode45 syntax and options.
 - Understand the shooting method basics if you encounter boundary value problems.
-

If you want, I can help you by:

- Providing example MATLAB code for Euler, modified Euler, or Runge-Kutta methods.
- Guiding you step-by-step on how to implement the two-body problem solver.
- Explaining the shooting method or adaptive time stepping.

Would you like to start with any of those?

Here's a concise breakdown of the **Euler Method** explanation from the video transcript, plus some key points:

Euler Method for Solving ODEs: Overview

Problem Setup:

- Solve an ODE of the form:

$$\frac{dx}{dt} = f(t, x)$$

- Initial condition:

$$x(t_0) = x_0$$

- Goal: Find $x(t)$ for $t > t_0$.

How Euler's Method Works:

1. **Start at initial point** (t_0, x_0) .
2. Compute slope (derivative) at this point:

$$f(t_0, x_0) = \left. \frac{dx}{dt} \right|_{t=t_0, x=x_0}$$

3. Approximate solution at the next time step $t_1 = t_0 + \Delta t$ by moving along the tangent line:

$$x_1 = x_0 + \Delta t \cdot f(t_0, x_0)$$

4. Repeat for subsequent steps:

$$x_{n+1} = x_n + \Delta t \cdot f(t_n, x_n)$$

Intuition:

- Euler's method marches forward by using the slope at the current point to predict the next value.
 - It assumes the slope stays constant over the small interval Δt .
 - This is like stepping along the tangent line at each point.
-

Limitations:

- Since slope can change over the interval, this method accumulates error.
 - Euler's method is **first order accurate** — error per step scales with Δt^2 , but overall error scales with Δt .
 - Not very accurate for stiff or complex problems.
-

Next Steps:

- More accurate methods improve by sampling slopes at multiple points in the interval.
 - This leads to **Runge-Kutta methods**, which you will study next.
 - Runge-Kutta methods reduce error and improve stability.
-

If you want, I can help you:

- Write MATLAB code for Euler's method on a simple ODE.
- Explain the error behavior with examples.
- Introduce the modified Euler or basic Runge-Kutta methods next.

Would you like to start coding Euler's method or learn more theory?

Here's a clear summary of the **Modified Euler Method** (also known as the second-order Runge-Kutta method, RK2) explained in the video:

Modified Euler Method (RK2)

Starting Point:

We want to solve:

$$\frac{dx}{dt} = f(t, x)$$

with initial condition x_n at time t_n .

Problem with Euler Method:

- Euler uses only the slope at the start of the interval t_n .
 - But slope generally changes over the interval $[t_n, t_n + \Delta t]$.
 - So Euler's method is not very accurate.
-

Idea of Modified Euler:

- Use **two slope evaluations**:
 - $f(t_n, x_n)$, the slope at the start.
 - $f(t_n + \Delta t, x_{n+1})$, the slope at the end.
- Take the **average slope** to get better accuracy:

$$x_{n+1} = x_n + \Delta t \cdot \frac{f(t_n, x_n) + f(t_n + \Delta t, x_{n+1})}{2}$$

Issue:

- x_{n+1} appears on both sides of the equation — implicit formula.
- To solve, use a **predictor-corrector approach**:

1. Predictor (Euler step):

$$x_{n+1}^{(p)} = x_n + \Delta t \cdot f(t_n, x_n)$$

2. Corrector (use predicted value):

$$x_{n+1} = x_n + \Delta t \cdot \frac{f(t_n, x_n) + f(t_n + \Delta t, x_{n+1}^{(p)})}{2}$$

This improves accuracy by averaging the slopes at the start and predicted end of the interval.

Runge-Kutta Notation:

- Define stages K_1 and K_2 :

$$K_1 = \Delta t \cdot f(t_n, x_n) \quad K_2 = \Delta t \cdot f(t_n + \Delta t, x_n + K_1)$$

- Then:

$$x_{n+1} = x_n + \frac{1}{2}(K_1 + K_2)$$

Summary:

- Modified Euler method is a simple 2nd order Runge-Kutta method.
- More accurate than Euler because it uses the slope at both the beginning and the end of the interval.
- It introduces the idea of "stages" K_i , foundational for more advanced Runge-Kutta methods.
- Next step: generalize to other RK methods that sample slope at other points and have higher order accuracy.

If you'd like, I can help you:

- Write MATLAB or Python code for this Modified Euler method.
- Show how the error compares with Euler's method on an example ODE.
- Explain how to extend this to 4th order Runge-Kutta (RK4), the most common method.

Would you like to do one of these?

Here's a detailed but clear explanation of the **full family of second-order Runge-Kutta (RK2) methods**, based on the video:

General Second-Order Runge-Kutta Methods (RK2)

We want to solve the ODE:

$$\frac{dx}{dt} = f(t, x)$$

from time t_n to $t_{n+1} = t_n + \Delta t$, starting with value x_n .

The General RK2 Scheme

- Stage 1:**

$$K_1 = \Delta t \cdot f(t_n, x_n)$$

2. Stage 2:

$$K_2 = \Delta t \cdot f(t_n + \alpha \Delta t, x_n + \beta K_1) \quad K_2 = \Delta t \cdot f(t_n + \alpha \Delta t, x_n + \beta K_1)$$

where α and β are parameters to be determined.

3. Update:

$$x_{n+1} = x_n + a K_1 + b K_2 \quad x_{n+1} = x_n + a K_1 + b K_2$$

where a and b are also parameters to be determined.

Goal: Determine α, β, a, b such that

The method is **second-order accurate**, i.e., error is proportional to $(\Delta t)^3$.

How to find constraints on the parameters?

Step 1: Taylor series expansion of the true solution

$$x(t_n + \Delta t) = x_n + \Delta t \cdot f(t_n, x_n) + \frac{\Delta t^2}{2} x''(t_n) + O(\Delta t^3) \quad x(t_n + \Delta t) = x_n + \Delta t \cdot f(t_n, x_n) + \frac{\Delta t^2}{2} x''(t_n) + O(\Delta t^3)$$

where $x''(t_n)$ is the second derivative with respect to time.

Using the chain rule:

$$x''(t_n) = \frac{d}{dt} f(t, x) = f_t + f_x \cdot \frac{dx}{dt} = f_t + f_x \cdot f(t_n, x_n) \quad x''(t_n) = \frac{d}{dt} f(t, x) = f_t + f_x \cdot f(t_n, x_n)$$

with partial derivatives $f_t = \frac{\partial f}{\partial t}$ and $f_x = \frac{\partial f}{\partial x}$.

Thus,

$$x(t_n + \Delta t) = x_n + \Delta t f(t_n, x_n) + \frac{\Delta t^2}{2} (f_t + f_x f) + O(\Delta t^3) \quad x(t_n + \Delta t) = x_n + \Delta t f(t_n, x_n) + \frac{\Delta t^2}{2} (f_t + f_x f) + O(\Delta t^3)$$

Step 2: Taylor series expansion of the RK2 method

$$x_{n+1} = x_n + a K_1 + b K_2 \quad x_{n+1} = x_n + a K_1 + b K_2$$

Expand the second term $f(t_n + \alpha \Delta t, x_n + \beta K_1)$ around (t_n, x_n) :

$$f(t_n + \alpha \Delta t, x_n + \beta K_1) = f(t_n, x_n) + \alpha \Delta t f_t + \beta K_1 f_x + O(\Delta t^2) \quad f(t_n + \alpha \Delta t, x_n + \beta K_1) = f(t_n, x_n) + \alpha \Delta t f_t + \beta K_1 f_x + O(\Delta t^2)$$

Since $K_1 = \Delta t f(t_n, x_n)$, we have:

$$f(t_n + \alpha \Delta t, x_n + \beta K_1) = f(t_n, x_n) + \alpha \Delta t f_t + \beta \Delta t^2 f_x f + O(\Delta t^3) \quad f(t_n + \alpha \Delta t, x_n + \beta K_1) = f(t_n, x_n) + \alpha \Delta t f_t + \beta \Delta t^2 f_x f + O(\Delta t^3)$$

Substitute back:

$$x_{n+1} = x_n + a\Delta t + b\Delta t(f + \alpha\Delta t f_t + \beta\Delta t f_x) + O(\Delta t^3) \quad x_{n+1} = x_n + a\Delta t + b\Delta t(f + \alpha\Delta t f_t + \beta\Delta t f_x) + O(\Delta t^3)$$

Collect terms:

$$x_{n+1} = x_n + (a+b)\Delta t f + b\alpha\Delta t^2 f_t + b\beta\Delta t^2 f_x + O(\Delta t^3) \quad x_{n+1} = x_n + (a+b)\Delta t f + b\alpha\Delta t^2 f_t + b\beta\Delta t^2 f_x + O(\Delta t^3)$$

Step 3: Equate coefficients with the true solution expansion

Match terms with the exact solution's Taylor series:

| Term | From exact solution | From RK2 method |
|-----------------------------|-----------------------------|-------------------------|
| $\Delta t f$ | f | $(a+b)f$ |
| $\frac{1}{2}\Delta t^2 f_t$ | $\frac{1}{2}\Delta t^2 f_t$ | $b\alpha\Delta t^2 f_t$ |
| $\frac{1}{2}\Delta t^2 f_x$ | $\frac{1}{2}\Delta t^2 f_x$ | $b\beta\Delta t^2 f_x$ |

This gives the system of equations:

$$\begin{cases} a+b=1 \\ b\alpha=\frac{1}{2} \\ b\beta=\frac{1}{2} \end{cases} \quad \begin{cases} a+b=1 \\ b\alpha=\frac{1}{2} \\ b\beta=\frac{1}{2} \end{cases}$$

Interpretation

- We have **four parameters** (a, b, α, β) but only **three constraints**.
- So there is **one degree of freedom**, meaning infinitely many RK2 methods.
- Different choices lead to famous RK2 variants.

Examples of RK2 methods satisfying these conditions:

| Method | α | β | a | b |
|-------------------------|---------------|---------------|---------------|---------------|
| Modified Euler (Heun's) | 1 | 1 | 0.5 | 0.5 |
| Midpoint method | 0.5 | 0.5 | 0 | 1 |
| Ralston's method | $\frac{2}{3}$ | $\frac{3}{4}$ | $\frac{1}{4}$ | $\frac{3}{4}$ |

Summary

- The **general RK2 method** is a weighted average of two slopes.
- The slopes are evaluated at:

- The beginning of the interval (t_n, x_n)
- A point inside the interval $(t_n + \alpha \Delta t, x_n + \beta k_1)$
- The weights a, b and parameters α, β satisfy constraints ensuring **second-order accuracy**.
- This creates a whole **family of methods** with the same order but slightly different properties.

If you want, I can help you:

- See a step-by-step algebra derivation from the Taylor expansions.
- Implement one or more of these RK2 methods in code.
- Compare the numerical performance of different RK2 variants on a test problem.

Would you like to explore any of those?

Great! Here's a concise summary of what you just described about deriving specific second-order Runge-Kutta methods from the general family:

Recap: General second-order Runge-Kutta method

- We solve $x' = f(t, x)$ from t_n to $t_{n+1} = t_n + \Delta t$
- Two stages:

$$k_1 = \Delta t \cdot f(t_n, x_n) \quad k_2 = \Delta t \cdot f(t_n + \alpha \Delta t, x_n + \beta k_1)$$

- The update step:

$$x_{n+1} = x_n + a k_1 + b k_2$$

- The parameters α, β, a, b satisfy these constraints for second-order accuracy:

$$a + b = 1, \quad \alpha b = \frac{1}{2}, \quad \beta b = \frac{1}{2} \quad \Rightarrow \quad a + b = 1, \quad \alpha b = \frac{1}{2}, \quad \beta b = \frac{1}{2}$$

1. Modified Euler method

- Known as the improved Euler or Heun's method.
- Parameters:

$$a = \frac{1}{2}, b = \frac{1}{2}, \alpha = 1, \beta = 1 \quad \Rightarrow \quad a = \frac{1}{2}, b = \frac{1}{2}, \alpha = 1, \beta = 1$$

- So:

$$k_1 = \Delta t \cdot f(t_n, x_n) \quad k_2 = \Delta t \cdot f(t_n + \Delta t, x_n + k_1)$$

- Update:

$$x_{n+1} = x_n + \frac{1}{2} (k_1 + k_2)$$

2. Midpoint method

- Evaluates the slope at the midpoint of the interval.
- Choose:

$$\alpha = \frac{1}{2}$$

- From constraints:

$$b = 1, \beta = \frac{1}{2}, a = 0$$

- So:

$$k_1 = \Delta t \cdot f(t_n, x_n) \quad k_2 = \Delta t \cdot f\left(t_n + \frac{\Delta t}{2}, x_n + \frac{1}{2} k_1\right)$$

- Update:

$$x_{n+1} = x_n + k_2$$

Summary

- Both methods are second-order accurate, but they differ in how they weigh the intermediate slopes.
 - Modified Euler averages slopes at the start and end of the interval.
 - Midpoint method uses the slope at the midpoint for the update.
 - You can generate other second-order Runge-Kutta methods by choosing one parameter and solving the constraints accordingly.
-

If you want, I can help you write out code implementations for these methods or explain how they compare in practice! Would you like that?

Here's a clear summary of the key points from your detailed explanation about higher-order Runge-Kutta methods, focusing especially on the third and fourth order methods:

Generalizing Runge-Kutta Methods Beyond Second Order

1. Setup for Third-Order Runge-Kutta (RK3)

- We solve $x' = f(t, x)$ from t_n to $t_{n+1} = t_n + \Delta t$.
- Three stages (three evaluations of f):

$$k_1 = \Delta t \cdot f(t_n, x_n) \quad k_2 = \Delta t \cdot f(t_n + \alpha \Delta t, x_n + \beta k_1) \quad k_3 = \Delta t \cdot f(t_n + \gamma \Delta t, x_n + \delta k_1 + \epsilon k_2)$$

- Update formula combines the three slopes:

$$x_{n+1} = x_n + \alpha k_1 + \beta k_2 + \gamma k_3$$

- Parameters $\alpha, \beta, \gamma, \delta, \epsilon, a, b, c$ must satisfy constraints derived via Taylor series expansions for third-order accuracy.
- Third-order methods require 3 stages (3 evaluations of f).

2. Number of Stages vs. Order

- **Second order:** 2 stages
- **Third order:** 3 stages
- **Fourth order:** 4 stages
- **Fifth order:** 6 stages (not 5 — it requires more)
- **Sixth order:** 7 stages
- **Seventh order:** 9 stages
- **Eighth order:** 11 stages

Higher-order methods quickly require more function evaluations, increasing computational cost.

3. The Popular Fourth-Order Runge-Kutta Method (RK4)

- Four stages, evaluating f at:
 1. The start of the interval: t_n
 2. The midpoint (twice): $t_n + \frac{\Delta t}{2}$
 3. The end of the interval: $t_n + \Delta t$
- Stages:

$$k_1 = \Delta t \cdot f(t_n, x_n) \quad k_2 = \Delta t \cdot f(t_n + \frac{\Delta t}{2}, x_n + \frac{k_1}{2}) \quad k_3 = \Delta t \cdot f(t_n + \frac{\Delta t}{2}, x_n + \frac{k_2}{2}) \quad k_4 = \Delta t \cdot f(t_n + \Delta t, x_n + k_3)$$

- Update:

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

- This method is fourth-order accurate (local error $\sim \Delta t^5$).
- Became very popular due to good balance of accuracy and computational cost.

- Still widely taught and used, though modern software like MATLAB often uses even higher-order adaptive methods.

4. Summary and Practical Notes

- RK4 is a "sweet spot" in accuracy and complexity.
- Deriving coefficients for third and higher-order RK methods requires careful Taylor series expansion and solving constraints.
- Higher-order methods need more stages, increasing computational effort.
- Software packages often handle the choice of method internally.

If you want, I can help you:

- Derive or explain the RK3 coefficients and formulas.
- Write sample code implementing RK3 or RK4.
- Compare RK4 with other integration methods or adaptive solvers.

Would you like to dive deeper into any of those?

Here's a clear summary and explanation of the video content about integrating **second order ODEs and systems of ODEs** using Runge-Kutta methods:

Integrating Second Order ODEs by Converting to First Order Systems

The Problem:

- You often encounter **second order ODEs**, e.g., Newton's law:

$$d^2x/dt^2 = f(t, x, \dot{x})$$

where the second derivative depends on time t , position x , and velocity \dot{x} .

The Key Trick: Define a New Variable

- Introduce a new variable for the first derivative of x :

$$u = dx/dt = \dot{x}$$

- Now rewrite the original second order ODE as a **system of two first order ODEs**:

$$\begin{cases} dx/dt = u \\ du/dt = f(t, x, u) \end{cases}$$

- This converts one second order ODE into two coupled first order ODEs.

Generalizing to Systems of Coupled First Order ODEs

Consider a system of two coupled first order ODEs:

$$\begin{cases} \frac{dx}{dt} = f(t, x, y) \\ \frac{dy}{dt} = g(t, x, y) \end{cases}$$

- You want to integrate **both simultaneously**.

Applying a Runge-Kutta Method (Modified Euler as example) to Systems

Parallel integration approach:

- For the variable x , define stages with k terms.
- For the variable y , define stages with l terms.

First stage:

$$k_1 = \Delta t \cdot f(t_n, x_n, y_n) \quad l_1 = \Delta t \cdot g(t_n, x_n, y_n)$$

Second stage:

$$k_2 = \Delta t \cdot f(t_n + \Delta t, x_n + k_1, y_n + l_1) \quad l_2 = \Delta t \cdot g(t_n + \Delta t, x_n + k_1, y_n + l_1)$$

Update step:

$$x_{n+1} = x_n + \frac{1}{2}(k_1 + k_2) \quad y_{n+1} = y_n + \frac{1}{2}(l_1 + l_2)$$

Summary:

- **Second order ODEs** can be transformed into **systems of first order ODEs** by introducing auxiliary variables.
- When integrating systems of coupled first order ODEs, Runge-Kutta methods are applied **in parallel** to each variable.
- Each variable has its own set of stages (k 's for x , l 's for y), but the updates depend on all variables because the derivatives are coupled.
- This approach extends naturally to larger systems or higher order equations converted to first order systems.

If you'd like, I can help you:

- Write example Python code for RK2 or RK4 integration of such systems.
- Show how to handle larger systems or include higher order methods.
- Explain how this applies to physics problems like Newton's equations or coupled oscillators.

Would you like to see that?

Here's a detailed explanation and summary of the video about **MATLAB's adaptive Runge-Kutta method for solving ODEs**, specifically focusing on the Dormand-Prince method used by ode45:

MATLAB's Adaptive Runge-Kutta ODE Solver (ode45)

The Problem with Fixed Time Step (Δt)

- Classical Runge-Kutta methods require you to specify a fixed time step Δt .
 - Choosing Δt is tricky:
 - Too large \rightarrow large errors (solution inaccurate)
 - Too small \rightarrow very slow, expensive computation
 - For complex or large systems, manual tuning of Δt is inefficient.
-

Adaptive Time Step and Error Control

- MATLAB uses an **adaptive time step** algorithm that automatically adjusts Δt to control error.
 - Instead of specifying Δt , you specify the **error tolerance** ϵ , which is the maximum allowed error per step.
 - Smaller tolerance \rightarrow smaller Δt \rightarrow more accurate but more expensive.
 - Larger tolerance \rightarrow larger Δt \rightarrow faster but less accurate.
-

Dormand-Prince Method (Used in ode45)

- It is a **six-stage Runge-Kutta method** that provides:
 - A **5th order accurate solution** (more accurate)
 - A **4th order accurate solution** (less accurate)
 - Both solutions are computed **simultaneously** from the same six derivative evaluations.
 - This allows estimating the **local error** without extra derivative calculations.
-

How Error is Estimated

- Let:
 - x_{n+1} = 5th order solution (more accurate, error $\sim \Delta t^6$)
 - X_{n+1} = 4th order solution (less accurate, error $\sim \Delta t^5$)

- The difference between these two solutions approximates the **actual error** e_e of the step:

$$e_e \approx |x_{n+1} - X_{n+1}| \approx |x_{n+1} - X_{n+1}|$$

- Since the 4th order solution has larger error, the difference is dominated by Δt^5 , giving a reliable error estimate.

Adjusting the Time Step

- Suppose you did a step with time step Δt , got error estimate e_e , but want error less than tolerance ϵ .
- The error scales like Δt^5 , so:

$$e_e = (\Delta t / \Delta \tau)^5 \epsilon \Rightarrow \left(\frac{\Delta t}{\Delta \tau} \right)^5 = \frac{e_e}{\epsilon}$$

where $\Delta \tau$ is the *ideal* time step that would yield error ϵ .

- Solving for $\Delta \tau$:

$$\Delta \tau = (\epsilon / e_e)^{1/5} \Delta t$$

- Interpretation:
 - If $e_e > \epsilon$, error too large \rightarrow reduce time step \rightarrow reject current solution and redo with smaller $\Delta t = \Delta \tau$.
 - If $e_e < \epsilon$, error acceptable \rightarrow accept solution and maybe increase time step for next step.

Safety Factor

- To avoid too frequent rejections and unstable step sizes, multiply by a **safety factor** $s < 1$, typically $s = 0.9$.
- New time step formula becomes:

$$\Delta \tau = s \times (\epsilon / e_e)^{1/5} \Delta t = s \times \Delta t \times (\epsilon / e_e)^{1/5}$$

- This makes the algorithm more robust.

Summary of MATLAB ode45 Workflow

1. User specifies error tolerance ϵ , **not** Δt .
2. MATLAB tries a step with some Δt .
3. Computes both 4th and 5th order solutions and estimates error e_e .
4. If $e_e > \epsilon$, rejects step, reduces Δt , retries.
5. If $e_e \leq \epsilon$, accepts step, adjusts Δt for next step.
6. Process repeats until integration completes.

Why This is Important

- Automatically balances accuracy and efficiency.
 - User doesn't have to guess the time step.
 - Ensures error stays below the specified tolerance.
 - Crucial for solving stiff or complex differential equations.
-

If you want, I can also help you:

- Show example MATLAB code using ode45 and setting tolerances.
- Explain how to interpret solver output and tweak tolerances.
- Compare with fixed time step methods in code.

Would you like that?

Got it! You shared a detailed walkthrough of how to solve ODEs in MATLAB using ode45, covering:

1. Single first-order ODE (Logistic equation):

$$\frac{dx}{dt} = rx(1-x)$$

- Define the right-hand side as an anonymous function.
- Set parameter r , initial condition x_0 , and time span.
- Call `ode45(@(t,x) r*x*(1-x), tspan, x0)`.
- Plot the solution $x(t)$.

2. System of two first-order ODEs (Lotka-Volterra predator-prey model):

$$\begin{cases} \frac{dx}{dt} = rx(1-y) \\ \frac{dy}{dt} = \frac{1}{r}y(x-1) \end{cases}$$

- Define a sub-function that returns a two-component column vector for the derivatives.
- Set parameters and initial conditions for both x and y .
- Call `ode45(@(t,xy) lv(xy, r), tspan, [x0; y0])`.
- Plot both $x(t)$ and $y(t)$, and also y vs x phase plot.

3. Notes on accuracy and smoothness:

- The default solver steps and interpolation might make plots appear less smooth.
- Over longer integration times, numerical errors accumulate, affecting periodicity and plot overlap.
- You can improve accuracy with solver options (not covered yet).

If you want, I can help you write a clean MATLAB script based on this, explain how to add solver options to improve accuracy, or help with any other part of this process. Would you like me to?

Here's a concise summary of how to **improve accuracy and smoothness** when solving ODEs with ode45 in MATLAB, based on the video:

Problem 1: Plots Are Not Smooth

- **Cause:** ode45 returns solution at a relatively small number of time points (e.g., 57 points), leading to coarse plots.
 - **Fix:** Instead of specifying `tspan = [t0 tf]`, specify `tspan` as a vector with many points using `linspace`.
 - `tspan = linspace(0, 6*pi, 500); % 500 points instead of just 2`
 - `[t, xy] = ode45(@(t,xy) lv(xy,r), tspan, [x0; y0]);`
 - This makes MATLAB interpolate the solution at the requested finer time points, giving a smoother plot.
-

Problem 2: Solution Is Not Exactly Periodic (Accuracy Issues)

- **Cause:** Default relative tolerance of ode45 is 10^{-3} , which might not be precise enough for subtle periodicity in the solution.
 - **Fix:** Use `odeset` to create options with tighter relative tolerance, e.g., 10^{-4} or smaller:
 - `options = odeset('RelTol', 1e-4);`
 - `[t, xy] = ode45(@(t,xy) lv(xy,r), tspan, [x0; y0], options);`
 - This improves the numerical accuracy, reducing cumulative errors and ensuring periodicity appears correctly.
-

Summary MATLAB example incorporating both fixes:

```
r = 1;
```

```
x0 = 1.1; y0 = 1;
```

```
tspan = linspace(0, 6*pi, 500); % many points for smooth plotting
```

```
options = odeset('RelTol', 1e-4); % tighter tolerance for accuracy
```

```
[t, xy] = ode45(@(t,xy) lv(xy, r), tspan, [x0; y0], options);
```



```
% Plot solutions

figure;

plot(t, xy(:,1), t, xy(:,2));

xlabel('Time t');

ylabel('Population');

legend('Prey (x)', 'Predator (y)');

title('Lotka-Volterra Solution');


figure;

plot(xy(:,1), xy(:,2));

xlabel('Prey (x)');

ylabel('Predator (y)');

title('Phase plot of Lotka-Volterra');
```

Additional Notes:

- `lv` function defines the system of ODEs as before.
 - Increasing the number of output points smooths plots by forcing interpolation at those points.
 - Reducing the error tolerances makes the numerical solver more accurate but might increase computation time.
-

Would you like me to help you implement these improvements into your full MATLAB code or explain how to tune other `ode45` options?

Here's a clear explanation of the **shooting method** for solving two-point boundary value problems (BVPs) as described in the video:

What is a Boundary Value Problem (BVP)?

- Given a **second-order ODE** like

$$d^2y/dx^2 = f(x, y, dy/dx) \quad \frac{d^2 y}{dx^2} = f(x, y, \frac{dy}{dx})$$

- Boundary conditions specify values of y at **two points**:

$$y(x_0) = y_0, y(x_f) = y_f \quad y(x_0) = y_0, \quad y(x_f) = y_f$$

- Goal: find a function $y(x)$ that satisfies the ODE and these two boundary conditions.

Why is BVP Different from Initial Value Problem (IVP)?

- To solve a second order ODE numerically with methods like ode45, we rewrite as a system of first order ODEs:

$$\frac{dy}{dx} = z, \frac{dz}{dx} = f(x, y, z) \quad \frac{dy}{dx} = z, \quad \frac{dz}{dx} = f(x, y, z)$$

- For IVP, we know both $y(x_0)$ and $z(x_0) = \frac{dy}{dx}(x_0)$ (initial slope).
- For BVP, $y(x_0) = y_0$ is known, **but $z(x_0)$ is unknown**. We call this unknown initial slope parameter α (or *kosai* in the video).

Shooting Method Idea

- The problem becomes: **Find α such that when we solve the ODE from x_0 to x_f with initial slope α , the solution satisfies $y(x_f) = y_f$.**
- Define a function

$$F(\alpha) = y_\alpha(x_f) - y_f$$

where $y_\alpha(x_f)$ is the solution at x_f after integrating with initial slope α .

- We want to solve for

$$F(\alpha) = 0$$

This is a **root-finding problem** for α .

How the Shooting Method Works

1. **Guess α :** Pick an initial guess for the unknown initial slope $z(x_0) = \alpha$.
2. **Integrate ODE:** Solve the ODE IVP with $y(x_0) = y_0$, $z(x_0) = \alpha$ using ode45.
3. **Check boundary condition at x_f :** Calculate $F(\alpha) = y_\alpha(x_f) - y_f$.
4. **Update α :** Use a root-finding method (like fzero in MATLAB) to find a better α based on the sign and value of $F(\alpha)$.
5. **Repeat:** Continue guessing and integrating until $F(\alpha)$ is close enough to zero, i.e., the boundary condition at x_f is satisfied.

Why Is It Called "Shooting"?

- Imagine the unknown slope α as the angle you "aim" a gun at x_0 .

- Solving the ODE with this slope is like “firing” the bullet along a trajectory (the solution curve).
- Your goal is to “hit the target” y_f at x_f .
- If you miss, adjust the angle α and shoot again.
- Repeat until you hit the target.

Summary in MATLAB Context

- Use ode45 inside a function $F(\alpha)$ that:
 - Integrates from x_0 to x_f with initial slope α ,
 - Returns the difference $y(x_f) - y_f$.
- Use MATLAB’s fzero or other root-finders to find α such that $F(\alpha) = 0$.

Would you like me to help you with a MATLAB code example implementing the shooting method for a sample BVP?

Here’s a clear summary of the key points from the video on the **two-body problem** and its reduction to a one-body problem for numerical solution:

Two-Body Problem Setup

- You have two masses, m_1 and m_2 , at positions \mathbf{r}_1 and \mathbf{r}_2 .
- Newton's second law for each mass:

$$m_1 \ddot{\mathbf{r}}_1 = \mathbf{F}, m_2 \ddot{\mathbf{r}}_2 = -\mathbf{F} \quad \text{where } \mathbf{F} = -\frac{G m_1 m_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3} (\mathbf{r}_1 - \mathbf{r}_2)$$

- The gravitational force \mathbf{F} follows Newton’s universal law of gravitation:

$$\mathbf{F} = -\frac{G m_1 m_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3} (\mathbf{r}_1 - \mathbf{r}_2)$$

- Force acts along the vector between the two masses with an inverse-square law.

Coordinate Transformation: From Two-Body to One-Body

- Define the **relative position vector**:

$$\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2$$

- Define the **center of mass (COM) position vector**:

$$\mathbf{R} = \frac{m_1 \mathbf{r}_1 + m_2 \mathbf{r}_2}{m_1 + m_2}$$

- Using \mathbf{r} and \mathbf{R} , the system can be rewritten:

- The acceleration of the center of mass is zero:

$$\ddot{\mathbf{R}} = 0$$

Meaning the COM moves uniformly or can be fixed by choice of reference frame.

- The relative vector \mathbf{r} satisfies:

$$\mu \ddot{\mathbf{r}} = \mathbf{F}$$

where μ is the **reduced mass**:

$$\mu = \frac{m_1 m_2}{m_1 + m_2}$$

- This reduces the two-body problem to a single particle with mass μ moving under the central force \mathbf{F} .

Simplified Equation for Numerical Solution

- Writing the equation explicitly:

$$\ddot{\mathbf{r}} = -k \frac{\mathbf{r}}{r^3} = -k \frac{\mathbf{r}}{r^3}$$

where

$$k = G(m_1 + m_2)$$

- By non-dimensionalizing, you can scale k to 1 for easier computation.

Recovering Original Positions

- After solving for $\mathbf{r}(t)$, you find the original positions by:

$$\mathbf{r}_1 = \frac{m_2}{m_1 + m_2} \mathbf{r}, \quad \mathbf{r}_2 = -\frac{m_1}{m_1 + m_2} \mathbf{r}$$

Summary

- The two-body problem reduces to solving one second-order ODE for the relative vector \mathbf{r} .
- The center of mass moves with zero acceleration and can be treated separately.
- After solving $\mathbf{r}(t)$, original trajectories $\mathbf{r}_1(t)$ and $\mathbf{r}_2(t)$ follow.
- The numerical solution to the reduced problem lets you plot the orbits of the two masses under gravity.

If you want, I can help you set up the numerical solution in code next, or clarify any of these steps!

Here's a structured summary and explanation of the continuation for numerically solving the two-body problem, based on the video transcript you shared:

Setting Up the Numerical Problem in the Center of Mass Frame

1. Fixing the Center of Mass (COM) Reference Frame

- Since the **acceleration of the center of mass \mathbf{R} is zero** ($\ddot{\mathbf{R}} = 0$), the center of mass moves with **constant velocity**.
- Choose a coordinate system moving with this velocity so that:

$$\dot{\mathbf{R}} = 0, \mathbf{R} = 0 \Rightarrow \dot{\mathbf{R}} = 0, \quad \mathbf{R} = 0$$

- This means the center of mass is fixed at the origin for the numerical solution.
-

2. Reducing to 2D Motion

- If the initial positions $\mathbf{r}_1, \mathbf{r}_2$ and velocity vectors lie in the **xy-plane**, the motion stays in the xy-plane due to symmetry.
 - So, we can **set $z = 0$** and solve the problem in two dimensions only.
-

3. Non-Dimensionalization and Choosing Units

- Use the gravitational constant and masses combined into a constant k .
 - Choose units so that:
 - $k = 1$
 - The **minimum distance between the two masses** is 1 unit.
 - This simplification (non-dimensionalization) simplifies the numerical solution by reducing constants.
-

4. Coordinate System Orientation and Initial Conditions

- Define the coordinate system so that:
 - At $t = 0$, the relative position vector is at its **minimum distance** along the **x-axis**:

$$x(0) = -1, y(0) = 0 \Rightarrow x(0) = -1, \quad y(0) = 0$$

- The velocity at $t = 0$ is **perpendicular** to the position vector and directed along the y-axis:

$$\dot{x}(0) = 0, \dot{y}(0) = 1 \Rightarrow \dot{x}(0) = 0, \quad \dot{y}(0) = 1$$

- Here, **e is the eccentricity** of the orbit:
 - $e = 0 \rightarrow$ circular orbit,

- $0 < e < 1 \rightarrow$ elliptical orbit.

5. Final Equations to Solve Numerically

The second-order coupled ODE system for the relative position $\mathbf{r} = (x, y)$:

$$\ddot{x} = -\frac{x}{(x^2 + y^2)^{3/2}}, \quad \ddot{y} = -\frac{y}{(x^2 + y^2)^{3/2}}$$

This can be rewritten as **four first-order ODEs**:

$$\begin{cases} \dot{x} = v_x \\ \dot{v}_x = -\frac{x}{(x^2 + y^2)^{3/2}} \\ \dot{y} = v_y \\ \dot{v}_y = -\frac{y}{(x^2 + y^2)^{3/2}} \end{cases}$$

6. Initial Conditions for the ODE System

$$x(0) = -1, y(0) = 0, v_x(0) = 0, v_y(0) = 1 \Rightarrow \mathbf{r}(0) = (-1, 0), \quad \mathbf{v}(0) = (0, 1)$$

7. Orbit Period

- The period T of the orbit depends on e by:

$$T = 2\pi(1 - e)^{3/2}$$

- You can use this T to set how long to run the numerical solver to complete one full orbit.

8. Recovering Original Mass Positions

After solving for $x(t), y(t)$, you can find the positions of the two masses:

$$\begin{aligned} \mathbf{r}_1 &= \frac{m_2}{m_1 + m_2} \mathbf{r} \\ \mathbf{r}_2 &= -\frac{m_1}{m_1 + m_2} \mathbf{r} \end{aligned}$$

Summary

- Use the COM frame to fix $\mathbf{R} = 0$.
- Solve the coupled ODEs for (x, y) with given initial conditions related to orbit eccentricity e .
- Translate the solution back to positions of the two bodies.
- Plot trajectories for visualization.

If you want, I can help you write MATLAB code to implement this numerical solver with these initial conditions and equations! Would you like that?

Nice overview! Here's a quick summary of what you'll cover in Week 6 of Numerical Methods for Engineers, focusing on PDEs and how to solve them computationally:

Week 6: Introduction to Solving Partial Differential Equations (PDEs)

1. Importance of PDEs

- PDEs describe many real-world phenomena (fluid mechanics, electromagnetism, chemistry, heat flow, etc.).
- Week 6 focuses on **two key PDEs** and their numerical solution.

2. Two-Dimensional Laplace Equation

- This is a **boundary value problem (BVP)**:
 - The solution inside a domain depends **only on boundary values**.
- Use the **finite difference method (FDM)** to discretize the equation:
 - This converts the PDE into a system of algebraic equations.
- Leads to a **matrix equation**:
 - Natural ordering of points forms a linear system $\mathbf{A}\mathbf{u}=\mathbf{b}$ $\mathbf{u} = \mathbf{b}$.
 - Can solve this system directly in MATLAB using the backslash operator (`\`).
- Alternatively, solve it iteratively via:
 - Jacobi method
 - Gauss-Seidel method
 - Successive Overrelaxation (SOR) method
- Iterative methods are convenient to code with while loops.

3. One-Dimensional Diffusion Equation

- This is an **initial value problem (IVP)**:
 - You start with initial conditions and **evolve the solution over time**.
- Numerical methods:
 - **Explicit methods** (forward in time):
 - Simple but can be unstable (depends on timestep and grid size).
 - **Implicit methods** (backward in time):

- More stable, often require solving linear systems at each timestep.
 - **Von Neumann stability analysis:**
 - Used to analyze the stability of explicit methods.
 - The **Crank-Nicolson method:**
 - A popular implicit method.
 - Second-order accurate in time.
 - Combines accuracy and stability.
 - Implement solutions in MATLAB.
-

4. Final Project

- Solve the **two-dimensional diffusion equation** using:
 - The Crank-Nicolson method
 - Matrix algebra techniques
 - This project combines methods learned from both Laplace and diffusion equation solving.
-

If you'd like, I can help you understand the finite difference discretization, write MATLAB code for Jacobi/Gauss-Seidel or Crank-Nicolson methods, or explain Von Neumann stability analysis in detail. Just say the word!

Great explanation! Here's a clear summary to help you organize the main ideas from the video about **types of PDE problems** and their numerical solutions:

Understanding PDEs: Boundary Value Problems vs. Initial Value Problems

Why PDEs are harder than ODEs:

- One PDE corresponds to infinitely many ODEs.
 - So, solving PDEs numerically requires more care and computational effort.
-

1. Boundary Value Problem (BVP)

- Example: **Laplace equation**.
- You have a fixed spatial domain (e.g., a 2D grid in x and y).
- The **boundary values** of the solution (function values at edges of the domain) are known and given.
- The goal is to find the values of the solution **at interior points**.

- This means solving for the entire spatial field all at once.
- Boundary conditions specify the function value (or derivatives, but in this course, function values) at the edges.
- Numerical methods find the solution by solving a system over all interior points simultaneously.

2. Initial Value Problem (IVP)

- Example: **Diffusion equation**.
- The domain involves space (e.g., x) **and time (t)**.
- You specify initial conditions: the value of the solution at **$t = 0$** across space.
- You also have boundary conditions in space (e.g., at smallest and largest x).
- The solution evolves **forward in time**.
- You compute the solution step-by-step: first at the next time point, then the next, etc.
- Unlike BVPs, you find solutions one time slice at a time rather than all points at once.

Key Differences:

| Aspect | Boundary Value Problem | Initial Value Problem |
|-------------------|--|---|
| Domain | Spatial only (fixed grid) | Space + Time |
| Known conditions | Boundary values at domain edges | Initial values at time zero + boundaries in space |
| Solution approach | Solve for all interior points simultaneously | Step forward in time, solving sequentially |
| Example | Laplace equation | Diffusion equation |

If you'd like, I can help you write MATLAB code for either problem type, explain how to implement boundary or initial conditions, or work through specific examples of discretization methods. Just let me know!

This is a great explanation of how finite difference approximations for derivatives are derived using Taylor series expansions! Here's a concise summary to help organize the key points:

Finite Difference Method: Central Difference Approximations

Setup:

- Consider three points along the x -axis: $x-h$, x , and $x+h$.

- The function values at these points are $y(x-h)y(x-h)$, $y(x)y(x)$, and $y(x+h)y(x+h)$.
 - Goal: Approximate derivatives of y at x using these function values.
-

1. Taylor Series Expansions

- Around x , expand $y(x+h)y(x+h)$ and $y(x-h)y(x-h)$:

$$y(x+h) = y(x) + hy'(x) + \frac{h^2}{2}y''(x) + \frac{h^3}{6}y^{(3)}(x) + \frac{h^4}{24}y^{(4)}(x) + \dots$$

$$y(x-h) = y(x) - hy'(x) + \frac{h^2}{2}y''(x) - \frac{h^3}{6}y^{(3)}(x) + \frac{h^4}{24}y^{(4)}(x) - \dots$$

2. First Derivative Approximation (Central Difference)

- Subtract $y(x-h)y(x-h)$ from $y(x+h)y(x+h)$:

$$y(x+h) - y(x-h) = 2hy'(x) + \frac{h^3}{6}y^{(3)}(x) + \dots$$

- Solve for $y'(x)$:

$$y'(x) \approx \frac{y(x+h) - y(x-h)}{2h}$$

- **Error term:** $O(h^2)$, meaning the approximation error scales as h^2 .
-

3. Second Derivative Approximation (Central Difference)

- Add $y(x+h)y(x+h)$ and $y(x-h)y(x-h)$:

$$y(x+h) + y(x-h) = 2y(x) + h^2y''(x) + \frac{h^4}{12}y^{(4)}(x) + \dots$$

- Solve for $y''(x)$:

$$y''(x) \approx \frac{y(x+h) - 2y(x) + y(x-h)}{h^2}$$

- **Error term:** $O(h^2)$ again, indicating a good accuracy.
-

Why This Matters

- The **central difference formulas** give second-order accurate approximations to derivatives, better than forward or backward difference formulas which are typically first-order.
 - This means you can use coarser grids (larger h) while keeping reasonable accuracy.
 - These approximations are the key step in turning **partial differential equations (PDEs)** into **discrete algebraic equations** that can be solved numerically.
-

If you want, I can help you implement these approximations in MATLAB or Python for some example PDEs! Just say the word.

Here's a clear summary of the key steps in discretizing the 2D Laplace equation on a rectangular grid using finite differences, based on your video transcript:

Solving the 2D Laplace Equation Using Finite Differences

1. Problem Setup:

- Solve for $\Phi(x,y)$ inside a 2D domain $[0,L_x] \times [0,L_y]$.
- Boundary values of Φ are known (Dirichlet conditions).
- Goal: Find Φ at the interior grid points.

2. Grid Construction:

- Define grid points along x:

$$x_i = (i-1)\Delta x, i=1,2,\dots,n_x \quad i = 1, 2, \dots, n_x$$

- Define grid points along y:

$$y_j = (j-1)\Delta y, j=1,2,\dots,n_y \quad j = 1, 2, \dots, n_y$$

- Typically take $\Delta x = \Delta y$ for simplicity (square grid).
- Φ values on the grid are denoted as $\Phi_{i,j} = \Phi(x_i, y_j)$.

3. Discretizing the Laplace Equation:

The Laplace equation:

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0$$

Use central difference approximations for second derivatives:

$$\frac{\partial^2 \Phi}{\partial x^2} \approx \frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{\Delta x^2} \quad \frac{\partial^2 \Phi}{\partial y^2} \approx \frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{\Delta y^2}$$

4. Combine and Simplify:

Assuming $\Delta x = \Delta y$, multiply through by Δx^2 :

$$\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1} - 4\Phi_{i,j} = 0$$

Or equivalently:

$$4\Phi_{i,j} = \Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}$$

5. Interpretation:

- At each interior grid point, the value $\Phi_{i,j}$ is the average of its four nearest neighbors.
 - This discrete equation forms a system of linear equations for all interior points.
 - The boundary values Φ are known and fixed.
-

6. Next Steps:

- Assemble all these equations for interior points into a matrix equation.
 - Solve using direct methods (e.g., matrix backslash in MATLAB) or iterative methods (Jacobi, Gauss-Seidel, SOR).
 - This will give Φ values inside the domain consistent with boundary conditions.
-

If you'd like, I can help you with MATLAB or Python code for setting up and solving this system! Would you want me to do that?

Great! Here's a detailed summary of what you just described about turning the discretized Laplace equation into a matrix system suitable for direct linear algebra solution (like in MATLAB):

From the Discrete Laplace Equation to a Matrix Equation

1. Discrete Laplace Equation Recap:

For interior points, the equation is:

$$4\Phi_{i,j} - \Phi_{i+1,j} - \Phi_{i-1,j} - \Phi_{i,j+1} - \Phi_{i,j-1} = 0$$

2. Need for Vectorization:

- Φ is naturally arranged as a 2D array over indices (i,j) .
- To solve with matrix methods, we reshape Φ into a **1D column vector**.
- The matrix equation form:

$$A\Phi = b$$

where:

- A is the coefficient matrix,
- Φ is the unknown vector of interior Φ values,
- b includes known boundary terms.

3. Natural Ordering of Grid Points:

- The ordering goes **row-wise** starting from the bottom row ($j=1$), moving left to right across $i=1, 2, \dots, n_x$.
- Then move to the next row $j=2$, again left to right.
- Continue up to the top row $j=n_y$.

Example ordering for Φ values:

$\Phi_{1,1}, \Phi_{2,1}, \dots, \Phi_{n_x,1}, \Phi_{1,2}, \Phi_{2,2}, \dots, \Phi_{n_x,2}, \dots, \Phi_{n_x,n_y}$
 $\Phi_{1,1}, \Phi_{2,1}, \dots, \Phi_{n_x,1}, \Phi_{1,2}, \Phi_{2,2}, \dots, \Phi_{n_x,2}, \dots, \Phi_{n_x,n_y}$

4. Mapping 2D Indices (i,j) to 1D Index k :

$$k = i + (j-1) \times n_x$$

- k runs from 1 to $n_x \times n_y$.
- So the 2D grid point (i,j) corresponds to vector element Φ_k .

5. Rewrite Discrete Equation Using k :

Substitute $i,j \rightarrow k$ to:

$$4\Phi_k - \Phi_{k+1} - \Phi_{k-1} - \Phi_{k+n_x} - \Phi_{k-n_x} = 0$$

- Φ_{k+1} and Φ_{k-1} correspond to neighbors in the same row (ii direction).
- Φ_{k+n_x} and Φ_{k-n_x} correspond to neighbors in the rows above and below (jj direction).

6. Matrix AA Structure:

- The matrix AA will be **sparse** with a block structure.
- The diagonal entries are 4.
- The entries -1 appear immediately to the left and right of the diagonal (for neighbors in the same row).
- The entries -1 appear n_x positions away on the diagonals above and below the main diagonal (for neighbors in adjacent rows).
- AA is symmetric and banded.

7. Incorporating Boundary Conditions:

- Boundary values Φ at edges (where $i=1$, $i=n_x$, $j=1$, $j=n_y$) are known.
- These known values contribute to the right-hand side vector \mathbf{b} .
- For example, if a neighbor is on the boundary, move its Φ value to \mathbf{b} .
- Unknowns correspond only to interior points.

8. Summary of the Process:

1. Discretize PDE to difference equations on 2D grid.
2. Convert 2D indices (i,j) to 1D index k (natural ordering).
3. Write the difference equations in terms of vector Φ_k .
4. Construct matrix A reflecting the Laplace stencil and ordering.
5. Form right side \mathbf{b} from boundary conditions.
6. Solve $A\Phi = \mathbf{b}$ with a linear solver.

If you want, I can help you **write MATLAB or Python code** that builds matrix A and vector \mathbf{b} based on these steps and solves the Laplace equation numerically. Would you like me to do that?

Sure! Here's a clear explanation and step-by-step outline of the approach Jeff Chasnov explained for solving the 2D Laplace equation on a small grid, which you can generalize to larger grids:

Solving the 2D Laplace Equation via Matrix Methods on a Grid

Problem Setup:

- Solve $\nabla^2 \Phi = 0$ on a 2D domain discretized into a grid.
- Grid size: $n_x \times n_y$ points.
- Use **finite difference** approximation for derivatives.
- Use **natural ordering** to convert $\Phi(i,j)$ (2D array) into a vector Φ_k (1D).

Natural Ordering:

- Index grid points row-wise from bottom to top:

$\Phi_1 = \Phi_{1,1}, \Phi_2 = \Phi_{2,1}, \dots, \Phi_{n_x,1}, \Phi_{n_x+1} = \Phi_{1,2}, \dots, \Phi_{n_x n_y} = \Phi_{n_x, n_y}$
 $\Phi_1 = \Phi_{1,1}, \Phi_2 = \Phi_{2,1}, \dots, \Phi_{n_x,1}, \Phi_{n_x+1} = \Phi_{1,2}, \dots, \Phi_{n_x n_y} = \Phi_{n_x, n_y}$

- Total number of points: $N = n_x \times n_y$.

Discrete Laplace Equation in Vector Form:

For interior points (ignoring boundaries for now), the equation is:

$$4\Phi_k - \Phi_{k-1} - \Phi_{k+1} - \Phi_{k-n_x} - \Phi_{k+n_x} = 0 \quad \Phi_k - \Phi_{k-1} - \Phi_{k+1} - \Phi_{k-n_x} - \Phi_{k+n_x} = 0$$

- Here $k-1, k+1$ correspond to neighbors in the same row.
- $k-n_x, k+n_x$ correspond to neighbors one row below and one row above, respectively.
- Note: Boundary points do **not** use this equation; their values are given by boundary conditions.

Example: Small 3×3 Grid

- Number of points: $N=9$.
- Points are ordered as:

1, 2, 3, 4, 5, 6, 7, 8, 9

- Interior point: only $k=5$ (center point).
- Boundary points: $k=1, 2, 3, 4, 6, 7, 8, 9$.

Matrix System Setup:

We want:

$$A\Phi = b$$

- A : 9×9 matrix.
- Φ : 9×1 vector of unknowns.
- b : 9×1 right-hand side vector.

Boundary Conditions Enforcement:

- For boundary points $k \neq 5$, $\Phi_k = b_k$ (known values).
- Implement this by making corresponding rows in A **identity rows**, i.e.:

$A_{kk} = 1$, and zeros elsewhere in row k

- The right side b_k is set to the known boundary value.

Interior Point Equation (Row 5):

The discrete Laplace equation at $k=5$:

$$4\Phi_5 - \Phi_4 - \Phi_6 - \Phi_2 - \Phi_8 = 0 \quad \Phi_5 - \Phi_4 - \Phi_6 - \Phi_2 - \Phi_8 = 0$$

- Translate into matrix row:

Column index 1 2 3 4 5 6 7 8 9

Matrix row 5 0 -1 0 -1 4 -1 0 -1 0

- Right side $b_5 = 0$ since interior Laplace equals zero with no source.

Complete Matrix AA (Rows 1-9):

- Rows for boundary points: identity rows
- Row 5: Laplace stencil as above

$A = [1000000000100000000010000000001000000-10-14-10-100000010000000001000000001000000001]A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$

Right-hand Side Vector bb:

$b = [b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9]b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ 0 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{bmatrix}$

where b_k are known boundary values.

Generalization for Large Grids:

- Matrix size: $N \times N$ with $N = n_x \times n_y$.
- Matrix AA is sparse with five diagonals:
 - Main diagonal with 4's.
 - Two diagonals just above and below with -1 (for horizontal neighbors).
 - Two diagonals n_x above and below main diagonal with -1 (for vertical neighbors).
- Boundary rows replaced with identity rows for boundary conditions.

MATLAB Implementation Note:

- Use `spdiags` to construct sparse matrix with five diagonals.
- Then replace boundary rows by rows of the identity matrix.
- Solve $A\Phi = b$ with sparse solvers.

If you want, I can help write code snippets for MATLAB or Python to build this matrix and solve the system! Would you like that?

Here's a concise summary of the video tutorial you shared on solving the 2D Laplace equation in MATLAB using the direct method:

2D Laplace Equation in MATLAB — Direct Method Overview

Problem setup:

- Solve Laplace's equation on a unit square domain.
 - Boundary conditions:
 - $\Phi = 0$ on bottom and side edges.
 - $\Phi = 4x(1 - x)$ on the top edge (creates a maximum $\Phi = 1$ at $x = 0.5$).
-

Step-by-step outline:

1. **Discretization:**
 - Use central difference approximation for the Laplace equation.
 - Define a grid with $N_x \times N_y$ intervals (e.g., 100x100).
 - Grid points are $(N_x+1) \times (N_y+1)$.
2. **Natural ordering:**
 - Flatten 2D grid into a vector using column-wise ordering (MATLAB style).
 - Identify boundary indices for bottom, top, left, and right edges.
3. **Construct Laplacian matrix A:**
 - Banded sparse matrix with 5 diagonals:
 - Main diagonal = 4
 - Two diagonals immediately above and below = -1
 - Two diagonals offset by N_x above and below = -1
 - Use `spdiags` for memory-efficient sparse matrix.
4. **Apply boundary conditions:**
 - Replace rows in A corresponding to boundary points with identity matrix rows (`speye`).
 - This enforces fixed Φ values on boundaries.
5. **Right-hand side vector b:**

- Initialize zero matrix b (size $N_x \times N_y$).
- Assign boundary values to b accordingly.
- Reshape b into a vector consistent with natural ordering.

6. Solve the linear system:

- Use MATLAB's backslash operator: $\phi = A \setminus b$;
- This step uses Gaussian elimination optimized for sparse matrices.

7. Reshape and plot:

- Reshape ϕ vector back to matrix form.
- Use `meshgrid` to create X and Y matrices for plotting.
- Plot contours with `contour(X, Y, phi')` (transpose needed due to MATLAB matrix indexing).
- Customize contour levels, axis scaling (axis equal), ticks, and fonts.

Notes:

- MATLAB goes column-wise, so physics coordinates (x horizontal, y vertical) and matrix indices (row first, column second) have a mismatch, handled by transposing ϕ before plotting.
- The direct solve (backslash operator) is computationally heavy but straightforward.
- Plotting code often takes more time to get visually right than the numerical solution code.
- MATLAB environment features (breakpoints, workspace variables) are helpful for debugging.

If you want, I can help you write the MATLAB code based on this or explain any step in more detail! Would you like that?

Sure! Here's a clear summary and explanation of the lecture on iterative methods for solving the discrete Laplace equation, based on the transcript you provided:

Overview:

The lecture introduces **iterative methods** for solving the discrete Laplace equation, contrasting them with the direct method like Gaussian elimination.

The Problem Setup:

- We want to solve the Laplace equation $\nabla^2 \Phi = 0$
- Using finite differences, the discrete Laplace equation at grid point (i,j) is:

$$\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1} - 4\Phi_{i,j} = 0. \quad \Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1} - 4\Phi_{i,j} = 0.$$

Iterative Methods:

Instead of solving the linear system $A\Phi = b$ directly (e.g., by Gaussian elimination), **iterative methods** update guesses for Φ repeatedly until convergence.

Jacobi Method:

- Rearrange the discrete equation to solve for $\Phi_{i,j}$:

$$\Phi_{i,j} = \frac{1}{4}(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}). \quad \Phi_{i,j} = \frac{1}{4}(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}).$$

- Define iteration index n , then update:

$$\Phi_{i,j}(n+1) = \frac{1}{4}(\Phi_{i+1,j}(n) + \Phi_{i-1,j}(n) + \Phi_{i,j+1}(n) + \Phi_{i,j-1}(n)). \quad \Phi_{i,j}^{(n+1)} = \frac{1}{4}(\Phi_{i+1,j}^{(n)} + \Phi_{i-1,j}^{(n)} + \Phi_{i,j+1}^{(n)} + \Phi_{i,j-1}^{(n)}).$$

- Steps:
 - Initialize $\Phi(0)$, e.g., all zeros.
 - Keep boundary values fixed.
 - Update interior $\Phi_{i,j}$ using above formula.
 - Repeat until $\Phi(n)$ converges.
-

Gauss-Seidel Method:

- Similar to Jacobi but updates points **in place** as you sweep through the grid.
 - When updating $\Phi_{i,j}$, use the most recent values:
 - Values from the current iteration for indices less than i,j .
 - Values from the previous iteration for indices greater than i,j .
 - This often converges faster because it uses fresher information during the sweep.
-

Successive Over-Relaxation (SOR) Method:

- Modify the update to accelerate convergence.
- Write the Jacobi update as:

$$\Phi_{i,j}(n+1) = \Phi_{i,j}(n) + \frac{1}{4}(\sum_{\text{neighbors}} \Phi(n) - 4\Phi_{i,j}(n)). \quad \Phi_{i,j}^{(n+1)} = \Phi_{i,j}^{(n)} + \frac{1}{4}(\sum_{\text{neighbors}} \Phi^{(n)} - 4\Phi_{i,j}^{(n)}).$$

- Introduce relaxation parameter λ :

$$\Phi_{i,j}(n+1) = \Phi_{i,j}(n) + \lambda \cdot \frac{1}{4} (\sum_{\text{neighbors}} \Phi(n) - 4\Phi_{i,j}(n)). \quad \Phi_{i,j}^{(n+1)} = \Phi_{i,j}^{(n)} + \lambda \cdot \frac{1}{4} \left(\sum_{\text{neighbors}} \Phi^{(n)} - 4\Phi_{i,j}^{(n)} \right).$$

- When $\lambda > 1$: **over-relaxation** (speeds up convergence, but $\lambda < 2$ for stability).
- When $\lambda < 1$: **under-relaxation** (used for stability if iterations are unstable).

Summary:

- **Jacobi**: simple, uses only values from the previous iteration.
- **Gauss-Seidel**: updates values in place, using the latest available.
- **SOR**: modifies the update step with a relaxation factor λ to speed up or stabilize convergence.

If you'd like, I can help you write out the code for these methods or explain any step in more detail! Would you like to see how these iterative methods are implemented in MATLAB or Python?

Got it! Here's a clear summary of the **Red-Black Ordering** iterative method for solving the discrete Laplace equation, based on the video transcript you shared:

Red-Black Ordering Method (also called Red-Black Gauss-Seidel)

Concept:

- Imagine your 2D grid as a checkerboard with alternating **red** and **black** points (here purple is used to represent red, and blue to represent black).
- Each grid point's new value depends on the values of its four neighbors (up, down, left, right).

How It Works:

- **Red points** update their values using the values of the **black points** from the previous iteration.
- **Black points** update their values using the values of the **red points** from the previous iteration.

Thus, updates happen in two stages per iteration:

1. **Update all red points** simultaneously, using the black points' values from the previous iteration.
2. **Update all black points** simultaneously, using the newly updated red points.

Then repeat: red \rightarrow black \rightarrow red \rightarrow black ...

Why Use Red-Black Ordering?

- This splitting allows for a form of parallelism and can speed up convergence slightly compared to updating all points at once.
- Historically, this was an important optimization for performance on slower computers.
- Modern computers are fast enough that for simple problems like Laplace's equation, the speedup may not be noticeable.
- However, it remains a useful conceptual and practical method, especially in parallel implementations.

Summary:

Step Update grid points using values from previous step at other color

- 1 Update **Red** points using **Black** points at iteration n
- 2 Update **Black** points using **Red** points at iteration $n+1$

This approach usually converges a bit faster than the plain Jacobi or Gauss-Seidel methods updating all points simultaneously.

If you want, I can help you implement the red-black ordering iteration in code or explain its connection to parallel computing. Would you like that?

Great! Here's a clear, step-by-step summary and explanation of the **MATLAB implementation of the Jacobi method** for solving the 2D Laplace equation based on your video transcript:

Problem Setup

- **Equation:**

$$\nabla^2 \Phi = 0$$

(Laplace equation in 2D)

- **Domain:** Rectangular grid, discretized.
- **Boundary conditions:**
 - $\Phi = 0$ on bottom, left, and right boundaries.
 - $\Phi = 4x(1-x)$ on the top boundary (a function of x).

Jacobi Iteration Scheme

- Discretize the Laplace equation using finite differences.

- Rearrange the equation so that the new value $\Phi_{i,j,n+1}$ depends on the average of neighboring values at iteration n :

$$\Phi_{i,j,n+1} = \frac{1}{4} (\Phi_{i+1,j,n} + \Phi_{i-1,j,n} + \Phi_{i,j+1,n} + \Phi_{i,j-1,n})$$

- The iteration updates **all interior points simultaneously** based on the previous iteration values.

MATLAB Code Algorithm Summary

1. **Define the grid and domain parameters** (same as in the direct method).
2. **Set initial conditions and boundary conditions:**
 - Initialize Φ to zero everywhere.
 - Fix boundary values: zero on bottom, left, and right; $4x(1-x)$ on top.
3. **Define convergence criteria:**
 - $\epsilon = 10^{-6}$ (error tolerance).
4. **Indexing interior points:**
 - $\text{index_x} = 2 : \text{nx}-1$ (exclude boundaries in x)
 - $\text{index_y} = 2 : \text{ny}-1$ (exclude boundaries in y)
5. **Iteration loop:**
 - Initialize error to something larger than ϵ , e.g., $2 * \epsilon$.
 - Initialize iteration counter $\text{ncount} = 0$.
 - **While** error > ϵ :
 - Increment ncount .
 - Save current Φ as Φ_{old} .
 - Update Φ at interior points using the Jacobi formula vectorized over the grid indices:
$$\Phi(\text{index_x}, \text{index_y}) = 0.25 * (\Phi_{\text{old}}(\text{index_x}+1, \text{index_y}) + \Phi_{\text{old}}(\text{index_x}-1, \text{index_y}) + \Phi_{\text{old}}(\text{index_x}, \text{index_y}+1) + \Phi_{\text{old}}(\text{index_x}, \text{index_y}-1));$$
 - Compute error as max absolute difference between Φ and Φ_{old} over all points:
$$\text{error} = \max(\text{abs}(\Phi(:) - \Phi_{\text{old}}(:)));$$
 - Optional: print error at each iteration to monitor convergence.
6. **Stop when error < ϵ .**

7. Print number of iterations and plot the solution (reuse plotting code from direct method).

Additional Notes

- The code includes a check for divergence (NaN or Inf in $\Phi \backslash \Phi$), which helps catch bugs or unstable iteration schemes.
 - Printing the error after each iteration is useful during debugging but slows down the code and can be commented out once stable.
 - The Jacobi method converged after about **10,311 iterations** for this problem with this tolerance.
 - The final $\Phi \backslash \Phi$ matrix is available for inspection or further analysis.
-

If you want, I can help you write the full MATLAB code based on this summary or help with other iterative methods like Gauss-Seidel or Red-Black ordering. Would you like that?

Here's a concise summary and explanation of the key points from the video transcript about solving the diffusion equation as an initial value problem using the Forward Time Centered Space (FTCS) scheme:

Topic: Initial Value Problem for PDEs — The Diffusion Equation

1. The Diffusion Equation:

- We consider $u(x,t)$, a function of position x (one-dimensional) and time t .
- The diffusion equation is:

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}$$

where D is the diffusion constant.

2. Boundary Conditions (BC):

- Values of u are fixed at the boundaries of the spatial domain for all time.
- Example:

$$u(-L, t) = 0, u(L, t) = 0 \text{ for all } t \quad \Rightarrow \quad u(-L, t) = 0, \quad u(L, t) = 0 \quad \text{for all } t$$

3. Initial Condition (IC):

- Specifies $u(x, 0)$ at time zero as a known function:

$$u(x, 0) = u_0(x)$$

4. Discretizing the Domain:

- Space is discretized into grid points x_j with spacing Δx .
- Time is discretized into time steps t^n with time step size Δt .

- Notation:

$$u_{jl} = u(x_j, t_l) \quad u_j^l = u(x_j, t^l)$$

where j indexes space and l indexes time.

5. The Forward Time Centered Space (FTCS) Scheme:

- Approximate the time derivative (forward difference, Euler method):

$$\frac{\partial u}{\partial t} \approx \frac{u_{j,l+1} - u_{j,l}}{\Delta t} \quad \frac{\partial u}{\partial t} \approx \frac{u_j^{l+1} - u_j^l}{\Delta t}$$

- Approximate the second spatial derivative (centered difference):

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{j-1,l} - 2u_{j,l} + u_{j+1,l}}{(\Delta x)^2} \quad \frac{\partial^2 u}{\partial x^2} \approx \frac{u_{j-1}^l - 2u_j^l + u_{j+1}^l}{(\Delta x)^2}$$

6. Combining these into the discretized diffusion equation:

$$u_{j,l+1} - u_{j,l} = D \frac{u_{j-1,l} - 2u_{j,l} + u_{j+1,l}}{(\Delta x)^2} \Delta t \quad u_{j,l+1} - u_{j,l} = D \frac{u_{j-1}^l - 2u_j^l + u_{j+1}^l}{(\Delta x)^2} \Delta t$$

7. Rearranged as an explicit iteration scheme:

$$u_{j,l+1} = u_{j,l} + \frac{D \Delta t}{(\Delta x)^2} (u_{j-1,l} - 2u_{j,l} + u_{j+1,l}) \quad u_{j,l+1} = u_{j,l} + \frac{D \Delta t}{(\Delta x)^2} (u_{j-1}^l - 2u_j^l + u_{j+1}^l)$$

- This allows calculation of the solution at the next time step $l+1$ explicitly from the known values at time l .

8. Explicit Method:

- "Explicit" means $u_{j,l+1}$ depends only on known values at the previous time step l .
- This makes the method easy to implement and understand.

9. Stability Consideration:

- The next video will discuss the Von Neumann stability analysis.
- If the time step Δt is too large relative to Δx , this explicit method can become unstable (solution can blow up).

Summary:

- You define spatial and temporal grids.
- You apply initial and boundary conditions.
- Use FTCS to advance the solution explicitly in time.
- Careful choice of Δt relative to Δx and D is necessary to ensure stability.

If you want, I can help you write MATLAB code implementing this FTCS scheme for the diffusion equation or explain the stability criterion next! Would you like that?

Here's a clear summary and explanation of the video's key points on stability analysis of the Forward Time Centered Space (FTCS) scheme for the 1D diffusion equation:

1. The Setup: FTCS Scheme for Diffusion Equation

- The diffusion equation describes how something like heat or a chemical concentration diffuses over time.
 - The **Forward Time Centered Space (FTCS)** scheme is an **explicit** numerical method.
 - Explicit means: the solution at the next time step depends only on known values at the current time step.
 - The update formula for the solution u at grid point j and time step $l+1$ depends on values at time step l .
-

2. Stability Issue with Explicit Schemes

- Explicit schemes can become **unstable** if the time step Δt is too large.
 - Instability means the numerical solution "blows up," grows without bound, which is non-physical for diffusion (where solutions should smooth out and decay over time).
 - We want the scheme stable so that numerical solutions behave physically (diffusion causes concentration to decrease smoothly).
-

3. Von Neumann Stability Analysis

- To analyze stability, we assume solutions are made of **Fourier modes**: waves with different wave numbers k .
- The trial solution (ansatz) is:

$$u_j^l = \zeta^l e^{i k j \Delta x} = \zeta^l e^{i k j \Delta x}$$

where ζ is a growth factor per time step, $i = \sqrt{-1}$, k is wave number, and Δx is spatial grid size.

- If $|\zeta| > 1$, the scheme is unstable because that mode grows exponentially with time step l .
-

4. Deriving the Growth Factor ζ

- Substitute the ansatz into the FTCS scheme.
- After algebraic simplification, using Euler's formula:

$$e^{i k \Delta x} + e^{-i k \Delta x} = 2 \cos(k \Delta x) \Rightarrow e^{i k \Delta x} + e^{-i k \Delta x} = 2 \cos(k \Delta x)$$

- The growth factor comes out as:

$$\zeta = 1 + 2r(\cos(k\Delta x) - 1) \quad \zeta = 1 + 2r(\cos(k\Delta x) - 1)$$

where

$$r = \frac{D\Delta t}{(\Delta x)^2}$$

and D is the diffusion coefficient.

5. Interpreting Stability Condition

- Since $\cos(k\Delta x)$ ranges between -1 and $+1$:
 - Maximum ζ is at $\cos(k\Delta x) = 1$, yielding $\zeta = 1$ (neutral stability).
 - Minimum ζ is at $\cos(k\Delta x) = -1$, yielding $\zeta = 1 - 4r$.

$$\zeta_{\min} = 1 + 2r(-1) = 1 - 4r \quad \zeta_{\min} = 1 + 2r(-1) = 1 - 4r$$

- For stability, require:

$$-1 \leq \zeta_{\min} \leq 1 \quad -1 \leq 1 - 4r \leq 1$$

so

$$-1 \leq 1 - 4r \Rightarrow 4r \leq 2 \Rightarrow r \leq \frac{1}{2} \quad -1 \leq 1 - 4r \Rightarrow 4r \leq 2 \Rightarrow r \leq \frac{1}{2}$$

- Therefore, the stability condition is:

$$D\Delta t \leq \frac{(\Delta x)^2}{2} \Rightarrow \Delta t \leq \frac{(\Delta x)^2}{2D} \quad D\Delta t \leq \frac{(\Delta x)^2}{2} \Rightarrow \Delta t \leq \frac{(\Delta x)^2}{2D}$$

- If Δt is larger than this, the scheme becomes unstable.

6. Summary

- The FTCS explicit scheme for diffusion is **conditionally stable**.
- The time step must satisfy:

$$\Delta t \leq \frac{(\Delta x)^2}{2D}$$

- Larger time steps cause the scheme to blow up.
- In the next part (not in this video), implicit schemes will be shown to be **unconditionally stable**, meaning you can pick any time step size without instability.

If you want, I can also help write out the detailed algebra or explain how to implement this scheme or do a similar stability analysis for other PDEs. Would you like that?

Great summary! Let me break down the key points and explain the stability difference between the explicit Forward Time Centered Space (FTCS) scheme and the implicit scheme for the diffusion equation:

Explicit FTCS Scheme Recap:

- **Scheme:**

$$u_j^{l+1} = u_j^l + \Delta t D \Delta x^2 (u_{j+1}^l - 2u_j^l + u_{j-1}^l) \quad u_j^{l+1} = u_j^l + \frac{\Delta t D}{\Delta x^2} \left(u_{j+1}^l - 2u_j^l + u_{j-1}^l \right)$$

- Here, the spatial second derivative is computed at the **current time step l** .

- **Stability:**

The von Neumann stability analysis shows that for stability,

$$\Delta t \leq \frac{\Delta x^2}{2D}$$

Otherwise, the scheme becomes unstable and solutions can blow up.

- This means the time step size is severely limited by the spatial grid size and diffusion coefficient.

Implicit Scheme:

- **Scheme:**

$$u_j^{l+1} = u_j^l + \Delta t D \Delta x^2 (u_{j+1}^{l+1} - 2u_j^{l+1} + u_{j-1}^{l+1}) \quad u_j^{l+1} = u_j^l + \frac{\Delta t D}{\Delta x^2} \left(u_{j+1}^{l+1} - 2u_j^{l+1} + u_{j-1}^{l+1} \right)$$

- Now, the second derivative is calculated at the **future time step $l+1$** .
- Because the right-hand side contains unknowns at $l+1$, you must solve a linear system (matrix equation) for all grid points at each time step.

- **Stability:**

Von Neumann analysis yields:

$$\cos \theta = 1 + 2r(1 - \cos(k\Delta x)) \quad \cos \theta = \frac{1}{1 + 2r(1 - \cos(k\Delta x))}$$

where

$$r = \Delta t D \Delta x^2 = \frac{\Delta t D}{\Delta x^2}$$

- Since the denominator is always ≥ 1 , the magnitude of $\cos \theta$ is always ≤ 1 .
- **Result:** The implicit method is **unconditionally stable** — no matter how large Δt is, the scheme will not blow up.

Why is this important?

- **Explicit methods** are simple but require very small time steps to maintain stability, especially for small spatial grid sizes.

- **Implicit methods** are more complex computationally (solve a system of equations each step) but allow larger time steps without instability.
 - This makes implicit methods preferable for stiff problems or when long time simulations are needed.
-

Next Step: Crank-Nicolson Method

- Combines explicit and implicit ideas.
 - It is second-order accurate in time and unconditionally stable.
 - We'll discuss that next!
-

If you'd like, I can help you write out the matrix form for the implicit scheme or guide you through the Crank-Nicolson method next. How do you want to proceed?

Great detailed explanation! Let me summarize and clarify the Crank-Nicolson method and its matrix formulation for solving the 1D diffusion equation:

Problem Setup:

We want to solve the diffusion equation

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}$$

on a spatial grid indexed by j and time steps indexed by l .

Recap of Methods:

- **Explicit FTCS:**
Uses spatial second derivative at time l only, conditionally stable.
 - **Fully Implicit:**
Uses spatial second derivative at time $l+1$, unconditionally stable but requires solving a matrix system each step.
-

Crank-Nicolson Method:

- **Idea:** Average the spatial second derivative between time levels l and $l+1$. This makes the method:
 - **Second-order accurate** in both space and time.
 - **Unconditionally stable** like the fully implicit method.
- **Finite difference equation:**

$$u_{j+1} = u_j + \Delta t D \Delta x^2 \left[(u_{j+1} - 2u_j + u_{j-1}) + (u_{j+1} - 2u_j + u_{j-1}) \right] u_j^{l+1} = u_j^l + \frac{\Delta t D}{\Delta x^2} \left[(u_{j+1}^l - 2u_j^l + u_{j-1}^l) + (u_{j+1}^{l+1} - 2u_j^{l+1} + u_{j-1}^{l+1}) \right]$$

Rearranged form:

Define

$$\alpha = \Delta t D \Delta x^2 \quad \alpha = \frac{\Delta t D}{\Delta x^2}$$

Multiply both sides by 2 to clear denominators and rearrange terms to isolate u_{j+1}^{l+1} :

$$- \alpha u_{j+1}^{l+1} + 2(1+\alpha)u_j^{l+1} - \alpha u_{j-1}^{l+1} = \alpha u_{j+1}^l + 2(1-\alpha)u_j^l + \alpha u_{j-1}^l - \alpha u_{j+1}^{l+1} + 2(1+\alpha)u_j^{l+1} - \alpha u_{j-1}^{l+1} = \alpha u_{j+1}^l + 2(1-\alpha)u_j^l + \alpha u_{j-1}^l$$

Matrix Form:

For a grid with 4 points (including boundaries), assume boundary conditions $u_1 = u_4 = 0$ (Dirichlet zero BCs):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -\alpha & 2(1+\alpha) & -\alpha & 0 \\ 0 & -\alpha & 2(1-\alpha) & \alpha \\ 0 & 0 & \alpha & 1 \end{bmatrix} \begin{bmatrix} u_1^{l+1} \\ u_2^{l+1} \\ u_3^{l+1} \\ u_4^{l+1} \end{bmatrix} = \begin{bmatrix} 0 \\ \alpha u_1^l + 2(1-\alpha)u_2^l + \alpha u_3^l \\ 2(1+\alpha)u_j^l - \alpha u_{j-1}^l \\ 0 \end{bmatrix}$$

- The first and last rows implement the boundary conditions.
- The interior rows form a **tridiagonal system** that you solve each timestep.

Important notes:

- The matrix is **constant in time**, so you can factorize it once (LU decomposition or Thomas algorithm) for efficient solving.
- At each time step, the right-hand side vector updates based on the previous solution u^l .
- The tridiagonal structure means solving the system is very fast (order $O(N)$).

Summary:

- Crank-Nicolson method is an **average of explicit and implicit schemes**.
- It is **unconditionally stable**.
- It is **second-order accurate** in both space and time.
- Leads to a **tridiagonal matrix system** to solve each timestep.
- Boundary conditions are directly incorporated into the matrix system.

If you'd like, I can help you write MATLAB or Python code to implement this, or explain the Thomas algorithm for efficiently solving the tridiagonal system. Would you like to see that next?

Great! Here's a concise summary of the MATLAB implementation of the Crank-Nicolson method for the 1D diffusion equation based on the video and transcript you shared:

MATLAB Code Outline for Crank-Nicolson Diffusion Solver

1. Define domain and parameters:

- Domain length $L_x = 1$ (so $x \in [-1, 1]$)
- Number of intervals $N_x = 500$, grid points $n_x = N_x + 1 = 501$
- Grid spacing $\Delta x = \frac{L_x}{N_x}$
- Diffusion constant $D = 1$ (unit diffusion)
- Time step $\Delta t = \frac{\Delta x^2}{2D}$ (borderline stable FTCS step, adjustable)
- Number of time steps (e.g., 10,000)
- Output interval (e.g., plot every 500 steps)

2. Create spatial grid:

```
x = linspace(-Lx, Lx, nx); % row vector of grid points
```

3. Define $\alpha = \frac{D \Delta t}{\Delta x^2}$

4. Construct the time-independent tridiagonal matrix **AA**:

- Main diagonal: $2(1 + \alpha)$
- Off-diagonals (upper and lower): $-\alpha$
- Use MATLAB's `spdiags` for efficient sparse matrix creation
- Impose Dirichlet boundary conditions by replacing first and last rows with identity rows

```
main_diag = 2*(1 + alpha) * ones(nx,1);
```

```
off_diag = -alpha * ones(nx-1,1);
```

```
A = spdiags([off_diag main_diag off_diag], [-1 0 1], nx, nx);
```

```
% Boundary conditions (u=0 at boundaries)
```

```
A(1,:) = 0; A(1,1) = 1;
```

```
A(end,:) = 0; A(end,end) = 1;
```

5. Initialize **uu** as a Gaussian centered in the domain:

```
sigma = Lx / 16;
```

```
u = (1/(sigma*sqrt(2*pi))) * exp(-0.5 * (x/sigma).^2);
```

```
u = u'; % convert to column vector
```

6. Plot initial condition:

```
plot(x, u, 'r');
```

```
xlabel('x'); ylabel('u');
```

```
title('Initial Condition');
```

```
drawnow;
```

7. Time-stepping loop:

For each time step:

- Construct the right-hand side vector BB using values of uu at previous step, according to Crank-Nicolson scheme
- Impose boundary conditions on BB (zero at boundaries)
- Solve the linear system $Au_{l+1} = B$ using MATLAB's backslash operator
- Plot solution every nout steps

```
for m = 1:ntime
```

```
    B = zeros(nx,1);
```

```
    % Boundary conditions for B
```

```
    B(1) = 0; B(end) = 0;
```

```
    % Fill B for interior points (2 to nx-1) using previous u and alpha
```

```
    B(2:end-1) = alpha*u(1:end-2) + 2*(1 - alpha)*u(2:end-1) + alpha*u(3:end);
```

```
    % Solve linear system
```

```
    u = A \ B;
```

```
    % Plot every nout steps
```

```
    if mod(m, nout) == 0
```

```
        hold on;
```

```
        plot(x, u);
```

```
        drawnow;
```

```
    end
```

end

Key Points:

- The matrix AA is sparse, tridiagonal, and time-independent \rightarrow efficient setup.
- Boundary conditions (Dirichlet zero) are enforced by fixing first and last rows of AA and BB .
- The Crank-Nicolson method is implicit but stable and second-order accurate.
- MATLAB's \backslash operator makes solving the system simple and fast.
- The solution uu starts as a sharp Gaussian and diffuses over time, spreading out and flattening.

If you'd like, I can help you write the **full MATLAB script** ready to run, or help with any specific part of the code or explanation! Would you like that?

Great! This is an excellent summary and guidance on tackling the **2D diffusion equation using the Crank-Nicolson method** in MATLAB. Let me break down the key steps and the main ideas from your project description, so you have a clear roadmap for coding:

Overview: 2D Diffusion Equation

$$\frac{\partial u}{\partial t} = D(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2})$$

- $u = u(x, y, t)$
- Domain: square $[-L, L] \times [-L, L]$
- Boundary conditions: $u = 0$ on all boundaries
- Initial condition: large concentration at the center, decaying to zero at boundaries

Step-by-Step Solution Outline

1. Define the grid

- Create a uniform grid for xx and yy with spacing $h = \Delta x = \Delta y$.
- Total number of points $N_x = N_y = N$ (fixed for grading).
- Use **natural ordering** (e.g., row-major or column-major) to flatten the 2D grid $u_{i,j}$ into a 1D vector u_k where $k = 1, 2, \dots, N^2$.

2. Discretize the PDE with Crank-Nicolson

- Time discretization: implicit, average of the spatial discretization at time n and $n+1$.
- Spatial discretization: standard 5-point stencil for Laplacian

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

3. Construct the matrix AA

- The Crank-Nicolson scheme can be expressed as

$$A u^{n+1} = B u^n + b$$

where AA and BB are large sparse matrices of size $N^2 \times N^2$.

- AA includes diagonal elements and off-diagonals that connect points in the grid according to the 5-point stencil.
- Apply boundary conditions by setting rows corresponding to boundary nodes to enforce $u=0$.

4. LU Decomposition

- To accelerate solving $A u^{n+1} = \text{RHS}$ at every time step, compute LU factorization once outside the time loop.
- Inside the loop, solve two triangular systems instead of inverting or directly solving AA.

5. Initial condition

- Initialize u as a large peak at the center of the domain (e.g., Gaussian).
- Flatten it into a vector matching the ordering scheme.

6. Time stepping loop

- For each time step:
 - Compute the right-hand side vector $b = B u^n$ plus boundary conditions.
 - Solve for u^{n+1} using the LU factors.
 - Optionally save or plot u^{n+1} .

7. Graphics

- Plot u as a 3D surface or contour plot.
- Animate or update plot every few steps to show diffusion.

Summary of main MATLAB implementation ideas:

- Use **sparse matrices** for AA and BB.
- Use **natural ordering**: e.g., flatten 2D indices (i,j) into 1D index $k = i + (j-1) * N$.
- Enforce boundary conditions by replacing rows in AA with identity rows.
- Use lu factorization in MATLAB once outside time loop.
- Use `\` operator with LU factors for fast solve inside time loop.

- Plot results using surf or mesh for 3D visualization.
-

If you'd like, I can help you by writing a **basic MATLAB template** for this 2D diffusion problem based on the above outline. Would you like me to do that?

This is a great course summary highlighting the main topics you covered in your Numerical Methods for Engineers class, including all six projects! Here's a quick recap and the big picture for each week/project:

Week 1: Basics & Roundoff Errors

- Learned MATLAB programming fundamentals.
- Explored roundoff errors.
- **Project:** Compute the bifurcation diagram for the logistic map.

Week 2: Root Finding Methods

- Learned Newton's method and fzero.m for root finding.
- **Project:** Use Newton's method to compute Feigenbaum Delta.

Week 3: Numerical Linear Algebra

- Focused on Gaussian elimination with partial pivoting (MATLAB's \ operator).
- Applied to solving nonlinear systems and PDEs.
- **Project:** Compute a fractal from the Lorenz equations.

Week 4: Quadrature & Interpolation

- Learned numerical integration with integral.m.
- Learned interpolation methods such as cubic splines with interp1.m.
- **Project:** Compute zeros of Bessel functions.

Week 5: Ordinary Differential Equations

- Learned the Runge-Kutta method, especially MATLAB's ode45.
- Learned shooting methods for boundary value problems.
- **Project:** Compute orbits of two gravitationally interacting masses using Newton's laws.

Week 6: Finite Difference Methods for PDEs

- Learned finite difference schemes for PDEs including diffusion equations.
 - Solved 1D and 2D diffusion equations.
 - **Project:** Solve the two-dimensional diffusion equation using Crank-Nicolson.
-

Final Thoughts

- These numerical methods and MATLAB skills are essential for engineering and scientific computing.
- They're highly valued by professors and research groups who need capable programmers for solving complex numerical problems.

If you want, I can help you review or practice any particular topic or code from these weeks. Which part would you like to dive into?