# TTK 4147 miniproject

The program was in practice split into 3 threads at first, a controller thread, a receiving thread and an signal-thread, responsible for answering to the signals from the server. The receiving thread would either write the value to a global variable, which would later be read by the controller thread, or signal the signal-thread through the use of a semaphore. The signal-thread was there since the sendto()- function does not give any guarantees for being non-blocking (it is OS-specific). After reading the global variable, the controller thread would calculate the new system input, and send the value back to the server directly.

Due to simplicity and performance reasons, the signal thread was later omitted, resulting in a two-thread design. In the two thread design the answering thread was omitted and its task taken over by the receiving thread. The receiving thread will upon receiving a signal from the server either respond to the signal directly, or write the value to a global variable which will before in turn be read by the controller thread. This causes the controller to rad the same value twice if no new value has arrived in the meantime, for instance in the event of a packet loss or due to how the two threads are scheduled.  An alternative solution would have been to let the controller wait until a new value is written, through the use of a semaphore. This would lead to a more complex implementation, and was not implemented as the cost of additional communication between threads was expected to outweigh the benefit of more proper interpolation.

The controller in use ended up being a PID controller, with limited derivative input ( ud = Kd * ( tau*s/( 1+ tau*s) ). As in the original design, the controller thread sends the updated system input back to the server. In order to have a derivative effect on the controller, which does not overreact to changes in the reference, or to disturbances, we chose a limited D-part (Low-passed derivative). The parameters for the controller were found through experimenting until a satisfactory response was found, and the systems ability to quickly reach the reference value was given priority over other criteria like overshoot and limiting high frequent system input.

One of the largest problems when manually implementing a regulator like in this case is the fact that that the frequency which packets arrive at is varying. There are two possible remedies for this:

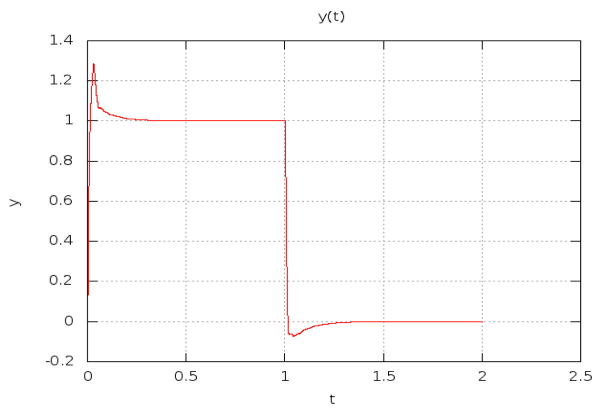
1) Measure the time whenever something is done
2) Use a preiodic controller which reads the last value.

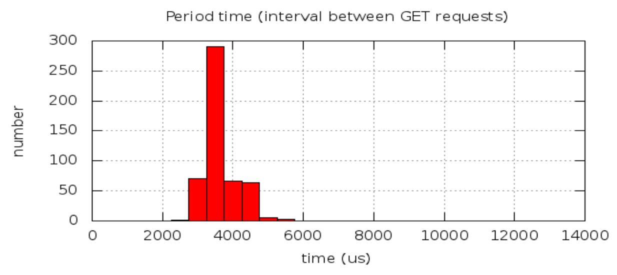
The first approach was chosen, and implemented through the use of clock_gettime(). For the implementation that was chosen, there is not a clear sense of priority between reading controller thread and receiver/answering thread. If we would have implemented this, we would have needed to give the threads priorities, or use xenomai, and its thread-priorities. Alternatively, since the linux-version is >=2.6, the kernel is preemptive[1], and the problem could have been solved with a kernel module.

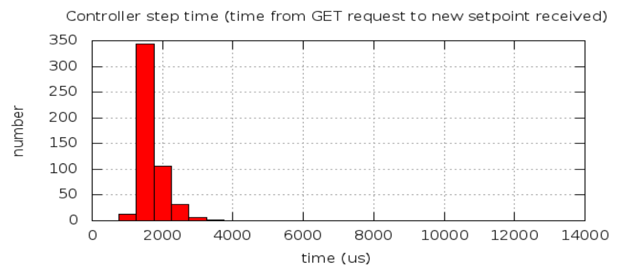
The time-delay in the system makes it a bit more difficult to implement more advanced controllers, since analysing $e^{tau*s}$ is kind of hard, beyond saying if it is stable, by using nyquist-plots, additionally there is then no way to say if you still are able to achieve the desired poles set from an ideal linear system. As the PID was considered to provide a satisfactory system behavior based on the criteria given in the project, this controller was chosen.

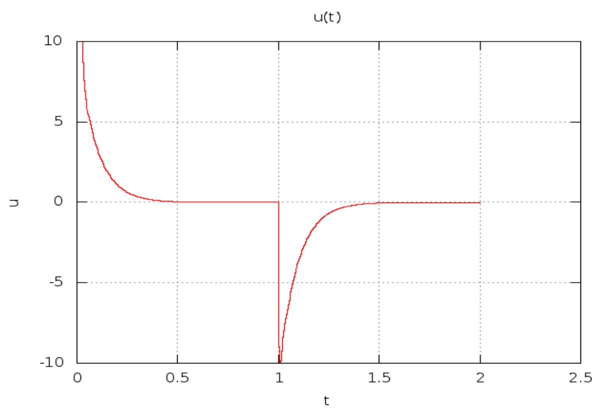
1 http://www.informit.com/articles/article.aspx?p=101760&seqNum=3 [Read: 10.11.18 ]

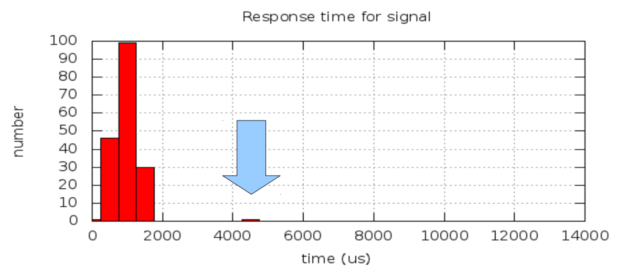
The variance from the response time when awnsering signals, most likely comes from the time the regulator is called to when it has completed. The visible spread that seemingly follows a normal distribution most likely comes from the varaible time it takes when waking up the receive-thread. When running the program multiple times, we were able to provoke the (un)desired response:





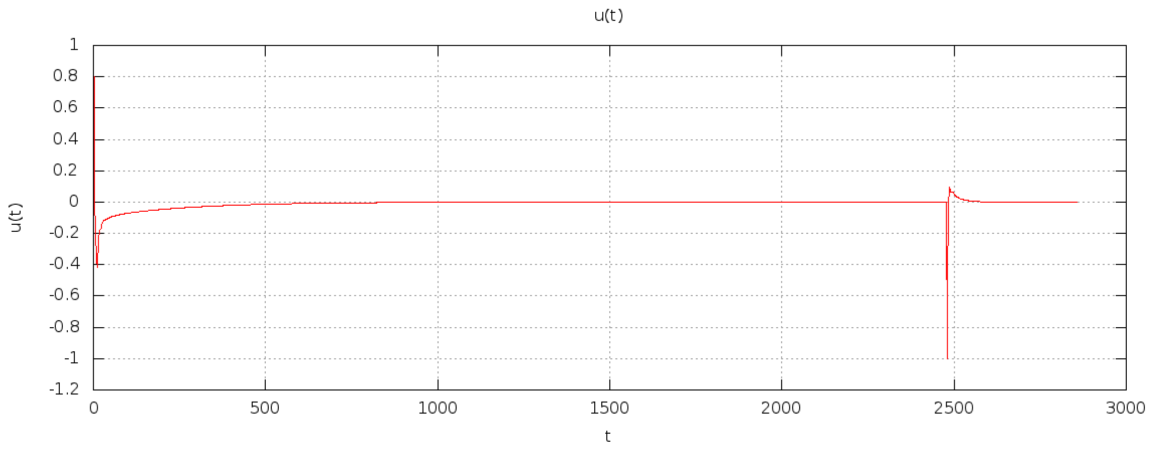




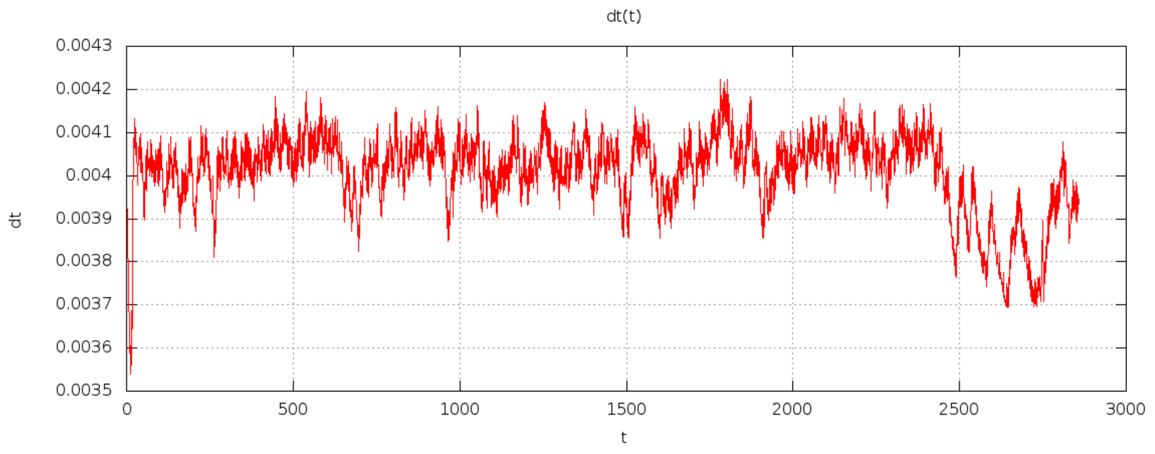


Which means that there is a reasonable possibility for the system to take 4 milliseconds before it responds to a signal.  This results in the problem that even though the system has a rather good average response-time, there is a non-negligeable percentile of responses that are above 4000, and we will thus have to work with a response time of 4000, when talkiing about this system in the real-time sense. The easiest solution in our case is to avoid syscalls, since enforcing priorities is kind of tricky on linux. After running the system for at 10* normal time, 3 times, the result was a max response time of 3184 us, the only one above 2500. This is quite quite a bit better, but the response time still has to be regarded as at least roughly 3200 us.

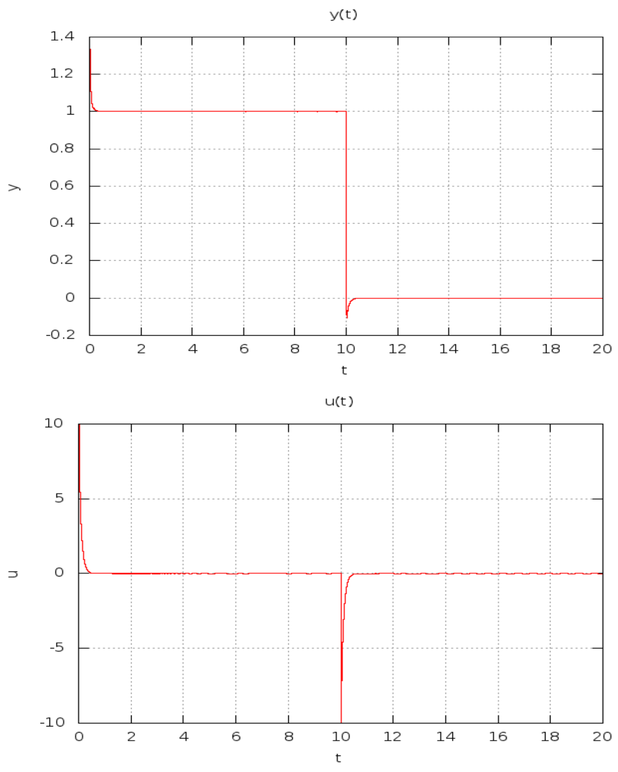
Due to a lack of time, implementing a kernel-module, as well as functionality for sending UDP-messages in kernel-space became rather infeasable, and we chose not to do it.

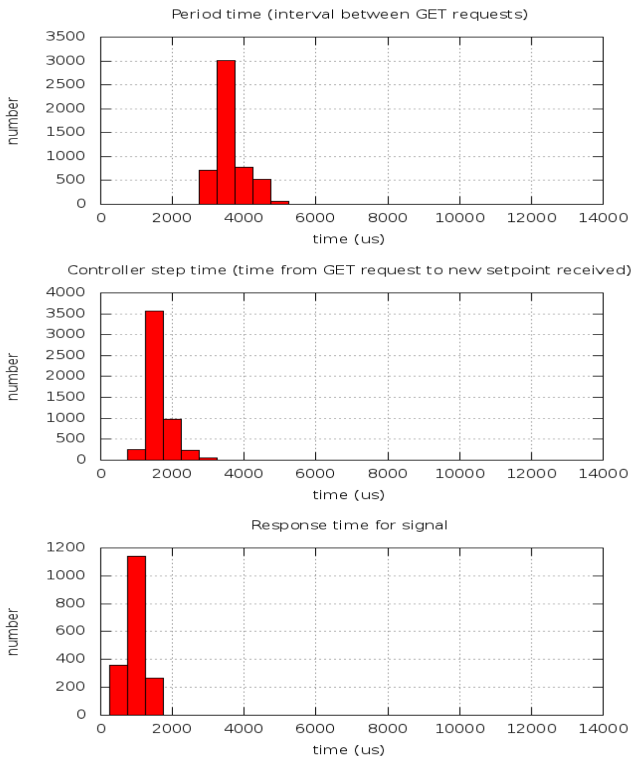


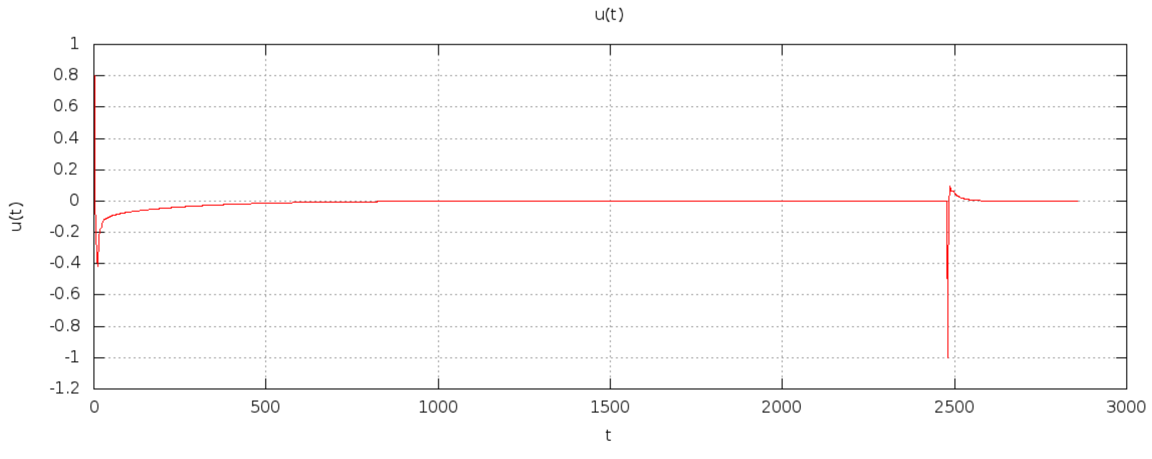
u(t)
u(t)
t
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.2
0
500
1000
1500
2000
2500
3000

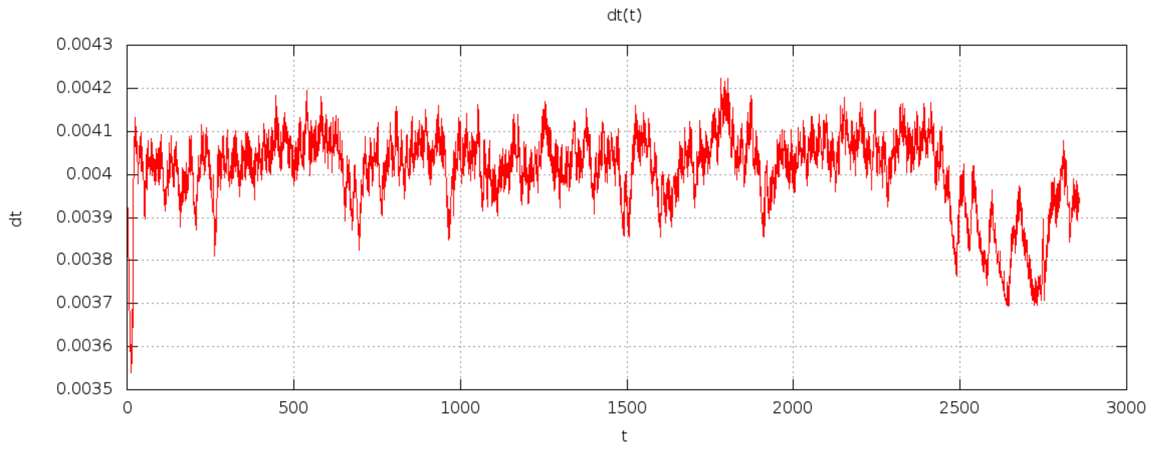


dt(t)
dt
t
0.0043
0.0042
0.0041
0.004
0.0039
0.0038
0.0037
0.0036
0.0035
0
500
1000
1500
2000
2500
3000


y(t)
y
t
u(t)
u
t


Period time (interval between GET requests)
number
time (us)
Controller step time (time from GET request to new setpoint received)
number
time (us)
Response time for signal
number
time (us)


u(t)
u(t)
t


dt(t)
dt
t