

Hybrid Systems
Lab class - Stateflow
 C. Lesire & F. Defay, October 12th, 2010

Goal of this lab class: Model and simulate a hybrid system using Simulink and Stateflow.

Contents

1	A <i>Hybrid</i> control of a platform by a satellite's reaction wheel	1
1.1	Introduction	1
1.2	System Description	2
1.3	Specification	2
2	Modeling the hybrid control	4
2.1	PowerOn/PowerOff transition	5
2.2	Control law	5
2.3	Position error	6
2.4	Speed sensor failure	6
2.5	Remarks on Stateflow execution	6
3	Simulating your Stateflow	6
4	Executing your Stateflow on the target	7
A	Getting started	9

1 A *Hybrid* control of a platform by a satellite's reaction wheel

1.1 Introduction

Let us consider the attitude control of a satellite with respect to a defined frame of reference, here its gravity center. From a reference attitude and sensor measurements of the current attitude, a control algorithm commands the actuators that apply the torques needed to put the satellite to the desired attitude. In a satellite with 3 axes, the attitude control is realized using:

- a star tracker for the attitude measurement (the gravity center of the satellite);
- 3 gyro-meters for the angular speed measurement and the estimated attitude (angular speed $> 0.1^\circ$);
- 3 reaction wheels to apply torques to the platform;
- 3 magneto torquers (used to reduce wheels speed).

The following tasks have already been realized:

- the design of the different control laws to control the platform, to simulate them and to complete the C-code provided;
- the design of the embedded software considering the temporal constraints and the different tasks (to manage request, to manage law and setpoint, to acquire curves information, to manage duration and periods).

1.2 System Description

The components of this platform are:

- a moving platform around one axe;
- a reaction wheel driven by a DC motor;
- the following sensors:
 - 1 platform angular speed sensor;
 - 1 motor speed sensor (tachometer);
 - 1 absolute angular position sensor of the platform;
 - 1 sensor measuring the motor current;
- a computer embedded in the satellite for executing the command laws.

1.3 Specification

The goal is to design a (hybrid) attitude control considering only the control laws; the communication between the satellite and the ground station *is not taken into account*. The attitude control must consider different modes according with some conditions: speed sensor failure, eclipse occurrence, error value, etc. We consider in this project a platform representing only one satellite's axe.

Simulink and Stateflow will be used to model and to simulate the control algorithm, to generate the code that will be embedded in the satellite and also to execute this embedded code in the target using xPC Target¹.

As the communication with the ground station is not taken into account at this phase of the project, it is necessary to simulate in the Matlab environment the followings signals (see also section 2 and the interface in fig. 1(a) that will be provided in the skeleton of your attitude controller): PowerOn/PowerOff, Speed Sensor OK/ Speed Sensor KO and the reference (setpoint).

The control algorithm to be implement is the following: in the initial state of the attitude control system, the power is off and so the attitude control u has the initial value 0. At the rising edge (power is On), the control system is activated. The control law to be used is a function of the position and the speed as before, but you must consider the state of the sensor (failure or not) and the degree of magnitude of the position error (big or small):

¹xPC Target provides a high-performance host-target environment that enables to connect Simulink and Stateflow models to physical systems and execute them in real time on low-cost PC-compatible hardware. xPC Target includes proven capabilities for rapid prototyping, hardware-in-the-loop testing, and application deployment in an open hardware architecture (Mathworks).

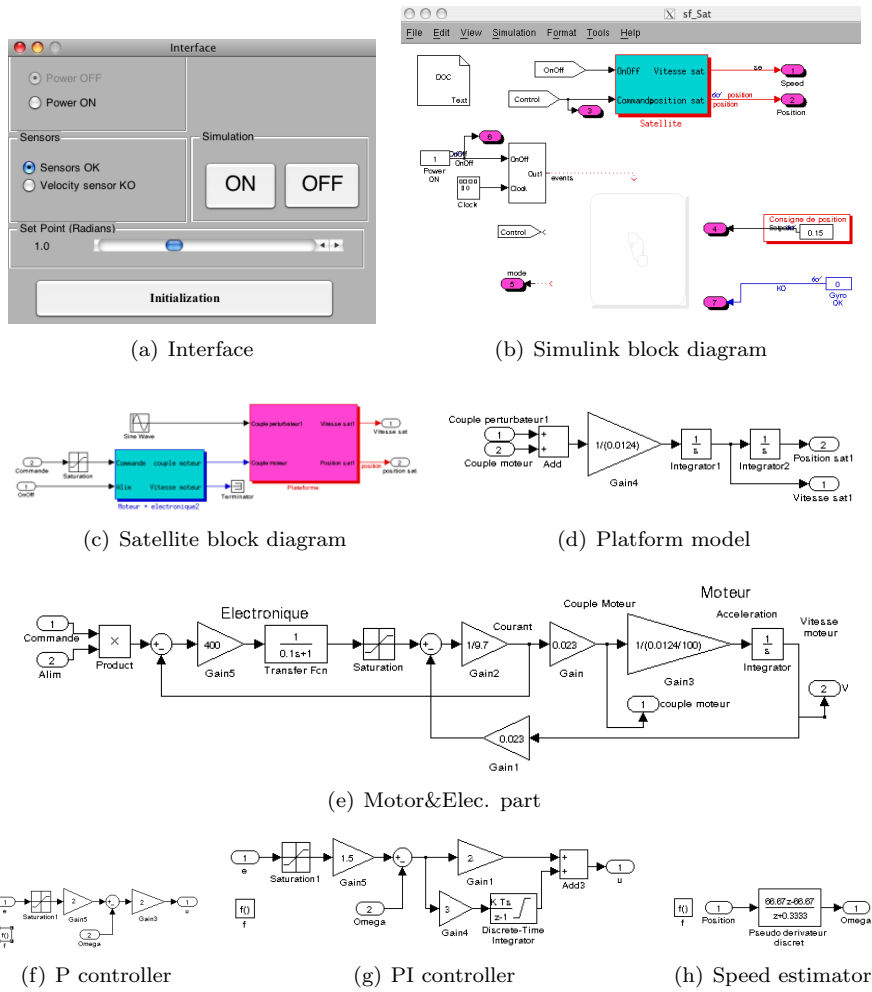


Figure 1: The Simulink skeleton model provided

- if the position error is big (for example, $\text{abs}(\text{error}) > 0.1$), the control law is a proportional controller; otherwise (if the position error is small) a proportional-integral controller must be used. A P controller allows to lead the system quickly to the reference attitude when the error is big; the PI controller insures a good precision and it is used only when the error is small;
- if there is an angular speed sensor failure, instead of using the angular speed value, it is necessary to estimate the speed from the position sensor value using a pseudo derivation.

The following algorithms/models will be provided in the skeleton of your attitude controller:

- the platform model (see fig. 1(d)),
- the motor and electronic model (see fig. 1(e)),
- the angular speed estimator from the position sensor (see fig. 1(h)),
- the P and PI controllers (see fig. 1(f) and (g)) and the pseudo derivative estimator.

If the power is turned Off, the system comes back to an inactive state where the attitude control u is 0 again, independently of the error value and/or the sensor failure.

2 Modeling the hybrid control

You must model the hybrid control system (the different discrete states (or modes) and the different continuous equations), using Simulink/Stateflow (see appendix A for a quick user guide). It is necessary to use the Interface (provided in the skeleton) allowing to simulate in the Matlab environment the followings signals (see fig. 1(a)):

- PowerOn/PowerOff: is a boolean signal whose initial state is PowerOff;
- buttons ON and OFF, for starting and stopping the control simulation or the control execution;
- a setpoint (in radians) for changing the reference attitude;
- an Initialization button that you need to click *before* you start your simulation or execution;
- Speed Sensor OK/ Speed Sensor KO : a boolean signal with SpeedKO=1 if there is a sensor failure and SpeedKO=0 if the sensor is working well.

To help you progress in the controller design, the lab is decomposed in several steps. Please, save your models at each step using different names to get a trace of your progress.

2.1 PowerOn/PowerOff transition

First, consider the PowerOn/PowerOff signal. Write a StateFlow model with two states representing the fact that the control is either On or Off. For the moment, in both states the control must be set to 0.

Creating your Stateflow:

- Download file `BE_Sat.zip` from

`\\panpan\public\distrib\f.defay`

in your folder; there are 2 folders, `sf_Sat` for simulation and `sf_cible` for execution.
- Open Matlab; define the active directory as `sf_Sat`, open `sf_Sat.mdl`: you can see the Simulink blocks of fig. 1;
- Double-click on the main Chart to open the corresponding block;
- Create the two states dragging the first icon on the left top to the Stateflow window;
- On each state, write first the state label;
- Define the inputs/outputs and events of your StateFlow:
 - from the Add menu, select Data/Output from Simulink (select General tab); in the Name field, choose the name of your output (u); in the Type field, select the data type;
 - you can see in the Simulink model (by clicking the icon up-arrow in the Stateflow Editor) that the output u appears in the SF block;
 - connect the output of the SF block to the satellite model;
 - create your transitions by putting the mouse on the border of the source state and dragging it to the target state; click on the transition (it will become red), then click next to the question mark "?" and type the name of the event;
 - define edge-triggered events: in the Stateflow editor, Add Menu, select Event/Input from Simulink; define the **Power** event, with the good trigger type (either "Rising", "Either" or "Falling"); using the up-arrow icon, verify that there is a trigger symbol in the SF;
 - connect the edge-triggered events to the input signals;
- Add the command value to both states ($u = 0$);.

Save the model and simulate it (see sec. 3) to verify its behavior.

2.2 Control law

Modify the 'On' state to have u computed using the P controller.

2.3 Position error

Consider now the position error. Depending on the magnitude of this error, the control law must be either a P or a PI controller (see sec. 1.3).

2.4 Speed sensor failure

Consider now that the speed sensor can fail. In that case, you should use the speed estimator in addition to the controller used depending on the position error value (see sec. 1.3).

2.5 Remarks on StateFlow execution

The stateflow is awoken at each event occurrence (rising or falling edge) or at some sampling frequency, given by the clock block of simulink. The continuous functions executed in a state can need a particular sampling frequency. For example, the sensor must be sampled ($T_s = 0.01s$) and the control laws must be calculated at the same frequency. There are two ways to do this: choosing a fixed step solver for Simulink or put a self-transition in this state with an event clock as a label.

3 Simulating your StateFlow

Run `interface.m` to launch the simulation control interface.

Two scopes to analyze your input and output signals are already in the provided skeleton. To connect a signal, right click on its line and choose **Connect to Existent Viewer**.

Suggestion: put in the scopes the attitude control, PowerOn/PowerOff, SensorKO, position, setpoint. Put in a same scope signals that have roughly the same magnitude.

At each simulation:

1. click on the Initialization button in the Interface window (fig. 1(a));
2. enter a new set of input signals allowing you to really test several configurations for validate your model by simulation; Remember that the simulation is not exhaustive, and a good *set* of input values are a key for the model validation; try to start testing the most important cases;
3. the simulation will stop after 20s with the provided skeleton; you can change it if you want;
4. if you click on the scope after the simulation is finished, sometimes the scale is to big due to the maximal values; you can type the command line `trace_sim` on the Matlab window (command window) to better see the simulating results.

4 Executing your Stateflow on the target

The Stateflow will be compiled and run on the target (embedded in the satellite). At this step, the satellite is no more simulated (as in fig. 1(b) to fig. 1(d)) but will receive the control generated by the stateflow and provide the inputs for it.

When using Matlab for simulation, the model run directly on the PC under development. When using xPC Target, the Simulink model is compiled and loaded in the XPC target in real time. The PC is connected to the process via Ethernet. To validate a control law, the simulated physical model (in our case, the platform, the motor and electronic models, see upper part of fig. 1(b)) is replaced by the physical system itself.

Instructions for execute the code in the target

Remark: you cannot follow the algorithm execution on the Stateflow. For having the values of the input and output signals, you must connect them in a scope, and after the end of execution you type the command line **trace** on the Matlab window (command window).

1. Set the current directory **sf_cible** on MAtlab;
2. Run the file **interface_cible.m**;
3. Open the skeleton with the real platform, called **sf_cible/sf_cible.mdl**; the input are the same as in the simulated model, but there is a additional input **target number**; put the number of **cibleitr**, $i=\{2, 4, 6\}$;
4. Copy your Stateflow model in the skeleton and connect the inputs and outputs;
5. Compile your model, Ctrl-B (build) on the Simulink window;
6. Click on the Initialization button in the Interface window (it puts PowerOn=0, SpeedKO=0).

You are now ready to execute your control algorithm:

1. Configure the targetnumber depending on wich target you choose on the room with the matlab command (**lance_cible(1..6);**);
2. Click on the button ON on the Interface window to start the algorithm execution; the Interface will be used in the same way that in the simulation;
3. Click on the button near to the green LED of the target, the white plastic case (the one like a generator function) in order to switch on the power supply;
4. Use the Interface window to simulate the different test conditions: turn On the power, introduce a failure in the angular speed sensor, change the setpoint, turn Off the power, etc;
5. Click on the OFF button in the Interface window for stopping the execution on the target;

6. Stop the target execution with the Matlab command `(-tg);`
7. Use the matlab command `(trace.m)` to show the results;
8. Unload the code from the target with the matlab command `(tg.unload;);`
9. Click on the button near to the red LED of the white case to switch off the power supply.

Remark about wheel saturation:

If the red LED of the white case is *On*, it indicates that the motor is not supplied; you must click on the left button (near the red LED). The green LED will be *On*.

A general remark about Matlab

Remember that Matlab simulates a continuous system in a digital computer, using two types of solver (fixed step and variable step) based in methods as Bogacki-Shampine and Dormand-prince (respectively). You must choose one of the different solvers that can be used to simulate the differential equations according to the system characteristics. Go to Simulation/Configuration parameters/Solver, and choose a solver and a Relative tolerance. The dynamic behavior of the system can be completely incoherent if a good solver is not used, and you must try someone. Read again the help Simulink/Running Simulations/Choosing a Solver, in particular "Improving Simulation Accuracy".

Appendix

A Getting started

Remark: only few features of Stateflow will be used. Please open the Help of Stateflow: menu Help/Product Help; on the left, click on Stateflow (blue icon, alphabetical order), then on Getting started. Or open the pdf file (last item, at the page bottom, Printable (PDF) Documentation on the Web) and save it in your folder.

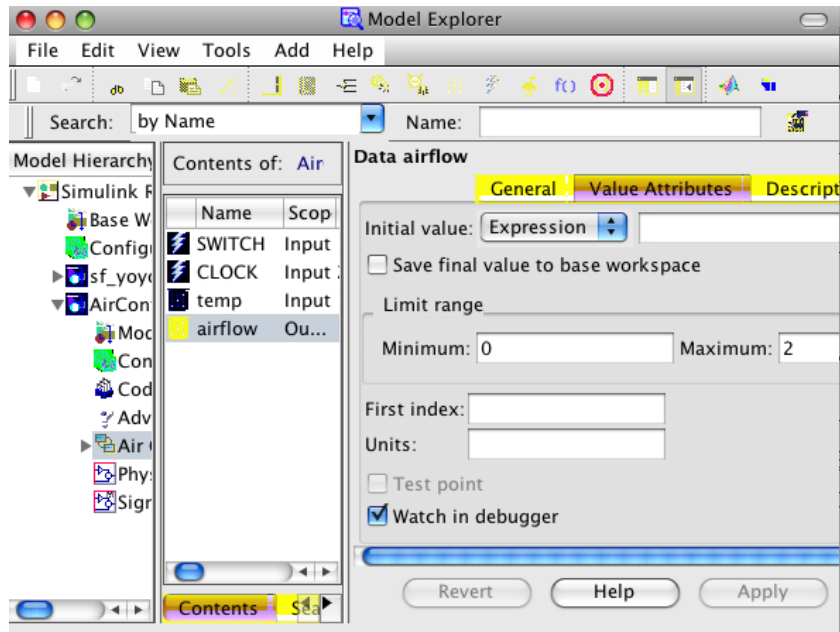



Figure 2: Model explorer

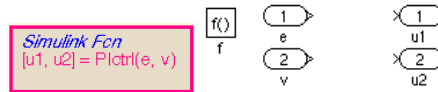
Simulink Function

A Simulink function is a graphical object that you fill with Simulink blocks and call in the actions of states and transitions. It behaves like a function-call subsystem block of a Simulink model. If the function is in a state, it can be called in that state and all its substates; if it is in the chart, it can be called anywhere in the chart. When the state is entered (exited), the function is enabled (disabled).

These are the steps to define a Simulink Function in a Stateflow:

- Add a Simulink function by dragging the icon  from the Stateflow Editor toolbar on the left;
- Enter the function signature (function name `simfcn`, formal argument names `a_i` and return values `r_i`):
$$[r_1, r_2, \dots, r_n] = \text{simfcn}(a_1, a_2, \dots, a_n)$$

The result will be this:



To rename the function click the function box in the Stateflow Editor;

- Define the elements of the Simulink function (don't delete the trigger port `f(c)`)
- Configure the Input and Output ports of the Simulink function (double-click on the port).

In this example, to see the values of `PID_TRESH`, open **Model Explorer** (see fig. 2), click on `sf_slswitch.exemple` in **Model Hierarchy**. Now click on **Base Workspace** on **Model Hierarchy** to change the value. The way the stateflow is updated: **Model Explorer**, click on `SwitchingController`, update method = `Inherited`.

Remark: for using the same step of the solver: `file/Chart Properties/Updated method = Inherited`

Update=`Discrete` (sample time: `f(clock)`); keep the clock when a code must be generated (it is like a hardware interruption).

For continuous part (satellite) use the solver `ode45`.