Controltheoylib manual

Jort Stammen

July 10, 2025

Contents

1 Import syntax

To import all functionalities of the library in one line, one can use:

```
from controltheorylib import *
```

Listing 1: Importing all functionalities

However, it is advised to only import the needed functionalities because this can overwrite existing names silently and makes it unclear where functions/classes come from. However, in most cases this should not be a problem. How to only import the needed functionalitites is explained below:

1.1 Importing mechanical systems visualization functions

The mechanical systems visualization functions can be imported from the library by adding the following lines to the start of your `python` script:

```
from manim import *
from controltheorylib import mech_vis
```

Listing 2: Importing mech. system visualization functions

1.2 Importing `PoleZeroMap` class

The `PoleZeroMap` class can be imported from the library by adding the following lines to the start of your `python` script:

```
from manim import *
from controltheorylib import PoleZeroMap
import sympy as sp
```

Listing 3: Importing PoleZeroMap class

1.3 Importing `ControlSystem`

The `ControlSystem` class can be imported from the library by adding the following lines to the start of your `python` script:

```
from manim import *
from controltheorylib import ControlSystem
```

Listing 4: Importing ControlSystem class

1.4 Importing `BodePlot` class

The `BodePlot` class can be imported from the library by adding the following lines to the start of your `python` script:

```
from manim import *
from controltheorylib import BodePlot
```

Listing 5: Importing BodePlot class

1.5 Importing `Nyquist` class

The `Nyquist` class can be imported from the library by adding the following lines to the start of your `python` script:

```
from manim import *
from controltheorylib import Nyquist
```

Listing 6: Importing Nyquist class

2 Mechanical visualization functions

In control theory, we often analyze and design dynamical systems. In mechanical engineering, such systems typically include elements like springs, dampers, and masses. Visualizing how control inputs affect these physical components provides deeper insight into system behavior. However, Manim does not natively support these mechanical components. To address this gap, a set of custom functions as part of the `mech_vis.py` module has been developed to create these elements, enabling the creation of dynamic system visualizations.

2.1 Fixed world function

Fixed world, [Example]

```
fixed_world(start=2*LEFT,end=2*RIGHT, spacing=None, mirror=False, line_or="right", diag_line_length
    =0.3, **kwargs)
```

This function creates a fixed boundary based on the start and end points. This is useful for emphasizing constraints in mechanical systems.
Parameters:

- **start** (*np.ndarray or tuple*) – The start point of the fixed boundary.
- **end** (*np.ndarray or tuple*) – The end point of the fixed boundary.
- **spacing** (*float or None*) - Spacing between the diagonal lines. If `None`, spacing is computed based on total length.
- **mirror** (*bool*) - If `True`, mirrors the diagonal line orientation.
- **line_or** (*string*) - Orientation of diagonal lines: `"right"` (default) or `"left"`.
- **diag_line_length** (*float*) – Length of the diagonal support lines.
- **kwargs** (*Any*) - Additional parameters to be passed to `Line`

Returns:
A `VGroup` object containing the main line and a set of diagonal support lines representing the fixed-world constraint.

2.2 Spring function

Spring, [Example]

```
spring(start=ORIGIN, end3*UP, num_coils = 6, coil_width = 0.4, type = "zigzag", **kwargs)
```

Generates a customizable spring element for illustrating force-transmitting components in mechanical systems. Supports two styles: zigzag and helical.
Parameters:

- **start** (*np.ndarray or tuple*) – The starting point of the spring.
- **end** (*np.ndarray or tuple*) – The endpoint of the spring.
- **num_coils** (*int*) - Number of coils in the spring
- **coil_width** (*float*) - Width of the spring coils
- **type** (*string*) - Spring style: `"zigzag"` or `"helical"`
- **kwargs** (*Any*) - Additional parameters to be passed to `Line`

Returns:

A `VGroup` object representing the spring.

2.3 Mass functions

Mass, [Example]

```
rect_mass(pos=ORIGIN, width=1.5, height=1.5,  label="m", font_size = None, label_color=WHITE, **kwargs)

circ_mass(pos=ORIGIN, radius=1.5, label="m", font_size = None, label_color=WHITE, **kwargs)
```

Creates a simple mass object, either rectangular or circular, with a label at its origin. This is used to visually represent point or distributed mass elements.
Parameters:

- **pos** (*3D vector*) - The center position of the mass.
- **width** (*float*) - width of the rectangular mass
- **height** (*float*) - height of the rectangular mass
- **radius** (*float*) - radius of the circular mass
- **label** (*string*) - The label text inside the shape
- **font_size** (*float or None*) - Font size of the label. If `None`, it scales with the object's size.
- **label_color** (*color*) - Color of the label
- **kwargs** (*Any*) - Additional parameters to be passed to `Rectangle` or `Circle`

Returns:
A Manim `VGroup` object containing the mass shape and its label, centered at the specified position.

2.4 Damper function

Damper, [Example]

```
damper(start=ORIGIN, end=UP*3, width = 0.5, box_height=None, **kwargs)
```

Constructs a damper element commonly used to model energy dissipation in translational systems. The visual consists of a rectangular casing and a movable central rod.
Parameters:

- **start** (*np.ndarray or tuple*) – The start point of the damper.
- **end** (*np.ndarray or tuple*) – The end point of the damper.
- **width** (*float*) - The width of the damper casing
- **box_height** (*float or None*) - The height of the rectangular damper box. If not specified, it is set to half of the total damper length.
- **kwargs** (*Any*) - Additional parameters to be passed to `Line`

Returns:
A Manim `VGroup` object representing the damper, which includes:

- `damper_box`: the stationary rectangular casing.
- `damper_rod`: the dynamic rod.

These parts are returned separately to allow accurate motion simulation. Without this separation, the entire damper would stretch in dynamical animations.

2.5 How to use

When the mechanical visualization functions have been imported, the functions are ready to be used. Consider the following example:

Static example

```
from manim import *
from controltheorylib import mech_vis
config.background_color = "#3d3d3d"

class CoupledSpringDamper(Scene):
    def construct(self):
        # Fixed world
        floor = mech_vis.fixed_world(3.5*LEFT, 3.5*RIGHT, mirror=True, line_or="left").shift(3*DOWN)

        # Masses
        m1 = mech_vis.rect_mass(width=4,height=1.5, label="m_1",color=BLUE).next_to(floor,UP, buff
            =1.5).align_to(floor,LEFT)
        m2 = mech_vis.rect_mass(width=7,height=1.5, label="m_2",color=BLUE).next_to(m1,UP, buff=1.5)
            .align_to(m1,LEFT)

        #springs and their labels
        k1 = mech_vis.spring(start=[-3,-3,0], end=[-3,-1.5,0], coil_width=0.4, num_coils=4)
        k2 = mech_vis.spring(start=[-3,0,0], end=[-3,1.5,0], coil_width=0.4, num_coils=4)
        k3 = mech_vis.spring(start=[3,-3,0], end=[3,1.5,0], coil_width=0.4, num_coils=8)

        k1_label = MathTex("k_1", font_size=35).next_to(k1,LEFT, buff=0.3)
        k2_label = MathTex("k_2", font_size=35).next_to(k2,LEFT, buff=0.3)
        k3_label = MathTex("k_3", font_size=35).next_to(k3,LEFT, buff=0.3)

        springs = VGroup(k1,k2,k3,k1_label,k2_label,k3_label)

        #dampers and their labels
        c1 = mech_vis.damper(start=[-2,-3,0], end=[-2,-1.5,0])
        c2 = mech_vis.damper(start=[-2,0,0], end=[-2,1.5,0])
        c3 = mech_vis.damper(start=[0,-3,0], end=[0,-1.5,0])

        c1_label = MathTex("c_1", font_size=35).next_to(c1,RIGHT, buff=0.2)
        c2_label = MathTex("c_2", font_size=35).next_to(c2,RIGHT, buff=0.2)
        c3_label = MathTex("c_3", font_size=35).next_to(c3,RIGHT, buff=0.2)

        dampers = VGroup(c1,c2,c3,c1_label,c2_label,c3_label)

        #Force arrows
        f1 = Arrow(start=[-0.7,0,0], end=[-0.7,1,0], buff=0)
        f1_label = MathTex("F_1", font_size=35).next_to(f1, RIGHT, buff=0.1)

        f2 = Arrow(start=[1,1.5,0], end=[1,0.5,0], buff=0)
        f2_label = MathTex("F_2", font_size=35).next_to(f2, RIGHT, buff=0.1)

        forces = VGroup(f1,f2,f1_label,f2_label)

        #x1,x2
        x1_line = Line(start=[0.3,-0.75,0], end=[0.6,-0.75,0])
        x1_arrow = Arrow(start=x1_line.get_end(), end=x1_line.get_end()+0.7*UP, buff=0, stroke_width
            =8)
        x1_label = MathTex("x_1", font_size=35).next_to(x1_arrow,buff=0.2)

        x2_line = Line(start=[3.3,2.25,0], end=[3.6,2.25,0])
        x2_arrow = Arrow(start=x2_line.get_end(), end=x2_line.get_end()+0.7*UP, buff=0, stroke_width
            =8)
        x2_label = MathTex("x_2", font_size=35).next_to(x2_arrow,buff=0.2)

        position = VGroup(x1_line,x1_arrow,x1_label,x2_line,x2_arrow,x2_label)

        self.add(floor, m1, m2,springs,dampers,forces,position)
```

The static mobjects can be added staticaly to the scene using `self.add()`.

If you would like to animate the movement of the mobjects, one should use the `self.play()` command. To update the position of the mobjects such as spring, one could use updater functions. Consider the following example:

Oscillating mass example, [output]

```python
from manim import *
from controltheorylib import mech_vis
import numpy as np

class MassSpringSys(Scene):
    def construct(self):

        #Parameters
        m = 1       # mass
        k = 100     # spring constant
        c = 1.5      # damping coefficient
        omega = np.sqrt(k/m)
        zeta = c/(2*np.sqrt(k*m))
        omega_d = omega*np.sqrt(1-zeta**2)
        A = 2       # amplitude
        phi = 0     # phase
        t_end = 6/(zeta*omega) if zeta > 0 else 8

        #Create fixed world
        fixed = mech_vis.fixed_world([-5,3.5,0],[-1,3.5,0])

        #Spring and damper start and equillibrium positions
        spring_start = [-4, 3.5, 0]
        spring_eq = [-4, 0.5, 0]
        damper_start = [-2.5, 3.5, 0]
        damper_eq = [-2.5, 0.5, 0]

        #Create spring and damper
        spring = mech_vis.spring(spring_start,spring_eq)
        damper_box, damper_rod = mech_vis.damper(damper_start,damper_eq)

        #Create spring and damper labels
        k = MathTex("k").next_to(spring,LEFT, buff=0.5)
        c = MathTex("c").next_to(damper_rod,RIGHT, buff=0.5)

        #Create mass
        mass_size = 2
        mass_x = (spring_eq[0] + damper_eq[0])/2
        mass_y_eq = spring_eq[1] - mass_size/2
        mass = mech_vis.mass([mass_x,mass_y_eq,0], size=mass_size)

        #Create axis for displacement plot
        axis =  Axes(x_range=[0,t_end,1], y_range=[-A,A,0.5], x_length=6, y_length=6, axis_config={"color": WHITE})
        axis.move_to([3,mass_y_eq,0])

        # Add axis labels
        axis_labels = axis.get_axis_labels(x_label=MathTex("t"), y_label=MathTex("y"))

        time = ValueTracker(0)
        def displacement(t):
            return mass_y_eq + A*np.exp(-zeta*omega*t)*np.cos(omega_d*t+phi)

        def oscillator(mob):
            t = time.get_value()
            y = displacement(t)
            spring_end = [spring_start[0], y+mass_size/2, 0]
            damper_end = [damper_start[0], y+mass_size/2, 0]

            # Update mass
            mass.move_to([mass_x, y, 0])

            # Update spring and damper rod
            spring.become(mech_vis.spring(spring_start, spring_end))
            damper_rod.become(mech_vis.damper(damper_start, damper_end)[1])

            # Update labels
            k.next_to(spring, LEFT, buff=0.5)
            c.next_to(damper_rod, RIGHT, buff=0.5).shift(0.5*UP)

        dot = Dot(color=YELLOW)
        def update_dot(mob):
            t = time.get_value()
            y_disp = displacement(t)-mass_y_eq
            mob.move_to(axis.c2p(t, y_disp))
        dot.add_updater(update_dot)

        graph_points = []
```

```
        def update_trace():
            t = time.get_value()
            y_disp = displacement(t)-mass_y_eq
            graph_points.append(axis.c2p(t, y_disp))
            if len(graph_points) < 2:
                return VGroup()
            return VMobject().set_points_smoothly(graph_points).set_stroke(YELLOW, width=2)
        trace = always_redraw(update_trace)

        mass.add_updater(oscillator)
        spring.add_updater(oscillator)
        damper_rod.add_updater(oscillator)

        self.add(fixed,spring,damper_box, damper_rod,mass,k,c, axis, axis_labels, dot, trace)
        self.play(time.animate.set_value(t_end), run_time=t_end, rate_func=linear)
```

Here we used the analytical solution to define the dynamics of the problem. However, for more complex problems one must use numerical tools to solve the equation of motion using, for instance, the `solve.ivp` function from the `scipy.integrate` module.

3 PoleZeroMap class

The `PoleZeroMap` class is designed to facilitate the visualization of pole-zero maps. This class supports both continuous- and discrete-time systems and is built on top of the Manim animation library to provide animated, highly customizable maps.

3.1 Defining sysem transfer function

An essential input to the `PoleZeroMap` class is the system transfer function: $H(s) = \frac{\mathtt{num}(s)}{\mathtt{den}(s)}$ for continuous-time systems (Laplace domain) and $H(z) = \frac{\mathtt{num}(z)}{\mathtt{den}(z)}$ for discrete-time systems (Z-domain). To automate the configuration of the pole-zero map, the class includes an internal method to identify whether the system being analyzed is continuous-time (Laplace domain) or discrete-time (Z-domain). This is determined by inspecting the symbolic content of the provided transfer function's numerator and denominator.
If the symbol `s` appears in the numerator and/or denominator as a symbolic expression or string, the system is classified as *continuous-time*. Whereas, if the symbol `z` appears in the numerator and/or denominator as a symbolic expression or string, the system is classified as *discrete-time*. If the system is defined using transfer function coefficients or `Scipy` LTI objects, it is *assumed* that the system is a continuous-time system. This is illustrated in the following example:

System type determination

```
z = sp.symbol('z') # Define 'z' as symbolic expression

system = ("(s-1)/((s+2)*(s-6))") # continuous-time (string)
system1 = ([1],([1,0.2,1]) # continuous-time (coefficients)
system2 = (10/(z**2+0.25)) # discrete-time (symbolic expression)
system3 = ("(z-2)/((z+0.8)*(z-4)") # discrete-time (string)
```

This automatic determination of system type simplifies user interaction, allowing for general-purpose use of the class without requiring manual specification of the system domain.

3.2 Creating pole-zero map attributes

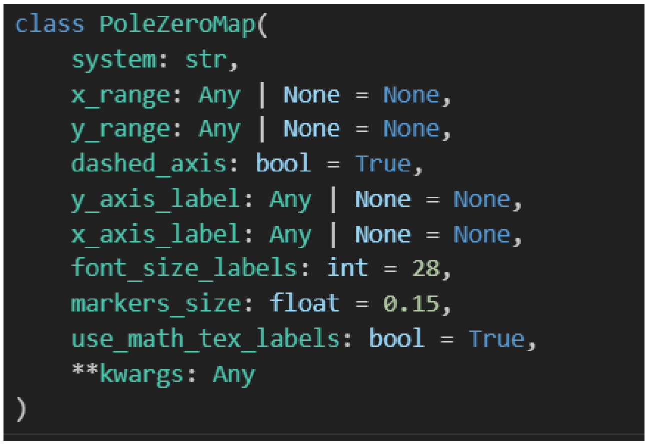
After the system transfer function has been defined, one can create the pole-zero map via calling the `PoleZeroMap` class with `num` and `den` as inputs. Take the following example:

```
# 'variablename' can take any valid variable name.
variablename = PoleZeroMap(system)

# for instance pzmap
pzmap = PoleZeroMap(num,den)
```

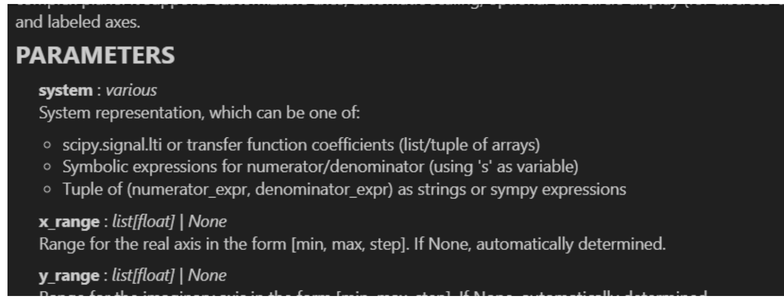
Listing 7: Constructing pole zero map

To see the other `__init__` constructor inputs, one can pan or hold their cursor over PoleZeroMap. a list of inputs will pop up. When scrolled down, all the relavant input parameters are explained.



```
class PoleZeroMap(
    system: str,
    x_range: Any | None = None,
    y_range: Any | None = None,
    dashed_axis: bool = True,
    y_axis_label: Any | None = None,
    x_axis_label: Any | None = None,
    font_size_labels: int = 28,
    markers_size: float = 0.15,
    use_math_tex_labels: bool = True,
    **kwargs: Any
)
```

Figure 1: Input list



and labeled axes.

PARAMETERS

system : *various*
System representation, which can be one of:

- scipy.signal.lti or transfer function coefficients (list/tuple of arrays)
- Symbolic expressions for numerator/denominator (using 's' as variable)
- Tuple of (numerator_expr, denominator_expr) as strings or sympy expressions

x_range : *list[float] | None*
Range for the real axis in the form [min, max, step]. If None, automatically determined.

y_range : *list[float] | None*

Figure 2: Parameters explanation

Additionally, one can add a title or stability regions to the plot. This is done *after* the creation of the standard pole-zero map attributes.

Adding a title to the plot

One can add a title to the plot using the `title` function. Take the following example:

```
...
pzmap = PoleZeroMap(system)
pzmap.title("Pole-zero map")
```

Listing 8: Adding title to the plot

Additional inputs to the title function can be found using the same method as discussed in Figure 1 and Figure 2.

Adding the stability regions

One can add the stability regions of the pole-zero map using the `add_stability_regions` function. Take the following example:

```
...
pzmap = PoleZeroMap(system)
pzmap.add_stability_regions()
```

Listing 9: Adding stability regions

Once again, Additional inputs to the add_stability_regions function can be found using the same method as discussed in Figure 1 and Figure 2.

3.3 Plotting the pole-zero map

After the pole-zero map attributes have been created according to the specified inputs, one can create a static pole-zero map plot using `self.add()`. Take the following example:

```
...
pzmap = PoleZeroMap(system)
pzmap.add_stability_regions() #optional
self.add(pzmap) #adds the pole-zero map to the scene
```

Listing 10: Adding the pole-zero map to the scene

3.4 Animating the plot components step-by-step

Instead of adding the whole pole-zero map staticly to the scene, one can create custom animations of all the plot components of the pole-zero map. This can be done by using the `self.play()` command. Take the following example:

```
...
pzmap = PoleZeroMap(system)
pzmap.add_stability_regions() #optional

# Animate the plot components step-by-step
self.play(Create(pzmap.surrbox), Create(pzmap.dashed_x_axis),Create(pzmap.
    dashed_y_axis))
self.wait(0.5)
self.play(Create(pzmap.x_ticks), Create(pzmap.y_ticks))
self.wait(0.5)
self.play(Write(pzmap.x_tick_labels), Write(pzmap.y_tick_labels))
self.wait(0.5)
self.play(Write(pzmap.title_text))
self.wait(0.5)
self.play(Create(pzmap.unit_circle))
self.wait(0.5)
self.play(Create(pzmap.stable_region), Write(pzmap.text_stable))
self.wait(1)
self.play(Create(pzmap.unstable_region), Write(pzmap.text_unstable))
self.wait(1)
self.play(GrowFromCenter(pzmap.zeros), GrowFromCenter(pzmap.poles))
self.wait(2)
```

Listing 11: Animating pole-zero map example

Any built-in Manim animation class can be used to animate the components.

3.5 Component reference

The list of components which can be animated are tabulated below:

Table 1: Components of the PoleZeroMap Class

Component	Description
zeros	Blue circles representing the zeros of the transfer function in the complex plane.
poles	Red crosses representing the poles of the transfer function in the complex plane.
stable	Highlighted region (blue by default) indicating the stable region of the system *and* stable label
unstable	Highlighted region (red by default) indicating the unstable region of the system *and* unstable label
stable_region	Highlighted region (blue by default) indicating the stable region of the system.
unstable_region	Highlighted region (red by default) indicating the unstable region of the system
text_stable	stable label
text_unstable	unstable label
unit_circle	Unit circle displayed *only* for discrete-time systems.
axis_labels	Labels for the real and imaginary axes.
title_text	Optional title text that can be added above the plot.
box	White rectangular border surrounding the entire plot area.
x_axis	white line representing the real axis (x-axis).
y_axis	white line representing the imaginary axis (y-axis).
x_ticks	Tick marks along the x-axis (both top and bottom of the plot).
y_ticks	Tick marks along the y-axis (both left and right sides of the plot).
x_tick_labels	Numerical labels for the x-axis ticks, positioned below the plot.
y_tick_labels	Numerical labels for the y-axis ticks, positioned to the left of the plot.

3.6 Tranistioning between pole-zero maps

One can use the `Transform()` or `ReplacementTransform()` animation command to show how the pole and zero locations transform between different transfer functions. *Tip*: Make sure to set the ranges (if desired) to predefined ranges; otherwise, the auto-range determination will change the ranges, resulting in different sizes for certain plot components. Take the following example, where we aim to explain how the locations of the poles and zeros change between two transfer functions:

```

        pzmap1 = PoleZeroMap(("(s+1)/(s**2+0.2*s+5)"), x_range=[-4,3,1], y_range
            =[-3,3,1])
        pzmap2 = PoleZeroMap(("(s-1)/((s+3)*(s-2))"), x_range=[-4,3,1], y_range
            =[-3,3,1])

        pzmap1.title(r"H(s)=\frac{s+1}{s^2+0.2s+5}", use_math_tex=True, font_size
            =25)
        pzmap2.title(r"H(s)=\frac{s-1}{(s+3)(s-2))}", use_math_tex=True, font_size
            =25)

        # Adds first pzmap to the scene
        self.add(pzmap1)
        self.wait(2) #wait 2 seconds

        #Fadeout the first TF and write the second TF
        self.play(FadeOut(pzmap1.title_text), Write(pzmap2.title_text))
        self.wait(1)

        # Transition the pole and zero locations
```

```
        self.play(Transform(pzmap1.zeros, pzmap2.zeros), Transform(pzmap1.poles,
            pzmap2.poles))
```

Listing 12: Animating pole-zero map example

4 ControlSystem class

The ControlSystem class, along with its constituent ControlBlock, Connection, and Disturbance classes, forms a modular and extensible library for generating animated block diagrams of control systems within the Manim environment.

4.1 Initiating a ControlSystem

Creating a control system diagram starts with initiating the system. Consider the following example:

initiating control system

```
from manim import *
from controltheorylib import ControlSystem

class ControlSystemScene(Scene):
    def construct(self):

        # Initiate controlsystem
        cs = ControlSystem() # "cs" can be any valid variable name
```

Here we initiate the control system, in this example we call it cs but this can be any valid variable name. Tip: keep the name concice as this name will be used a lot further in the code.

4.2 Creating blocks, inputs and outputs

Blocks can be created using the `.add_block()` command. A ControlBlock is instantiated with a name for easy reference, a defined block_type (which dictates its visual form and inherent functionality), and a specified position. A block can be one of two types: "transfer_function" or "summing_junction". A "transfer_function" type will result in a rectangular shaped block while a summing junction will be a circular shaped block. The blocks can be customized using the "param" input. nternally, ControlBlock manages its connection points through input_ports and output_ports, which are essentially invisible objects strategically placed on the block's edges.

Additionally one can create inputs using `add_input` or outputs using `add_output`. For inputs, a target_block, and its accompanying input_port should be specified. For outputs, a source_block, and its accompanying output_port should be specified.

Additionally, one can create a disturbance. Similar to the inputs, a target_block, and its accompanying input_port should be specified. Take the following example:

Block creation

```
from manim import *
from controltheorylib import ControlSystem

class ControlSystemScene(Scene):
    def construct(self):

        # Initiate controlsystem
        cs = ControlSystem()

        # Create blocks
        sum_block1 = cs.add_block("sum1", "summing_junction", 4*LEFT, params={"input1_dir": LEFT, "
            input2_dir": DOWN, "input2_sign": "-", "input1_sign": "+","fill_opacity": 0})
        ref = cs.add_input(sum_block1, "in1", label_tex=r"r(s)")
        controller = cs.add_block(r"controller", "transfer_function", 1.5*LEFT, {"use_mathtex":True,
            "font_size":50,"label":r"K_p(1+Ds)"})
        sum_block2 = cs.add_block("sum2", "summing_junction", RIGHT, params={"input1_dir": LEFT, "
            input2_dir": UP, "output1_dir": RIGHT, "output2_dir":DOWN,"input2_sign": "+", "
            input1_sign": "+", "fill_opacity":0})
        plant = cs.add_block("Plant", "transfer_function", RIGHT*3.5, {"text_font_size":40, "label":
            "Plant"})
        output = cs.add_output(plant, "out", label_tex=r"y(s)")
```

The blocks can be connected to each other using the `.connect()` method. A connection is created by specifying the source_block and the desired output_port, along with the dest_block and its corresponding input_port. The Connection class then automatically renders an arrow from the source port to the destination port. Elaborating further on the previous example:

Connecting the blocks

```
from manim import *
from controltheorylib import ControlSystem

class ControlSystemScene(Scene):
    def construct(self):

        # Initiate controlsystem
        cs = ControlSystem()

        # Create blocks
        sum_block1 = cs.add_block("", "summing_junction", 4*LEFT, params={"input1_dir": LEFT, "input2_dir": DOWN, "input2_sign": "-", "input1_sign": "+","fill_opacity": 0})
        ref = cs.add_input(sum_block1, "in1", label_tex=r"r(s)")
        controller = cs.add_block(r"K_p(1+Ds)", "transfer_function", 1.5*LEFT, {"use_mathtex":True,"font_size":50,"label":r"K_p(1+Ds)"})
        sum_block2 = cs.add_block("", "summing_junction", RIGHT, params={"input1_dir": LEFT, "input2_dir": UP, "output1_dir": RIGHT, "output2_dir":DOWN,"input2_sign": "+", "input1_sign": "+", "fill_opacity":0})
        plant = cs.add_block("Plant", "transfer_function", RIGHT*3.5, {"text_font_size":40, "label":"Plant"})
        output = cs.add_output(plant, "out", label_tex=r"y(s)")
        feedback = cs.add_feedback_path(plant, "out", sum_block1, "in2")

        #Connect
        conn1 = cs.connect(sum_block1, "out1", controller, "in", label_tex=r"e(s)")
        conn2 = cs.connect(controller, "out", sum_block2, "in1")
        conn3 = cs.connect(sum_block2, "out1", plant, "in")

        # Add disturbance
        disturbance = cs.add_disturbance(sum_block2, "in2", label_tex=r"d(s)", position="top")
```

If desired, one can create a feedback or feedforward connection using the `add_feedback_path` or `add_feedforward_path`. Once again, a connection is created by specifying the source_block and the desired output_port, along with the dest_block and its corresponding input_port.

4.3 Adding to the scene

To add all components to the scene using `self.add()` one can use the `get_all_components()` function. This way, the user does not have to manually add each and every component to the scene. Take the following fully completed example:

f

```
rom manim import *
from controltheorylib import ControlSystem

class ControlSystemScene(Scene):
    def construct(self):

        # Initiate controlsystem
        cs = ControlSystem()

        # Create blocks
        sum_block1 = cs.add_block("", "summing_junction", 4*LEFT, params={"input1_dir": LEFT, "input2_dir": DOWN, "input2_sign": "-", "input1_sign": "+","fill_opacity": 0})
        ref = cs.add_input(sum_block1, "in1", label_tex=r"r(s)")
        controller = cs.add_block(r"K_p(1+Ds)", "transfer_function", 1.5*LEFT, {"use_mathtex":True,"font_size":50,"label":r"K_p(1+Ds)"})
        sum_block2 = cs.add_block("", "summing_junction", RIGHT, params={"input1_dir": LEFT, "input2_dir": UP, "output1_dir": RIGHT, "output2_dir":DOWN,"input2_sign": "+", "input1_sign": "+", "fill_opacity":0})
        plant = cs.add_block("Plant", "transfer_function", RIGHT*3.5, {"text_font_size":40, "label":"Plant"})
        output = cs.add_output(plant, "out", label_tex=r"y(s)")
        feedback = cs.add_feedback_path(plant, "out", sum_block1, "in2")

        #Connect
        conn1 = cs.connect(sum_block1, "out1", controller, "in", label_tex=r"e(s)")
        conn2 = cs.connect(controller, "out", sum_block2, "in1")
```

```
        conn3 = cs.connect(sum_block2, "out1", plant, "in")

        # Add disturbance
        disturbance = cs.add_disturbance(sum_block2, "in2", label_tex=r"d(s)", position="top")

        # get all components
        diagram = cs.get_all_components()

        # add diagram to scene
        self.add(diagram)
```

4.4 Animating individual components

The ControlSystem class is designed such that its component creation methods, such as add_block, connect, add_input, add_output, add_disturbance, add_feedback_path, and add_feedforward_path, return the corresponding Manim Mobject(s). This way, it allows intuitive animation of the individual components of the control system. Namely, by using the following syntax; self.play(Animationfunction(MobjectName)). Here MobjectName is the variable name the user has given the Mobject and Animationfunction is the animation function the user would like to use to animate the Mobject, such as FadeIn or Create. Take the same example, but now we animate the individual components instead of adding it all at once to the scene:

Animating individual components

```
from manim import *
from controltheorylib import ControlSystem

class Animation_Example1(Scene):
    def construct(self):

        # Initiate controlsystem
        cs = ControlSystem()

        # Create blocks
        sum_block1 = cs.add_block("", "summing_junction", 4*LEFT, params={"input1_dir": LEFT, "input2_dir": DOWN, "input2_sign": "-", "input1_sign": "+","fill_opacity": 0})
        ref = cs.add_input(sum_block1, "in1", label_tex=r"r(s)")
        controller = cs.add_block(r"K_p(1+Ds)", "transfer_function", 1.5*LEFT, {"use_mathtex":True,"font_size":50,"label":r"K_p(1+Ds)"})
        sum_block2 = cs.add_block("", "summing_junction", RIGHT, params={"input1_dir": LEFT, "input2_dir": UP, "output1_dir": RIGHT, "output2_dir":DOWN,"input2_sign": "+", "input1_sign": "+", "fill_opacity":0})
        plant = cs.add_block("Plant", "transfer_function", RIGHT*3.5, {"text_font_size":40, "label":"Plant"})
        output = cs.add_output(plant, "out", label_tex=r"y(s)")
        feedback = cs.add_feedback_path(plant, "out", sum_block1, "in2")


        #Connect
        conn1 = cs.connect(sum_block1, "out1", controller, "in", label_tex=r"e(s)")
        conn2 = cs.connect(controller, "out", sum_block2, "in1")
        conn3 = cs.connect(sum_block2, "out1", plant, "in")

        # Add disturbance
        disturbance = cs.add_disturbance(sum_block2, "in2", label_tex=r"d(s)"
                                         , position="top")

        # add diagram to scene
        self.play(Create(ref), run_time=0.5)
        self.wait(0.1)
        self.play(FadeIn(sum_block1), run_time=0.5)
        self.wait(0.1)
        self.play(Create(conn1),run_time=0.5)
        self.wait(0.1)
        self.play(FadeIn(controller))
        self.wait(0.1)
        self.play(Create(conn2), run_time=0.5)
        self.wait(0.1)
        self.play(FadeIn(sum_block2),run_time=0.5)
        self.wait(0.1)
        self.play(Create(disturbance), Create(conn3), run_time=0.5)
        self.wait(0.1)
        self.play(FadeIn(plant))
        self.wait(0.1)
        self.play(Create(output), Create(feedback))
```

```
        title=Text("Feedback loop", font_size=30).move_to(ORIGIN+3*UP)
        self.play(Write(title))
        self.wait(2)
```

4.5 Animating signal flow

5 BodePlot class

The BodePlot class provides comprehensive visualization of Bode plots (magnitude and phase frequency responses) for both continuous- and discrete-time systems. Built on Manim, it supports extensive customization and animation capabilities.

5.1 Defining the system transfer function

The system can be specified in several formats:

- Scipy LTI objects (TransferFunction, ZerosPolesGain, StateSpace)
- Tuple of numerator and denominator coefficients (arrays/lists)
- Symbolic expressions using 's' or 'z' variables
- String representations of transfer functions (e.g., "(s+1)/(s^2+2*s+1)")

```
# From scipy LTI object
sys = signal.TransferFunction([1], [1, 2, 1])
bode = BodePlot(sys)

# From coefficients
bode = BodePlot(([1], [1, 2, 1]))

# From symbolic expressions
s = sp.symbols('s')
bode = BodePlot(s+1, s**2 + 2*s + 1)

# From string
bode = BodePlot("(s+1)/(s**2+2*s+1)")
```

Listing 13: Creating BodePlot with different system specifications

The system input is the only required input. Additional inputs to the BodePlot class can be found using the same method as discussed in Figure 1 and Figure 2.

5.2 Customizing plot elements

Similar to the pole-zero map, additional attributes can be created *after* the creation of the standard attributes.

Adding a title

A title can be added to the Bode plot using the title() function

```
...
system = ...
bode = BodePlot(system, ..)
bode.title("Second␣Order␣System", font_size=30, color=WHITE)
```

Listing 14: Adding a title

Showing/hiding components

Both the magnitude and phase plots are plotted by default. To hide them, one can use the show_magnitudes or show_phase function to set the Boolean to false. This will hide the magnitude or phase plot. Additionally, one can add grid lines using the grid_on function. To turn the grid back off, one can use the grid_off function or just simply remove the line where the grid is turned on.

```
bode.show_magnitude(False)  # Hide magnitude plot
bode.show_phase(False)      # Hide phase plot
bode.grid_on()              # Show grid lines
bode.grid_off()             # Hides the grid lines
```

Listing 15: Controlling plot visibility

Adding stability margins

Stability margins such as the phase margin and gain margin can be visualized using the `show_margins` function.

```
bode.show_margins(
    show_values=True,
    margin_color=YELLOW,
    text_color=WHITE,
    font_size=24
)
```

Listing 16: Showing stability margins

Showing asymptotes

The asymptotes of the Bode plot can be plotted using the `show_margins` function. Take the following example

```
..
bode = BodePlot(system, ..)
bode.show_asymptotes(
    color=YELLOW,
    stroke_width=2,
    opacity=0.7
)
```

Listing 17: Adding asymptotes

5.3 Plotting and animation

Static plotting

The Bode plot attributes can be added statically to the scene using the `self.add()` command.

```
..
bode = BodePlot(system, ..) # Define main attributes
bode.show_asymptotes(color=YELLOW,
stroke_width=2,
opacity=0.7) #Define additional attributes (optional)
self.add(bode)  # Add bode plot attributes
```

Listing 18: Adding to scene

Component-wise animation

Similar to the pole-zero map, each plot component can be animated in any arbitrary order.

```
...
bode = BodePlot(system, ..)

# Animate axes and boxes
self.play(
    Create(bode.mag_box),
    Create(bode.phase_box),
    Create(bode.mag_ticks),
    Create(bode.phase_ticks)

# Animate plots
self.play(Create(bode.mag_plot))
self.play(Create(bode.phase_plot))
```

```

# Add labels
self.play(
    Write(bode.mag_ylabel),
    Write(bode.phase_ylabel),
    Write(bode.freq_xlabel)
)
```

Listing 19: Animating components

5.4 Component reference

The list of individual components which can be animated is tabulated below:

Table 2: Components of the BodePlot Class

Component	**Description**
mag_plot	Magnitude frequency response curve
phase_plot	Phase frequency response curve
mag_axes	Magnitude plot axes
phase_axes	Phase plot axes
mag_box	White bounding box for magnitude plot
phase_box	White bounding box for phase plot
mag_yticks	Horizontal tick marks for magnitude plot
phase_yticks	Horizontal tick marks for phase plot
mag_xticks	Vertical tick marks for magnitude plot
phase_xticks	Vertical tick marks for phase plot
mag_yticklabels	Magnitude axis tick labels
phase_yticklabels	Phase axis tick labels
mag_ylabel	"Magnitude (dB)" label
phase_ylabel	"Phase (deg)" label
freq_xlabel	"Frequency (rad/s)" label
freq_ticklabels	Frequency tick labels (10^n)
mag_hor_grid	Horizontal grid lines for magnitude plot
phase_hor_grid	Horizontal grid lines for phase plot
mag_vert_grid	Vertical grid lines for magnitude plot
phase_vert_grid	Vertical grid lines for phase plot
mag_asymp_plot	Magnitude asymptotes (when shown)
phase_asymp_plot	Phase asymptotes (when shown)
title_text	Plot title (when added)
zerodB_line	Horizontal zero dB line for magnitude plot (if show_margins)
minus180deg_line	Horizontal minus 180 degree line for phase plot (if show_margins)
vert_gain_line	Vertical gain line in phase plot indicating gain crossover frequency (if show_margins)
gm_dot	Dot indicating the gain crossover frequency (if show_margins)
gm_vector	Vector indicating the size of the gain margin (if show_margins)
gm_text	Label to the side of gm vector indicating size of the gain margin (if show_margins)
vert_phase_line	Vertical phase line in phase plot indicating phase crossover frequency (if show_margins)
pm_dot	Dot indicating the phase crossover frequency (if show_margins)
pm_vector	Vector indicating the size of the phase margin (if show_margins)
pm_text	Label to the side of pm vector indicating size of the gain margin (if show_margins)

5.5 Transitioning between Bode plots

One can use the `Transform()` animation command to show how, for instance, a system reacts to certain controllers (how adjusting P-gain affects magnitude plot etc.). *Tip*: Make sure to set the ranges (if desired) to predefined ranges; otherwise, the auto-range determination will change the ranges, resulting in different sizes for certain plot components. Take the following (more complex) example.

```python
from manim import *
from controltheorylib.control import BodePlot
import sympy as sp

class Bode(Scene):
    def construct(self):

        # Define first bode plot
        s = sp.symbols('s')
        num1 = 1
        den1 = (s+2)*(s+10)*(s+15)
        system1 = (num1, den1)

        bode1 = BodePlot(system1, magnitude_yrange=[-200,25], phase_yrange=[-270,0], freq_range=[0.1,1000])
        bode1.grid_on()

        # Animate the first bode plot
        self.play(Create(bode1.mag_box),Create(bode1.phase_box))
        self.wait(0.5)
        self.play(Create(bode1.mag_yticks),Create(bode1.mag_xticks), Create(bode1.phase_yticks),Create(bode1.phase_xticks))
        self.wait(0.5)
        self.play(Write(bode1.mag_yticklabels),Write(bode1.phase_yticklabels), Create(bode1.freq_ticklabels))
        self.wait(0.5)
        self.play(Write(bode1.mag_ylabel),Write(bode1.phase_ylabel), Create(bode1.freq_xlabel))
        self.wait(0.5)
        self.play(Create(bode1.mag_vert_grid),Create(bode1.mag_hor_grid), Create(bode1.phase_vert_grid),Create(bode1.phase_hor_grid))
        self.wait(0.5)
        self.play(Create(bode1.mag_plot),Create(bode1.phase_plot))
        self.wait(2)

        #Show the two Transfer functions
        text1 = MathTex(r"H(s)=\frac{1}{(s+2)(s+10)(s+15)}", font_size=35).next_to(bode1.mag_box, UP, buff=0.3)
        self.play(Write(text1))
        self.wait(0.5)
        text2 = MathTex(r"H(s) = \frac{1500}{(s+2)(s+10)(s+15)}", font_size=35).move_to(text1)
        self.play(ReplacementTransform(text1, text2))
        num2 = 1500
        den2 = (s+2)*(s+10)*(s+15)
        system2 = (num2, den2)

        # Define second bode plot
        bode2 = BodePlot(system2,magnitude_yrange=[-200,25], phase_yrange=[-270,0], freq_range=[0.1,1000])
        bode2.grid_on()

        # Calculate the Bode data for the second system
```

```
46         bode2.calculate_bode_data()
47         bode2.plot_bode_response()
48 
49         target_freq = 1.0  # 10^0 = 1 rad/s
50         freq_idx = np.argmin(np.abs(np.array(bode1.frequencies) - target_freq))
51         freq = bode1.frequencies[freq_idx]
52         log_freq = np.log10(freq)
53 
54         # Get the points for both plots at this frequency
55         mag1_point = bode1.mag_axes.coords_to_point(log_freq, bode1.magnitudes[freq_idx])
56         mag2_point = bode1.mag_axes.coords_to_point(log_freq, bode2.magnitudes[freq_idx])
57 
58         # Create an arrow pointing from bode1 to bode2
59         arrow = Arrow(start=mag1_point,end=mag2_point,
60             color=YELLOW,buff=0,
61             stroke_width=4,tip_length=0.2)
62         delta_db = bode2.magnitudes[freq_idx] - bode1.magnitudes[freq_idx]
63         arrow_label = MathTex(fr"\Delta|H| = {delta_db:.1f}\,dB", font_size=24)
64         arrow_label.next_to(arrow, RIGHT, buff=0.1)
65 
66         # Transform the first plot into the second plot
67         self.play(
68             Transform(bode1.mag_plot, bode2.mag_plot),
69             Transform(bode1.phase_plot, bode2.phase_plot),
70             GrowArrow(arrow),
71             FadeIn(arrow_label),
72             run_time=2)
73         self.wait(2)
```

Listing 20: Transitioning between bode plots example

Note, the transforming boils down to lines 62-68. Copy-paste the code to see how it works and try to see what happens when you change stuff.

6 Nyquist class

The `Nyquist` class provides visualization of Nyquist plots for both continuous- and discrete-time systems. Built on Manim, it supports extensive customization and animation capabilities, including stability margin visualization and unit circle display.

6.1 Defining the system transfer function

The system can be specified in several formats:

- Scipy LTI objects (`TransferFunction`, `ZerosPolesGain`, `StateSpace`)
- Tuple of numerator and denominator coefficients (arrays/lists)
- Symbolic expressions using 's' variable
- String representations of transfer functions (e.g., `"(s+1)/(s^2+2*s+1)"`)

```
# From scipy LTI object
sys = signal.TransferFunction([1], [1, 2, 1])
nyquist = Nyquist(sys)

# From coefficients
nyquist = Nyquist(([1], [1, 2, 1]))

# From symbolic expressions
s = sp.symbols('s')
nyquist = Nyquist(s+1, s**2 + 2*s + 1)

# From string
nyquist = Nyquist("(s+1)/(s^2+2*s+1)")
```

Listing 21: Creating Nyquist plot with different system specifications

6.2 Customizing plot elements

Similar to the pole-zero map and bode plot, additional attributes can be created *after* the creation of the standard attributes.

Adding a title

A title can be added to the Nyquist plot using the `title()` function.

```
nyquist = Nyquist(system)
nyquist.title("Second Order System", font_size=30, color=WHITE)
```

Listing 22: Adding a title

Grid

The grid can be turned on and off using the `grid_on` and `grid_off` functions.

```
nyquist.grid_on()              # Show grid lines
nyquist.grid_off()             # Hide grid lines
```

Listing 23: Controlling plot visibility

Showing stability margins

Phase margin, gain margin, and modulus margin can be visualized:

```
nyquist.show_margins(
    pm_color=YELLOW,           # Phase margin color
    mm_color=ORANGE,           # Modulus margin color
    gm_color=GREEN_E,          # Gain margin color
    font_size=18,              # Label font size
```

```
    show_pm=True,              # Show phase margin
    show_gm=True,              # Show gain margin
    show_mm=True               # Show modulus margin
)
```

Listing 24: Showing stability margins

6.3 Plotting and animation

Static plotting

The Nyquist plot can be added statically to the scene:

```
nyquist = Nyquist(system)
self.add(nyquist)  # Add all components at once
```

Listing 25: Adding to scene

Component-wise animation

Individual components can be animated separately:

```
# Animate axes and grid
self.play(
    Create(nyquist.plane),
    Create(nyquist.grid_lines),
    Create(nyquist.unit_circle)
)

# Animate Nyquist curve
self.play(Create(nyquist.nyquist_plot))

# Add labels
self.play(
    Write(nyquist.x_label),
    Write(nyquist.y_label)
)
```

Listing 26: Animating components

6.4 Component reference

The list of individual components which can be animated is tabulated below:

Table 3: Components of the Nyquist Class

Component	Description
box	White bounding box
plane	Complex plane axes
nyquist_plot	Nyquist curve (positive frequencies)
x_axislabel	Real axis label
y_axislabel	Imaginary axis label
x_ticks	Tick marks on real axis
y_ticks	Tick marks on imaginary axis
x_ticklabels	Real axis tick labels
y_ticklabels	Imaginary axis tick labels
x_axis	Dashed Real axis
y_axis	Dashed Imaginary axis
grid_lines	Grid lines (circles and radial lines)
unit_circle	Unit circle (when shown)
minus_one_marker	Marker at (-1,0) point
minus_one_label	Label at (-1,0) point
title_text	Plot title (when added)
margin_indicators	Group containing all margin indicators
pm_dot	Phase margin point marker
pm_label	Phase margin label
pm_arc	Phase margin arc
gm_line	Gain margin line
gm_label	Gain margin label
mm_line	Modulus margin line
mm_label	Modulus margin label
mm_circle	Modulus margin circle

6.5 Transitioning between Nyquist plots

The `Transform()` command can be used to animate between different Nyquist plots:

```
# Create initial plot
sys1 = signal.TransferFunction([1], [1, 1])
nyquist1 = Nyquist(sys1)

# Create modified plot
sys2 = signal.TransferFunction([2], [1, 1])
nyquist2 = Nyquist(sys2)

# Animate transition
self.play(Transform(nyquist1.nyquist_plot, nyquist2.nyquist_plot))
```

Listing 27: Transitioning example