tanuna Documentation

Release 0.1

Adrian Schlatter

Apr 14, 2023

CONTENTS

1 tanuna **1**
1.1 Diving In 1

2 Examples **5**
2.1 Mode-Locked Laser 5

3 API Documentation **9**
3.1 Package Root 9
3.2 Continuous-Time LTI Library 12

Bibliography **13**

Python Module Index **15**

Index **17**

CHAPTER

ONE

TANUNA

tanuna provides tools to work with dynamic systems. This currently includes

- continuous- but not discrete-time systems
- linear systems
- time-independent systems
- Single-Input Single-Output (SISO) and Multiple-Input Multiple-Output (MISO) systems

In the following, we will explain how to:

- create systems
- analyze systems
- solve systems
- combine systems

1.1 Diving In

Let's start with some examples based on a continuous-time, second-order LTI SISO system:

```
import tanuna as dyn
import numpy as np
import matplotlib.pyplot as pl

w0 = 2 * np.pi * 10
zeta = 0.5
k = 1.

A = np.matrix([[0, w0], [-w0, -2*zeta*w0]])
B = np.matrix([0, k*w0]).T
C = np.matrix([k, 0.])
D = np.matrix([0.])

G = dyn.CT_LTI_System(A, B, C, D)
```

This creates the system G from state-space matrices A, B, C, D. The system provides some interesting information:

```
>>> G.stable
True
```

(continues on next page)

(continued from previous page)

```
>>> G.poles
array([-31.41592654+54.41398093j, -31.41592654-54.41398093j])
>>> G.reachable
True
>>> # Reachability matrix:
... G.Wr
matrix([[    0.        ,  3947.84176044],
        [   62.83185307, -3947.84176044]])
>>> G.observable
True
>>> # Observability matrix:
... G.Wo
matrix([[  1.        ,   0.        ],
        [  0.        ,  62.83185307]])
```

Furthermore, it calculates step- and impulse-responses, Bode- and Nyquist-plots:

```
# -*- coding: utf-8 -*-

import tanuna as dyn
import numpy as np
import matplotlib.pyplot as pl

w0 = 2 * np.pi * 10
zeta = 0.5
k = 1.

A = np.matrix([[0, w0], [-w0, -2*zeta*w0]])
B = np.matrix([0, k*w0]).T
C = np.matrix([k, 0.])
D = np.matrix([0.])

G = dyn.CT_LTI_System(A, B, C, D)


pl.figure(figsize=(6, 12))

# STEP RESPONSE
pl.subplot(4, 1, 1)
pl.title('Step-Response')
t, sr = G.stepResponse()
pl.plot(t, sr[:, 0, 0])
pl.xlabel('Time After Step (s)')
pl.ylabel('y')

# IMPULSE RESPONSE
pl.subplot(4, 1, 2)
pl.title('Impulse-Response')
t, ir = G.impulseResponse()
pl.plot(t, ir[:, 0, 0])
pl.xlabel('Time After Impulse (s)')
pl.ylabel('y')
```

(continues on next page)

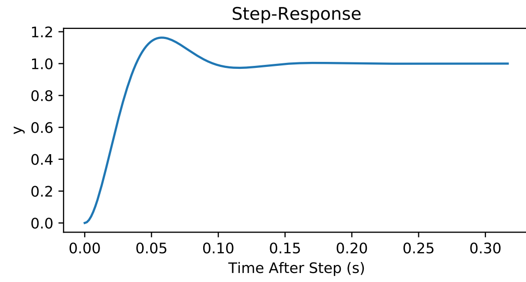
(continued from previous page)

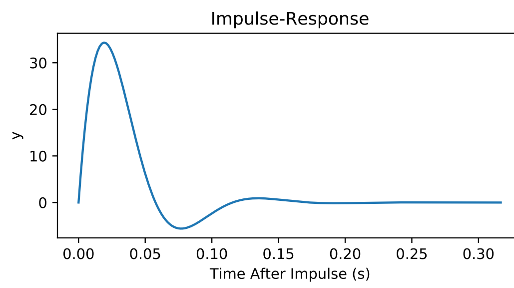
```
# BODE PLOT
ax1 = pl.subplot(4, 1, 3)
ax1.set_title('Bode Plot')
f, Chi = G.freqResponse()
ax1.semilogx(f, 20 * np.log10(np.abs(Chi[:, 0, 0])), r'b-')
ax1.set_xlabel('Frequency (Hz)')
ax1.set_ylabel('Magnitude (dB)')
ax2 = ax1.twinx()
ax2.semilogx(f, np.angle(Chi[:, 0, 0]) / np.pi, r'r-')
ax2.set_ylabel('Phase ($\pi$)', va='bottom', rotation=270)

# NYQUIST PLOT
ax = pl.subplot(4, 1, 4)
pl.title('Nyquist Plot')
pl.plot(np.real(Chi[:, 0, 0]), np.imag(Chi[:, 0, 0]))
pl.plot([-1], [0], r'ro')
pl.xlim([-3., 3])
pl.ylim([-1.5, 0.5])
ax.set_aspect('equal')
pl.axhline(y=0, color='k')
pl.axvline(x=0, color='k')
pl.xlabel('Real Part')
pl.ylabel('Imaginary Part')

pl.subplots_adjust(top=0.96, bottom=0.06, right=0.87, hspace=0.5)
```

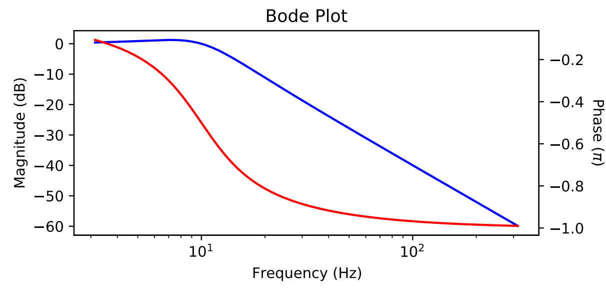
The duration of the trace and the density of samples is automatically determined for you based on the Eigenvalues of the system (but you can provide your own if you prefer).

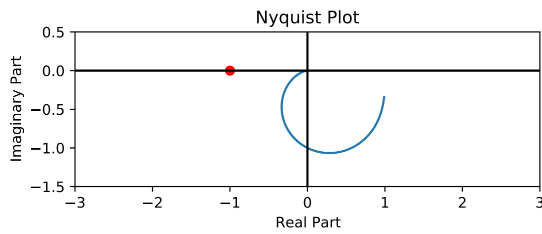
System-algebra is supported: You can connect systems in series, in parallel (creating a MIMO system from 2 SISO systems for example), and in feedback configuration:

```
>>> # Connect G in series with G:
... H = G * G
>>> # Connect G in parallel with G:
... J = G + G
>>> # This is the same as 2 * G:
... G + G == 2 * G
True
>>> # Check number of inputs and outputs:
... (2 * G).shape
(1, 1)
>>> G.shape
(1, 1)
>>> H.shape
(1, 1)
```


Step-Response
y
Time After Step (s)


Impulse-Response
y
Time After Impulse (s)


Bode Plot
Magnitude (dB)
Phase (π)
Frequency (Hz)


Nyquist Plot
Imaginary Part
Real Part

CHAPTER

TWO

EXAMPLES

2.1 Mode-Locked Laser

A mode-locked laser is a type of laser that operates in pulsed mode. This is achieved, e.g., by placing a saturable absorber inside the laser cavity. The saturable absorber has high losses for low light intensity but low losses for high intensity. This forces the laser to concentrate its light in short (and therefore intense) pulses.

However, the saturable absorber also leads to stability issues: When the pulse energy increases from its steady-state value, the saturable losses decrease - and vice versa. This tends to amplify deviations from the steady state and leads to so called Q-switched mode locking if not properly controlled. In Q-switched mode locking the laser emits bunches of pulses instead of a continuous stream of equally strong pulses.

Typically, Q-switched mode locking is avoided by proper design of the laser. Here, we don't do that. Instead, we design a state-feedback controller that stabilizes the laser by acting on its pump power.

The mode-locked laser is governed by the following differential equations:

$$\dot{P} = \frac{g - l - q_P(E_P)}{T_R} \cdot P$$
$$\dot{g} = \frac{\eta_P P_P}{E_{sat,L}} - \frac{g}{\tau_L} - \frac{P \cdot g}{E_{sat,L}}$$
$$q_P(S = E_P/E_{sat,A}) = \frac{\Delta R}{S} \cdot \left(1 - e^{-S}\right)$$

where P the power inside the laser cavity, g the gain provided by the gain medium, l and $q_P(E_P)$ the linear and non-linear losses, respectively, $E_P = P \cdot T_R$ the pulse energy, and T_R the time it takes the pulse to travel around the cavity once. $E_{sat,L}$ and $E_{sat,A}$ are the saturation energies of the gain medium and the saturable absorber, respectively, and τ_L the relaxation time of the gain.

The examples package includes the module 'laser' that provides a class to simulate such a laser. The class also includes a method 'approximateLTI' that returns the linear approximation around the steady state, i.e., a CT_LTI_System.

2.1.1 Q-Switching Instability

First, let's have a look at this Q-switching instability. We instantiate the NdYVO4 laser class defined in the examples package and choose a low pump power to assure that it Q-switches (0.1 Watts is appropriate). Then, we solve the differential equations to obtain $P(t)$ and $g(t)$:

```
from tanuna.examples.laser import NdYVO4Laser
from tanuna.sources import SourceConstant
from tanuna import connect
import numpy as np
```

(continues on next page)

(continued from previous page)

```
Ppump = 0.1
NdYVO4 = NdYVO4Laser(Ppump)
pump = SourceConstant(y=np.matrix(Ppump))
pumped_NdYVO4 = connect(NdYVO4, pump)

# ODE solving
# ============================================================================

Psteady, gsteady = NdYVO4.steadystate(Ppump)
t = np.arange(35000) * NdYVO4.TR

Pout, state = pumped_NdYVO4(t, return_state=True, method='DOP853')
P, g = state
```

The streamplot below shows that the laser's state spirals away from the (unstable) steady state towards a limit cycle. The energy of the mode-locked pulses (and therefore P_{out}) is pulsating. This is what we will elliminate in the next section.

2.1.2 Control

By linear approximation around the (unstable) steady-state, we create a second order LTI. This system is also modified so that it not only provides the laser power as output but the internal state as well:

$$\dot{\vec{x}} = A\vec{x} + Bu$$
$$\vec{y} = \begin{bmatrix} 0 & T_{oc} \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \vec{x}$$

where $\vec{x} = \left[\delta \dot{P}/\omega_0, \delta P\right]^T$ is the (transformed!) state and $u = \delta P_P$ the deviation from the (design-) pump power.

```
from tanuna.examples.laser import NdYVO4Laser
from tanuna.sources import SourceConstant
import numpy as np
import tanuna as dyn

# Setup laser
# ============================================================================

Ppump = 0.1
NdYVO4 = NdYVO4Laser(Ppump=0.)


# Linearized
# ============================================================================

M, system_lin = NdYVO4.approximateLTI(Ppump)
# Add state outputs:
A, B, C, D = system_lin.ABCD
Toc = NdYVO4.Toc
```

(continues on next page)

(continued from previous page)

```
C = np.matrix([[0, Toc],
               [1, 0],
               [0, 1]])
D = np.matrix(np.zeros((3, 1)))
system_lin = dyn.CT_LTI_System(A, B, C, D)
```

Next, we add (state-) feedback to obtain the controlled system:

Fig. 1: Block diagram of laser with state feedback.

r is the control input, k_r a constant, and $K = [0, k_1, k_2]$ the feedback matrix.

We can now choose K in such a way that the stabilized systems has poles where we want them. It can be shown that to obtain poles at:

$$p_{1,2} = -\gamma\omega_0 \pm \sqrt{\nu}|\omega_0|$$

we have to choose

$$K = \left[0, 2(\gamma - \zeta)/\rho, (\gamma^2 - \nu - 1)/\rho\right]$$

where ζ is the damping ratio of the free-running system. Further, we choose $k_r = \gamma^2 - \nu$ to obtain the same DC-gain as in the uncontrolled system. If the system is known perfectly and if the feedback is implemented exactly as calculated, the dynamics of the controlled system will be exactly as intended. In reality, neither is true. Therefore, we assume errors in the knowledge of P_P (factor 1.5) and the calibration of the feedback, i.e., in k_r, k_1, and k_2 (factor of 0.8, each):

```
# Where we want the poles to be:
gamma = 0.05
nu = -1.
# => poles will be at -gamma * w0 +/- sqrt(nu) * w0 = -0.05 * w0 +/- i * w0


# We assume that the controller is not calibrated perfectly.
#   a) The assumed pump power is factor rPp from real pump power
#   b) The implemented feedback is factors rkr, rk1, rk2 from calculated values
# Note how large we choose the errors!
rPp = 1.5
rkr = 0.8
rk1 = 0.8
rk2 = 0.8

# Calculate and apply feedback:
Ppump_assumed = rPp * Ppump
kr = rkr * (gamma**2 - nu)
k1 = rk1 * 2 * (gamma - NdYVO4.zeta(Ppump_assumed)) / NdYVO4.rho(Ppump_assumed)
k2 = rk2 * (gamma**2 - nu - 1.) / NdYVO4.rho(Ppump_assumed)

stateoutput = np.matrix([[1, 0, 0]])
K = np.matrix([[0, k1, k2]])
L = np.vstack([stateoutput, K])
summing = np.matrix([kr, -1])
stabilized_lin = L * system_lin * summing
stabilized_lin = dyn.feedback(stabilized_lin, Gout=(1,), Gin=(1,))
```

Now, let's compare to the free-running system. The figure below shows the step-response and the poles of the free-running and the controlled system. As expected, the relaxation oscillation are damped, resulting in a stable system. The gain is not exactly what we aimed for (green curve is not converging towards the target gain (dashed line). Given the large errors we have assumed this is - however - still a quite acceptable result.

Let's see whether this works as nicely when applied to the original, non-linear system. Particularly, after turning on the laser it is far away from the steady-state we linearized it around and may behave quite differently than shown in the linear simulation shown above. In contrast to the simulation of the linearized system above, we will not start from the steady-state and feed a step. Instead, we will start with the laser turned off (:math: *P_{pump} = 0*), the set the pump power to a level that usually leads to Q-switching:

```
class StateOutputLaser(NdYVO4Laser):
    """Same as NdYVO4Laser but outputs (Pout, g, P) instead of only Pout."""

    def __init__(self, Ppump=0.):
        super().__init__(Ppump=Ppump)
        self.shape = (3, 1)

    def g(self, t, s, u):
        """
        This is the output function of the CT_System and returns the
        output power of the laser. Despite its name, is *not* related
        to the laser's gain!
        """
        P, g = s
        return np.matrix([self.Toc * P, P, g])


so_NdYVO4 = StateOutputLaser(Ppump=0.0)
so_NdYVO4.s = np.matrix([[0.1, 0]]).T  # "noise photons"

P0, g0 = so_NdYVO4.steadystate((Ppump))

y0 = np.matrix([[0], [-P0], [-g0]])
stabilized = so_NdYVO4.offset_outputs(y0)
stabilized = stabilized.offset_inputs(Ppump)
Maugmented = np.eye(3)
Maugmented[1:, 1:] = M
stabilized = L * Maugmented * stabilized * summing
stabilized = dyn.feedback(stabilized, Gout=(1,), Gin=(1,))
stabilized = dyn.connect(stabilized, SourceConstant(y=np.matrix(0.)))
system = dyn.connect(so_NdYVO4, SourceConstant(y=np.matrix(Ppump)))
```

The laser still goes through a few Q-switching cycles but the control manages to damp them. Note, however, that some modulation remains. Apparently, the system still has a limit cycle, albeit a much smaller one than the unstabilized laser.

CHAPTER

THREE

API DOCUMENTATION

3.1 Package Root

Root module of tanuna package.

@author: Adrian Schlatter

exception tanuna.root.**ApproximationError**

class tanuna.root.**CT_LTI_System**(*A*, *B*, *C*, *D*, *x0=None*)

Continuous-time, Linear, time-invariant system

property Wo

Observability matrix

property Wr

Reachability matrix

freqResponse(*f=None*)

Returns (f, r), where

f : Array of frequencies r : (Complex) frequency response

f is either provided as an argument to thin function or determined automatically.

impulseResponse(*t=None*)

Returns (t, yimpulse), where

yimpulse : Impulse response (*without* direct term D) t : Corresponding array of times

t is either provided as an argument to this function or determined automatically.

property observable

Returns True if the system is observable.

property order

The order of the system

property poles

Eigenvalues of the state matrix

property reachable

Returns True if the system is reachable.

property shape

Number of outputs and inputs

stepResponse(*t=None*)

Returns (t, ystep), where

ystep : Step response t : Corresponding array of times

t is either provided as an argument to this function or determined automatically.

property tf

Transfer-function representation [b, a] of the system. Returns numerator (b) and denominator (a) coefficients.

$$G(s) = \frac{b[0] * s^0 + ... + b[m] * s^m}{a[0] * s^0 + ... + a[n] * s^n}$$

property zpk

Gain, Pole, Zero representation of the system. Returns a tuple (z, p, k) with z the zeros, p the poles, and k the gain of the system. p is an array. The format of z and k depends on the number of inputs and outputs of the system:

For a SISO system z is an array and k is float. For a system with more inputs or outputs, z and k are lists of 'shape' (nout, nin) containing arrays and floats, respectively.

class tanuna.root.**CT_System**(*f*, *g*, *s0*)

Describes a continuous-time system with dynamics described by ordinary differential equations.

s: Internal state (vector) of the system s0: Initial state of the system u: External input (vector)

f(t, s, u): Dynamics of the system (ds/dt = f(t, s, u)) g(t, s, u): Function that maps state s to output y = g(t, s, u)

It is solved by simply calling it with an argument t. t is either a float or array-like. In the latter case, the system is solved for all the times t in the array.

observable(*t*)

Returns whether the system is observable at time t (i.e. its internal state is determinable from inputs u and outputs y).

reachable(*t*)

Returns whether the system is reachable at time t (i.e. all states are reachable by providing an appropriate input u(t)).

steadyStates(*u0*, *t*)

Returns a list of tuples (s_i, stability_i) with:

- s_i: A steady-state at time t, i.e. f(t, s_i, u0) = 0
- stability_i: True if this steady-state is stable, false otherwise

tangentLTI(*s0*, *u0*, *t*)

Approximates the CT_System at time t near state s0 and input u0 by an LTISystem (linear, time-invariant system). Raises ApproximationError if the system can not be linearized.

exception tanuna.root.**ConnectionError**

class tanuna.root.**DT_LTI_System**(*A*, *B*, *C*, *D*, *x0=matrix([[0.], [0.]])*)

Implements the discrete-time linear, time-invariant system with input vector u[t], internal state vector x[t], and output vector y[t]:

x[t+1] = A * x[t] + B * u[t] y[t] = C * x[t] + D * u[t]

where
A: state matrix B: input matrix C: output matrix D: feedthrough matrix

The system is initialized with state vector x[0] = x0.

`classmethod fromTransferFunction(phi)`
Initialize DiscreteLTI instance from transfer-function coefficients 'Theta' and 'phi'.

`observable()`
Returns true if the system is observable

`proper()`
Returns true if the system's transfer function is strictly proper, i.e. the degree of the numerator is less than the degree of the denominator.

`reachable()`
Returns True if the system is observable

`stable()`
Returns True if the system is strictly stable

`tf()`
Returns the transfer function (b, a) where 'b' are the coefficients of the nominator polynomial and 'a' are the coefficients of the denominator polynomial.

`class tanuna.root.DT_LTV_System(At, Bt, Ct, Dt, X0)`
Implements the discrete linear, time-variant system with input vector u[t], internal state vector x[t], and output vector y[t]:

x[t+1] = A[t]*x[t] + B[t]*u[t] y[t] = C*x[t] + D*u[t]

where
A[t]: state matrices B[t]: input matrices C[t]: output matrices D[t]: feedthrough matrices

The system is initialized with state vector x[0] = X0.

`exception tanuna.root.MatrixError`

`tanuna.root.Thetaphi(b, a)`
Translate filter-coefficient arrays b and a to Theta, phi representation:

phi(B)*y_t = Theta(B)*x_t

Theta, phi = Thetaphi(b, a) are the coefficient of the back-shift-operator polynomials (index i belongs to B^i)

`tanuna.root.ba(Theta, phi)`
Translate backshift-operator polynomials Theta and phi to filter coefficient array b, a.

a[0]*y[t] = a[1]*y[t-1] + … + a[n]*y[t-n] + b[0]*x[t] + … + b[m]*x[t-m]

`tanuna.root.cofactorMat(A)`
Cofactor matrix of matrix A. Can handle matrices of poly1d.

`tanuna.root.connect(H, G, Gout=None, Hin=None)`
Connect outputs Gout of G to inputs Hin of H. The outputs and inputs of the connected system are arranged as follows:

- remaining outputs of G get lower, the outputs of H the higher indices
- inputs of G get the lower, remaining inputs of H the higher indices

connect(H, G) is equivalent to H * G.

`tanuna.root.determinant(A)`

Determinant of square matrix A. Can handle matrices of poly1d.

`tanuna.root.differenceEquation(b, a)`

Takes filter coefficient arrays b and a and returns string with difference equation using powers of B, where B the backshift operator.

`tanuna.root.feedback(G, Gout, Gin)`

Create feedback connection from outputs Gout to inputs Gin

`tanuna.root.minor(A, i, j)`

Returns matrix obtained by deleting row i and column j from matrix A.

`tanuna.root.polyDiag(polyList)`

Construct diagonal matrix from list of poly1d

3.2 Continuous-Time LTI Library

Library of ready-to-use continuous-time LTI systems.

@author: Adrian Schlatter

`class tanuna.CT_LTI.HighPass(fC, k=1.0)`

High-Pass Filter with 3-dB frequency fC and pass-band gain k

`class tanuna.CT_LTI.LowPass(fC, k=1.0)`

Low-Pass Filter with 3-dB frequency fC and DC-gain k

`class tanuna.CT_LTI.Order2(w0, zeta, k)`

A second-order system with

- w0: Natural frequency
- zeta: Damping ratio (0: undamped, 1: critically damped)
- k: Gain

BIBLIOGRAPHY

[feedback_systems] Karl Johan Åström and Richard M. Murray, "Feedback Systems", Princeton University Press, 2012

PYTHON MODULE INDEX

t

tanuna.CT_LTI, 12
tanuna.root, 9

INDEX

A

ApproximationError, 9

B

ba() (*in module tanuna.root*), 11

C

cofactorMat() (*in module tanuna.root*), 11
connect() (*in module tanuna.root*), 11
ConnectionError, 10
CT_LTI_System (*class in tanuna.root*), 9
CT_System (*class in tanuna.root*), 10

D

determinant() (*in module tanuna.root*), 12
differenceEquation() (*in module tanuna.root*), 12
DT_LTI_System (*class in tanuna.root*), 10
DT_LTV_System (*class in tanuna.root*), 11

F

feedback() (*in module tanuna.root*), 12
freqResponse() (*tanuna.root.CT_LTI_System method*), 9
fromTransferFunction() (*tanuna.root.DT_LTI_System class method*), 11

H

HighPass (*class in tanuna.CT_LTI*), 12

I

impulseResponse() (*tanuna.root.CT_LTI_System method*), 9

L

LowPass (*class in tanuna.CT_LTI*), 12

M

MatrixError, 11
minor() (*in module tanuna.root*), 12
module
    tanuna.CT_LTI, 12
    tanuna.root, 9

O

observable (*tanuna.root.CT_LTI_System property*), 9
observable() (*tanuna.root.CT_System method*), 10
observable() (*tanuna.root.DT_LTI_System method*), 11
order (*tanuna.root.CT_LTI_System property*), 9
Order2 (*class in tanuna.CT_LTI*), 12

P

poles (*tanuna.root.CT_LTI_System property*), 9
polyDiag() (*in module tanuna.root*), 12
proper() (*tanuna.root.DT_LTI_System method*), 11

R

reachable (*tanuna.root.CT_LTI_System property*), 9
reachable() (*tanuna.root.CT_System method*), 10
reachable() (*tanuna.root.DT_LTI_System method*), 11

S

shape (*tanuna.root.CT_LTI_System property*), 9
stable() (*tanuna.root.DT_LTI_System method*), 11
steadyStates() (*tanuna.root.CT_System method*), 10
stepResponse() (*tanuna.root.CT_LTI_System method*), 9

T

tangentLTI() (*tanuna.root.CT_System method*), 10
tanuna.CT_LTI
    module, 12
tanuna.root
    module, 9
tf (*tanuna.root.CT_LTI_System property*), 10
tf() (*tanuna.root.DT_LTI_System method*), 11
Thetaphi() (*in module tanuna.root*), 11

W

Wo (*tanuna.root.CT_LTI_System property*), 9
Wr (*tanuna.root.CT_LTI_System property*), 9

Z

zpk (*tanuna.root.CT_LTI_System property*), 10