

Report: Performance and robustness of switched control with experimental verification

Stephan Trenn

11/03/2005

1 Introduction

Control theory is a wide interdisciplinary area in mathematics and engineering. In many applications a given system should behave in an appropriate way. Even if the system is completely known it is not trivial which control action yields the best result. Mathematics were the key to find the “best” controllers for a given system. In reality one normally doesn’t know the system in every detail, there are uncertainties in the parameters of the plant as well as in the dynamical behaviour. As long as the uncertainties are small the controller designed for the nominal plant will in most cases also work with the uncertain plant. In some cases it can not be assumed, that the uncertainties are small and hence a fixed controller will not work. In general an adaptive controller is then necessary. In the special case that one can assume that there is a set of candidate plants, with a suitable fixed controller for each, switched control might be considered. The main idea is, that the switched controller observes the input and output signals of the system and concludes then which candidate controller suits best. In this report the set of candidate plants is finite and each plant is a discrete linear system, which arises from a discretisation of a continuous plant. As candidate controllers the corresponding dead-beat controller is chosen, which allows an easy analysis and nice theoretical results.

The aim of this report is to show that switched control is robust. The main result is Theorem 2 which shows that switched control has a finite l_p -gain, which implies robustness. These theoretical results are then considered for a real plant. The experiments show that the switched controller stabilizes the real plant. It must be admitted that the experiments are not very substantial and there are a lot more interesting experiments which could be done.

2 Theoretical background

2.1 Discretisation of continuous systems

Since a computer will be used for the experimental implementation of the switched controller one have to consider a discretised version of the continuous plant. The discretisation arises from adding digital to analog and analog to digital converters with a given sampling period $\tau > 0$ (see Figure 1). It is assumed

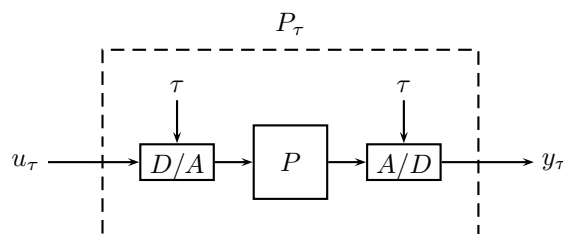


Figure 1: Discretisation of a continuous plant

that the continuous plant is a finite dimensional linear operator, that the sampling period is constant and that the digital to analog converter has a constant output between to sampling times. Under these assumptions the resulting discrete plant is again a finite dimensional linear operator. To be more precisely,

let the continuous plant $P : \mathcal{U} \rightarrow \mathcal{Y}, u \mapsto y$, with \mathcal{U} some continuous input space (e.g. $L_\infty(\mathbb{R}, \mathbb{R})$) and \mathcal{Y} some output space, be described by the following state space model (for $t \geq 0$):

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t). \end{aligned}$$

The matrixes A, B, C and D should have appropriate dimensions and the initial condition is $x(0) = 0$. Let $\mathcal{U}_\tau \subset \mathcal{U}$ and $\mathcal{Y}_\tau \subset \mathcal{Y}$ be the signal spaces of step functions, which are constant on the intervals $[k\tau, (k+1)\tau)$ for all $k \in \mathbb{N}$. The space of discrete signals can easily be identified with \mathcal{U}_τ or \mathcal{Y}_τ , resp., and therefore these symbols will be used for the discrete signal spaces. The discrete version $P_\tau : \mathcal{U}_\tau \rightarrow \mathcal{Y}_\tau, u_\tau \mapsto y_\tau$ of the continuous plant can then be described by (for $k \in \mathbb{N}$)

$$\begin{aligned} x_\tau(k+1) &= A_\tau x_\tau(k) + B_\tau u_\tau(k) \\ y_\tau(k) &= C_\tau x_\tau(k) + D_\tau u_\tau(k). \end{aligned}$$

As in Section 2.3 of [1] shown the coefficient matrices can be obtain by

$$\begin{aligned} A_\tau &= e^{A\tau}, \\ B_\tau &= \int_0^\tau e^{As} ds B, \\ C_\tau &= C \quad \text{and} \\ D_\tau &= D. \end{aligned}$$

2.2 Gap metric and robustness

Consider the closed loop system from Figure 2. It will be assumed that $P : \mathcal{U}_e \rightarrow \mathcal{Y}_e$ and $C : \mathcal{Y}_e \rightarrow \mathcal{U}_e$

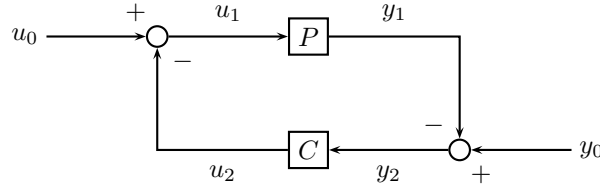


Figure 2: The closed loop system $[P, C]$

are causal operators with $P0 = 0$ and $C0 = 0$, where \mathcal{U} and \mathcal{Y} are appropriate (discrete or continuous) complete signal spaces and \mathcal{U}_e and \mathcal{Y}_e are the extended signal spaces. Let $\mathcal{W} := \mathcal{U} \times \mathcal{Y}$ and $\mathcal{W}_e := \mathcal{U}_e \times \mathcal{Y}_e$. Furthermore it will be assumed that the closed loop system is well-posed, then the following equations hold:

$$\begin{aligned} u_0 &= u_1 + u_2 \\ y_0 &= y_1 + y_2 \\ y_1 &= Pu_1 \\ u_2 &= Cy_2 \end{aligned}$$

and

$$H_{P,C} : \mathcal{W} \rightarrow \mathcal{W}_e \times \mathcal{W}_e : \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} \mapsto \left(\begin{pmatrix} u_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} u_2 \\ y_2 \end{pmatrix} \right)$$

is a well defined causal operator. The norm of $H_{P,C}$ is defined as

$$\|H_{P,C}\| := \sup \left\{ \frac{\left\| \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} \right\|}{\left\| \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} \right\|} \mid \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} \neq 0, \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} \in \mathcal{W} \times \mathcal{W} \right\}$$

If the set on the right hand side is empty the norm is set to ∞ . The operator $H_{P,C}$ is called stable if its norm is finite, i.e. if bounded disturbances result in uniformly bounded internal signals. Consider the “component operators” of $H_{P,C}$

$$\begin{aligned} H_{P,C}^1 : \mathcal{W} &\rightarrow \mathcal{W}_e, \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} \mapsto \begin{pmatrix} u_1 \\ y_1 \end{pmatrix}, \\ H_{P,C}^2 : \mathcal{W} &\rightarrow \mathcal{W}_e, \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} \mapsto \begin{pmatrix} u_2 \\ y_2 \end{pmatrix}, \end{aligned}$$

then it easy to see, that

$$H_{P,C}^1 + H_{P,C}^2 = I.$$

Hence

$$\|H_{P,C}\| < \infty \Leftrightarrow \|H_{P,C}^1\| < \infty \Leftrightarrow \|H_{P,C}^2\| < \infty.$$

Assume that an appropriate metric (the gap metric) $\delta(\cdot, \cdot)$ is given for the space of operators $P : \mathcal{U}_e \rightarrow \mathcal{U}_e$ (see [2]) for possible definitions), then the following theorem holds

Theorem 1 ([2]) *Consider the closed loop system $[P, C]$ as in Figure 2. If a system $P_1 : \mathcal{U}_e \rightarrow \mathcal{U}_e$ fulfills*

$$\delta(P_1, P) < \frac{1}{\|H_{P,C}^1\|},$$

then the closed loop system $[P_1, C]$ is stable.

This means, that if the gain from the disturbance to the internal signals is bounded for the nominal plant, then the same controller stabilises a whole environment around P . Therefore it is for robustness sufficient to consider the performance of a closed loop system, which is done in the next Sub-Section.

Note that in the context of discretisation only the robustness of the discrete closed loop system will be considered. The question, if there exists an environment around the continuous system, such that the resulting new discrete closed loop system is still stable, is not considered here.

2.3 Performance of switched control

In this Sub-Section only discrete systems will be considered, therefore the index τ , which was used in Sub-Section 2.1 to distinguish the continuous plant from the discrete plant, will be suppressed.

2.3.1 Definition of switched control

Consider the closed loop system from Figure 2 and assume that $(u_0, y_0) \in V^2$, where V is $l_r(\mathbb{N}, \mathbb{R})$ for $r \in [1, \infty]$.

For $p \in \mathcal{P} = 1, \dots, N$, $N \in \mathbb{N}$, the operator

$$P_p : V_e \rightarrow V_e, \quad u_1 \mapsto P\{u_1\} = y_1,$$

V_e the extended space of V , is described by

$$y_1(k) = \sum_{i=1}^{\sigma} a_p^i y_1(k-i) + b_p u_1(k-i), \quad y_1(-i) = u(-i) = 0 \quad \forall i \in \mathbb{N} \quad (1)$$

where $a_p^1, \dots, a_p^\sigma, b_p \in \mathbb{R}^\sigma$ are known and $\sigma \in \mathbb{N}$ is the maximum order of the plants P_p . It will be assumed that

$$b_p \neq 0 \quad \forall p \in \mathcal{P}$$

It is also known that $P = P_{p^*}$ for an unknown $p^* \in \mathcal{P}$. The aim is to construct a causal operator

$$C : V_e \rightarrow V_e, \quad y_2 \mapsto C\{y_2\} = u_2$$

such that the output y_2 of the closed loop $[P, C]$ fulfills for every $(u_0, y_0) \in V^2$ the property $y_2 \in V$, i.e. (norm-)bounded disturbances yield a (norm-)bounded output. Furthermore we are aiming for performance results, which than can be used to prove robustness.

It will be assumed that there exists suitable controllers C_p for every $p \in \mathcal{P}$, here it will be assumed that for $p \in \mathcal{P}$ that C_p is a "dead-beat" controller, i.e.

$$C_p : V_e \rightarrow V_e, \quad y_2 \mapsto \left(k \mapsto u_2(k) = -\frac{1}{b_p} \left(\sum_{i=1}^{\sigma} a_p^i y_2(k-i+1) \right) \right). \quad (2)$$

The aim is then to switch to the "right" controller. The structure of C is illustrated in Figure 3.

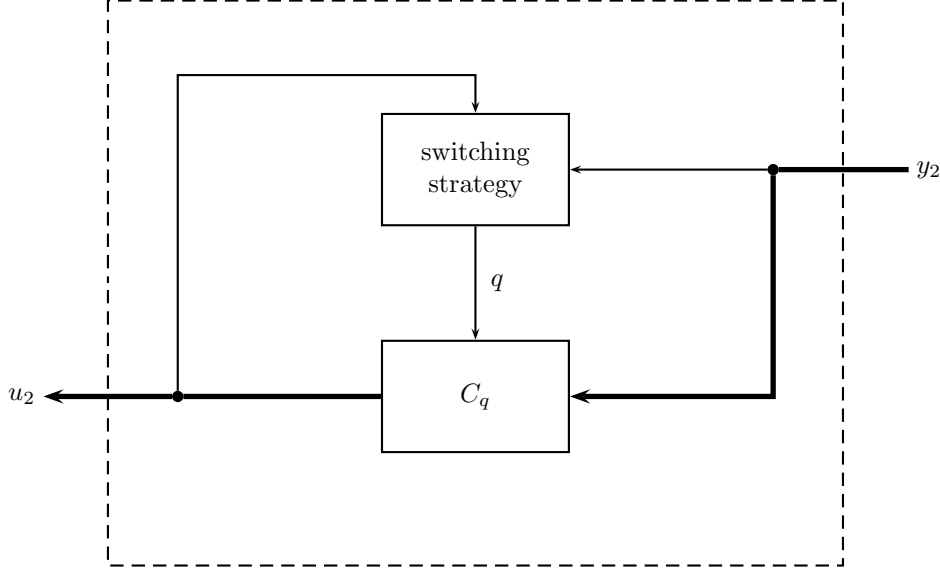


Figure 3: The structure of the controller C

The switching strategy S is a causal operator of the form

$$S : V_e \times V_e \rightarrow \text{map}(\mathbb{N}, \mathcal{P}), \quad (y_2, u_2) \mapsto q,$$

with the property

$$S\{y_2, u_2\}|_{[0, k]} = S\{y_2|_{[0, k]}, u_2|_{[0, k-1]}\}|_{[0, k]}.$$

With $q(k) = S\{y_2, u_2\}(k)$ the controller C can then be described by

$$C\{y_2\}(k) = C_{q(k)}\{y_2\}(k) \quad \forall k \in \mathbb{N}. \quad (3)$$

The switching strategy S will be structured as indicated by Figure 4.

For $p \in \mathcal{P}$ the disturbance estimation d_p^k at time $k \in \mathbb{N}$ is a vector of length k , i.e.

$$d_p^k = (d_p^k(0), d_p^k(1), d_p^k(2), \dots, d_p^k(k))$$

where $d_p^k(i) \in \mathbb{R}^d$ for $i \in \{1, \dots, k\}$ where $d \in \mathbb{N}$ is a number depending on the specific chosen estimation and the order σ . No specific estimation will be considered, but only some general assumptions on them. The switching strategy uses for $p \in \mathcal{P}$ the value

$$D_p(k) := \|d_p^k\|_V \quad \forall k \in \mathbb{N}, \quad (4)$$

where d_p^k will be treated as a truncated element of V , and the strategy chooses then as actual controller $C_{q(k)}$, where

$$S\{y_2, u_2\}(k) = q(k) = \underset{p \in \mathcal{P}}{\text{argmin}} D_p(k). \quad (5)$$

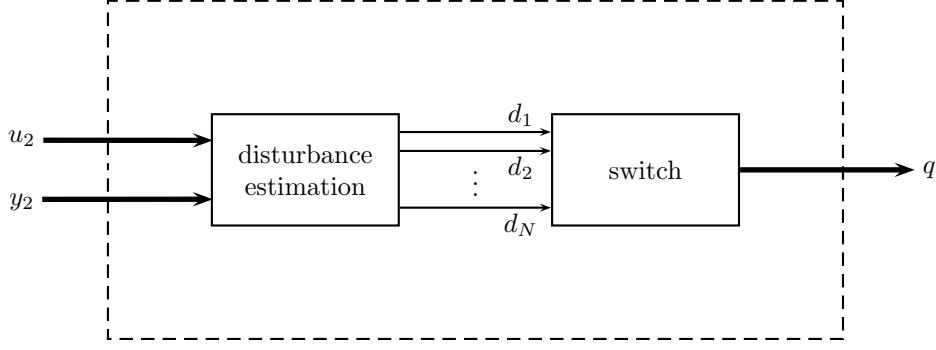


Figure 4: The structure of the switching strategy S

2.3.2 The disturbance estimations

Assumptions on the estimation:

1. For every $k \in \mathbb{N}$ and $p \in \mathcal{P}$ the estimation d_p^k only depends on $y_2|_{[0,k]}$ and $u_2|_{[0,k-1]}$.
2. There exists a constant $c_1 > 0$ such that $D_{p^*}(k) \leq c_1 \|u_0, y_0\|_V$ for all $k \in \mathbb{N}$.
3. There exists a constant $c_2 > 0$ such that $|y_2(k)| \leq c_2 \left\| d_{q(k-1)}^k|_{[k-\sigma, k]} \right\|_V$ for all $k \in \mathbb{N}$.
4. For all $p \in \mathcal{P}$ and $0 \leq k < k'$: $\|d_p^k\|_V \leq \|d_p^{k'}\|_{[0, k]}$

Property 1 yields that for the disturbance estimation only information is used, which is available at time k , i.e. causality. Assumption 2 ensures that for the estimation of the real plant the estimated disturbances are bounded by the real disturbances. The third assumption is technical and the idea is, that if at time $k-1$ the controller q was chosen, then the value of $y_2(k)$ can in plant P_q only arise of the current (estimated) disturbances, since the “memory” of P_q was cancelled by the corresponding controller C_q . Assumption 4 reflects a kind of minimality of every disturbance estimation.

2.3.3 Performance result

Theorem 2 *Let $V = l_r(\mathbb{N}, \mathbb{R})$ for $r \in [1, \infty]$ and consider the known plants $P_p : V_e \rightarrow V_e$, with p an element of some finite set \mathcal{P} , which are given by (1). Let $P = P_{p^*}$ for an unknown $p^* \in \mathcal{P}$, and let the controller $C : V_e \rightarrow V_e$ be defined as in (3) with a switching strategy defined by (5) and (4) and let the disturbance estimations fulfill Assumptions 1-4. Then the closed-loop system $[P, C]$ with notation as in Figure 2 fulfill*

1. $(u_0, y_0) \in V^2 \Rightarrow y_2 \in V$
2. There exists $\gamma > 0$, such that for all $(u_0, y_0) \in V^2$

$$\|y_2\|_V \leq \gamma \|u_0, y_0\|_V$$

Proof. It is clear that the second assertion implies the first one and therefore only the existence of $\gamma > 0$ such that $\|y_2\|_V \leq \gamma \|u_0, y_0\|_V$ for all $(u_0, y_0) \in V^2$ will be shown.

Let $(u_0, y_0) \in V^2$, then there exist unique signals $(u_2, y_2) \in V_e^2$ of the closed-loop system $[P, C]$ as well as unique disturbance estimations d_p^k for $p \in \mathcal{P}$ and for $k \in \mathbb{N}$. Write $q(k) = S\{y_2, u_2\}(k)$ for the switching-signal and let $Q = \{k_0 = 0, k_1, k_2, \dots\}$ be the set of switching times with $k_i < k_{i+1}$ for all $i \in \mathbb{N}$, i.e.

$$q_i := q(k_i) = q(k_i + l) \neq q(k_{i+1}) \quad \forall i \in \mathbb{N} \quad 0 \leq l < k_{i+1} - k_i.$$

If Q is a finite set we set $k_{|Q|+2} = \infty$ and ignore all k_i s for $i > |Q| + 2$. Write for $i \in \mathbb{N}$

$$y_2^i := y_2|_{[k_i+1, k_{i+1}-1]}$$

and observe that

$$\|y_2\|_V = \left\| \|y_2^0, y_2^1, y_2^2, \dots\|_V, \|y_2(k_0), y_2(k_1), y_2(k_2), \dots\|_V \right\|_V$$

We will show that there exist $\gamma_1 > 0$ and $\gamma_2 > 0$ such that

$$\|y_2^0, y_2^1, y_2^2, \dots\|_V \leq \gamma_1 \|u_0, y_0\|_V.$$

and

$$\|y_2(k_0), y_2(k_1), y_2(k_2), \dots\|_V \leq \gamma_2 \|u_0, y_0\|_V$$

The proof of the theorem would then with $\gamma = \|\gamma_1, \gamma_2\|_V$ be complete.

STEP 1: Showing $\exists \gamma_1 > 0$: $\|y_2^0, y_2^1, y_2^2, \dots\|_V \leq \gamma_1 \|u_0, y_0\|_V$

We first show inductively that for every $n \in \mathbb{N}$

$$\|y_2^0, y_2^1, \dots, y_2^n\|_V \leq c_2 \left\| \begin{array}{c} d_{q_n}^{k_{n+1}-1} \\ d_{q_n}^{k_{n+1}-2} \\ \vdots \\ d_{q_n}^{k_{n+1}-(\sigma+1)} \end{array} \right\|_V. \quad (6)$$

If for any $n \in \mathbb{N}$ the difference $k_{n+1} - k_n$ is smaller than $\sigma + 1$ then the last entry would be $d_{q_n}^{k_n}$ instead of $d_{q_n}^{k_{n+1}-(\sigma+1)}$.

We introduce the notation

$$d_q^a|_{[-b]} := d_q^a|_{[a-b, a]} \quad \text{for } q \in \mathcal{P}, a, b \in \mathbb{N}$$

Starting with $n = 0$ we observe that by Assumption 3

$$\begin{aligned} \|y_2^0\|_V &= \|y_2(k_0 + 1), y_2(k_0 + 2), \dots, y_2(k_1 - 1)\|_V \\ &\leq c_2 \left\| \left\| d_{q_0}^{k_0+1}|_{[-\sigma]} \right\|_V, \left\| d_{q_0}^{k_0+2}|_{[-\sigma]} \right\|_V, \dots, \left\| d_{q_0}^{k_1-1}|_{[-\sigma]} \right\|_V \right\|_V \\ &= c_2 \left\| \begin{array}{c} \left(d_{q_0}^{k_0+1}|_{[-\sigma]}, d_{q_0}^{k_0+1+(\sigma+1)}|_{[-\sigma]}, d_{q_0}^{k_0+1+2(\sigma+1)}|_{[-\sigma]}, \dots \right) \\ \left(d_{q_0}^{k_0+2}|_{[-\sigma]}, d_{q_0}^{k_0+2+(\sigma+1)}|_{[-\sigma]}, d_{q_0}^{k_0+2+2(\sigma+1)}|_{[-\sigma]}, \dots \right) \\ \vdots \\ \left(d_{q_0}^{k_0+\sigma+1}|_{[-\sigma]}, d_{q_0}^{k_0+2(\sigma+1)}|_{[-\sigma]}, d_{q_0}^{k_0+3(\sigma+1)}|_{[-\sigma]}, \dots \right) \end{array} \right\|_V \end{aligned}$$

Note that every row is finite and the last entries are

$$d_{q_0}^{k_1-1}|_{[-\sigma]}, d_{q_0}^{k_1-2}|_{[-\sigma]}, \dots, d_{q_0}^{k_1-(\sigma+1)}|_{[-\sigma]}$$

but not necessarily in this order. Using now successively Assumption 4 and the simply general fact for any sequence s and $a_1 < a < b < b_1$

$$\|s|_{[a,b]}\|_V \leq \|s|_{[a_1, b_2]}\|_V$$

we arrive at

$$\|y_2^0\|_V \leq \left\| \begin{array}{c} d_{q_0}^{k_1-1} \\ d_{q_0}^{k_1-2} \\ \vdots \\ d_{q_0}^{k_1-(\sigma+1)} \end{array} \right\|_V$$

Therefore $n = 0$ is shown, for $n > 0$ observe first that

$$\left\| \begin{array}{c} d_{q_{n-1}}^{k_n-1} \\ d_{q_{n-1}}^{k_n-2} \\ \vdots \\ d_{q_{n-1}}^{k_n-(\sigma+1)} \end{array} \right\|_V = \left\| \begin{array}{c} D_{q_{n-1}}(k_n - 1) \\ D_{q_{n-1}}(k_n - 2) \\ \vdots \\ D_{q_{n-1}}(k_n - (\sigma + 1)) \end{array} \right\|_V \stackrel{(5)}{\leq} \left\| \begin{array}{c} D_{q_n}(k_n - 1) \\ D_{q_n}(k_n - 2) \\ \vdots \\ D_{q_n}(k_n - (\sigma + 1)) \end{array} \right\|_V \leq \left\| \begin{array}{c} d_{q_n}^{k_n} \\ d_{q_n}^{k_n-1} \\ \vdots \\ d_{q_n}^{k_n-\sigma} \end{array} \right\|_V.$$

Now

$$\begin{aligned}
\|y_2^0, y_2^1, \dots, y_2^n\|_V &= \left\| \|y_2^0, y_2^1, \dots, y_2^{n-1}\|_V, y_2^n \right\| \\
&\leq c_2 \left\| \left\| \begin{array}{c} d_{q_n}^{k_n} \\ d_{q_n}^{k_n-1} \\ \vdots \\ d_{q_n}^{k_n-\sigma} \end{array} \right\|_V, \left\| d_{q_n}^{k_n+1} \right\|_{[-\sigma]}, \left\| d_{q_n}^{k_n+2} \right\|_{[-\sigma]}, \dots, \left\| d_{q_n}^{k_n+1-1} \right\|_{[-\sigma]} \right\|_V \\
&= c_2 \left\| \begin{array}{cccc} d_{q_n}^{k_n-\sigma}, & d_{q_n}^{k_n+1+0(\sigma+1)}|_{[-\sigma]}, & d_{q_n}^{k_n+1+1(\sigma+1)}|_{[-\sigma]}, & \dots \\ d_{q_n}^{k_n+1-\sigma}, & d_{q_n}^{k_n+2+0(\sigma+1)}|_{[-\sigma]}, & d_{q_n}^{k_n+2+1(\sigma+1)}|_{[-\sigma]}, & \dots \\ & \vdots & & \\ d_{q_n}^{k_n}, & d_{q_n}^{k_n+\sigma+1}|_{[-\sigma]}, & d_{q_n}^{k_n+2(\sigma+1)}|_{[-\sigma]}, & \dots \end{array} \right\|_V
\end{aligned}$$

and as in the case $n = 0$ we can conclude (6).

Since

$$\|d_{q_n}^{k_n+1-i}\|_V \leq d_{q_n}^{k_n+1-1} \quad \forall i \in \{1, \dots, \sigma+1\}$$

inequality (6) yields for all $n \in \mathbb{N}$

$$\begin{aligned}
\|y_2^0, y_2^1, \dots, y_2^n\|_V &\leq c_2 \underbrace{\|1, 1, \dots, 1\|_V}_{\sigma+1} \|d_{q_n}^{k_n+1-1}\|_V \\
&= c_2 \|1, 1, \dots, 1\|_V D_{q_n}(k_{n+1} - 1) \stackrel{(5)}{\leq} c_2 \|1, 1, \dots, 1\|_V D_{p^*}(k_{n+1} - 1) \\
&\stackrel{\text{Ass. 2}}{\leq} \underbrace{c_2 \|1, 1, \dots, 1\|_V c_1}_{=:\gamma_1} \|u_0, y_0\|_V.
\end{aligned}$$

Hence Step 1 is finished.

STEP 2: Showing $\exists \gamma_2 > 0$: $\|y_2(k_0), y_2(k_1), y_2(k_2), \dots\|_V \leq \gamma_2 \|u_0, y_0\|_V$.

For $\hat{q} \in \mathcal{P}$ consider the subset $Q^{\hat{q}} \subseteq Q$ of all times where the switching strategy switched from controller $C_{\hat{q}}$ to another one, i.e. for all $k \in Q^{\hat{q}}$: $q(k-1) = \hat{q}$ and for all $k \in Q \setminus Q^{\hat{q}}$: $q(k-1) \neq \hat{q}$. Writing $Q^{\hat{q}} = \{k_1^{\hat{q}}, k_2^{\hat{q}}, \dots\}$ we will show that for all $\hat{q} \in \mathcal{P}$ there exists $\gamma_2^{\hat{q}} > 0$ such that

$$\left\| y_2(k_1^{\hat{q}}), y_2(k_2^{\hat{q}}), \dots \right\|_V \leq \gamma_2^{\hat{q}} \|u_0, y_0\|_V. \quad (7)$$

For

$$\gamma_2 := \left\| \gamma_2^{\hat{q}_1}, \gamma_2^{\hat{q}_2}, \dots, \gamma_2^{\hat{q}_{|\mathcal{P}|}} \right\|_V$$

Step 2 would then be shown.

Let $n \in \mathbb{N}$ such that $k_n^{\hat{q}} \in Q^{\hat{q}}$ we know then from Assertion 3 that for $i \in \{1, 2, \dots, n\}$

$$\left| y_2(k_i^{\hat{q}}) \right| \leq c_2 \left\| d_{\hat{q}}^{k_i^{\hat{q}}}(k_i^{\hat{q}} - \sigma), \dots, d_{\hat{q}}^{k_i^{\hat{q}}}(k_i^{\hat{q}}) \right\|_V.$$

Using successively Assumption 4 similar as in Step 1 we arrive at

$$\left\| y_2(k_1^{\hat{q}}), y_2(k_2^{\hat{q}}), \dots, y_2(k_n^{\hat{q}}) \right\|_V \leq c_2 \underbrace{\|1, 1, \dots, 1\|_V}_{\sigma+1} \underbrace{\left\| d_{\hat{q}}^{k_n^{\hat{q}}} \right\|_V}_{=D_{\hat{q}}(k_n^{\hat{q}})}$$

If $k_n^{\hat{q}}$ is not the last element in the ordered set $Q^{\hat{q}}$ then there must exist $k > k_n^{\hat{q}}$ such that the switching strategy is switching again to the controller $C_{\hat{q}}$ at time k (otherwise it could not switch away from \hat{q} later). In particular

$$D_{\hat{q}}(k_n^{\hat{q}}) \leq D_{\hat{q}}(k) \leq D_{p^*}(k) \leq c_1 \|u_0, y_0\|_V$$

and hence, with $\gamma_1 > 0$ as in Step 1,

$$\left\| y_2(k_1^{\hat{q}}), y_2(k_2^{\hat{q}}), \dots, y_2(k_n^{\hat{q}}) \right\|_V \leq \gamma_1 \|u_0, y_0\|_V. \quad (8)$$

Therefore if $Q^{\hat{q}}$ is infinite we have shown the existence of $\gamma_2^{\hat{q}} > 0$ such that (7) holds.

It remains to consider the cases where $Q^{\hat{q}}$ is finite. Define

$$F := \{ \hat{q} \in \mathcal{P} \mid Q^{\hat{q}} \text{ is finite} \}$$

and let

$$K_F := \{ k \in \mathbb{N} \mid k = \max Q^{\hat{q}}, \hat{q} \in F \}$$

be the set of all times at which the switching strategy switches away from a controller $C_{\hat{q}}$ the last time. Writing $K_F = \{k_1^F, k_2^F, \dots, k_{|F|}^F\}$ we will for $i \in \{1, \dots, |F|\}$ show that there exists $\alpha > 0$ and $\tilde{\gamma}_1 > 0$ such that

$$|y_2(k_i^F)| < \tilde{\gamma}_1 \alpha^i \|u_0, y_0\| \quad (9)$$

We are proving this inductively and consider first the case $i = 1$. By Step 1 and (8) we know

$$|y_2(k)| \leq \gamma_1 \|u_0, y_0\|_V \quad \forall k < k_1^F.$$

Furthermore we know by (2) that

$$|u_2(k)| \leq c_3 \|y_2|_{[k-\sigma, k]}\|_V \leq \overbrace{c_3 \|1, 1, \dots, 1\|_V}^{\tilde{\gamma}_1} \gamma_1 \|u_0, y_0\|_V \quad \forall k < k_1^F.$$

We assume $\tilde{\gamma}_1 \geq \gamma_1$. Now (1) yields

$$\begin{aligned} |y_2(k_1^F)| &\leq \sum_{i=1}^{\sigma} (|a_{p^*}^i| |y_2(k-i)| + |b_{p^*}^i| |u_2(k-i)|) + \left| y_0(k) - \sum_{i=1}^{\sigma} (a_{p^*}^i y_0(k-i) + b_{p^*}^i u_0(k-i)) \right| \\ &\leq \gamma_1 \sum_{i=1}^{\sigma} |a_{p^*}^i| \|u_0, y_0\|_V + \tilde{\gamma}_1 \sum_{i=1}^{\sigma} |b_{p^*}^i| \|u_0, y_0\|_V + \left(1 + \sum_{i=1}^{\sigma} (|a_{p^*}^i| + |b_{p^*}^i|) \right) \|u_0, y_0\|_V \\ &\leq \tilde{\gamma}_1 \alpha \|u_0, y_0\|_V \end{aligned}$$

where

$$\alpha := \sum_{i=1}^{\sigma} (|a_{p^*}^i| + |b_{p^*}^i|) + \frac{1}{\tilde{\gamma}_1} \left(1 + \sum_{i=1}^{\sigma} (|a_{p^*}^i| + |b_{p^*}^i|) \right).$$

We can assume that $\alpha \geq 1$. For the case $i > 1$ we then know

$$|y_2(k)| \leq \tilde{\gamma}_1 \alpha^{i-1} \|u_0, y_0\| \quad \text{and} \quad |u_2(k)| \leq \tilde{\gamma}_1 \alpha^{i-1} \|u_0, y_0\|_V \quad \forall k < k_i^F.$$

The same calculation as in the case $i = 0$ yields then

$$|y_2(k_i^F)| \leq \tilde{\gamma}_1 \alpha \alpha^{i-1} \|u_0, y_0\|_V = \tilde{\gamma}_1 \alpha^i \|u_0, y_0\|_V.$$

For $\hat{q} \in \mathcal{P}$ let $i_{\hat{q}}$ such that $\{k_{i_{\hat{q}}}^F\} = K_F \cap Q^{\hat{q}}$ if $Q^{\hat{q}}$ is finite and $i_{\hat{q}} = -\infty$ otherwise. Then (8) and (9) together yields (7) with

$$\gamma_2^{\hat{q}} = \|\gamma_1, \tilde{\gamma}_1 \alpha^{i_{\hat{q}}}\|_V.$$

□

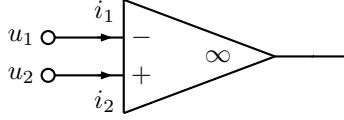


Figure 5: An idealistic operational amplifier.

2.4 Properties of electrical circuits

It is recalled that for an idealistic resistor the following equation holds

$$R = \frac{u(t)}{i(t)},$$

where $t \mapsto u(t)$ is the voltage over the resistor, $t \mapsto i(t)$ the current through it and R the constant quotient of both. An idealistic capacitor has the characteristic equation

$$i(t) = C \frac{du(t)}{dt}.$$

An idealistic operational amplifier (see Figure 5) fulfills the following two equations (if used with negative and/or positive feedback)

$$u_1 = u_2 \quad \text{and} \quad i_1 = i_2 = 0.$$

Together with Kirchhoff's circuit laws it is now possible to obtain linear differential equations which describe the behaviour of any circuit consisting of resistors, capacitors and operational amplifiers.

3 Experimental setup

3.1 Hardware

In the experiments a standard PC (Intel Pentium III 450 MHz Processor, 192 MB installed RAM) was used. For the analog-digital and digital-analog conversion the ISA-PC-card *CIO-DAS08/JR-AO* (which is the card *CIO-DAS08/JR* with the additional chip *CIO-DUAL-DAC*) from the company "Measurement Computing Corp." was installed in the PC. To have easy access to the signals the screw terminal *CIO-MINI37* from the same company was connected with the card. The electrical circuit consists of standard resistors and capacities as well as the standard operational amplifier *LM324N*. The circuit is illustrated in Figure 6.

With Sub-Section 2.4 it is easy to see that the systems behaviour is described by

$$\dot{u}_{\text{out}} = \left(\frac{1}{R_1 C} - \frac{1}{R_2 C} \right) u_{\text{out}} - \frac{2}{R_1 C} u_{\text{in}}.$$

In the implementation the resistors and the capacitor had the following ideal values:

$$\begin{aligned} R_1 &= 22k\Omega \parallel 2.2M\Omega \approx 21.87k\Omega, \\ R_2 &= 22k\Omega, \\ C &= 47pF \\ (R_3 &= 2.2M\Omega). \end{aligned}$$

Using the notation of Sub-Section 2.1 this results in

$$\begin{aligned} A &= \left(\frac{1}{R_1 C} - \frac{1}{R_2 C} \right) \approx 9.67, \\ B &= -\frac{2}{R_1 C} \approx -1954, \\ C &= 1, \quad D = 0. \end{aligned}$$

with

$$u = u_{\text{in}}, \quad y = u_{\text{out}}.$$

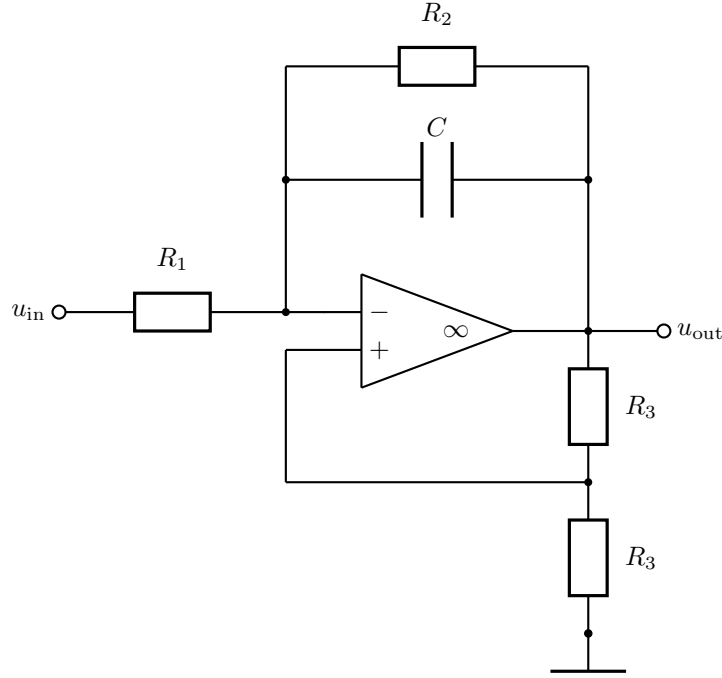


Figure 6: The electrical circuit which should be controlled.

The accuracy of the resistors with value $22k\Omega$ was 1% and of the $2.2M\Omega$ the accuracy is 5%. It is assumed that the capacitor has also an accuracy of 5%. This yields the following possible intervals for A and B if it is still assumed that the circuit elements, in particular the operational amplifier are ideal,

$$A \in [-98, 141] \quad B \in [1753, 2187].$$

Although negative values for A are possible, they need not to be considered since the real system showed clearly an unstable behaviour.

A sampling rate of $1000Hz$ was used therefore, for the nominal values,

$$\begin{aligned} A_\tau &= e^{A\tau} \approx 1.01, \\ B_\tau &= B/A(e^{A\tau} - 1) \approx -1.96, \\ C_\tau &= 1, \quad D_\tau = 0. \end{aligned}$$

It is important to note that the maximum input voltage range is $-5V$ to $+5V$ and the output voltage is truncate to this range as well. Since the power supply of the operational amplifier is $\pm 12V$ the systems behaviour is well described by the equations above if the input and output signals are within $[-5V, +5V]$. At the beginning of the experiment the output voltage will not be in the normal range, because the system is unstable. Therefore some action is necessary to get the system's signals back to the "normal" range. One possibility is to shortcut the capacitor which results in a zero output. The problem is then that the shortcut must be open manually at the beginning of the experiment. Since the duration of the experiment could be less than a second this is not very practicable.

In experiments and simulations it was observed that the dead beat controller is working even if the signals are not in the normal working range. It should be possible to show this theoretical, but this is out of focus of this report. Therefore the switched controller is implemented in such a way that first a certain number of steps the dead beat controller is running which results in an output near zero. After these steps the real experiment is started.

3.2 Software

As mentioned in Sub-Section 2.1 it is important that the sampling period is constant. Under modern operation system, like Microsoft Windows, this is in general not possible, since one have no direct access to

the hardware and because of multitasking it is not clear, when a programme is actual running. Therefore the operation system DOS was used. In particular the freely available operation system *FreeDOS* (Beta 9 Release #1) was installed. The programming language was *C* and the freely available development system *DJGPP* (Version 2.03) was chosen for programming and compiling.

For accomplishing the experiment it was necessary to write a small programm with the following main features:

- Changing the timer interrupt
- Implementing a (discrete) controller
- Saving the experimental data to disc
- Testing the AD/DA-card

The structure of the programme is illustrated in Figure 7.

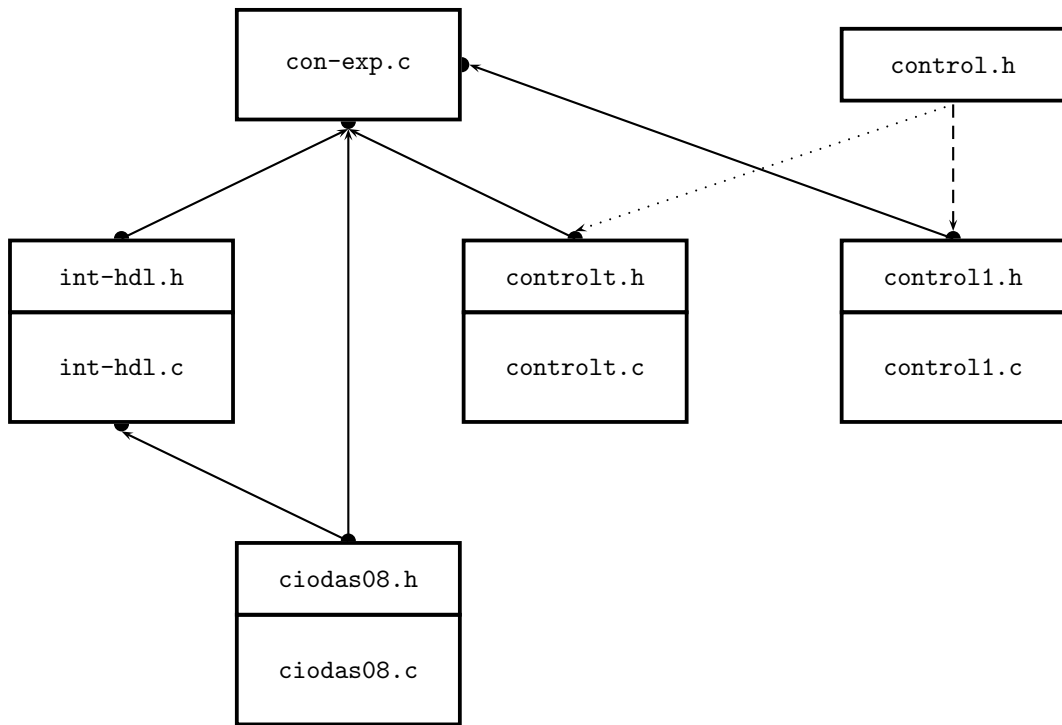


Figure 7: Structure of the control experiment software

In the file `int-hdl.c` the timer interrupt is handled. Since the interrupt service routine starts the analog digital conversion there is a connection to the file `ciodas08.c`. The latter provides all necessary routines for the use of the AD/DA-card. A test routine for the AD/DA-card is implemented in `controlt.c`. The controller is implemented in `control1.c`. If another controller should be used, then the file `control.h` should be implemented in an appropriate way. If the controller is implemented in another file than `control1.c` (which is recommended) then it is sufficient to change the corresponding line `#include "control1.h"` in `con-exp.c`.

It turned out that changing the timer interrupt was the most difficult task. The main reason for this was that the programme runs in protected mode. In protected mode the programme does not have direct access to the memory, but on the other hand there are no practical restriction for the size of variables.

The current implementation of the interrupt service routine is derived from the source code of [3].

It is possible to save the data of the experiment into files (one for the input data and one for the output data). These files are text files and the numbers are separated by spaces. With *Matlab* it is very easy to read these files and to display the data.

The source code of all files can be found in the appendix.

4 Results

4.1 The switched controller

The theoretical results in Sub-Section 2.3 considers all discrete Lebesgue-spaces l_r for $r \in [1, \infty]$. Since there is always a discretisation error in the values of the input and output voltages an appropriate choice is the discrete Lebesgue-space l_∞ . In addition this choice allows an easy implementation of the switched controller. In the given experimental setup the discrete plant P is given by, for $k \in \mathbb{N}$,

$$\begin{aligned} y_1(k) &= ay_1(k-1) + bu_1(k-1), \\ y_1(-k) &= u_1(-k) = 0, \end{aligned}$$

for some $a, b \in \mathbb{R}$. Therefore, in view of Figure 2,

$$y_2(k) - ay_2(k-1) - bu_2(k) = y_0(k) - ay_0(k-1) - bu_0(k-1). \quad (10)$$

The switched controller “knows” the values of $y_2(k)$, $y_2(k-1)$ and $u_2(k-1)$. An obvious disturbance estimation is to choose three values for $y_0(k)$, $y_0(k-1)$ and $u_0(k-1)$ such that (10) holds and the norm is minimum, i.e.

$$d(k) := \operatorname{argmin}_{d \in \mathbb{R}^3} \{ \|d\|_\infty \mid y_2(k) - ay_2(k-1) - bu_2(k) = (1, -a, -b)d \}.$$

It is easy to see that the minimum $d_{\min} = (d_1, d_2, d_3)$ has the property $|d_1| = |d_2| = |d_3|$ and can therefore be calculated very easily. This controller is implemented in the file `control1.c`.

4.2 Large uncertainties in the parameters

It will be assumed that there are four possible system descriptions, i.e. $\mathcal{P} = \{1, 2, 3, 4\}$, of the form, for $p \in \mathcal{P}$,

$$y_1(k) = a_p y_1(k-1) + b_p u_1(k-1)$$

with

$$\begin{aligned} a_1 &= 1.01, & a_2 &= 4, & a_3 &= a_1, & a_4 &= a_2, \\ b_1 &= 1.96, & b_2 &= b_1, & b_3 &= -b_1, & b_4 &= -b_1. \end{aligned}$$

Note that for the given nominal system $p^* = 3$. It is easy to see that each two of these plants are not simultaneously stabilisable with a single proportional controller.

The data of two experimental runs is shown in Figure 8. The qualitative behaviour is the same in both cases: At the first step the switching strategy does not have any knowledge, therefore an arbitrary candidate controller is chosen. Since this controller does not fit to the real plant, the output gets larger. Already in the second step the switching strategy chooses the right controller, based on the observed input and output signals. In the case of the significant initial disturbance the error after the second step is fairly large. This error seems to result from parameter uncertainties in the “real” plant model. Furthermore it is noticeable that the steady state error is not zero. The reason for this might be an offset error in the voltage measurement as well as in the control input voltage.

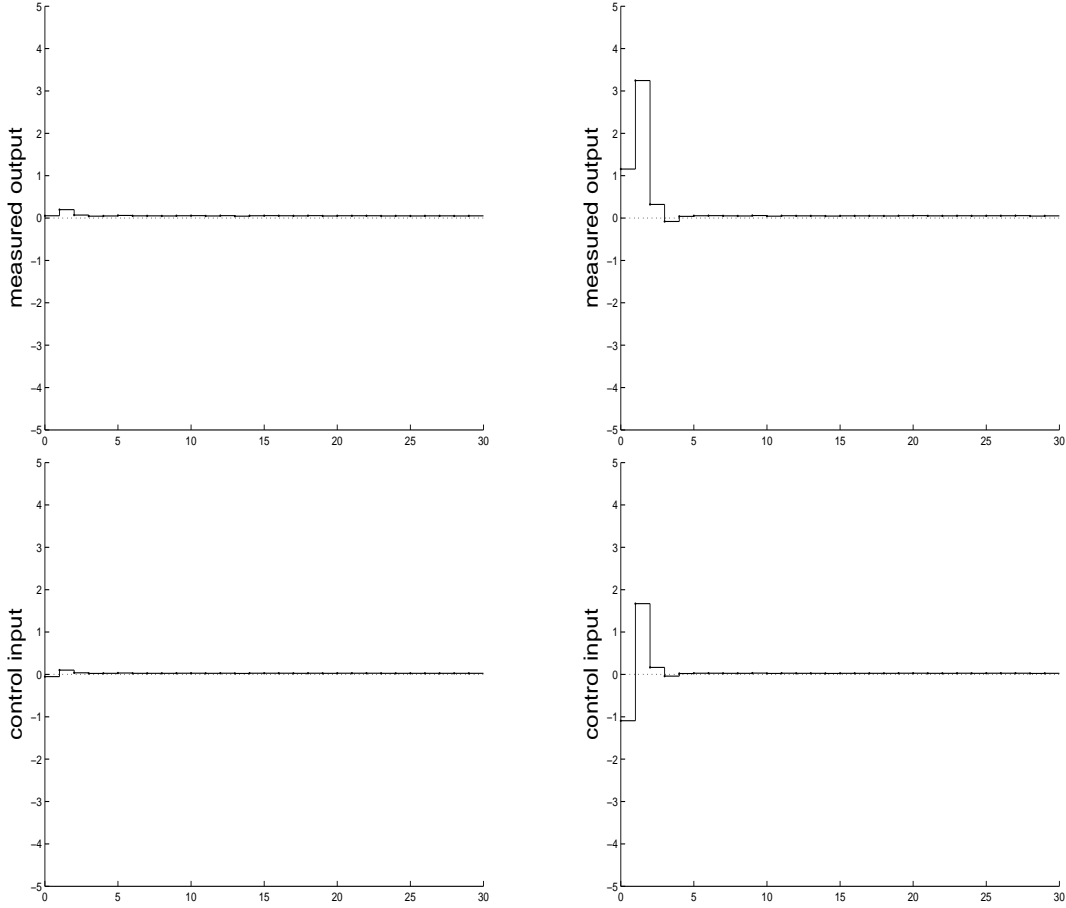


Figure 8: Measured output signal and control input signal, on the left with a small initial disturbance and on the right with a significant initial disturbance.

4.3 Small uncertainties in the parameters

It will be assumed now that $\mathcal{P} = \{1, 2, \dots, 9\}$ and that there are three different values for α and β in the transfer function $G(s) = \beta/(s - \alpha)$, which results in

$$\begin{aligned}
 a_1 &= 1.005, \quad a_2 = 1.01, \quad a_3 = 1.02, \quad a_4 = a_7 = a_1, \quad a_5 = a_8 = a_2, \quad a_6 = a_9 = a_3, \\
 b_1 &= -1.90, \quad b_2 = -1.91, \quad b_3 = -1.92, \quad b_4 = -1.96, \quad b_5 = -1.96, \quad b_6 = -1.97, \quad b_7 = -2.01, \\
 b_8 &= -2.01, \quad b_9 = -2.02.
 \end{aligned}$$

Note that the nominal plant is P_5 . The data of two experimental runs is shown in Figure 9. The corresponding switching signal is in both cases the same and is illustrated in Figure 10. Note that the switching signal was produced out of the the input and output signals. This is possible since the implemented controller only has this information available and it is therefore possible to reproduce the behaviour. It is remarkable, that the switching strategy chooses plant P_3 instead of plant P_5 . An implication of this observation is, that the real plant's model P_5 is less accurate then the alternative (artificial) model P_3 .



Figure 9: Measured output signal and control input signal, on the left with a small initial disturbance and on the right with a significant initial disturbance.

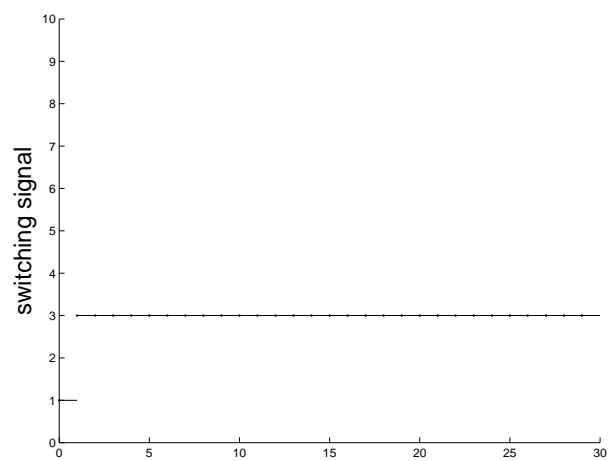


Figure 10: Switching signal.

4.4 Frequency response test

Although it is possible to describe the system precisely as in Sub-Section 3.1 it is not clear whether the parts of the circuit all have idealistic behaviours. It is still assumed that the system behaviour is approximated by a transfer function of the form

$$G(s) = \frac{-\beta}{s - \alpha}.$$

The parameters α and β , which corresponds to the parameters A and B in Sub-Section 3.1, can be experimentally obtained by a frequency response test. Since the system is unstable a direct frequency response is not possible and therefore the system must be stabilized. This is done by a simple proportional feedback controller, which is implemented with two additional operational amplifiers (see Figure 11). The gain of the proportional controller was 0.99. The frequency response for the resulting new system is

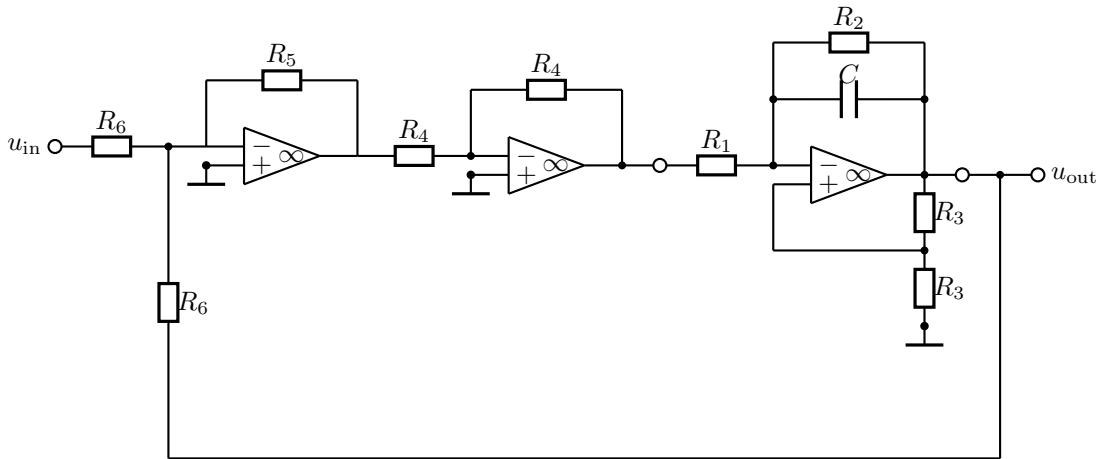


Figure 11: Stabilized system for frequency response.

illustrated in Figure 12.

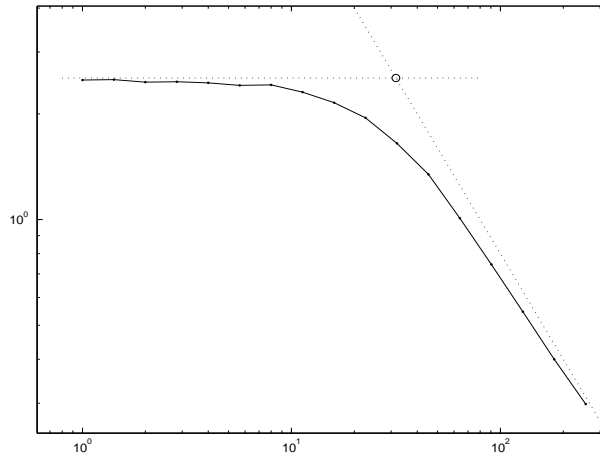


Figure 12: Frequency response of the stabilized system. Both axis are logarithmic.

It is easy to derive the parameters of the unstable system, which are

$$\alpha = 23.7 \quad \text{and} \quad \beta = 808.$$

These values are quite different from the calculated ones, $\alpha = 9.67$ and $\beta = 1954$ (Sub-Section 3.1). The value of α is very sensitive to small changes in the values of the resistors, therefore this difference is not surprising. The dramatic difference in the value for β is not obvious. The author hasn't got a satisfying

explanation but the operational amplifier might consist of capacitor-like elements, which have a capacity of similar size as the capacitor in the circuit. For the discretisation this yields values of

$$a = 1.024 \quad \text{and} \quad b = 0.82.$$

This explains the switching behaviour of Sub-Section 4.3, since plant 3 is the "nearest" plant to the real plant.

5 Conclusion

It was possible to show with experiments that the theoretical results for switched control could be implemented in the "real world". The limits of the switched controller were also visible: The first step will in most cases yield an increasing error since the switched controller has no knowledge about the system at the first step.

6 Acknowledgements

I thank Mark French for his support, we had many fruitful discussion and it was fun working together. Thanks to Nagendra Mandalaju who helped me a lot with experimental questions. I would also like to thank Bernd Bandemer who had the idea for the simple unstable circuit, which was used to do the experiments. I am glad that the University of Southampton partly funded my stay here in Southampton.

References

- [1] K. Åström and B. Wittenmark, *Computer-Controlled Systems*, 3rd ed., ser. Prentice Hall Information and System Sciences Series, T. Kailath, Ed. New Jersey: Prentice Hall, 1997.
- [2] T. Georgiou and M. Smith, “Robustness analysis of nonlinear feedback systems: An input-output approach,” *IEEE Transactions on Automatic Control*, vol. 42, no. 9, pp. 1200–1221, September 1997.
- [3] C.-H. Tsai, “Pctimer: Millisecond resolution timing with djgpp v2 and dpmi,” online: <http://technology.chtsai.org/pctimer/>, 1998.

A Source codes of the experiment software

A.1 ciodas08.h

```
1 /*
2  Routines and makros for the ISA-card CIO-DAS08/JR-A0
3  (or the CIO-DAS08/JR with the additional chip
4  CIO-DUAL-DAC).
5  All routines which do not relate to DA conversion could
6  also be used for the CIO-DAS08/JR.
7  02/02/2005 written by Stephan Trenn.
8 */
9
10 #define BASE 0x0300
11
12 #define CL_IN 0
13 #define CL_OUT 0
14 /* This two constants specify which channels are used for the
15 closed-loop (CL) control. CL_IN is the channel number of the
16 analog input and CL_OUT is the channel number of the analog
17 output */
18
19 #define MAX_IN 5
20 #define MIN_IN -5
21 #define MAX_OUT 5
22 #define MIN_OUT -5
23 /* These values describe the range (in V) of the ADC (*_IN) and of the
24 DAC (*_OUT) */
25
26 #define BITRANGE 0x0FFF
27 /* Twelve bits for AD and DA-conversion, i.e. the bit range is 0 to
28 4095 = 0FFFh */
29
30 #define MAX_SAMPLING_RATE 40000
31 /* The conversion time is 25e-6 seconds which implies a maximal sampling
32 rate of 40000 Hz */
33
34 #define START_ADC(channel) outportb(BASE+2,channel&8);outportb(BASE+1,0)
35 /* This macro starts the analog-digital-conversion from the specified
36 channel */
37
38 double get_voltage(void);
39 /* This function should be called after the START_ADC-Makro. It returns
40 the measured value as soon as it is available. */
41
42 void set_voltage(unsigned short channel, double voltage);
43 /* The analog output on the specified channel will set to the
44 specified voltage. */
45
46 int round(double x);
47 /* C does not provide a build-in round()-function, but we need one */
```

A.2 ciodas08.c

```
1 /*
2  Routines and makros for the ISA-card CIO-DAS08/JR-A0
3  (or the CIO-DAS08/JR with the additional chip
4  CIO-DUAL-DAC).
```

```

5     02/02/2005 written by Stephan Trenn.
6 */
7
8 #include "CIODAS08.h"
9
10 #include <pc.h>
11 /* Using: inportb (...) */
12
13 /*****
14  * Implementation of ciodas08.h :
15  *****/
16
17 #define ADC_FINISHED ((inportb(BASE+2)&128) == 0)
18 /* This macro is true if the AD conversion is finished and
19    false if the AD conversion is still in progress */
20
21 int round(double x)
22 {
23     int value= floor(x);
24     if (x-value >= 0.5) {value++;};
25     return value;
26 }
27
28 double get_voltage(void)
29 {
30     unsigned short  Bits0_7, Bits8_11;
31     unsigned int  BitValue;
32
33     do {} while (!ADC_FINISHED);
34     Bits8_11 = inportb(BASE);
35     Bits0_7 = inportb(BASE+1);
36     BitValue = (((unsigned int) Bits0_7) << 4) + (Bits8_11 >> 4);
37
38     return ( ( (double) BitValue ) / BITRANGE ) * (MAX_IN - MIN_IN) + MIN_IN;
39 }
40
41 void set_voltage(unsigned short channel, double voltage)
42 {
43     unsigned int  bitValue;
44     unsigned short  bit0_7, bit8_11;
45
46     if (voltage < MIN_OUT) {voltage = MIN_OUT;};
47     if (voltage > MAX_OUT) {voltage = MAX_OUT;};
48
49     bitValue = round( (voltage-MIN_OUT)/((double) (MAX_OUT-MIN_OUT)) * BITRANGE);
50     bit0_7 = bitValue & 0x00FF ;
51     bit8_11 = bitValue >> 8;
52     channel = (channel & 1) << 1;
53
54     outportb( BASE+4+channel , bit0_7 );
55     outportb( BASE+5+channel , bit8_11 );
56
57     inportb( BASE+3 ); //starts DA conversion
58 }

```

A.3 int-hdl.h

```

1  /*
2   Routines for handling the timer interrupt
3   01/02/2005 written by Stephan Trenn
4   Derived from PCTIMER 1.4 by Chih-Hao Tsai
5  */
6  #define IRQ0    0x8
7
8  #define PIT0    0x40
9  #define PIT1    0x41
10 #define PIT2    0x42
11 #define PITMODE 0x43
12 #define PIT_CHANGE_FREQUENCY 0x36
13 #define PIT_CONST 1193180
14 #define PIT0_HZ 18.2067597
15
16 #define ADC_READY 0
17 #define ADC_STARTED 1
18
19 extern unsigned char Flag_ADC;
20 /* The variable Flag_ADC can have the two values ADC_READY and
21 ADC_STARTED. The Timer ISR will start the AD conversion and will
22 set this variable to ADC_STARTED. The main program should poll
23 the value of Flag_ADC and should then set it back to ADC_READY. */
24
25 extern unsigned int Missed_Count;
26 /* The number of interrupts which were missed by the main program.
27 It is count how often the variable Flag_ADC is not set back to
28 ADC_READY. */
29
30 double get_real_Hz(double Hz);
31 /* Since the programmable timer interrupt (PIT) can produce only
32 a discrete range of frequency, the real frequency at which the
33 ISR is called is different. The formular is
34  $real\_Hz = PIT\_CONST / round(PIT\_CONST / Hz)$ . */
35
36 void setup_timer_handles(double Hz);
37 /* This function installs own timer interrupt handles for real and
38 protected mode.
39 The parameter Hz is the frequency at which own ISR is called.
40 The old ISR is still called at the normal rate of approximately
41 18.2 Hz. The exact value is
42  $real\_Hz / (round(real\_Hz / PIT0\_HZ))$ ,
43 where  $real\_Hz = get\_real\_Hz(Hz)$ . */
44
45 void restore_timer_handles(void);
46 /* Sets the timer at normal speed and restore old ISRs.
47 You should always call restore_timer_handles() before
48 exiting your program, otherwise the system will crash. */
49
50
51
52 /*The following functions are only for internal use and should not
53 be called from the outside
54
55 void PM_ISR(void)
56 void lock_PM_ISR(void)
57

```

```

58 void RM_ISR(void)
59 void lock_RM_ISR(void)
60 */

```

A.4 int-hdl.c

```

1  /*
2   Routines for handling the timer interrupt
3   01/02/2005 written by Stephan Trenn
4   Derived from PCTIMER 1.4 by Chih-Hao Tsai
5  */
6
7  #include "int-hdl.h"
8
9  #include "CIODAS08.h"
10 /* Using: CL_IN
11          STARTADC(channel) */
12
13 #include <dpmi.h>
14 /* Using: _go32_dpmi_registers ,
15          _go32_dpmi_seginfo ,
16          _go32_dpmi_lock_code (...),
17          _go32_dpmi_lock_data (...),
18          _go32_dpmi_get_protected_mode_interrupt_vector (...),
19          _go32_my_cs (),
20          _go32_dpmi_chain_protected_mode_interrupt_vector (...),
21          _go32_dpmi_get_real_mode_interrupt_vector (...),
22          _go32_dpmi_allocate_real_mode_callback_iret (...),
23          _go32_dpmi_set_real_mode_interrupt_vector (...),
24          _go32_dpmi_set_protected_mode_interrupt_vector (...),
25          _go32_dpmi_free_real_mode_callback (... ) */
26
27 #include <pc.h>
28 /* Using: outportb (...) */
29
30 #include <dos.h>
31 /* Using: enable (),
32          disable ()
33          delay (...) */
34
35
36 #include <string.h>
37 /* Using: memset (...) */
38
39 /*
40  Implementation of int-hdl.h :
41  */
42
43 #define ISR_NOT_INSTALLED 0
44 #define ISR_INSTALLED 1
45
46 #define PM_FAILURE 0
47 #define PM_CHAIN 1
48 #define PM_EOI 2
49
50 #define EOI outportb(0x20,0x20)
51

```

```

52 unsigned char Flag_ADC = ADCREADY;
53 unsigned int Missed_Count = 0;
54
55 static _go32_dpmi_registers ISR_r, r;
56 static _go32_dpmi_seginfo PM_old_ISR, PM_new_ISR, RM_old_ISR, RM_new_ISR;
57 static unsigned int ISR_counter, ISR_counter_reset;
58
59 static unsigned char Flag_INSTALLED = ISR_NOT_INSTALLED;
60 static unsigned char Flag_PM = PM_FAILURE;
61
62 void PM_ISR()
63 {
64     disable();
65
66     START_ADC(CL_IN);
67
68     if (Flag_ADC == ADC_STARTED)
69         Missed_Count++;
70     else
71         Flag_ADC = ADC_STARTED;
72
73     ISR_counter++;
74
75     Flag_PM = PM_FAILURE;
76
77     if (ISR_counter == ISR_counter_reset)
78     {
79         Flag_PM = PM_CHAIN;
80         ISR_counter = 0;
81         enable();
82     }
83     else
84     {
85         Flag_PM = PM_EOI;
86         EOI;
87         enable();
88     }
89 };
90
91 void lock_PM_ISR()
92 {
93     _go32_dpmi_lock_code( PM_ISR , (unsigned long) (lock_PM_ISR - PM_ISR) );
94 };
95
96 void RM_ISR()
97 {
98     disable();
99
100    if (Flag_PM == PM_FAILURE)
101    {
102        START_ADC(CL_IN);
103
104        if (Flag_ADC == ADC_STARTED)
105            Missed_Count++;
106        else
107            Flag_ADC = ADC_STARTED;
108

```

```

109     ISR_counter++;
110 }
111
112 if ( (ISR_counter == ISR_counter_reset) || (Flag_PM == PMCHAIN) )
113 {
114     Flag_PM = PMFAILURE;
115     ISR_counter = 0;
116     memset( &r , 0 , sizeof(r) );
117     r.x.cs = RM_old_ISR.rm_segment;
118     r.x.ip = RM_old_ISR.rm_offset;
119     r.x.ss = r.x.sp = 0;
120     _go32_dpmi_simulate_fcall_iret( &r );
121     enable();
122 }
123 else
124 {
125     Flag_PM = PMFAILURE;
126     EOI;
127     enable();
128 }
129 }
130
131 void lock_RM_ISR()
132 {
133     _go32_dpmi_lock_code( RM_ISR , (unsigned long) (lock_RM_ISR - RM_ISR) );
134 }
135
136 double get_real_Hz(double Hz)
137 {
138     unsigned int pit0_value = round(PIT_CONST / Hz);
139     if (pit0_value > 0x00FFFF) pit0_value = 0x010000;
140
141     return ((double) PIT_CONST) / pit0_value;
142 }
143
144 void setup_timer_handles(double Hz)
145 {
146     unsigned int pit0_value;
147     unsigned char pit0_set;
148     double real_Hz;
149     if (Flag_INSTALLED == ISR_INSTALLED)
150     {
151         restore_timer_handles();
152     }
153     Missed_Count=0;
154
155     // Doing floating point operations outside disable() ... enable():
156     pit0_value = round(PIT_CONST / Hz);
157     if (pit0_value > 0x00FFFF) pit0_value = 0x010000;
158
159     real_Hz = ((double) PIT_CONST) / pit0_value;
160
161     ISR_counter_reset = round( real_Hz / PIT0_HZ );
162
163     disable();
164
165     lock_PM_ISR();

```

```

166 lock_RM_ISR ();
167
168 _go32_dpmi_lock_data( &Flag_ADC , sizeof(Flag_ADC) );
169 _go32_dpmi_lock_data( &Flag_PM , sizeof(Flag_PM) );
170 _go32_dpmi_lock_data( &ISR_counter , sizeof(ISR_counter) );
171 _go32_dpmi_lock_data( &ISR_counter_reset , sizeof(ISR_counter_reset) );
172 _go32_dpmi_lock_data( &Missed_Count , sizeof(Missed_Count) );
173
174 _go32_dpmi_get_protected_mode_interrupt_vector( IRQ0 , &PM_old_ISR );
175 PM_new_ISR.pm_offset = (int) PM_ISR;
176 PM_new_ISR.pm_selector = _go32_my_cs();
177 _go32_dpmi_chain_protected_mode_interrupt_vector( IRQ0 , &PM_new_ISR );
178
179 _go32_dpmi_get_real_mode_interrupt_vector( IRQ0 , &RM_old_ISR );
180 RM_new_ISR.pm_offset = (int) RM_ISR;
181 _go32_dpmi_allocate_real_mode_callback_iret( &RM_new_ISR , &ISR_r );
182 _go32_dpmi_set_real_mode_interrupt_vector( IRQ0 , &RM_new_ISR );
183
184 outportb( PITMODE , PIT_CHANGE_FREQUENCY );
185 pit0_set = (pit0_value & 0x00FF);
186 outportb( PIT0 , pit0_set );
187 pit0_set = ( (pit0_value >> 8) & 0x00FF);
188 outportb( PIT0 , pit0_set );
189
190 Flag_INSTALLED = ISR_INSTALLED;
191 ISR_counter = 0;
192
193 enable ();
194
195 }
196
197 void restore_timer_handles(void)
198 {
199     if (Flag_INSTALLED == ISR_INSTALLED)
200     {
201         disable ();
202
203         outportb( PITMODE , PIT_CHANGE_FREQUENCY );
204         outportb( PIT0 , 0x00);
205         outportb( PIT0 , 0x00);
206
207         _go32_dpmi_set_protected_mode_interrupt_vector( IRQ0 , &PM_old_ISR );
208
209         _go32_dpmi_set_real_mode_interrupt_vector( IRQ0 , &RM_old_ISR );
210         _go32_dpmi_free_real_mode_callback( &RM_new_ISR );
211
212         enable ();
213
214         Flag_INSTALLED = ISR_NOT_INSTALLED;
215     }
216 }

```

A.5 controlt.h

```

1 /*
2  This controller is just for test purpose. It is assumed, that
3  the DA-output is directly connected with the AD-input. This

```

```

4     controller can be used to test, how far away the DA value is
5     from the measured AD value. Since the system is in this case
6     only a simple cable the system inputs are the DA outputs and
7     the system outputs are the AD inputs.
8 */
9
10 #define MAX_SAMPLES_t 5000
11 /* The maximum number of samples. If needed this number could be
12    increased. Note that at the programme start the whole amount
13    of memory will be allocated (twice, one for the input signals
14    and one for the output signals), even if not so many samples
15    are needed. */
16
17 void reset_controller_t();
18 /* Sets the internal states of the controller to zero, i.e.
19    all previous output-signals of the system (input-signals
20    to the controller) will be assumed to be zero. */
21
22 double control_t(double current_output);
23 /* This is the main function and should be called sequently. The
24    controller will have an internal state, which changes with every
25    call of the function control(). In particular a twice call of
26    control() will in general lead to different results. */
27
28 unsigned int get_all_inputs_t(double **input_ptr);
29 /* This function will return all input values (output of the
30    controller) via a pointer to the position of the values.
31    The return value is the number of values. */
32
33 unsigned int get_all_outputs_t(double **output_ptr);
34 /* This function will return all output values (input of the
35    controller) via a pointer to the position of the values.
36    The return value is the number of values. */
37
38 /* Note: The controller should be implemented in such a way that
39    get_all_inputs(&ptr1) == get_all_outputs(&ptr2),
40    i.e. the two functions should return the same number of values,
41    if the function control() is not called between. */
42
43 extern double current_input;
44 /* This variable stores the current input used by the test-
45    programme. */

```

A.6 controll.c

```

1 #include "controll.h"
2
3
4 /******
5  * Implementation of controll.h :
6  *****/
7
8 static double inputs[MAX_SAMPLES_t], outputs[MAX_SAMPLES_t];
9 static unsigned int sample_count = 0;
10
11 double current_input;
12

```

```

13 void reset_controller_t()
14 {
15     sample_count = 0;
16
17 }
18
19 double control_t(double current_output)
20 {
21     outputs[sample_count] = current_output;
22     inputs[sample_count] = current_input;
23
24     return inputs[sample_count++];
25 }
26
27 unsigned int get_all_inputs_t(double **input_ptr)
28 {
29     *input_ptr = inputs;
30     return sample_count;
31 }
32
33 unsigned int get_all_outputs_t(double **output_ptr)
34 {
35     *output_ptr = outputs;
36     return sample_count;
37 }

```

A.7 con-exp.c

Since most of the source code of `con-exp.c` deals with user menus and user input not the whole source code is given.

```

1  /* Main programme. Written by Stephan Trenn, 04/02/2005.
2
3  The purpose of this programme is to do experiments with various
4  controllers in the "real world". The real world will be a given
5  circuit which is connected via an analog-digital converter (ADC)
6  with the computer. The computer is connected via a digital-analog
7  converter (DAC) with the circuit and can therefore control the
8  circuit.
9
10 For the AD and DA conversion the ISA-card CIO-DAS08/JR-AO from
11 "Measurement Computing Corporation" is used. All relevant
12 routines for the usage of this card are described in CIODAS08.h
13 and implemented in CIODAS08.c.
14
15 If the sampling period is constant then the continuous system
16 (of the circuit) can exactly be described as a discrete system
17 (see K.J. Astrom and B. Wittenmark, 1997). Since the card
18 CIO-DAS08/JR-AO does not provide a periodic sampling it is
19 necessary to re-program the PC timer (PIT 8253) and write an own
20 interrupt handler (or interrupt service routine - ISR). This
21 routines are described in INT-HDL.h and implemented in INT-HDL.c.
22
23 It is easy to use different controllers. Just write an own controller
24 (read CONTROL.h) and include the corresponding header file here:
25 */
26
27 #include "control.h"

```

```

28
29 /*****
30
31 #include "ciodas08.h"
32 #include "int-hdl.h"
33 #include "controлт.h"
34
35 #include <conio.h>
36 /* Using: clrscr()
37          COLORS
38          textcolor(...)
39          textbackground(...) */
40
41 #include <stdlib.h>
42 /* Using: atof(...) */
43
44 #include <stdio.h>
45 /* Using: FILE
46          fopen(...)
47          fclose(...) */
48
49 #include <grx20.h>
50 /* Using: GrSetDriver(...)
51          GrSetMode(...)
52          _GR_graphicsModes
53          GrTextOptions */
54
55 #include <grxkeys.h>
56 /* Using: GrKeyRead() */
57
58 #include <string.h>
59 /* Using: strlen(...) */
60
61 #define TRUE 1
62 #define FALSE 0
63
64 #define EXIT 0
65 #define GOON 1
66
67 unsigned char exp_poss = FALSE;
68 unsigned char save_poss = FALSE;
69 unsigned char data_saved = FALSE;
70
71 double current_period;
72 double current_running_time;
73
74 int show_period_time()
75 {
76     ...
77
78     if (getch()==27) return EXIT;
79     else return GOON;
80 }
81
82 void myscanf(int max_length, char *string)
83 {
84     int current_pos = 0;

```

```

85     int current_length = 0;
86     int i;
87     char key;
88     int xpos , ypos;
89
90     xpos=wherex ();
91     ypos=wherey ();
92
93     string [0]=0;
94
95     unsigned char finished = FALSE;
96     do
97     {
98         gotoxy(xpos , ypos);
99         textcolor(WHITE);
100        textbackground(BLUE);
101        for (i=0;i<max_length;i++)
102            cprintf(" ");
103        gotoxy(xpos , ypos);
104        cprintf(string);
105        gotoxy(xpos+current_pos , ypos);
106        textbackground(BLACK);
107
108        key = getch();
109        switch (key)
110        {
111            case 8: if (current_pos > 0)
112                {
113                    current_pos --;
114                    current_length --;
115                    for (i=current_pos ; string [i] != 0; i++)
116                        string [i] = string [i + 1];
117                }; break;
118            case 127: if (current_length > current_pos)
119                {
120                    current_length --;
121                    for (i=current_pos ; string [i] != 0; i++)
122                        string [i] = string [i + 1];
123                }; break;
124            case 27: string [0]=0; finished=TRUE; break;
125
126            case 13: finished = TRUE; break;
127            default : if ((key > 31) && (current_length < max_length))
128                {
129                    for (i=max_length ; i > current_pos ; i--)
130                        string [i] = string [i - 1];
131                    string [current_pos] = key;
132                    current_pos ++;
133                    current_length ++;
134                }; break;
135        }
136    } while (finished == FALSE);
137 }
138
139 int change_period ()
140 {
141     char input_string [20];

```

```

142     int posx, posy;
143     double value;
144
145     ...
146
147     myscanf(19, input_string);
148
149
150     if (input_string[0]==0) return(EXIT);
151     if ( (input_string[1]==0) && (PERIOD != 0) &&
152         ( (input_string[0]=='C') || (input_string[0]=='c') )
153         )
154     {
155         current_period = 1/get_real_Hz(1/PERIOD);
156         if (current_running_time>0) exp_poss=TRUE;
157         return(EXIT);
158     }
159     value = atof(input_string)/1000;
160
161     if (value > 0) value=1/get_real_Hz(1/value);
162
163     if (value < 1.0/MAX_SAMPLING_RATE)
164     {
165         ...
166         getch();
167         return(GOON);
168     }
169     else
170     {
171         current_period = value;
172         if (current_running_time/current_period > MAX_SAMPLES)
173             current_running_time = current_period * MAX_SAMPLES;
174         if (current_running_time>0) exp_poss=TRUE;
175         return(EXIT);
176     };
177 }
178
179 int change_Hz ()
180 {
181     char input_string [20];
182     int posx, posy;
183     double value;
184
185     ...
186
187     myscanf(19, input_string);
188
189
190     if (input_string[0]==0) return(EXIT);
191     if ( (input_string[1]==0) && (PERIOD != 0) &&
192         ( (input_string[0]=='C') || (input_string[0]=='c') )
193         )
194     {
195         current_period = 1/get_real_Hz(1/PERIOD);
196         if (current_running_time>0) exp_poss=TRUE;
197         return(EXIT);
198     }

```

```

199     value = atof(input_string);
200     if (value > 0) value=1/get_real_Hz(value);
201
202     if (value <= 1.0/MAX_SAMPLINGRATE)
203     {
204         ...
205         getch();
206         return(GOON);
207     }
208     else
209     {
210         current_period = value;
211         if (current_running_time/current_period > MAX_SAMPLES)
212             current_running_time = current_period * MAX_SAMPLES;
213         if (current_running_time>0) exp_poss=TRUE;
214         return(EXIT);
215     }
216
217 }
218
219 int change_running_time()
220 {
221     char input_string[20];
222     int posx, posy;
223     double value;
224
225     ...
226
227     myscanf(19, input_string);
228
229
230     if (input_string[0]==0) return(EXIT);
231     if ( (input_string[1]==0) && (current_period > 0) &&
232         ( (input_string[0]=='M') || (input_string[0]=='m') )
233         )
234     {
235         current_running_time = current_period*MAX_SAMPLES;
236         exp_poss=TRUE;
237         return(EXIT);
238     }
239     value = atof(input_string);
240
241     if ((value <= 0) || (value>current_period*MAX_SAMPLES))
242     {
243         ...
244         getch();
245         return(GOON);
246     }
247     else
248     {
249         current_running_time = value;
250         exp_poss=TRUE;
251         return(EXIT);
252     }
253
254 }
255

```

```

256 int menu_change()
257 {
258     ...
259
260     char key = getch();
261     switch (key)
262     {
263         case '1': do {} while (change_period()==GOON);
264                 return(GOON);
265
266         case '2': do {} while (change_Hz()==GOON);
267                 return(GOON);
268
269         case '3': if (current_period > 0)
270                 do {} while (change_running_time()==GOON);
271                 return(GOON);
272
273         case 27 : return(EXIT);
274
275         default : return(GOON);
276     }
277
278     return EXIT;
279 }
280
281 void do_control()
282 {
283     unsigned int count;
284     unsigned int i=0;
285     double system_input;
286     double system_output;
287
288     ...
289
290     count = round(current_running_time / current_period);
291     if (count>MAX_SAMPLES)
292     {
293         count=MAX_SAMPLES;
294         current_running_time = count * current_period ;
295     }
296
297     reset_controller ();
298
299     setup_timer_handles(1/current_period);
300
301     do
302     {
303         do {} while (Flag_ADC != ADC_STARTED);
304         system_output = get_voltage();
305         system_input = control(system_output);
306         set_voltage(CL_OUT, system_input);
307         Flag_ADC = ADC_READY;
308
309         i++;
310     }
311     while ( i < count );
312

```

```

313     restore_timer_handles ();
314
315     ...
316
317     getch ();
318
319 }
320
321 void start_experiment ()
322 {
323     char key;
324
325     ...
326
327     do
328     {
329         key=getch ();
330     }
331     while ((key!=13) && (key!=27));
332
333     if (key==13)
334     {
335         do_control ();
336         save_pong = TRUE;
337         data_saved = FALSE;
338     }
339 }
340
341 void show_data ()
342 {
343     char *output_label = "Measured_Output";
344     char *input_label = "Input_signal";
345     char *presskey = "Press_any_key_to_return_to_main_menu";
346     double rel_bound = 1.0/20.0;
347     GrTextOption grt;
348     int xmax, ymax, outputx, outputy, outputw, outputh;
349     int inputx, inputy, inputw, inputh;
350     double *inputs, *outputs;
351     int sample_number, i;
352     int input_points[2*MAX_SAMPLES][2], output_points[2*MAX_SAMPLES][2];
353     GrSetDriver("VESA_gw_640_gh_480_nc_16");
354     GrSetMode( GR_default_graphics );
355     GrColor *egacolors;
356
357     ...
358
359     GrKeyType key = GrKeyRead ();
360
361     GrSetMode( GR_default_text );
362
363
364 }
365
366 int save_data ()
367 {
368     char input_string [9];
369     char file_name1 [13], file_name2 [13];

```

```

370     int posx, posy;
371     FILE *input_file, *output_file;
372     double *input_data, *output_data;
373     unsigned int sample_number, i;
374     unsigned char Write_OK = TRUE;
375
376     ...
377
378     myscanf(8, input_string);
379     strcpy(file_name1, input_string);
380     strcpy(file_name2, input_string);
381
382     strcat(file_name1, ".IDA");
383     strcat(file_name2, ".ODA");
384
385     input_file = fopen(file_name1, "wt");
386     output_file = fopen(file_name2, "wt");
387
388     if ((input_file==NULL) || (output_file==NULL))
389     {
390         ...
391         data_saved = FALSE;
392         getch();
393         return(EXIT);
394     }
395
396     sample_number = get_all_inputs(&input_data);
397     if (get_all_outputs(&output_data) != sample_number)
398     {
399         ...
400         data_saved = FALSE;
401         fclose(input_file);
402         fclose(output_file);
403         getch();
404         return(EXIT);
405     }
406
407     for (i=0; i<sample_number; i++)
408     {
409         if (fprintf(input_file, "%g\n", input_data[i])==0) Write_OK = FALSE;
410         if (fprintf(output_file, "%g\n", output_data[i])==0) Write_OK = FALSE;
411     }
412     fclose(input_file);
413     fclose(output_file);
414
415     if (Write_OK = TRUE)
416     {
417         ...
418         data_saved = TRUE;
419         getch();
420         return(GOON);
421     }
422     else
423     {
424         ...
425         data_saved = TRUE;
426         getch();

```

```

427         return(EXIT);
428     }
429 }
430
431 int menu_not_saved()
432 {
433     char key;
434
435     ...
436
437     do
438     {
439         key=getch();
440     }
441     while ((key!=13)&&(key!='s')&&(key!='S')&&(key!=27));
442
443     if (key==13) return GOON;
444     if (key==27) return EXIT;
445     else
446     {
447         return(save_data());
448     }
449 }
450
451 void card_test()
452 {
453     double *inputs;
454     double *outputs;
455     unsigned int count;
456     unsigned int i;
457     double system_output;
458     double average_error=0;
459     double average_abs_error=0;
460     double max_abs_error=0;
461     char key;
462
463     current_input = ((double) rand())/RANDMAX * (MAX_OUT-MIN_OUT) + MIN_OUT;
464     set_voltage(CL_OUT, current_input);
465     count = round(current_running_time / current_period);
466     if (count>MAX_SAMPLES_t)
467     {
468         count=MAX_SAMPLES_t;
469         current_running_time = count * current_period ;
470     }
471
472     ...
473
474     do
475     {
476         key=getch();
477     }
478     while ((key!=13) && (key!=27));
479
480     if (key==13)
481     {
482         ...
483

```

```

484     reset_controller_t ();
485
486     setup_timer_handles (1/current_period);
487
488     i=0;
489     do
490     {
491         do {} while (Flag_ADC != ADC.STARTED);
492         system_output = get_voltage ();
493         current_input=control_t (system_output);
494         Flag_ADC = ADC.READY;
495
496         current_input =
497             ((double) rand())/RANDMAX * (MAX.OUT-MIN.OUT) + MIN.OUT;
498         set_voltage (CL.OUT, current_input);
499
500         i++;
501     }
502     while ( i < count );
503
504     restore_timer_handles ();
505
506
507     unsigned int count_inputs = get_all_inputs_t (&inputs);
508     unsigned int count_outputs = get_all_outputs_t (&outputs);
509     if (count_inputs != count)
510     {
511         ...
512         key=getch ();
513     }
514     else
515     {
516         ...
517
518         double diff=0;
519         for (i=0; i<count; i++)
520         {
521             diff=(outputs[i]-inputs[i]);
522             average_error += diff;
523             average_abs_error += fabs(diff);
524             max_abs_error =
525                 (fabs(diff)>max_abs_error) ? fabs(diff) : max_abs_error;
526         }
527         average_error /= count;
528         average_abs_error /= count;
529
530         ...
531
532         key=getch ();
533     }
534 }
535 }
536
537 void about_screen ()
538 {
539     ...
540

```

```

541     getch();
542 }
543
544 int menu_main()
545 {
546
547     ...
548
549     char key = getch();
550     switch (key)
551     {
552         case '1': do {} while (show_period_time()==GOON);
553                 return(GOON);
554
555         case '2': do {} while (menu_change()==GOON);
556                 return(GOON);
557
558         case '3': if (exp_poss==TRUE)
559                   if (!( (save_poss==TRUE) && (data_saved == FALSE) ))
560                   {
561                       start_experiment();
562                   }
563                   else
564                   {
565                       if (menu_not_saved()==GOON) start_experiment();
566                   };
567                 return(GOON);
568
569         case '4': if (save_poss==TRUE) show_data();
570                 return(GOON);
571
572         case '5': if (save_poss==TRUE) save_data();
573                 return(GOON);
574
575         case '6': if (exp_poss==TRUE) card_test();
576                 return(GOON);
577
578         case 'a':
579         case 'A': about_screen();
580                 return(GOON);
581
582         case 27 : return(EXIT);
583
584         default : return(GOON);
585     }
586
587     return EXIT;
588 }
589
590
591 void init ()
592 {
593     current_period = (PERIOD>0) ? (1/get_real_Hz(1/PERIOD)) : 0;
594     current_running_time = current_period * MAX_SAMPLES;
595     if (current_running_time > 0) exp_poss = TRUE;
596 }
597

```

```

598 int main()
599 {
600     unsigned char todo = GOON;
601     textmode(C80);
602     textcolor(WHITE);
603     textbackground(BLACK);
604     init();
605     do
606     {
607         todo = menu_main();
608         if ( (todo==EXIT) && (save_poss==TRUE) && (data_saved==FALSE) )
609             todo = (menu_not_saved()==GOON) ? EXIT : GOON;
610     }
611     while (todo == GOON);
612     clrscr();
613     textattr(LIGHTGRAY | (BLACK << 4));
614     return 0;
615 }

```

B Source code of the implemented controller

B.1 Template control.h

```

1  /*
2  This header file should be used for all controllers which
3  should be implemented. Just rename it to controlX, where X
4  is a number from 0 to 9 and write an implementation in the
5  file controlX.c There should not exist a file control.c.
6  */
7  #define DESCRIPTION "This_header_file_does_not_belong_to_a_controller"
8  /* Write here a short description of the implemented controller,
9   e.g. "Dead beat control" or "Switched control"
10
11 #define PERIOD 0.001
12 /* Most controllers need to know the sampling period. If the
13    value is 0, then the controller doesn't need to know the
14    period. */
15
16 #define MAXSAMPLES 5000
17 /* The maximum number of samples. If needed this number could be
18    increased. Note that at the programme start the whole amount
19    of memory will be allocated (twice, one for the input signals
20    and one for the output signals), even if not so many samples
21    are needed. */
22
23 void reset_controller();
24 /* Sets the internal states of the controller to zero, i.e.
25    all previous output-signals of the system (input-signals
26    to the controller) will be assumed to be zero. */
27
28 double control(double current_output);
29 /* This is the main function and should be called sequently. The
30    controller will have an internal state, which changes with every
31    call of the function control(). In particular a twice call of
32    control() will in general lead to different results. */
33

```

```

34 unsigned int get_all_inputs(double (double **input_ptr);
35 /* This function will return all input values (output of the
36 controller) via a pointer to the position of the values.
37 The return value is the number of values. */
38
39 unsigned int get_all_outputs(double (double **output_ptr);
40 /* This function will return all output values (input of the
41 controller) via a pointer to the position of the values.
42 The return value is the number of values. */
43
44 /* Note: The controller should be implemented in such a way that
45 get_all_inputs(&ptr1) == get_all_outputs(&ptr2),
46 i.e. the two functions should return the same number of values,
47 if the function control() is not called between. */

```

B.2 controll1.h

```

1 /*
2 Control1 implements a switched controller. It will be
3 assumed that the transfer-function of the system is
4
5 
$$G(s) = \text{BETA}_p / (s - \text{ALPHA}_p)$$

6
7 for  $p$  in some finite set  $P = \{0, \dots, N\}$ 
8 The values of  $\text{BETA}_p$  and  $\text{ALPHA}_p$  are known, but it is not known which
9  $p$  is the "right" one.
10 */
11
12 // Specific definitions for this controller:
13 #define ALPHA {5, 9.67, 20, 5, 9.67, 20, 5, 9.67, 20 }
14 #define BETA {-1900,-1900,-1900,-1954, -1954, -1954, -2000,-2000,-2000}
15 #define PLANTS 9
16
17 #define TRUE 1
18 #define FALSE 0
19
20 // The following definitions are necessary if one wants to use
21 // the dead beat controller at the beginning to start with the
22 // values in [v_min, v_max]
23 #define USE_DEADBEAT TRUE
24 #define REALPLANT 2
25 // number of the "real" plant (starting with 0)
26 #define DEADBEATSAMPLES 10
27 //Number of samples where deadbeat controller is running
28 #define FREESAMPLES 0
29 //Number of additional samples were no controller is running
30
31
32 // General definitions (from control.h):
33 #define DESCRIPTION "Switched_controller"
34
35 #define PERIOD 0.001
36 /* Most controllers need to know the sampling period. If the
37 value is 0, then the controller doesn't need to know the
38 period. */
39
40 #define MAXSAMPLES 10000

```

```

41 /* The maximum number of samples. If needed this number could be
42    increased. Note that at the programme start the whole amount
43    of memory will be allocated (twice, one for the input signals
44    and one for the output signals), even if not so many samples
45    are needed. */
46
47 void reset_controller();
48 /* Sets the internal states of the controller to zero, i.e.
49    all previous output-signals of the system (input-signals
50    to the controller) will be assumed to be zero. */
51
52 double control(double current_output);
53 /* This is the main function and should be called sequently. The
54    controller will have an internal state, which changes with every
55    call of the function control(). In particular a twice call of
56    control() will in general lead to different results. */
57
58 unsigned int get_all_inputs(double **input_ptr);
59 /* This function will return all input values (output of the
60    controller) via a pointer to the position of the values.
61    The return value is the number of values. */
62
63 unsigned int get_all_outputs(double **output_ptr);
64 /* This function will return all output values (input of the
65    controller) via a pointer to the position of the values.
66    The return value is the number of values. */
67
68 /* Note: The controller should be implemented in such a way that
69    get_all_inputs(&ptr1) == get_all_outputs(&ptr2),
70    i.e. the two functions should return the same number of values,
71    if the function control() is not called between.
72    Note furthermore that a change in the returned array will cause
73    a change in the values for the controller as well, since they
74    have the same physical adress. */

```

B.3 controll.c

```

1 #include "control1.h"
2
3 #include <math.h>
4 /* Using: exp(...) */
5
6 /*****
7  * Implementation of control1.h :
8  *****/
9
10 static double inputs[MAX_SAMPLES], outputs[MAX_SAMPLES];
11 static unsigned int sample_count = 0;
12
13 static double alphas[PLANTS] = ALPHA;
14 static double betas[PLANTS] = BETA;
15
16 static double a[PLANTS];
17 static double b[PLANTS];
18
19 static resetted = FALSE;
20

```

```

21 static double D_Norm[PLANTS];
22
23 void reset_controller()
24 {
25     int i;
26
27     sample_count = 0;
28
29     for (i = 0; i < PLANTS ; i++)
30     {
31         a[i] = exp(alphas[i] * PERIOD);
32         b[i] = betas[i] / alphas[i] * ( exp(alphas[i] * PERIOD) - 1 );
33     }
34     resetted = TRUE;
35 };
36
37 #define MIN(a,b) ((a<b)?a:b)
38
39 double find_min_disturb(double m, double n, double c)
40 /* This function solves the following optimization problem
41
42  $min ||x, y, z||$  with  $x + my + nz = c$ 
43
44 In this implementation as norm the maximum norm is chosen,
45 in this case the minimum is attained at  $|x|=|y|=|z|$ .
46 Therefore only the following four cases have to be considered:
47
48 (1)  $x = y = z$ 
49 (2)  $x = -y = z$ 
50 (3)  $x = y = -z$ 
51 (4)  $x = -y = -z$ 
52 */
53 {
54     double x1, x2, x3, x4;
55
56     x1 = fabs(c/(1+m+n));
57     x2 = fabs(c/(1-m+n));
58     x3 = fabs(c/(1+m-n));
59     x4 = fabs(c/(1-m-n));
60
61     return MIN(MIN(x1, x2), MIN(x3, x4));
62 }
63
64
65 double norm(double x1, double x2)
66 /* Maximum norm implemented */
67 {
68     return (x1 > x2) ? x1 : x2;
69 }
70
71
72
73 double find_p_min(int k)
74 {
75     double min_disturb_norm[PLANTS];
76     int i;
77     double current_min = -1;

```

```

78     int current_min_index =0;
79
80     for (i=0;i<PLANTS;i++)
81     {
82         if (k>DEADBEATSAMPLES+FRESAMPLES)
83             min_disturb_norm[i] =
84                 find_min_disturb(-a[i],-b[i],
85                 outputs[k]-a[i]*outputs[k-1]-b[i]*inputs[k-1]);
86         else
87             // in the first step of the switched controller
88             // it doesn't have any past:
89             min_disturb_norm[i]=find_min_disturb(-a[i],-b[i],outputs[k]);
90
91         D_Norm[i] = norm(D_Norm[i],min_disturb_norm[i]);
92
93         if ((D_Norm[i]<current_min) || (current_min<0))
94         {
95
96             current_min = D_Norm[i];
97             current_min_index = i;
98         }
99     }
100
101     return current_min_index;
102 }
103
104 double control(double current_output)
105 {
106     int p_min;
107
108     if (resetted == FALSE) {reset_controller();};
109
110     if (sample_count < MAX_SAMPLES)
111     {
112         outputs[sample_count] = current_output;
113
114         if ((USE_DEADBEAT == TRUE) && (sample_count < DEADBEATSAMPLES))
115
116             inputs[sample_count] = -a[REALPLANT]/b[REALPLANT] * current_output;
117
118         else
119         if((USE_DEADBEAT == TRUE) && (sample_count<DEADBEATSAMPLES+FRESAMPLES))
120
121             inputs[sample_count] = 0;
122
123         else
124         {
125             p_min = find_p_min(sample_count);
126
127             inputs[sample_count] = -a[p_min]/b[p_min] * current_output;
128         }
129         return inputs[sample_count++];
130     }
131     return(0);
132 };
133
134 unsigned int get_all_inputs(double **input_ptr)

```

```
135 {
136     *input_ptr = inputs;
137     return sample_count;
138 };
139
140 unsigned int get_all_outputs(double **output_ptr)
141 {
142     *output_ptr = outputs;
143     return sample_count;
144 };
```