ChatGPT

Manim Cheat Sheet for Scenes, Mobjects, Animations, and Cameras

Manim (Manim Community edition) is a Python library for creating precise programmatic animations, especially useful for math, crypto, and AI explainer videos. This cheat sheet covers the **basic and intermediate** functionalities of Manim's core modules – **Scenes**, **Mobjects**, **Animations**, and **Cameras** – with tips on using them to build rich animated videos. It is structured to help an AI coding agent iteratively write Manim code for each scene of a video.

Scenes: Organizing Your Animation

- **Scene Class** – The fundamental canvas for animations. You create a scene by subclassing `Scene` and overriding the `construct()` method with your animation code [1]. Within `construct()`, you can **add** mobjects to display (using `self.add(...)`), **remove** mobjects, and **play** animations (using `self.play(...)`) [2]. For example:

```
from manim import Scene, Write, Text

class MyScene(Scene):
    def construct(self):
        self.play(Write(Text("Hello World!")))
```

In this example, a `Text` mobject is written to the screen with an animation [3]. A Manim script can contain multiple Scene subclasses; each scene will render as a separate segment of the final video.

- **2D Scenes (Moving Camera)** – By default, `Scene` uses a fixed camera. To move or zoom the camera in a 2D scene, use `MovingCameraScene`. This subclass makes it easy to pan/zoom by manipulating the camera during animations [4]. For instance, in a `MovingCameraScene`, you can animate the camera frame: `self.play(self.camera.frame.animate.scale(0.5).move_to(new_center))` to zoom in and pan. (Internally, `MovingCameraScene` uses a `MovingCamera` to allow camera movement [4].)

- **3D Scenes** – Use `ThreeDScene` for scenes with three-dimensional content. It comes pre-configured with a `ThreeDCamera` for 3D rendering [5]. You can set the initial camera angle with `self.set_camera_orientation(phi=..., theta=...)` and animate 3D camera movement. For example, `self.move_camera(phi=60*DEGREES, theta=30*DEGREES)` rotates the camera to new angles during the animation [6]. You can also start an ambient continuous rotation with `self.begin_ambient_camera_rotation(rate=0.1)` for a slowly spinning scene [7]. Remember to call `self.wait()` after camera moves to give viewers time to absorb the view change.

- **Zoomed Scene (Inset Zoom)** – Manim's `ZoomedScene` is useful when you want to zoom in on a part of the scene with a magnifier effect [8]. It provides a secondary zoomed camera and a display window. You can activate it with `self.activate_zooming()` (optionally animated) and then play with the zoomed camera frame (by default it shows an inset that zooms into a defined portion of your scene) [9] [10]. This is great for highlighting detail in a math equation or diagram without cutting away from the main scene.

- **Other Specialized Scenes** – Manim includes scene classes for specific purposes:

- `LinearTransformationScene`: a scene setup for visualizing linear transformations in linear algebra (provides a grid and vectors that you can transform with matrices).
- `VectorScene`: a scene for demonstrating vectors and vector operations.
- These are more advanced; for most math/AI explainer needs, you will primarily use `Scene` or the variants above. It's recommended to stick to the basic Scene types unless a specific advanced use-case arises.

Best Practice: Keep each Scene focused on one concept or step of your explanation. This makes it easier to iterate on and for a coding LLM to manage one scene at a time. You can later concatenate the rendered scenes into a full video.

Mobjects: Building Blocks of Content

A **Mobject** ("mathematical object") is any object that can be displayed on the scene – text, shapes, graphs, etc. You create and manipulate mobjects to design your visuals, then add them to a scene. Key points and types of mobjects:

- **Creating and Adding Mobjects** – To display a mobject, instantiate it and then call `scene.add(mobject)`. Objects added last are drawn on top (foreground) by default [11]. You can remove a mobject with `scene.remove(mobject)`. Mobjects have various methods to position and style them, many of which can be chained (since most methods return the mobject itself) [12]. For example, `circle = Circle().shift(LEFT).set_fill(YELLOW, opacity=0.5)` creates a circle, moves it left, and fills it with semi-transparent yellow in one line.

- **Geometric Shapes** – Manim provides many basic shapes out of the box, all as mobjects:

- **Circle**, **Square**, **Rectangle**, **Triangle**, **Dot**, **Line**, **Arrow**, etc. For example, `Circle(radius=1.0, color=BLUE)` creates a circle of radius 1 with blue outline [13], and `Square(side_length=2)` creates a 2x2 square. Most shape constructors accept common style arguments like `color` or `fill_opacity`. You can further style shapes with methods: e.g. `shape.set_stroke(color=GREEN, width=10)` to change outline color/width, or `shape.set_fill(RED, opacity=0.8)` to fill with red [14]. By default, shapes have transparent fill (opacity 0) until you set it [15]. You can move or rotate shapes with methods like `shift()`, `rotate()`, `scale()`, etc., which can be chained as mentioned.

- **Grouping**: Use `VGroup` or `Group` to combine multiple mobjects into one group for easier manipulation [16] [17]. For instance, `group = VGroup(circle, square).arrange(buff=1)` will place a circle and square side by side with a gap of 1 unit and treat them as a single mobject thereafter. This is helpful for moving multiple objects together or applying one animation to many objects at once.

- **Text and Math** – Manim excels at rendering text, especially mathematical notation:

- **Text**: for regular text (using system fonts). Example: `Text("Hello world", font_size=36, color=WHITE)`. This creates a text mobject you can add to a scene. You can change font, size, color, etc. via parameters or methods like `set_color`.
- **MathTex / Tex**: for LaTeX-formatted math. Use `MathTex` for mathematical expressions. For example, `MathTex("E = mc^2")` will render the equation $E = mc^2$ as LaTeX [18]. If you need text in math mode or want more manual control, `Tex` is also available (similar usage). Ensure a TeX distribution is installed for this to work. You can scale or color parts of equations: e.g. `formula = MathTex("E", "=", "m c^2"); formula[0].set_color(YELLOW)` would color the *E* in the equation.
- **Isolating parts of formulas**: You can cause specific sub-parts of a formula to be separate sub-mobjects by wrapping them in double braces in the LaTeX string. For example: `MathTex("{{x}}^2 + {{y}}^2 = {{z}}^2")` treats each of *x*, *y*, *z* as separate elements [18]. This is extremely useful for highlighting or transforming those parts independently (as we'll see with animations like TransformMatchingTex). It also allows matching same-text parts between different formulas.

- **MarkupText**: if you need rich text formatting (bold, italics, different colors in one text), `MarkupText` allows Pango markup in the string. This is more advanced and not usually needed for simple math videos.

- **Shapes for Emphasis** – Some mobjects exist mainly to annotate or emphasize other mobjects:

- **SurroundingRectangle**: creates a rectangle that *surrounds* a given mobject or group, useful for highlighting something by drawing a box around it. For example, `highlight = SurroundingRectangle(formula_part, color=YELLOW, buff=0.1)` will create a rectangle tightly around `formula_part` (could be a term in an equation). You can then animate this (e.g. using `Create(highlight)` to draw it) to focus attention.

- **Brace / BraceLabel**: places a curly brace adjacent to a mobject (usually underneath or on the side). For instance, `brace = Brace(group, direction=DOWN)` creates a brace under a group of mobjects, and `brace_text = brace.get_text("Explanation")` puts a text label at the brace [19] [20]. This is great for annotating parts of an equation (like summation terms or numerators/denominators) with an explanation.

- **Graphs and Plots**:

- **Axes**: for coordinate systems. `Axes` creates a set of x-y axes to plot functions or data [21]. You can configure the ranges and appearance: e.g. `ax = Axes(x_range=[0, 10, 1], y_range=[-2, 6, 1], tips=False)` makes an axes from 0 to 10 on x and -2 to 6 on y with no arrow tips [22] [23]. Once you have an `Axes` object, you can plot functions: `graph = ax.plot(lambda x: x**2, x_range=[0, 4])` will return a curve (a ParametricFunction internally) for $y = x^2$. Add both the axes and graph to the scene:

`self.add(ax, graph)` [24]. There are also helpers like `ax.get_graph()` or `ax.plot_line_graph()` for specific tasks. Use `Axes` when explaining graphs (e.g. loss curves in AI or mathematical functions like *xy*=k*).

- **NumberPlane**: a grid of horizontal and vertical lines with axes, often used as a background grid for visualizations (e.g., to illustrate geometry or coordinate space). Create with `NumberPlane()` or configure similar to Axes. You can animate the NumberPlane (e.g., apply transformations to it in linear transformation scenes).
- **BarChart** and others: Manim includes chart mobjects (like `BarChart`) for visualizing data distributions. For example, to illustrate a probability distribution (which might be relevant in a cross-entropy context), you could use a `BarChart` with appropriate values. This is intermediate usage; you can also manually construct bars using rectangles for full control.
- **Advanced Mobjects (for reference)**:
- **Graph (network graphs)**: Manim can create network graphs using the `Graph` mobject, which takes a list of vertices and edges. This could be used to illustrate neural network architecture (vertices as neurons, edges as connections) or any graph structure. You can specify positions or use automatic layouts. For a neural network, however, it might be simpler to manually position circles and connect them with `Line` or `Arrow` mobjects for full control.
- **ValueTracker and Variable**: These are useful for dynamic values. A `ValueTracker` is not visible on screen but tracks a numeric value that you can update over time (especially helpful in animations). A `Variable` mobject combines a numeric display with a label. For example, `var = Variable(5, "x")` gives you a number 5 labeled x, and you can animate `var.tracker` (a ValueTracker) to change the number, causing the on-screen value to update. This can be used to animate a changing loss value, or an increasing step count, etc., in your videos.

Best Practice: Construct complex formulas or diagrams out of simpler mobjects and groups. This not only allows reusing pieces in animations, but it also enables targeted animations (like highlighting one term in an equation). Leverage grouping (`VGroup`) and submobjects (like the parts of MathTex) to your advantage for fine control.

Animations: Bringing Objects to Life

Animations in Manim interpolate mobjects from a start state to an end state over time [25]. You trigger animations by calling `self.play(Animation(obj, ...), ...)` inside a Scene's `construct()`. Here are common animation types and how to use them:

- **Fade and Appearance Animations**:
- **FadeIn / FadeOut** – Fades a mobject into or out of view. For example, `self.play(FadeIn(mobject))` will start with `mobject` fully transparent and smoothly increase its opacity to full [25]. Conversely, `FadeOut(mobject)` interpolates from opaque to transparent [25]. Use these to introduce or remove objects without drawing their outline.
- **Create / Uncreate** – Draws a shape's outline (or outline of text) as if being sketched. `Create(mobject)` will animate the stroke of a shape or letters of text appearing stroke-by-stroke (particularly nice for geometric shapes or diagrams). `Uncreate` does the reverse, animating the stroke being erased. Similarly, **Write** is typically used for writing out text or LaTeX equations letter by

letter (internally it's akin to `ShowCreation` for text) [3]. For example, `self.play(Write(formula))` will animate the formula's appearance as if being written.

- **GrowFromCenter / GrowFromEdge** – These animations (in the "growing" category) make a mobject appear starting from a point. For instance, `GrowFromCenter(mobj)` scales a mobject from a single point at its center to full size [26], and `GrowFromEdge(mobj, edge=LEFT)` would make it appear stretching out from its left edge [26]. These are useful for emphasizing introduction of new parts of a diagram (e.g., popping in a new node in a network).

- **Transforming and Moving Objects**:

- **Transform / ReplacementTransform** – Morph one mobject into another. If you want to smoothly change an object into a different one (e.g., change one equation into a new equation, or morph a shape into another shape), you can use `self.play(Transform(obj1, obj2))`. This will interpolate every point of `obj1` into `obj2`'s shape (both must be on screen; by default `obj2` is introduced). `ReplacementTransform(old, new)` is similar but implicitly removes the old object at the end. Use these when you want a *continuous* transformation instead of a sudden switch.
- **TransformMatchingTex** – A powerful variant of transform specifically for transforming one LaTeX string to another while matching similar parts [27]. If two equations share sub-expressions (like a term that appears in both), `TransformMatchingTex(old_eq, new_eq)` will move the matching pieces from the old to the new positions rather than fading them out and in [27]. For this to work best, you should isolate terms in your MathTex with double braces as noted earlier, so that, for example, the "x^2" in equation 1 can directly map to "x^2" in equation 2. This animation is extremely useful for **step-by-step derivations**: you can show an initial formula, then play `TransformMatchingTex(formula, next_formula)` to morph it into the next stage, preserving any terms that remain the same [18] [28]. It creates a smooth experience where only the changed parts move/replace, and unchanged parts stay in place.
- **MoveAlongPath** – Moves an object along a given path. You supply a path mobject (usually a curve) and the object will slide along it. For instance, if you have a Dot named `dot` and a circular path `circle`, `self.play(MoveAlongPath(dot, circle))` will animate the dot moving along the circle's circumference.
- **Rotate** – Rotates a mobject about its center (or a specified point). Usage: `self.play(Rotate(mobject, angle=PI/4))` will animate rotating the object 45° (π/4 radians) [29] [30]. There's also a convenience `mobject.animate.rotate(angle)` which we discuss below.

- **.animate syntax** – *Any changable property of a mobject can be animated* via the `animate` shorthand [31]. Instead of explicitly using a Transform, you can do things like `self.play(mobject.animate.shift(RIGHT * 2))` to move `mobject` 2 units to the right. Manim will internally handle creating the proper animation (in this case, a transform of the object's position) [32] [33]. You can chain multiple transformations in one call: e.g. `self.play(square.animate.shift(UP).rotate(PI/3))` will move the square up while simultaneously rotating it by 60° [33]. This syntax works for any method that changes the mobject's state: `mobject.animate.set_fill(WHITE)` will animate a color change [33], etc. The `.animate` syntax keeps code concise and is highly recommended for simple property changes.

- **Emphasis Animations** (Indication/Attention):

- **Indicate** – Briefly highlights a mobject by flashing it (typically by changing color and scaling up slightly, then back). `self.play(Indicate(mobj))` will make `mobj` momentarily larger or colored to draw attention. This is great for pointing out a term in an equation or a component of a diagram without permanently changing it.
- **Circumscribe** (formerly known as ShowCreationThenFadeAround) – Draws a temporary highlight shape (usually a rectangle or circle) around a mobject. For example, `self.play(Circumscribe(mobj, color=YELLOW))` will outline `mobj` with a rectangle and then fade it. This is another way to draw viewer's focus.
- **Wiggle** – Shakes an object back and forth a little. `self.play(Wiggle(mobj))` can be used in a light-hearted way to indicate something of interest (like wiggling an arrow or a pointer).
- **Flash** – Flashes a radial light (like an expanding circle) at a point. `self.play(Flash(dot))` would create a quick flash at the location of `dot` (often used to indicate a point on a graph or a click effect). There's also **FocusOn** which dims the scene except a small area around a point, for a spotlight effect [34].

- These indication animations are short and often used in combination with waits to emphasize parts of the scene. They don't fundamentally change the mobjects, just draw attention.

- **Sequential and Parallel Animations**:

- By default, each `self.play()` call runs animations in **parallel** (simultaneously) if you pass multiple animations to one play. If you do `self.play(FadeIn(mobj1), FadeIn(mobj2))`, both will fade in together.

- To sequence animations back-to-back, call multiple `play()` in a row (each call waits for previous animations to finish). For more complex choreography:

    - **AnimationGroup** – explicitly group animations to play together as one (useful if you need to treat a combo as a single animation in a longer list).
    - **LaggedStart** – plays a group of animations or the same animation on multiple mobjects in a staggered way (each starts a bit after the previous). For example, `LaggedStart(FadeIn(obj1), FadeIn(obj2), lag_ratio=0.5)` would fade in obj1, and halfway through its fade start fading in obj2.
    - **Succession** – queue animations to run one after the other automatically. This is like doing multiple `play` calls but encapsulated.
    - These are intermediate tools; often you can manage with multiple `play` calls and some careful ordering. They become handy when many things need to be animated in complex overlaps or staggers.

- **Waiting**:

- Use `self.wait(seconds)` to pause the scene for a given duration (in seconds). `Wait` is an animation (e.g., `self.play(Wait(2))` is equivalent to `self.wait(2)`). This is useful to hold the final state of an animation on screen, giving the viewer time to process, or to create a pause between phases of your explanation. Always include brief waits after important motions or before scene transitions so the video isn't too fast.

Best Practice: Animate one clear idea at a time. For example, to explain a formula, you might first *FadeIn* the formula, then *Indicate* a particular term, then use *TransformMatchingTex* to replace that term with something else, etc., with short waits in between. This pacing helps a viewer follow along. Also, when animating multiple objects, consider whether they should move together (then play them in parallel) or sequentially (play calls back-to-back). Use easing (rate functions) if needed to adjust motion style (Manim defaults to smooth, but you can import linear, rush_into, etc., if desired).

Cameras: Controlling the View

Manim's camera determines what portion of the scene is visible and at what angle. Typically you don't have to manage the camera for simple scenes (the default camera fits all objects added), but for zooming, panning, or 3D rotations, understanding the camera is important:

- **Default Camera (Scene)** – Every Scene has a `camera` attribute (usually an instance of `Camera` or `MovingCamera`). In standard scenes, this camera is fixed. The visible area is the frame. You can think of `self.camera.frame` as a mobject representing the current view window. By moving or scaling that frame, you change the view. For example, in a `MovingCameraScene`, you could zoom in by `self.play(self.camera.frame.animate.scale(0.5))` (making the frame half-size zooms in by 2x) and pan by shifting that frame to a new position [4]. This approach gives you programmatic control to focus on different parts of a large scene.

- **MovingCamera and Scenes** – If you plan a lot of camera motion in 2D, use `MovingCameraScene` as mentioned. This ensures that `self.camera` is a `MovingCamera` and that the frame adjustments will interpolate smoothly. The camera's position or zoom can be animated just like any mobject (as shown with `animate` on `camera.frame`). By default, when you add mobjects, the camera auto-adjusts to include them, but with a moving camera you might want to manually set the initial frame size/position for consistency.

- **ThreeDCamera (Perspective)** – In `ThreeDScene`, the camera is a `ThreeDCamera` which allows rotation around the scene and depth perception. Important controls:

- `set_camera_orientation(phi, theta, distance)` to set the spherical coordinates (angles) of the camera view [35]. Phi is the polar (vertical) angle downward from z-axis, theta is the azimuth (rotation in the plane), and distance is how far the camera is (zooming out if larger).
- `move_camera(phi, theta, frame_center=...)` to animate moving to a new orientation [6]. You can pass a target `frame_center` (a point to center on) as well.
- `begin_ambient_camera_rotation(rate)` to start a slow continuous rotation (e.g., rotating around z-axis) [7]. This runs in the background (you can stop it with `stop_ambient_camera_rotation()`).
- You can also add mobjects that should stay fixed in screen orientation (like UI or labels that shouldn't tilt) using `add_fixed_in_frame_mobjects(mobj)` so they don't rotate with the 3D scene [36].

- When making 3D plots or shapes, you might start your scene with `self.set_camera_orientation(phi=75 * DEGREES, theta=-45*DEGREES)` to get a nice angle, then add your 3D mobjects, and use camera animations for rotation effects.

- **Multi-Camera and Advanced** – Manim allows multiple cameras if needed (see classes like `MultiCamera`, `SplitScreenCamera`), but these are advanced and rarely needed for typical videos. One case is the `ZoomedScene` which actually uses a `MultiCamera` under the hood to show the main scene and a zoomed sub-scene together [37]. In `ZoomedScene`, after `activate_zooming()`, you get attributes like `self.zoomed_camera` and a `self.zoomed_display` (the little window). You can animate those (e.g., play `self.get_zoom_in_animation()` to smoothly pop out the zoom window) [38], but again, this is a special case. For most needs, a single moving camera or 3D camera suffices.

Best Practice: Use camera moves sparingly and with purpose – e.g. zoom in to show detail or rotate the perspective to reveal a 3D structure. Sudden or frequent camera moves can confuse viewers. Always give a pause (`self.wait()`) after a camera move so the audience can orient themselves. If an animation can be achieved either by moving objects or moving the camera, consider the clarity: moving the camera can feel like changing the viewer's perspective (good for big picture changes), while moving the objects feels like manipulating the content itself (better for demonstrating the concept).

Putting It Together – Tips for Math, Crypto, and AI Explainers

Finally, here are some targeted tips for using the above tools to illustrate concepts in AI, math, and cryptography:

- **Step-by-Step Equations (e.g. Cross-Entropy)**: When explaining a formula like the cross-entropy loss *H(p,q) = -Σ p(x) log q(x)*, you can write the full formula with `MathTex`, then break it down term by term. Use `TransformMatchingTex` to go from the general formula to a specific expanded example (for instance, transform *-Σ p log q* into *-(p(x_1)\log q(x_1) + p(x_2)\log q(x_2) + ...)*, matching the common parts like the minus sign and log). You might **FadeIn** each term of the summation one at a time (or use `ShowIncreasingSubsets` which reveals elements of a VGroup sequentially). Accompany each appearance with a brief explanation (maybe a `Text` annotation or a **Brace** grouping the term with a label like "probability * log likelihood"). Highlight important terms with **Indicate** or **Circumscribe** (e.g., circle the *log q(x)* part to discuss prediction confidence). This incremental approach – reveal, highlight, explain – is effective for complex equations.

- **AI Algorithms (e.g. Policy Gradient, PPO)**: For algorithms, consider flowcharts or process diagrams. You can use **Text** mobjects in rectangles (make a rectangle with `Rectangle()` and put a `Text` on top, group them) to represent steps or components (like "Policy Network" or "Reward Signal"). Connect them with **Arrow** mobjects to show the flow of information. Animate the flow: for example, move a small dot or a flashing **FadeIn** along the arrows to indicate data passing (like a trajectory of states, or gradients flowing back). If there are equations (like update rules), display them on the side and use **TransformMatchingTex** to update symbols (e.g., show how an advantage estimate is calculated). Reinforcement learning concepts often benefit from showing a loop: you can animate a **Dot** moving in a circular path to represent the iterative nature of updates or episodes, with each loop adding to a counter (using a ValueTracker with an updating number).

- **Cryptography Math (e.g. RSA, Uniswap's *x*y=k*)**: For pure math relationships like *x*y = k* (Uniswap's invariant), a great visualization is plotting it and using an animated point:

- Create Axes for x and y, and plot the hyperbola curve *y* = *k/x*. Then place a `Dot` on the curve. Use a `ValueTracker` for x-value, and update the Dot's position so that as x changes, y = k/x is computed – this can be done with an updater function or by parameterizing the Dot's position along the curve. Now animate the `ValueTracker`: as x increases, y decreases, and the Dot moves along the curve, illustrating the inverse relationship. Meanwhile, you could use two small vertical/horizontal bars or arrows on the axes to show the coordinates (or even a `DecimalNumber` that updates for x and y values). This dynamic view makes the constraint *x*y=k* clear.
- For cryptographic processes (like RSA encryption flow), combine text and arrows similarly to the AI flowchart suggestion: show plaintext → (math operations) → ciphertext. Use animations like **FadeTransform** to morph a plaintext number into an encrypted number to symbolize encryption. If explaining a formula (like modular exponentiation), you can do the step-by-step reveal of the equation as with cross-entropy.

- If illustrating a concept like a blockchain or Merkle tree, you might use `VGroup` to form layers of a tree and animate how a change in one leaf causes changes in the root (highlighting boxes and using FadeTransform on hash values, etc.).

- **Neural Networks and Deep Learning**: To depict a neural network:

- Use circles or small dots for neurons (e.g., `Circle(radius=0.1, fill_color=BLUE, fill_opacity=1)` for each neuron). Arrange them in layers using `VGroup(...).arrange(DOWN, buff=0.3)` for each layer, then position layers side by side (with `.arrange(RIGHT, buff=1.0)`). Connect neurons with lines: you can loop through neurons in adjacent layers and draw a `Line` or `Arrow` between each pair (or for a cleaner look, between each neuron in layer L and the corresponding neuron in layer L+1 only). Group all lines in a `VGroup` as well (e.g., `connections = VGroup(*all_lines)`).
- Once your network diagram is set, you can animate **feedforward activation** by highlighting nodes and edges. For example, use **Indicate** or change color of a neuron when it "activates". Or animate a small dot traveling along an Arrow to represent a signal. You could use `LaggedStart` to sequentially flash neurons in one layer then the next, showing the propagation of information.
- If explaining training, you might show an error value decreasing: use a `ValueTracker` attached to a `DecimalNumber` to display the error, and animate it decreasing over time in sync with some highlight on the network. If discussing concepts like cross-entropy loss in this context, tie it back to the earlier formula visualization (perhaps showing the loss calculation for a specific output, then updating the network accordingly).

- **Tip:** Keep the network diagram on screen (maybe in a corner or faded) while showing equations on another part, to remind the viewer of context. Manim's ability to animate multiple objects means you can have the network on one side and equations on the other; use camera framing or `self.play(FadeIn(group_of_equations, shift=RIGHT))` to bring equations in without removing the network diagram.

- **General Clarity**: Always synchronize your narration (or on-screen text explanations) with the animations. For every term that appears or changes in an equation, consider adding a brief text label or voiceover explanation. Manim can even handle **subtitles or captions** via the `add_subcaption` method, but that's beyond the scope here. For our purposes, ensure the cheat sheet guides the coding agent to create *self-explanatory animations* – for instance, labeling axes, naming variables with `Tex` labels, etc., so the visuals alone carry meaning.

Using this cheat sheet, a coding LLM should be able to **iteratively construct a Manim scene**: first setting up the `Scene` class, then adding `Mobjects` (shapes, text, etc.), and finally applying `Animations` to bring the concept to life. By assembling multiple such scenes (for each segment of the explanation) and leveraging camera transitions when needed, the entire explainer video can be generated step by step. Happy animating with Manim! [1] [25]

---

[1] [2] [3] Scene - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.scene.scene.Scene.html

[4] MovingCameraScene - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.scene.moving_camera_scene.MovingCameraScene.html

[5] [6] [7] [35] [36] ThreeDScene - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.scene.three_d_scene.ThreeDScene.html

[8] [9] [10] [37] [38] ZoomedScene - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.scene.zoomed_scene.ZoomedScene.html

[11] [12] [14] [15] [25] [29] [30] [31] [32] [33] Manim's building blocks - Manim Community v0.19.0
https://docs.manim.community/en/stable/tutorials/building_blocks.html

[13] [16] [17] Circle - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.mobject.geometry.arc.Circle.html

[18] [27] [28] [34] TransformMatchingTex - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.animation.transform_matching_parts.TransformMatchingTex.html

[19] [20] Brace - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.mobject.svg.brace.Brace.html

[21] [22] [23] [24] Axes - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.mobject.graphing.coordinate_systems.Axes.html

[26] creation - Manim Community v0.19.0
https://docs.manim.community/en/stable/reference/manim.animation.creation.html