A comprehensive introduction
to the dynamics of

First Order Systems

for game devs
(and other curious people)

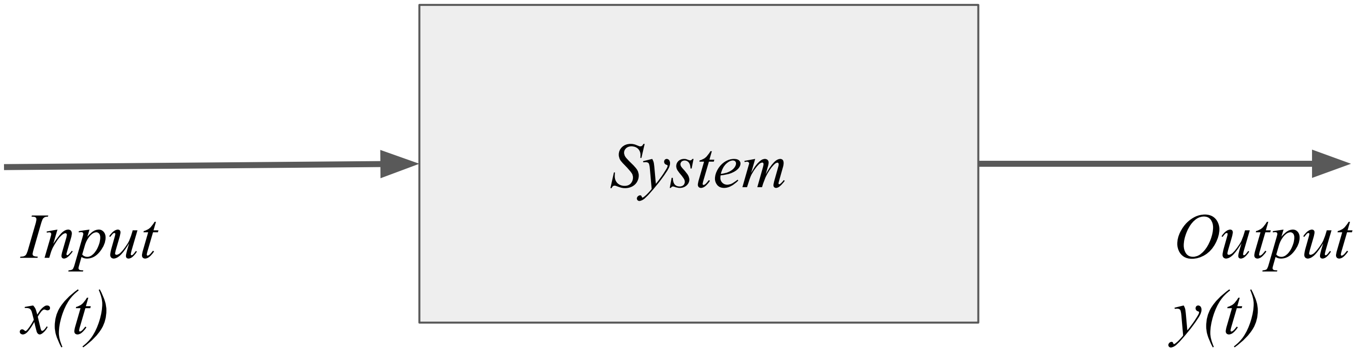
Valentin Sagrario
Jan 2020

@val_sagrario

☝️

First Order System

What's a First Order system?



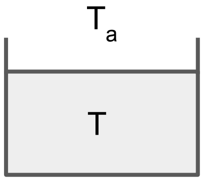
**Systems considered in this presentation are assumed linear and time invariant (LTI)*

Any physical system that can be modeled by a 1st order differential equation

$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

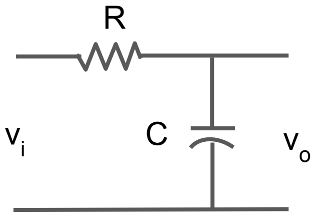
Many real system can be modeled and approximated by this equation.

Cooling cup of coffee



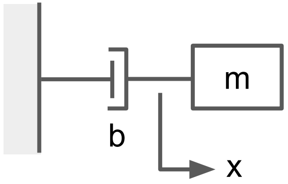
$$RC\frac{dT}{dt} + T = T_a$$

Camera flash discharge



$$RC\frac{dv_o}{dt} + v_o = v_i$$

Braking automobile



$$\frac{m}{b}\frac{dx_o}{dt} + x_o = x_i$$

No matter the nature of the system,
the equation has always the same structure

Term that depends on the input

$$\tau \frac{dy}{dt} + y_{(t)} = \boldsymbol{x_{(t)}}$$

Terms that depends on the output

$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

Higher derivative order is 1

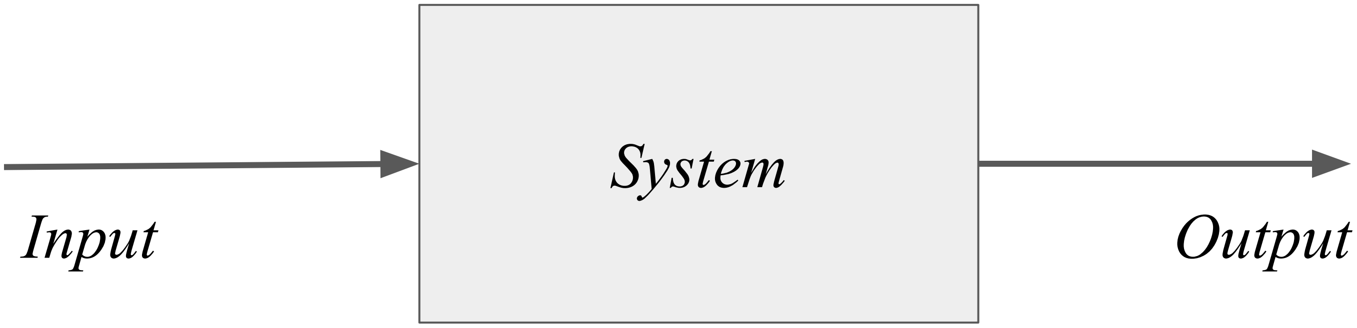
$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

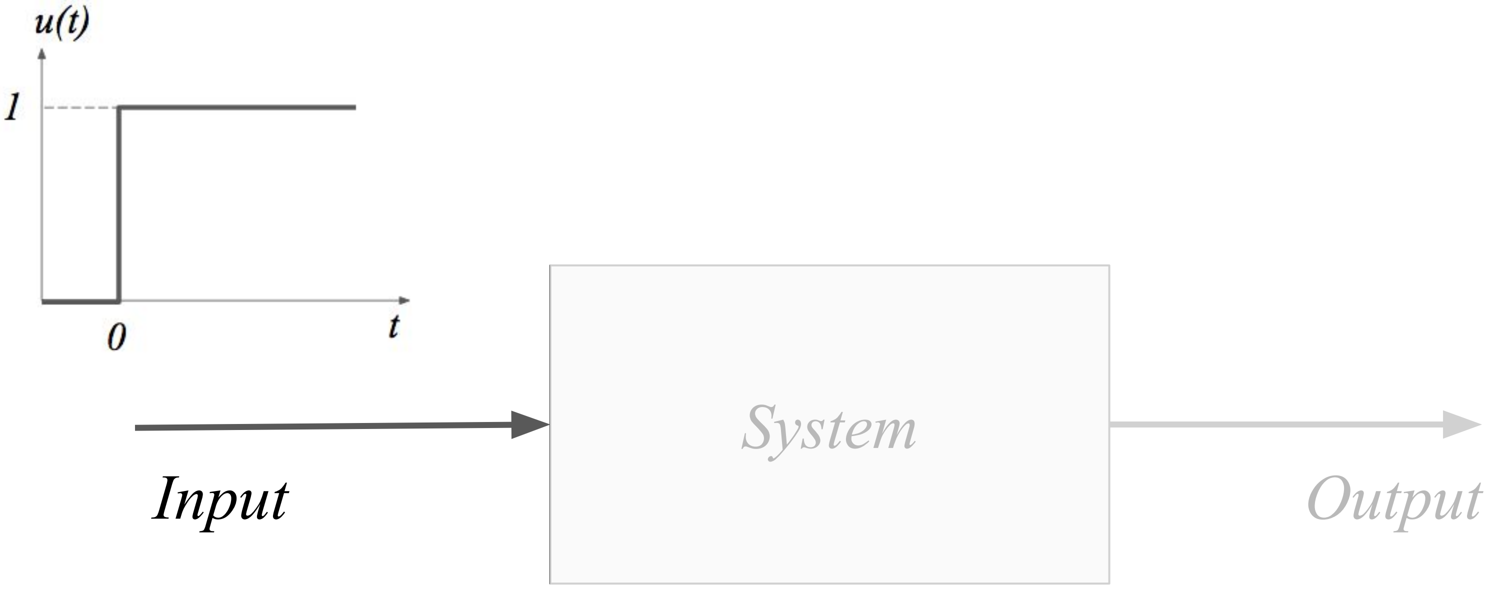
Time Constant *(Greek letter **tau**)*

$$\boldsymbol{\tau}\,\frac{dy}{dt} + y_{(t)} = x_{(t)}$$

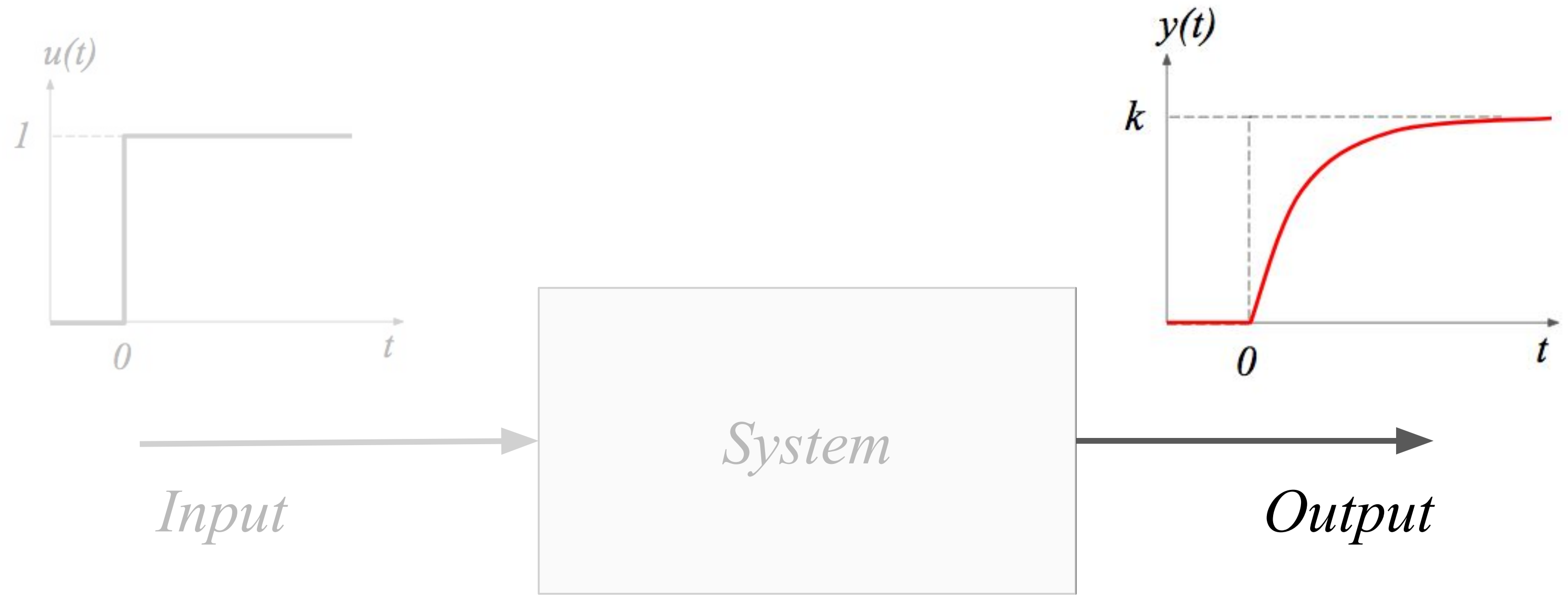
$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

Time response analysis

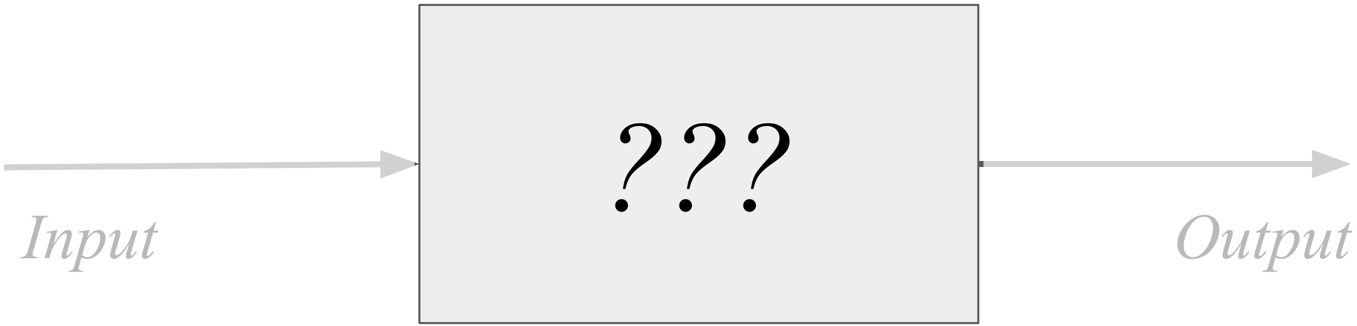

System
Input
Output



A known reference input is injected



The output waveform is measured



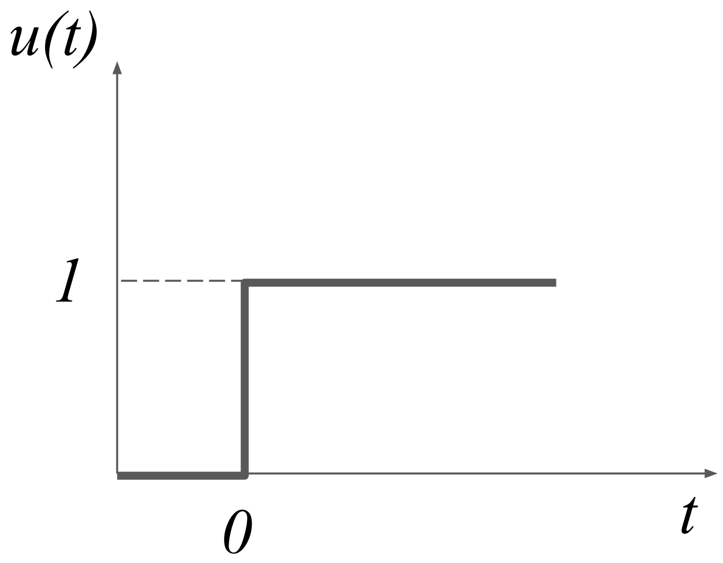
We don't care about what's inside the system

We care about
how the output changes with time
for a given input signal

*A system can be characterized by its response to the **step function**.*

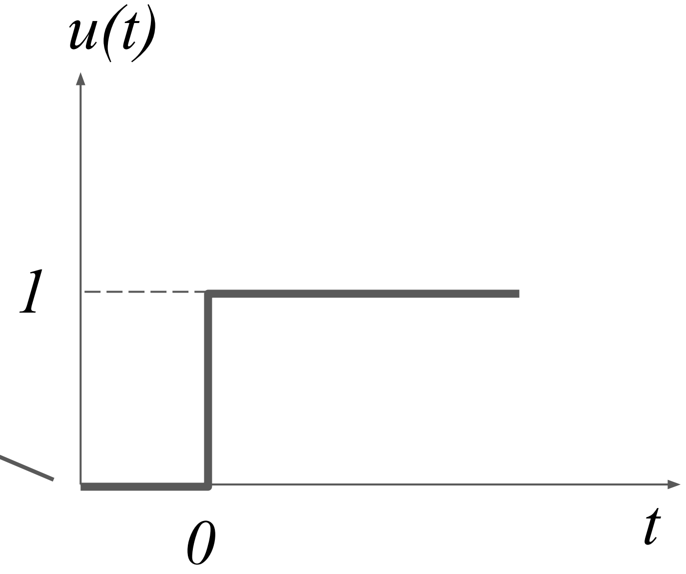
Step function

$$u(t) = \begin{cases} 0, & t < 0 \\ 1, & t \geq 0 \end{cases}$$

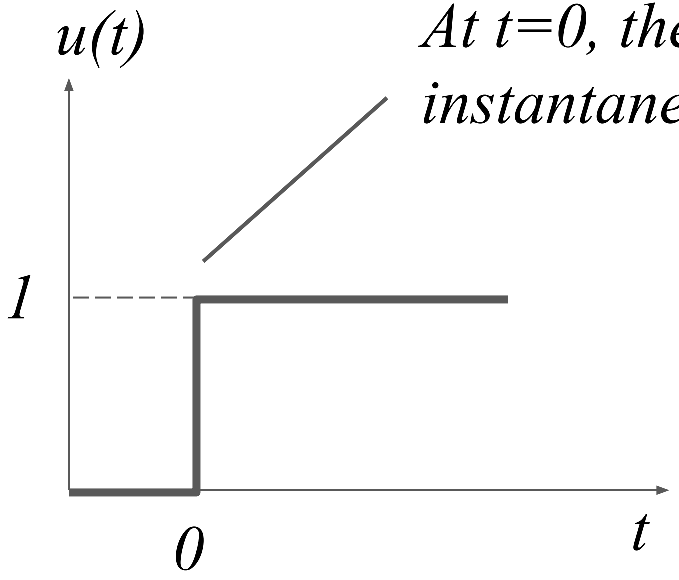


Step function

The input has been maintained with the value 0 for a very long time



Step function



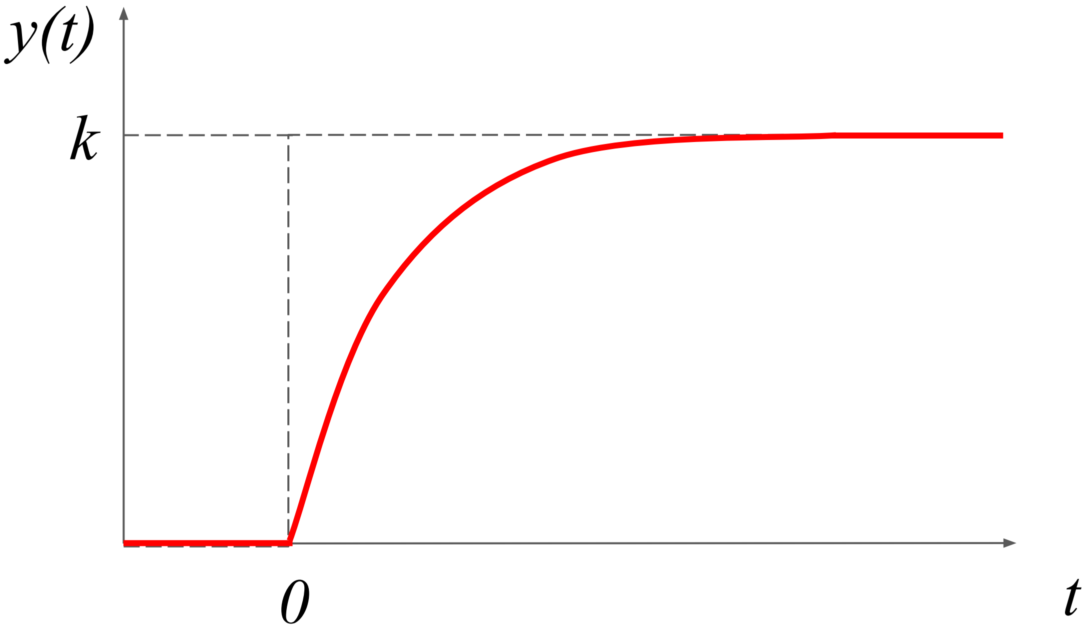
At t=0, the input is taken to 1 instantaneously and held

1st Order System Step Response

$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

The solution to this equation when x(t)=1 for t ≥ 0 and null initial conditions is:

$$y_{(t)} = k(1 - e^{-t/\tau})$$



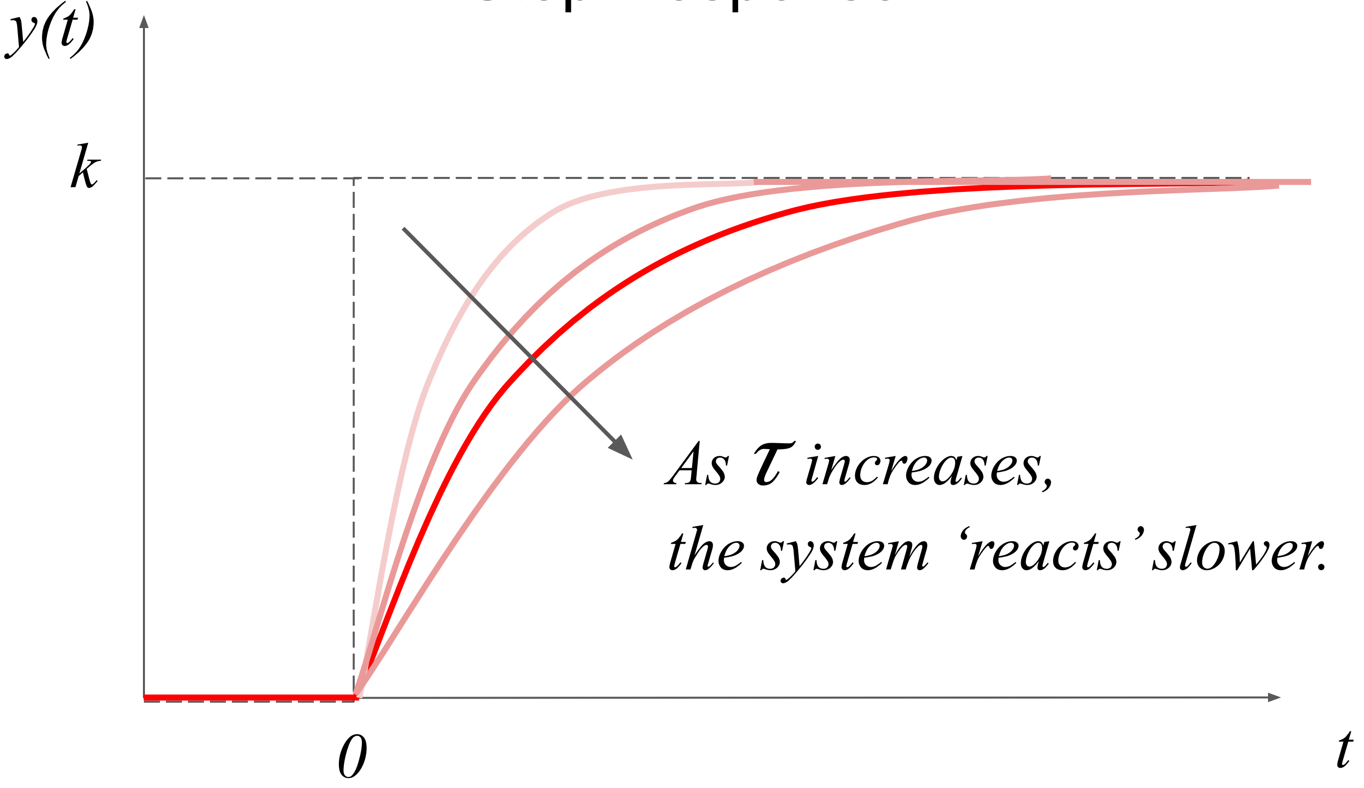
The step response of a 1st order system
is always an exponential curve
that approaches its final value as t increases.

How fast the system reaches its final value?

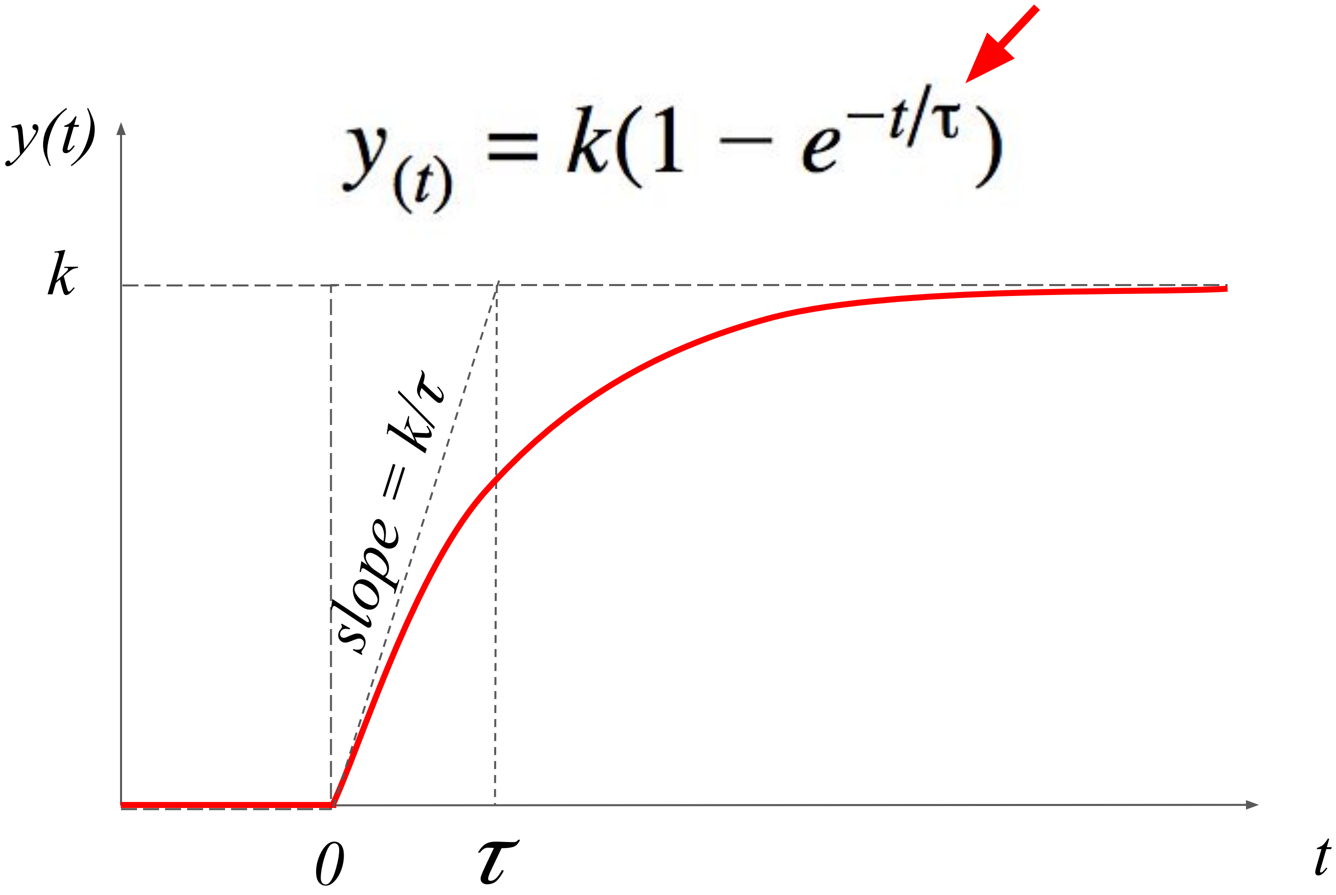
How fast the system reaches its final value?

Depends solely on the system ***Time Constant*** $\boldsymbol{\tau}$

First Order System
Step Response



How slower exactly?

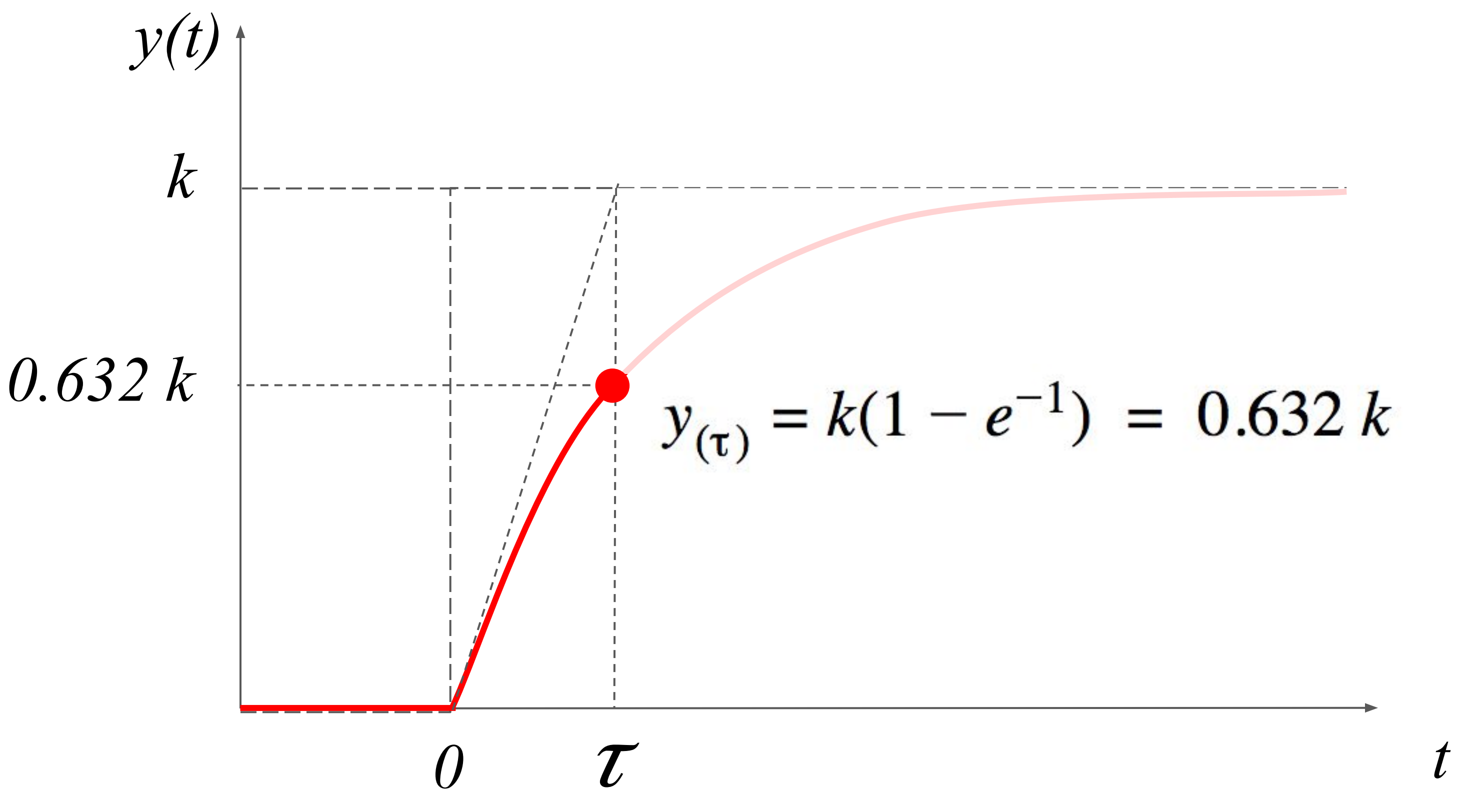


$y_{(t)} = k(1 - e^{-t/\tau})$
y(t)
k
slope = k/τ
0
τ
t

τ

Time Constant

The time it takes the system to reach
63.2% *of the final value*

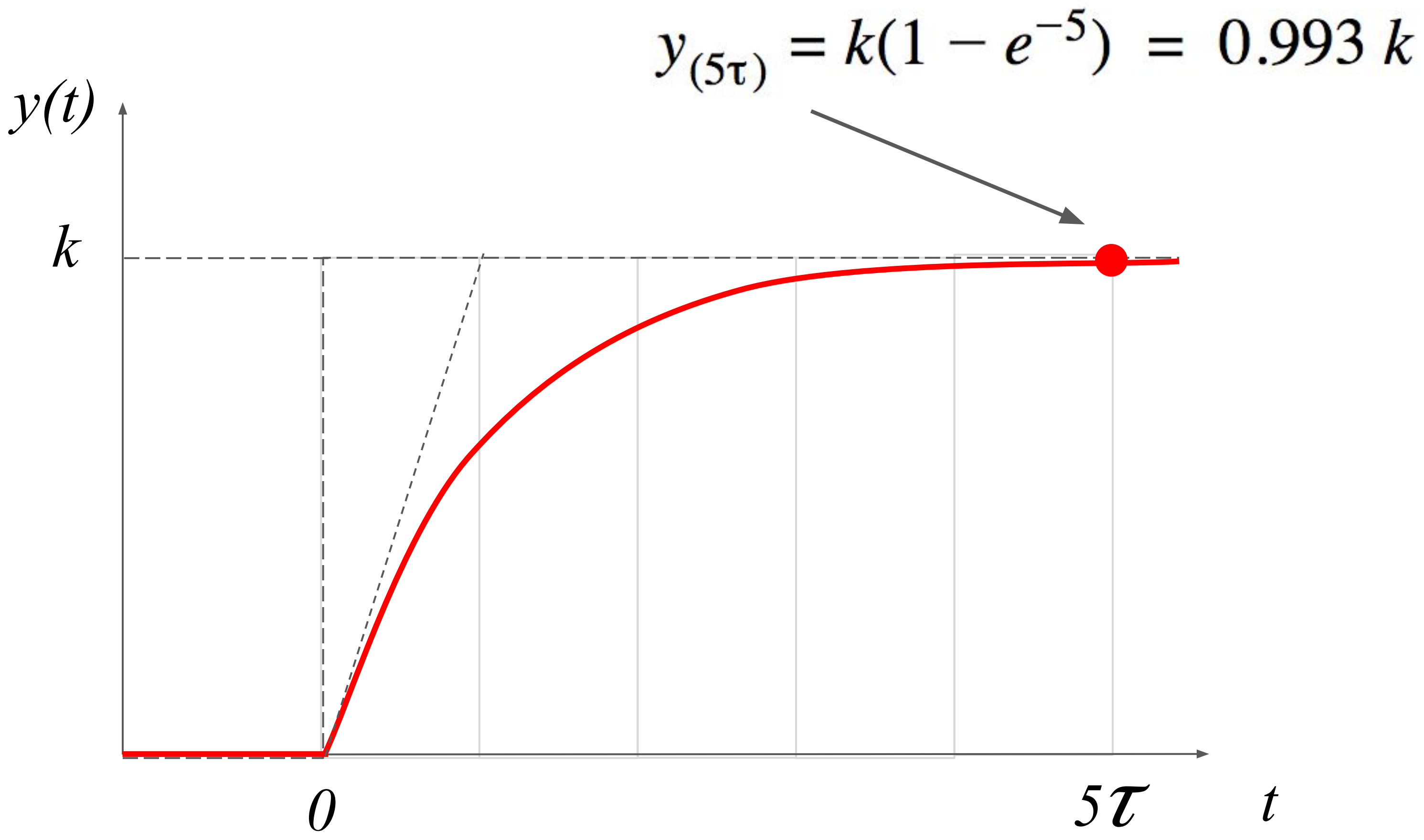


y(t)
k
0.632 k
y(τ) = k(1 − e−1) = 0.632 k
0
τ
t

$$5\tau$$

Five Time Constants

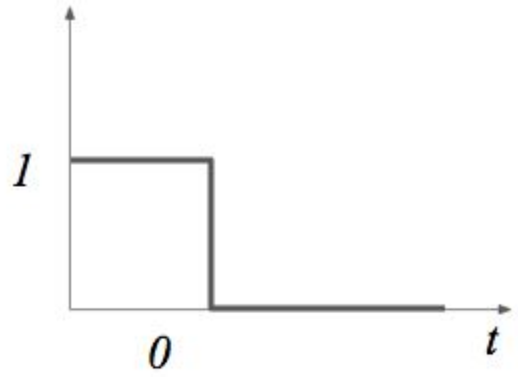
The time at which the output reaches
99.3% *of the final value.*

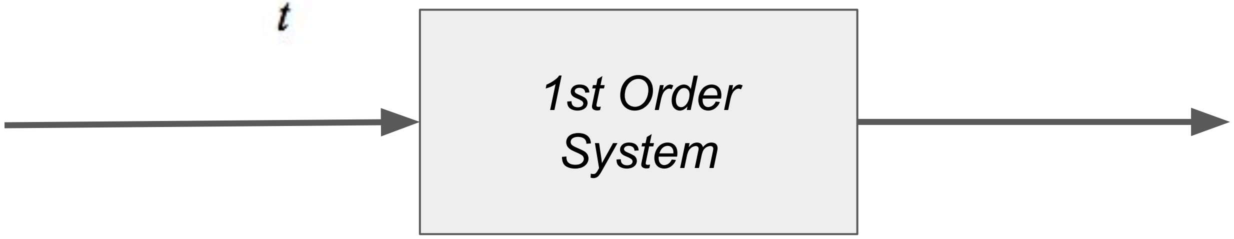

$y_{(5\tau)} = k(1 - e^{-5}) = 0.993\ k$
$y(t)$
k
0
5τ
t

$$5\tau = 99.3\%$$

In practice, it's considered that the output has reached its final value.

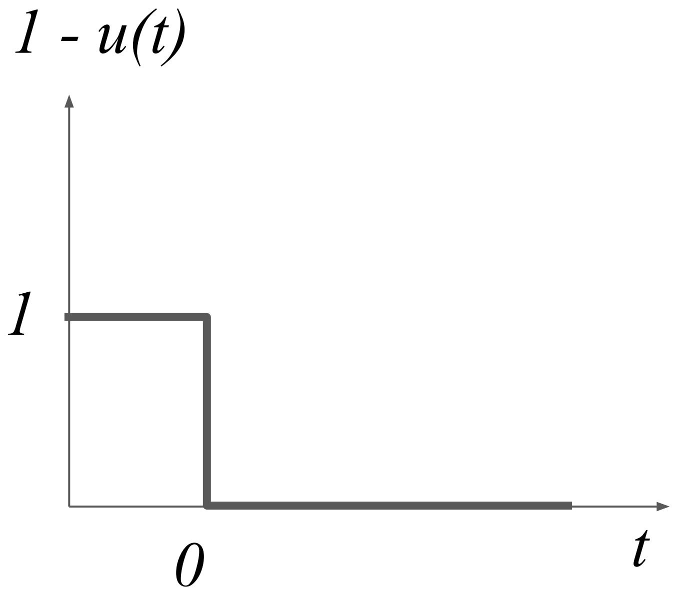
System Decay





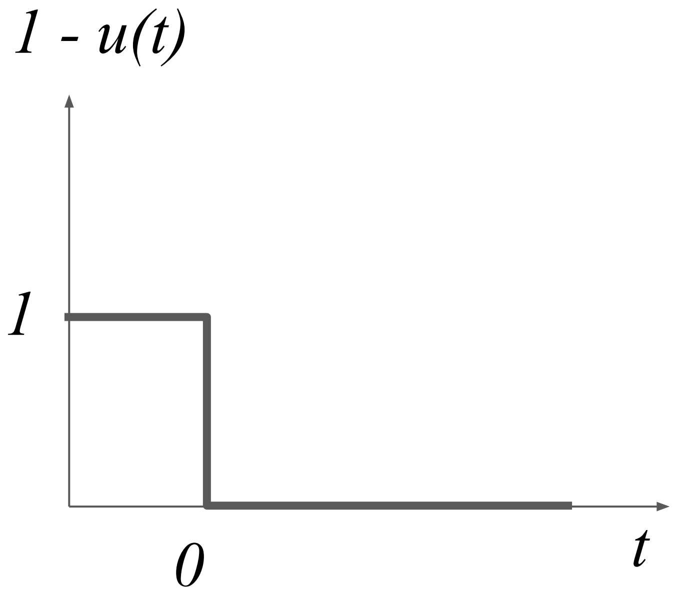
What's the time response when the input is taken to 0?

Input: Inverse Step

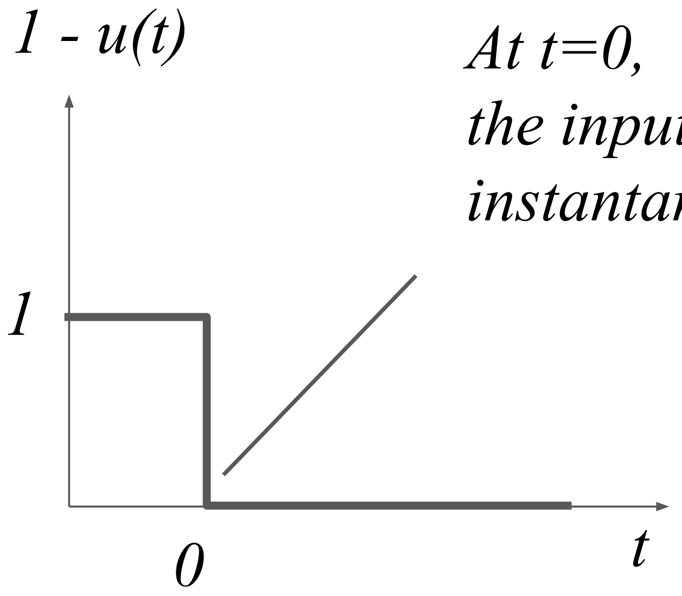


Input: Inverse Step

The input has been maintained with the value 1 for a very long time



Input: Inverse Step



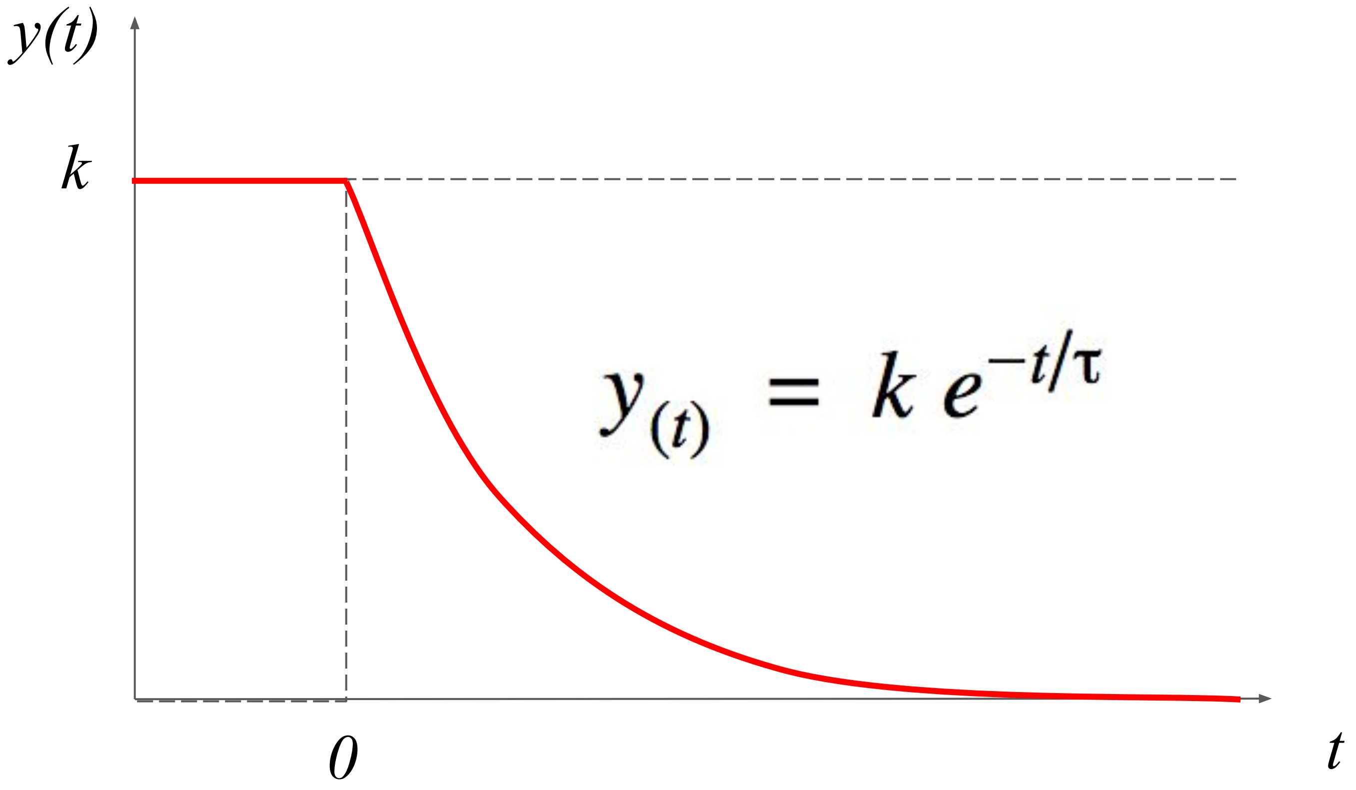
At t=0,
the input is taken to 0
instantaneously

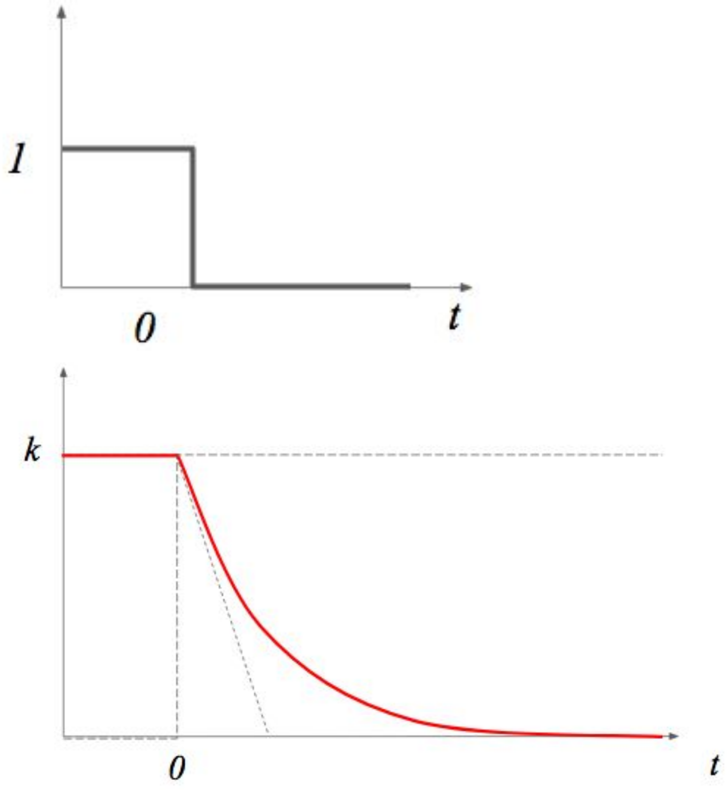
1st Order System Inverse Step Response

$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

The solution to this equation when x(t)=0 for t ≥ 0 and initial condition y(t)=k for t < 0 is:

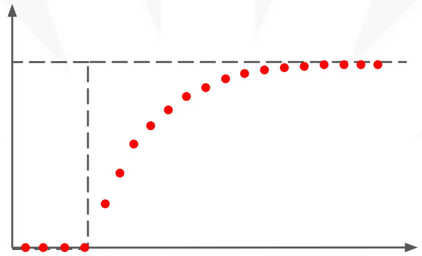
$$y_{(t)} = k\, e^{-t/\tau}$$


y(t)
k
0
t
$y_{(t)} = k\,e^{-t/\tau}$



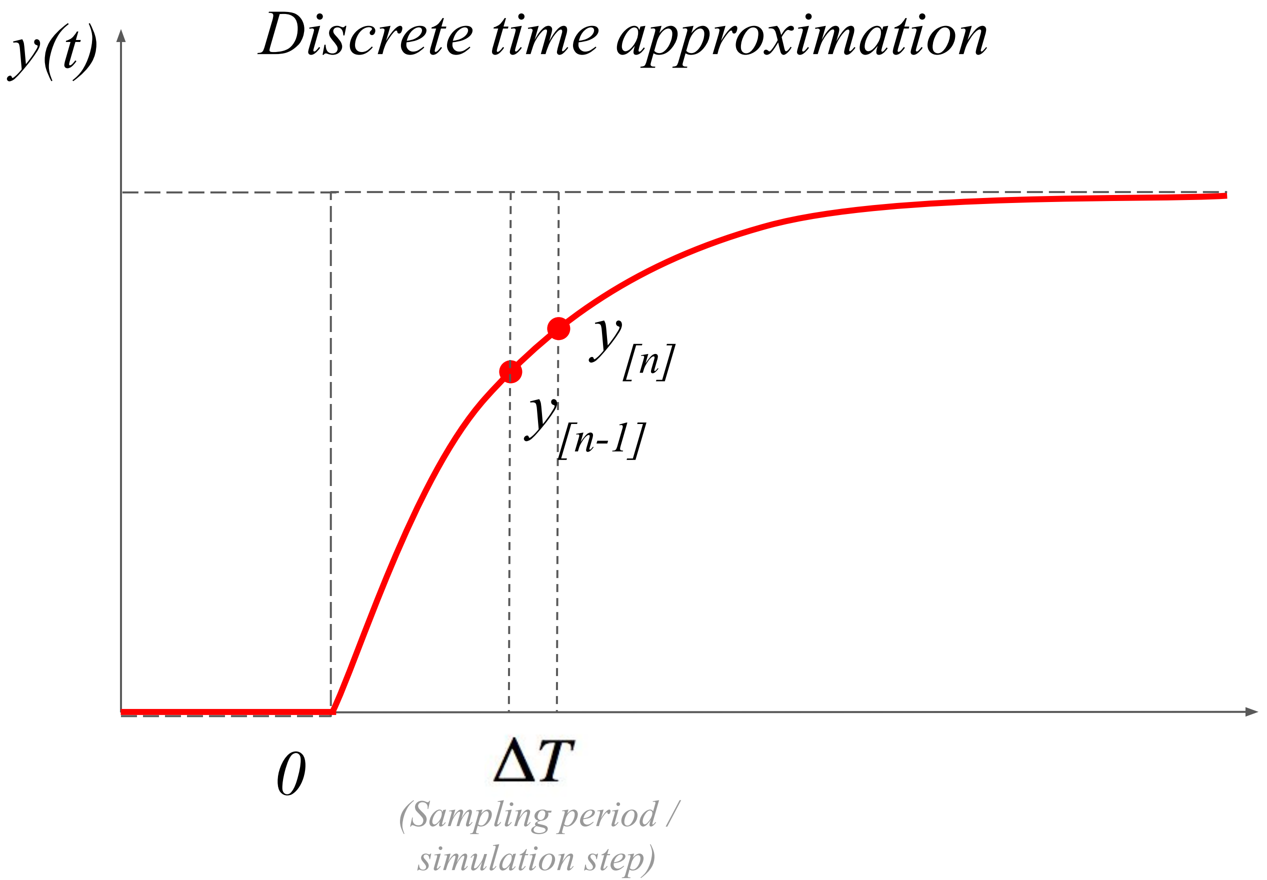
If the input is taken to zero instantaneously, the system still takes some time to release the accumulated energy.

It does it following an ***exponential decay*** *curve.*

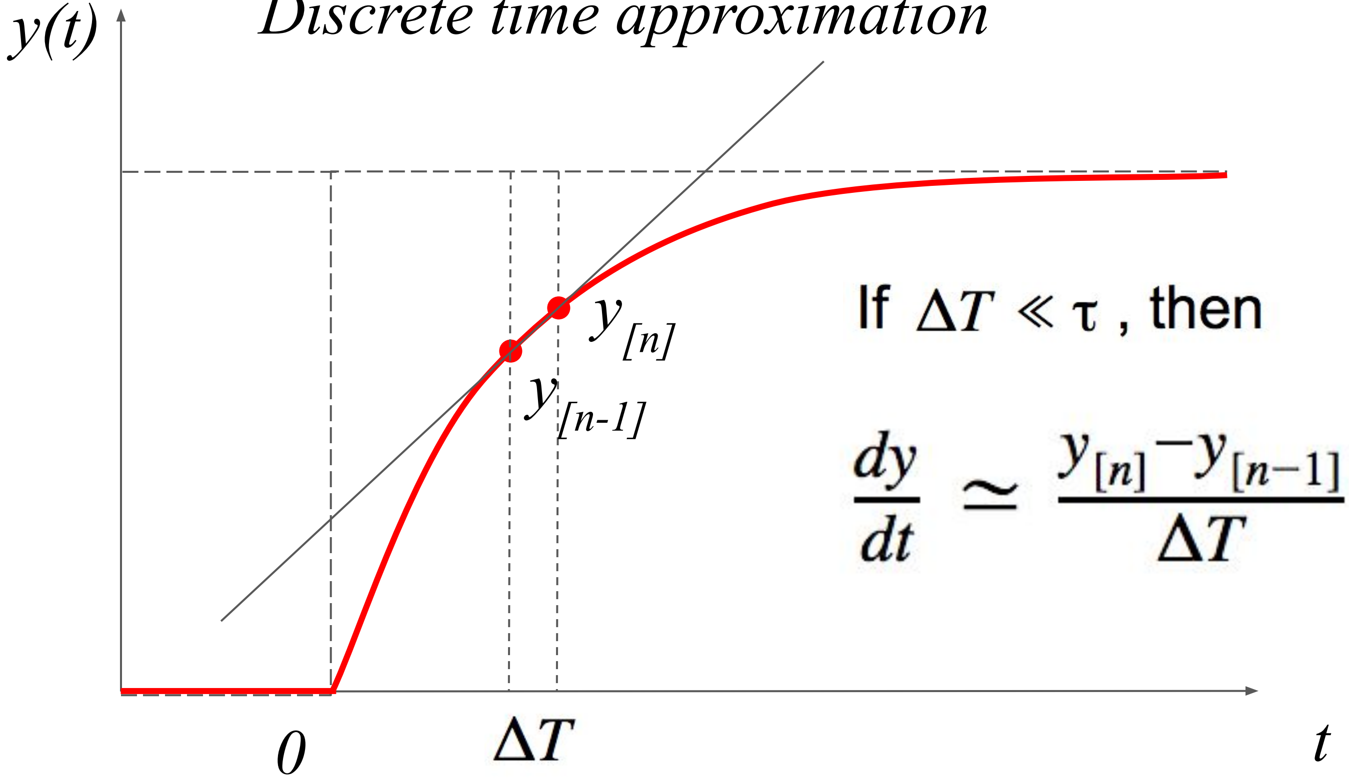


Discrete time approximation

(How to generate these curves in software)


Discrete time approximation
y(t)
y[n]
y[n-1]
0
ΔT
(Sampling period /
simulation step)
t

Discrete time approximation



If $\Delta T \ll \tau$, then

$$\frac{dy}{dt} \simeq \frac{y_{[n]} - y_{[n-1]}}{\Delta T}$$

$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

becomes

$$\tau \frac{y_{[n]} - y_{[n-1]}}{\Delta T} + y_{[n]} = x_{[n]}$$

Solving:

$$y_{[n]} = \left(\frac{\Delta T}{\tau+\Delta T}\right) x_{[n]} + \left(\frac{\tau}{\tau+\Delta T}\right) y_{[n-1]}$$

Renaming:

$$y_{[n]} = \underbrace{\left(\frac{\Delta T}{\tau+\Delta T}\right)}_{a} x_{[n]} + \left(\frac{\tau}{\tau+\Delta T}\right) y_{[n-1]}$$

Renaming:

$$y_{[n]} = \underbrace{\left(\frac{\Delta T}{\tau+\Delta T}\right)}_{a} x_{[n]} + \underbrace{\left(\frac{\tau}{\tau+\Delta T}\right)}_{1\text{ - }a} y_{[n-1]}$$

$$y_{[n]} = a\, x_{[n]} + (1 - a)\, y_{[n-1]}$$

$$y_{[n]} = a\, x_{[n]} + (1 - a)\, y_{[n-1]}$$

This simple equation can approximate the output of any 1st order system.

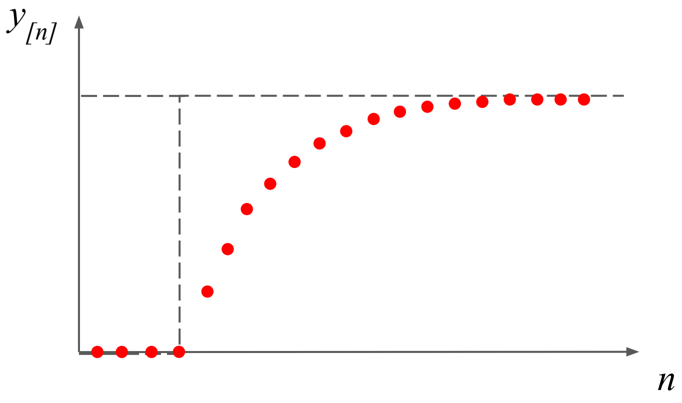
$$y_{[n]} = \underbrace{a\, x_{[n]}}_{\substack{\textit{Depends on} \\ \textbf{\textit{Current input}}}} + \underbrace{(1 - a)\, y_{[n-1]}}_{\substack{\textit{Inertia from} \\ \textbf{\textit{Previous output}}}}$$

$$y_{[n]} = a\,x_{[n]} + (1 - a)\,y_{[n-1]}$$

$$a = \frac{\Delta T}{\tau + \Delta T}$$

NOTE

Keep in mind all graphs presented are qualitative with exaggerated distances for clarity.

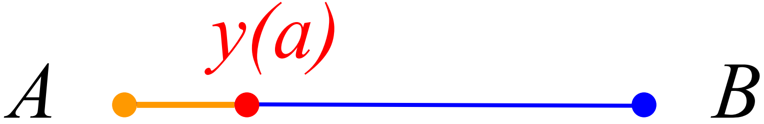


The continuous analog version of the curve is overlaid for clarity but an actual digital signal is just a bunch of ordered discrete values.

Interpretation as a
linear interpolation

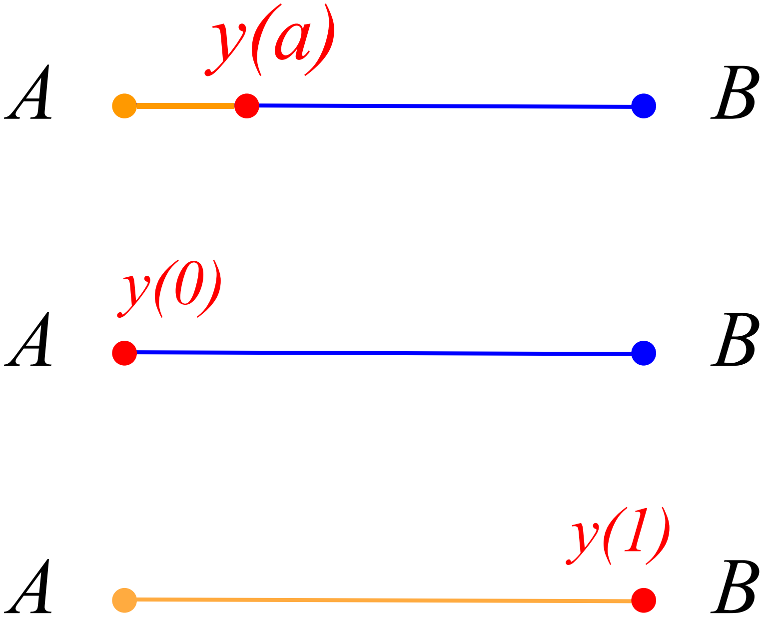
Linear Interpolation (Lerp)

$$y(a) = aB + (1 - a)A$$



$$0 \leq a \leq 1$$

Linear Interpolation (Lerp)



$$y(a) = aB + (1 - a)A$$

$$\text{If } a = 0 \text{ then } y = A$$

$$\text{If } a = 1 \text{ then } y = B$$

The 1st Order step response looks like a Linear Interpolation

$$y_{[n]} = a\, x_{[n]} + (1 - a)\, y_{[n-1]}$$

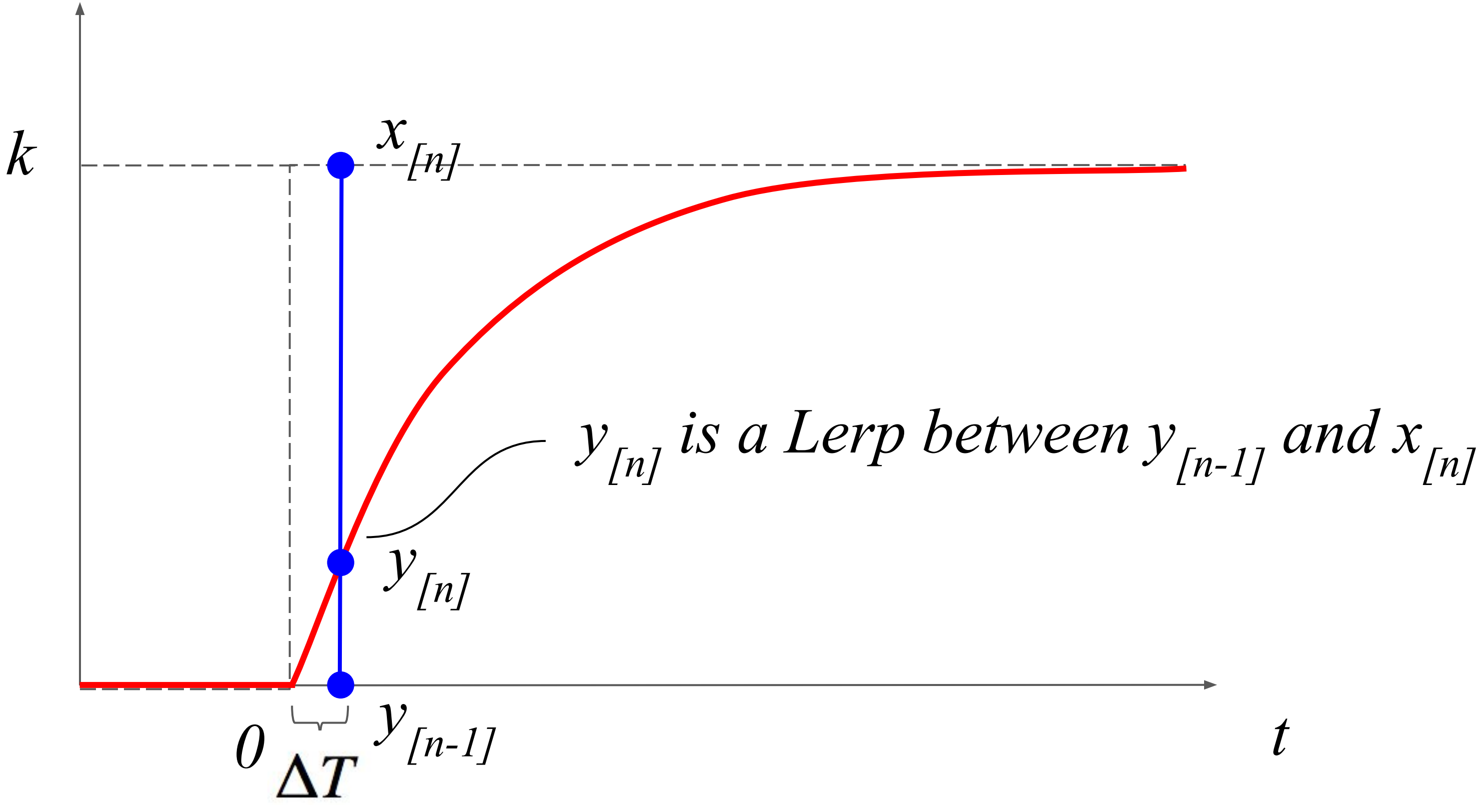
The 1st Order step response looks like a Linear Interpolation

$$y_{[n]} = a\,x_{[n]} + (1-a)\,y_{[n-1]}$$

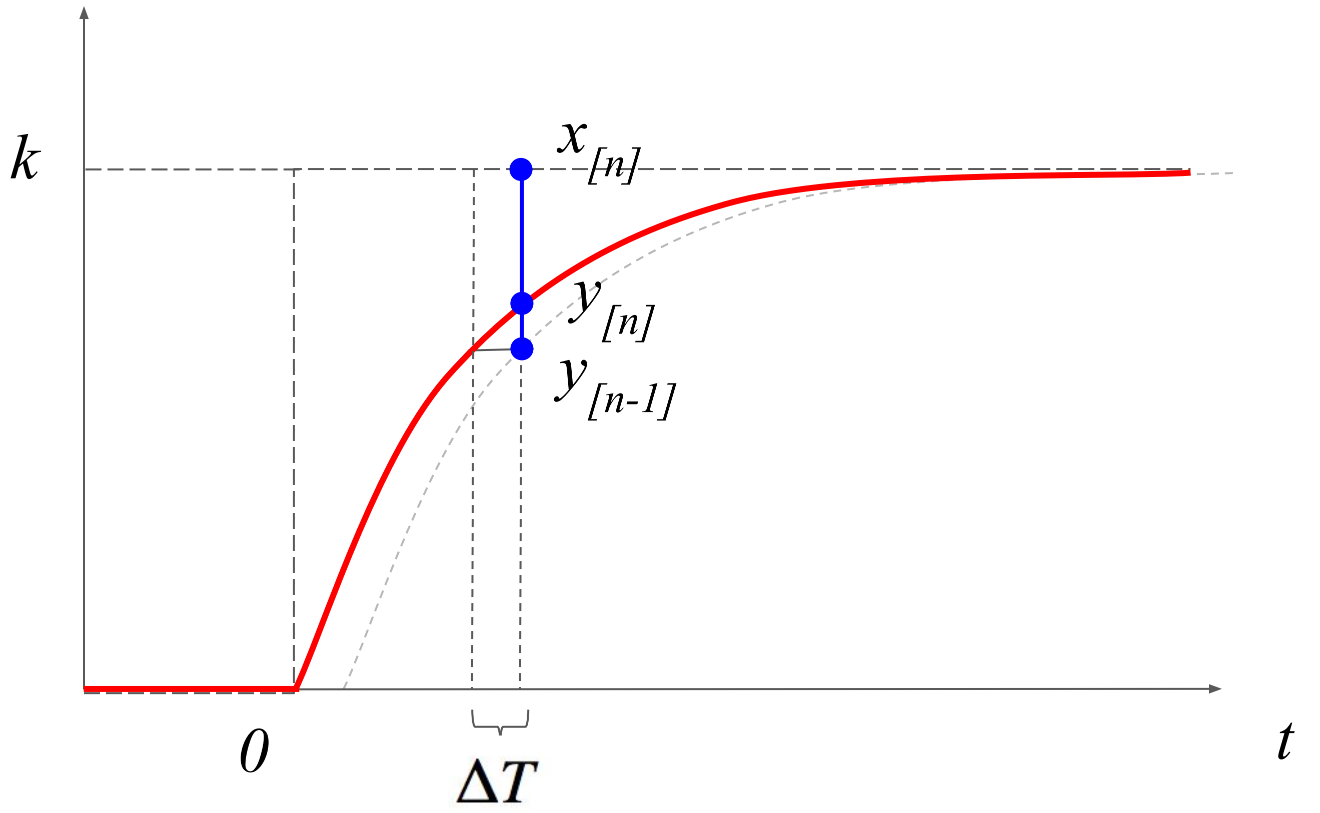
🤔 *Why?*

*At instant **n**,*
***y** is actually a linear interpolation*
*between previous output **y** and current input **x**.*

$$y_{[n]} = (1 - a)\, y_{[n-1]} + a\, x_{[n]}$$

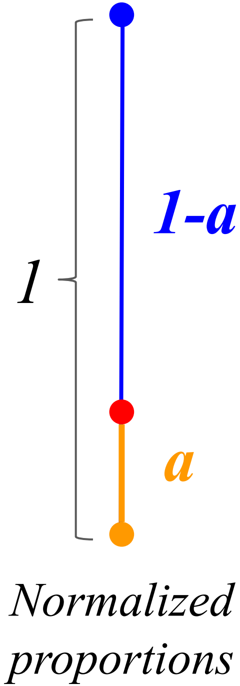


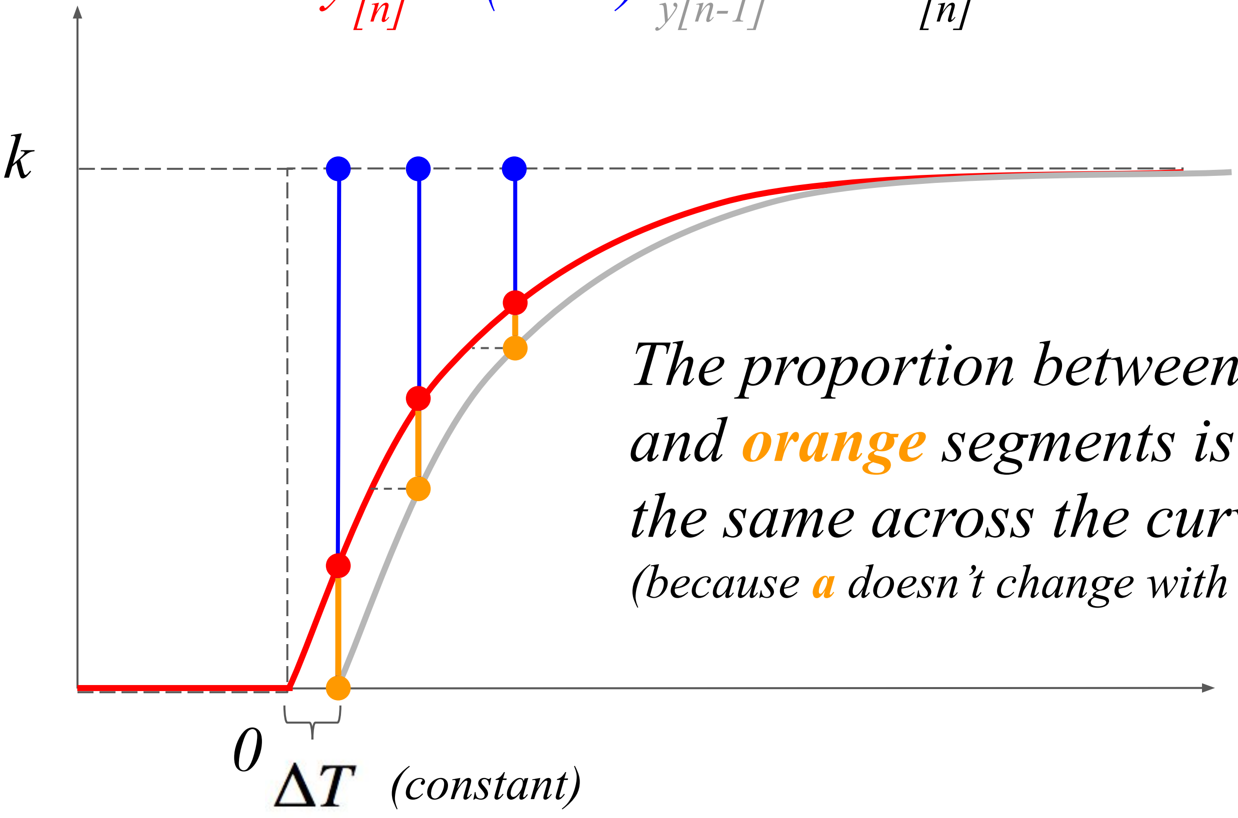
$$y_{[n]} = (1 - a)\, y_{[n-1]} + a\, x_{[n]}$$



If ΔT is constant

$$y_{[n]} = (1 - a)\; y_{[n-1]} + a\, x_{[n]}$$





*The proportion between **blue** and **orange** segments is always the same across the curve.*

*(because **a** doesn't change with time)*

Which totally makes sense since:

The grow rate of an exponential function is directly proportional to the value of the function.

What if If ΔT is not constant?

On real-time digital systems, the elapsed time between simulation steps or frames might not be constant!

The equation still approximates an exponential curve!

$$a = \frac{\Delta T}{\tau + \Delta T}$$

The ΔT factor compensates small variations.

(given ΔT varies reasonably)

Approximation accuracy

Further simplification:

$$\text{If } \Delta T \ll \tau \implies \tau + \Delta T \simeq \tau$$

*Then **a** can be approximated as*

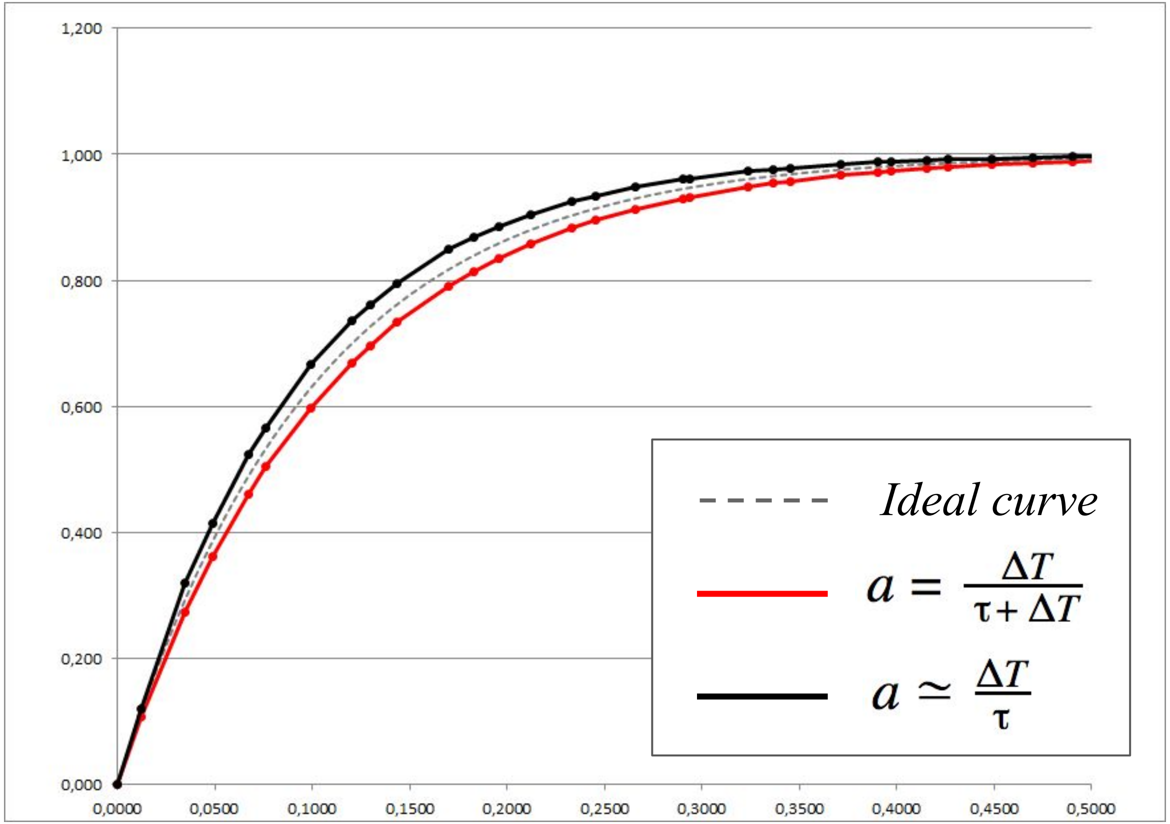
$$a \simeq \frac{\Delta T}{\tau}$$

$$a \simeq \frac{\Delta T}{\tau}$$

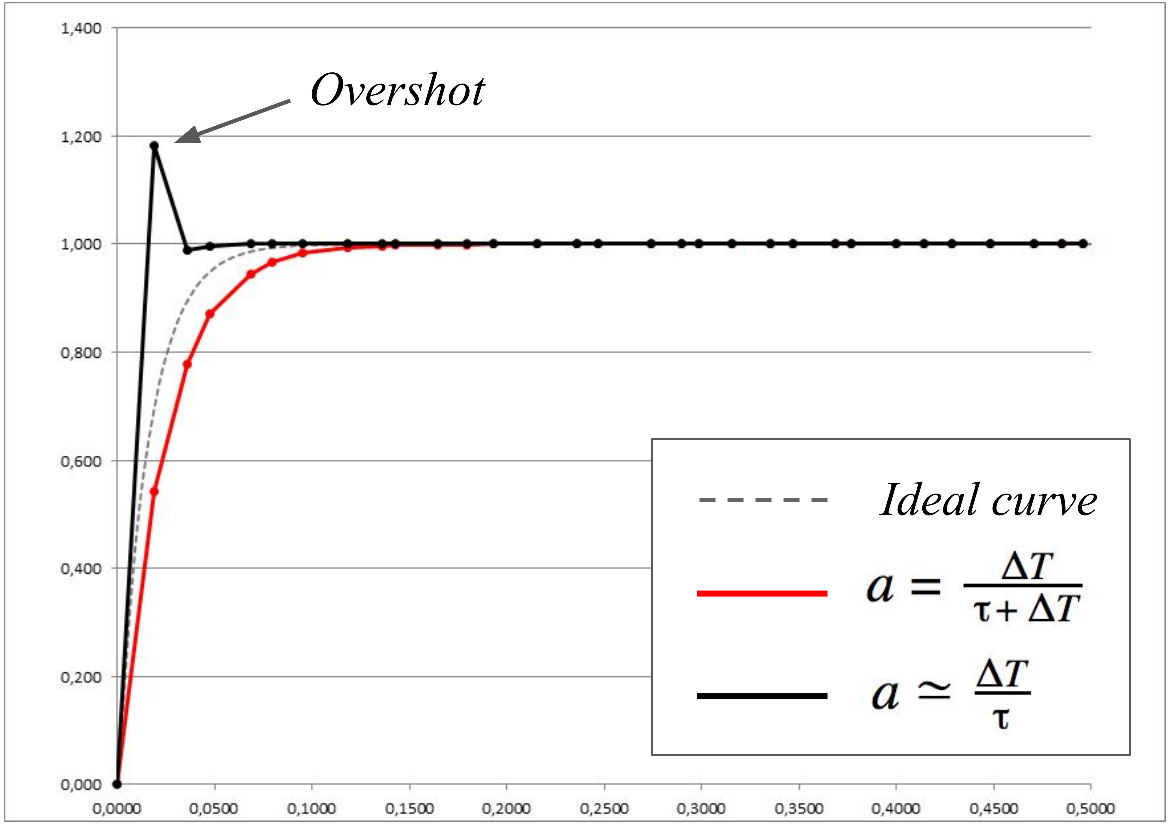
This approximation can only be used if ΔT is at least ***10 times smaller*** *than τ.*

Some examples

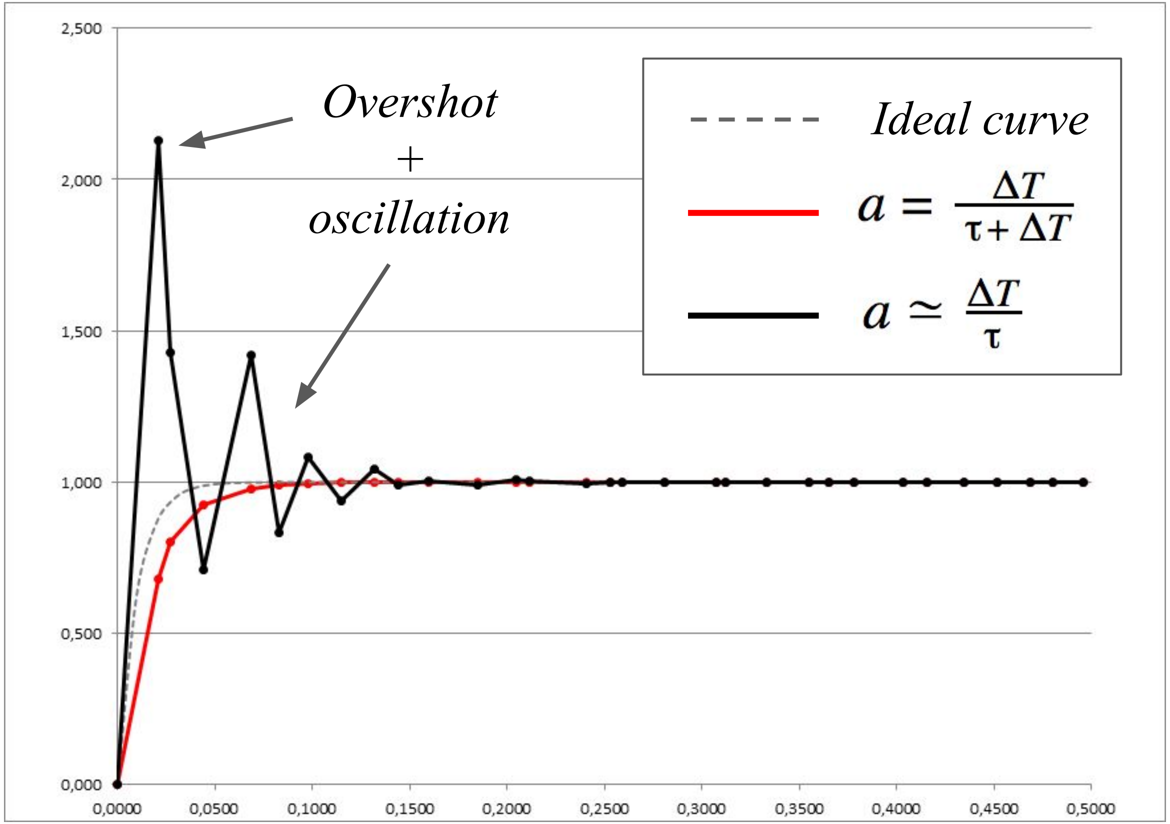
$\tau = 100ms,\ \Delta T \simeq 16ms$



$\tau = 16ms, \Delta T \simeq 16ms$



$\tau = 10ms,\ \Delta T \simeq 16ms$



NOTE

In real physical systems, a 1st order system will never produce an overshot or oscillation. Only 2nd or higher order systems can produce overshots.

This is just a side effect of a discrete time approximation going too off.

Interpolation Implementation

Since ***Linear Interpolation*** *is such a basic operation, the first order time response can be implemented virtually anywhere using the proper built-in function.*

*Since **Linear Interpolation** is such a basic operation, the first order time response can be implemented virtually anywhere using the proper built-in function.*

Some examples:

GLSL:	`y = mix(x, y, a);`
HLSL:	`y = lerp(x, y, a);`
Unreal:	`y = FMath::Lerp(x, y, a);`
Unity:	`y = Mathf.Lerp(x, y, a);`

Besides a linear interpolation implementation, most environments provide ***more specific and handy methods*** *to achieve this behavior.*

*Besides a linear interpolation implementation, most environments provide **more specific and handy methods** to achieve this behavior.*

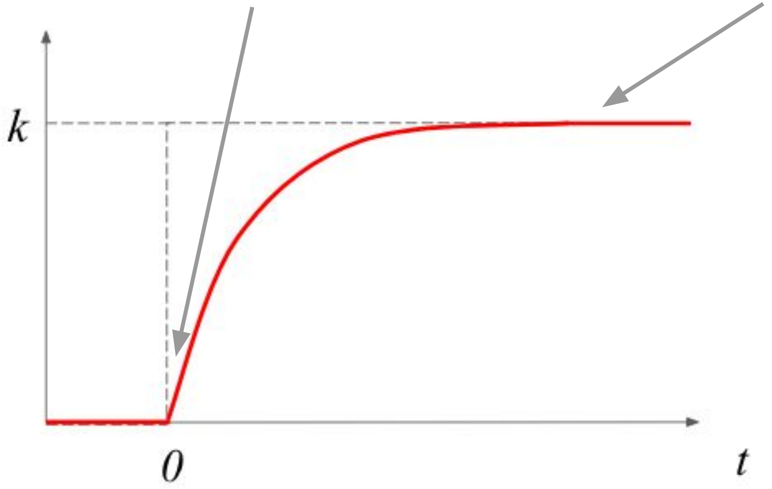
Example:*Unreal's* `InterpTo`

FMath::FInterpTo

```
/** Interpolate float from Current to Target. Scaled by distance
to Target, so it has a strong start speed and ease out. */
```

FMath::FInterpTo

/** Interpolate float from Current to Target. Scaled by distance to Target, so it has a **strong start** speed and **ease out**. */



```
static CORE_API float FInterpTo(
    float Current,
    float Target,
    float DeltaTime,
    float InterpSpeed
);
```

```
static CORE_API float FInterpTo(
    float Current,        ←  y[n-1]
    float Target,         ←  x[n]
    float DeltaTime,      ←  ΔT
    float InterpSpeed     ←  What is exactly this?
);
```

```
static CORE_API float FInterpTo(
    float Current,
    float Target,
    float DeltaTime,
    float InterpSpeed
);
```

Current ← $y[n-1]$

Target ← $x[n]$

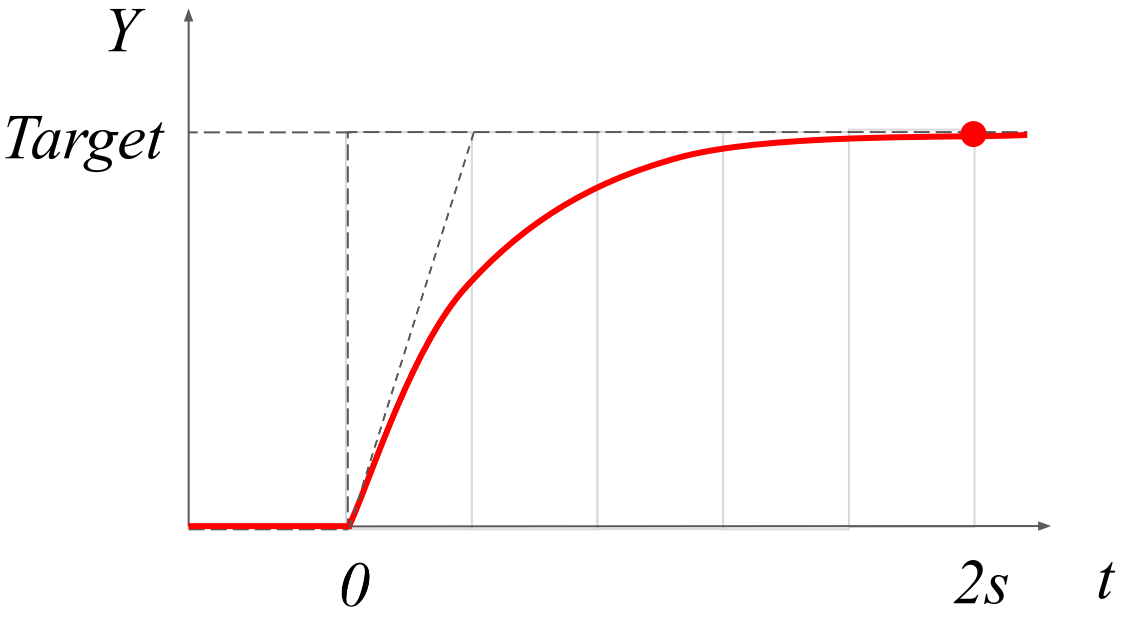
DeltaTime ← ΔT

InterpSpeed ← *What is exactly this?*

$$InterpSpeed = \frac{1}{\tau + \Delta T} \simeq \frac{1}{\tau}$$

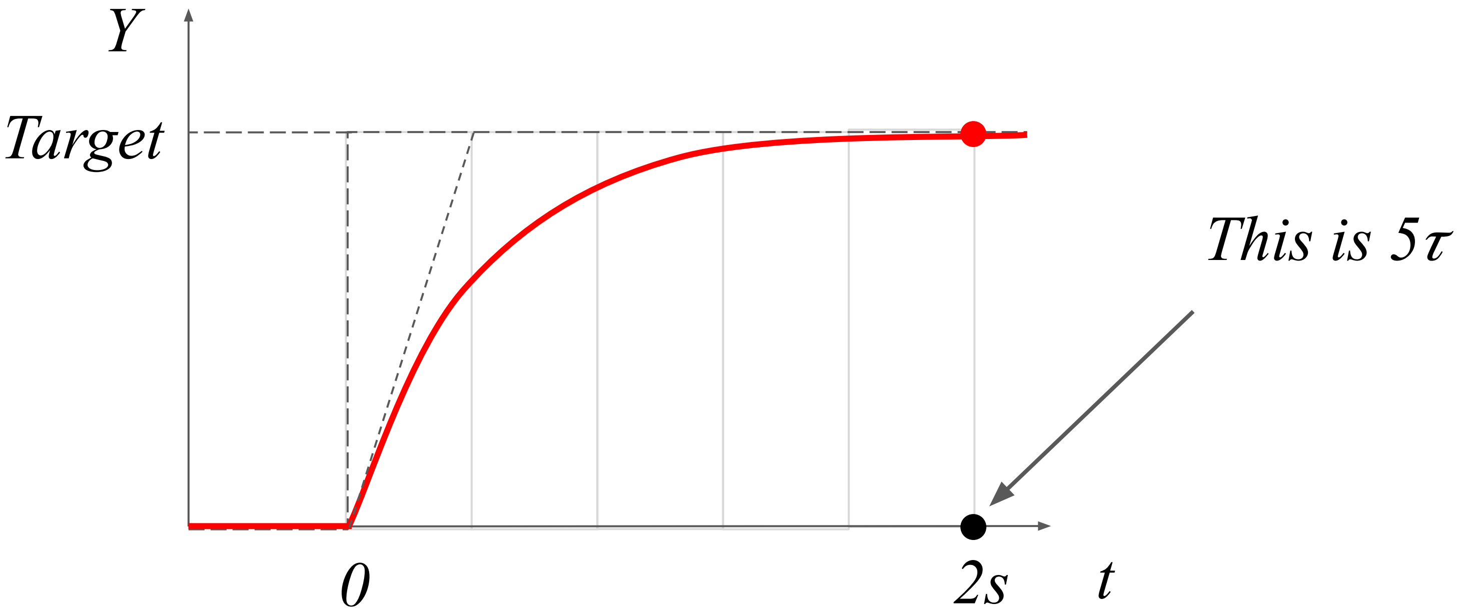
EXAMPLE:

We want the output to reach its target value in 2 seconds.



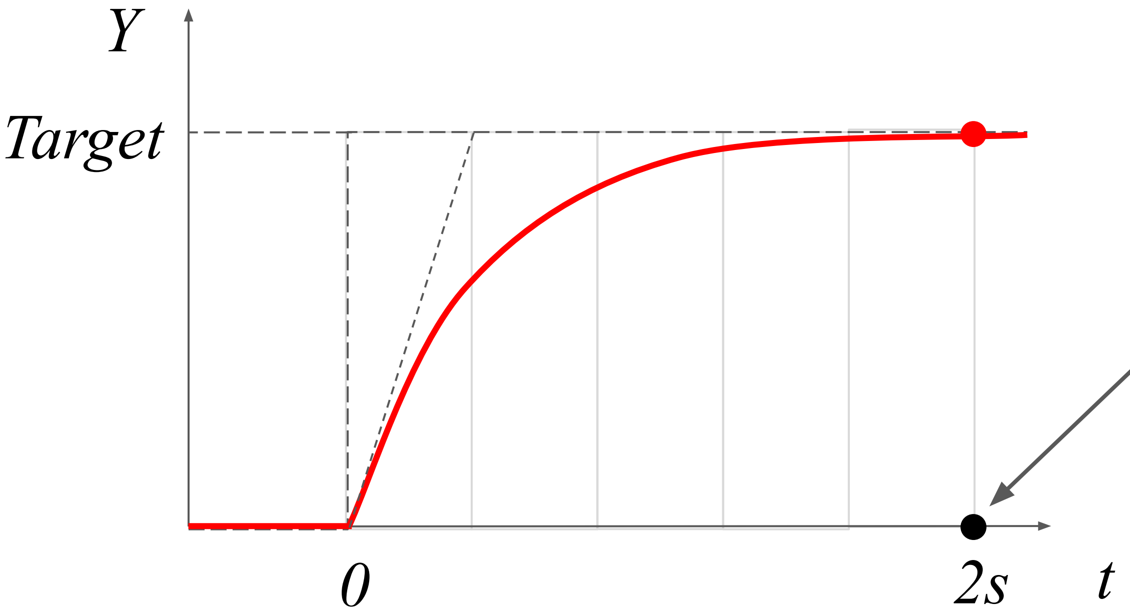
EXAMPLE:

We want the output to reach its target value in 2 seconds.



EXAMPLE:

We want the output to reach its target value in 2 seconds.



This is 5τ

$$5\tau = 2s$$
$$\tau = 0.4s$$

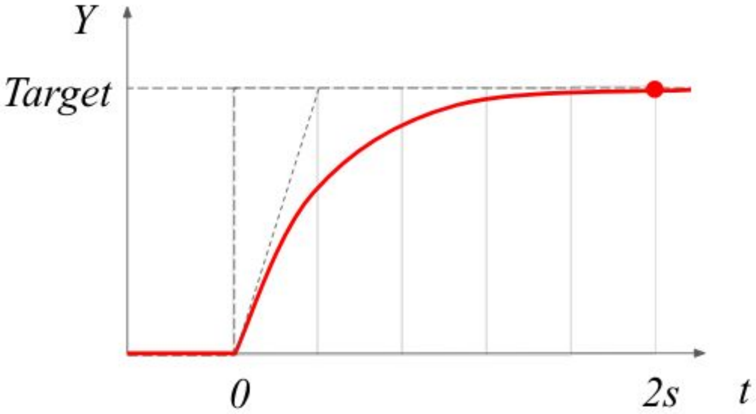
If running at about 60fps,

ΔT is 25 times smaller than τ

*Then, **a** can be roughly approximated to $\Delta T/\tau$*

This means that InterpSpeed = $1/\tau = 2.5$

```cpp
void AMyActor::Tick(float DeltaTime)
{
    // This will make Y to go from its current value
    // to Target in about 2 seconds.
    Y = FMath::FInterpTo(Y, Target, DeltaTime, 2.5f);
}
```



Examples of possible uses in games

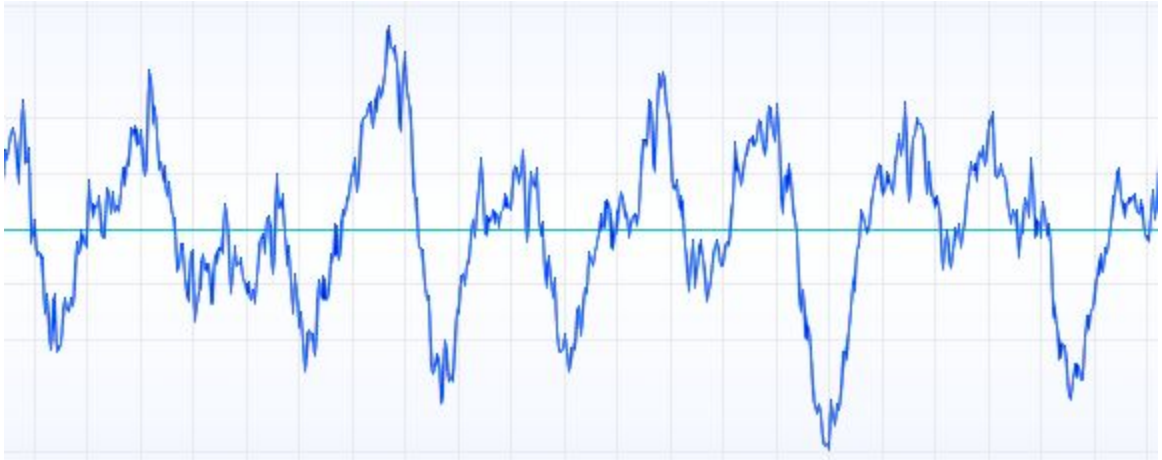
- *Simple control of vehicles acceleration / deceleration.*
- *Sliding door animation interpolation.*
- *Camera animation interpolation.*
- *UI elements animation interpolation.*

1st order system as a digital filter

What if the input is not a step,
but an arbitrary time changing signal?

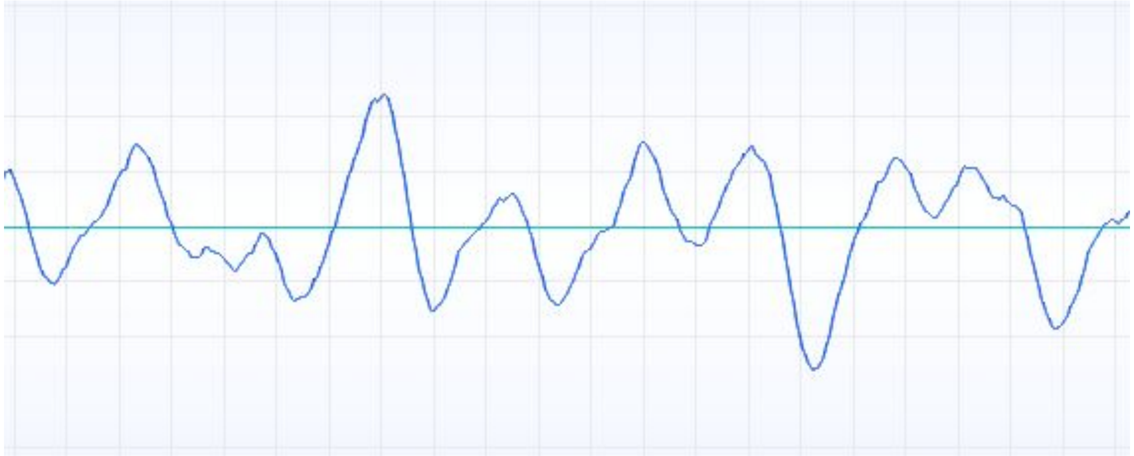
Effect on an arbitrary input signal

Input:



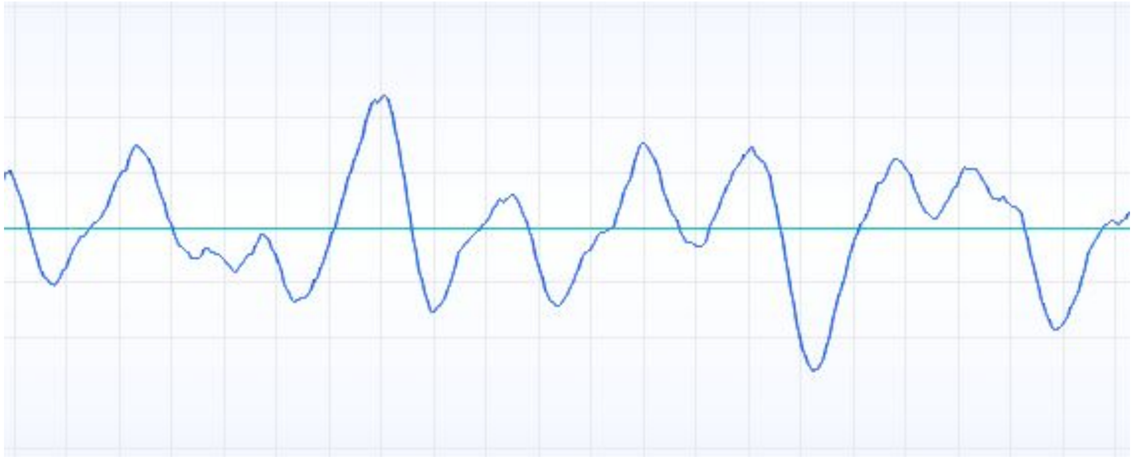
Effect on an arbitrary input signal

Output:



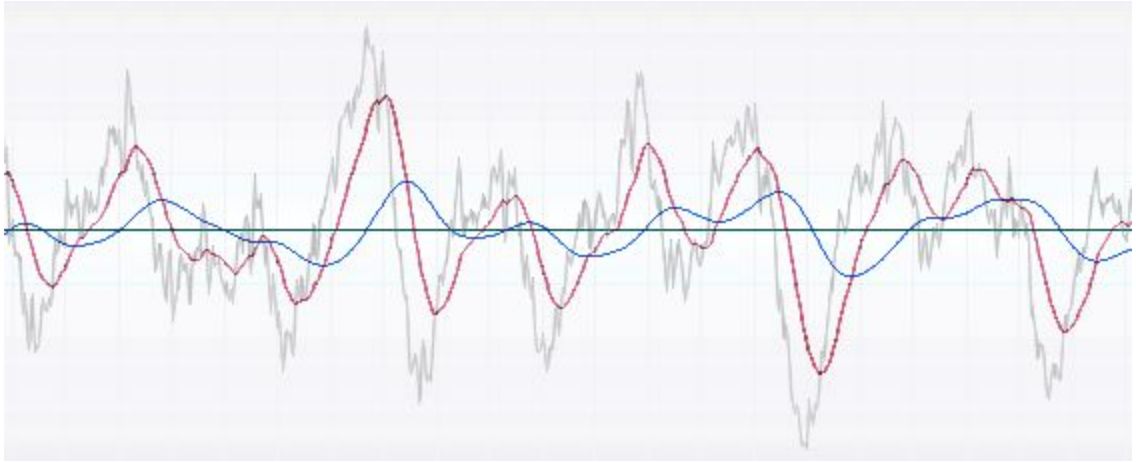
Effect on an arbitrary input signal

Output:



The system smooths out the signal

Effect on an arbitrary input signal



Input

Output with $\boldsymbol{\tau} = \boldsymbol{\tau 1}$

Output with $\boldsymbol{\tau} = \boldsymbol{\tau 2}$

$\boldsymbol{\tau 2} > \boldsymbol{\tau 1}$

As $\boldsymbol{\tau}$ *increases, the system reacts slower, filtering out more signal.*

In Digital Signal Processing,
this is called a ***Low Pass Filter***

In Digital Signal Processing,
*this is called a **Low Pass Filter***

Because it let low frequencies pass, while blocking the high frequencies of the signal.

Examples of possible uses in games

- *Smooth user input*
- *Smooth camera transitions*
- *Smooth "jumpy" incoming network data.*
- *Temporally blend frames in shaders.*

Conclusion

Conclusion

The 1st order system discrete time approximation formula is a very powerful tool that allows extremely simple yet organic simulations without the need to physically model a system.

Conclusion

All you need to remember is this simple formula:

$$\boxed{y_{[n]} = a\, x_{[n]} + (1 - a)\, y_{[n-1]}} \qquad a = \frac{\Delta T}{\tau + \Delta T}$$

Conclusion

All you need to know to remember is this simple formula:

```
y = a*x + (1-a)*y;
```

$$a = \frac{\Delta T}{\tau + \Delta T}$$

Conclusion

All you need to remember is this simple formula:

```
y = a*x + (1-a)*y;
```

$$a = \frac{\Delta T}{\tau + \Delta T}$$

*Also, beware of the parameter **a** magnitude in relation to the simulation step of your system.*

Appendix A

Typical 1st order systems time constants

τ for typical 1st order systems

Type of System	Time Constant
translating friction-mass	m/b
translating friction-spring	b/k
rotating friction-flywheel	J/B_r
rotating friction-spring	B_r/K_r
resistor-capacitor	R·C
resistor-inductor	L/R
thermal	R·C

Appendix B

Exponential decay derivation

1st order general equation:

$$\tau \frac{dy}{dt} + y_{(t)} = x_{(t)}$$

Nulling x(t):

$$\tau \frac{dy}{dt} + y_{(t)} = \cancelto{0}{x_{(t)}}$$

x(t) is 0 for t ≥ 0

Rearranging:

$$\tau \frac{dy}{dt} = -\, y_{(t)}$$

$$\frac{dy}{y_{(t)}} = \frac{-dt}{\tau}$$

Integrating: $$\ln y_{(t)} = -\frac{t}{\tau} + C$$ *being C the integration constant*

Solving y(t): $$y_{(t)} = e^{C} e^{-t/\tau}$$

Defining $k = e^{C}$, as the value the system had in steady state:

$$\boxed{y_{(t)} = k\, e^{-t/\tau}}$$

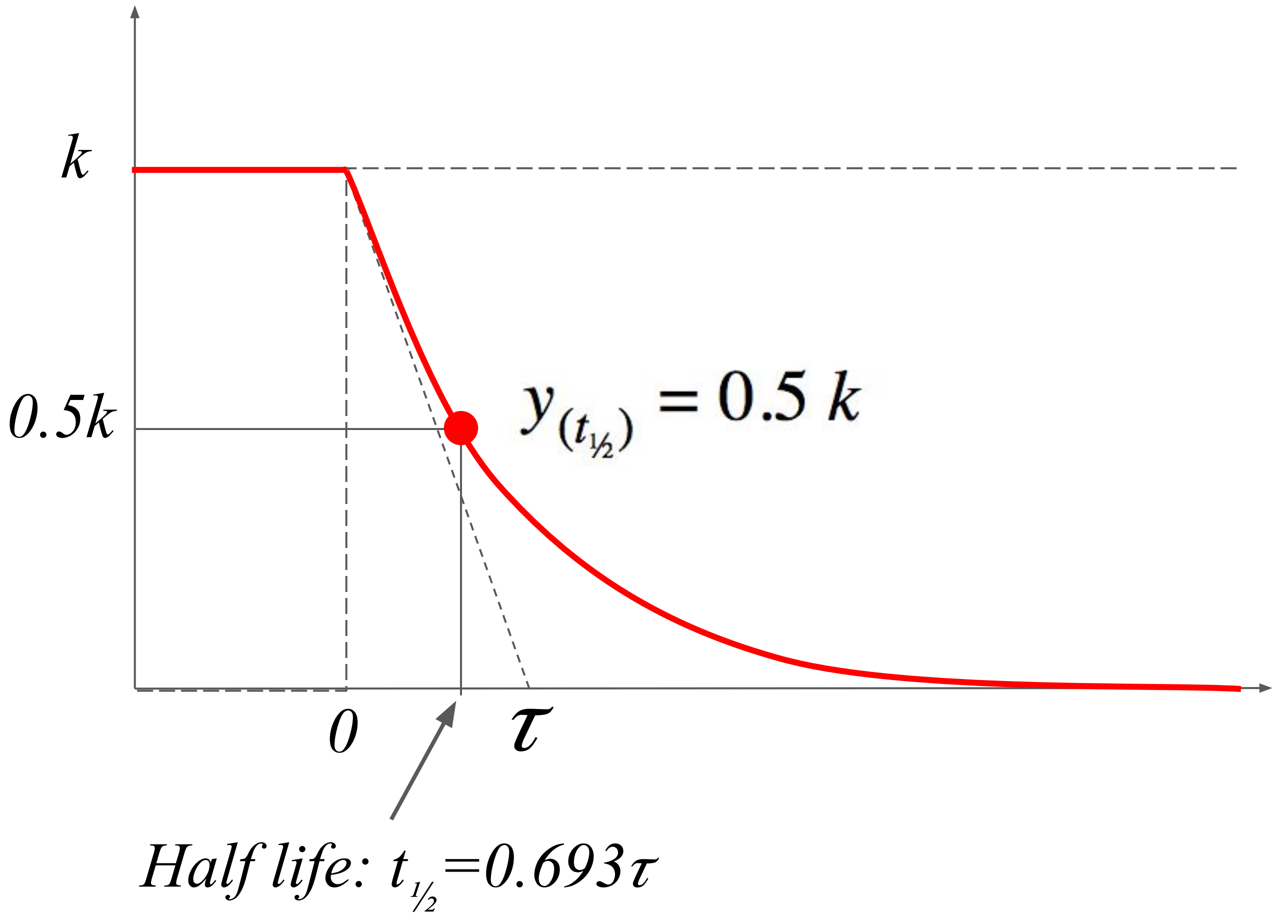
Appendix C

Half Life

Half Life ($t_{½}$)

The time required by a decaying quantity to reduce to half its initial value

$$\frac{1}{2} = ke^{-t_{½}/\tau} \quad \Longrightarrow \quad t_{½} = ln(2)\,\tau$$


k
$0.5k$
$y_{(t_{1/2})} = 0.5\,k$
0
τ
t
Half life: $t_{1/2}$=0.693τ

In some fields (like nuclear physics),
the exponential decay equation is defined in terms of
the inverse of the Time Constant τ :

The ***Decay Constant*** $\lambda = \frac{1}{\tau}$

$$y(t) = ke^{-\lambda t}$$