# Modèle et génération automatique de code

Alexandre Chapoutot

ENSTA Paris

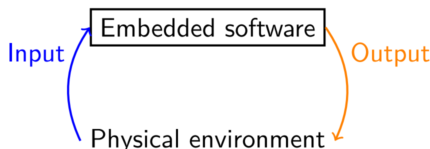
2022-2023

# Part I

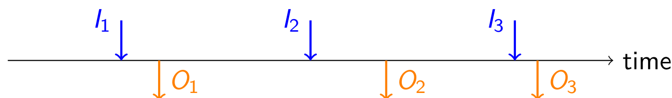
## Lecture 6

# Reactive software

- **Embedded software** are also known as **reactive programs**: they continuously produce outputs in response to inputs coming from the physical environment.

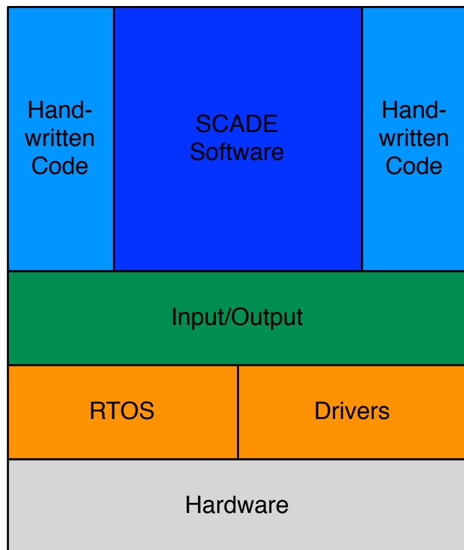



- The execution of embedded software is described by **discrete-time dynamics** *i.e.* it is a sequence of reactions.




- Ideally we should have that:
  - Output $O_i$ should be emitted before input $I_{i+1}$ and no important input $I_i$ is missed.
  - The software is deterministic: same input produces same output.
  - A finite amount of memory is used.

# Model-based: kind of software target




SCADE function is based on

- data-flow equations
- state machines

Note: it is the same for Simulink/Stateflow

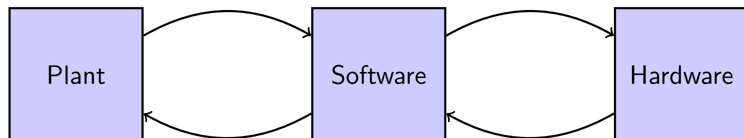
# Simulink

1 Simulink

2 Stateflow

# Anatomy of an embedded system

Embedded systems are made of different components which highly interact together.




The jobs of the designer and the programmer are:

- Design an algorithm to control a physical process
- Implement this algorithm on a given hardware

**Remark:** we deal with two different worlds

- continuous-time evolution for the plant.
- discrete-time (periodic sampling) evolution for the software.

# Main steps of modeling

Using a model-based design method to build a system, we usually follow these steps:

- Definition of the system (interface);
- Break down the system into components;
- Modeling the **behaviors** of the components with **data-flow equations** or **state-diagrams**;
- Write these models into a software tool like Simulink/Stateflow or SCADE or etc.;
- Simulate the model;
- Validate the simulation results.

## Goal of this part

Presentation of Simulink/Stateflow tool for the model-based design.

# A short tutorial on Simulink

# Overview of Simulink features

Simulink is a tool for modeling and simulating dynamical systems that are systems evolving with time.

The main features of Simulink are:

- a language to **model continuous-time**, **discrete-time** or a **mix of both kinds** of systems
- a complete set of **numerical algorithms** to simulate those kinds of systems
- a complete set of **data-types**: integer, floating-point, fixed-point arithmetic
- a good debugger

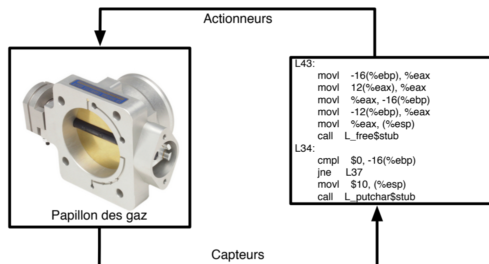
We can also add:

- a complete set of extension libraries to handle various cases: mechanical systems, hydraulic systems, electronics systems, DSP, control design, etc.
- code generators: for embedded software, for hardware synthesis.

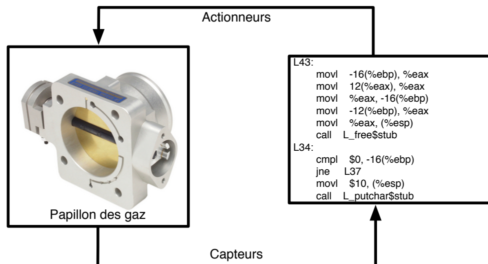
# An electronic throttle example

In order to introduce the different kind of Simulink model, we consider a small example of an electronic throttle system made of:

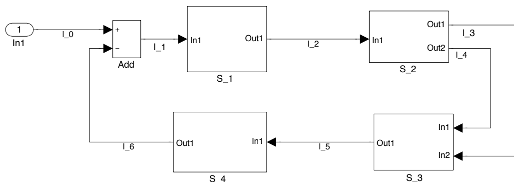
- an acceleration pedal;
- an electronic throttle subsystem;
- a control subsystem
- a sensor: measuring the position of the throttle;
- an actuator: an electric motor.



# An electronic throttle example



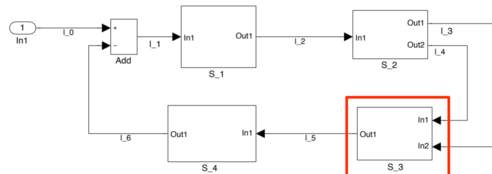

## Simulink model

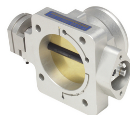


# An electronic throttle example

## Throttle model

Full Simulink model

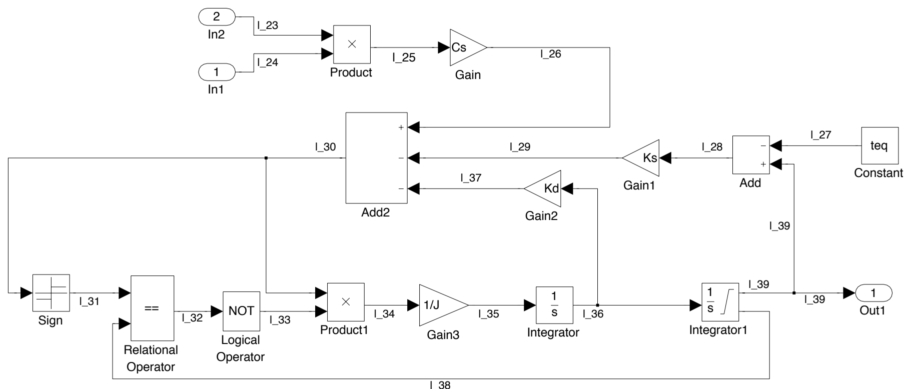


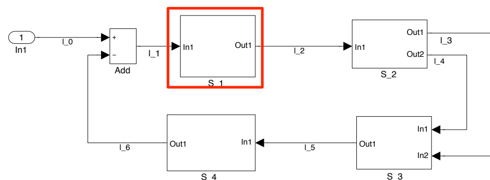
Mathematical model of the throttle

$$
\begin{cases}
T(t) = \text{Direction} \times \text{Duty} \times C_s \\
\dot{\omega}(t) = \frac{1}{J}(-K_s(\theta(t) - \theta_{eq}) - K_d\omega(t) + T(t)) \quad 0 < \theta < \pi/2 \\
\qquad \text{if } (\theta < 0 \wedge sgn(\dot{\omega}(t) = -1)) \vee ((\theta > \pi/2 \wedge sgn(\dot{\omega}(t) = 1))) \\
\qquad \quad \text{then } \dot{\omega}(t) = 0
\end{cases}
$$

Simulink model (continuous-time subsystem)



## Full Simulink model



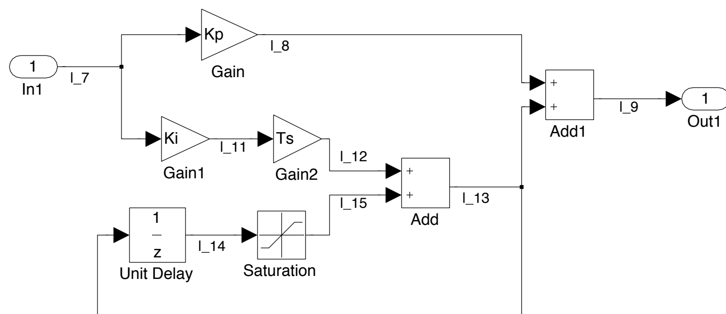

```
L43:
        movl   -16(%ebp), %eax
        movl   12(%eax), %eax
        movl   %eax, -16(%ebp)
        movl   -12(%ebp), %eax
        movl   %eax, (%esp)
        call   L_free$stub
L34:
        cmpl   $0, -16(%ebp)
        jne    L37
        movl   $10, (%esp)
        call   L_putchar$stub
```

Mathematical model (input $e$, output $y$)

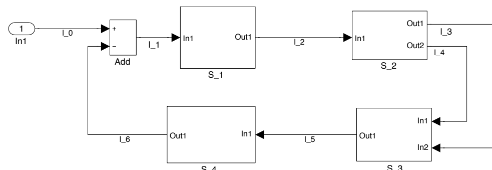
$$\begin{cases} y(k) = y_p(k) + y_i(k) \\ y_p(k) = K_p e(k) \\ y_i(k+1) = y_i(k) + K_i T_s e(k) \quad 0 < y_i(k) < 1 \end{cases}$$

PI controller (Proportional-Integral)

Controller model (discrete-time subsystem)



### Full Simulink model

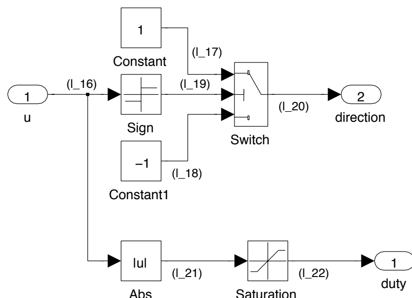



### Mathematical model

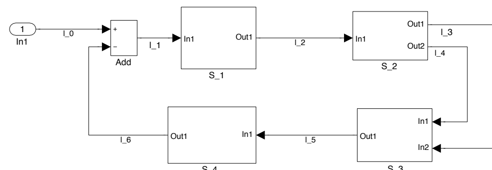
$$Duty = \begin{cases} |u| & \text{if } 0 \leq |u| \leq 1 \\ 1 & \text{otherwise} \end{cases}$$

$$Direction = \begin{cases} 1 & \text{if } u \leq 0 \\ -1 & \text{otherwise} \end{cases}$$

### Actuator model (logical subsystem)



Full Simulink model



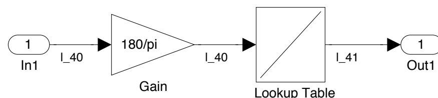

Mathematical model

$$Angle = \frac{180}{\pi} \times u$$

$$y = \begin{cases} 0.5 & \text{if } angle = 0^\circ \\ 4.5 & \text{if } angle = 90^\circ \end{cases}$$

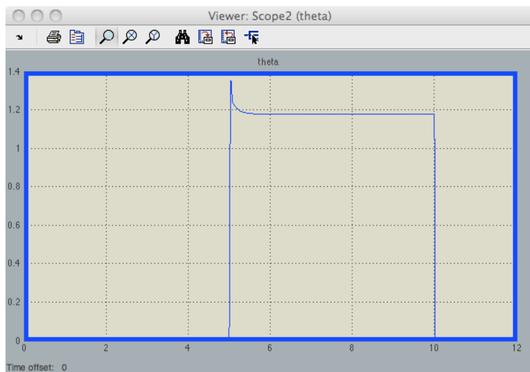
$y(t)$ is defined as a linear interpolation between these two points

Sensor model (a potentiometer and a converter from rad into degree)



# An electronic throttle example

## Simulation result




- The input is a step function and a look-up table ($[0\ 1] \mapsto [3.5\ 0]$).
- The solver is `ode45`.
- The simulation stop time is 11.

**Remark**

The validation of the model is based on this kind of results.

# Simulink as a language

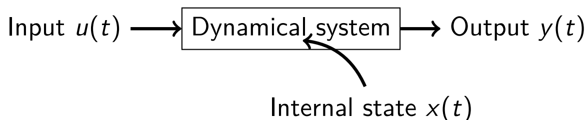
Simulink is a graphical language representing block diagrams that is a Simulink models is made of blocks and wires.

A piece of vocabulary:

- **Wires:** represent the values (signals) exchange between blocks, they evolve with time during the simulation.
- **Blocks:** represent the operations applying on signals.
- **States:** a piece of information needed by a block to compute its output.
- **Parameters:** values of the system which are constant during the simulation.

# Simulink block as dynamical systems

Overview of a dynamical system:




A generalized mathematical description of dynamical systems:

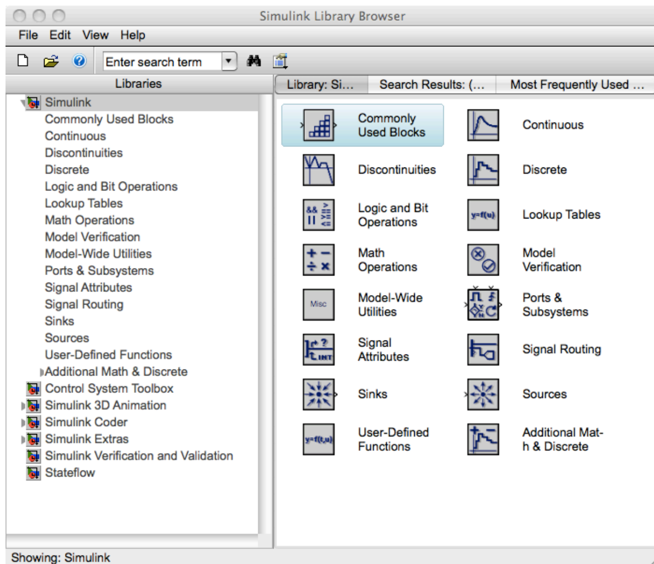
$$\dot{x}(t) = f_c(x, u, t) \qquad \text{Continuous-time behaviors}$$
$$x_{k+1}(t) = f_d(x, u, t) \qquad \text{Discrete-time behaviors}$$
$$y(t) = g(x, u, t) \qquad \text{Output function}$$

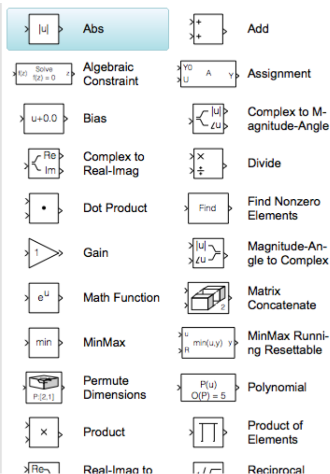
**Remark**

Each block of Simulink is associated to a dynamical system.

# Simulink library

## The standard library





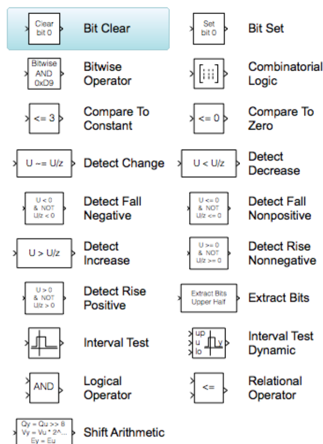

For example, `Add` block is associated with a degenerated (no state) dynamical systems:

$$y(t) = u_1(t) + u_2(t)$$

**Remark:** this operator can be n-ary

# Simulink library

## The library for bit and logical operations

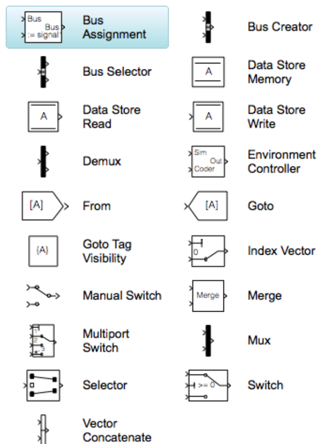



For example, `Compare To Zero` block is associated with a degenerated (no state) dynamical systems:

$$y(t) = u_1(t) <= 0$$

**Remark:** truth value follows the C language convention *i.e.* 0 is false and 1 is true (but we can enforce the type to be Boolean).

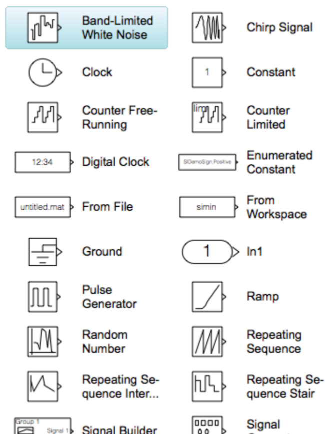
The Signal Routing library




For example, `Switch` block is associated with a degenerated (no state) dynamical systems:

$$y(t) = \begin{cases} u_1(t) & \text{if } u_2(t) \bowtie 0 \\ u_3(t) & \text{otherwise} \end{cases}$$

where $\bowtie \in \{>, \geq, \neq\}$.

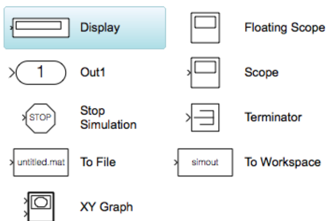



This library gathers blocks which generates input values.

For example, the `Clock` block is associated to the simulation time.

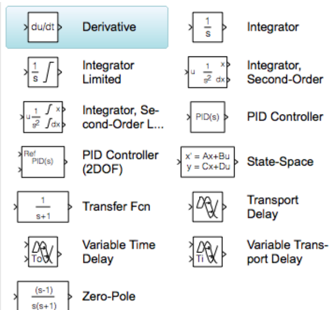
For example, `Constant` block is associated with a degenerated (no state) dynamical systems:

$$y(t) = \text{cst}$$




This library gathers blocks used to visualize or save output values.

For example, `Scope` block display the temporal evolution of signals.



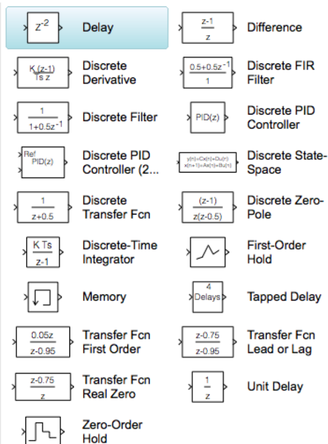

For example, the `Integrator` block is associated to the dynamical systems:

$$\dot{x}(t) = u(t) \quad \text{with} \quad x(0) = x_0$$
$$y(t) = x(t)$$

**Remark:** we need numerical integration scheme to solve such kind of equations.

# Simulink library

## The discrete library




For example, the `Unit Delay` block is associated to the dynamical systems:

$$x_{k+1}(t) = u(t) \quad \text{with} \quad x(0) = x_0$$
$$y(t) = x_k(t)$$

**Remark:** we need numerical integration scheme to solve such kind of equations.

# A step-by-step model writing

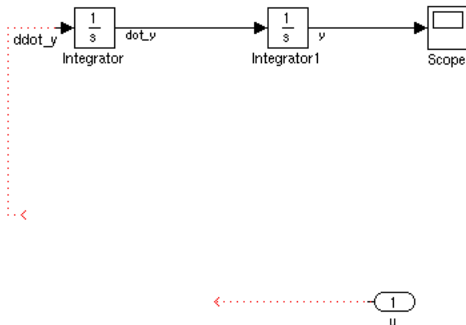
We consider as a simple example a continuous-time system associated to a second order linear systems given by:

$$\ddot{y}(t) + a_1\dot{y}(t) + a_2y(t) = bu(t) \Leftrightarrow \ddot{y}(t) = bu(t) - a_1\dot{y}(t) - a_2y(t)$$

# A step-by-step model writing

We consider as a simple example a continuous-time system associated to a second order linear systems given by:

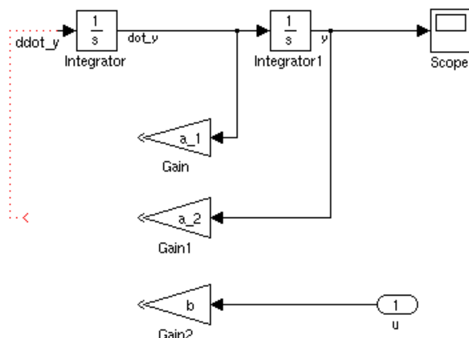
$$\ddot{y}(t) + a_1\dot{y}(t) + a_2 y(t) = bu(t) \Leftrightarrow \ddot{y}(t) = bu(t) - a_1\dot{y}(t) - a_2 y(t)$$



# A step-by-step model writing

We consider as a simple example a continuous-time system associated to a second order linear systems given by:

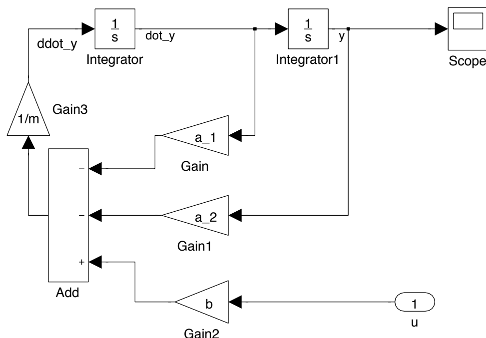
$$\ddot{y}(t) + a_1\dot{y}(t) + a_2 y(t) = bu(t) \Leftrightarrow \ddot{y}(t) = bu(t) - a_1\dot{y}(t) - a_2 y(t)$$



# A step-by-step model writing

We consider as a simple example a continuous-time system associated to a second order linear systems given by:

$$\ddot{y}(t) + a_1\dot{y}(t) + a_2 y(t) = bu(t) \Leftrightarrow \ddot{y}(t) = bu(t) - a_1\dot{y}(t) - a_2 y(t)$$



# A step-by-step model writing

After building the Simulink model we have to set the values of the parameters: $a_1$, $a_2$ and $b$.

We use a Matlab file to do it, as Simulink reads parameter values from the Matlab workspace.

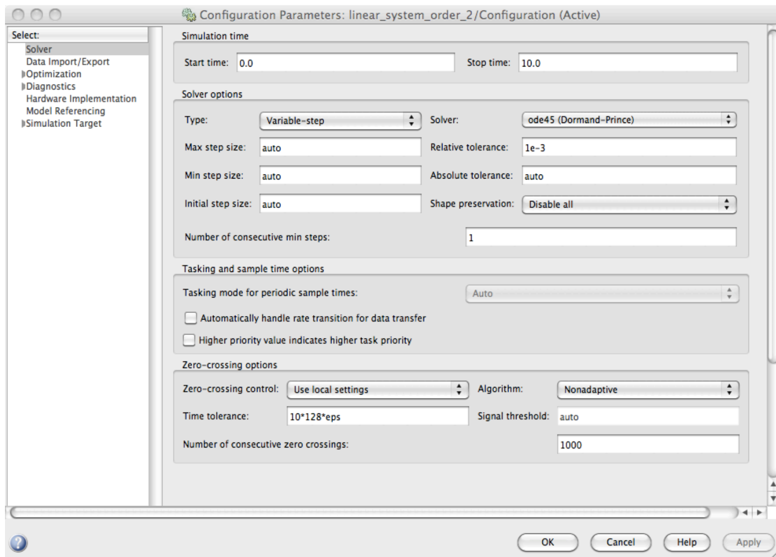
For example,

```
a_1 = 1;
a_2 = 2;
b = 3;
```

Next, we set the simulation parameters.

# A step-by-step model writing

## Choosing the numerical scheme



# A step-by-step model writing

## Defining the input

Configuration Parameters: linear_system_order_2/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target

**Load from workspace**

- [ ] Input: [t, u]
- [ ] Initial state: xInitial

**Save to workspace**

Time, State, Output

- [x] Time: tout — Format: Array
- [ ] States: xout — [x] Limit data points to last: 1000
- [x] Output: yout — Decimation: 1
- [ ] Final states: xFinal — [ ] Save complete SimState in final state

Signals

- [ ] Signal logging: logsout — Signal logging format: ModelDataLogs
- Configure Signals to Log...

Data Store Memory

- [x] Data stores: dsmout

**Save options**

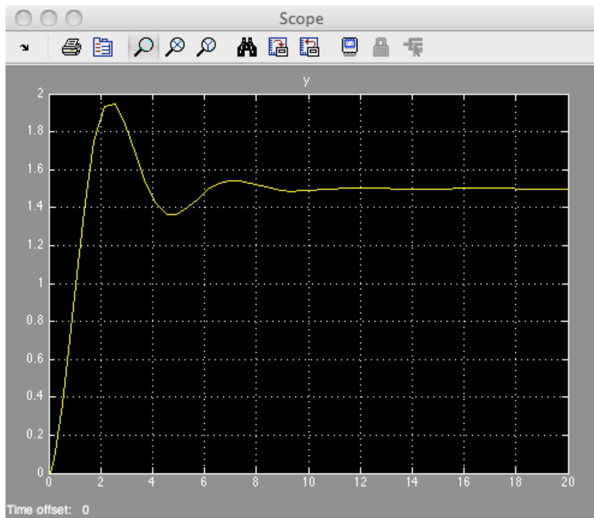
- Output options: Refine output — Refine factor: 1
- [ ] Save simulation output as single object: out
- [ ] Record and inspect simulation output

OK | Cancel | Help | Apply

## Automatizing the simulation

We can pilot the simulation process from a Matlab script.
For example,

```
% model parameter
a_1 = 1;
a_2 = 2;
b = 3;
% set the input
t = (0:0.01:20)';
u = ones(length(t),1);
% name of the Simulink model
mdl_name = 'linear_system_order_2'
% set the simulation Stop Time and the Solver
set_param(mdl_name, 'Solver', 'ode23', 'StopTime', '20');
% run the simulation
set_param(mdl_name,'simulationcommand','start')
```


Scope
y
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
12
14
16
18
20
Time offset: 0

# Simulink model and vector values

We consider as a simple example a continuous-time system associated to a second order linear systems given by:

$$\ddot{y}(t) + a_1\dot{y}(t) + a_2 y(t) = bu(t)$$

Note that we can rewrite this formula into state-space form:

$$\begin{pmatrix}\dot{x}_1 \\ \dot{x}_2\end{pmatrix} = \begin{pmatrix}0 & 1 \\ -a_2 & -a_1\end{pmatrix}\begin{pmatrix}x_1 \\ x_2\end{pmatrix} + \begin{pmatrix}0 \\ b\end{pmatrix} u$$

$$y(t) = \begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}x_1 \\ x_2\end{pmatrix}$$

That is in the form:

$$\dot{x} = Ax + Bu$$

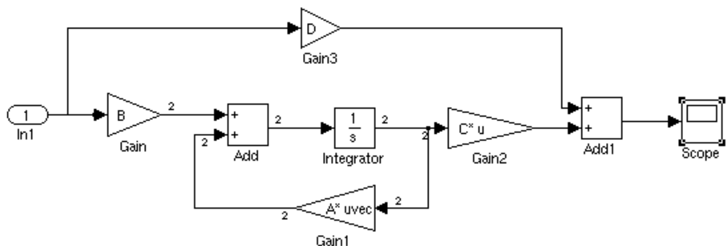
$$y = Cx + Du$$

where $A$, $B$, $C$, $D$ are matrices and $x$ is a vector.

# Simulink model and vector values

We have a generic pattern in Simulink to represent system in the form:

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

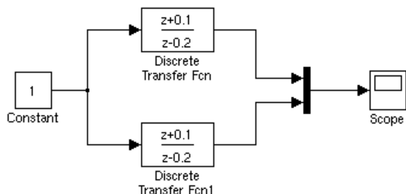



## Remark

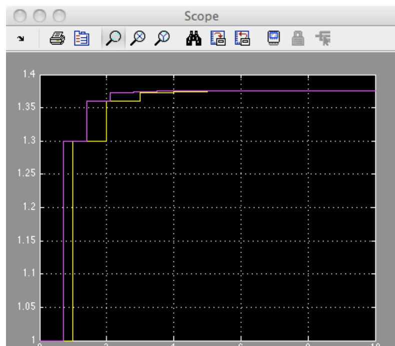
Simulink can handle vector value signals.
Signal dimension: `Format->Port/Signal Displays->Signal Dimension`.

# Multi-rate Simulink models

Each block of a Simulink model, except continuous ones, is associated to a sampling rate, *i.e.* when updating the state or the output of the block.

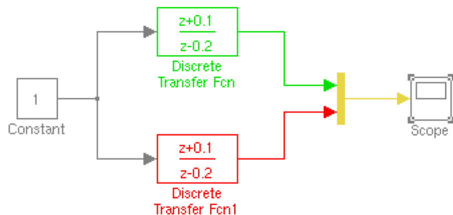



- Sampling rate (Transfer Fcn): 1s
- Sampling rate (Transfer Fcn1): 0.7



# Multi-rate Simulink models

Each block of a Simulink model, except continuous ones, is associated to a sampling rate, *i.e.* when updating the state or the output of the block.




Legend:

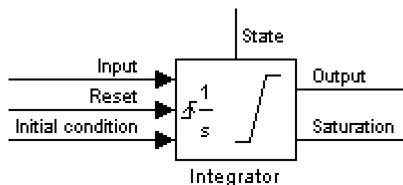
| Color | Description | Value |
|---|---|---|
| (gray) | Fixed in Minor Step | [0,1] |
| (red) | Discrete 1 | 0.7 |
| (green) | Discrete 2 | 1 |
| (yellow) | Hybrid | Not Applicable |

- Display color of sampling rate:
  `Format->Sample Time Display->Colors`

**Remark**

We must use variable step-size solver in this case!

# A comment on the Integrator block




Input, Output : same as the basic case.

Reset : an event to reset the state value to initial value.

Initial condition : external port to set the initial value.

Saturation : indicate if the output is saturated or not.

State : "The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset."

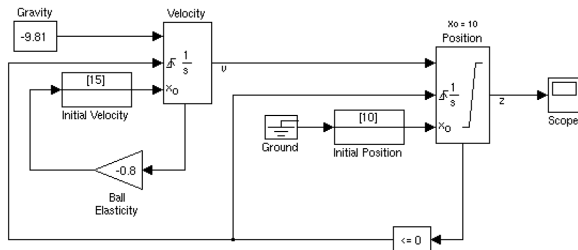
# An example of Integrator with reset – 1

The bouncing ball is defined by:

$$\dot{x}_1 = x_2;$$
$$\dot{x}_2 = -g$$

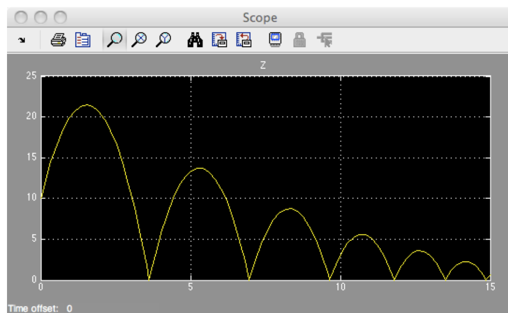
with if $x_1 \leq 0$ then $x_2 := -0.8x_2$.

Simulink model:



# An example of Integrator with reset – 2

The simulation output of the bouncing ball is:




## Remark

The detection of the time of the bounces requires special algorithms: zero-crossing detection.

# Summary

- Simulink is a complex language with a lot of features.
- It is easy to model and simulate complex hybrid systems.

**Question**

Is Simulink suitable for embedded software as is?

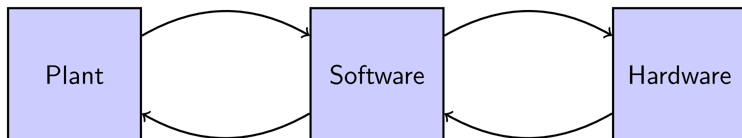
# Stateflow

1. Simulink

2. Stateflow

# Anatomy of an embedded system

Embedded systems are made of different components which highly interact together.




The jobs of the designer and the programmer are:

- Design an algorithm to control a physical process
- Implement this algorithm on a given hardware

**Remark:** we deal with two different worlds

- continuous-time evolution for the plant.
- discrete-time (periodic sampling) evolution for the software.

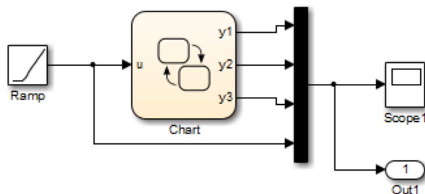
# A short tutorial on Stateflow

# Overview of Stateflow features

Stateflow is an extension of Simulink which is used to model and simulate:

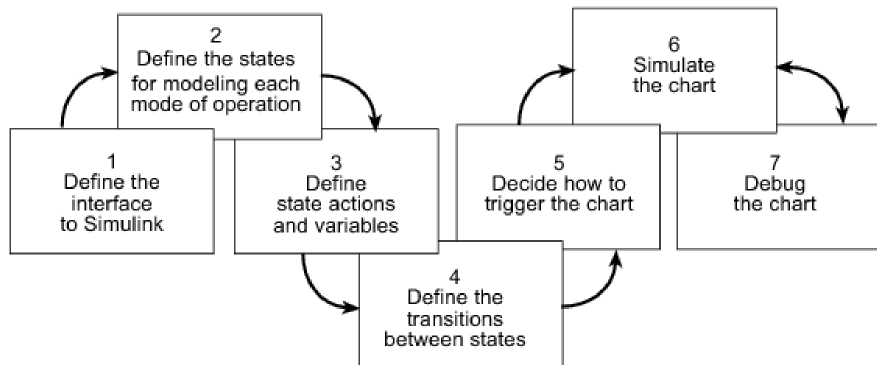
- Finite state machines;
- Flow charts;
- Truth tables.

Stateflow receives input from Simulink models and emits output to Simulink models.



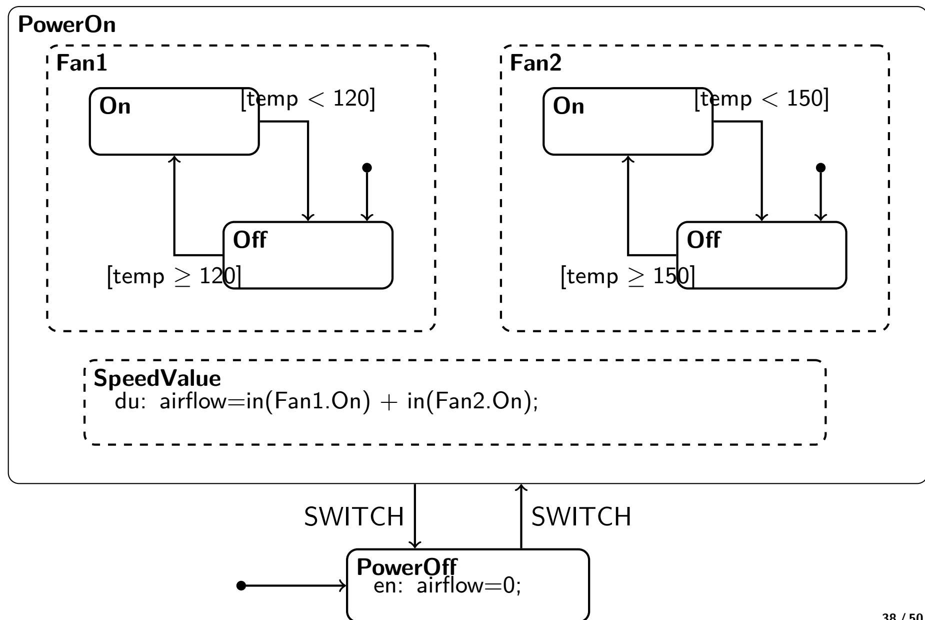

**Note:** Stateflow block are activated by on sampling period or on occurence of events

# Design methodology[1]




[1] According to The Mathworks

# A simple FSM in Stateflow



# FSM in Stateflow

Finite State Machines in Stateflow are a combination of Mealy and Moore charts with variables. In consequence, their semantics is more complex.

## Two kinds of states

- `OR` states, *i.e.* states executed in exclusion
- `AND` states, *i.e.* states executed in parallel (but the simulation has a sequential execution)

Moreover states can contain other states, we have hence **hierarchy**.

# State actions

Computations are associated to states (state action) at different time:

- `entry` (en): action executed once when the state is activated.
- `during` (du): action executed each time step of simulation when the state is activated and no active transition is available.
- `exit` (ex): action executed once when the state is. deactivated

## Example

```
du: airflow = in(Fan1.On) + in(Fan2.On);
```

`in` is a predicate which returns 0 if false and 1 if true.

# Action language

## Binary operations

| Example | Prec | C-Like Bit Ops Enabled |
|---|---|---|
| a * b | 10 | Multiplication of two operands |
| a / b | 10 | Division of one operand by the other |
| a %% b | 10 | Modulus |
| a + b | 9 | Addition of two operands |
| a - b | 9 | Subtraction of one operand from the other |
| a >> b | 8 | Shift operand a right by b bits.<br>(See * note below.) |
| a << b | 8 | Shift operand a left by b bits.<br>(See * note below.) |
| a > b | 7 | Comparison of the first operand greater than the second operand |
| a < b | 7 | Comparison of the first operand less than the second operand |

# Action language

## Binary operations

| Example | Prec | C-Like Bit Ops Enabled |
|---|---|---|
| `a >= b` | 7 | Comparison of the first operand greater than or equal to the second operand |
| `a <= b` | 7 | Comparison of the first operand less than or equal to the second operand |
| `a == b` | 6 | Comparison of equality of two operands |
| `a ~= b` | 6 | Comparison of inequality of two operands |
| `a != b` | 6 | Comparison of inequality of two operands |
| `a <> b` | 6 | Comparison of inequality of two operands |
| `a & b` | 5 | Bitwise AND of two operands |
| `a ^ b` | 4 | Bitwise XOR of two operands |
| `a \| b` | 3 | Bitwise OR of two operands |
| `a && b` | 2 | Logical AND of two operands |
| `a \|\| b` | 1 | Logical OR of two operands |

| Example | Description |
|---|---|
| `~a` | Logical not of a<br>Complement of a (if `bitops` is enabled) |
| `!a` | Logical not of a |
| `-a` | Negative of a |

| Example | Description |
|---|---|
| `a++` | Increment a |
| `a--` | Decrement a |

| | | | | | |
|---|---|---|---|---|---|
| abs | acos | asin | atan | atan2 | ceil |
| cos | cosh | exp | fabs | floor | fmod |
| labs | ldexp | log | log10 | pow | rand |
| sin | sinh | sqrt | tan | tanh | |

# Action language

## Assignments

| Example | Description |
|---|---|
| `a = expression` | Simple assignment |
| `a += expression` | Equivalent to `a = a + expression` |
| `a -= expression` | Equivalent to `a = a - expression` |
| `a *= expression` | Equivalent to `a = a * expression` |
| `a /= expression` | Equivalent to `a = a / expression` |

### An more

- call homemade C functions
- manipulate arrays
- use pointers
- temporal operators: `after(n,E)`, `before(n,E)`, `at(n,E)` and `every(n, E)`.

# Temporal logic

Temporal logic operator can be found in state actions or on transition conditions.

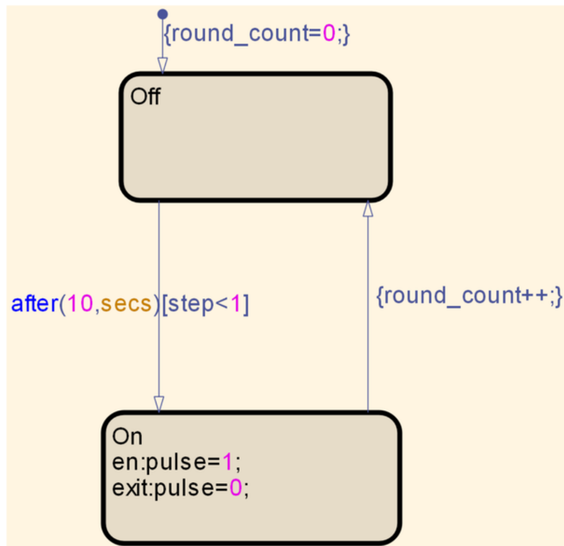
| Operator | Syntax | Description |
|---|---|---|
| `after` | `after(n, E)`<br>where `E` is the base event for the `after` operator and `n` is one of the following:<br>• A positive integer<br>• An expression that evaluates to a positive integer value | Returns true if the base event `E` has occurred at least `n` times since activation of the associated state. Otherwise, the operator returns false.<br>In a chart with no input events, `after(n, tick)` or `after(n, wakeup)` returns true if the chart has woken up `n` times or more since activation of the associated state. |

Events can be:

- user defined events
- existing events: `sec`, `msec`, `tick`

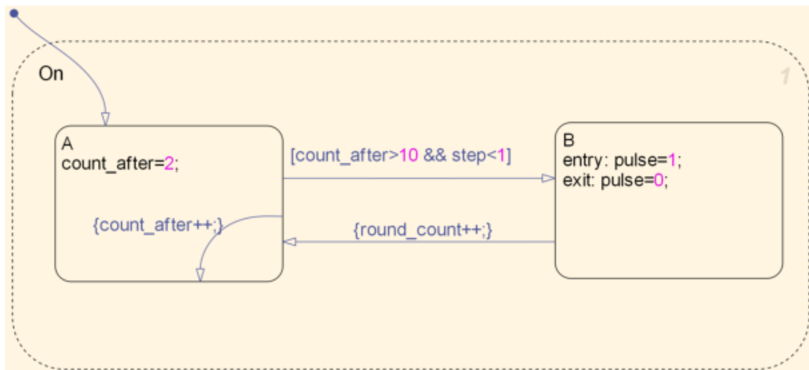
# Temporal logic: example

With temporal logic



# Temporal logic: example

Without temporal logic




We need to create a new variable and a new transition to count the time elapsed.

# Transition between states – 1

The more general form of a Stateflow transition is:

```
EVENT [ condition ] {condition_action} / transition_action
```

- `EVENT`: the activation of the transition is enabled if the event is present.
- `condition`: a Boolean expression which enables the transition if this expression is true.
- `condition_action`: piece of code executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid.
- `transition_action`: piece of code executed after the transition destination has been determined.

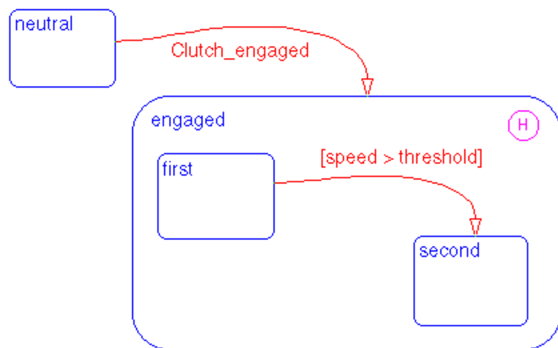
# Transition between states – 2

The validation/activation of the transition depends of its kind:

- Event only: valid is that event occurs.
- Event and condition: valid if that event occurs and the condition is true.
- Condition only: valid if any event occurs and the condition is true.
- Action only: valid if any event occurs.
- Not specified: valid if any event occurs.

**Default transition** is a special transition gives among several states which one is the first executed.

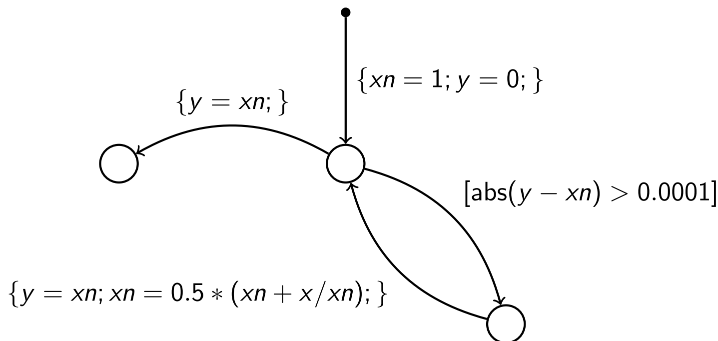
# History junction




- History junction is used to set the activate sub-state as the most recently visited.
- It overrides the default transition.

# A simple flow chart in Stateflow

A flow chart to compute the square root using Newton method.




- Connective junctions are used to define decision point in a chart.
- Simplification of the chart as a connective junction is not a state.

# Simulation algorithm (old)

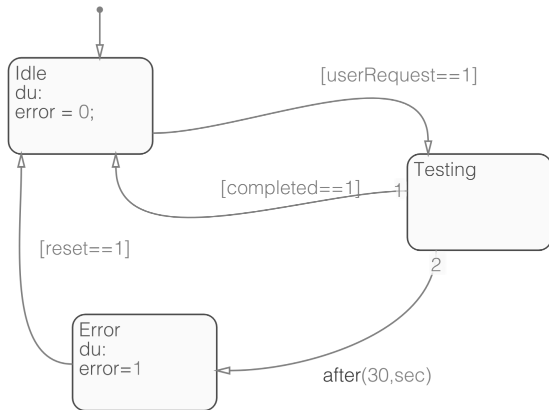
Mainly, the execution of Stateflow model follows four steps at each time an input event occurs:

1. search for active states
   hierarchically from top to bottom and sequentially for parallel states `clock rules`;
2. search for valid transition (twelve o'clock rule);
3. execution of valid transition;
   execute exit action, set source state inactive, execute transition action, set destination state active, execute entry action.
4. during action executed.

**Remark**

Order of execution and priorities can be added on states and transitions.

# Example: built step-by-step




Main steps

- create inputs/outputs
- create states
- create transitions

# Summary

- Stateflow is a complex language with a lot of features.
- It is easy to model discrete-time systems.

## Question

Is Stateflow suitable for embedded software as is?

## Choosing between Simulink and Stateflow

- If the function mainly involves complicated logic operations, Stateflow should be used.
- If the function mainly involves numerical operations, Simulink should be used.