Controltheoylib manual

Jort Stammen

June 5, 2025

Contents

1 Import syntax

When the library has been installed succesfully, one can import the libraries functions and classes. The import syntax depends on which functions or classes you would like to use.

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 control
```

Listing 1: 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.control import PoleZeroMap
import sympy as sp
```

Listing 2: Importing PoleZeroMap class

The third line allows the use for using symbolic expressions for the systems transfer function. This is **optional** for most plotting tools. However, it is required for the `PoleZeroMap` class to define whether the system is continuous or discrete, see .. for more information.

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.control import ControlSystem
```

Listing 3: 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.control import BodePlot
import sympy as sp #optional
```

Listing 4: 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.control import Nyquist
import sympy as sp #optional
```

Listing 5: Importing Nyquist class

2 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.

2.1 Defining sysem transfer function

An essential input to the `PoleZeroMap` class is the system transfer function. The system transfer function is divided into two seperate inputs: `num` and `den`. These represent the numerator and denominator coefficients of the system's transfer function: $H(s) = \frac{\texttt{num}(s)}{\texttt{den}(s)}$ for continuous-time systems (Laplace domain) and $H(z) = \frac{\texttt{num}(z)}{\texttt{den}(z)}$ for discrete-time systems (Z-domain). Both should be passed as lists or arrays of polynomial coefficients in descending powers of s or z. Before defining `num` and `den` one needs to define s and z as a symbolic expression.

Take the following examples:

```python
# Define system transfer function for continuous-time systems
s = sp.symbols('s') #defines 's' as symbolic expression
num = s+2
den = (s-1)*(s+6)
```

Listing 6: Example for defining continuous-time transfer function

```python
# Define system transfer function for discrete-time systems
z = sp.symbols('z') #defines 'z' as symbolic expression
num = z+2
den = z**2+0.25 # note: syntax for x^n is x**n in python
```

Listing 7: Example for defining discrete-time transfer function

2.2 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:

```python
# 'variablename' can take any valid variable name.
variablename = PoleZeroMap(num,den)

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

Listing 8: Example for defining discrete-time transfer function

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.

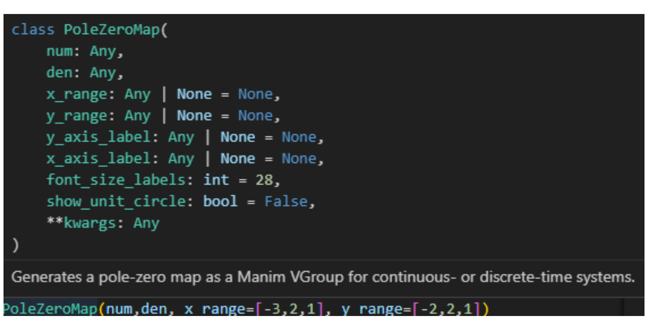

Figure 1: Input list

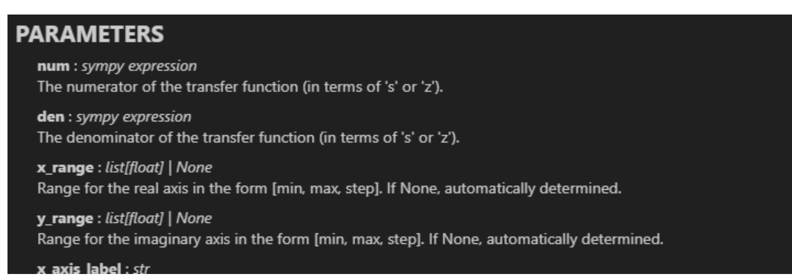

Figure 2: Parameters explanation

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(num,den)
pzmap.title("Pole-zero map")
```

Listing 9: 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(num,den)
pzmap.add_stability_regions()
```

Listing 10: 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.

2.3 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(num,den)
pzmap.add_stability_regions() #optional
self.add(pzmap) #adds the pole-zero map to the scene
```

Listing 11: Adding the pole-zero map to the scene

2.4 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(num,den)
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 12: Animating pole-zero map example

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

2.5 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	Optional unit circle displayed 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.
surrbox	White rectangular border surrounding the entire plot area.
dashed_x_axis	Dashed white line representing the real axis (x-axis).
dashed_y_axis	Dashed 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.

2.6 Tranistioning between pole-zero maps

One can use the `Transform()` 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:

```
s = sp.symbols('s')
        num1 = s+1
        den1 = s**2+0.2*s+5

        num2 = s-1
```

```
        den2 = (s+3)*(s-2)
        pzmap1 = PoleZeroMap(num1,den1, x_range=[-4,3,1], y_range=[-3,3,1])
        pzmap2 = PoleZeroMap(num2,den2, 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 13: Animating pole-zero map example

3 ControlSystem class

4 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.

4.1 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)"`)

```python
# 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 14: 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.

4.2 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

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

Listing 15: 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.

```python
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 16: 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 17: 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 18: Adding asymptotes

4.3 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 19: 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 20: Animating components

4.4 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)

4.5 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
```

```python
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 21: 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.

5 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.

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' 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 22: Creating Nyquist plot with different system specifications

5.2 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 23: 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 24: 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 25: Showing stability margins

5.3 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 26: 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 27: Animating components

5.4 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)
neg_nyquist_plot	Nyquist curve (negative 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
dashed_x_axis	Dashed Real axis
dashed_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

5.5 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 28: Transitioning example