How the program works

The program I created is a simple Graphical User Interface, where the user can input a *goal speed* as text (e.g. 33.333 m/s ≅ 120 km/h), and several simulators below will each use a customized cruise controller to reach it. Each simulator displays the current speed, a graph as history of the speeds over time.

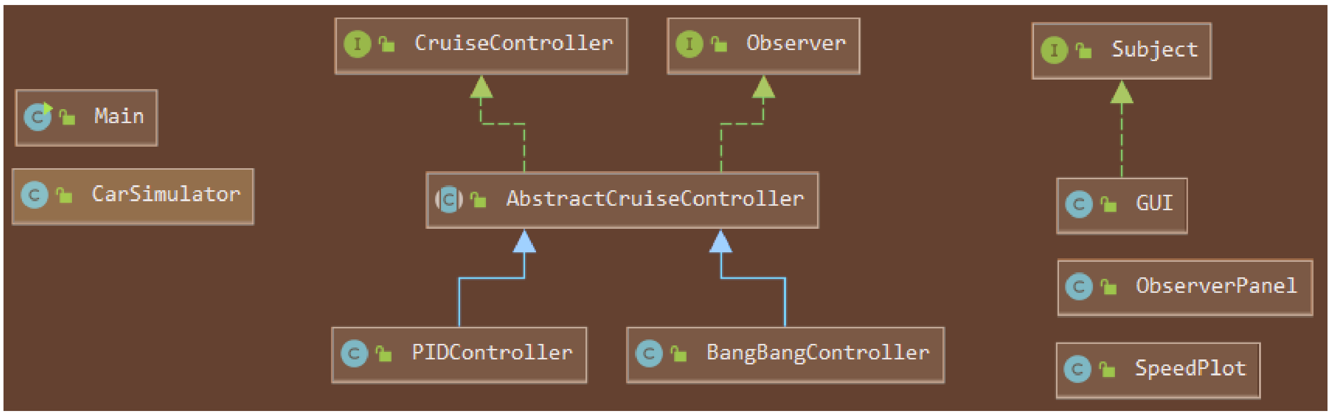
Implemented classes

The program starts from the *Main* class wich creates the *GUI* and simulators for each controller. Each controller is displayed in an *ObserverPanel* which uses the *SpeedPlot*. I put the two required *BangBangController* and *PIDController* classes, but it would be possible to add as much controllers as wanted, if they all extend the *AbstractCruiseController* class.

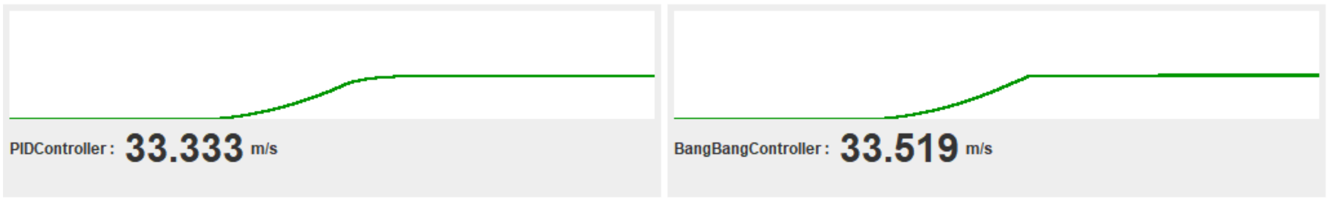
This *AbstractCruiseController* class runs a new thread per controller at 100 Hertz, to detect and react to events in the simulator. It declares variables for an "acceleration window" (similarly to the Sliding Window protocol) and declares a Factory method *computeAcceleration()* that must be implemented by each Concrete controller.

To be able to accelerate the car smoothly, the "acceleration window" was implemented in *computeAcceleration()* for the two subclasses. The window progressively grows when a big acceleration is required, leading smoothly to a maximum acceleration (or deceleration if negative). When no big acceleration is required, the window shrinks back to the minimal acceleration. Furthermore, the physics disadvantages related to speed (such as the wind and the slope) are compensated.

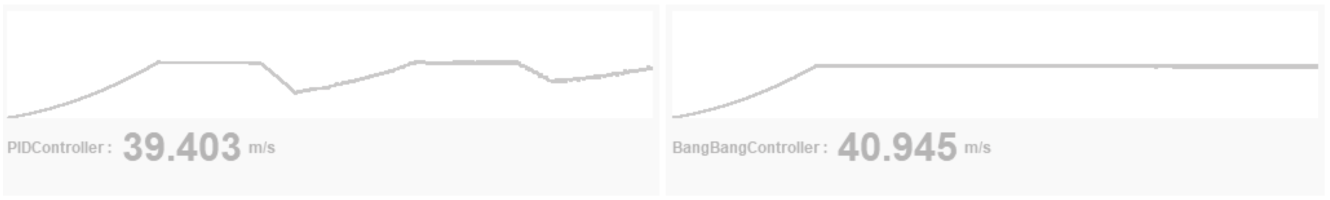
The abstract controller implements two interfaces : *CruiseController*, to make each algorithm accessible in a Strategy pattern through any program, and *Observer,* because each Controller must update its *goal speed* by using an Observer pattern.



Classes in the improved Cruise Control



The improved Cruise Control execution



The old Cruise Control execution, with added graphs

Bang-Bang Controller

The Bang-Bang controller has a strict acceleration policy. In *computeAcceleration*, it computes the error of acceleration and compensates it immediately. The speed is however limited by the acceleration window. The physics disturbance are compensated, regardless of a needed acceleration.

PID Controller

The PID controller is based on the 3 values defined at the top of the class.

When the Controller computes the acceleration in *computeAcceleration*, it first calculates the error, and then the acceleration error. Exceptionally for this controller, I removed the division by the Δt constant to get the acceleration error, because the car would receive a huge acceleration even when the speed was reached, and a weighting of the new acceleration was required. The weighting by 1 proved to be much more satisfactory in the new version of the cruise control.

Next, the error is stored in a LinkedList of instances of the inner *ErrorsAcceleration* class. Hence we can perform operation about them. This LinkedList, the *errorsAcceleration* field, holds the errors for a maximal duration of 0.1 "*optimal seconds".* Note that this duration reduces the list to a decade of errors instead of hundreds (like in the old cruise control), making it an affordable amount of computing for an embedded system.

Then, the PID computation takes place and calculates the *reaction* with a proportional part, a sum of all the reactions, and the derivative of the last acceleration. The average of the first cruise control was removed, as it added some overshoot in a high-frequency "waving-effect".

Next, the acceleration Window grows or shrink, according to the *reaction*, setting so a maximum throttling or braking. The physics effect are taken into account, as well when returning the needed acceleration (instead of just setting a huge steady-state error correction). This allowed me to create a very precise PID.

Finally, if the *reaction* is bigger than the maximum or smaller than the minimum, it is replaced.

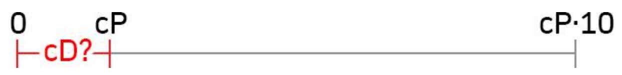
Research of the PID constants

Beforehand, notice that the total reaction is divided by the sum of the 3 constants, making a weighted reaction from the errors, as we did in the robotics course.

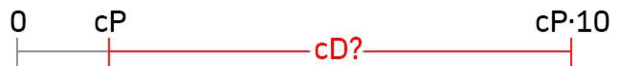
First, the proportional *cP* value was set at 100. The *cI* integral and *cD* derivative were zero.

Secondly the *cD* value was choosed relatively to *cP*. The method to find the value were two binary searches :

1. from *0* to *cP*



2. from *cP* to *cP * 10*



Thirdly, I've set the *cI* value with the same method.

In the improved cruise controller, the result was much better, as I corrected a bad implementation of the *derivative* method. Once the *goal speed* is reached, the new PID rarely exceeds a 0.005 error.

Results for the PID controller

cP (proportional) : 100 *cI* (integral) : 19 *cD* (derivative) : 34

Comparison

The Bang-Bang Controller was able to reach the desired speed fast and with a good enough precision but tends to be a little rough and jumpy when reaching the desired speed. The acceleration abruptly stops, and it would need additional programming measures to counter that effect. This problem is solved with the PID controller.

PID Controller	Bang-Bang Controller
- acceleration window and physics managed - smooth acceleration / braking - smooth goal reaching - steady sustaining - suits e.g. to a vehicle tempomat	- acceleration window and physics managed - smooth acceleration / braking - fast goal reaching - jumpy, rough sustaining - suits e.g. to electrical heating

Conclusion

In conclusion, we can say that the created PID controller works perfectly, as well as the Bang-Bang. A possible improvement would be to detach completely the *CruiseController* implementation from the GUI (passed as parameter), making it a pure Strategy model to implement in embedded systems.

What I learned

I learned that when a variable of the real world must be analysed, it's better to correct only one element using it at a time (e.g the derivative constant), thus it's possible to know the amplitude of its effect on the program.

Furthermore, by applying the design patterns from the Software Engineering course, I noticed that it's not always easy to apply a pattern to random classes, as double extension is not allowed in java, so I used interfaces or aggregation instead, and eventually a merging of concrete methods in abstract classes with concrete components. Hence, we can deduce that patterns should be planned soon and used with parsimony.