ISSN 0280–5316
ISRN LUTFD2/TFRT--5582--SE

Automatic tuning of a KaMeWa WaterJet Servo

Anders Hansson
Per-Inge Tallberg

Department of Automatic Control
Lund Institute of Technology
September 1997

Department of Automatic Control
Lund Institute of Technology
Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
September 1997

Document Number
ISRN LUTFD2/TFRT--5582--SE

Author(s)
Anders Hansson and Per-Inge Tallberg

Supervisor
Karl Johan Åström, Tore Hägglund
and Peter Mähler(KaMeWa)

Sponsoring organisation

Title and subtitle
Automatic tuning of a KaMeWa WaterJet Servo

Abstract

A water jet system is a propulsion system for vessels. The principle of water jet is to admit water into the hull, then add head to the water via the impeller and then let the water go through a guide vane package and then discharge it at the stern. The reaction force will bring the vessel into motion.

KaMeWa Group, a Vickers company who manufactures water jet propulsion systems, initiated the work. This thesis discuss dynamic modeling of the system and design of a controller for the water jet with automatic tuning. The purpose of the auto-tuning is to get an objective, personal depending tuning of the system. It will also save precious time at the trail trips.

First of all a short description of a commercial KaMeWa water jet system is given. Then the identification of the system is briefly described. The next section deals with the PID-controller and aspects there of. Eventually the automatic tuning is described. The code of the auto-tuner, both C and matlab, is incorporated in the appendix.

Key words
PID, autotuning, waterjet

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
42

Recipient's notes

Security classification

Abstract

A water jet system is a propulsion system for vessels. The principle of water jet is to admit water into the hull, then add head to the water via the impeller and then let the water go through a guide vane package and then discharge it at the stern. The reaction force will bring the vessel into motion. KaMeWa Group, a Vickers company who manufactures water jet propulsion systems, initiated the work. This thesis discuss dynamic modeling of the system and design of a controller for the water jet with automatic tuning. The purpose of the auto-tuning is to get an objective, personal depending tuning of the system. It will also save precious time at the trail trips. First of all a short description of a commercial KaMeWa water jet system is given. Then the identification of the system is briefly described. The next section deals with the PID-controller and aspects there of. Eventually the automatic tuning is described. The code of the auto-tuner, both C and matlab, is incorporated in the appendix.

Sammanfattning

Ett vattenjet system är ett framdrivningssystem för båtar. Principen är att vatten sugs in i skrovet, för att sedan nå en impeller som ökar vattnets hastighet. Tryckdifferensen som bildas skapar en reaktions kraft som driver vattnet till utblåset i aktern vilket ger båten fart. KaMeWa Group, ett företag i Vickers PLC koncernen som tillverkar vattenjet system, initierade arbetet. Den här rapporten behandlar dynamisk modellering av systemet och design av en regulator med automat inställning. Anledningen till att ha en automat inställbar regulator är att få objektivt inställda parametrar, som är oberoende av vem som trimmar regulatorn. Det kommer också att spara värdefull tid vid provturer. Först ges en kort beskrivning av funktion och uppbyggnad av ett vattenjet system. Sedan beskrivs identifieringen av systemet. Nästa del behandlar PID-regulatorn och dess funktion. Därefter beskrivs automat justeringen av regulatorn. Kod till regulator och automat inställningen, både C och för matlab, har inkluderats som bilagor.

Table of contents

1. The water jet system

KaMeWa was founded in 1849 to manufacture boilers and steam locomotives. Water turbines was added to the product range in 1870. In the 1930's KaMeWa applied all its previous knowledge and experience in the creation of the controllable pitch propeller. Today KaMeWa has concentrated entirely on shipboard equipment. KaMeWa's comprehensive product range comprises fixed-pitch and controllable-pitch main propellers, tunnel thrusters, azimuth thrusters, water jet propulsion systems, electronic remote-control systems and a wide range of deck machinery.

1.1 Water jet propulsion for vessels

The water jet principle consists of water being admitted to a intake at the bottom of the hull, flowing through a duct, and being discharged at the stern. While it flows through the duct, the water is pressurised by a pump driven by the propulsion machinery. The speed of the water jet discharged at the stern is much higher than the speed of the vessel, which gives rise to a reaction force that propels the craft.
A water jet installation includes the pump unit, complete with shaft, the hydraulic and electronic systems and design of the inlet duct.
The steering nozzle is hydraulically operated and can be swung 30 degrees to either side.
Thrust reversal is achieved by a reversing bucket located under the steering nozzle. The thrust is reversed by gradually moving the bucket into the water jet and thus deflecting a gradually increasing proportion of the jet in a forward/downward direction.
The purpose of the pump is to pressurise the water. The pressure differential on each side of the impeller gives rise to a thrust which is taken up by a thrust bearing located in the pump unit hub. From the hub the force is transmitted through fixed guide vanes to the transom.
The impeller is bolted to a short shaft journalled in the hub of the guide vane assembly in self-aligning roller bearings. Downstream of the impeller is a guide-vane unit designed to eliminate the rotation of the flow created by the impeller. The aft part of the guide-vane package is shaped into an outlet nozzle. The water jet has a relative high number of impeller blades and the inlet is designed to ensure that the content of higher harmonics in the water flow to the pump is small. In addition, the geometry of the impeller blades has been optimised to generate minimum noise and pressure pulses.
The generated sideforce is only depending on the jet velocity, and not on the velocity of the water approaching the inlet. This means that the sideforce is not reduced during the turning of the ship or when a broaching situation is encountered.

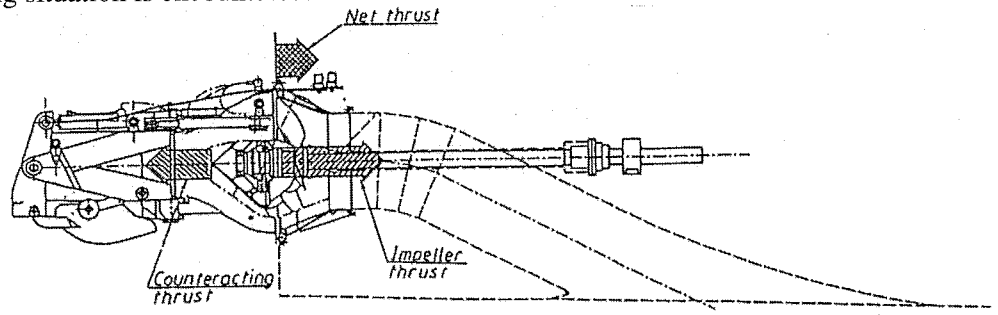


Figure 1: Schematic view of a water jet.

At the speeds of 20-50 knots that are typical of water-jet propelled vessels internal noise levels are considerably lower than in a vessel with an equivalent propeller system. Water jets provide flexible manoeuvring over the entire speed range and maximum ship control down to speed zero. Water jets have improved efficiency which means that less power is required to reach a given vessel speed or that higher speed is reached with the same amount of power. Short acceleration and retardation distances and the excellent manoeuvrability are features that makes the jet propulsion system ideal for fast commuter traffic in busy and confined waterways.

A comparison with conventional propulsion systems gives that a water jet system has:

- Higher top speed
- Reduced fuel consumption
  at speeds above 20-25 knots
- Improved manoeuvrability
  2-4 times greater "rudder forces"
  doubled rate of turn
  40-50 % smaller tactical diameter
  improved acceleration and stopping
- Reduced hydroacoustic noise
- Reduced inboard noise and vibration
- Reduced wear of engine, gearbox etc.
- Easy maintenance
- No fouling while the ship is idle
  possible to empty jets from water by compressed air

1.2 Water jet standard control system

Reversing command

A command signal to the reversing control is generated as a function of the selected thrust command signal. There are two function different curves available, one for joystick control in HARBOUR mode and one for all other modes.
If the oil flow is not sufficient for both reversing and steering, a ramp delay function is available. The ramp delay is used to avoid that the reversing hydraulics consumes too much of the available oil flow and leaving too little to the steering control. This could otherwise happen in situations when external forces on the reversing bucket work in the same direction as the applied forces from the control valve.

Steering command

A command signal to the steering control is generated as a function of the selected steering command signal. There are two different function curves available, one for autopilot and joystick control in HARBOUR mode and one for all other modes.
As for the reversing command, if the oil flow is not sufficient for both reversing and steering, the ramp delay function is used to avoid that the steering hydraulics consumes too much of the available oil flow. This corresponds of course to the same function in the reversing command and situations when external forces on the steering work in the same direction as the applied forces from the control valve.

Reversing and steering control

Function
The purpose is to shape PWM signals to the proportional hydraulic valves in order to obtain fast and smooth control of the servos for reversing and steering. The control functions are exactly the same for reversing and steering.

Derivative compensation
Before the control error determines the valve control signal, a derivative compensation signal is added. This signal is obtained by subtracting the previous value of the response signal from the current value. The difference is then multiplied with a parameter REV-DFACT (ST-DFACT) and subtracted from the control error.

Valve compensation
Proportional hydraulic valves have some inherent non-linearity's, which must be compensated for in order to achieve smooth, fast and accurate control. In this system, there are, per control valve (steering respectively reversing), separate parameters for overlap compensation in both directions, a common gain adjustment and a common proportional band adjustment. If the control error is greater than the proportional band (will happen at large commands) the system gives maximum valve command. The resulting relation between the valve control signal REV-VALVE (ST-VALVE) and the derivative compensated control error REV-CTRL (ST-CTRL) (see figure 2) .

Supply voltage compensation
In order to compensate the hydraulic valve for variations in the supply voltage, the ratio between nominal voltage (24 V) and the actual voltage is calculated. This ratio, with signal name UCOMP, is used in the PWM control. If the voltage increases, the PWM increases in order to maintain the solenoid current.

PWM control
The control signal REV-VALVE (ST-VALVE) is multiplied with the supply voltage ratio UCOMP before it is fed to the PWM block. It is also limited to values between +/- UCOMP, which means that the DC voltage over the solenoid coils can never exceed 24 V. The compensated signal REV-PWM (ST-PWM) controls the PWM block, which produces the desired valve control currents on the selected digital outputs.

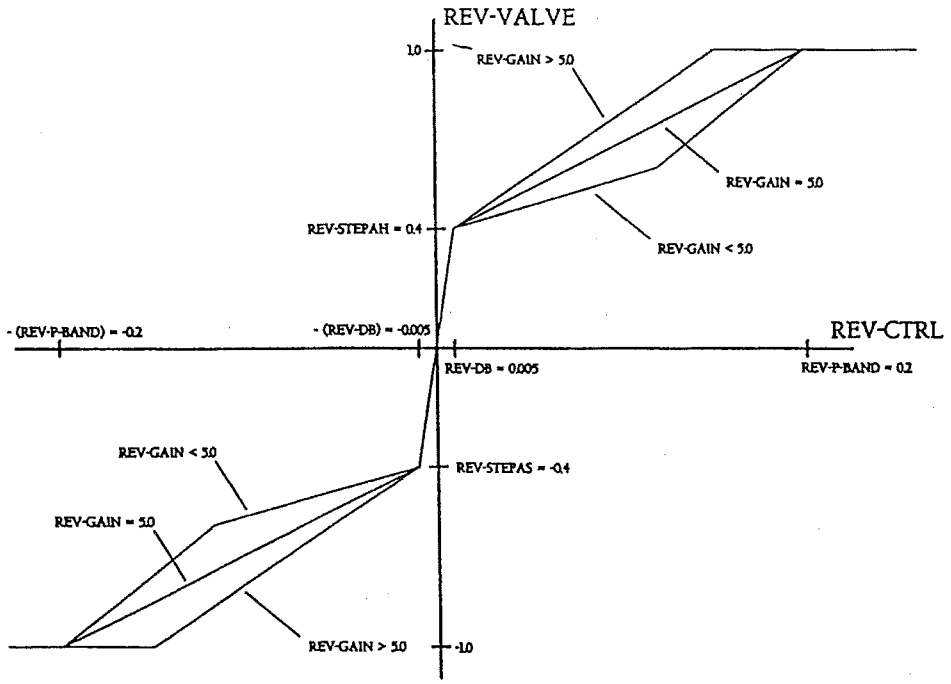
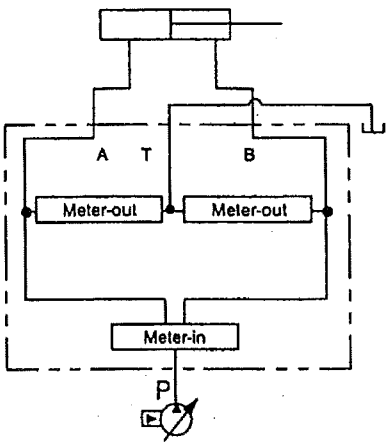


Figure 2: Proportional band control with overlap compensation.

CMX valves
The CMX sectional valve is a stackable, load sensing, proportional directional control valve. A characteristic feature of the CMX valve is the concept of separate meter-in and meter-out elements. The meter-in element is a pilot operated, flow force, pressure compensated, proportional sliding spool and controls fluid from the pump to the actuator.
The meter-out elements are pilot controlled metering poppets, and control exhaust fluid from the actuator to tank. Each meter-out poppet works as a variable orifice between one of the actuator's ports and the tank port, with the degree of opening proportional to the pilot signal.



The controller parameters

The controller of today is a PD-controller with overlap compensation as described above. It is built up in blocks, who are executed in sequence. The code is automatically generated from a AutoCad block diagram (figure 3).

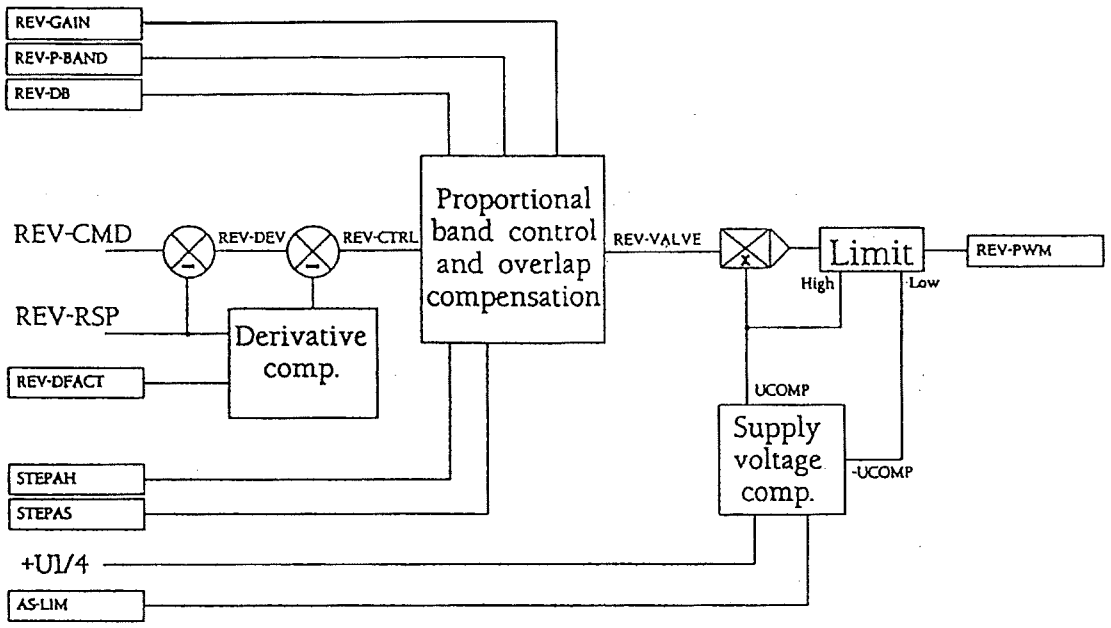


Figure 3: Block diagram of the reversing control.

There are 16 parameters today that can be used to tune the control.

-FREQ modulation frequency for the pulse width modulated valve control currents.

-UCOMP-LIM Used to switch off the supply voltage compensation if the measured voltage is less than 15.6 V or more than 32.4 V.

Parameters for reversing control (same for steering with prefix ST- instead of REV-)

-REV-DB Dead band for the reversing deviation. Inside this dead band with respect to hysteresis the system is idle.

-REV-DFACT Compensation factor for the derivative of the reversing response signal. Used to stabilise the system performance. Increase the value will stabilise the system more. Decrease the value if the system is to sluggish.

The derivative compensation of the system is not used today because it is to hard to tune, and often oscillates. The explanation to this phenomenon can be a variety of things, the D-part might be implemented wrong or the sample rate might be to low. The identification gave a system of first order and for first order systems derivative action does not improve the control.

REV-GAIN	Gain for reversing deviation. The gain factor multiply the range of the valve i.e. the range between the positive STEP values and 1.0.
REV-P-BAND	Proportional band for the reversing deviation. Decreasing the proportional band gives faster response but can cause overshoot.
REV-STEPAH	Overlap compensation of the reversing valve in ahead direction. Smallest value of the valve command to make the servo move to ahead.
REV-STEPAS	Overlap compensation of the reversing valve in astern direction. Smallest value of the valve command to make the servo move to astern.

2. Identification of the process

The key element of system identification is selecting of a model structure. It is important to choose a correct structure to get a good model. With to few parameters it isn't possible to get a good model and if there are too many parameters the fit to the measured data will be very good but the fit to other data sets will be poor, that is over-fitting.

2.1 The system

The CMX-valves (see above) are used to steer the hydraulic pistons. They are controlled by giving a current to the valve solenoid. The signal out from the controller goes to a pulse width modulator block with the modulation-frequency 50 Hz. That means if the control signal is larger than zero the solenoid gets a current which is maximal for the ratio between the control signal and the maximum control signal of the period. The rest of the period the signal level is zero. If the control signal is negative the solenoid gets a negative PWM-signal.

The basic knowledge of the process is that there is a large dead zone and that there is a time delay in the process. The signal range is between +10 V and -10 V and the dead zone is between +4 V and -4 V. The time delay can partly be explained by that the solenoid that controls the valve takes some time before it has built up enough current so it can move the hydraulic pistons. The valve has three different working modes.

1. Opens the hydraulic piston. The control signal is above 4 V.
2. The hydraulic piston isn't moved. The control signal is between +4 V and -4 V.
3. Closes the hydraulic piston. The control signal is below -4 V.

This explains the dead band.

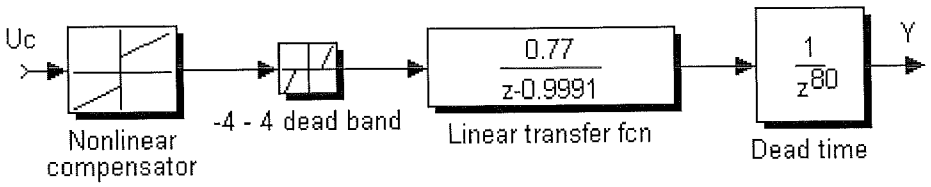


Figure 4: The system consists of a dead band, a transfer function and a dead time (h=5 ms).

2.2 The experiment

The experiment was performed on the catamaran M/S Felix in traffic between Limhamn and Dragör. The purpose of the measurements was to measure the setpoint signal, the control signal, the signal out from the PWM-block and the output signal. Due to problems with to few output channels from the

control system it was only possible to measure the control signal and the output (response) signal. This also made it necessary to involve the PWM-function in the model.
Since the vessel is in passenger traffic the security do not allow us to perform any tests of our own. The wanted data had to be collected while the boat was in traffic. The data was collected with a notebook computer, equipped with a PC-card A/D-converter. Not to loose any information the sampling interval was set to 5 ms, the maximum sampling rate of the card. The computer also was equipped with a object-oriented measurement environment, in which a measurement program was built. The data was saved in ASCII-files for further use in Matlab.

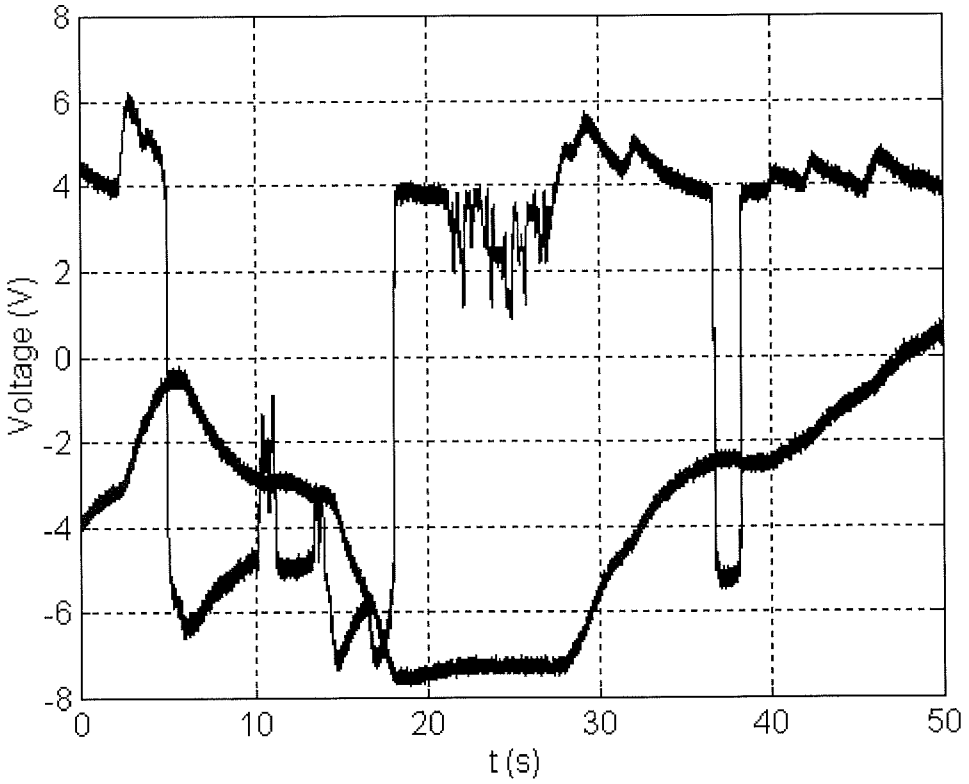


Figure 5: A data set, describing a maneuver of the reversing bucket.

2.3 The identification

The insight we have gained from studying the system, can be used in a *semi-physical grey box model.* Physical insight is used to suggest certain non-linear combinations of measured data signals. These new signals are then subjected to model structures of black box character.
Looking at the data sets a qualified guess would be; A dead band caused by the valve, a integrator or a 1st order sytem as the linear black box part, and a time delay.
The model structure to identify is, after pre-treatment by a dead band, a model of first order with time delay.
Before using the function ARX in Matlab we put every signal level between -4 V to 4 V to zero on the input signal. Looking at the measurements from the reversing the time delay was 0.3s and for steering 0.4s in time delay. This will give time delays on 60 and 80 samples (h=5 ms).
Doing the estimation a Matlab function was written, it starts with that the data sets is loaded to the workspace of Matlab.

The next important step in Matlab is to set the level of the input signal to zero if the level is between -4 V to 4 V. To get the model estimate the Matlab function arx is used. The input data to arx is:

$$th = arx([y\ u],[na\ nb\ nk])$$

where y is the output signal of the process, u is the input signal to the process, na is the order of the denominator, nb is the order of the nominator and nk is the time delay from input signal to output signal. When the arx function has been computed the estimated model is showed with the Matlab function present which takes the result from arx as input. The parameters for the polynomials A and B are given together with estimated standard deviation, innovation variance and loss function. When a model is made and the parameters have low standard deviation the model has to be validated. This is done with the Matlab function resid which as input takes:

$$E = resid([y\ u],th)$$

where y is the output signal of the validation data and u is the input signal to the process of the validation data. When calling resid the auto correlation function of E and the cross correlation between E and the input is computed and displayed. 99 % confidence limits for these values are also given. A prediction of 10 step ahead is also made with the Matlab function predict. The input to predict is:

$$P = predict([y\ u],th,m)$$

Where y and u the data the model th shall be simulated on and m is the prediction horizon. Parameters are found by applying the least square method to the measured data sets.

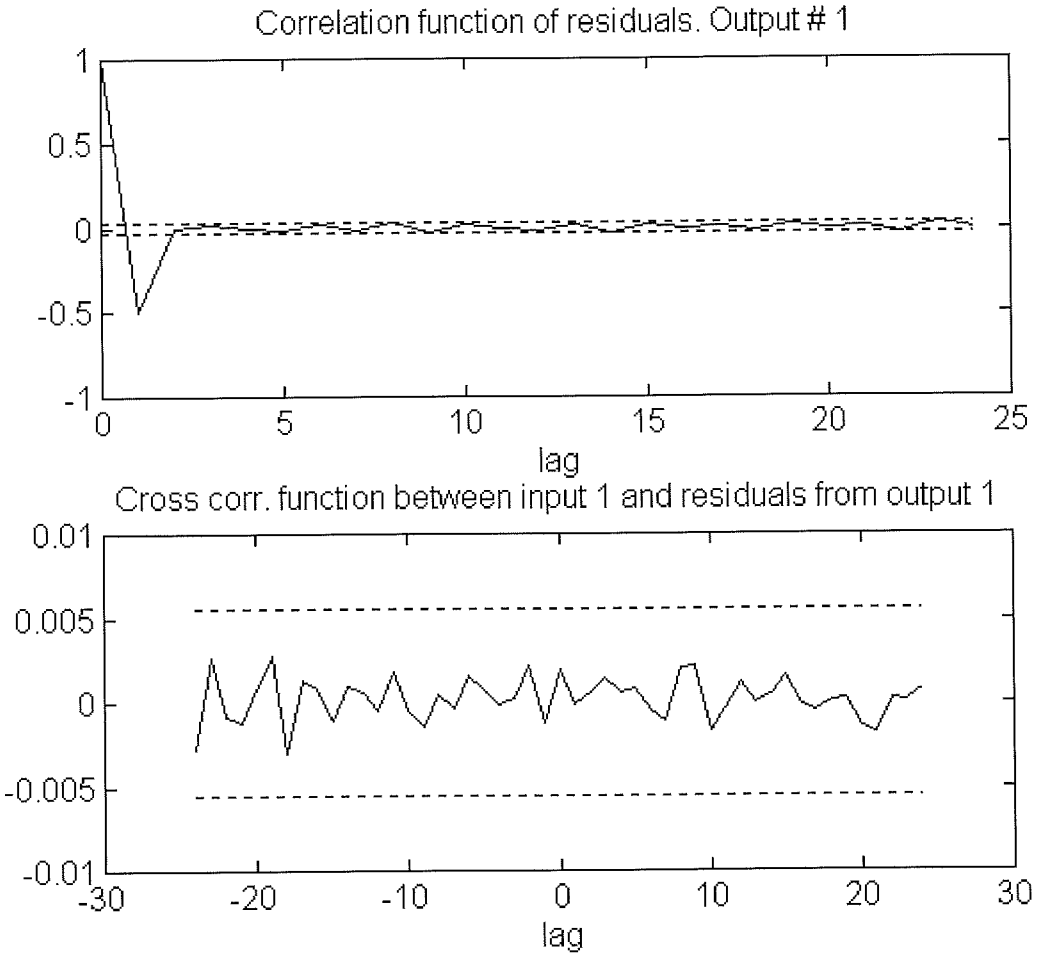


Figure 6: Validation of the model made from prov__006, on the data of prov__005.

Looking at the correlation function of residuals and cross correlation function the best model estimate for steering is from measurement prov_006.asc which gives the model:

$$y(t) = -0.994 \cdot y(t-1) + 0.9657 \cdot u(t-80)$$

The validation is made on five different data sets and the residuals look like white noise when doing a prediction ten steps ahead.

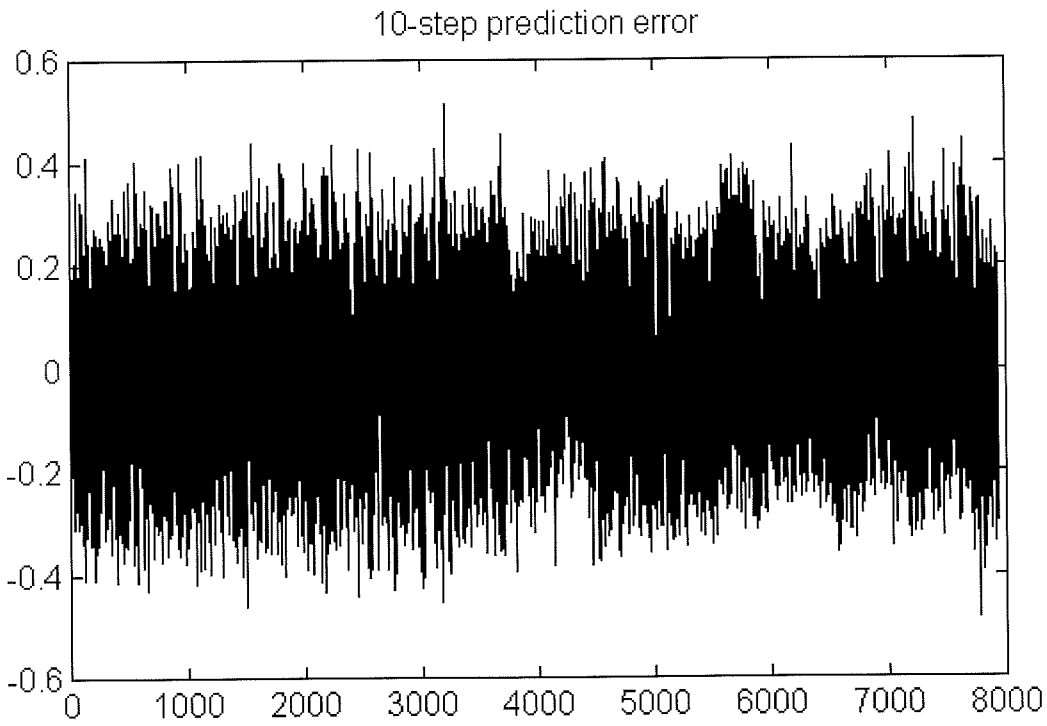


Figure 7: The prediction error from the 10-step prediction compared with the real output.

The best model for the reversing bucket is from measurement rev__028.asc which gives the best correlation function and cross correlation function for the validation sets.
The model after estimate:

$$y(t) = -0.993 \cdot y(t-1) + 0.6041 \cdot u(t-60)$$

3. The controller

The choice of PID-structure for the controller was not very hard. A PID controller is the most commonly used controller in commercial applications. It is simple to implement and there are some extra features, anti-windup and bumpless parameter change, which will be useful in this case.
There is also a lot of different methods for tuning a PID-controller, depending on what the purpose of the control is.
For the water jet system which we has identified as a first order process. That is why there is no need of the derivative part of the controller because the D-part will not give any improvment in the control of the system.
Because the valve has a deadband between -4 V to 4 V it is important to compensate this non-linearity so the system will appear linear from input to output. The solution to this is to turn the non-linearity. It is impossible to make a exact compensation around zero because the gain go to infinity if the compensation is exact.

3.1 The algorithm

The name PID-controller is a description of it's function. It consists of three parts, the P = proportional, the I = integral and the D = derivative part. Through the years of development many adjustments have been made on the small-signal behaviour of the algorithm. The original function can be written as:

$$u(t) = K_c\left[e(t) + \frac{1}{T_i}\int^{t} e(s)ds + T_d\frac{de(t)}{dt}\right] = P + I + D$$

The proportional part of the controller can be described as:

$$u = K_c\left(y_{sp} - y\right) + u_b$$

K_c is the proportional gain and u_b is a bias. u_b is used to adjust the control signal so that steady state is obtained at the desired operating point. In order not to make this adjustment manually, the integral part was introduced. What the integral part does is that it low-pass filter the controller output to automatically adjust the bias. This means that there will be no error at steady state, which can be easily shown.
If a process has input (u_0), output and the error (e_0) is constant, the control law will give that:

$$u_0 = K_c\left(e_0 + t\frac{e_0}{T_i}\right)$$

This gives that u_0 is a constant unless e_0 is zero, the controller will work until the error is zero.

This means that a small error may generate a large control signal if it exists for a long time. The parameter T_i is called integral time or reset time, it is the time constant of the low-pass filter. The proportional part uses the knowledge of the present, the integral part uses the past. The thing missing in the concept is the knowledge of the future, which is exactly what the derivative part is using. By using the tangent of the error curve, the controller can predict which way the error is going to change. The derivative part can be defined as:

$$u(t) = T_d \cdot \frac{de(t)}{dt}$$

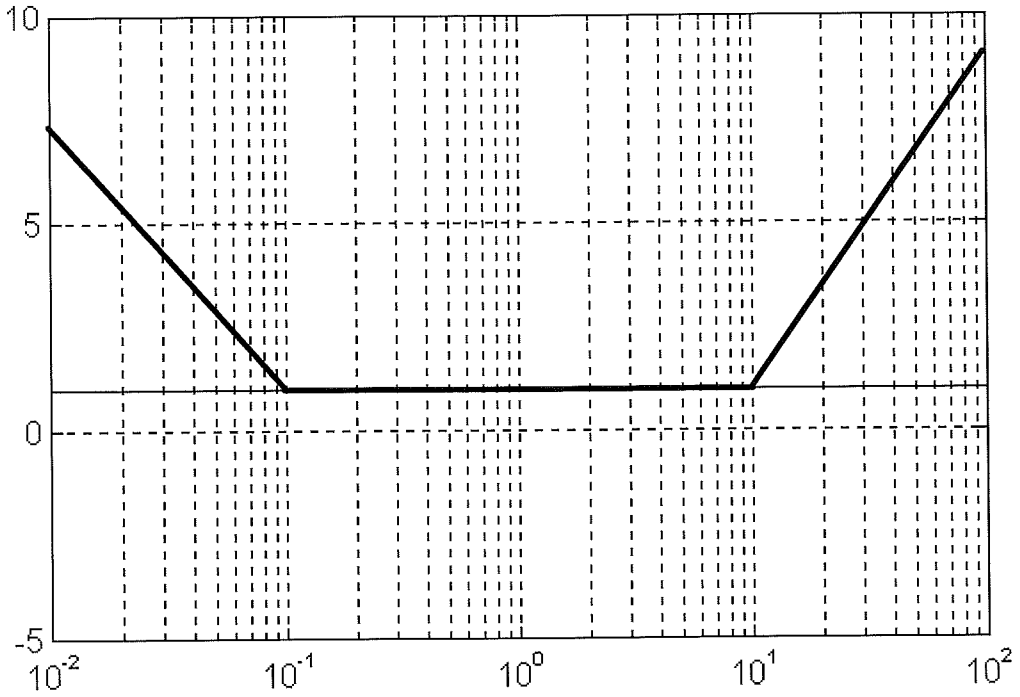


Figure 8: Bodeplot of a PID-controller

The parameter Td is called the derivative time and represents the horizon of prediction. Since the derivative part is a predictor, it is of no use if the process is of the 1 st order.
The bode plot of the full PID-controller shows that it has infinite gain at low frequencies. This is caused by the low pass-filter in the integral part. There is also infinite gain at high frequencies due to the derivative part. This is not desirable. In order to stop the controller from amplifying the high frequency noise to much, a low-pass filter is included in the derivative part:

$$D(s) = \frac{sT_d}{1 + s\,{}^{T_d}/_{N}} \cdot E(s)$$

Because of the filter the derivative part will still advance the phase in the requested frequency band, without amplifying the high frequency noise. This way it will stabilise the system without making it more sensitive to noise. An other drawback with the derivative part is that it is very hard to choose the parameter T_d. This is why the derivative part often is switched off in industrial systems.

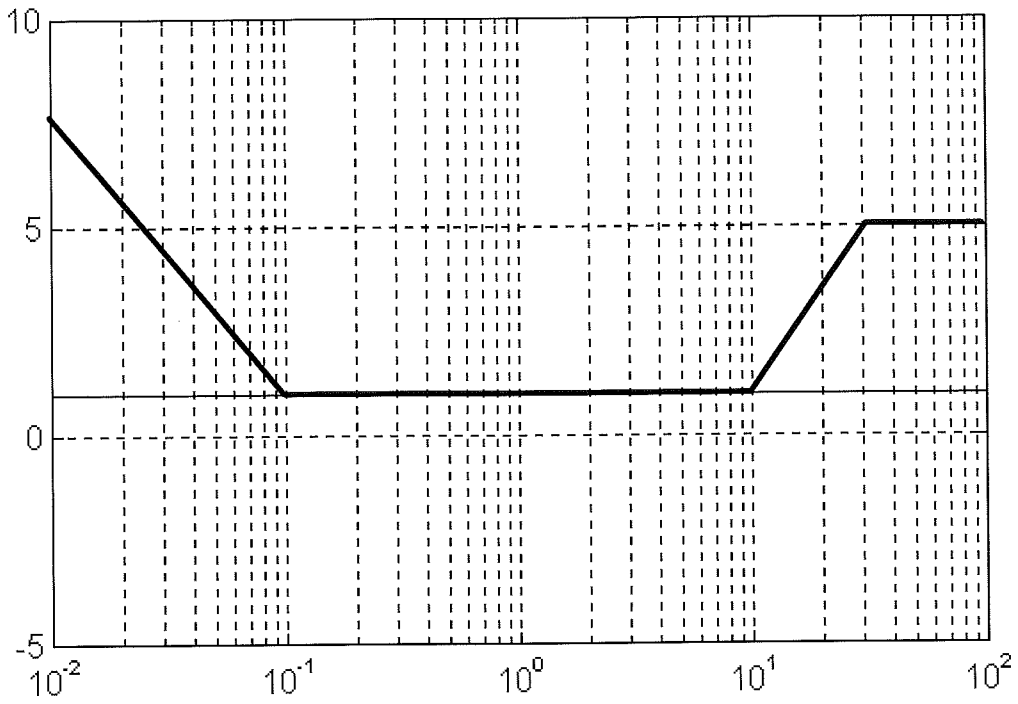


Figure 9: Bodeplot of PID-controller with low-pass filter in the derivative part

An other small-signal aspect of the derivative is how to treat step changes in the setpoint. Since a step has an infinite derivative, it has been empirically observed that it often is an advantage not to let the derivative act on the command signal. There for a modification of the derivative part has been made:

$$D(s) = \frac{sT_d}{1 + s\,{}^{T_d}\!/_{N}} \left(\gamma \cdot Y_{SP}(s) - Y(s)\right)$$

The most common case is $\gamma = 0$, which means that the derivative part does not operate on the command signal.
In order to provide different paths for the feedback signal and the setpoint, there has also been introduced a parameter β. Setpoint weighting makes it possible to tune different step responses for load disturbances and setpoint changes.
For example; a controller, K_c, T_i and T_d, is tuned to give a good load disturbance response. Then the setpoint response will have a large overshoot, if $\beta = 1$. By adjusting β a better setpoint response can be achieved without loosing the good response on load disturbances.

The PID-algorithm obtained is:

$$U(s) = K_c \cdot \left[\beta \cdot Y_{SP}(s) - Y(s) + \frac{1}{sT_i}\left(Y_{SP}(s) - Y(s)\right) + \frac{sT_d}{1 + s\,{}^{T_d}\!/_{N}} \left(\gamma \cdot Y_{SP}(s) - Y(s)\right)\right]$$

3.2 Integrator windup

The world is almost never nice and linear. The speed of a motor, the stroke of a piston and the opening of a valve are all non-linear because of saturation.
Since the integral part of a PID-controller is unstable the integral part may integrate up to very high values when the actuator saturates. This is called integrator wind-up. The effect of this is that the integrator part not will integrate down to normal level in time and there will be a large overshoot.

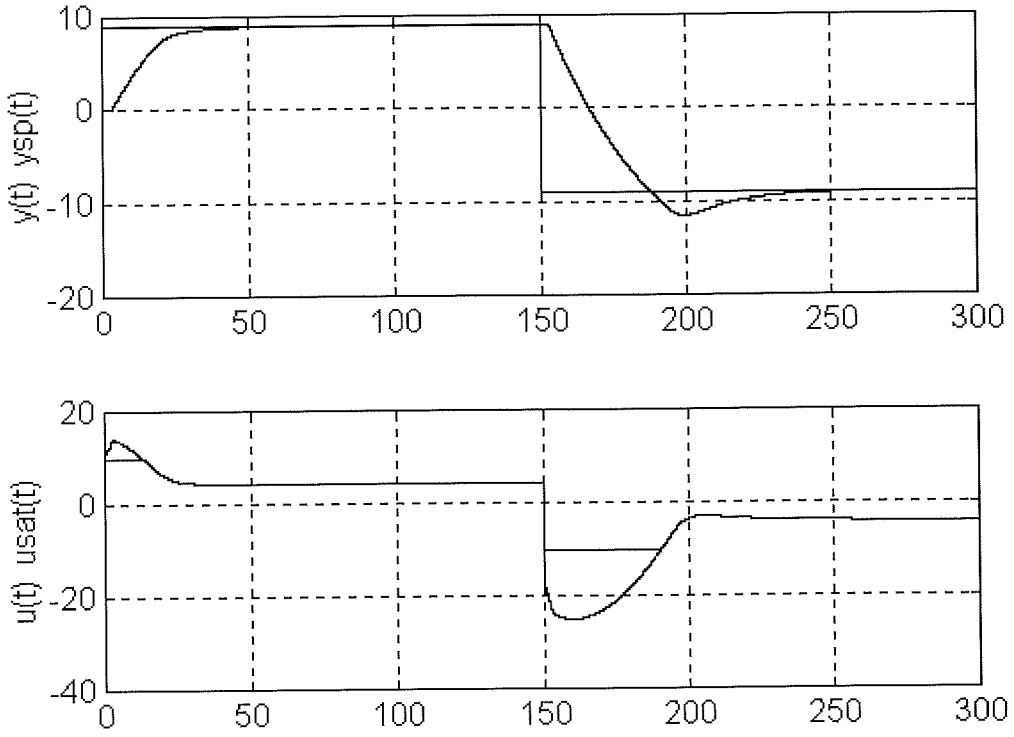


Figure 10: Step-response with a controller without tracking

This effect can easily occur in the Water Jet-system because of the limited opening of the valve. There are a lot of different ways of solving this problem. One way of solving it is called tracking or back-calculation. Tracking means that when the output saturates, the integral part is recalculated so that the new output is at the saturation limit. The reset of the integrator is done dynamically with a time constant T_t. This time-constant is typically larger then T_d and smaller then T_i. A rule of thumb is; For PID-controller $T_t = \sqrt{T_i \cdot T_d}$ and for PI-controller $T_t = T_i/2$.

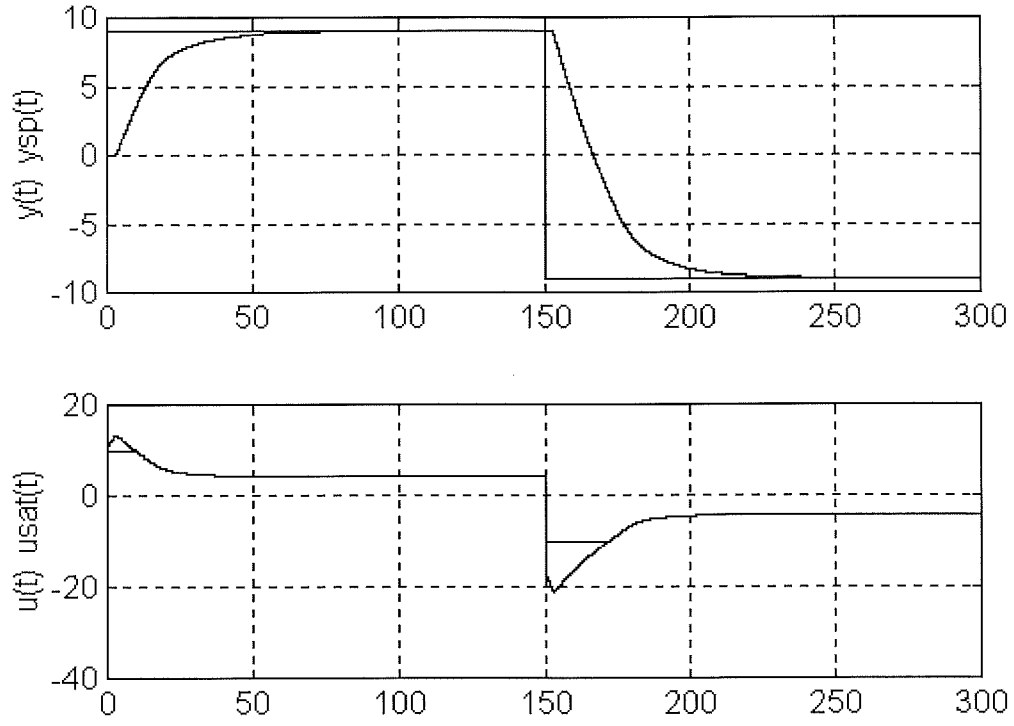


Figure 11: Step-response with a controller with tracking.

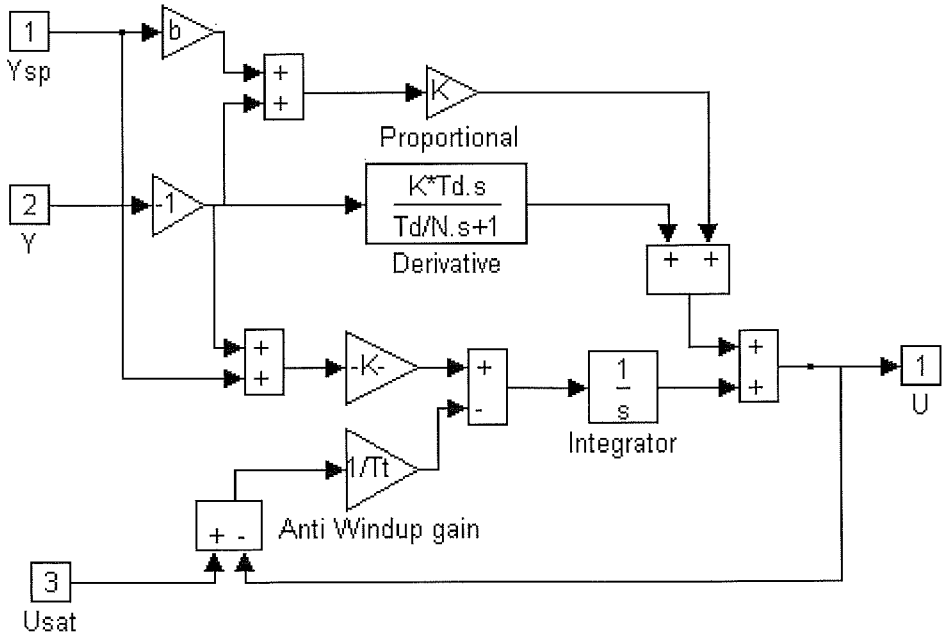


Figure 12: Block diagram of PID-controller with tracking

Tracking acts like a controller which wants to drive the controller output to follow the saturated actuator output. If the saturated output cannot be measured a mathematical model of the actuator can be used. Tracking can also be used to track manual control to avoid bumps when switching between manual and automatic control. This is called bumpless mode transfer and has not been implemented in our controller.

3.3 Bumpless parameter change

Because a controller is a dynamic system, a change in parameters will cause a change in output. In a position algorithm, which is used in this case, the change in output depends on the implementation. If the state is chosen as:

$$x_I = \int^t e(\tau) \cdot d\tau$$

the integral part will be:

$$I = \frac{K_c}{T_i} \cdot x_I$$

A change of K_c or T_i will give a change in I. To avoid bumps on parameter change, it is necessary to choose the state as:

$$x_I = \int^t \frac{K_c(\tau)}{T_i(\tau)} \cdot e(\tau) \cdot d\tau$$

when implementing the controller. Since $K_c(\tau)/T_i(\tau)$ is a part of the state, the integral part will not change while the error is zero. Therefore it can be guaranteed a bumpless parameter change, if the changes only take place when the error is zero. If setpoint weighting is used, we also have to keep the sum $P + I$ invariant to parameter changes. This is done by recalculating the state I when parameters are changed.

$$I_{new} = I_{old} + K_{old} \cdot (\beta_{old} \cdot y_{SP} - y) - K_{new} \cdot (\beta_{new} \cdot y_{SP} - y)$$

3.4 Discretization

The proportional part is:

$$P = K_c \cdot (\beta \cdot y_{SP} - y)$$

To get the discrete P-part you only replace the continuous variables with the sampled ones as:

$$P(t_k) = K_c \cdot (\beta \cdot y_{SP}(t_k) - y(t_k))$$

The integral part is:

$$I(t) = \frac{K_c}{T_i} \cdot \int_0^t e(s) \cdot ds$$

When updating the integral part we are using forward difference

$$\frac{I(t_{k+1}) - I(t_k)}{h} = \frac{K_c}{T_i} \cdot e(t_k)$$

Using forward difference has the advantage that the I-part can be updated at the end of the cycle, to get a more accurate sampling time. This is not possible if backward difference is used.

The continuous D-part can be written as:

$$\frac{T_d}{N} \cdot \frac{dD}{dt} + D = -K_c T_d \cdot \frac{dy}{dt}$$

if $\gamma = 0$.

If the D-part is updated with forward difference:

$$D(t_{k+1}) = \left(1 - \frac{Nh}{T_d}\right) \cdot D(t_k) - K_c N \cdot \left(y(t_{k+1}) - y(t_k)\right)$$

It requires that $T_d > Nh/2$ otherwise will the approximation become unstable.
Instead we are using backward difference which is the most commonly used

$$D(t_k) = \frac{T_d}{T_d + Nh} \cdot D(t_{k-1}) - \frac{K_c T_d N}{T_d + Nh} \cdot \left(y(t_k) - y(t_{k-1})\right)$$

which will give good results for all values of T_d.

4. Automatic tuning

The purpose of automatic tuning is to give good control parameters independent of who is doing the adjustment of the parameters.

4.1 Step response method

If the tuning is based on the step response of the process, you use three parameters to calculate the control parameters. The parameters are the static gain K_p, the dead time L and the lag T see figure 13.

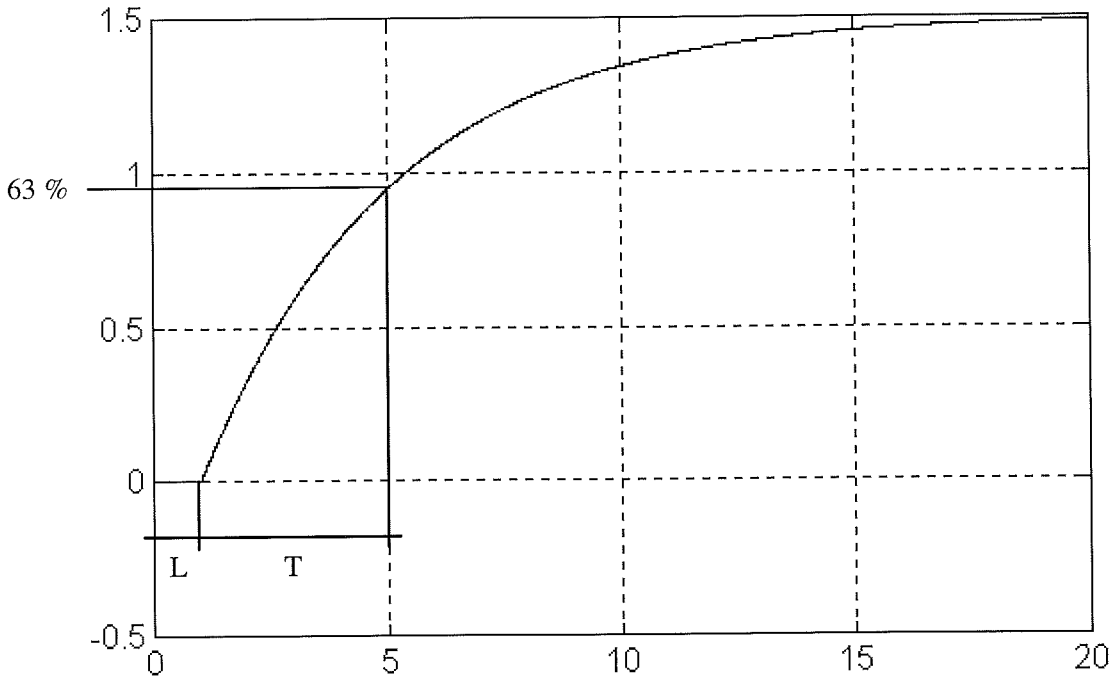


Figure 13: Step-response of a first order process.

To choose the right K_c for the controller you have to calculate the parameter a

$$a = K_p \cdot \frac{L}{T}$$

K_p is obtained from the steady state level of the process output scaled with the change in the process input. L is the dead time of the process which is the time until the process responds on a step change. When making a step to the process it is essential that the change in the control variable is as large as possible in order to get a maximum signal to noise ratio.

The relative dead time τ is used instead of T.

$$\tau = \frac{L}{L+T} = \frac{L}{T_{ar}}$$

A regular PID-controller have four parameters that can be adjusted, the gain K_c, the integration time T_i, the setpoint weighting β and the derivative time T_d. The parameters have to be normalised so that they will be dimensionless. The normalised controller gain is aK, the normalised integration time is T_i/L and normalised derivative time is T_d/L.

The method to find out the controller parameters is based on a new empirical method. The controller parameters have been computed for different processes in a test batch. To find out the controller parameters a relation has been found out between the normalised controller parameters and the normalised process parameters.

The function used to find the controller parameters has the form:

$$f(\tau) = a_0 \cdot e^{a_1\tau + a_2\tau^2}$$

The parameters a_0, a_1 and a_2 are the parameters that fits the curve when plotting the normalised controller parameters as a function of the normalised dead time τ.

When tuning the sensitivity, M_s, of the closed loop system, can be chosen.

$$M_S = \max_{0\le\omega<\infty}\left|\frac{1}{1+G_p(i\omega)G_c(i\omega)}\right| = \max_{0\le\omega<\infty}|S(i\omega)|$$

M_s represents robustness against model errors. It is the inverse of the distance from the Nyquist curve to the critical point -1. $1/M_s$ is the radius of a circle surrounding the critical point. The Nyquist curve of the loop transfer function is always outside the circle. This means that the controller gain can be raised with the factor $M_s/(M_s-1)$ before the system will be unstable. M_s are a value in the range between 1.2 to 2.0.
If the value of M_s is small the system will be able to handle non-linear actuator characteristics.

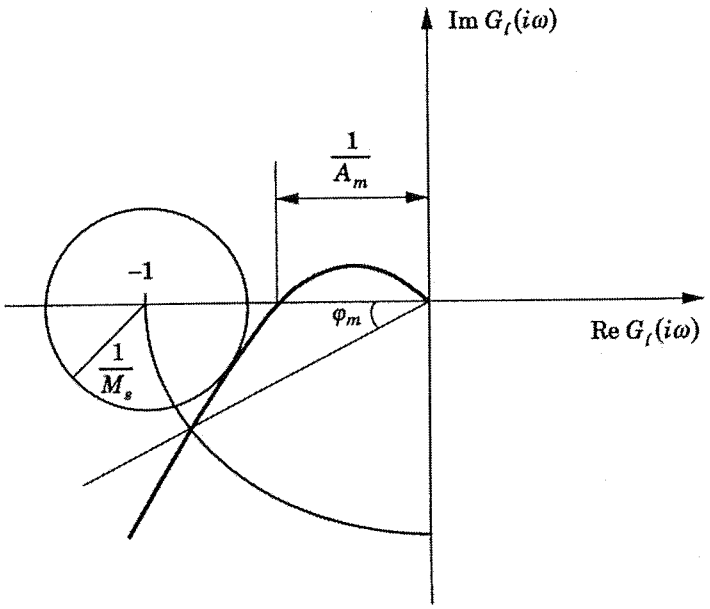


Figure 14: Definition of sensitivity, M_s .

	M_s=1.4			M_s=2.0		
	a_0	a_1	a_2	a_0	a_1	a_2
aK	0.29	-2.7	3.7	0.78	-4.1	5.7
T_i/L	8.9	-6.6	3.0	8.9	-6.6	3.0
T_i/T	0.79	-1.4	2.4	0.79	-1.4	2.4
b	0.81	0.73	1.9	0.44	0.78	-0.45

The tuning parameters for a PI-controller.

	M_s=1.4			M_s=2.0		
	a_0	a_1	a_2	a_0	a_1	a_2
aK	3.8	-8.4	7.3	8.4	-9.6	9.8
T_i/L	5.2	-2.5	-1.4	3.2	-1.5	-0.93
T_i/T	0.46	2.8	-2.1	0.28	3.8	-1.6
T_d/L	0.89	-0.37	-4.1	0.86	-1.9	-0.44
T_d/T	0.077	5.0	-4.8	0.076	3.4	-1.1
b	0.40	0.18	2.8	0.22	0.65	0.051

The tuning parameters for a PID-controller.

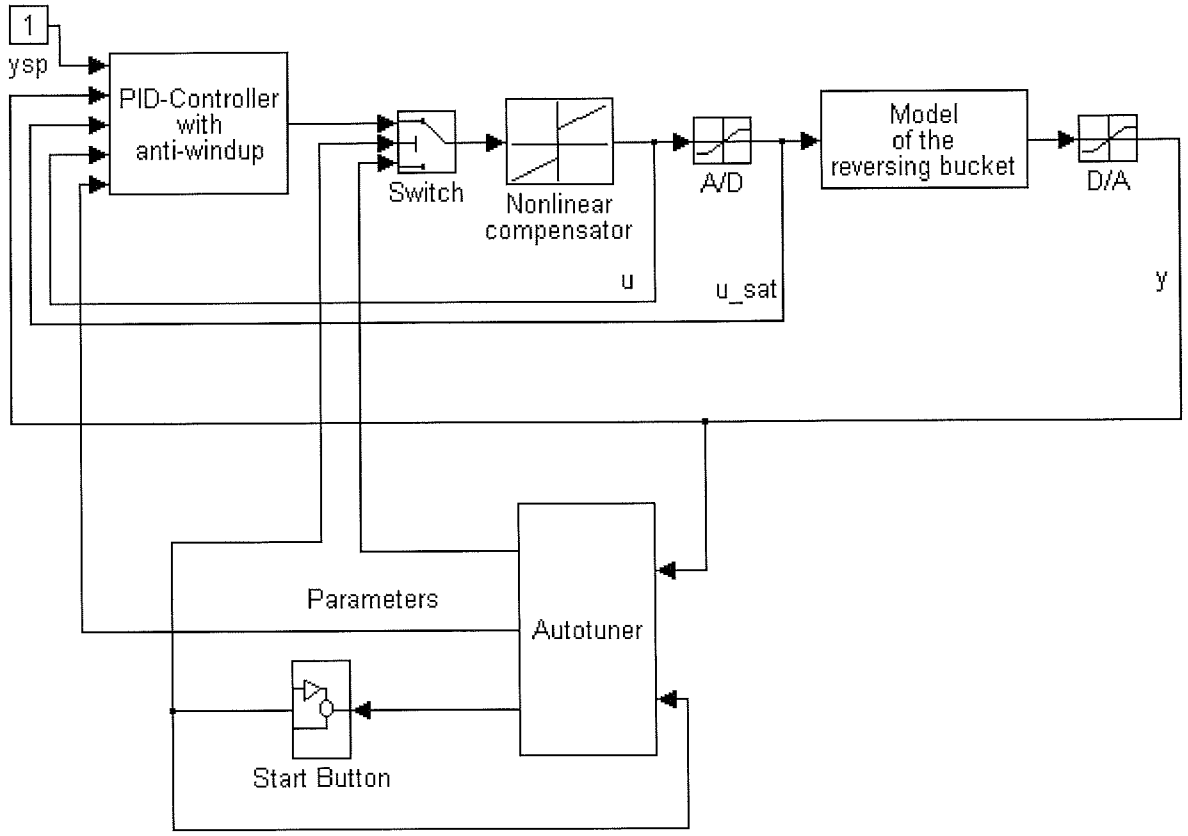


Figure 15: Block diagram of auto-tuned system.

When implementing the automatic tuning in our model we used a finite state machine.

1. In the first state the tuning is initialised, buffers are cleared etc.

2. In the second state the noise-level is examined for a couple of seconds to decide the tolerance during the tuning. The maximum and minimum value of the process output is measure to calculate the tolerance.

   Tolerance = maximum value - minimum value

3. After the listening state a step change is given. When giving the step the start time is logged.

4. In the next state the dead time is measured. When the process output starts to move i.e. the output level is higher than start level + tolerance the dead time L can be calculated as:

   L= *current time - start time*

5. In next step the static gain of the process is measured. The output of the process is logged in a ring-buffer which has 4/h elements. The output is considered to have reached steady state when the latest and the oldest value in the ring-buffer is within the tolerance. When this criteria is fulfilled the static gain K_p of the process is calculated as:

   K_p = *(level of the process output - start value of the process output) / the step height*

6. In this state step response two is started. A negative step change is made back to input the process had before the tuning algorithm started. A new start time is logged.

7. The dead time L is measured again and a average L is calculated. This will be used to calculate the relative dead time τ.

8. When the process have reached 37 % of the steady state level the time is logged. The lag of the process (T) can now be calculated.

9. Now the parameters a and τ is calculated as:

$$a = K_p \frac{L}{T}$$

$$\tau = \frac{L}{L+T}$$

   The controller parameters are also calculated in this state. When starting the automatic tuning the chose is made whether to tune parameters for a PI or a PID controller and which sensitivity the closed loop system will have. $M_s = 1.4$ or $M_s = 2.0$ can be chosen.

10. Finally the process is allowed to reach steady state again.

The code for the auto-tuner is written as a s-function and implemented in a Simulink model where it has been simulated with fairly good results.
The controller parameters from the tuning has been simulated with our suggested controller, the model of the water jet system with compensation for the non-linearity of the valve. The code for the auto-tuner is also written in C. This code has been simulated against a first order discrete process with delay which also has been written in C.

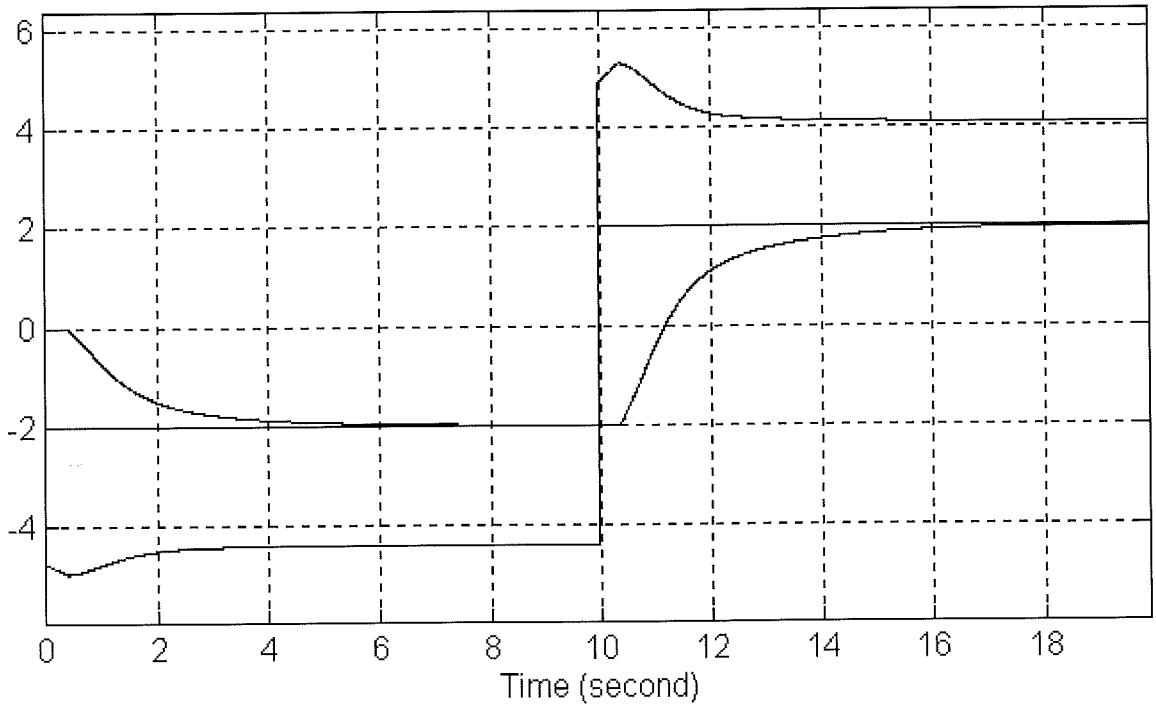


Figure 16: The response of the simulink model, with tuned parameters.
The step represents 3 degrees/volt. To increase the speed, β can be increased.

We decided to chose the $\kappa\tau$ step response method because it is easy to implement and it is reliable on the process, a water-jet system. Also the step experiment doesn't take very long time to perform. There are other methods to tune the controller parameters, e.g. the frequency response method where you use a relay with hysteresis to make the process oscillate. There are also different ways of calculating the parameters, e.g. Cohen-Coon method which tries to minimise the steady state error due to load disturbances for P and PD controllers and the integral error for PI and PID controllers.

4.2 Alternative method

In chapter 2, the decision was made to identify the linear part of the process as a first order function. If the model instead was identified as an integrator, the test and the calculations would be much easier to perform and understand.

A normalized linear process, in continuous time, would be:

$$G(s) = \frac{1}{s} e^{-s}$$

and the controller would be:

$$G_c(s) = K_c^0 \left(1 + \frac{1}{sT_i^0} \right)$$

It is possible to design controllers to control this normalized process. Then the control parameters can be recalculated to fit the identified process. The process will be:

$$G(s) = \frac{k}{s} e^{-sL} = \frac{kL}{sL} e^{-sL} = \frac{kL}{s'} e^{-s'}$$

The controller of the identified process will be:

$$G_c(s) = K_c \left(1 + \frac{1}{sT_i} \right) = K_c \left(1 + \frac{L}{T_i} \cdot \frac{1}{sL} \right) = K_c \left(1 + \frac{L}{s'T_i} \right)$$

These equations gives the recalculation formulas as follows:

$$K_c L = K_C^0 \quad \Rightarrow \quad K_c = \frac{K_c^0}{L}$$

$$\frac{1}{T_i^0} = \frac{L}{T_i} \quad \Rightarrow \quad T_i = T_i^0 \cdot L$$

When using this method it is possible to precalculate a table of controllers and with a minimum of effort recalculate the parameters to fit the present process. This table would be the same in all control systems, no matter what size of water jet to control.
The test performed will also be more simple since the only parameter needed is the time delay.
The table below contains precalculated values of K_c, T_i and β, for a range of different sensitivities (M_s). As before a high value of M_s gives speed, and a low value gives robustness:

M_s	K_c^0	T_i^0	β^0
1.4	0.2821	6.7439	1.3516
1.6	0.3687	5.0752	1.0168
1.8	0.4355	4.2402	0.8061
2.0	0.4885	3.7388	0.6532

This method is described further in an conferance article;

Panagopoulos H., K.J. Åström, T. Hägglund. :"A numerical method for design of PI-controllers.", In *6th IEEE Conference on Control Applications*, Hartford, Connecticut, October 1997.

5. Practical discussion

5.1 Controller implementation

A usual problem in sampled systems is aliasing. This means that frequencies above the half sampling frequency can appear as lower frequencies. This problem can be solved by using a analog anti aliasing filter before sampling.
When implementing a control system there are certain things that have to be considered. It is of great importance that the time between input and output is minimized, to make the mathematical model as accurate as possible. This means that there must be as few calculations as possible between reading the input and writing the output.

1. Read input from A/D
2. Calculate controller output
3. Write output to D/A
4. Update controller states
5. Wait for clock interrupt (Then go to step 1)

To implement the automatic tuning in the system it is important that the state of the system is correct when switching between tuning and controller mode. When the system is in the automatic tuning mode, the controller produces a control signal that may be different from the control signal from the automatic tuning block. The outputs from the controller and the automatic tuning block must coincide at the time of switching. This is called bumpless transfer.

5.2 Automatic tuning

The kappa-tau step response method tune controller parameters that gives a stable step response without overshoot. This may sometimes give a to conservative controller. In this case it is important with a fast response to setpoint changes. To adjust the speed without risking the stability of the closed loop system the parameter β can be changed. A increased value of β gives a faster response to setpoint changes but it can also cause overshoots of the output signal. The parameter β must not be greater than 1 otherwise the system can be unstable.

5.3 Future enhancements

A possible enhancement in the future could be adaptive dead band compensation. There is a article written by Mattias Grundelius on this issue and also a report by Gang Tao;

Grundelius M. : "Adaptiv reglering av system med glapp," (Adaptive control of systems with backlash). Master thesis ISRN LUTFD2/TFRT -- 5549 – SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, January 1996.

Tao G. and P.V. Kokotovic. : "Adaptive control of systems with actuator and sensor nonlinearities." ISBN 0-471-15654-X, John Wiley & Son Inc., 605 Third Avenue, New York, New York, 1996.

Another possible enhancement is gain-scheduling. Since different properties are wanted in sea-mode than in harbour-mode, it migth be a good thing to design different controllers for each mode. This can be made by tuning once and then calculate a robust controller for harbour-mode and a fast controller for sea-mode. Then when switching mode of steering, the control parameters also are changed.

6. References

Åström, K. J., T. Hägglund (1995): "PID Controllers: Theory, Design, and Tuning." Instrument Society of America, Research Triangle Park, North Carolina.

Ljung, L. (1987): "System Identification - Theory for the user." Prentice-Hall, Englewood Cliffs, New Jersey.

Svensson, R. : "Water jet propulsion of high speed passenger vessels" KaMeWa AB, Kristinehamn.

A. Appendix

A.1 File descriptions

The C-code is divided into 4 files:

`PIDTYPES.C`	Includes the type definitions and data structures of the PID-controller and the tuner.
`PID.C`	Includes the functions needed to implement a PID-controller, it is depending on `PIDTYPES.C`.
`AUTO.C`	Includes the functions needed to implement a automatic tuner for a PID-controller. It is depending on `PIDTYPES.C`.
`COMP.C`	Includes the functions needed to implement the dead band compensation.

There is also some test files available:

`PIDTEST.C`	This program tests the function of the PID, it uses `FIRSTORD.C` to simulate control of a first order process. The signals are stored in a ASCII-file.
`AUTOTEST.C`	This program performs a step-response experiment on a first order process, using the functions from `AUTO.C` and `FIRSTORD.C`.
`COMPTEST.C`	This program makes a test run of the dead band compensation.
`FIRSTORD.C`	Includes a function Process, which simulates a first order discrete process with time delay.

A.2 Implementation

PIDTYPES.C

```
#define TimeOut 30
#define alfa 0.05
#define index 2000
#define MaxClock 65535

                                    /* Typedefintions to Pid.c and Auto.c  */
typedef struct
{
  double K,Ti,Td,N,Tt,beta,gamma,h; /* h is the sample interval in seconds */
  char ion;                         /* ion=1 -> integral part active       */
} PARAMETERS;

typedef struct
{
  PARAMETERS params;
  double a,b,c,d;                   /* precalculated factors depending on */
} REGPAR;                           /* the parameters of the controller   */

typedef struct
{
  REGPAR par;
  double I,D,yold,yrefold,v;
} REGDATA;

typedef REGDATA* Regul;

typedef char STATE;

typedef struct{                     /* Structure including the parameters*/
  double Kp,T,L;                    /* of the tuning. The process is     */
  double a,tau,tol;                 /*           Kp/(sT+1)*e^(-sL)       */
  char type,sens;                   /* type=0 -> PI      type=1 -> PID   */
  PARAMETERS pars;                  /* sens=0 -> Ms=1.4  sens=1 -> Ms=2.0*/
} TUNER;

typedef struct{                     /* Structure that desc. the current  */
  unsigned int sek;                 /* sample.                           */
                                    /*   in=process output               */
  unsigned int milli;               /*   uncomp=uncompensated control    */
  double in,uncomp;
} NOWTYPE;
```

PID.C

```
/*****************************************************************************/
/*                                                                           */
/* This module is a standard implementation of a PID control algorithm       */
/* on direct form.                                                           */
/* Reference: Karl Johan Åström, "PID Control"                               */
/*                                                                           */
/* The used PID formula is:                                                  */
/*                                                                           */
/*                       v = K*( P + I + D )                                 */
/*                                                                           */
/*                       P = beta*yref - y                                   */
/*                       I = (yref - y) / (s*Ti) + (vsat - v)/(s*Tt*K)       */
/*                       D = (s*Td*(gamma*yref - y))/(1 + s*Td/N)            */
/*                                                                           */
/* where v is the control signal, y is the plant measurement, yref is the    */
/* setpoint, and vsat is the limited control signal that is used as tracking */
/* signal in the anti-reset windup. The controller also supports bumpless    */
/* parameter transfer.                                                       */
/*                                                                           */
/*                                                                           */
/*                    modified and translated to C for KaMeWa                */
/*                by by Anders Hansson and Per-Inge Tallberg 970812          */
/*                                                                           */
/*****************************************************************************/
#include <stdlib.h>
#include "pidtypes.c"



/******************************** GetPar_reg *********************************/
/* Returns the parameters of the controller reg                              */
/*****************************************************************************/

PARAMETERS GetPar_reg(Regul reg)
{
  return reg->par.params;
} /* GetPar_reg */


/******************************** SetPar_reg *********************************/
/* Sets the parameters figs to the controller reg. It is an error to set     */
/* h,Ti or Tt to zero.                                                       */
/*****************************************************************************/

void SetPar_reg(Regul reg, PARAMETERS figs)
{
  reg->par.a=figs.Td/(figs.Td+figs.N*figs.h);
  reg->par.b=figs.K*figs.N*reg->par.a;
```

```
  reg->par.c=figs.h*figs.K/figs.Ti;
  reg->par.d=figs.h/figs.Tt;
  reg->I=reg->I+reg->par.params.K*(reg->par.params.beta*reg->yrefold-reg->yold);
  reg->I=reg->I-figs.K*(figs.beta*reg->yrefold-reg->yold);
  reg->par.params=figs;
} /* SetPar_reg */


/******************************** CalcOut_reg ********************************/
/* Computes and returns the output of the controller reg                     */
/*****************************************************************************/

float CalcOut_reg(Regul reg, double yref, double y)
{
  double ep;

  ep=reg->par.params.beta*yref-y;
  reg->D=reg->par.a*reg->D+reg->par.b*(reg->par.params.gamma*
                                       (yref-reg->yrefold)-(y-reg->yold));

  if (reg->par.params.ion == 1)
     reg->v=reg->par.params.K*ep+reg->I+reg->D;
  else
     reg->v=reg->par.params.K*ep+reg->D;
  reg->yold=y;
  reg->yrefold=yref;

  return reg->v;
} /* CalcOut_reg */


/******************************** Update_reg *********************************/
/* This function updates the internal states of the controller. It must      */
/* always be called after CalcOut_reg to complete the current sample.        */
/*****************************************************************************/

void Update_reg(Regul reg,double vsat)
{
  double temp;

  if (reg->par.params.ion == 1)
    reg->I=reg->I+reg->par.c*(reg->yrefold-reg->yold)+reg->par.d*(vsat-reg->v);
  else
    reg->I=0.0;
} /* Update_reg */
```

```
/********************************** New_reg ***********************************/
/* Allocates memory and sets the initial parameters.                          */
/******************************************************************************/

Regul New_reg()
{
 Regul reg;
 PARAMETERS initial;

 reg=malloc(sizeof(REGDATA));

/* Setting initial values */
 initial.K=1.0525;
 initial.Ti=2.002;
 initial.Td=0.0;
 initial.N=1.0;
 initial.Tt=1.0;
 initial.ion=1;
 initial.beta=0.88;
 initial.gamma=0.0;
 initial.h=0.02;                /* fsamp=50Hz */
 SetPar_reg(reg,initial);

/* Reseting states */
 reg->I=0.0;
 reg->D=0.0;
 reg->yold=0.0;
 reg->yrefold=0.0;

 return reg;
} /* New_reg */


/********************************** Dispose_reg *******************************/
/* Cleans up and returns memory to the system.                                */
/******************************************************************************/

void Dispose_reg(Regul reg){

  free(reg);
} /* Dispose_reg */


/******************************** End of Module *******************************/
```

AUTO.C

```
#include <values.h>
#include <math.h>
#include "pidtypes.c"

/*************************************************************************/
/* This function updates a variable of NOWTYPE-type. It must be called  */
/* first in every sample. ex: now=*sample(y,t_sek,t_milli,u)             */
/*************************************************************************/

NOWTYPE* sample(double in,unsigned int sys_time_sek,unsigned int
sys_time_milli,double uncomp){

  static NOWTYPE temp;

  temp.sek=sys_time_sek;
  temp.milli=sys_time_milli;
  temp.in=in;
  temp.uncomp=uncomp;

  return &temp;
} /* sample */


/*************************************************************************/
/* Function to use when extracting the parameters from the tuning        */
/*************************************************************************/

PARAMETERS GetPar_tun(TUNER *tuner){
  return (*tuner).pars;
}


/*************************************************************************/
/* This function calculates the output of the tuning block. It is to be */
/* called directly after the call of sample                              */
/*************************************************************************/

double CalcOut_tun(STATE number,double step,NOWTYPE now){

  static double level;

  if (number == 1){
    level=now.uncomp;
    return level;
  }
  else if (number > 2 && number < 6 ){
    return (step+level);
  }
```

```
  else if (number <= 10){
    return level;
  }
  else{
    return 100;
  }
} /* CalcOut_tun */


/*********************************************************************/
/* This function is the main-function of the tuningblock. It is a kind  */
/* of state-machine, which each sample evaluates the current state, and */
/* if it is time, changes to the next.                                  */
/*********************************************************************/

STATE Update_tun(STATE number,double step,TUNER* values,NOWTYPE now){

  static unsigned int count,stime,stime_milli;
  static double maxtop,mintop,c_level,o_level,level37,o_high,Ms;
  static double ringbuff[index+1];
  int i;

  /* Initialization state */
  if (number == 1){
    if ((*values).sens > 0){
      Ms=2.0;
    }
    else{
      Ms=1.4;
    }
    stime=now.sek;
    maxtop=-10;
    mintop=10;
    c_level=now.uncomp;
    o_level=now.in;
    o_high=now.in;
    for(i=0;i<=index;i++){
      ringbuff[i]=100;
    }
    number=2;
  }

  /* Noise-checking state */
  else if (number == 2){
    if (now.in > maxtop){
      maxtop=now.in;
    }
    if (now.in < mintop){
      mintop=now.in;
```

```
  }
  if ((now.sek-stime)>5 || ((now.sek > 4) && (now.sek < stime))){
    (*values).tol=maxtop-mintop;
    number=3;
  }
}

/* Step-response-beginning state */
else if (number == 3){
  stime=now.sek;
  o_level=now.in;
  stime_milli=now.milli;
  number=4;
}

/* Measuring-the-deadtime state */
else if (number == 4){
  if (now.in > (0.6*(*values).tol+o_level)){
     (*values).L=(double)(now.milli-stime_milli)/1000;
     if ((*values).L < 0){
       (*values).L=(double)(MaxClock+now.milli-stime_milli)/1000;
     }
     count=0;
     number=5;
  }
  else if ((now.sek-stime) >  TimeOut){
     number=100;
  }
  else if (((MaxClock+now.sek-stime) > TimeOut) && ((now.sek-stime) < 0)){
     number=100;
  }
}

/* Measuring-the-staticgain state */
else if (number == 5){
  ringbuff[count]=now.in;
  o_high=o_high*alfa+(1-alfa)*now.in;
  if (count != index){
    if (fabs(ringbuff[count]-ringbuff[count+1]) < (*values).tol){
      (*values).Kp=(o_high-o_level)/step;
      number=6;
    }
  }
  else{
    count=-1;
    if (fabs(ringbuff[index]-ringbuff[0]) < (*values).tol){
      (*values).Kp=(o_high-o_level)/step;
      number=6;
    }
```

```
  }
  count=count+1;
}

/* Step-response-2-beginning state */
else if (number == 6){
  stime_milli=now.milli;
  o_high=o_high*alfa+(1-alfa)*now.in;
  level37=o_level+0.37*(o_high-o_level);
  number=7;
}

/* Measuring-the-deadtime2 state */
else if (number == 7){

  double L2;

  if ((o_high-now.in) > 0.6*(*values).tol){
    L2=(double)(now.milli-stime_milli)/1000;
    if (L2 < 0){
      L2=(double)(MaxClock+now.milli-stime_milli)/1000;
    }
    (*values).L=((*values).L+L2)/2;
    number=8;
  }
}

/* Measuring-the-timeconstant state */
else if (number == 8){
  if ((now.in-level37) <= -0.5*(*values).tol){
    (*values).T=((double)now.milli-((double)stime_milli+(*values).L*1000))/1000;
    if ((*values).T <= 0){
      (*values).T=(MaxClock+(double)now.milli-
                                  ((double)stime_milli+(*values).L*1000))/1000;
    }
    number=9;
  }
}

/* Calculating-the-parameters state */
else if (number == 9){

   double tau,a,L;                /* Local copys for readability only */

  (*values).a=(*values).Kp*(*values).L/(*values).T;
  (*values).tau=(*values).L/((*values).L+(*values).T);

  tau=(*values).tau;
  a=(*values).a;
```

```
    L=(*values).L;

    if ((*values).type == 0){                                    /* PI */
      if (Ms == 1.4){
        (*values).pars.K=0.29*exp(-2.7*tau+3.7*pow(tau,2))/a;
        (*values).pars.Ti=L*8.9*exp(-6.6*tau + 3.0*pow(tau,2));
        (*values).pars.beta=0.81*exp(0.73*tau + 1.9*pow(tau,2));
      }
      else if (Ms == 2.0){
        (*values).pars.K=0.78*exp(-4.1*tau + 5.7*pow(tau,2))/a;
        (*values).pars.Ti=L*8.9*exp(-6.6*tau + 3.0*pow(tau,2));
        (*values).pars.beta=0.44*exp(0.78*tau - 0.45*pow(tau,2));
      }
      (*values).pars.N=1;
      (*values).pars.Td=0;
      (*values).pars.Tt=(*values).pars.Ti/2;
    }
    else{                                                        /* PID */
      if (Ms == 1.4){
        (*values).pars.K=3.8*exp(-8.4*tau + 7.3*pow(tau,2))/a;
        (*values).pars.Ti=L*5.2*exp(-2.5*tau - 1.4*pow(tau,2));
        (*values).pars.Td=L*0.89*exp(-0.37*tau - 4.1*pow(tau,2));
        (*values).pars.beta=0.4*exp(0.18*tau + 2.8*pow(tau,2));
      }
      else if (Ms == 2.0){
        (*values).pars.K=8.4*exp(-9.6*tau + 9.8*pow(tau,2))/a;
        (*values).pars.Ti=L*3.2*exp(-1.5*tau - 0.93*pow(tau,2));
        (*values).pars.Td=L*0.86*exp(-1.9*tau - 0.44*pow(tau,2));
        (*values).pars.beta=0.22*exp(0.65*tau - 0.051*pow(tau,2));
      }
      (*values).pars.N=(*values).pars.Td*5;
      (*values).pars.Tt=sqrt((*values).pars.Ti*(*values).pars.Td);
    }
    number=10;
  }
  else if (number==10){
    if (fabs(now.in-o_level) <= (*values).tol){
      number=0;
    }
  }

  return number;

} /* Update_tun */



/****************************** End of Module ************************/
```

COMP.C

```
typedef struct{
  long double dblow,dbhigh;
  long double limit;
  long double k1,m1,k2;
  long double k3,k4,m4;
} COMPENSATOR;

void SetPar_comp(COMPENSATOR* com,double dblow,double dbhigh,double limit){
  (*com).dblow=dblow;
  (*com).dbhigh=dbhigh;
  (*com).limit=limit;
  (*com).k1=(-10-dblow)/(-10-(-(*com).limit));
  (*com).m1=dblow+(*com).k1*(*com).limit;
  (*com).k2=-dblow/limit;
  (*com).k3=dbhigh/limit;
  (*com).k4=(10-dbhigh)/(10-(*com).limit);
  (*com).m4=dbhigh-(*com).k4*(*com).limit;
}

double dbComp(COMPENSATOR* com,double input){

  double output;

  if (input < -(*com).limit){
     output=(*com).k1*input+(*com).m1;
  }
  else if (input < 0){
     output=(*com).k2*input;
  }
  else if (input < (*com).limit){
     output=(*com).k3*input;
  }
  else{
     output=(*com).k4*input+(*com).m4;
  }

  return output;

}
```