Automatic Control Laboratory, ETH Zürich
Prof. John Lygeros

Revised by Niklas Schmid
Revision from: September 10, 2022

# MATLAB Introduction

## Day 2

---

Today we will look at functionalities MATLAB offers to define, analyse and simulate dynamical systems. This can be used to analyse a system's behaviour based on a model, verify models against data or to design and analyse controllers.

# Contents


**1 Theory** **3**

1.1 Packages . . . 3

1.2 Linear system theory . . . 3

1.3 Linear systems: Definition . . . 4

1.3.1 Definition using standard MATLAB . . . 4

1.3.2 Definition using the Control Systems Toolbox . . . 6

1.4 Linear systems: Analysis . . . 6

1.4.1 Stability analysis . . . 7

1.4.2 Frequency response analysis . . . 7

1.4.3 Time response analysis . . . 8

1.5 Simulating nonlinear systems . . . 9

**2 Practice** **11**

# Chapter 1

# Theory

## 1.1 Packages

Last time we talked about classes like strings and matrices. However, you may want to perform fancier computations using methods like neural networks without having to program everything from scratch. Therefore, MATLAB offers a lot of additional packages which introduce new types of classes and functions. One of these packages that we will look at today is the *Control Systems Toolbox*, which allows you to define state space systems and transfer functions, as well as to solve differential equations and run simulations.

First let's have a look at linear systems. The following section serves as a brief summary. If you are not familiar with linear systems, just take the following facts for granted. They will be introduced in more detail in the ETH control systems and linear system theory lectures. There is also freely available literature which you may find useful.[1]

## 1.2 Linear system theory

A linear dynamical system can be represented using differential equations in the form of

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{1.1}$$

with $A, B, C, D$ being matrices, $u$ the input, $x$ the state, and $y$ the output.

An alternative representation is the Laplace-transform, where a rational transfer function

$$G(s) = \frac{Y(s)}{U(s)} = \frac{b_0 s^n + b_1 s^{n-1} + \cdots + b_{n-1} s + b_n}{s^n + a_1 s^{n-1} + \cdots + a_{n-1} s + a_n} \tag{1.2}$$

is used.

One can obtain the transfer function of system 1.1 as

$$G(s) = C(sI - A)^{-1} B + D \tag{1.3}$$

A transfer function of form 1.2 can be transferred into state space form, e.g. using the controller

[1] K. J. Astrom and R. Murray. Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press, 2010. (freely available online)

canonical form

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \dots & 1 \\ -a_n & -a_{n-1} & -a_{n-2} & \dots & -a_1 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} u \tag{1.4}$$

$$y = \begin{bmatrix} (b_n - a_n b_0) & (b_{n-1} - a_{n-1} b_0) & \dots & (b_2 - a_2 b_0) & (b_1 - a_1 b_0) \end{bmatrix} x + b_0 u. \tag{1.5}$$

When analysing a dynamical system we often obtain differential equations leading to the state space model. The system is stable if the eigenvalues of the matrix $A$ have a negative real part.

On the other hand, the transfer function is useful, since we can easily compute its poles and zeros and thus determine the frequency behaviour of the system. The system is stable if all poles have a negative real part. Furthermore, for second order systems we can easily obtain the damping coefficient and resonance frequency of the system.

## 1.3 Linear systems: Definition

So how can we define a linear system in MATLAB? We will have a look at how to do it using standard MATLAB code. We will then discuss a much more convenient way using the Control Systems Toolbox.

### 1.3.1 Definition using standard MATLAB

#### State space models

We can define a linear system by its state space matrices $A, B, C, D$. E.g., for the system,

$$\dot{x} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1 \end{pmatrix} u \tag{1.6}$$

$$y = \begin{pmatrix} 1 & 0 \end{pmatrix} x$$

we would define

```
A = [0, 1; 0, 0];
B = [0; 1];
C = [1, 0];
D = 0;
```

Alternatively, one could define the polynomials that define the transfer function. Let's have a closer look at how to store and work with polynomials in MATLAB.

#### Transfer functions

MATLAB offers a lot of functionality to work with polynomials with natural positive powers. Therefore, the coefficients of the polynomials are stored in a vector.

$$a_n s^n + a_{n-1} s^{n-1} + a_0$$
$$\updownarrow$$
$$[a_n, a_{n-1}, \dots, a_0]$$

Thus, for a transfer function

$$G(s) = \frac{b_0 s^n + b_1 s^{n-1} + \dots + b_{n-1} s + b_n}{s^n + a_1 s^{n-1} + \dots + a_{n-1} s + a_n} \tag{1.7}$$

we would define two polynomials $den = [b_0, \ldots, b_n], num = [1, a_1, \ldots, a_n]$ for the denominator and numerator, respectively.

You can evaluate a polynomial using the function *polyval(p,x)*, where the polynomial $p$ will be evaluated at every point in x:

```
polyval([1 0 0],[1 -2 3])
ans =
      1   4   9
```

You can add and subtract two polynomials by adding or subtracting their coefficient vectors. Note that both vectors have to be of the same size for this operation. If one polynomial is of lower order, you may add some zeros from the left to make them equally sized. Let's compute

$$(s^2 + s + 1) - (s + 2) = s^2 - 1. \tag{1.8}$$

```
[1 1 1] - [0 1 2]
ans =
      1   0  -1
```

The multiplication of two polynomials equals the convolution of their coefficients:

$$(s + 2)(s + 3) = s^2 + 5s + 6 \tag{1.9}$$

```
conv([1 2], [1 3])
ans =
      1   5   6
```

Equivalently, the division of two polynomials equals their deconvolution

$$\frac{s^2 + 5s + 7}{s + 2} = s + 3 + \frac{1}{s + 2} \tag{1.10}$$

```
[q,r] = deconv([1 5 7],[1 2])
q =
      1   3
r =
      0   0   1
```

The function *roots(p)* returns the roots of the polynomial $p$. On the contrary, given a vector of roots $N$, the function *poly(N)* returns the coefficients of a respective polynom. Thus the following command

```
roots(poly(1:n))
```

should give us the roots of a polynomial with roots at $1, 2, 3, \ldots, n$, which means it should return a vector $[1, 2, 3, \ldots, n]$. Try this command with $n = 30$. What do you observe? Can you explain this behaviour?

### Conversions between state space representation and transfer function

Assume you are given the state space representation of a system via matrices $A, B, C, D$. One can obtain the coefficients of the numerator and denominator of the corresponding transfer function using *[num,den] = ss2tf(A,B,C,D)*. Vice versa, the function *[A,B,C,D] = tf2ss(num,den)* can be used to obtain the state space representation of a given transfer function.

Let's look at an example. An integrator system is given by

$$y = \int u dt. \tag{1.11}$$

We can write these dynamics as state space

$$\dot{x} = Ax + Bu = u$$
$$y = Cx + Du = x$$

by choosing $A = 0, B = 1, C = 1, D = 0$, or as transfer function using

$$\frac{Y(s)}{U(s)} = \frac{b_0 s + b_1}{a_0 s + a_1} = \frac{1}{s} \tag{1.12}$$

by choosing $den = [b_0, b_1] = [0, 1], num = [a_0, a_1] = [1, 0]$. Verify that you obtain the corresponding conversion when using the commands *ss2tf()* and *tf2ss()*.

For multidimensional systems you must specify the desired output via an additional argument $n$ in *[Z,N] = ss2tf(A,B,C,D,n)*. Otherwise, the command will return a matrix with all transfer functions from every input to every output.

### 1.3.2 Definition using the Control Systems Toolbox

A much more convenient way of dealing with dynamical systems in MATLAB is provided by the Control Systems Toolbox. This toolbox provides dedicated classes for LTI systems (**L**inear **T**ime **I**nvariant systems). The definition of a system works in a similar way, but the resulting LTI-system will be stored in one single object.

```
sys1 = tf(num,den) % transfer function
sys2 = ss(a,b,c,d) % state space
```

MATLAB will create a tf or ss object, respectively. You can change their preferred representation using the commands *ss(sys1)* or *tf(sys2)*.

These objects have several nice features. You can again extract the state space matrices and the coefficients using the commands

```
[num,den] = tfdata(sys1);
[a,b,c,d] = ssdata(sys2);
```

You can can add, subtract, multiply and divide two transfer functions using the conventional operators

```
sys1 + sys1
2 * sys1 * sys1
```

You can also connect LTI-systems in special configurations, which are explained on the respective help-pages. Some helpful commands are: *append(), parallel(), series(), feedback(), star(), connect()*.

We will next talk about the useful tools that the toolbox provides to analyse systems.

## 1.4 Linear systems: Analysis

Throughout this section, we will look at the following example

```
A = [-1, 1; 0, 0];
B = [0; 1];
C = [1, 0];
D = 0;
example_system=ss(A,B,C,D)
```

and analyse its behaviour.

### 1.4.1 Stability analysis

In order to analyse the stability of a system in state space form we will look at its eigenvalues.

```
eig(A)
```

You can also pass a ss- or tf-object to the function *eig()*.

Alternatively, we can look at the roots of the characteristic polynomial

```
cp = poly(A)
rts = roots(cp)
```

For our initial example

```
eig(example_system)
ans =
   -1
    0
```

Thus we have a stable pole and an integrator.

### 1.4.2 Frequency response analysis

#### Nyquist-Plot

The Nyquist plot shows the frequency behaviour of the system for positive frequencies in the complex plane. You can generate a Nyquist plot using the command

```
[re,im,wout]=nyquist(example_system)
```

If you do not store any values, MATLAB will automatically generate a Nyquist plot. The variables *re*, *im* and *wout* are vectors of the same length. The elements in *re* and *im* are the real and imaginary values obtained when evaluating the system at the frequencies given in *wout*.

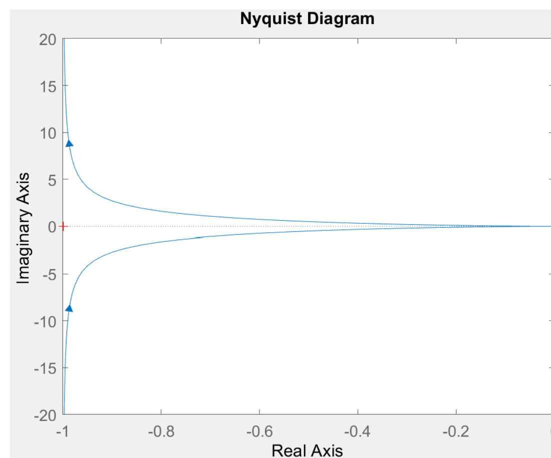


Figure 1.1: *nyquist(example_system)*. The arrows point towards increasing frequencies. The lower graph belongs to the positive frequencies. The upper graph belongs to the negative frequencies.

#### Bode-Plot

The Bode-Plot depicts the amplitude and phase of the system over the frequency range. The respective command is *[mag,phase,wout] = bode(sys)*. The variables *mag*, *phase* and *wout* are again vectors of the same length. The elements in *mag* and *phase* are the magnitude and phase values obtained when evaluating the system at the frequencies given in *wout*.

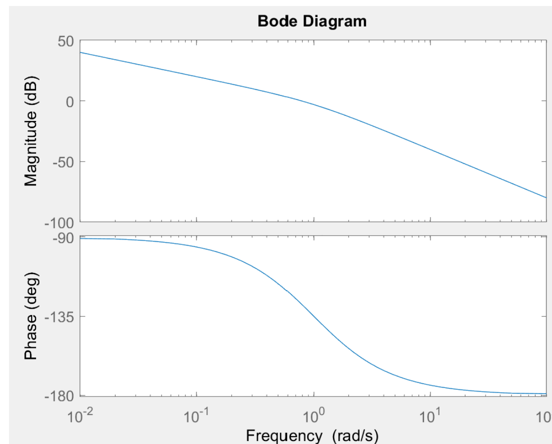


Figure 1.2: *bode(example_system)*. We can clearly see the effect of the integrator since we start at $-90°$ and the effect of the stable pole at 1 rad/s

MATLAB automatically picks the frequencies in *wout* at which the transfer function is evaluated. If you want to specify these frequencies, you can pass them as a vector *w* via *[re,im,wout]=nyquist(sys,w)*. Then, *wout=w*. Instead of passing a ss- or tf-object you can also pass the state space matrices or the polynomials of the transfer function directly (*[re,im,w]=nyquist(a,b,c,d,w)*, *[re,im,w]=nyquist(num,den,w)*). Everything mentioned also holds for the command *bode()*.

### 1.4.3 Time response analysis

We will now look at the time-response of a system to specific inputs, a.k.a. we will simulate the system.

#### Impulse and step response

The impulse and step response of an ss or tf object can easily be computed and plotted using the functions *impulse()* and *step()*. Similar as for the bode- and nyquist commands you can also pass the transfer function and state space matrices directly. You can define a vector $t$ specifying the points in time at which the response is evaluated (similar as you did with $w$ for the bode and nyquist commands. This is still optional).

```
figure()
subplot(2,1,1)
impulse(example_system)
subplot(2,1,2)
step(example_system)
```

#### Response to user-defined inputs

You can simulate the response of a system to arbitrary inputs by defining a vector of inputs $U$, which are applied at times $T$ and using the command

```
[Y,T] = lsim(example_system,U,T);
```

You can also track the evolution of the state using *[Y,T,X] = lsim(sys1,U,T)* and define a specific initial state *[Y,T,X] = lsim(example_system,U,T,X0)*. If no initial state is defined, MATLAB assumes it to be zero.

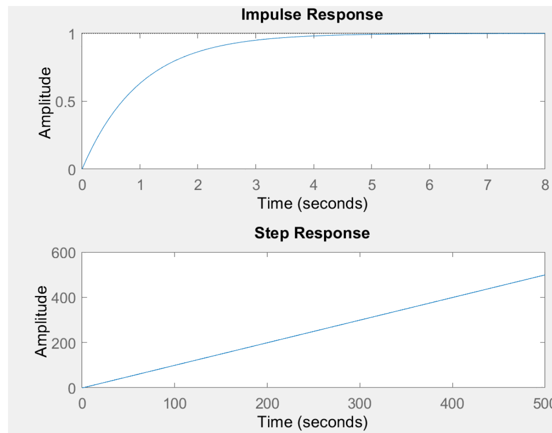


Figure 1.3: Impulse and step response of the system.

## 1.5 Simulating nonlinear systems

So far we only looked at linear, time invariant systems. The described methods cannot handle nonlinear or time-varying systems. For these cases MATLAB offers more dedicated tools.

Let's consider systems of the form

$$\begin{aligned}\dot{x} &= f(x,u,t)\\ y &= g(x,u,t)\\ x(0) &= x_0.\end{aligned}$$

We differentiate between the case $u = 0$ (called the initial value problem, since we only look at the system behaviour based on the initial condition $x_0$), and the case $u \neq 0$. The case $u \neq 0$ will not be discussed within this script. We refer the reader to the Simulink environment that is incorporated in MATLAB and offers vast functionality for arbitrary types of simulations. For now we will only look at the initial value problem.

Therefore, the system function $f(x,u,t)$ has to be put into a MATLAB function with input parameters $x$ and $t$ and output variable $dx$. The name of the function can be chosen arbitrarily and neither $x$ nor $t$ actually have to be used within the function.

The system can now be simulated using a specific solver that integrates the function $f(x,u,t)$ starting from $x_0$. Therefore, we use commands of the form

```
[T,X] = solver(functionName,tspan,x0)
```

where

- $T$ is a vector and $X$ a matrix, where $X(i,:)$ are the states at time $T(i)$
- *functionName* is the name of the function $f(x,u,t)$
- *tspan* is a vector of points $t$ in time at which the state $X(t)$ shall be stored in the vector $X$. If only the initial and terminal time are given, then MATLAB automatically picks these values in between. Otherwise, *tspan*$= T$.
- *solver* is the numerical integration method that MATLAB shall use. A choice that often works is ode45 or ode15. The benefit of ode45 is that it is very fast, but it does not work for every system. More on that later.

Let's look at an example. We want to simulate the system

$$\dot{x}(t) = -x(t)||x(t)||^3 \qquad x(0) = \begin{pmatrix}1\\3\end{pmatrix} \tag{1.13}$$

starting from time $t = 0$ until $t = 10$. Therefore, we define the MATLAB function

```
function dx = Tsystem1(t,x)
    dx = -x*norm(x)^3;
end
```

and call the solver

```
ode45('Tsystem1',[0 10],[1; 3]);
```

A plot of the results automatically appears because the output of the solver has not been assigned to any variables.

Great caution is needed when choosing an integration method. There exist big varieties of methods with individual benefits and weaknesses. A lack of knowledge or carelessness may lead to arbitrarily wrong simulation results. Even worse, since wrong simulation results are purely caused by numerical issues, no solver will print an error message in case the simulation result is off. A long list of solvers and examples explaining when to use which solver can be found on the respective MATLAB help page.

# Chapter 2

# Practice

Today's afternoon we will look at serial connections of the first order low pass filter

$$T\dot{y} + y = u \tag{2.1}$$

with $T = 5$. We want to put $N$ of these filters in series and analyse their behaviour.

1. Create a MATLAB script in which you generate a Nyquist- and Bode-Plots of the system's frequency behaviour for $\omega \in [0.01, 500]$. Also, plot the impulse- and step-response of the filter for $t \in [0, 50]$ with an adequate step-length.

2. Generate a vector that contains a sawtooth signal with an amplitude $A = 1$ and a period length $T_p = 12$ for the time interval $t \in [0, 60]$ and plot it. Hint: You need to generate two vectors: A vector $U$ that stores the signal values and a vector $T$ of same length with the corresponding time stamps of the values in $U$. Use the double-dot operator for the generation of a segment of the vector $U$ and concatenate the segments to obtain the desired sawtooth signal.

3. Simulate the filter response when using the sawtooth signal as an input. Plot the input and output signals of the five initial periods in one figure.

4. Write a MATLAB function *TFseries(num1, den1, num2, den2)* that takes the numerator and denominator of two transfer functions as inputs. The output shall be a transfer function that has the frequency response of the serial connection of the two input functions. Hint: The serial connection of two systems means that their transfer functions are multiplied. For now, only consider SISO systems.

5. Write another function *TFseriesN(num, den, N)* that outputs a transfer function, which equals $N$ serial connections of the inputted transfer function. Therefore, call and make use of your function *TFseries(num1, den1, num2, den2)* within *TFseriesN(num, den, N)*. Print an error message if the input $N$ is not meaningful.

6. We now want to compare the responses for different lengths of series of our low-pass filter. Compute the transfer functions for $N = 1, 2, 4, 8, 16$. Generate four figures. Inside these figures we will plot the Nyquist plots, Bode plots, the impulse responses and the step responses, respectively. In every figure, plot the respective responses for all $N$ in one single plot using the command *hold on*. Add a legend to the plots using the command *legend()*, so that you know which graph belongs to which $N$.