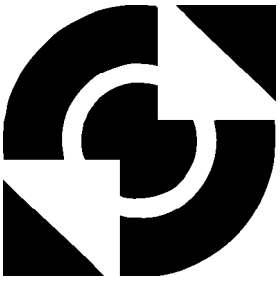
University of Twente

Faculty EE-Math-CS
Department of Electrical Engineering



Joystick Controller for JIWY

Tijs Lammertink

Individual Design Assignment

Supervisors dr.ir. J.F. Broenink
ir. G.H.Hilderink

June 2003

Report 011CE2003
Control Engineering
Department of Electrical Engineering
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

JIWY is a mechatronic device with two rotational degrees of freedom, with a camera as its 'end effector'. JIWY can be controlled with a joystick. Each joint is steered by a joystick axis. The x-axis and the y-axis control the horizontal joint respectively the vertical joint.

A model of the total system has been realized and simulated in 20-sim. C++ code is generated from the different controllers according to a new template. The code makes use of the CTC++ (Communicating Threads for C++) library, based on CSP (Communicating Sequential Processes). The code can be compiled for all sorts of platforms. The intention was to use a Real-Time Linux PC, but at the moment a MS-DOS PC is used, since Real-Time Linux encounters a problem.

A position controller and a velocity controller were made to control the position respectively the velocity of JIWY with the position of the joystick. The velocity controller seemed not very useful because of the end-stops of JIWY. However, for alignment of JIWY the velocity controller turned out to be very useful. The end-stops will be hit with a constant low speed.

Alignment is necessary, because the incremental encoders (measuring the position of JIWY) only measure relative displacement and cannot measure absolute displacement. The position controller is also used to home JIWY i.e. it returns to its alignment position when the controller terminates.

Samenvatting

JIWY is een mechatronische opstelling met twee rotatie vrijheidsgraden, waarop een camera geplaatst kan worden. JIWY kan dus bestuurd worden met een joystick, omdat die twee assen heeft. De x-as en y-as sturen respectievelijk de horizontale en verticale richting aan.

Een model van het totale systeem is gerealiseerd en gesimuleerd in 20-sim. Van de verschillende regelaars kan C++ code worden gegenereerd aan de hand van een nieuwe template. De code maakt gebruik van de CTC++ (Communicating Threads for C++) bibliotheek, gebaseerd op CSP (Communicating Sequential Processes). De code kan gecompileerd worden voor allerlei soorten platforms. De bedoeling was om een Real-Time Linux PC te gebruiken, maar op het moment wordt een MS-DOS PC gebruikt, omdat Real-Time Linux een probleem heeft.

Een positieregelaar en een snelheidsregelaar zijn gemaakt om respectievelijk de positie en de snelheid van JIWY te regelen met de positie van de joystick. De snelheidsregelaar bleek in eerste instantie niet erg nuttig vanwege de end-stops op JIWY. Echter, voor het uitlijnen van JIWY bleek de snelheidsregelaar wel degelijk nuttig, zodat de end-stops geraakt worden met een constante lage snelheid.

Het uitlijnen is noodzakelijk, omdat de incrementele encoders (die de positie van JIWY meten) alleen relatieve verplaatsingen kunnen meten en dus geen absolute verplaatsingen. De positieregelaar wordt ook gebruikt om JIWY te 'homen'. Dat wil zeggen dat hij zijn uitlijningspositie opzoekt wanneer de regelaar eindigt.

Contents

Preface

This project is part of the 3rd year of the Electrical Engineering study. It is called 'The Individual Design Assignment'.

I would like to thank my supervisors Jan Broenink and Gerarld Hilderink, because this project was not possible without their assistance. Among other things, Gerald assisted me on the code-generation part of this project and he came up with nice ideas about what to realize.

I also would like to thank Dusko Jovanovic for his help on this project. He assisted me on the modeling part of this project.

Enschede, June 2003

Tijs Lammertink

1 Introduction

JIWY is a little tabletop robot with two rotational degrees of freedom and a camera as its 'end effector' (see Figure 1). The goal of this project was to design a digital controller that drives JIWY with a joystick. The brand new CTC++ (communicating threads for C++) library had to be tested with this project. A model of the complete system has been made in 20-sim. This model can be used to simulate the system and optimize the parameters of the controllers.

From 20-sim version 3.2 code generation of models is possible. C++ code is generated from the controllers, according to a new C++ template in 20-sim, which can be compiled for all sorts of platforms (at the moment MS-DOS is used). This code has been made with the principles of CSP (communicating sequential processes) and it makes use of the CTC++ library. The CTC++ library makes it much easier to program event-driven and parallel control processes then in the sequential way.

Two controllers have been made: a position controller and a velocity controller. The position controller is used to control the position of JIWY with the position of the joystick. Because of the mechanical end-stops in both rotations, a joystick-controlled velocity controller is not very useful. However the velocity controller turned out to be very useful for aligning JIWY. Alignment is necessary because the relative incremental encoders cannot measure absolute position. Every time the controller starts, JIWY has to be aligned. Another feature that has been made is homing: when the position controller terminates, JIWY automatically homes to a predefined position.

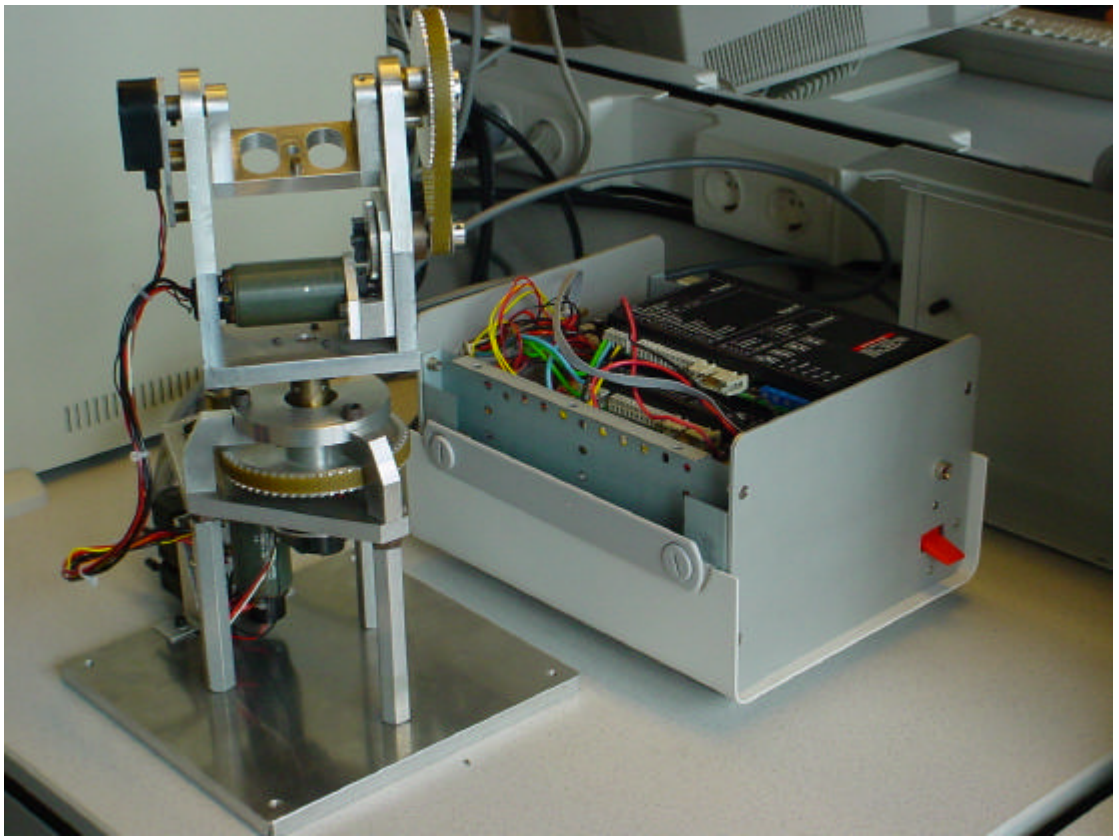


Figure 1 JIWY

The joystick controller for JIWY was first intended to run on a personal computer with RTLinux, but since there were several problems with making parallel processes, it now runs (temporarily) on another computer with MS-DOS.

RTLinux and MS-DOS have the advantage that they can achieve the time limits that are needed for hard Real-Time behavior. In contrast to MS-DOS, Linux is open source, so the kernels can be adjusted like one wants. RTLinux has a real-time kernel on top of Linux. The RTLinux kernel first schedules the RTLinux tasks as highest priority tasks and Linux has the lowest priority task.

The organization of the report is as follows. The architecture of the total system has been described in chapter 2. Chapter 3 is on modeling the total system: the top model, the IO-part and the bond graph are depicted. Chapter 4 describes the controllers that have been made: the Position controller, Alignment and Homing. The Code Generation part has been illustrated in chapter 5. The last chapter represents the conclusions and recommendations.

2 Architecture

The physical architecture of the total system shows two parallel joints that can work independently and simultaneously. See Figure 2 for what is called a CSP diagram. This diagram shows the communication graph as well as the composition graph. The three blocks at the bottom of the figure represent three processes that are executed sequentially, which is indicated by the '→' symbol. Each process consists of two parallel processes; the horizontal joint and the vertical joint. Only the position controller makes use of the joystick. The connections represent the nature of the time-related behavior. The joystick, the PC and the motors are parallel to each other, which is indicated by the '||' symbol. The arrows in the upper part represent the data-flow.

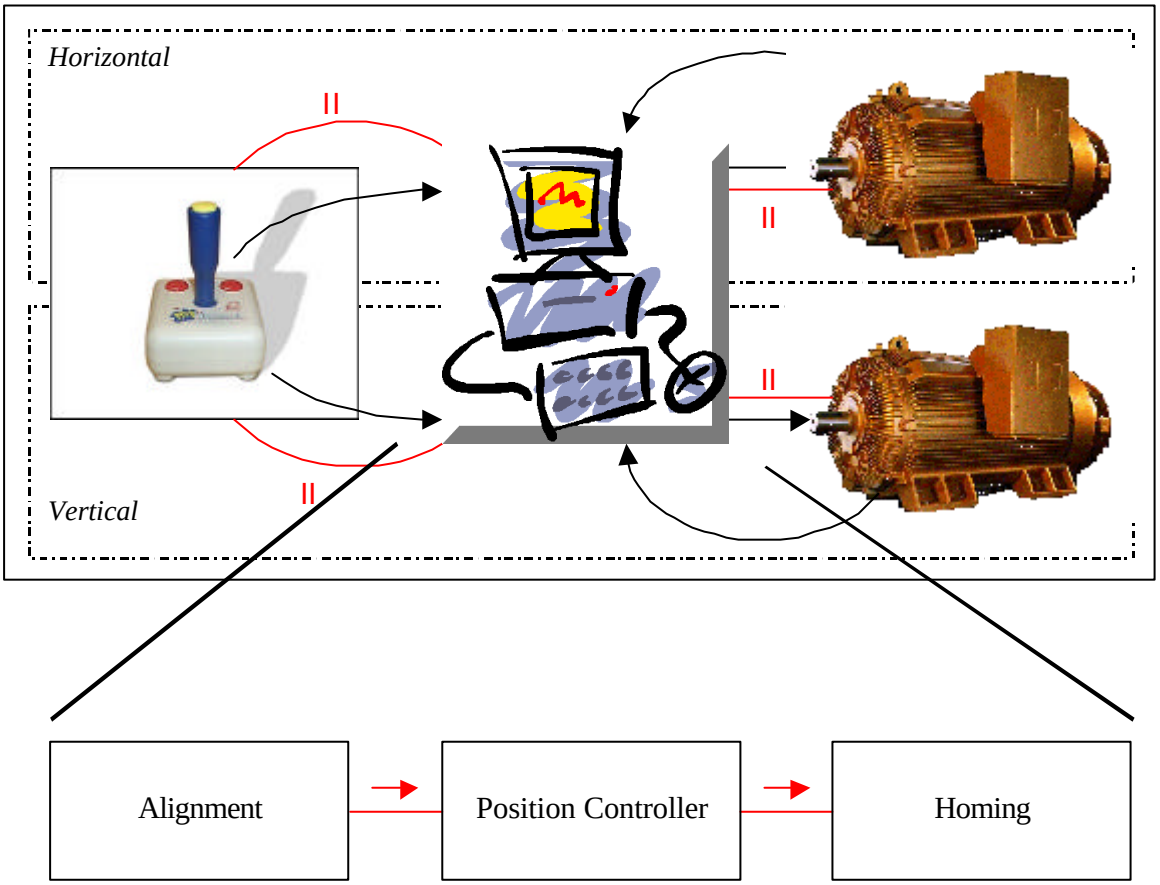


Figure 2 CSP Diagram

Since the motor and the joystick are physical objects, they will not be programmed as software objects. Therefore two parallel controllers with some inputs and outputs are identified. There are serial connections between the initialization, the controllers and the end. See Figure 3.

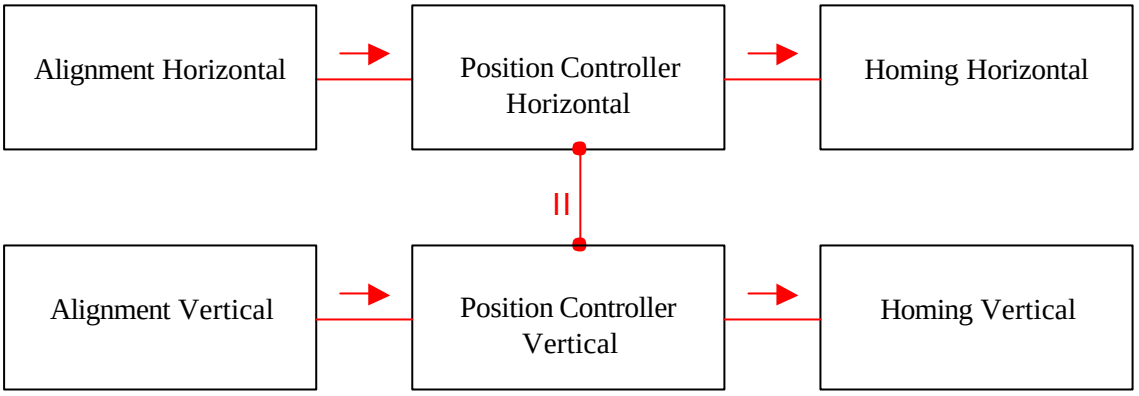


Figure 3 CSP Diagram

No data or whatever is flowing from one joint to another. We consider physical dependencies between the joints neglectable. Putting both systems into one sequential system has certain disadvantages. Therefore the controllers have been programmed in parallel. This has been realized with the use of the CTC++ library (Communicating Threads for C++). Some advantages of programming both systems in parallel are:

When one of the processes breaks down, the other one can continue working (like homing), so it becomes more robust.

The two parallel processes are both shorter and interleaved than one sequential process, so they will service events much faster.

You can use different sample frequencies for each parallel process, which is very hard to solve when all distinguishable processes are stuffed into one sequential process. Then, each process has to run as fast as the process with the highest sample frequency. That is not a very efficient way of processor usage.

Parallel processes are easier to expand then one sequential process. To expend a sequential process, the process itself has to be adapted and can result in dramatically structural changes. When using CSP, parallel processes can simply be added without changing the existing process(es) since CSP is compositional by nature.

In the beginning of this project, we were intended to use the Microsoft Sidewinder Force-Feedback Pro. Its force-feedback can be used to feed back external forces working on the setup.

Unfortunately no joystick driver exists that supports force-feedback for linux at the moment. When the force-feedback feature of a joystick does not work, it becomes very slack and therefore an analog joystick is used. The joystick that is used is the CH Flightstick Pro. A driver for this joystick is available for RTLinux and has been made for MS-DOS.

3 Modeling

3.1 Introduction

In the beginning of this project two models were needed for the same controller. One was used for simulation purposes and another one only for generating code. Dynamic link libraries (dll) were used to connect the inputs and outputs of the controllers to the hardware. These dll-files could not be used with simulations; so a second model was needed for code-generation.

Now, the principles of CSP (Communicating Sequential Processes) are used with JIWY. CSP does not consider a system from an object-oriented view, but from a process-oriented view. It is more natural to reason about real-time issues with processes than with objects. Processes are connected to each other via channels. The inputs and outputs of the controllers are therefore connected to channels, so the dll-files became superfluous. Moreover, just one model can now be used for simulations as well as for code-generation.

3.2 Top model

A model representation of the complete setup in 20-sim has been made. See Figure 5 for a top-level model of JIWY controlled by a joystick and with end-stop simulation. The x-axis and y-axis of the joystick are respectively used to control the horizontal and vertical joint of JIWY (see Figure 4).

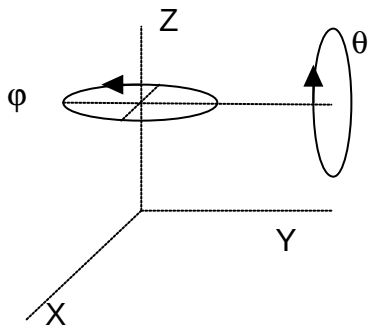


Figure 4 JIWY Rotations

The controllers have inputs from the joystick, the end-stops and the IO-part. The controller output modulates the current source that drives the motor via the NI6024E IO-card in the PC.

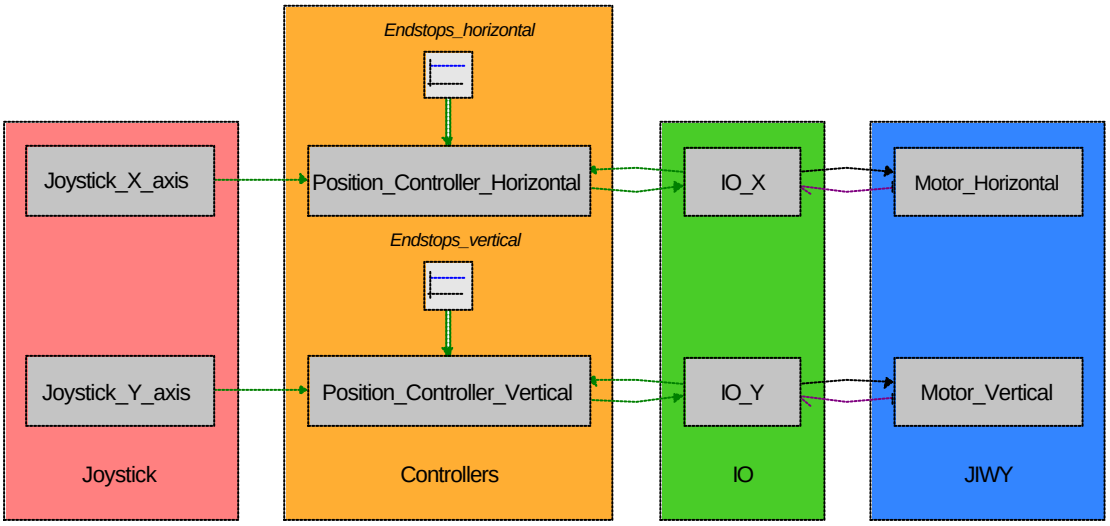


Figure 5 Joystick controller for JIWY

3.3 Input / Output (IO)

The IO-blocks in the 20-sim model contain all the parts that are in between the controller and JIWY, i.e. between software and physical system. See Figure 6 for what is in these blocks.

The DA-converter converts the output of the digital controller (the position- or velocity-error) to an analog value as an electrical current. The current will be amplified and fed to the motors.

The rotations of the axes are fed back via incremental encoders. These encoders divide one rotation of an axis into 2000 counts. The encoder in the model below is an equation model of the physical incremental encoder.

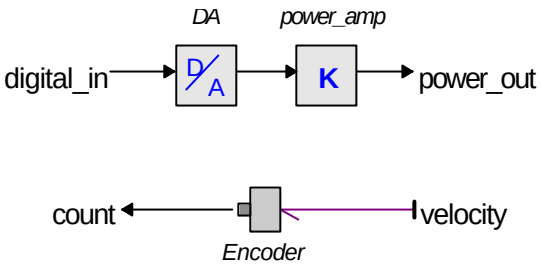


Figure 6 IO-block

3.4 Bond graph

A bond graph model of a joint of JIWY has been depicted in Figure 7. JIWY has two degrees of freedom so therefore JIWY can be modeled with two independent bond graph models. The electrical motors are driven by PWM (pulse width modulated) current sources, which are voltage-controlled. This means that the electrical resistances and inductances of both motors do not matter. If they instead where driven by current-controlled voltage sources, the electrical time constant could probably be ignored, because it lays far more away then the mechanical time constant.

The gravitational force does not affect the horizontal joint and will be neglected for the vertical joint.

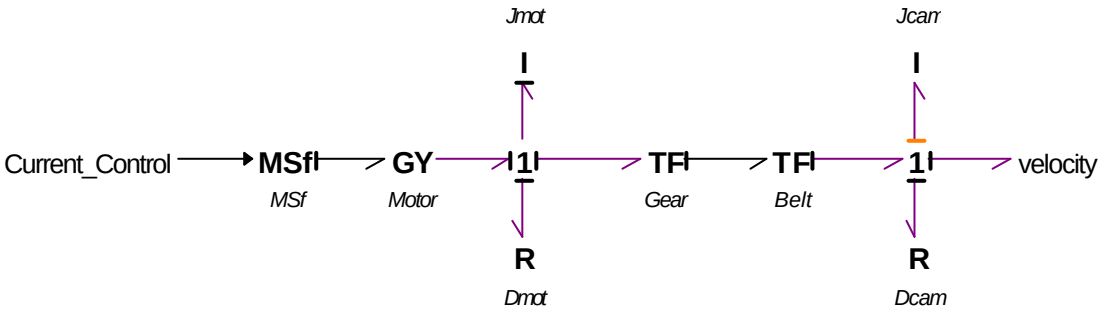


Figure 7 JIWY Bond graph horizontal

The inertias of the camera are not causal, because their connection with the inertias of the motors consists only of two gearings, so they are velocity dependent of each other (with a ratio of 1 to 20). This causality problem is not a big problem for simulations, because the 'backward differentiation' integration method can be used in 20-sim.

See Table 1 for the values of the bond graph components (A. Veltman (1988)).

Parameter	Value (horizontal)	Value (vertical)	Description
g	0.0394 N·m/A	0.0394 N·m/A	gyrator constant
J_{mot}	$2.63 \cdot e^{-6}$ kg·m^2	$2.63 \cdot e^{-6}$ kg·m^2	inertia of motor
D_{mot}	$1.77 \cdot e^{-6}$ N·m·s/rad	$1.77 \cdot e^{-6}$ N·m·s/rad	resistance of motor
gear	4	4	transformation rate
belt	5	5	transformation rate
J_{cam}	$4.5 \cdot e^{-3}$ kg·m^2	$3.0 \cdot e^{-3}$ kg·m^2	inertia of construction
D_{cam}	$1.35 \cdot e^{-5}$ N·m·s/rad	$1.35 \cdot e^{-5}$ N·m·s/rad	resistance of construction

Table 1 JIWY Parameters

The controllers are illustrated in the next chapter.

4 Controllers

There are two ways to control the setup. The first one is to control the position of JIWY with the joystick. The second one is to control the velocity of JIWY with the joystick. Because of the end-stops of JIWY, velocity control is less useful. However, for alignment it would turn out to be very useful. See Figure 8 for a position controller.

4.1 Position controller

4.1.1 Implementation

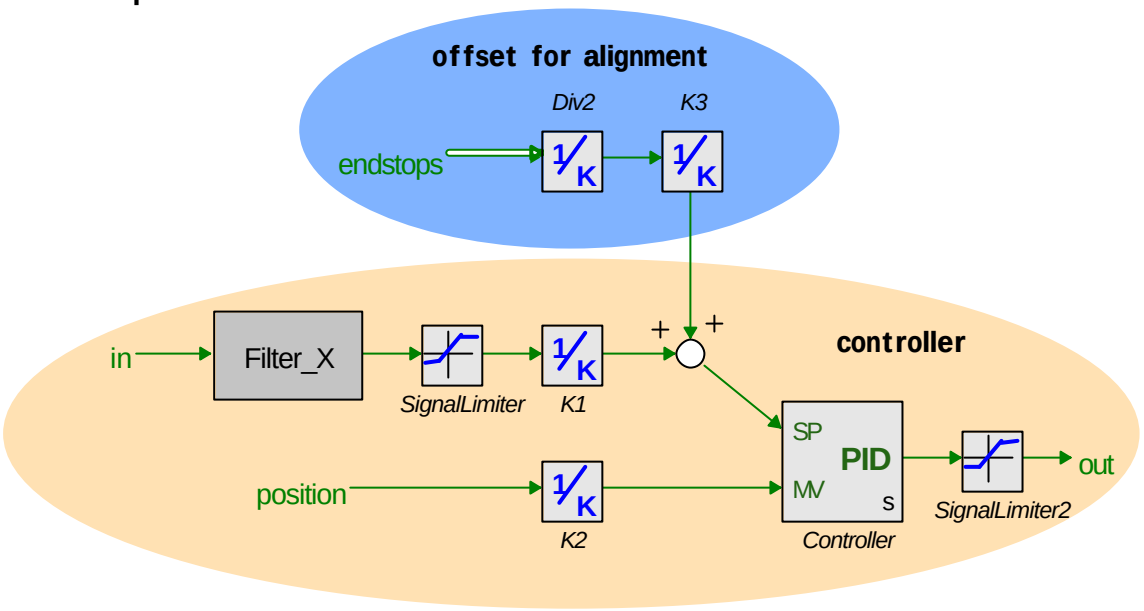


Figure 8 Position controller

A joystick has been connected to the input of the controller and provides the desired position for both joints of JIWY. A simple PID-controller is used to control JIWY. The position feedback is subtracted from the input, which results in a position-error. This error has to be steered to zero as fast and accurate as possible. The motors are driven by voltage-controlled current sources. The outputs of the controllers are connected to the inputs of the voltage-controlled current sources.

The joystick uses potentiometers to measure the x-y position. This method is not very accurate, because of the potentiometers and the logics of the joystick port in the PC. The boundaries of the axes are also not very accurate and do not always have the same values. Therefore we use a signal limiter to clip the boundaries at a certain value, so the joystick returns always the same value if it is pushed in his maximum position (positive or negative). To suppress noise, digital filters have been included that filter the output values of the joystick.

The *K1* and *K2* blocks are necessary to express the input and the position feedback in a same unit, otherwise they are not deductible. It is converted into radians, because that is the most natural unit. Moreover the position can be better recognized when it is expressed in radians.

The *K1* block converts the joystick input to the axis position in radians. In terms of radians the maximum rotation between the end-stops is 5,47 rad for the x-axis and 2,01 rad for the y-axis. (This has been measured with the incremental encoders). The joystick we are using has no straight transfer function between the real position of the joystick and its returned value, because of the analog potentiometers the joystick makes use of. The maximum returned values (i.e. when the joystick is pushed against the borders) vary around 450 till 550 in the

positive direction and around -450 till -550 in the negative direction. This holds for both axes. To get straight boundaries around the joystick axes, the signal limiters clip the joystick outputs at 450 and -450 (see Figure 9) which results in a fixed joystick range of 900.

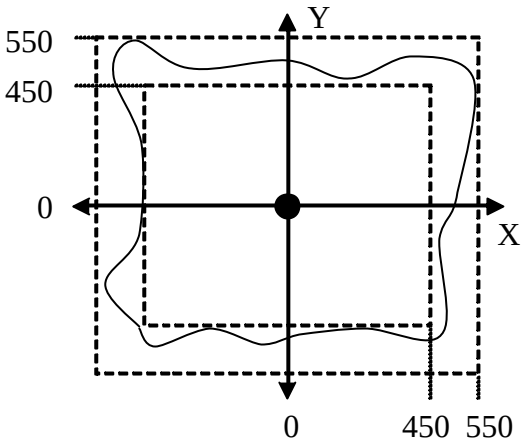


Figure 9 Joystick range

To ensure that the joystick ranges for both axes are linearly mapped and exactly fitted in the axes ranges, the conversion constants have to be the joystick ranges, divided by the axes ranges in radians:

$$K1_x = \frac{joystick\ range}{radians\ range} = \frac{900}{5{,}47} = 164{,}6$$

$$K1_y = \frac{joystick\ range}{radians\ range} = \frac{900}{2{,}01} = 447{,}8$$

These values have been rounded off upwardly to prevent hitting the end-stops.
The *K2* block converts the position feedback from the incremental encoders to the measured axes positions in radians. The incremental encoders divide a rotation into 2000 counts. So the conversion constants *K2* have to be:

$$K2_x = K2_y = \frac{encoder\ rotation}{rotation\,[radians]} = \frac{2000}{2p} = 318{,}3$$

The distance between the two end-stops of the x-axis and y-axis can be measured with a small program that just displays the output of the incremental encoders. These distances are respectively about 1740 and 640 counts.

See Table 2 for the constants.

		X	Y
Rotation Range	Joystick	900	900
	Encoder	1740	640
	Radians	5,47	2,01
	Degrees	313	115
Conversion Constant	K1	164,6	447,8
	K2	318,3	318,3

Table 2 Ranges and constants

The incremental encoders measure relative position and cannot measure absolute position, so the controller does not know the absolute position of JIWY. Therefore JIWY has to be aligned each time the controller (re)starts. The alignment controller aligns JIWY by finding its end-stops. The encoder-outputs at the end-stops are passed on to the position controller. In the 'div2'-blocks the offsets are calculated from these end-stops. These offsets will be added to the inputs from the joystick continuously. One way to align has been described below.

Generally JIWY starts at a random position, which will be called the initial position. From this position, steer JIWY with a constant velocity (direction does not matter) until it reaches its first end-stop and save the relative position at this end-stop (x_1). (This velocity cannot be too high, because then it would hit the end-stop to hard). After that, steer JIWY into the other direction until it reaches its second end-stop and save the relative position at this end-stop (x_2). Now, the distance d and the center c between the end-stops can be computed:

$$d = |x_1 - x_2| = |x_2 - x_1|$$

$$c = \frac{x_1 + x_2}{2} = \frac{x_2 + x_1}{2}$$

These computations show that the end-stops are equivalent, because they are interchangeable.

JIWY has fixed end-stops, so the distance between x_1 and x_2 is known if it is measured once. This information can be used to save time, because is not necessary that JIWY finds both end-stops to compute the distance in between, because this distance is already known.

So it may be sufficient to find one end-stop and then directly compute and find the center.

4.1.2 Digital filter

The analog joystick we use contains a lot of noise in its output signal. Closer analysis showed that the noise seemed to be digital spikes, which are generated by the counter logics of the analogue joystick input. Therefore a digital filter was designed to suppress this noise (D. Jovanovic). 20-sim has an integrated filter editor. With this editor a 4th order low-pass Butterworth filter was designed.

The transfer function of such a filter is:

$$H(z) = \frac{a_0 z^4 + a_1 z^3 + a_2 z^2 + a_3 z + a_4}{b_0 z^4 + b_1 z^3 + b_2 z^2 + b_3 z + b_4}$$

The parameters are dependent on the sample-frequency $\boldsymbol{F_s}$ and the cut-off frequency $\boldsymbol{F_c}$.

The parameters of this transfer function are shown in Table 3.

Numerator		Denominator	
Fs	100	Fc	2
a0	0.000416599204407	b0	1.0
a1	0.001666396817626	b1	-3.180638548875
a2	0.002499595226439	b2	3.861194348994
a3	0.001666396817626	b3	-2.112155355111
a4	0.000416599204407	b4	0.438265142262

Table 3 Filter parameters

When the sample-frequency of a joint is changed, another filter has to be designed, because the parameters of the filter depend on it.

4.1.3 Simulation

From simulation results concluded is that a PID controller seemed to control JIWY well enough. The optimal derivative time constant t_d for this controller depends on the inertias and dampers connected to JIWY. For example, if you attach a camera then t_d has to be adjusted to minimize overshoot. Lowering t_i only worsens the response. Figure 10 and Figure 11 show the response of the horizontal respectively the vertical part of the model. The parameters of the controllers and end-stops are listed in Table 4.

Parameter	Value (horizontal)	Value (vertical)	Description
K	300	900	proportional gain
τ_d	0.4	0.22	derivative gain
N	10	10	tameness constant
τ_i	1000	1000	integral gain
end-stop[1]	-1100	240	
end-stop[2]	640	-400	

Table 4 Parameters

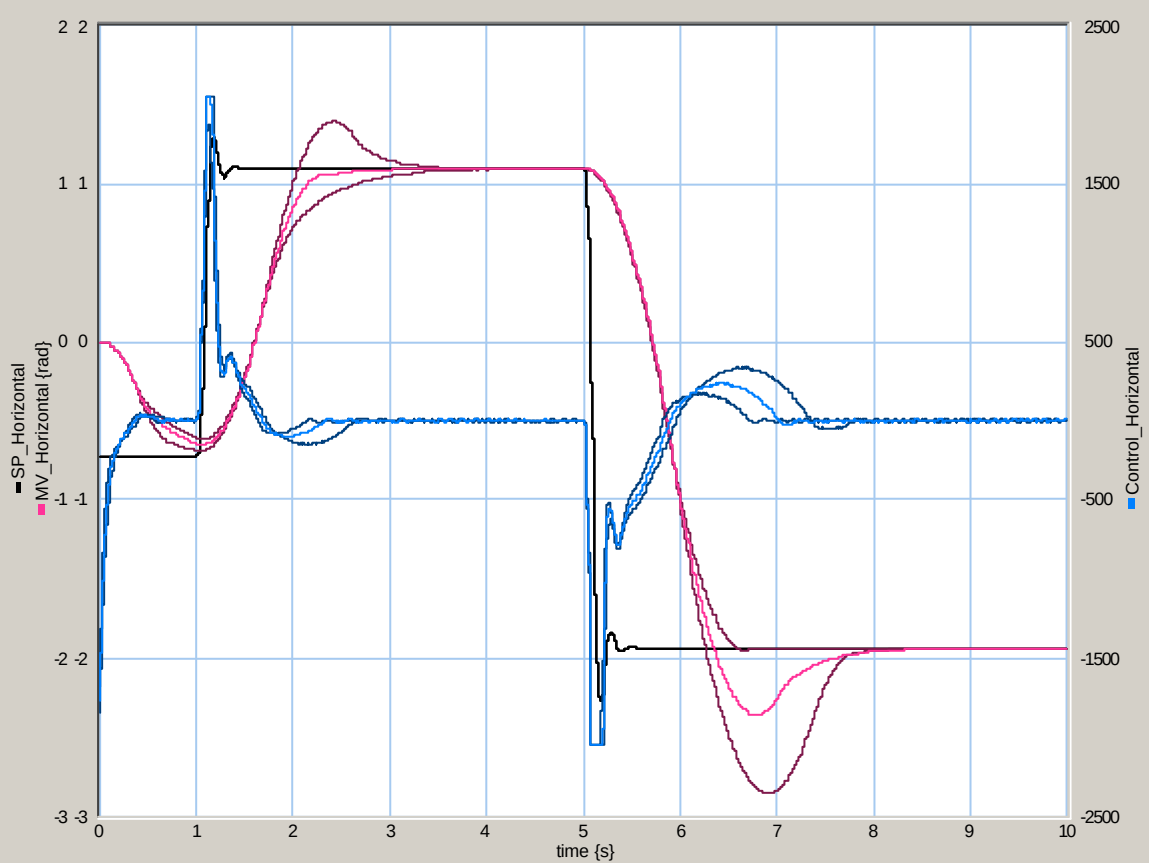


Figure 10 Simulation of Position Controller (Horizontal Part)

The joystick starts at a value of 0. First, JIWY will be aligned, according to the values of the end-stops. At t = 1 and t = 5 the horizontal axis of the joystick takes a value of 300 and -500. These step-functions are filtered by the digital filters. In the simulation the joystick output and the position feedback are plotted in radians since that will make it easier to compare.

Three simulations are made with different t_d (0.3, 0.4 and 0.5). For the first step-function, 0.4 seems a good value, but for the second step 0.5 is better. Since such steep functions are less probable, t_d = 0.4 has been chosen.

For the vertical joint simulations are made with different t_d (0.15, 0.22 and 0.3). t_d = 0.22 seems to be the optimal value.

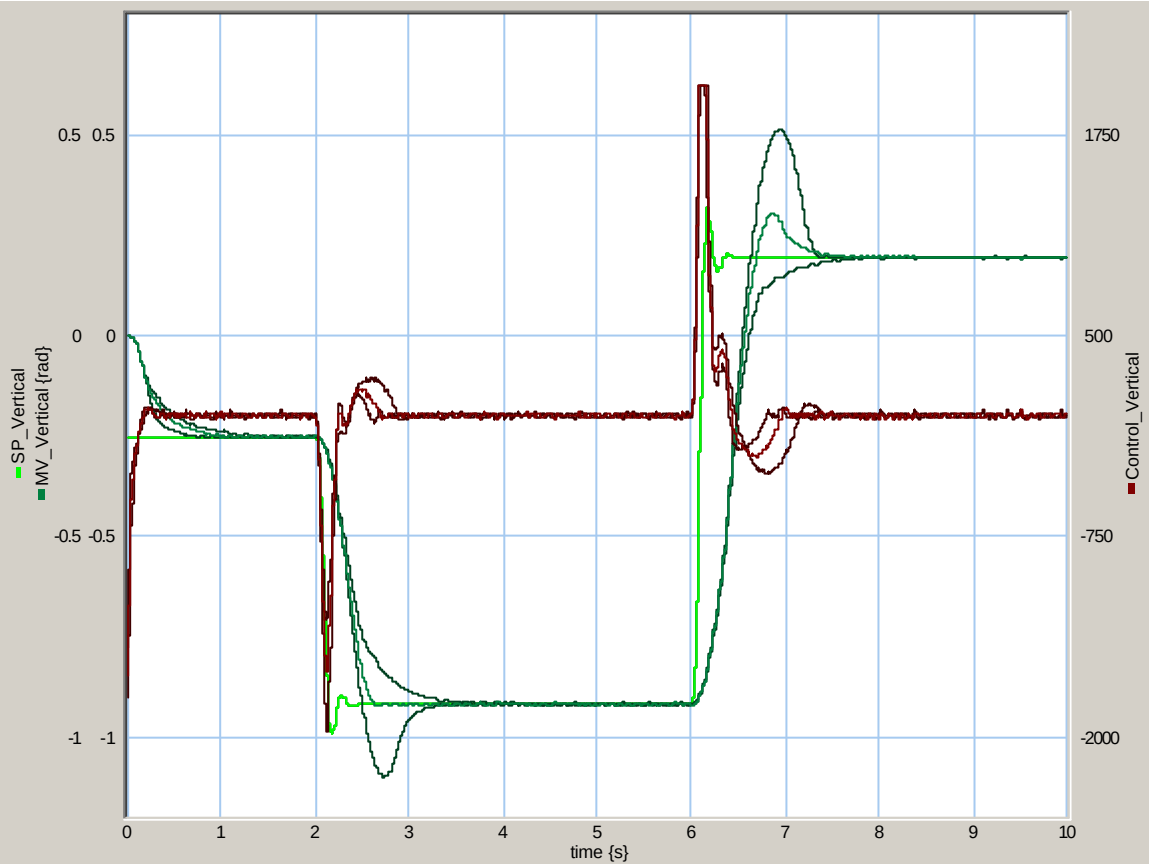


Figure 11 Simulation of Position Controller (Vertical Part)

Again the joystick starts at a value of 0 and after that JIWY will be aligned. At t = 2 and t = 6 the vertical axis of the joystick takes a value of -300 and 500.

Normally controllers are optimized in such a way that the outputs of the controllers do not exceed the maximum input value of the DA-converters. Otherwise the current through the motors will be clipped, which results in non-linear behavior. The block 'SignalLimiter2' models that behavior.

In this case the controllers are optimized in such a way that the outputs of the controllers do exceed the maximum input value of the DA-converters for a short time during the simulation. Clipping happens mainly when step-functions are used for the joystick. According to the simulations clipping for a short time does not result in unstable and/or unwanted behavior in the position of JIWY. Since step-functions are not very likely as a joystick output, the controller output does not clip often in reality. Tests have shown that clipping for a short time does not result in unstable and/or unwanted behavior. So the advantage remains that JIWY becomes faster and much stronger.

4.2 Alignment

4.2.1 Introduction

The incremental encoders measure relative position and cannot measure absolute position, so the controller does not know the absolute position of JIWY at turning on the system. Therefore JIWY has to be aligned on power on. This is to ensure that the positive and negative ranges of the rotation have the same length. If JIWY is not aligned when the controller starts running, it may become unstable or hit an end-stop, which may damage JIWY.

Aligning can be done in two ways: manually or automatically. Manually aligning will not be very accurate compared with automatically, because the encoders can measure the position of JIWY much more accurate then a human ever could. So a nice way of alignment would be automatically.

4.2.2 Implementation

Alignment has been implemented in four steps.

From the initial position, use the velocity controller to rotate left until the end-stop is reached.

Use the position controller to steer back to the initial position. That position is where the relative encoders return zeros.
The reason why the 'rotate right' model will not be executed right after 'rotate left' is only to safe time. At the moment the first end-stop has been found, you know that the second end-stop will lie somewhere behind the initial position. So you can steer with a high velocity back to the initial position and then look for the second end-stop at a lower velocity.

From the initial position, use the velocity controller to rotate right until the end-stop is reached.

Start the position controller, which initially finds the center, because it continuously adds an offset to the input of the joystick.

These steps will be executed sequentially for each joint. However, these two sequential paths can be executed parallel.

A model of the alignment controller has been depicted in Figure 12.

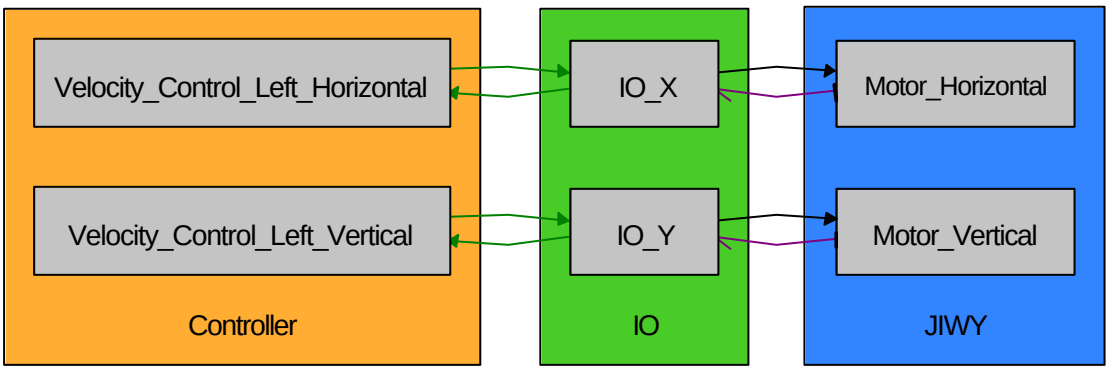


Figure 12 Alignment

See Figure 13 for the contents of the alignment controller. The only difference between the left and right is the direction of the velocity (sign of the constant). Again a PID-controller is used.

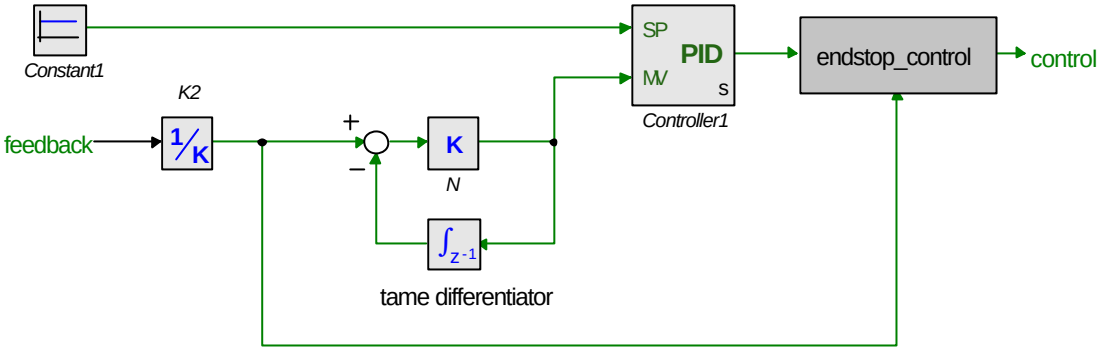


Figure 13 Alignment velocity controller

The position feedback will be differentiated to velocity with a tame differentiator; otherwise the numerical noise will be amplified too much.

4.2.3 Simulation

Virtual end-stops have been implemented to simulate the real end-stops. These are located in the IO-submodel (see Figure 14), because the end-stop is realized in hardware and therefore no code has to be generated from it. As can be seen from the simulation (see Figure 15) a virtual end-stop will make the velocity go to zero when the encoder output equals a randomly chosen value of 1500.

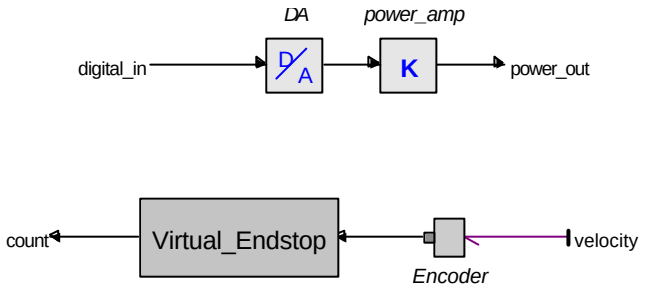


Figure 14 IO

See Figure 15 and Figure 16 for a simulation of the horizontal respectively the vertical velocity controller, turning left. It turns left until the end-stop is reached. See Table 5 for the parameters of the controller.

Parameter	Value (horizontal)	Value (vertical)	Description
K	400	400	proportional gain
τ_d	0.04	0.04	derivative gain
N	10	10	tameness constant
τ_i	1000	1000	integral gain
v_{left}	5	5	
v_{right}	-5	-5	

Table 5 Parameters

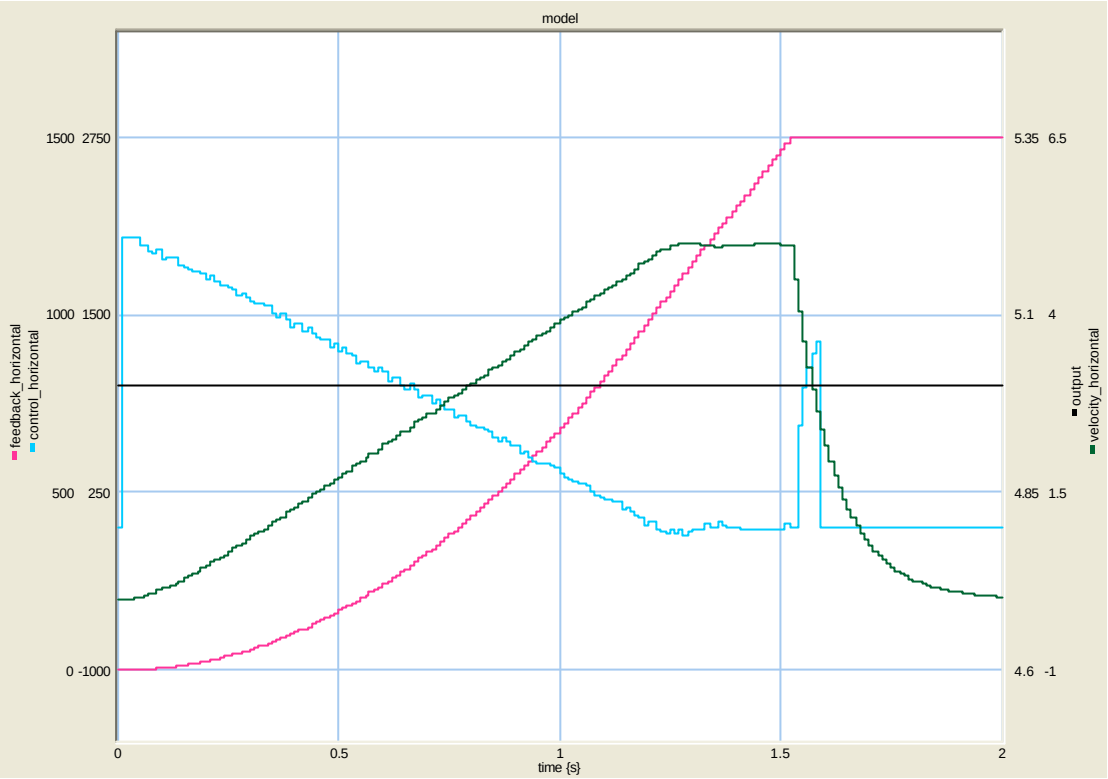


Figure 15 Simulation of Velocity Controller (Horizontal Part)

A velocity of 5 rad/s is reached after more than 1 second. In about 1,5 second the end-stop is hit.

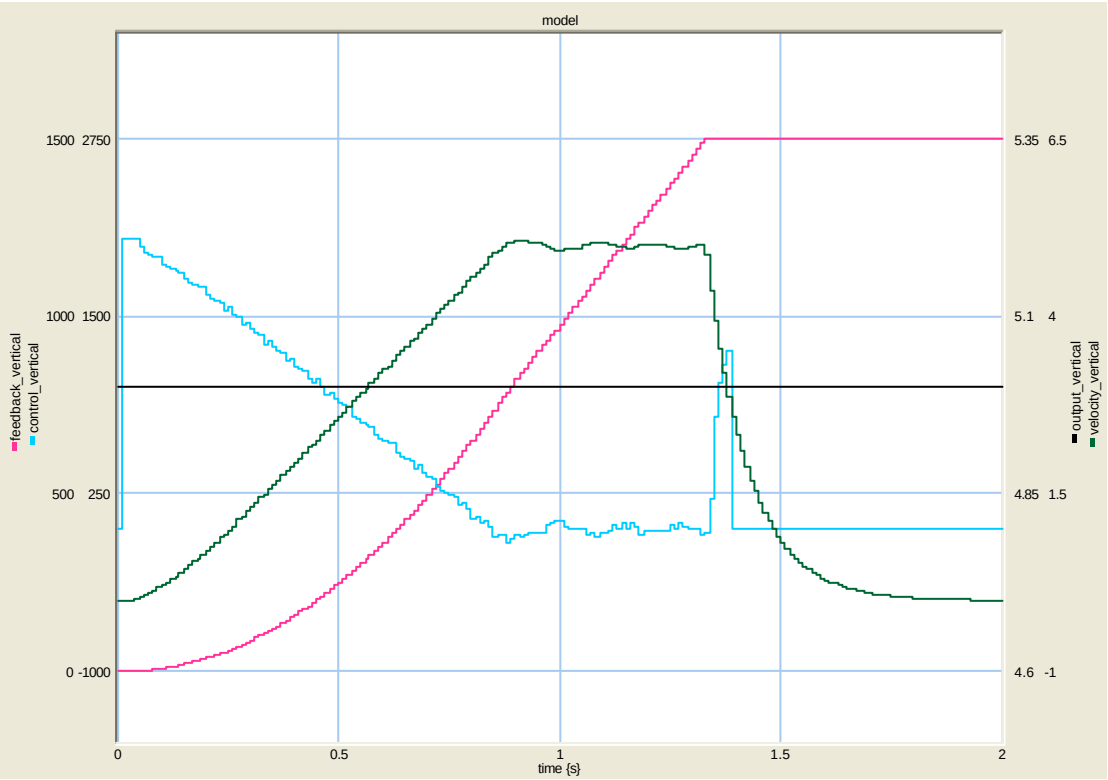


Figure 16 Simulation of Velocity Controller (Vertical Part)

The vertical velocity controller is a little bit faster, because the inertia is smaller.

Like the position controllers, these controllers are also optimized in such a way that the outputs of the controllers just exceed the maximum input value of the DA-converters.

The velocity controller that turns right is exactly the same as the one that turns left, except for the sign of the velocity. So a simulation of that one will not be given. Possible little differences in the bond graph for turning left and turning right are neglected. These differences may exist, because the cables between the turning parts and the fixed world may result in different frictions.

4.3 Homing

When the stop button of the joystick is pressed, JIWY automatically homes, before the controller terminates. In this case that will be the center. This could easily be implemented: Just use the position controller and connect the center values to the input channels instead of the joystick. So no other model had to be made. For this assignment it may be a bit superfluous, but for big machines in the industry homing may be very important.

The total software architecture that is made out of the position controller, alignment and homing is illustrated in the next paragraph.

4.4 Software architecture

See Figure 17 for the software architecture of the total system as it is implemented. There are two parallel processes, which consists both of five sequential processes. The position controller is connected to the joystick button via the alternative operator. This means that per step a random choice will be made which process is executed. Alignment passes the end-stop values on to the position controller via parameter transfers.

Code will be generated from these processes. This is explained in the next chapter.

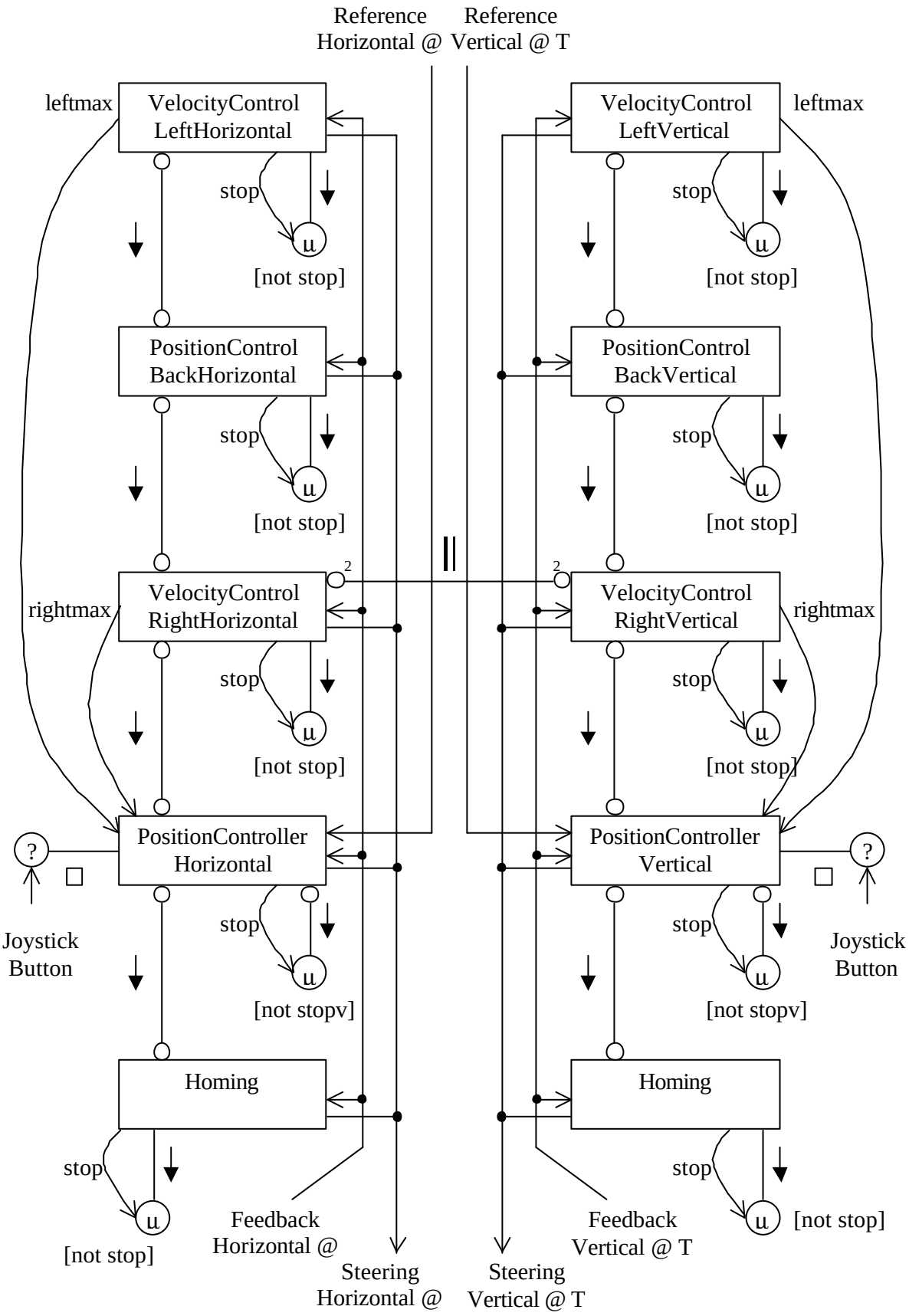


Figure 17 Software Architecture

5 Code Generation

5.1 Introduction

In this project code will be generated for each controller. Figure 18 shows how a digital controller is realized, starting with a 20-sim model.

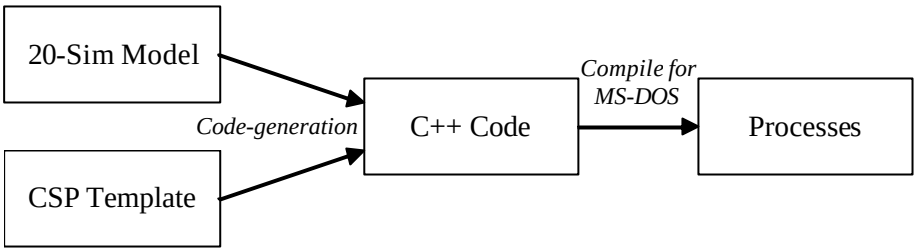


Figure 18 Code Generation

5.2 Template

20-sim uses templates to generate code of a model or a part of it. 20-sim 3.2 comes with four templates: stand-alone C (whole model or a submodel), a C function, or a Simulink S function. But you are also free to make your own templates. 20-sim makes use of so called *'tokens'*. A token is a placeholder for model-dependent information. For instance in the code generation diabg, the target destination directory contains the name of the selected submodel by default. Since this information is not yet known when the targets.ini file is created, a specific token that refers to this name is used instead.
To make use of CSP to program the controllers, a new template has been made. The existing submodel-template could be used to generate code of only one submodel. The CSP-template had to be extended such that it can generate code for one or more submodels, located in one model. An important change that has been made therefore is that the name of the submodel in the code of the template had to become a token instead of a predefined name. All global functions and variables now have the token %SUBMODEL_NAME% in their name to care for unique names.

The CSP-template is still under construction, because at the moment 20-sim has a few shortcomings with generating code for a system, which will make use of CTC++ (Communicating Threads for C++):

- Lack of information to generate processes in a generalized way.
- The impossibility of modeling a system with a hierarchical overview for its different controllers.

At the moment you have to connect the ports of the model in 20-sim to the channels by hand in the source code. The ports namely get numbers that are determined by the code generation. Then the information of the names of the ports is lost. It would be much easier if the ports of the model in 20-sim will automatically become channels in C++.

The different processes (in this case: alignment, controlling by joystick and homing) cannot easily be put in one overall 20-sim model. If you have different controllers in one system (in this case: JIWY) and want to change something (like names or parameters) in the system, then you have to adapt all the models every time. This costs a lot of time and moreover it is sensitive for mistakes. Also code-generation costs too much time. For each controller, the code has to be generated separately.

A nice feature for the future of 20-sim may be to support a hierarchical project structure with different controllers for the same system. Then it becomes possible to easy switch between models and so on.

The generated C++ code can be compiled for all sorts of platforms. At the moment a pc with DOS is used. But also a pc with RTLinux can be used or for example a DSP.

5.3 Hierarchy

It is important to think about the hierarchy of the source-code and the names of models and subdirectories. One reason is that the overview gets lost very easily.

Every submodel will return in a separate class in C++, so spaces in names of submodels are not allowed. Also 20-sim does not allow spaces in names of models. Therefore the choice has been made to use underscores instead of spaces for all names. The underscores are not very beautiful, but once such a choice has been made, changing all the names costs too much time.

Different models are used for the position controller and for alignment, so two subdirectories are made. The code of Homing has been placed in another directory. It makes use of the position controllers from *\JIWY\Position_Controller*. JIWY has two joints, so every controller is implemented twice. Therefore each controller always has two subdirectories (horizontal and vertical). Alignment has been divided in turning left and turning right, because they could not be put in one overall model. This leads to the following directories:

- *\JIWY\Position_Controller\Position_Controller_Horizontal*
- *\JIWY\Position_Controller\Position_Controller_Vertical*
- *\JIWY\Alignment\Velocity_Control_Left_Horizontal*
- *\JIWY\Alignment\Velocity_Control_Left_Vertical*
- *\JIWY\Alignment\Velocity_Control_Right_Horizontal*
- *\JIWY\Alignment\Velocity_Control_Right_Vertical*
- *\JIWY\Homing*
- *\JIWY\Common*

The controllers make use of common code, which is located *\JIWY\Common*. But each controller also has its own C++ code. These files are stated below and are located in each directory except for the common directory.

- *%SUBMODEL%.cpp*
- *%SUBMODEL%.h*
- *%SUBMODEL%process.cpp*
- *%SUBMODEL%process.h*

The token of a submodel in 20-sim is %SUBMODEL% and it will be used for the names of the source code files for each controller to take care for unique files. The %SUBMODEL%.cpp file contains the calculations of the model and the %SUBMODEL%process.cpp file contains the declaration and the execution of the process.

The file *main.cpp* is located in *\JIWY*. This file has to be written manually, because it contains the declaration of processes and channels. In this file the different processes are executed in parallel with the following statement:

```
Parallel *par = new Parallel;
```

Two sequential declarations are made. One for the horizontal joint and another one for the vertical joint. This with the following statement:

```
Sequential *seq = new Sequential;
```

Each sequence contains five processes. They are added with the following statement:

```
seq->add(process);
```

The different sequential processes are added to one parallel process:

```
par->add(seq);
```

6 Conclusions and recommendations

6.1 Conclusions

The step-wise refinement approach works. Verification by simulation of the model and validation and testing of the realization seems to be a good design trajectory for the realization of real-time parallel controllers.

CSP can be used very well to consider control systems. Two controllers can easily be programmed as two parallel processes with the CTC++ library. It is also easy to expand and add new processes, because existing processes do not have to be changed. The CTC++ library is like a higher abstraction level, which makes it easier for the user to program parallel processes.

The code-generation tool of 20-sim is a nice way to realize digital controllers, because the controllers can first be tested by simulation in 20-sim. Even if just a little is known about programming languages, digital controllers can be realized that are made of C++ code.

6.2 Recommendations

There is always a difference between the model and the real system. Among other things, the parameters of the components of the bond graph are not totally right and gravitation effects have been neglected. It is difficult to see if the controller behaves like the model does. For exact validation of the controllers, the positions of the joystick and JIWY and the controller-output should be saved to a file. This data can then be compared with the simulation. For example by exporting the data from the simulation to Matlab and plotting it together with the data from the file.

If joystick drivers for linux become available that support force-feedback, this feature can be used to feed back external forces that are working on JIWY. The force-feedback feature should be put in new processes to let the existing processes intact.

References

Amerongen, J. van and Vries, T.J.A. de (1998), *Digitale Regeltechniek*, Universiteit Twente, Enschede

Breedveld, P.C. and Amerongen, J. Van (1994), *Dynamische Systemen: modelvorming en simulatie met bondgrafen,* Open Universiteit, Heerlen, 90 358 1302 2

Broenink, J.F. and G.H. Hilderink (2001), A structured approach to embedded control systems implementation, *2001 IEEE Conference on Control Applications, Sept 5-7*, Mexico

Hilderink, G.H., (2001), *Communicating Threads for C++ - A White Paper*, Faculty of Electrical Engineering, Control Engineering, University of Twente, Enschede, The Netherlands.

Jovanovic, D, Hilderink, G.H, Broenink, J.F, (2002), *A Communicating Threads case study: JIWY,* Communicating Process Architectures 2002, pp. 231-330. Reading, United Kingdom

Groothuis M. (2001), *20-Sim code generation for PC/104 target,* Faculty of Electrical Engineering, Control Laboratory, University of Twente, Enschede, The Netherlands

Stephan R.A. (2002), *Real-time Linux in Control Applications Area,* Faculty of Electrical Engineering, Control Laboratory, University of Twente, Enschede, The Netherlands

Veltman A. (1988), *Regelen en modelleren van een 2-assige camera-opstelling,* Faculty of Electrical Engineering, Control Laboratory, University of Twente, Enschede, The Netherlands