Introduction to Dynamic Systems

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it

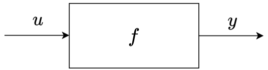
Robotic Systems

Systems

System

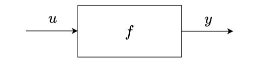
A **system** is a set of elements, that can be considered as a whole, that interact with each other and with the environment according to a **certain law**.

A system can be represented as a **black box** that interacts with the environment through an input (the **stimulus** u), producing a certan **effect** (onto the environment) which is the output (that we call y)



- $u(t)$, the input, that is any physical quantity that varies during time
- f, the **law**, $y(t) = f(u(t))$
- $y(t)$, the output that depends on $u(t)$ and f

The Law of a Systems



- $u(t)$, the input
- f, the **law**, $y(t) = f(u(t))$
- $y(t)$, the output

The law depends on:

- Characteristics of parts of the system
- Composition of the parts
- How the parts interact

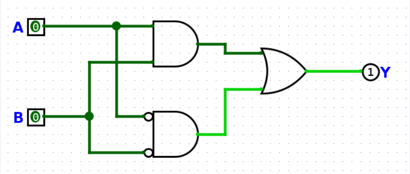
f can be a mathematical function, an algorithm, or any other abstraction that can model/represent the behaviour of the system

Indeed, according to the nature of the law f, the system can be:

- **Static** or **Dynamic**

Example of **Static** Systems

A Combinatorial Circuit with Logic Gates

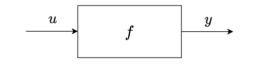


- $u = [a, b]$, the input $\in \{0, 1\}$
- y, the output $\in \{0, 1\}$
- f, the **law**, $y = ab + \overline{a}\overline{b}$

- The output depends totally and only on the instantaneous value of the input
- If u varies during time ($u(t)$), the output behaves accordingly
- Given two time instants t and t', $t \neq t'$, if $u(t) = u(t')$ then $y(t) = y(t')$ (time has no effect)

Example of **Static** Systems

A program computing the roots of quadratic equations

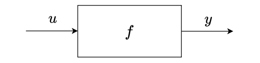


- $u = [a, b, c]$, the input $\in \mathcal{R}$
- $y = [x_1, x_2]$, the output $\in \mathcal{C}$
- f, the **law**, the algoritm to solve quadratic equations

- The output depends totally and only on the instantaneous value of the input
- If u varies during time ($u(t)$), the output behaves accordingly
- Given two time instants t and t', $t \neq t'$, if $u(t) = u(t')$ then $y(t) = y(t')$ (time has no effect)

Example of **Dynamic** Systems

A soccer ball on a playing field

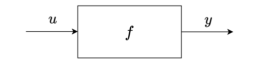


- $u = F(t)$, the force applied by the kick $\in \mathcal{R}$, in Newton
- $y = [v, p]$, the output, speed and position of the ball $\in \mathcal{R}$
- f, the **physic law** that gives the speed and position for each time instant

When you kick the ball...

- You are applying an impulsive force at time instant $t = 0$, but for $t > 0$ the input is null
- The speed v increases suddently at $t = 0$ and then decrease gradually for $t > 0$ (but input is 0), eventually reaching zero
- The position p increases for $t >= 0$, eventually reaching a constant value
- Given two time instants t and t', $t \neq t'$, if $u(t) = u(t')$ then $y(t)$ can be $\neq (t')$ (time **has** effect)

Analytical Representation of a Systems



- $u(t)$, the input
- f, the **law**, $y(t) = f(u(t))$
- $y(t)$, the output

The **law** f:

- In a **static system**, **does not depends** on time but only on $u(t)$
  $y(t) = f(u(t))$
- In a **dynamic system**, **depends** on time and on $u(t)$
  $y(t) = f(u(t), t)$
- A dynamic system is (analytically) expressed with (a system of) **differential equations**

Example of a Dynamic System in Godot

(godot/ball)

The Concept of “Discretisation”

- We consider **real-life** systems, so they evolve during **time**
- **Time** is a quantity belonging to $\mathcal{R}$ and evolves in a **continuous** way
- But this concept cannot be modeled or implemented in a computer system
- We subdivide the time in **Time Quanta**, i.e. time intervals that are very very small
- and we consider the **events** that occur **only** each time quantum
- In other words, we perform a **sampling** of the real (or simulated) world using a specific **sampling time Δt** that can be **constant** or **variable**
- This operation is called **discretisation**
- Δt is chosen so that between $t = i\Delta t$ and $t' = (i+1)\Delta t$ “**almost nothing**” happens
- When we consider **mechanical systems**, Δt can be in the order of **milliseconds**

Tools and Software

The Plotter

lib/data/dataplot.py

class **DataPlotter**

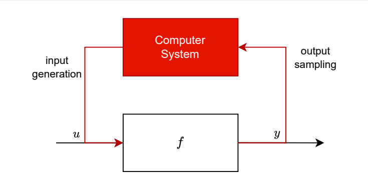
- **DataPlotter()**, constructor
- **set_x(descr:string)**
  sets the description of the X axis
- **add_y(var:string,descr:string)**
  adds a variable with description to the Y axis
- **append_x(value:float)**
  appends a new value to the X
- **append_y(var:string,value:float)**
  appends a new value to the specified variable of Y axis
- **plot()**
  plots the graph

Example of a DataPlotter Usage

(examples/dataplot/dataplot_example.ipynb)
(examples/dataplot/dataplot_example_2.ipynb)

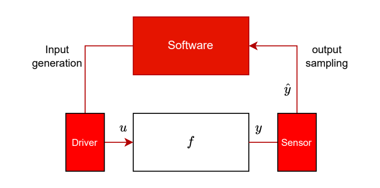
Interacting with a System

Driving a (Dynamic) System



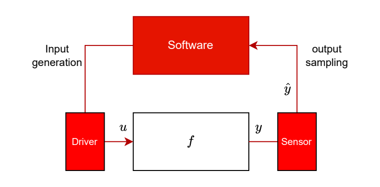
- In robotic systems, the physical system is connected to an **electronic system**, that tries to make the system behave as desired
- This electronic system is usually a **computer system** with a **software** that continuously (or periodically) **senses the output** and **generates the proper input** signal

Driving a (Dynamic) System



- Output sensing is performed by proper **electronic sensors** that “sense” the physical quantities needed and transform them into proper **software variables**
- Input driving is performed by proper **electronic drivers** that are able to transform **software variables** into physical quantities
- The **software** is implemented by means of an infinite loop that gets data from sensors, processes them and sends processed data to the driver

Driving a (Dynamic) System

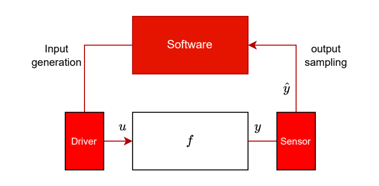


Timer-based sampling

while *True* **do**
- On each ΔT;
- $data \leftarrow read\_sensors()$;
- $proc\_data \leftarrow process(data, \Delta T)$;
- $send\_to\_driver(proc\_data)$;

end

Driving a (Dynamic) System

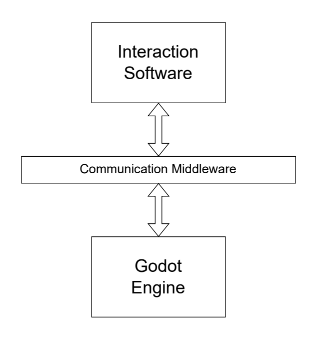


Sensor-based Timing

while *True* **do**
    data ← *wait_sensors*();
    Compute ΔT;
    proc_data ← *process*(*data*, ΔT);
    send_to_driver(*proc_data*);
end

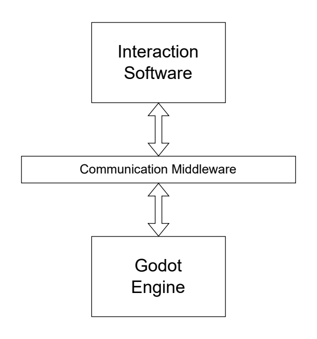
Our Godot-based System Simulation and Interaction Framework

Simulation



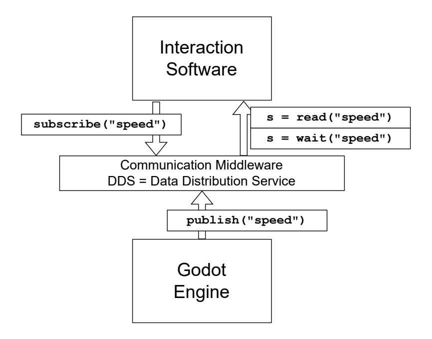
- We will use Godot as a **physical simulation engine**
- All data processing and driving will be made by means of an external python program
- The two worlds interact by means of a **communication middleware** that acts as a data interchange channel

Simulation



- Exchaged data are **variables** characterised by:
  - **Name** (a literal, e.g. “position”, “speed”)
  - **Type** (int or float)
  - **Value**

Interaction Protocol



- Interaction protocol is based on a **publish-subscriber** mechanism:
  - A peer interested to a variable make a **subscription** to its name
  - The peer that produces the variable performs a **publish**
  - The interested peer can wait the publication or directly read (if available) the variable value

The Data Distribution Service

`lib/dds/dds.py`

class **`DDS`**

- **`DDS()`**, constructor
- **`start()`**
  starts the DDS
- **`subscribe(var_list:list of strings)`**
  performs a subscription to the specified variables
- **`publish(name:string, value:float or int, type)`**
  publishes a variable
  type = DDS.DDS_TYPE_INT or DDS.DDS_TYPE_FLOAT
- **`read(name:string)`**
  reads a published variable
- **`wait(name:string)`**
  waits for a publication event and reads the given variable

The Time Helper Class

lib/utils/time.py

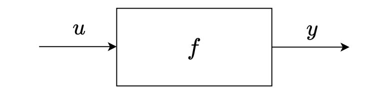
class **`Time`**

- **`Time()`**, constructor
- **`start()`**
  starts the time helper
- **`get()`** $\rightarrow$ **`float`**
  gets the current time (since object creation)
- **`elapsed()`** $\rightarrow$ **`float`**
  gets the time interval since last “elapsed” call

Example of Godot Interaction

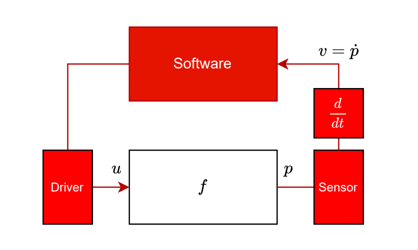
(examples/godot_plot/godot_ball_test.ipynb)
(examples/godot_plot/godot_ball_test_position.ipynb)

Implementation of Basic Systems



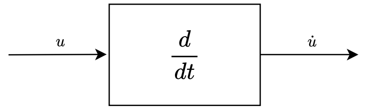

```
class System:

    def __init__(self):
        # initialise members

    def evaluate(self, delta_t : float, _input : any): -> any
        # implement a delta_t computation step using _input
        # and generate _output
        ....
        return _output
```

The Derivator



- Let's us consider that we have a **position sensor** but we need (also) the **speed**
- We must **derivate** the sensed data

The Derivator



The "dotted" notation

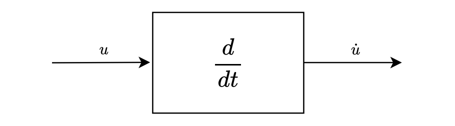
A **dot** over a variable represents the derivative w.r.t time:

$$\dot{p} = \frac{dp}{dt} = v$$

Two **dots** over a variable represent the second derivative w.r.t time:

$$\ddot{p} = \frac{d^2p}{dt^2} = \dot{v} = a$$

The Derivator

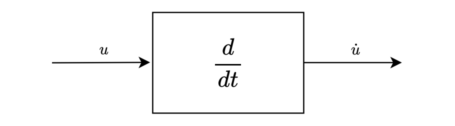


$$\dot{u} = \frac{du(t)}{dt}$$

To implement a derivator we approximate the derivative with the **incremental ratio**:

$$\dot{u} = \frac{du(t)}{dt} \simeq \frac{u(t + \Delta T) - u(t)}{\Delta T} = \frac{u(t) - u(t - \Delta T)}{\Delta T}$$

The Derivator

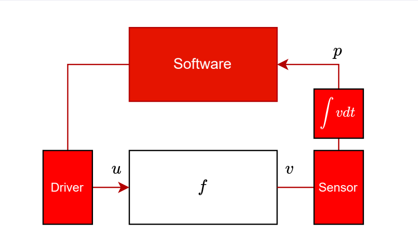



```
class Derivator:

    def __init__(self):
        self.prev_input = 0

    def evaluate(self, delta_t, _input):
        out = (_input - self.prev_input) / delta_t
        self.prev_input = _input
        return out
```

Example of Derivative

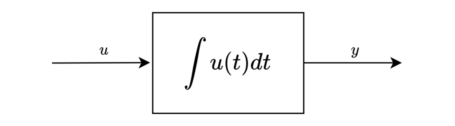
(examples/basic/godot_ball_test_derivative.ipynb)

The Integrator



- Let's us consider that we have a **speed sensor** but we need (also) the **position**
- We must **integrate** the sensed data

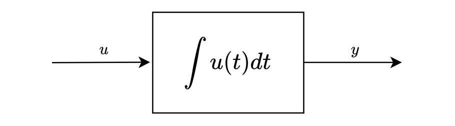
The Integrator



$$y = \int_0^t du(\tau)d\tau$$

To implement an integrator we compute the **inverse function** that is a **derivative**:

$$u(t) = \frac{dy(t)}{dt} \simeq \frac{y(t + \Delta T) - y(t)}{\Delta T}$$

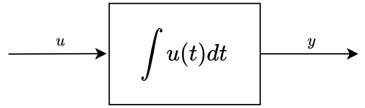


To implement an integrator we compute the **inverse function** that is a **derivative**:

$$u(t) = \frac{y(t + \Delta T) - y(t)}{\Delta T}$$

$$y(t + \Delta T) = y(t) + u(t)\Delta T$$

The Integrator

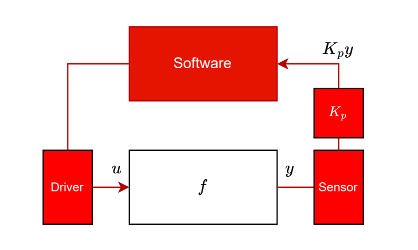



```
class Integrator:

    def __init__(self):
        self.prev_output = 0

    def evaluate(self, delta_t, _input):
        out = self.prev_output + _input * delta_t
        self.prev_output = out
        return out
```

Example of Integral

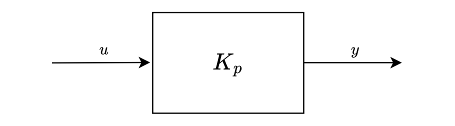
(examples/basic/godot_ball_test_integral.ipynb)

The Gain (Proportional System)



- Let's us consider that we have a **sensor** that gives data in a measure unit **different than** what we need
- We must apply a **proportional factor** to the sensed data

The Gain (Proportional System)



$$y(t) = K_p u(t)$$

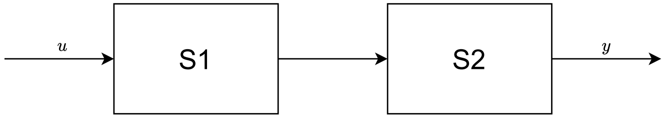
```
class Proportional:

    def __init__(self, _kp):
        self.kp = _kp

    def evaluate(self, delta_t, _input):
        return _input * self.kp
```

Any dynamic system (linear, time-invariant) can be represented as a **linear combination** of the basic systems:

- **Proportional**
- **Integral**
- **Derivative**

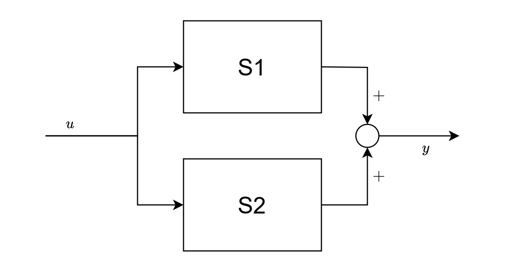
Composition of Systems

Series Composition of Systems



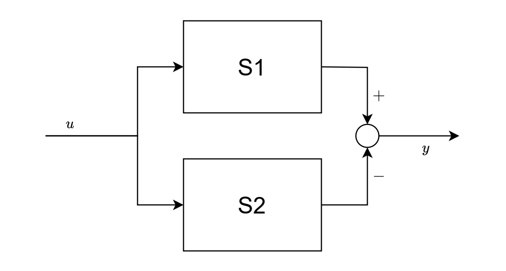

```
class Series:

    def __init__(self):
        self.s1 = System(...)
        self.s2 = System(...)

    def evaluate(self, delta_t, _input):
        out_s1 = self.s1.evaluate(delta_t, _input)
        out_s2 = self.s2.evaluate(delta_t, out_s1)
        return out_s2
```

Parallel Composition of Systems



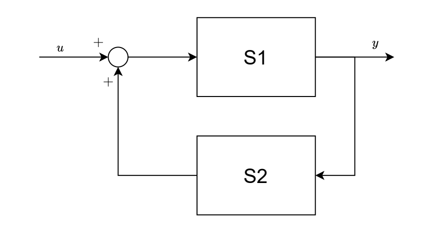

```
class Parallel:

    def __init__(self):
        self.s1 = System(...)
        self.s2 = System(...)

    def evaluate(self, delta_t, _input):
        out_s1 = self.s1.evaluate(delta_t, _input)
        out_s2 = self.s2.evaluate(delta_t, _input)
        out = out_s1 + out_s2
        return out
```

Parallel Composition of Systems



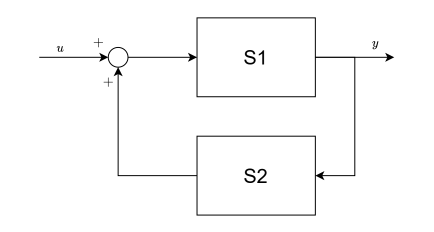

```
class Parallel:

    def __init__(self):
        self.s1 = System(...)
        self.s2 = System(...)

    def evaluate(self, delta_t, _input):
        out_s1 = self.s1.evaluate(delta_t, _input)
        out_s2 = self.s2.evaluate(delta_t, _input)
        out = out_s1 - out_s2
        return out
```

Feedback



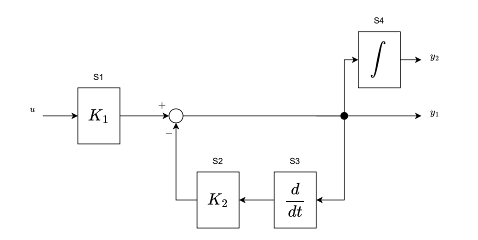
- The presence of a **Feedback** implies the concept of **memory**
- We must consider the **previous value** of the variable in feedback
- Indeed the input to $S1$ is the current value of u plus the output of $S2$ given the **previous value of y** as input
- Therefore we must save the previous value of y

Feedback



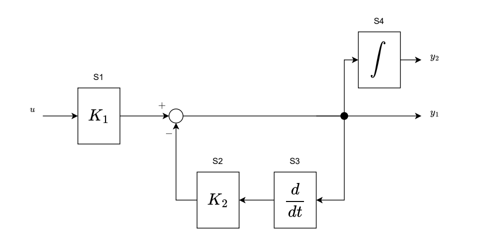

```
class Feedback:

    def __init__(self):
        self.s1 = System(...)
        self.s2 = System(...)
        self.prev_out = 0

    def evaluate(self, delta_t, _input):
        out_s2 = self.s2.evaluate(delta_t, self.prev_out)
        input_s1 = out_s2 + _input
        out = self.s1.evaluate(delta_t, input_s1)
        self.prev_out = out
        return out
```

A Compound System




```
class Compound:

    def __init__(self):
        self.s1 = Proportional(K1)
        self.s2 = Proportional(K2)
        self.s3 = Derivator()
        self.s4 = Integrator()
        self.y1 = 0

...
```

A Compound System




```
...

    def evaluate(self, delta_t, _input):
        out_s1 = self.s1.evaluate(delta_t, _input)

        out_s3 = self.s3.evaluate(delta_t, self.y1)
        out_s2 = self.s2.evaluate(delta_t, out_s3)

        y1 = out_s1 - out_s2
        y2 = self.s4.evaluate(delta_t, y1)

        self.y1 = y1
        return (y1, y2)

...
```

Introduction to Dynamic Systems

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it

Robotic Systems