ISSN 0280-5316
ISRN LUTFD2/TFRT--5818--SE

Controlling a Robot with Two Force Sensors in a Lead-Through Programming Scenario

Erik Gustafsson

Department of Automatic Control
Lund University
June 2008

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* June 2008 *Document Number* ISRN LUTFD2/TFRT--5818--SE
Author(s) Erik Gustafsson	*Supervisor* Anders Robertsson Automatic Control, Lund Rolf Johansson Autmatic Control, Lund (Examiner) *Sponsoring organization*

Title and subtitle

Controlling a Robot with Two Force Sensors in a Lead-Through Programming Scenario (Robotstyrning med två kraftsensorer i scenario för intuitivt instruerad programmering)

Abstract

To simplify the programming of industrial robots a method called lead-through programming has been developed where the programmer basically leads the robot using his/her hands. The objective of this thesis has been to see how a safety system (protecting the tool from large forces) for a lead-through programming controller could be achieved by using an extra force sensor mounted at the tool. A modified controller is presented with a description of its implementation, simulations results and test results from running on a real robot.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 130	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/

Preface

When I first started to look for a master thesis project I was not quite sure what I wanted to do. Since I had chosen mechatronics as my main subject and in that had taken classes in automatic control and robotics I decided to ask Anders Robertsson if he had any suggestions for possible topics. One of the topics he suggested was to modify an existing force controller for a robot. This controller was intended to be used in a lead-through programming scenario. It had the signals from one force sensor as input from the programmer to control the robots movements. Now it was to be complemented with one extra force sensor that was going to be used for protecting the tool of the robot. I thought the suggestion sounded interesting and thus I decided to choose this as the topic for my thesis.

When I have done my project I have had a lot of help and support from people around me. First of all, I want to thank Anders Robertsson who has been my supervisor during this project for his help and support and also for the idea of the project. Next, I want to thank the computer administrators and the rest of the people on the automatic control department who have assisted me during my work. A special thanks goes out to Silvia for supporting me both when things have been going well and when they have not. Thanks to all the people at the corridor and to the rest of my friends for taking my mind off my thesis on my spare time. And last, I want to thank my family.

Contents

1. Introduction

Chapter overview
This chapter will give a basic introduction to the subject of this thesis. It will explain why the subject of lead-through programming is interesting for companies today, explain what the purpose of this thesis is and motivate the need for doing this thesis.

After the introduction to the subject of this thesis, the methods that have been used in this project are briefly discussed.

1.1 Introduction to subject

Traditionally, robots have been a type of machine that mostly has been suited for larger companies, or companies making long series of products. This has, at least partly, been due to the fact that it has been a little cumbersome to program the robots. In the case of a large company that is going to use the same program on the robot for quite some time, the time overhead for the programming, and also maybe the cost of hiring specialist to perform the programming, has been justified.

In industry today automation is becoming more and more important. Some people even talk about automation as a second industrial revolution [1]. For smaller companies to also benefit from the advantages of automation, and in this particular case robots, the robots need to be adopted for this type of company. One way of doing this is to simplify the procedure of programming the robot. By doing that, robots could be used even in production of smaller series and for single specific tasks.

One approach of simplifying the programming is called *lead-through programming.* The main idea of this concept is for the programmer, or operator, of the robot to basically grip a handle, or something similar, on the robot (that is attached to a force sensor) and to then just lead the robot along a trajectory, to different points of interest or over a surface and in that way show the robot how to perform the task. This strategy has the advantage of being intuitive, easy and fast, thus well suited to fit the needs of smaller companies.

1.2 Motivation

When performing the lead-through programming, the robot is basically only aware of what the programmer tells it to do. A situation that might occur due to this fact, is that the programmer pushes the robot towards a wall or some object. When the robot has gained contact with this wall/object it will still not have any notion of it being there, only what the programmer tells it to do. So basically if the programmer keeps pushing the robot towards the wall/object, the robot will just keep trying to go through it. This can lead to that forces are build up on the robot.

If the robot would be equipped with some tool, maybe a fragile type of tool, one would not want large forces to be put on this tool. Therefor there is a need for a sort of protection system for the tool of the robot in lead-through programming.

The situation where the robot operator pushes the robot into a surface could be rather common in the process of lead-through programming. Imagine for example that the operator wants to measure a surface to be grinded by the robot. In that situation there is a need to guarantee a good contact with the surface while measuring it, but to still avoid the risk of damaging the tool of the robot.

1.3 Problem definition

The subject of this thesis has been to look on how an existing lead-through programming controller can be *complemented with an extra force sensor mounted right before the tool of the robot*. This extra force sensor would then be used to change the control in a way so that large forces on the tool are avoided. In a sense this extra force sensor would thus be a *safety sensor*.

1.4 Methods

It was decided that the work would be implemented on a modified robot system located in the robot lab at the Department of Automatic Control, LTH. This modified system allowed the user to implement controllers in Matlab Simulink and to then build them with Real Time Workshop to a version that could be loaded into the robot system. The existing lead-through programming controller was already implemented on this modified system.

Since the controller was supposed to be constructed in Matlab Simulink it was also decided that a nice way of testing the controller would be to simply simulate it in Matlab Simulink. To do this, a model of the robot was needed, and that was already available from a previous project. Two blocks that could simulate inputs on the force sensors were developed, signals were set to be logged and a script was written that could run series of simulations and save different graphs from these simulations.

The controllers have been implemented in a Matlab Simulink S-function Builder Block. Basically this block give the user a nice interface for building a Matlab Simulink S-function. For example, it can automatically generate a wrapper TLC file that is needed for building the controller with Real Time Workshop. The S-functions used are discrete and the discrete update part is written i C-code. One can also define external help-functions that can be called in the discrete update function. These functions also needs to be written in C-code.

The controller model contains some more parts where it has been useful to be able to write code. For these more simple cases it has felt unnecessary, or cumbersome, to use a S-function. Instead a type of block that can run embedded Matlab code has been used in these cases. Obviously, the code in these blocks has been written in Matlab syntax.

The strategy in the development stage has been to first have a brainstorm session where different possible ways of how to connect the second force sensor to the control were considered. After the brainstorm session the resulting concepts (two different) were developed simultaneously. They were implemented, simulated and developed until a working version was reached in the simulations. At that time, the controller was tested on the real robot and the results from these tests were analyzed. This development cycle was then repeated and the controllers were improved several times before a final working version was reached.

2. Robot theory

Chapter introduction

The first part of this chapter will give a brief introduction to the basic concepts of an *industrial robot (IRB)*. The purpose is to give a reader who have no knowledge of IRBs enough base to follow the rest of the report. For a more complete introduction to IRBs, it is suggested for the reader to look into, for example, [1] or [2].

The second part of this chapter will give some examples of different types of force control. The purpose besides giving examples is also to introduce the reader to how force control is achieved in this project. For more information about this subject, the reader can be referred to, for example, [3].

2.1 A short introduction to IRBs

The term industrial robot, or IRB, that is used in this report refers to the nowadays classical robot type that can be seen in Fig 2.1. The configuration of such a robot can differ some but it usually consists of *an arm and a wrist* mounted at the end of the arm. On the wrist different tools can be mounted, possibly tools with more *degrees of freedom (DOF)*. The purpose of the arm part of the robot is to position the tool and the purpose of the wrist part is to orient the tool at this position. Usually the arm has at least three DOF (think x-y-z directions) which it needs to fully locate the tool in space and in turn the wrist has three DOF in order to fully orient the tool [1]. Of course the robot can have less than six DOF which would put limits on how the robot can be oriented or positioned. It could also have more than six degrees of freedom which would give the robot a kinematic redundancy and thus more possibilities to move around objects etc [1].

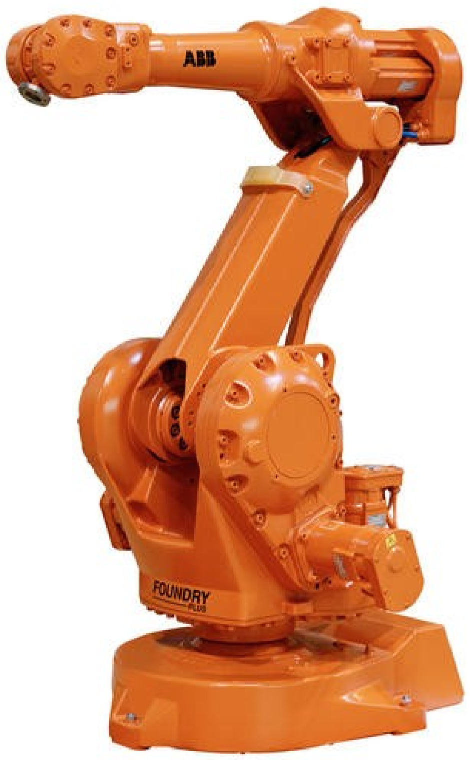


Fig 2.1: A typical industrial robot [6]. This is the type of robot that has been used in this project

A robot can be seen as a chain of bodies that are connected to each other in different ways [5]. The bodies are called *links* and the connections are called *joints* [5]. The joints can be of various types but are most commonly *revolute or prismatic* [5]. By a revolute joint is meant a connection which creates a rotary

movement possibility between two links and by prismatic is meant a connection which creates a linear movement possibility between the links [1].
See Fig 2.2 for a sketch over what links and joints are and principles of the different types of joints.

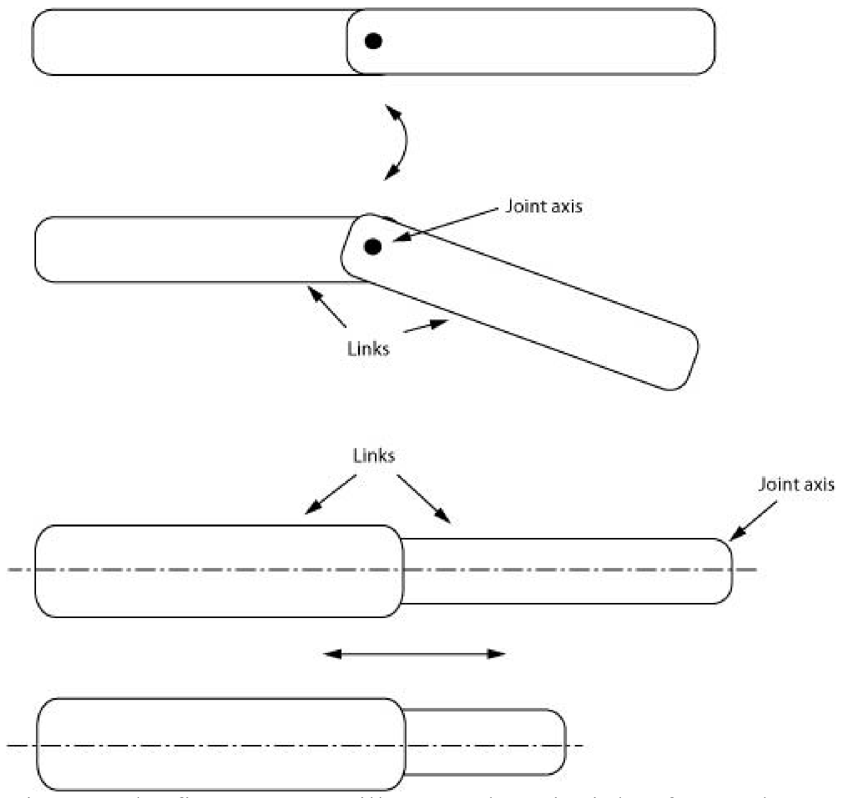


Fig 2.2: The figures on top illustrate the principle of a revolute joint and the figures below illustrate a prismatic joint.

2.1.1 Frames

To be able to control and work with the robot, one has to have an ability to express where in space the tool of the robot is located and which orientation the tool of the robot has at this place. To do this, a system for introducing coordinate systems into the robot model is needed.

To start with, a base coordinate system is attached to the base of the robot [4], see Fig 2.3. This will be the robot's reference to the world, a rigidly attached base frame. Of course, in turn, this frame could be mounted on a cart , hence moving the “world” of the robot according to some other frame, but let us ignore that for now.

Next another frame is attached to the center of the very end of the wrist and this frame is called the *flange* of the robot [4]. A tool can now be rigidly attached to the wrist and the location and orientation of the tool can be expressed in this flange-frame.

The last frame to be attached is a frame at the tip of the the tool called the *tool center point (TCP)*. This becomes important when programming and working with the robot since this usually is the point of interest, the point the robot can do something with, for example in welding. See Fig 2.3 for how the frames are located on the IRB.

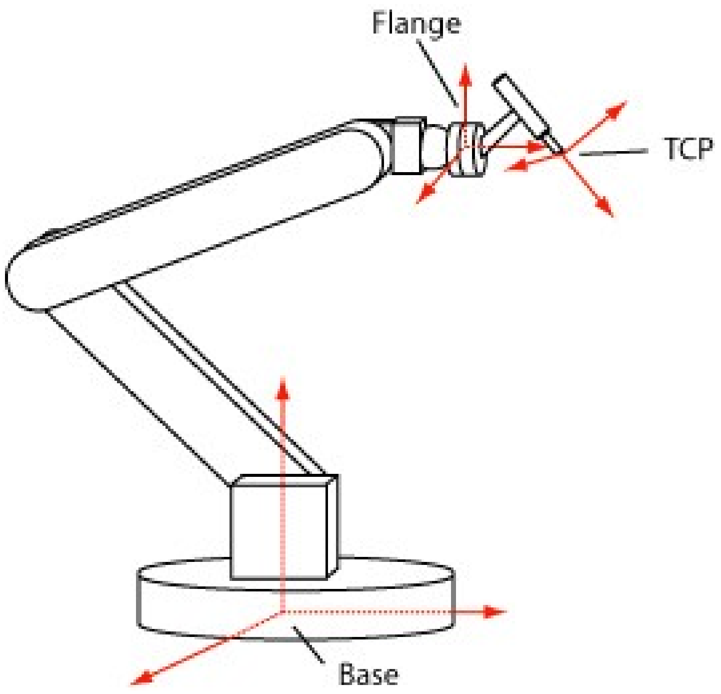


Fig 2.3: Locations of some frames on the robot

2.1.2 Some linear algebra

Given these basic frames the question now arises; what are the mathematical relations between the different frames attached to the robot? Well, before that question will be answered, a discussion about coordinate systems and vectors will be held.

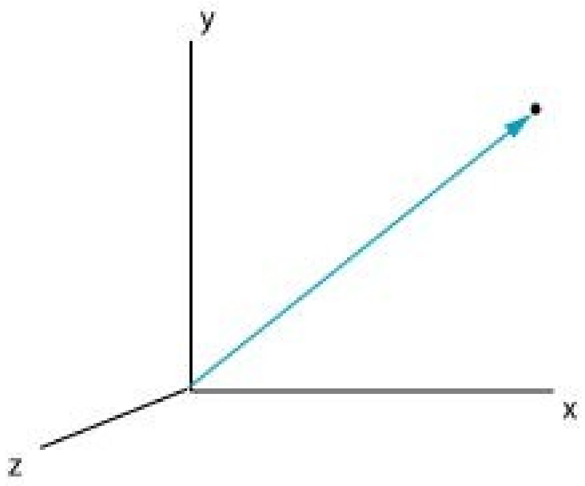


Fig 2.4: A point and its position represented in an orthogonal coordinate system.

Imagine a room in space where there is an orthogonal coordinate system and a point (Fig 2.4). The position of the point can be described by a vector containing three values, the x, y and z coordinates of the point in the coordinate system [5].

$$P = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

Now, instead of only having a point in the room, imagine there is a body somewhere in the room. Imagine also that there is a frame attached to this body. Not only the body now has a position in the room, it also has an orientation described by the frame attached to the body [5] (see Fig 2.5). To express the orientation of the body, the unit vectors of the frame that is attached to the body are expressed in the original coordinate system in the room [5]. Each of the three unit vectors of the body frame are now described by the three vector elements in the room coordinate system. By combining these three vectors you get a *rotation matrix* according to:

$$^{body}_{room}R = [_{room}X_{body}, {}_{room}Y_{body}, {}_{room}Z_{body},] = \begin{bmatrix} r_{11} r_{12} r_{13} \\ r_{21} r_{22} r_{23} \\ r_{31} r_{32} r_{33} \end{bmatrix}$$

where $_{room}X_{body}$, $_{room}Y_{body}$ and $_{room}Z_{body}$ are the unit vectors of the body frame expressed in the room coordinate system [5].

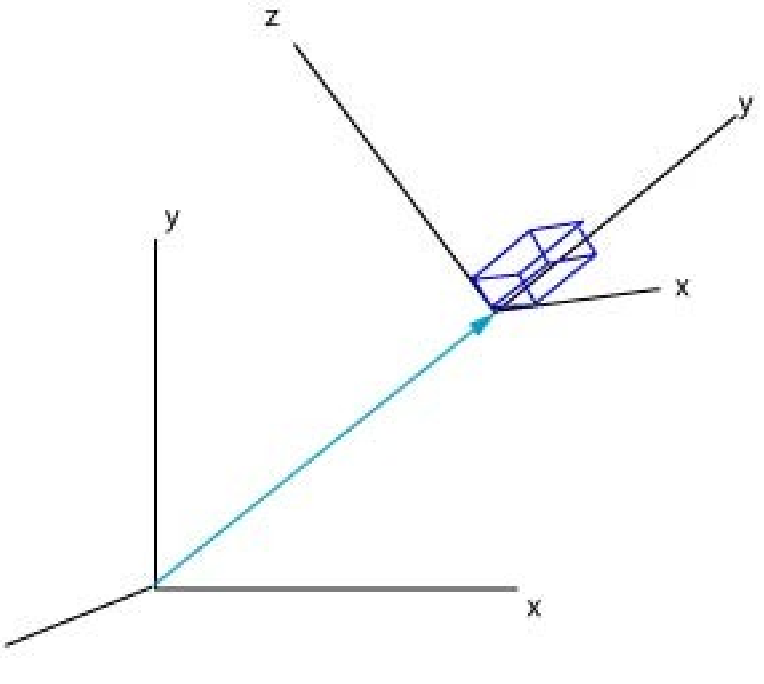


Fig 2.5: A body with a frame attached making it possible to represent the orientation of the body as well as the position.

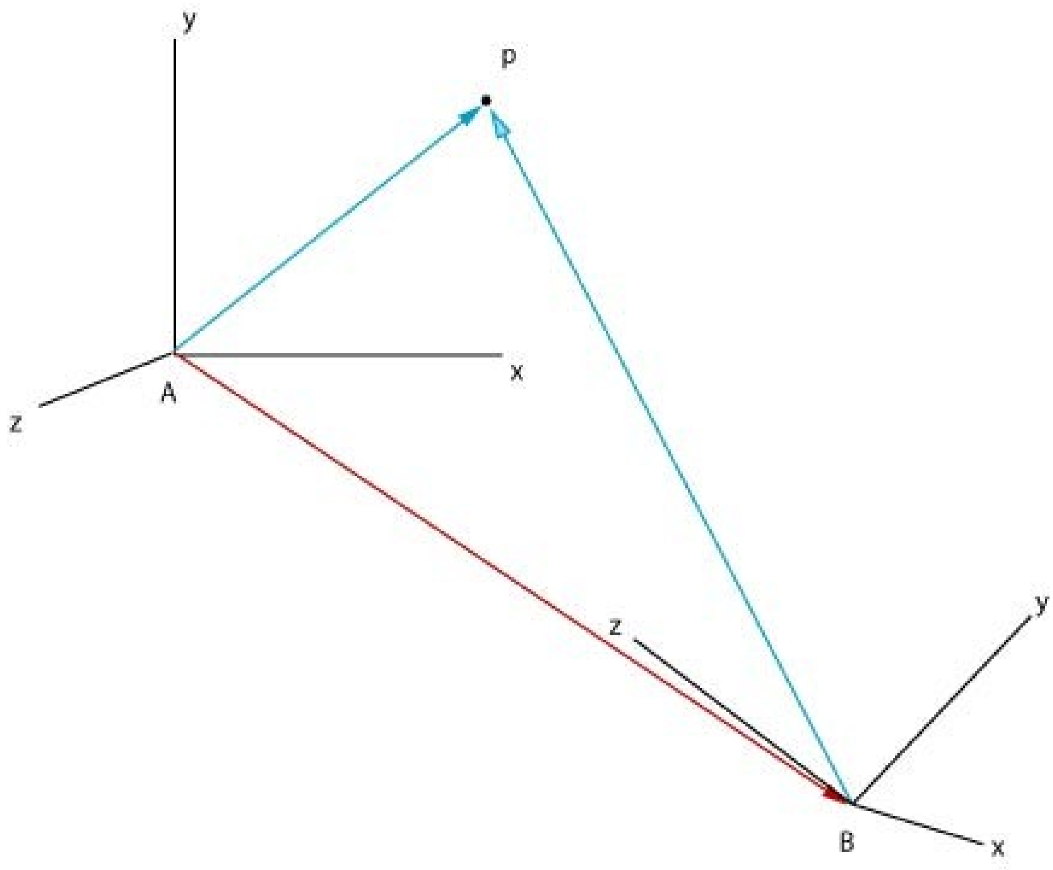


Fig 2.6: How to change the representation of a point from one frame to another.

Suppose now that there are two coordinate systems in a room, A and B, and that there is a point, P, expressed in B's coordinate system (Fig 2.6). How would this point be expressed in A's coordinate system? Well, given the above discussion, this is rather easy. First imagine A to be located at the same point as B. The difference between expressing P in A or B would now only be the difference in rotation between the coordinate systems, ${}^{B}_{A}R$ [5]. So simply multiply P as expressed in B with ${}^{B}_{A}R$ and the remaining difference now only is the difference in location between origo A and B respectively, ${}_{A}P_{B}$ [5], that is, the vector from A to B according to Fig 2.6. To compensate for this a simple addition of the difference ${}_{A}P_{B}$ has to be done [5]. All-in-all, we now have

$${}_{A}P={}^{B}_{A}R\,{}_{B}P+{}_{A}P_{B}$$

which, if adding an extra element to the position vectors (containing a one), can be expressed with a single transformation matrix as

$${}_{A}P^{4}={}^{B}_{A}T\,{}_{B}P^{4}$$

(the 4 denoting that the position vector now has four elements) or more completely written out as

$$\left[\begin{array}{c} {}_{A}P \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} {}_{A}^{B}R & {}_{A}P_{B} \\ \hline 000 & 1 \end{array}\right]\left[\begin{array}{c} {}_{B}P \\ \hline 1 \end{array}\right]$$

[5]. This four-by-four element matrix will from now on be referred to as a *T44 transformation matrix*. To have a single matrix expressing a transformation is handy when several transformations are done in a row since the whole transformation can be expressed by multiplying all the T44's one after another, thus obtaining a new T44.

2.1.3 Forward kinematics

Let us now go back to the robot and apply the above. Recall that a robot was built up by links and that the links were connected through joints. Every joint give one degree of freedom to the robot and can be defined by a *joint axis*. Around this joint axis the links rotate relative each other, in case of a revolute joint, or slide along relative each other, in case of a prismatic joint [5].

Two consecutive joint axes can be used to describe the link between them kinematically. This is done by two values: the shortest distance in between the joint axes and the angle between the two axes that is created when projecting the axes on a plane with a normal perpendicular to both of the joint axes [5] (Fig 2.7). These values are referred to as the *link length (a_i)* and the *link twist (α_i)* [5].

To describe how two consecutive links are connected, two more parameters can be defined. These parameters are the *link offset (d_i)*, which is the distance along the joint axis between the two links, and the *joint angle (θ_i)*, which defines the rotated angle between the links [5] (Fig 2.8).

Hence, there are four parameters for each link that will be used to describe the robot kinematically. This is done by a method referred to as the Denavit-Hartenberg convention. What is done in this method can be summarized as:

1. Number all the joints from 1 to n.
2. Attach a frame to the base of the robot and call it frame 0. Then attach a frame on each link and denote them frame 1 to n.
3. Identify the four link parameters for each link a_i, α_i, d_i and θ_i .
4. Create the T44-matrix for each link. Since each joint only has one degree of freedom the T44 for each link will be a function of only one variable. The transformation matrix is calculated according to:

$${}_{i-1}^{\;\;i}T_{44} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\alpha_i)\sin(\theta_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\sin(\alpha_i)\cos(\theta_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5. Calculate the total transformation matrix between the base of the robot and the flange (or TCP). [5]

Step two and three in the method above are done in a special manner and these steps are described more thoroughly in Appendix A.

Given each joint variable's value at a certain moment, the position and orientation of the flange (or TCP) at this moment can now be computed by using the total transformation matrix that is found in step 5 above [5]. This procedure of calculating the position and orientation of the robot given the joint variables values is called direct or forward kinematics ([5] and [1]).

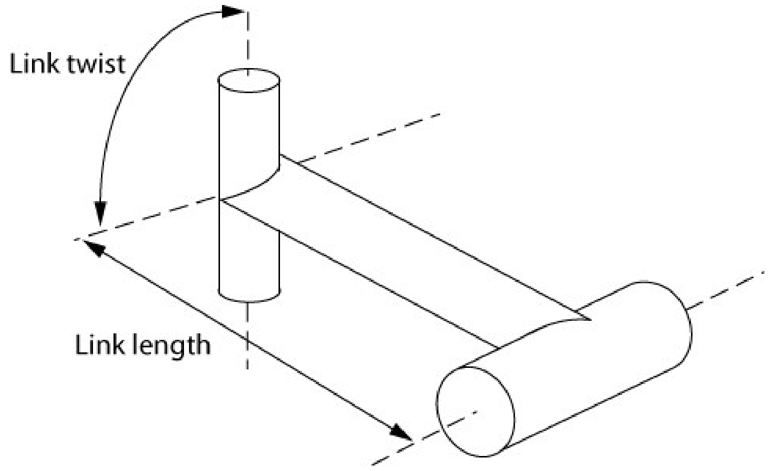


Fig 2.7: Definition of link with the help of its joint axes.

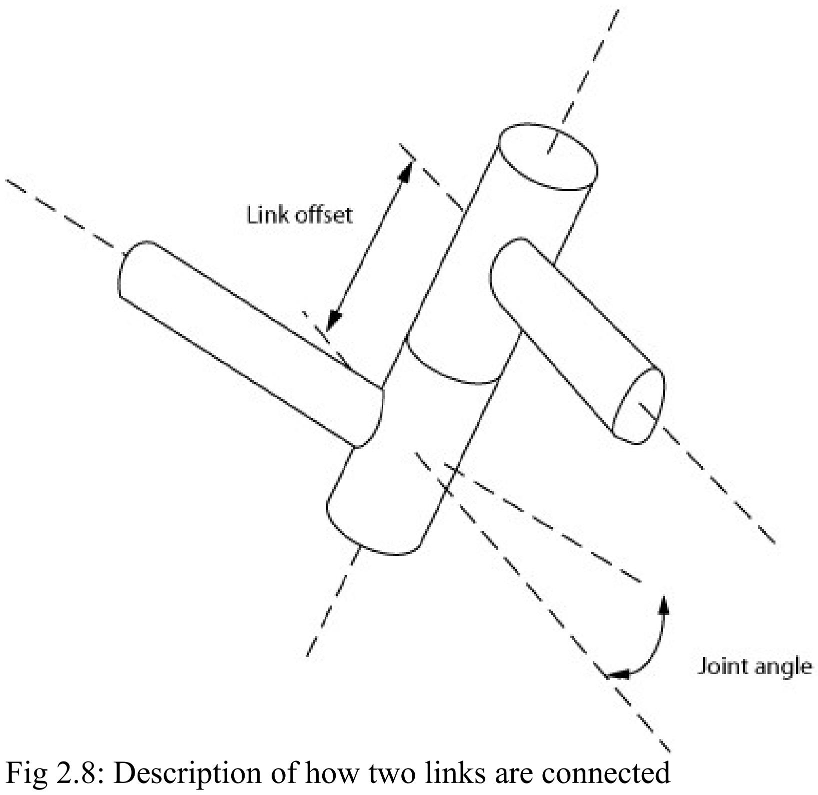


Fig 2.8: Description of how two links are connected

2.1.4 Inverse kinematics

The method above works out fine for deciding where in space the robot is when the joint variables are known. When it comes to controlling the robot, it is most often the opposite that is of interest, to calculate the joint variables given a certain position and orientation in space. This procedure is called the inverse kinematics problem.

Unfortunately this is not as easy as the forward kinematics problem ([5] and [1]). By analyzing the transformation matrix, one can see that it contains six independent elements. Hence a robot with six DOFs should be solvable [5]. The expression for the total transformation matrix of a robot with six DOFs is in general non-linear, quite complex and hard, if not impossible, to solve analytically [5]. It is an equation system of 12 non-linear trigonometric equations [1]. Moreover, it is easy to understand that multiple solutions can exist to a certain wanted position [5] (Fig 2.9). If two joint axes are aligned even an infinite number of solutions can exist. This creates a situation where the system has to choose one solution.

It has been showed that the inverse kinematic problem always can be solved numerically for robots of six DOFs [5]. However, it is desired to find ways of simplifying the solutions since the computations often have to be done online, hence they need to be performed fast [1]. Exactly how one can simplify the solutions depend on the configuration of the robot, limits etc. One way of simplifying can be to divide the problem into an inverse position kinematics part and an inverse rotation kinematics part(which can be done if the last three joints intersect at a point) [1].

Due to the complexity of the inverse kinematic problem and also to that it is not vital to know to understand the rest of the report, it won't be discussed further here. Interested readers are instead referred to other, more extended literature.

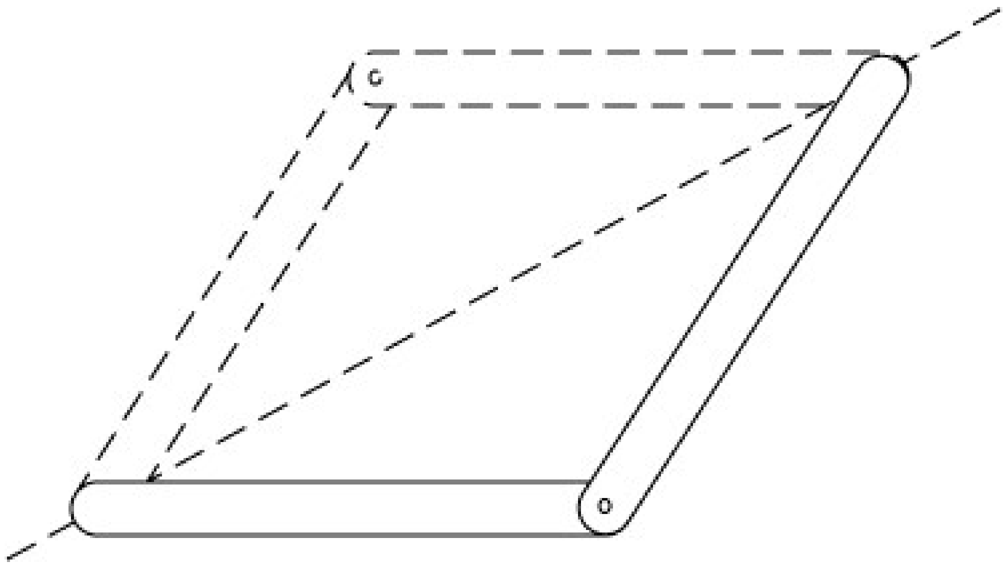


Fig 2.9: Alternative orientation of two links that lead to the same end point position.

2.1.5 The Jacobian

Everything considered so far have been static situations of the type: given the joint variables, what is the position, or the other way around. This works fine for just putting the robot at a certain position and orientation. Only controlling the robot in this static way would result in a behavior comparable with the typical dimestops that are frequently used in “the robot dance” that became popular some decades ago. Quite often it is of interest to move the robot along a trajectory with a specific speed instead. To be able to do so, some relationship between the joint velocities and the linear and angular velocities of the TCP or flange expressed in the base coordinate system has to be derived. In other words expressions of the form:

$$\boldsymbol{v}_0^n = J_v \dot{\boldsymbol{q}} \quad \text{and} \quad \boldsymbol{\omega}_0^n = J_\omega \dot{\boldsymbol{q}}$$

where $\boldsymbol{v}_0^n$ is the transversal velocity vector of the end of the robot relative its base, $\boldsymbol{\omega}_0^n$ the rotational velocity vector of the end of the robot relative its base and $\dot{\boldsymbol{q}}$ the joint velocity vector, needs to be derived. These expressions can be combined to one single expression according to

$$\begin{bmatrix} \boldsymbol{v}_0^n \\ \boldsymbol{\omega}_0^n \end{bmatrix} = J_0^n \dot{\boldsymbol{q}}$$

where J in the expression above is called *the Jacobian of the manipulator*.

The angular velocity in the expressions above, $\boldsymbol{\omega}_0^n$, is defined through

$$\dot{R} = S(\omega)R \quad \text{where} \quad \boldsymbol{\omega} = \begin{bmatrix}\omega_x & \omega_y & \omega_z\end{bmatrix}^T \quad \text{and}$$

$$\boldsymbol{S}(\boldsymbol{\omega}) = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \quad [3].$$

It can be shown that

$$\boldsymbol{\omega}_0^n = \boldsymbol{\omega}_0^1 + R_0^1 \boldsymbol{\omega}_1^2 + R_0^2 \boldsymbol{\omega}_2^3 + \cdots + R_0^{n-1} \boldsymbol{\omega}_{n-1}^n$$

which means that the angular velocities can be added as vectors if they are expressed in the same coordinate frame [1]. Hence a way of calculating the angular velocity at the end of the robot (TCP or flange) would be to add all the angular velocities of the different links as expressed in the base frame [1].

By considering which type of joint (prismatic or revolute) that is connecting two links and applying the above, the expression for the angular part of the Jacobian becomes

$$J_\omega = \begin{bmatrix}\rho_1 \boldsymbol{z}_0 \cdots \rho_n \boldsymbol{z}_{n-1}\end{bmatrix} \quad \text{where} \quad \boldsymbol{z}_{i-1} = R_0^{i-1} \boldsymbol{k} \quad \text{and} \quad \rho_i = 1 \quad \text{or} \quad \rho_i = 0$$

where the last is decided depending on if joint i is revolute or prismatic [1].

When it comes to the linear velocity, it is just the time derivative of the position part of the transformation matrix, hence

$$\dot{\boldsymbol{d}}_0^n = \sum_{i=1}^{n} \frac{\partial \boldsymbol{d}_0^n}{\partial q_i} \dot{q}_i \quad [1].$$

By evaluating $\dfrac{\partial \boldsymbol{d}_0^n}{\partial q_i}$ in the expression above depending on if the joint is prismatic or revolute, the linear part of the Jacobian becomes

$$J_v = \left[J_{v1} \cdots J_{vn}\right] \text{ where } J_{vi} = z_{i-1} \times (\boldsymbol{o_n} - \boldsymbol{o_{i-1}}) \text{ or } J_{vi} = z_{i-1}$$

depending on if joint i is revolute or prismatic [1].

By now combining the expressions for the angular and linear parts of the Jacobian to a common six-by-n matrix according to:

$$\begin{bmatrix} \boldsymbol{v_0^n} \\ \boldsymbol{\omega_0^n} \end{bmatrix} = J_0^n \dot{\boldsymbol{q}}$$

where n is the number of joints, the simple expression that relates the velocities of the TCP or flange of the robot with the joint velocities has been obtained [1].

2.2 Different types of force control

A difficulty with force control is that the problem of the control task often is to apply a certain force at a certain point in the robot's work space. Hence the control problem is a combination of the normal position control and the constraints of the force to apply. To achieve this there are a couple of different approaches.

A quite simple approach is to use passive compliance, by which is meant to have a mechanical device mounted at the end of the robot to add some compliance to the normally stiff manipulator [1]. This gives the opportunity of keeping some contact with the environment and to still have some compliance to avoid large forces to be accumulated [1]. A disadvantage with this approach is that it is quite limited to where it can be used. It still can be useful in for example some mounting scenarios which wouldn't be achievable with only position control [1].

Another, maybe more elaborate approach, is to keep the manipulator as it is and use an active strategy to achieve contact forces. The idea behind this approach is to think of the environment as a spring with a certain stiffness, *k*. By then putting the position reference to the robot a distance, *d*, below the contact surface of the environment the manipulator will create a force of the size F = k * d [1 (Fig 2.10)]. Variations of the environment, like *k* or where the contact surface is located, can result in problems with this control approach.

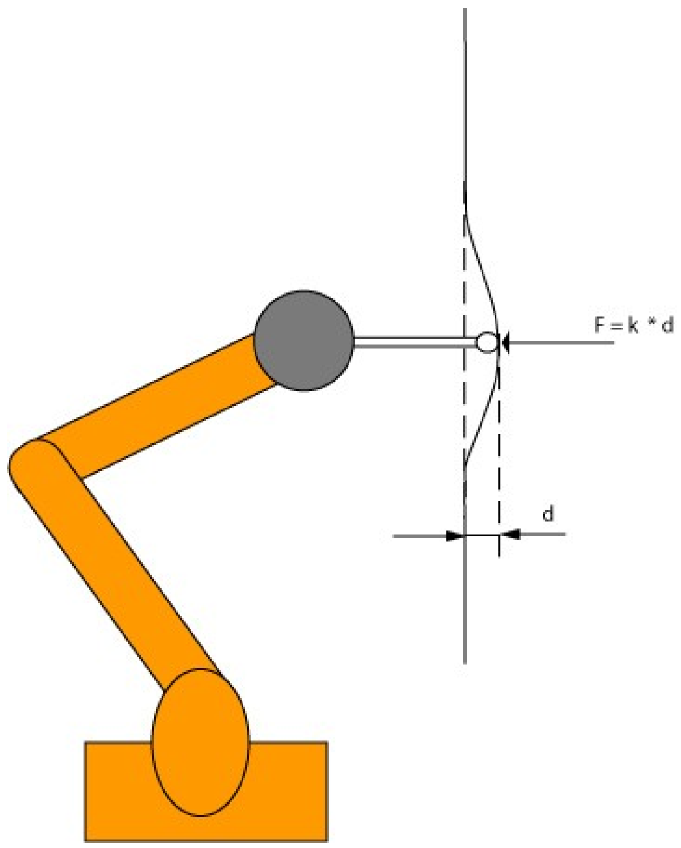


Fig 2.10: Principle of active compliance

There are also hybrid control systems that combine a normal position control with a “pure” force control [1]. These systems naturally become more advanced.

The type of force control that has been used in this project is called *impedance control*. It is slightly different than the previously mentioned strategies since it is not tracking certain positions but instead merely tries to keep a certain relation between the actual force and the velocity (or position) of the robot [1]. This is quite suitable for the imagined usage area of this project since it is merely desired to move the robot around with the measured force. Impedance control is often combined with position control. Basically a desired trajectory is sent into the impedance controller and is there slightly adjusted to achieve the desired force on the environment [3].

With V representing the velocity, F the force, X the position and Z the impedance, the impedance relationship can be expressed in the frequency domain as:

$$Z(s) = \frac{F(s)}{V(s)} = \frac{F(s)}{sX(s)} \quad [1].$$

To obtain an expression for Z(s), consider a mass with a spring and a damper attached under the influence of a force F(s). The quite common relationship of this system is

$$F(s) = ms^2X(s) + d\,sX(s) + kX(s) = (ms^2 + ds + k)X(s)$$

$$\Rightarrow sZ(s) = ms^2 + ds + k \;.$$

In a one dimensional, discrete controller taking F and X as inputs and setting V as an output, the impedance controller can be implemented as:

V(k) = V(k-1) + Ts * 1/m * (F(k) – d * V(k-1) – k * X(k-1))

where Ts is the sample time. Alternatively, one could measure the real velocity and use that in the calculations if this is a possibility.

3. An introductory exercise in how to modify an impedance control block.

Chapter introduction
To get familiar with the robot system, impedance control, how to implement controllers and how to simulate them, it was decided to first carry out an introductory exercise where an existing impedance controller was modified. How this was done, how the robot system that has been used works and the results from this exercise are the topics of this chapter. Also, some terms and definitions of the controller are introduced in this chapter. These will be used and referred to in the rest of the report why it is recommended to at least give a fast glance on this chapter.

3.1 The modified robot system

The experiments and work that have been done in this project are all performed on a modified industrial ABB S4CPlus robot controller [4]. Basically the modification is performed between the main controller and the axis controller of the robot [4]. Normally references of the position, velocity and torque are sent directly from the main controller to the axis controller [4]. Now, a possibility of reading these references into an external controller generated in Matlab Simulink has been introduced [4]. To be able to do some actual changes to the robot behavior, a possibility of switching the source of the references to the axis controller to be from an external controller instead of the main controller of the robot, has also been introduced [4]. See the principle sketch of Fig 3.1. All-in-all this gives a quite simple way of performing control experiments on the IRB.

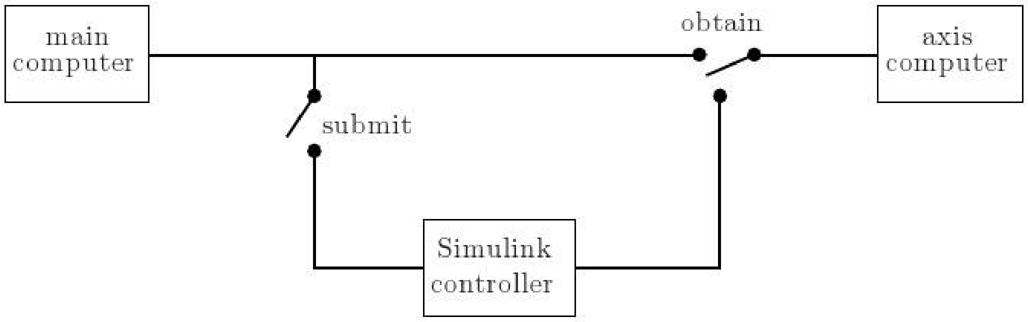


Fig 3.1: How the Simulink controller is connected to the robot

As mentioned a controller can be implemented as a Matlab Simulink model. Which input and output ports it has is specified in a Labcom file [4]. After a model has been made to be run on the robot in Simulink, it can be built with Real Time Workshop [4]. This build model can then be loaded into the robot through an *opcom interface* (see Fig 3.2). The interface allows the user to load a model, choose to submit the model (which means to start reading the references sent from the main controller of the robot into the external controller, but not sending the signals from the external controller to the axis controllers, compare Fig 3.1), choose to obtain a model (which means to both read the signals from the main controller and send out the new signals from the external controller to the axis controller, compare Fig 3.1), values of force sensors can be read and some parameters of the external controller can be changed if that has been specified.

While experiments are performed, an external log-program can be run from a terminal [4]. This program log the input and output signals of the Simulink model. It also logs signals that have been specifically stated to be logged. After the logging has been performed, the log can be imported to Matlab and the results of the experiments evaluated. For a system map, see Fig 3.3.

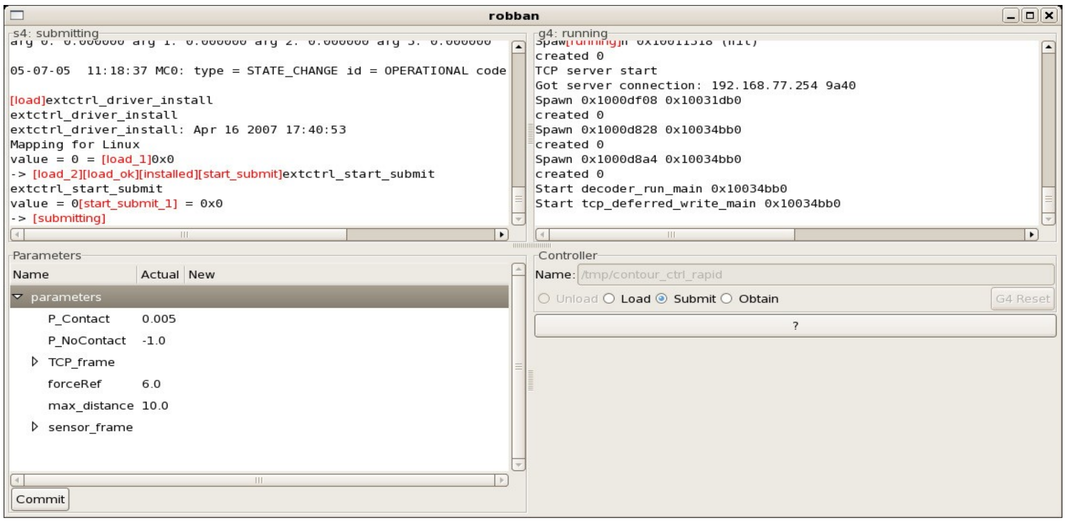
To simplify the development of Matlab Simulink models for the modified robot, there is an available function block library (extctrl) that can be used. In this library there are functions for, for example, calculating the Jacobian and the inverse Jacobian, calculating the forward kinematics, performing velocity transformations etc.

3.2 The model in use

It was decided that the exercise would be to modify an existing controller model. The model of choice was implemented as an impedance controller with the force references set to zero. What that means is that the controller should try to keep the forces on the force sensor to zero. Hence, if a robot programmer would grip the force sensor, the robot should follow his/her lead. This is a very good feature since it makes lead-through programming possible. See the Matlab Simulink block scheme in Fig 3.4 with numbered references.

The signals that come from the force sensor (ref.1 in Fig 3.4) are first reset when the *f_switch variable* is set to one (ref.2 in Fig 3.4). The f_switch variable is used to activate the controller and the idea is that it should put all states to zero when it is turned off to avoid unexpected behavior of the robot when the controller is turned back on next time [4]. The reason for resetting the force sensor input when f_switch is turned on is to eliminate possible offset forces due to for example gravity.

After the force signals have been reset, they go through a transformation block that calculates what the forces that are measured in the sensor frame would be equal to in the TCP frame of the robot (ref.3 in Fig 3.4). The reason for this is that the control is performed in the TCP frame.



```
arg 0: 0.000000 arg 1: 0.000000 arg 2: 0.000000 arg 3: 0.000000

05-07-05  11:18:37 MC0: type = STATE_CHANGE id = OPERATIONAL code

[load]extctrl_driver_install
extctrl_driver_install
extctrl_driver_install: Apr 16 2007 17:40:53
Mapping for Linux
value = 0 = [load_1]0x0
-> [load_2][load_ok][installed][start_submit]extctrl_start_submit
extctrl_start_submit
value = 0[start_submit_1] = 0x0
-> [submitting]
```

```
Spaw[running]n 0x10011318 (init)
created 0
TCP server start
Got server connection: 192.168.77.254 9a40
Spawn 0x1000df08 0x10031db0
created 0
Spawn 0x1000d828 0x10034bb0
created 0
Spawn 0x1000d8a4 0x10034bb0
created 0
Start decoder_run_main 0x10034bb0
Start tcp_deferred_write_main 0x10034bb0
```

Name	Actual	New
parameters		
P_Contact	0.005	
P_NoContact	-1.0	
TCP_frame		
forceRef	6.0	
max_distance	10.0	
sensor_frame		

Name: /tmp/contour_ctrl_rapid

Unload Load [✓ Submit] Obtain

Fig 3.2: Opcom [4]. This interface is used to load a controller made in Matlab Simulink on the robot. In the bottom left corner one can set parameter values of the controller, if they have been specified to be modifiable. The controller is loaded and the different modes (obtain/submit/load/unload) are set in the lower right corner.

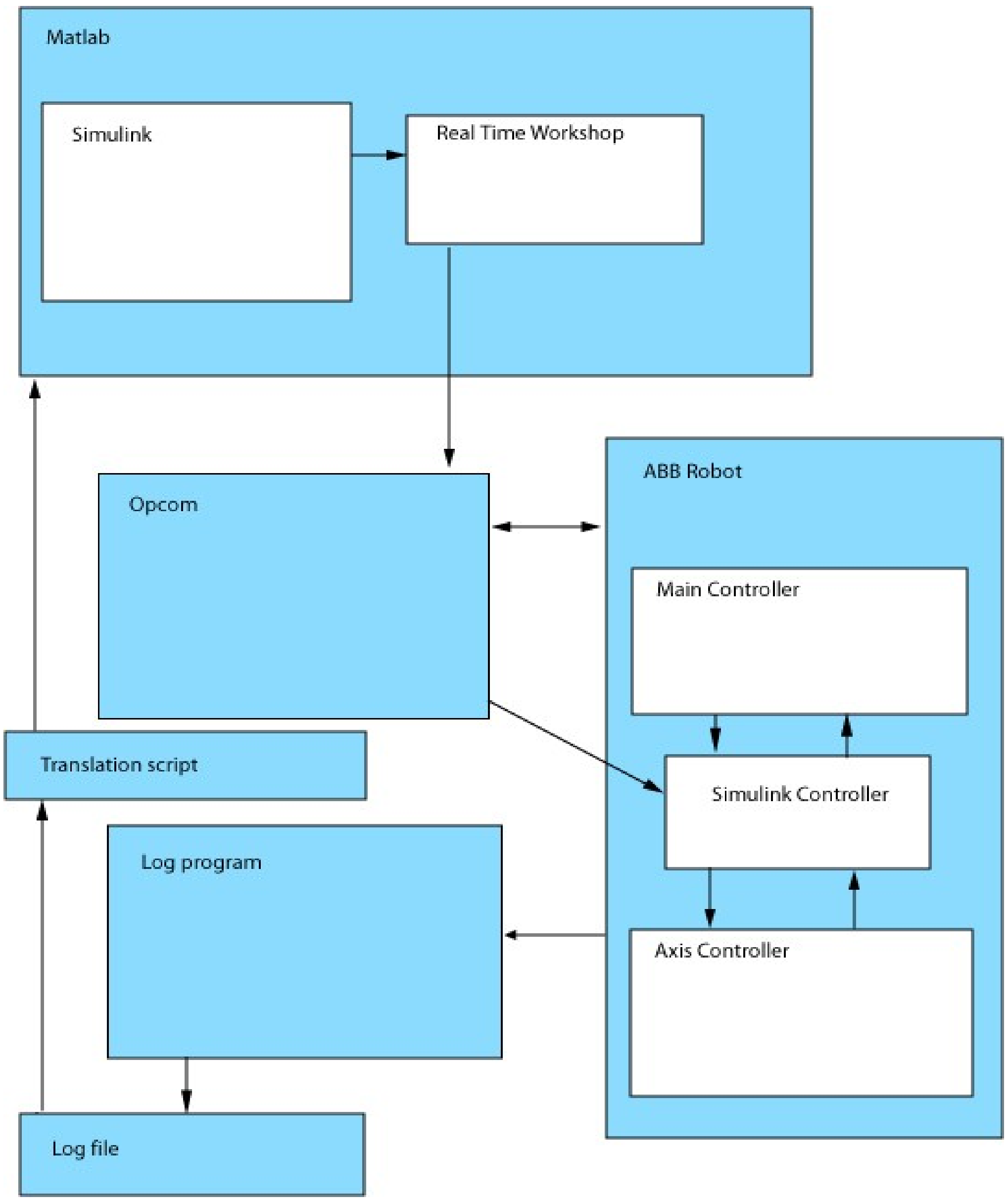
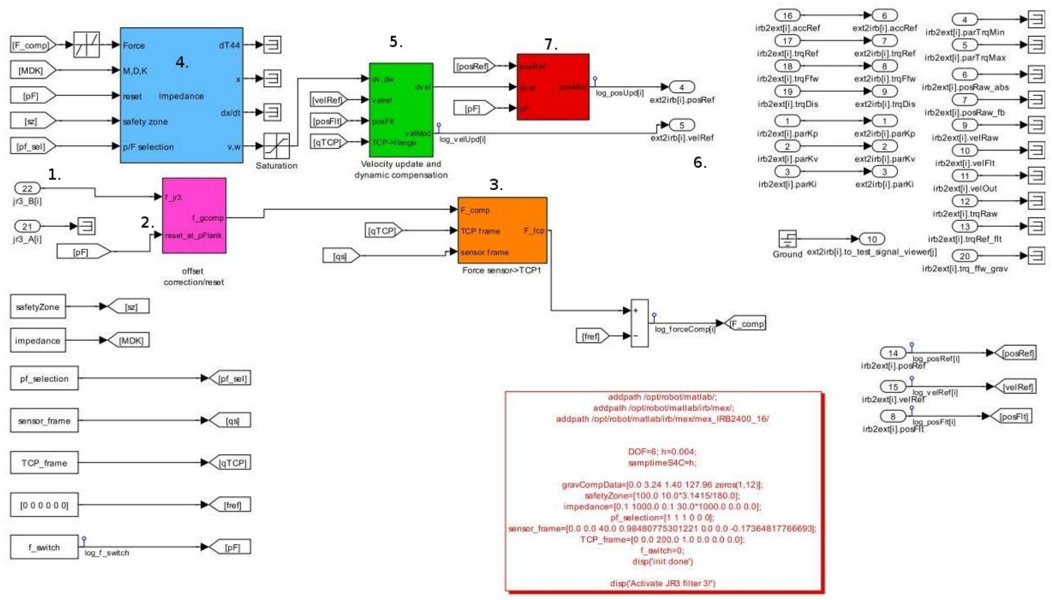


Fig 3.3: A system map over the modified robot system that has been used. It makes up a nice environment to develop the controller in.



```
addpath /opt/robot/matlab/;
addpath /opt/robot/matlab/irb/mex/;
addpath /opt/robot/matlab/irb/mex/mex_IRB2400_16/

DOF=6; h=0.004;
samptimeS4C=h;

gravCompData=[0.0 3.24 1.40 127.96 zeros(1,12)];
safetyZone=[100.0 10.0*3.1415/180.0];
impedance=[0.1 1000.0 0.1 30.0*1000.0 0.0 0.0];
pf_selection=[1 1 1 0 0 0];
sensor_frame=[0.0 0.0 40.0 0.98480775301221 0.0 0.0 -0.17364817766693];
TCP_frame=[0 0.0 200.0 1.0 0.0 0.0 0.0];
f_switch=0;
disp('init done')

disp('Activate JR3 filter 3!')
```

Fig 3.4: Original Simulink model that was going to be changed during the exercise. Notice the numbers of different blocks referred to in the text.

Next, the signals are sent into the controller that calculates an output velocity in the TCP frame (ref.4 in Fig 3.4). This velocity is then sent to a block that calculates the motor velocities that are equal to the velocity calculated in the controller (ref.5 in Fig 3.4). The actuator velocity references are added to the velocity references that are received from the main controller (which should be zero if everything is working out) and are then sent out as new velocity references to the axis controllers (ref.6 in Fig 3.4). The calculated actuator velocities are also sent to a block that integrates them to position references (ref.7 in Fig 3.4). These integrated position references are added to the position references that are received from the main controller (which should stay constant if everything is working out) and are then sent to the axis controllers. Moreover the position reference block has a backtracking feature that moves the robot back to its initial position when f_switch is turned off.

3.3 Concept of the safety zone and what the modification will be

As a safety precaution, the controller model limits the space in which the robot can move. This space is called *the safety zone of the robot*. The borders of the safety zone are defined as a cube in space from the initial position the robot had when the f_switch variable was set to one (and also an angle that specifies how much the flange can rotate). When the controller notice that the robot has reached the safety zone border it will set its output velocity reference to zero to stop the robot. This will be done until the force input tries to move the robot back into the safety zone again.

The modification that was to be implemented on the controller now, was to modify the behavior of the robot when it got close to the safety zone. Instead of abruptly setting the output velocity to zero when the robot reached the safety zone, it was supposed to get harder to move when it got within a certain distance from the safety zone border. Possibly, even a spring effect should be implemented in this zone that would try to bring the robot back to a “safe” distance from the border. The purpose was to give the operator a way of sensing when he/she was getting close to the border.

3.4 Brainstorming for solutions

To come up with ideas on how to modify the impedance controller to get the desired behavior, a brainstorm session was held. Some main ideas and thoughts of this session are presented below.

Different ways of changing the robot behavior
To be able to change the behavior of the robot, one has to change the velocity references set by the impedance controller in the right way. Different ways of doing this can be grouped into two categories; to modify the velocity references after the impedance block or to modify the output of the impedance block itself. The last category can in turn be divided into two more categories; either changing the inputs to the block or to simply change the S-function describing the block.

Next, different possible solutions based on the categories above will be considered. Advantages and disadvantages will be discussed with each solution and the preferred one will be presented with an implementation.

3.4.1 Changing the velocity references after the impedance block

This approach has one main disadvantage that is due to how the original impedance controller works. In the initialization of this block its states are set to zero. The new states (positions) are then calculated only due to the force and MDK (see below) inputs to the block (that is, the positions are calculated by integrating the velocity outputs, they are not feedback from the actual positions of the robot). If one change the output, the controller will not notice this and the states will not have correct values any more. Therefore this idea was abandoned.

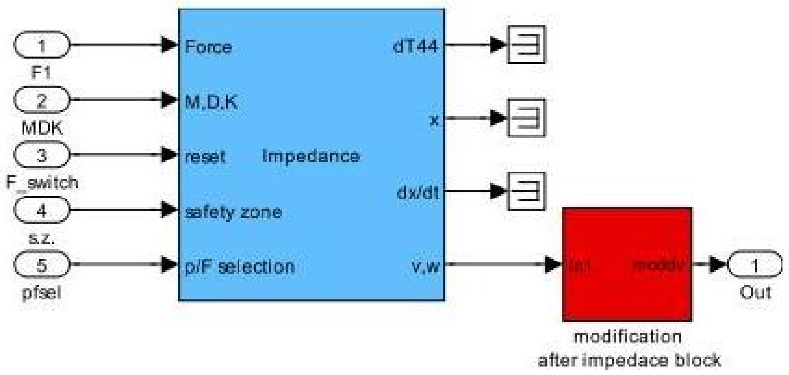


Fig 3.5: Principle of modifying the velocity reference after the impedance block

3.4.2 Changing the s-function of the block

The maybe most natural approach to the problem would be to change the content of the function block. In this way the change would be hidden to the user of the system, which could be an advantage. The problems in the approach above would be avoided since the states are directly calculated from the modified version. Yet another advantage would be that the solution of the problem would be rather easy, a couple of *if-statements* would probably solve it. A disadvantage, on the other hand, would be that the block would lose some of its generality and the solution was not really what was wanted. Preferably the impedance controller should stay the way it already was. An add-on block would be a better approach since one then easily could change the model.

3.4.3 Changing the inputs to the block

The last type of approach would be to change the input to the impedance block and in that manner indirectly change the output.

The first approach of this kind of solution considers the measured force input. When a measured force would be sent in to the impedance block, it would be translated to an acceleration of the manipulator in the given direction (see the discrete implementation of an impedance in previous chapter). By, for example, pre-multiplying the force with a factor < 1 depending on how far into the ”stiff” region the manipulator is at the moment, it would take more force to accelerate the manipulator. However, if the manipulator already has got a velocity when getting to the ”stiff” region, it would still be the same damping of the manipulator as before. In other words, it would be harder to push the manipulator with a high force into the safety zone but one still could “throw” it in there by accelerating the manipulator before it reaches the modified region. A solution to this could be to let the force change sign a certain distance from the safety zone. This would cause the manipulator to decelerate once it passed this line. However, this would cause the manipulator to act a little weird in the stiff region and it would also be a clumsy solution, why it was abandoned.

Another similar solution would be to let the modified region act as a spring. The solution would then be to, depending on the distance into the region, add a force directed towards the safety zone center. The problem of this solution would be that, unless one applies a force on the sensor, the robot would not be able to keep a steady state in the modified region. Instead it would be pushed back towards the center of the safety zone. Depending on the desired behavior, this was considered an accepted solution. Another disadvantage it has though, is that, given enough applied force, one could still crash the manipulator into the safety zone.

The other approach of changing the inputs to the block, would be to change the MDK-parameters that are sent into the impedance control block. *The MDK-parameters specify the mass, damping and stiffness of the impedance controller* (compare Chapter 2).

The K-parameter makes the environment work like a spring, which in this case was not wanted, since the robot should be controlled by pushing it. Thus K should be kept at the value 0. The parameter M works like an inverted amplification of the acceleration. If M is increased, the control of the robot will be more calm and if M is decreased the control will be more jerky. Changing M would thus not put the desired effect on the system. Remaining is the damping D. This is proportional to the velocity of the manipulator and gives a damping effect.

Increasing the damping of the system, depending on the direction of the velocity and the position of the manipulator, would give the desired effect of the system and was considered relatively easy to do.

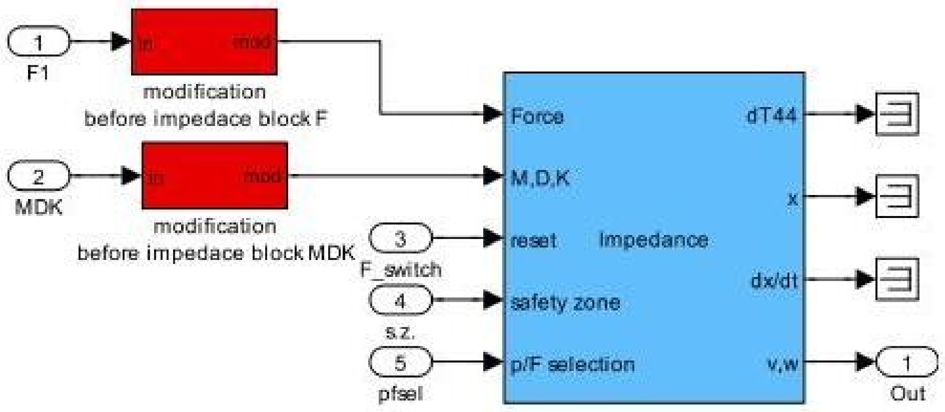


Fig 3.6: Principle of modifying the input signals of the impedance block

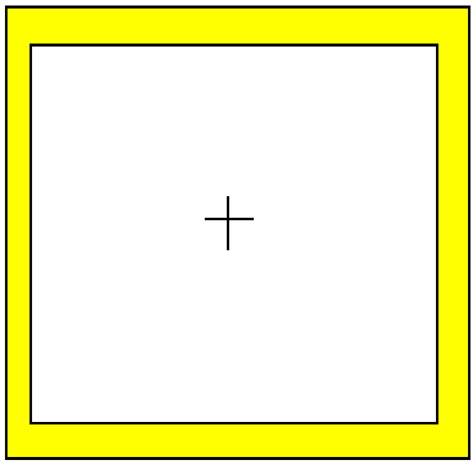
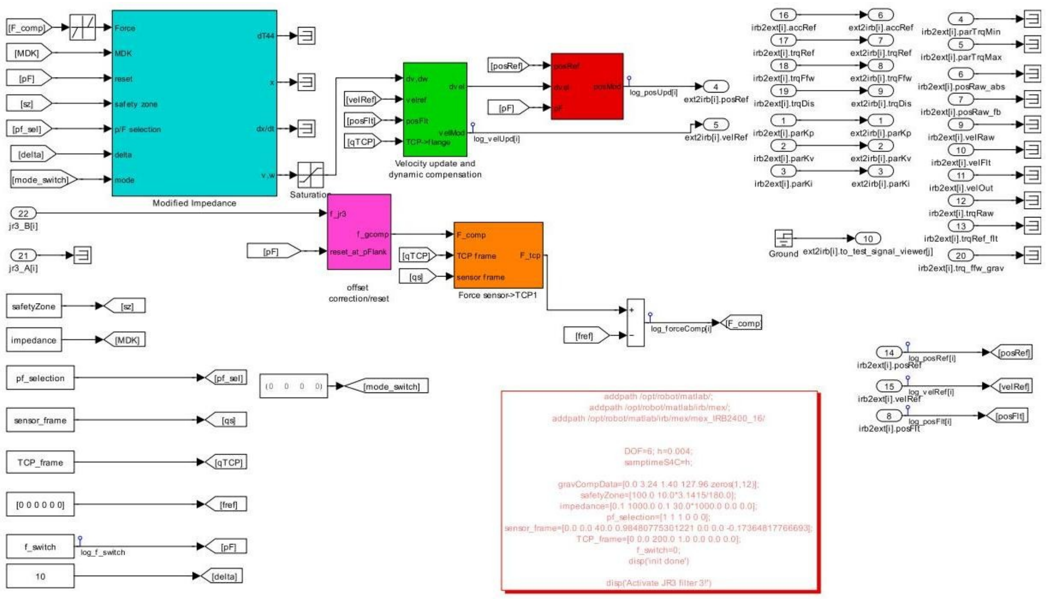


Fig 3.7: The safety zone is defined as the whole figure and the border region, defined by delta, is what is marked yellow in the figure (that is, the outer limit of the safety zone minus a distance delta).



```
addpath /opt/robot/matlab/;
addpath /opt/robot/matlab/irb/mex/;
addpath /opt/robot/matlab/irb/mex/mex_IRB2400_16/

DOF=6; h=0.004;
samptimeS4C=h;

gravCompData=[0.0 3.24 1.40 127.96 zeros(1,12)];
safetyZone=[100.0 10.0*3.1415/180.0];
impedance=[0.1 1000.0 0.1 30.0*1000.0 0.0 0.0];
pf_selection=[1 1 1 0 0 0];
sensor_frame=[0.0 0.0 40.0 0.98480775301221 0.0 0.0 -0.17364817766693];
TCP_frame=[0 0.0 200.0 1.0 0.0 0.0 0.0];
f_switch=0;
disp('init done')

disp('Activate JR3 filter 3!')
```

Fig 3.8: The modified Simulink control model. The controller block (blue) has some new inputs after the modification.

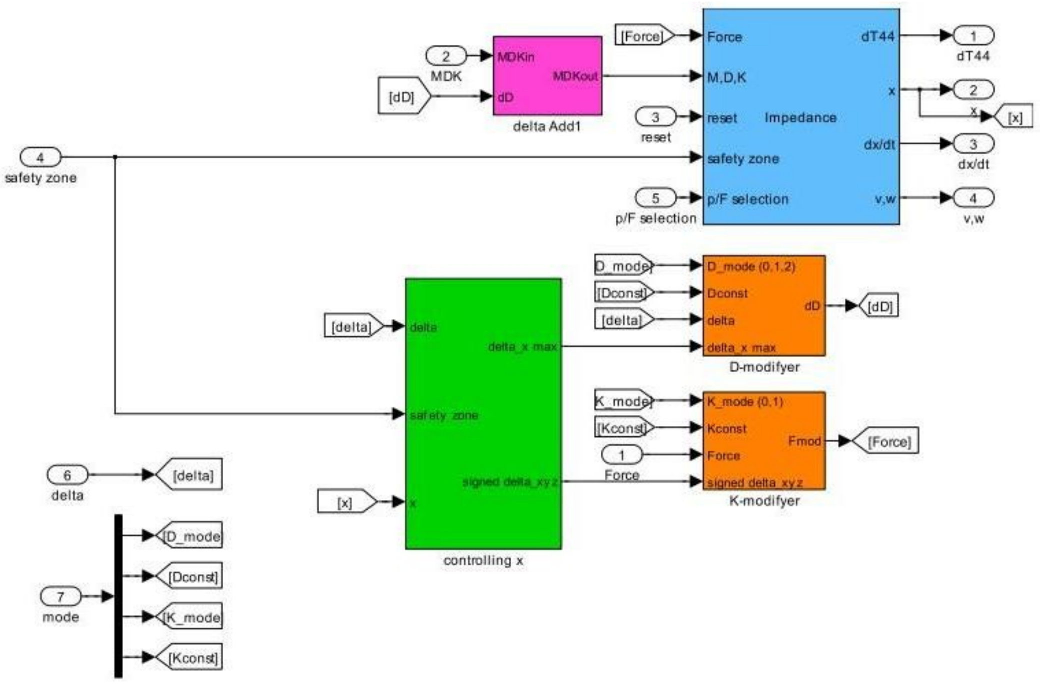


Fig 3.9: The inside of the modified impedance block used in the model in Fig 3.8. Notice that the original impedance block is captured inside this new version.

3.5 Chosen solution and implementation

Given the discussion above, it was decided to use the solution that modified the damping parameter. In addition to this, it was also desired to add an extra K-modifier. However, this K-modifier was only supposed to work in the modified region. This region, as can be seen in Fig 3.7, is defined by a parameter, *delta*. From now on the region will be referred to as *the border region of the manipulator*.

The modified impedance block basically captures the original impedance block and adds a D-modifier-block and a K-modifier-block (see Fig 3.4, Fig 3.8 and Fig 3.9 for the block schemes of the original controller, the modified controller and the inside of the modified impedance block).

Since the damping only is modified by adding an extra damping term to the general damping, this was easily implemented by adding delta D to the MDK input of the impedance block. Under the implementation time it was also decided to add an extra feature that makes it possible to let the damping increase linearly, depending on how far into the border region the manipulator is.

The K-modifier on the other hand was not as straight forward since this term depends on the position of the manipulator. The problem here was that the impedance block defined the distance from its origin, the start position of the manipulator in other words, as the spring distance when it calculated the influence of the K-parameter. Therefore, if it was desired to change the spring behavior only in the border region (and to avoid steps in the K value while doing this) it had to be done indirectly by adding a corresponding term to the force input. By studying the code of the impedance block (see discrete implementation in Chapter 2), one can see that the change of the force that corresponds to an extra spring being present in the border region, should be -K*dX. This term is calculated and added to the force input of the original impedance block.

3.5.1 How to configure the modified impedance

How the behavior should differ from the rest of the safety zone in the border region is defined by two mode parameters, D_mode & K_mode, which changes the damping and spring qualities of the system respectively. The different available modes are as follows:

D_mode 0 –	In this mode the damping in the border region is no different from the rest of the safety zone.
D_mode 1 –	When the manipulator reaches the safety zone the damping of the system is increased with the value defined by Dconst, that is D = D + Dconst.
D_mode 2 – within	When in this mode, the damping is increased linearly the border region according to D = D + max(dX,dY,dZ)*Dconst/delta where dX,dY,dZ are defined as the distances into the border region along the TCP axes where the manipulator is.
K_mode 0 –	This mode means that the spring behavior in the border region is no different from the rest of the working space of the manipulator.
K_mode 1 –	This mode corresponds an extra spring being present in the border region. That is, depending on how far into the border region the manipulator is, it creates a force in the opposite direction. This force is defined as Fx = - Kconst * dX (and in a similar way for Fy & Fz).

By combining D_modes and K_modes in the desired way, one should be able to reach the desired behavior of the system. These modes are then sent into the modified impedance block in the form of a vector defined by:

mode = [D_mode, Dconst, K_mode, Kconst]

where Dconst and Kconst give the characteristics of the respective mode (see mode descriptions above). Dconst is measured in Ns/mm and a typical value could be 0.4-1.0Ns/mm. Kconst is measured in N/mm and a typical value for this parameter could be 0.5-1 N/mm.

3.6 Simulations

To validate the implementation of the modified impedance block and to see that the controller didn't behave unexpectedly, some simulations in Matlab were run. To be able to do this the Simulink model had to be altered slightly in the following way:

- Instead of sending the output of the modified position and velocity to the robot, these were sent to a model of the robot arm that calculated the motor angles the robot would reach given these inputs. This model of the robot arm was already available to run simulations and didn't need to be developed.
- A Simulink block that could send out different signals simulating a measured force input was developed.
- Since the force sensor isn't located at the TCP, the measured force normally is transformed from the sensor frame to the TCP frame and then sent in to the impedance block. However, to simplify the testing so that the output of the impedance block would correspond directly to the simulated force, the developed force sensor simulation block was connected directly to the impedance block. In other words the forces that are simulated are the ones at the TCP and not the ones at the sensor frame.
- The rest of the signals that usually are just sent through the model were removed. Now the model was free of all external inputs and outputs.
- To be able to see what happens when the model is run, some signals were exported to workspace. These signals were the force input (Fin), the relative position (RelPos) and the relative velocity (RelVel) from the outputs of the impedance block (labeled x & vw). Also from within the impedance block, the signals delta D (dD) that show how the damping is changed, the signed delta XYZ (delta_xyz) that show how far into the border region the manipulator is and the delta force (dF) were logged.

To simplify and gain efficiency when running the different simulation scenarios, a Matlab script was written. This script set the different environment variables like for example the different mode settings of the border region, ran the simulation and then presented the logged signals in graphs. See the modified model in Fig 3.10 below.

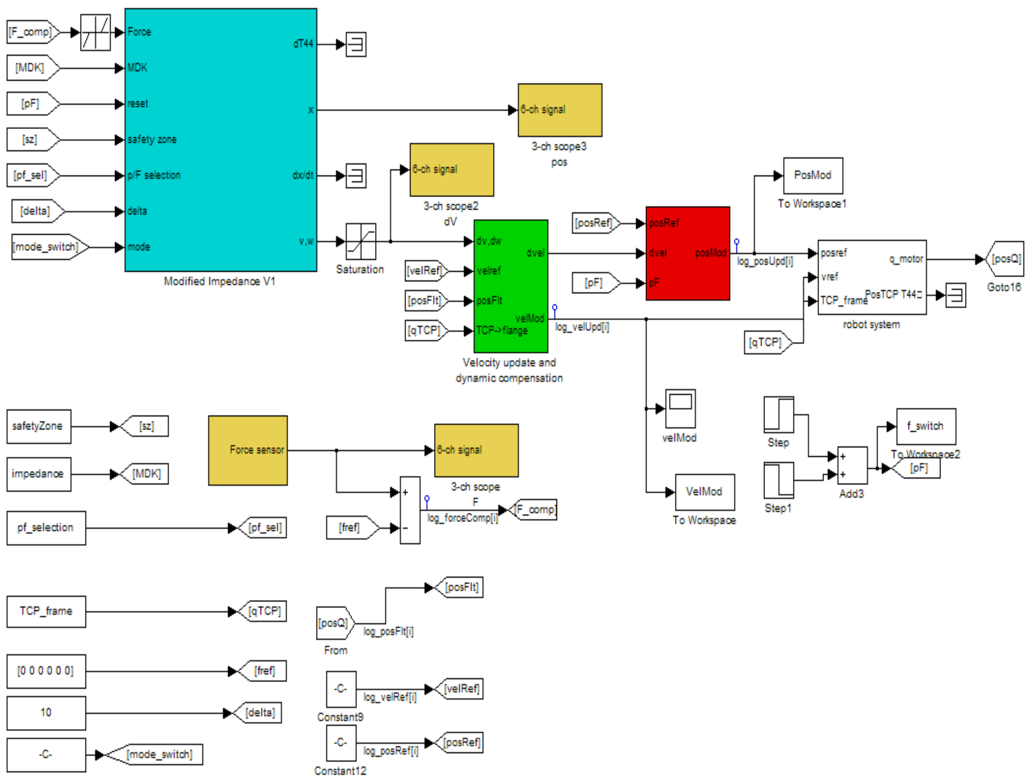


Fig 3.10: Simulated Simulink model

3.6.1 Simulation results

The main difference between the simulations were the mode settings. Recall that the mode settings consisted of a vector with four elements, [Dmode, Dconst, Kmode, Kconst]. See "How to configure the modified impedance" above for further references about these settings.

The magnitude of the force sent into the system was 5 N. This was chosen since it is a force that can easily be achieved by a human hand and the intended interaction is with the robot and a human. The force was applied in the positive x-direction and the negative z-direction of the TCP to see that the arm responds in the same way independently of the coordinate signs. No force was applied in the y-direction.

Simulation 1 – mode = [0 0 0 0] In the first experiment the border region modes were all set to zero. In other words this experiment corresponds to what the old impedance block would act like. Thus, this experiment in a sense acts like a reference for the other experiments. The results can be seen in Figure 1, Appendix B.1.

Characteristics for this experiment was that the relative velocity is quite roughly set to zero once the safety zone is reached (RelPos = 100). This was what the modified impedance block should change. DeltaD and dF (the values that are added to the normal damping term and force input to achieve the actual changes) are zero since both their modes are set to zero.

Simulation 2 – mode = [1 0.4 0 0] This experiment intended to test how the system behaved with an extra damping constant being added to the MDK parameters, while in the border region. The standard damping was set to 0.1 Ns/mm and the added damping to 0.4 Ns/mm.

The results are presented in Figure 2, Appendix B.1. As can be seen the velocity of the manipulator (RelVel) first started stabilizing around a level where the damping cancels out the effect of the applied force. When the border region was reached, delta D increased as a step to 0.4 and RelVel decreased and starts to stabilize around a new, lower value until the safety zone was reached. Then RelVel was set to zero. In this way the manipulator has significantly less speed once it reaches the safety zone, which is good.

These results are pretty much what was intended for the mode. Maybe the velocity decreased a little too rapidly when the border region was reached. However this was what one could expect to happen when a stable velocity state had been reached and all of a sudden the damping was a lot higher. As a conclusion it can be said that the experiment went as expected.

Simulation 3 – mode = [2 1.0 0 0] The next step was to try how the system behaved when the damping increased linearly with max[deltaXYZ]. The results, that can be seen in Figure 3a and b in Appendix B.1, were quite similar to the previous ones. What can be noticed here is that, when the border region was reached, the relative velocity decreased a bit more smoothly. Also it can be seen that delta D no longer is increased as a step but depends on deltaXYZ. The conclusions are that also this experiment had good results and behaved as expected.

Simulation 4 – mode = [0 0 1 0.5] Now, since the damping modifier seemed to be working, the spring behavior was to be tested. As can be seen from the results in Figure 4, Appendix B.1, this is quite different behavior from before. The stabilized position was reached through an oscillation caused by the spring constant. Also the stabilized position was no longer at the safety zone limit but a distance away from it. This is due to the spring constant and the input force. Even these results were as expected.

Simulation 5 – mode = [2 1.0 1 0.5] This experiment was run to see the behavior when both Kmode and Dmode is used. This is pretty much a merging of the previous experiments. The result were quite good since it gave a system with smooth changes both of the position and velocity (Figure 5, Appendix B.1).

As a conclusion a nice system can be achieved by setting the mode-parameters in a good way. However, these parameters have to be chosen according to the expected forces on the system. For example, if one expects high forces both the damping and the spring constant have to be tuned up if they should make much difference. If one expects to have inputs that differs a lot in between each other, the system could be improved by instead of setting the mode parameters statically, update them dynamically. On the other hand it can be said that since the system is mostly intended to be used in human interaction with the robot, these forces should be around a quite limited range and it would probably be enough to choose them statically.

Simulation 6 – mode = [2 1.0 1 0.5], Fin = pulse To see how the system behaved if the force input was set to zero again after the system had reached stability due to a force input, this simulation was run. According to Figure 6a and 6b, Appendix B.1, the manipulator then was pushed out of the border region due to the spring force and the delta D also returned to zero when leaving this region. Even these results are smooth and as expected.

Safety simulations As a safety experiment it was looked upon how the system and the states changed if the f_switch was turned off while operating. Since no real changes of the model where f_switch was effected had been done it should be no problem with this test. The results are presented in Appendix B.2.

During these experiments, three extra signals were logged; the f_switch signal, the PosMod signal (the modified position reference sent to the robot controller) and the VelMod signal (the modified velocity reference sent to the robot controller). The two last signals are given in motor coordinates (angles of the motors and angle velocities of the motors).

If the results are studied it can be seen that the PosMod signal in all cases was smooth and without any jumps. The VelMod signal on the other hand was not as good. As can be seen, whenever the velocity output of the impedance block is set to zero (which happens when f_switch is set to zero or when the safety zone is reached) the VelMod signal gets a transient with opposite sign to the velocity it used to have. After this the signal is set to zero. By looking closer at the model, these transients were tracked down to a discrete transfer function in the “velocity update and dynamic compensation”-block. Since this was nothing that had been introduced in the model now, it was still considered good to run.

3.7 Tests on the actual robot and results

To verify that the model did not only work in the simulations, some experiments were also run on the actual robot. These tests were done by setting a certain mode and then creating a force on the manipulator (with a stick) to see, and feel, the results. To make it easy to see the difference in the tests, the delta region was set to 50 mm. The different modes that were tested were:

[0 0 0 0]
[1 0.4 0 0]
[2 1.5 0 0]
[0 0 1 0.5]
[1 0.4 1 0.5]

These are pretty much the same modes as were set during the simulations described above. The results from these tests confirmed the simulated results. It was also confirmed that the length unit in the impedance block was mm and that the magnitude of the mode constants were reasonable or maybe even a little low (in order to feel the results easily).

4. Controlling the robot with two force sensors

Chapter introduction
The purpose of this chapter is to present the main topic of this thesis. It will explain what the main task has been and what it seeks to solve and fulfill. Also, the chapter presents the final solution and explains how it is intended to work. Last, the chapter show simulation results of this solution and the results from actual tests on the robot together with an analysis of the results.

4.1 The task and objective

4.1.1 The task

The idea of this project was to extend an existing impedance controller that was run by one force sensor. This existing controller was intended to be used in a lead-through programming scenario. What this means is that a robot operator should be able to grip a handle, or something similar, that is connected to a force sensor and then program the robot by leading it to different points, over a surface etc. Lead-through programming has the advantage of being intuitive, fast and easy compared to, for example, pre-programming the robot in a computer environment. Since it gives an increased possibility for companies running smaller series to use robots it is an interesting topic for the robot industry. If it would take a lot of time to program the robot or if this would be cumbersome, robots would simply not be a good alternative for these companies.

A disadvantage the existing controller had was that the controller only took into account what the operator gave as input. If the operator then for example would run the robot into a wall, the robot would not be aware of this new obstacle and it would instead keep trying to go through the wall, hence creating large forces. Of course, the robot safety system eventually would react if the currents got too large due to the forces on the robot. At this point however, the forces could have been built up enough to damage the tool of the robot or the environment.

To make the controller not only react to the input from the operator and to avoid large forces, if possible, the idea was now to complement the existing controller with an additional force sensor. This second sensor should be connected to the tool of the robot in order to protect it and should in a sense be the master sensor. How to make the controller take the new input into account when calculating the output was now the topic of this project.

4.1.2 Demands on a solution

First step in finding a solution was to try to get an idea of how the new controller should work, in other words, to put some desires, or demands, on the required solution. Most of these were of a general nature and more of desires than hard specific demands.

One of the initial desires was that the new controller should resemble the existing controller as much as possible. For example, if there were no forces measured on the second force sensor, the controller should basically work in the same way as the current impedance controller already did. The same thing went for the case when there only were forces measured on the second force sensor and none on the first one. It should in that case also be as similar to a simple impedance controller (but now run by the second force sensor instead of the first) as possible.

Another demand was that the controller should use the second force sensor as a master control input. By this is meant that if the measured force on *F2 (tool force sensor)* is large enough, specified by a limit Fmax, the controller should neglect the input on *F1 (operator force sensor)*. Also it was desired to create some merged control run by both F1 and F2 if F2s magnitude was less than Fmax. The reason for this was to try to create a smooth control behavior.

Some other basic demands on the solution was that it should, as far as possible, limit forces on the tool and in that way protect the tool and the environment, and that the solution still should implement the safety zone to limit the space in which the robot could move.

Since F1 is obtained from the programmer, it can be considered quite safe from the programmers point of view. The robot does as the programmer commands through this sensor. F2 on the other hand doesn't come directly from the programmer. Since F2 was supposed to be the master controller it was important to keep the resulting output from being to sudden and violent out of safety precautions. For example if a large force suddenly would be applied to F2 it shouldn't make the robot change direction of movement to rapidly since this could harm the programmer.

4.1.3 Limitations

Some limits were also put on the required solution to make it a little more simple and achievable. One limit was to the lead-through programming scenario in which the solution is intended to work. This means that the safety zone is static for example and no desired trajectory along which the robot should move needs to be taken into account.

Another limit that wasn't put initially is that the controller only considers three degrees of freedom, more specifically it doesn't consider rotational movements of the tool. The reason for this limit, and also the reason for that it wasn't a limitation from the beginning, was a lack of time to implement and test it.

4.2 Solution and how it works

The first thing that was done in order to find a solution after the problem had been defined, was to have a brainstorm session on different possible ways of connecting F2. While doing this, it was helpful with what was learned from the previously done exercise described in Chapter 3.

The brainstorm session resulted in two possible ways of solving the problem. The first solution built on the idea of using two separate impedance blocks, one run by F1 and the other by F2. The outputs from these blocks were then supposed to be merged by the help of F2 into one single output. This solution is called *impx2* and the principle can be seen in Fig 4.1.

The second solution build on the idea of merging the readings from F1 and F2 into a common value and then supplying that to a single impedance block. This solution is called *Fmod* and the principle can be seen in Fig 4.2.

These two solutions were developed simultaneously as long as possible in order to not only getting locked to one specific solution and in that way missing possible good alternatives.

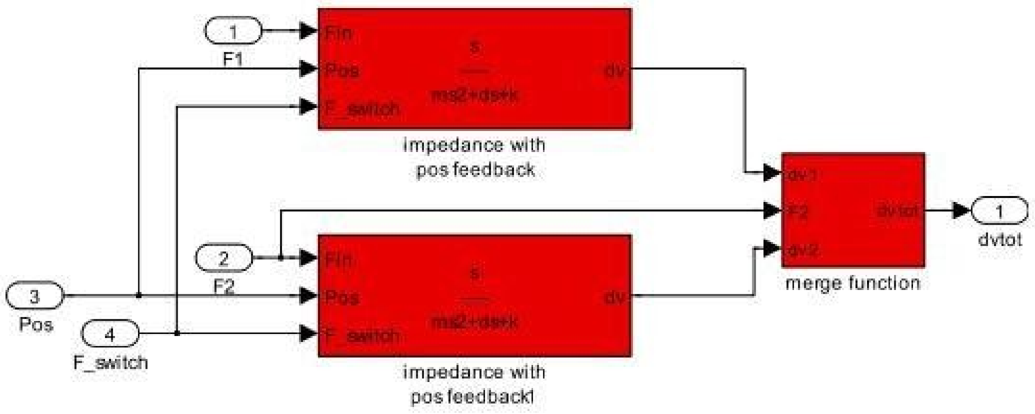


Fig 4.1: Basic principle of the impx2 solution

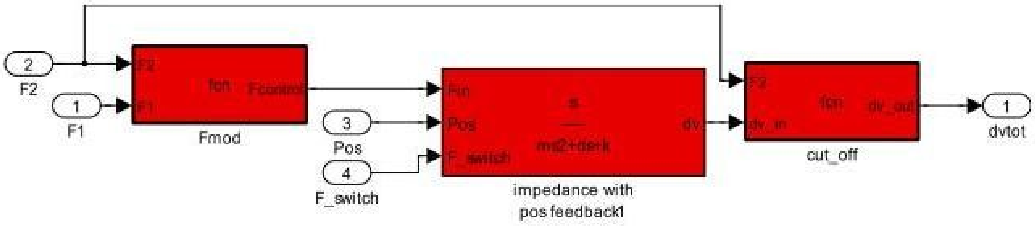


Fig 4.2: Basic principle of the Fmod solution

After some development and different versions of the two suggested solutions had been simulated, tested and evaluated, one final, working solution was reached. It is this solution that will be presented in the following text. The internal structure of the controller can be seen in Fig 4.3.

4.2.1 Concept of final solution

The final solution builds upon the idea of merging the force sensors readings to a single value and then feeding this value to a single impedance controller to create an output. This initial idea has been modified during the development stage and put together in a Matlab Simulink s-function together with a couple of files defining some help functions. The implementation of the safety zone has also been put in a separate Simulink block after the controller s-function block and basically works as a filtration of the output signal. Moreover, the controller now also utilizes position feedback. This is mostly important for the safety zone implementation, to avoid that the controller loses track of its position.

Main strategy

As mentioned above, one of the demands of the solution was that it should resemble the original controller as much as possible in its behavior. What the final solution does is thus to weight F1 together with F2 depending on how large F2 is in contrast to a predefined value, *Fmax*. This is basically done according to:

$$Ftot = \frac{|F2|}{Fmax} * F2 + \left(1 - \frac{|F2|}{Fmax}\right) * F1$$

If F2 is larger than Fmax, Ftot is set to F2. This value is then used to calculate an output velocity according to a basic impedance relationship. The difference from the original controller so far is only that it is controlled by a merged force from F1 and F2.

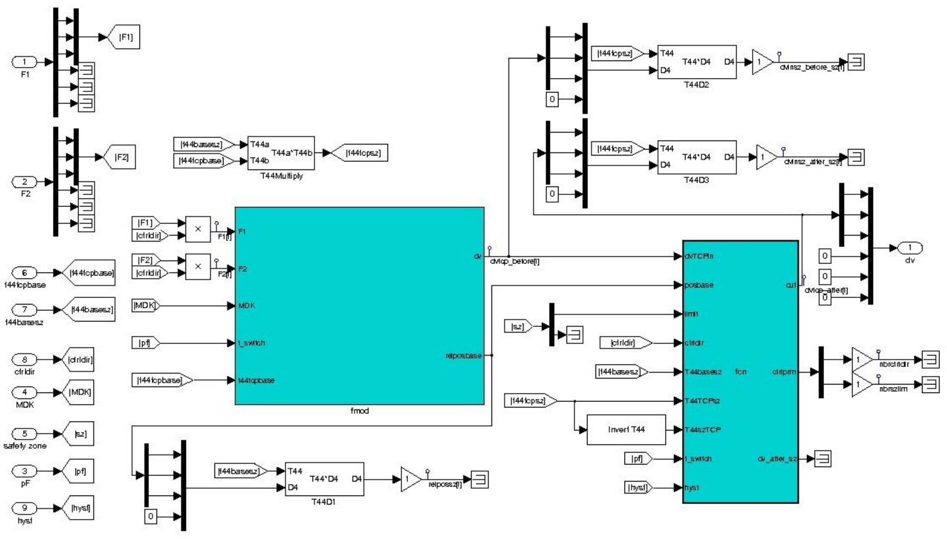


Fig 4.3: The controller and safety zone are separated into two different blocks.

To avoid that forces remain on F2 under an extended period of time, a variable (called *beta*) is used to keep track of how long time F1 and F2 have indicated different things to be done (i.e. have had opposite signs). The basic idea of this is to increase beta with a specific amount every sample F1 and F2 have different signs and are not zero, and otherwise decrease it with the same amount. By doing this, and only letting beta take values between zero and one, one gets a notion of how long F2 has been active. If one now also decreases the importance of F1, and increases the importance of F2 in the weight function, depending on the size of beta, one can guarantee that the influence of F2 increases with time. When beta is one the impedance is only influenced by F2 and the controller moves entirely according to this, thus trying to reduce F2.

Contact scenarios

When the robot is run into a wall or an object, a situation where F2 become large quite fast, easily arises. When F2 is large, Ftot is equal to F2 and the impedance is completely run by this force. The velocity that is calculated from the impedance relationship soon become large and moves the robot away from the wall. When it lose the contact with the wall the robot will keep moving a while until the velocity has been damped down and F1 has retained its influence on the control. In this manner it becomes hard to make the robot stay in contact with an object/wall.

Since the objective of this project is to use the controller in a lead-through programming scenario it can be important to be able to stay in contact with an object. A new strategy to achieve this is therefore needed.

The main strategy at a wall contact now becomes to still transfer the control to F2 as before. The difference will be that F2 instead should be reduced to a low value and thus stay in contact. While F2 still has a low contact force present it will make sure that beta stays high as long as F1 and F2 have opposite signs, thus letting F2 stay in control. This effectively removes situations where chattering might occur when the robot gains contact with the environment.

There are three remaining things that need to be taken into account when creating the controller according to the above. How to decide when the new behavior should be used, how to detect when F2 is decreasing and how to reduce the force on F2 and keep it at a low value. First thing first.

It has been decided that the controller should change its behavior to a more restricted behavior if beta is larger than a certain limit (0.9 in this case). The reason for this is that beta being large indicates that F1 and F2 want different control behavior. This fact quite nicely captures the situation that occur when the robot crashes into a wall. If F1 and F2 have the same sign a situation where the robot needs to stay in contact with a surface is not going to arise. Also when F1 and F2 are directed in the same direction, beta will start decreasing and the controller will return to its normal behavior.

The next problem was to decide when F2 is decreasing. Basically one should be able to merely compare F2s current value with its previous sample and see if it is more or less than that. However, tests on the real robot show that F2 has a noisy character which makes this approach impossible. The natural way of solving this is to filter F2. Using a filtered value of F2, *filtF2*, on the other hand makes the reaction time of the controller slower. Therefore it is important to keep the filter time as low as possible. By comparing the current value of the filtered F2 signal with the previous value now, an idea can be made of if F2 is increasing or decreasing. To increase the performance a little, the filter is in some situations (see 'Discrete update and functions step-by-step' below) reset. In this way one can avoid old readings on F2 to have too much influence.

Now, the difference between filtF2 and the previous value of filtF2 is denoted *dfiltF2*. When in the more restrictive mode (beta > 0.9) this signal decides the behavior of the controller. If dfiltF2 indicates that filtF2 is increasing the controller just works according to an impedance relation. If it indicates that filtF2 is decreasing quite fast, the output velocity is set in a decreasing manner described below. Last, if dfiltF2 indicates that the output velocity should decrease, but dfiltF2 is limited in size, a limited impedance relation is used. This is done in order to get a smooth control and avoid situations where dv in one case is decreasing and in the next is increasing.

The last problem was to find a nice way to decrease F2. This is done by saving the top values of F2 and dv before F2 starts decreasing. By now letting the output velocity be calculated according to

$$dv = \frac{F2}{topF2} * topdv$$

instead of the normal impedance relationship a nice reduction of dv is obtained. To keep F2 at a low value, dv is also set to zero if F2 is less than a certain limit and if beta is larger than 0.9.

Some more efforts have been made to try to stay in contact with the wall/object at fast changes of F2. If F2 has decreased more than a specified value from one sample to another, one can assume that it is too much of a change in between the samples for the change to be due to noise on F2. In this situation the new output velocity is calculated by estimating how many samples it would take for F2 to become zero with the current velocity. If this would be less than a specified number of samples it is considered to go too fast for the controller to stop the robot and still maintain contact. To deal with this, the output velocity is reduced to match the number of samples it takes for F2 to become zero.

4.2.2 Discrete update and functions step-by-step

The s-function of the control block is called every sample of the robot (i.e. every 4 ms) and a discrete update function is then executed to calculate the new output. What is done in the controller in an overall aspect is as follows:

1. First the force sensors are read
2. The merged force, Ftot, is calculated from these two readings
3. Ftot is sent as an argument to a function that calculates an output velocity.
4. If f_switch is turned off, all states are set to zero and in that way, also the output velocity.

To implement this some extra help functions and variables need to be in use. What these functions and variables are and exactly how the implementation is done will be described next by going through the discrete update stage step-by-step and explaining how the different functions work.

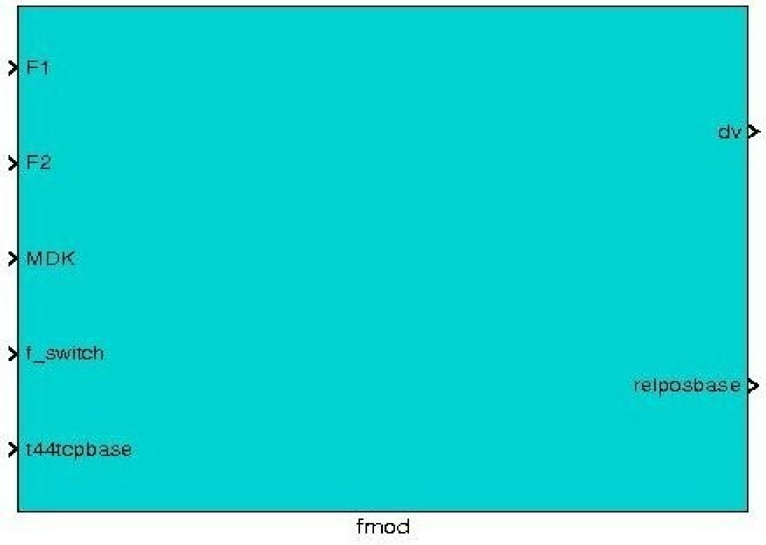


Fig 4.4: The control block in the final solution

The main purpose of the discrete update stage is to update all the internal discrete states that the s-function has. Some of these states are then used as outputs of the control block. The first thing that is done in the discrete update function is therefor to read in the old states into local variables with suitable names, see the code for the discrete update and help functions in Appendix C.1. Also some input values and parameters to the s-function are read into local variables.

Since the control is performed in three orthogonal directions these can in most cases be considered as separate controllers implemented in the same function. After all the variables are set, a for-loop is executed. This loop runs one lap for each controlled direction. When writing the code like this, the separate functions for running the control only need to be implemented for one direction. The first steps in the for loop mostly have to do with calculating values for some help variables.

The first value is beta. Recall that this factor takes values between zero and one. Initially it was created to indicate how long time there had been non-zero measurements on F2. It increased with the amount 1/nbrsamples (*nbrsamples = number of samples it would take beta to go from zero to one*) for each sample a force was measured and decreased with the same amount for every sample that there was no force measured. Later this was modified a little so that it also takes into account how large F2 is when it increases. It still decreases with the same velocity as from the beginning though. The function that calculates beta, Calbeta(...), can be seen below:

```
double calbeta(double F1, double F2, double nsmp, double beta)
{
        double res = beta;
        if(absd(F2) > 0.0 && absd(F1) > 0.0 && signd2(F1) != signd2(F2)){
                res = beta + absd(F2)/nsmp;
                if(res > 1.0){
                        res = 1.0;
                }
        }else{
                if(0.0 < beta){
                        res = beta - 1.0/nsmp;
                }
                if(res < 0.0){
                        res= 0.0;
                }
        }
        return res;
}
```

As can be seen, beta is increasing if both F1 and F2 register some forces and if they have opposite signs. If this is not true, beta is decreasing with the amount 1/nbrsamples, as explained above. The functions signd2(...) and absd(...) are help functions that returns the sign of a double argument (returning zero if the argument is zero, otherwise one or minus one) and the absolute value of a double argument respectively.

The next value is filtF2. Recall from above that this was the filtered value of F2. FiltF2 is not used directly itself. Instead it is compared with its last value to detect if the filtered value is increasing or decreasing. The difference is stored in the variable dfiltF2. DfiltF2 is in turn used later in the function that calculates the output velocity to decide the behavior of the control block.

Before the function that calculates filtF2 is called, filtF2 is reset in the case that F2 is zero. If F2 is zero it means that the tool is not in contact with anything. To keep the values in filtF2 in this case instead of resetting them results in a less good performance of the controller. This is due to the fact that filtF2 only is used to decide whether the force on F2 is decreasing or not. If F2 has become zero it means that the tool has lost contact and new values measured on F2 after this moment mean that the tool once again is in contact with the environment. If filtF2 is not reseted this would mean that the previous contact affects the control behavior at the new contact moment, which is not desired.

The filterF2(...) function that calculates filtF2, dfiltF2, topdv and topF2 can be seen below:

```
double filterF2(double F2, double F2old, double F2filter, double* dfiltF2, double Fmax, double
                                        Ts, double Tf, double* topF2, double* topdv, double dv)
{
        double newfilt = F2filter*(1-Ts/Tf)+F2*Ts/Tf;
        double dfiltF2old = *dfiltF2;
        *dfiltF2 = (newfilt-F2filter);
        if(signd2(*dfiltF2) != signd2(dfiltF2old)){
                *topdv = dv;
                *topF2 = F2old;
        }
        return newfilt;
}
```

The new value of filtF2 is here calculated according to:

$$filtF2(k) = filtF2(k-1) \; * \; (1-\frac{T_s}{T_f}) \; + \; F2(k) \; * \; \frac{T_s}{T_f}$$

dfiltF2 is after this calculated as the difference between the new value of filtF2 and the previous value of filtF2. In this way dfiltF2 becomes an indicator of if the filtered value of F2 is increasing or decreasing.

If one now simplifies the expression for dfiltF2, one can notice the following:

$$dfiltF2(k) \; = \; filtF2(k) \; - \; filtF2(k-1) \; =$$
$$= \; filtF2(k-1) \; * \; (1-\frac{T_s}{T_f}) \; + \; F2(k) \; * \; \frac{T_s}{T_f} \; - \; filtF2(k-1) \; =$$
$$= \; \frac{T_s}{T_f} \; * \; (F2(k) - filtF2(k))$$

Hence, dfiltF2(k) is merely a comparison between F2 and its filtered value. If F2 has a certain constant level but with some noise, filtF2 will be quite constant too and the noise on F2 will thus be present also in dfiltF2. This is a problem that has to be dealt with later in the function that calculates dv since it then use the sign of this signal to decide the control behavior.

The variables topdv and topF2 are set when dfiltF2 changes sign. In other words, when the filtered value of F2 stops increasing and starts decreasing, or the other way around, new values are put into topdv and topF2. The problems that topdv and topF2 are set quite often if filtF2 is somewhat constant and F2 is noisy so that dfiltF2 changes sign often, doesn't really matter. This is due to that dfiltF2 in that case will be quite low and topdv and topF2 will not be used in the control in this case. Of course, one could argue that noise with a high amplitude on F2 could still cause dfiltF2 to reach a high magnitude and still switch sign often. That is correct, but on the other hand one has to expect at least some quality of the signals in use and if noise with a high amplitude is present on F2 maybe one has to do something about that instead. Also one could raise the level that dfiltF2 has to have to use topdv and topF2 to still try to keep a satisfying control.

After beta and filtF2 have been calculated, a couple of performance increasing exceptions are handled. These exceptions are only performed if |F2| > Fmax. The first one says that if F1 and F2 have different signs, beta should instantly be set to one. This is done since if F2 is larger than Fmax, the control should only be run by F2 and if F2 and F1 have different signs it should have a more restricted behavior, which is indicated by beta being one. Also, if |F2| > Fmax the control will only be influenced by F2 either way and setting beta to one in this case ensures that the control will be transferred back to F1 in a smooth manner.

The second exception says that if the current velocity does not have a sign that is in conformance with the direction that F2 wants to move the robot in, the velocity should be set to zero. This is done in order to not have to wait for the impedance relation to change direction of the velocity, hence improving the reaction time of the control a little.

To improve the control and increase the chattering resistance of the controller, one last help signal is used. This signal is called *boost* and is basically a measurement of how long time since last occasion beta was one. See the code for calculating boost below:

```
void betabooster(double* boost, double beta, double nbrsamples){
        double res = 0;
         if(beta == 1){
                res = nbrsamples;
        }else{
                res = *boost-1;
                if(res < 0){
                        res = 0;
                }
        }
        *boost = res;
}
```

If the boost signal has not reached zero and if beta once again starts to increase, beta is instantly set to one, thus boosting beta in a manner. One could see this situation as one where the robot has crashed into a wall/object and failed to stay in contact with the surface with a low value on F2. If the robot then is moved back into the wall/object the control is directly transferred to F2.

The last two important steps in the control loop is to calculate the Ftot value (recall that this was the merged value of F1 and F2) and to, of course, calculate the output velocity. The calcftot(...) function can be seen below:

```
double calcftot(double F1, double F2, double Fmax, double beta)
{
        if(absd(F2) < Fmax){
                /*Purpose here is to if possible not reduce importance of F2*/
                if((signd2(F2) == signd2(F1) || absd(F1) < 0.5) && absd(F2) > absd(F1)){
                        return F2;
                }else{
                        return ((absd(F2)/Fmax+beta*(1-absd(F2)/Fmax))*F2 + (1-beta)
                                                                *(1-absd(F2)/Fmax)*F1);
                }
         }else{
                return F2;
        }
}
```

Basically, if |F2| is larger than Fmax, F2 is returned. Otherwise a merged value is calculated and returned. The merging formula mainly increases the importance of F2 depending on how large the factor |F2|/Fmax is. The formula also decreases the influence of F1 depending on how large beta is. In the case where |F2| is less than Fmax and there are small, or no readings on F1 the importance of F2 might be reduced of the formula without it making much sense to do so. Therefor an exception has been added that returns F2 if F1 is small and F2 is larger than F1.

The last function in the control for-loop that will be gone through is calcdv(...). This is a quite large function that has a great importance since it decides the behavior of the control block. Basically this function is build on an original impedance control function but has now been extended and modified. It only calculates values if a mass has been specified in the impedance parameters. Originally this function could also handle some cases without a mass being specified. Since this functionality mostly made the modifications more cumbersome and since it wasn't something that was really needed, this functionality has now been removed.

The function can be separated into blocks that handle different situations. First the function controls what sign F2 has. If the sign is positive or negative the function does pretty much the same thing. However it needs to change a little depending on the sign of F2 and some comparisons etc. Together these are therefor two cases. Yet another case is if F2 is zero since F2 then is not in contact and the behavior is a little different in that case. The code for the function can be seen below:

```
double calcdv(double Ftot, double F1, double F2, double F2old, double Fmax, double m, double
d,              double k, double dv, double pos, double Ts, double mdvc, double beta,
                double* topdv, double* topF2, double* filtF2, double* dfiltF2)
{
  if(m==0.0){
        return 0;
        if(d==0){
                return 0.0;
        }else{
                return (Ftot/d-k/d*pos);
        }
  }else{
        double res = 0;
        double F2lim = 4;
        double F2lim2 = 8;
        double stoprate = 0.25;
        double stoprate2 = 0.5;
        double difflim = 2;
        double breakrate = 0.75;
        double breaklim = 100;
        double nslim = 50;
        if(absd(F2) > 50){
                difflim = absd(F2)/20;
        }

     if(F2<0){
        if(((absd(F2old)-absd(F2)) > difflim) && beta > 0.9){
```

```
				if(absd(F2) < F2lim2){
					res = stoprate2*dv;
				}else{
					*topF2 = F2old;
					*topdv = dv;
					*filtF2 = F2;
					double ns = absd(F2)/(absd(F2old)-absd(F2));
					if(ns > nslim){
						breakrate = 1;
					}else{
						breakrate = ns/nslim;
					}
					if(absd(F2)<breaklim){
						res = breakrate*dv;
					}else{
						res = dv;
					}
				}
			}else{
				if(((*dfiltF2) > 0) && beta > 0.9){
					if(absd(F2) < F2lim){
						res = stoprate*dv;
					}else{
						if((*dfiltF2)>0.05){
							res = (*topdv)*absd(F2)/absd(*topF2);
						}else{
							double fct = 1 - (*dfiltF2)/0.05;
							res = (dv+fct*Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
							*topdv = dv;
							*topF2 = F2old;
						}
					}
				}else if((*dfiltF2) < -0.1){
					res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
				}else{
					if(absd(F2) < F2lim && beta > 0.9){
						res = stoprate*dv;
					}else{
						res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
					}
				}
			}
		}else if(F2>0){
			if(((absd(F2old)-absd(F2)) > difflim) && beta > 0.9){
				if(absd(F2) < F2lim2){
					res = stoprate2*dv;
				}else{
					*topF2 = F2old;
					*topdv = dv;
					*filtF2 = F2;
					double ns = absd(F2)/(absd(F2old)-absd(F2));
					if(ns > nslim){
						breakrate = 1;
					}else{
						breakrate = ns/nslim;
					}
					if(absd(F2)<breaklim){
						res = breakrate*dv;
					}else{
						res = dv;
```

```
                        }
                }
        }else{
                if((*dfiltF2) < 0 && beta > 0.9){
                        if(absd(F2) < F2lim){
                                res = stoprate*dv;
                        }else{
                                if((*dfiltF2)<-0.05){
                                        res = (*topdv)*absd(F2)/absd(*topF2);
                                }else{
                                        double fct = 1 – absd(*dfiltF2)/0.05;
                                        res = (dv+fct*Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
                                        *topdv = dv;
                                        *topF2 = F2old;
                                }
                        }
                }else if((*dfiltF2) > 0.1){
                        res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
                }else{
                        if(absd(F2) < F2lim && beta > 0.9){
                                res = stoprate*dv;
                        }else{
                                res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
                        }
                }
        }
    }else{
        if(beta > 0.9){
                res = stoprate*dv;
        }else{
                res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
        }
    }
        res = limitdv(res, dv, F2, mdvc, Fmax);
        return res;
  }
}
```

The case where F2 is zero, which are the last few lines of the function, is quite simple. In this case the behavior of the controller should mostly be as a normal impedance. This is the case when beta is not larger than 0.9. If beta is larger than 0.9 this means that F2 recently was in contact with the environment and that it was (at least mostly) running the control without the influence from F1. In other words, most of the part of dv is due to the forces on F2, which has now lost contact. It makes sense to reduce this influence of F2 on dv to avoid “ghost”-movements of the robot (movements that are not caused by the programmer and movements that cant be related to any present force). In the case where beta is larger than 0.9, the new velocity is therefor calculated as the old velocity multiplied by a stop rate (that is less than one).

As previously stated, the other two cases, when F2 is not zero, are pretty much the same. The case of F2 being larger than zero will now be explained. This can be divided into two different subcases. The first case takes care of the special case where F2 is considerably smaller than its previous value. Considerably smaller is defined through the value difflim. Beta also has to be larger than 0.9. If this occur it is considered that F2 is decreasing fast and a restrictive behavior where dv is decreasing, in other words braking, might be achieved. The reason for this is that if the robot is pushed into an object, a wall or something similar, the desired behavior would be that the robot stays in contact with the wall until the operator moves it away. At a crash into a wall, the force on F2 will increase and cause the robot to stop and move away from the wall, thus decreasing the contact force. To not lose contact with the wall, the controller needs to stop the robot before F2 becomes zero. If the force is decreasing fast it needs to slow down fast enough to not lose the contact. This is done by estimating how many samples it will take until the robot lose contact. If this is less than a certain number of samples, nslim, a rate at which the controller will slow down is calculated and this value is multiplied with the previous output velocity to create the new output. Also, this only occur if F2 is under a certain level specified by breaklim. Another addition is that the velocity will decrease if F2 is under a certain limit, F2lim2, independently of the above. All these limits give an opportunity of tuning in the controller after one's desires.

The second subcase occurs if F2 is not decreasing very fast. This should be considered as the normal case and it can in turn be divided into three different new subcases. Which of these three cases that will be run is decided by the size of dfiltF2. Recall that this value describes how fast and in which direction filtF2 is changing.

If dfiltF2 is larger than 0.1 (considered high) the controller will work as a normal impedance. If dfiltF2 is negative and beta is larger than 0.9 a restrictive behavior of the controller will be used. The reason for this is, as above, that the desired behavior of the controller should be to stay in contact with an object, but to still limit F2. If dfiltF2 in this case is smaller than -0.05 (decided through log results from tests on robot) the new value of the output will be calculated from the topdv and topF2 values as described in the concept part above.

If dfiltF2 is between zero and -0.05 a modified impedance relation is used to calculate the output velocity. Basically, this is a standard impedance formula with a limit on how much the force causes the velocity to increase. In this way the speed can still increase if dfiltF2 is small and negative. If filtF2 would stay constant, an increase of velocity would thus still occur.

If neither dfiltF2 is larger than 0.1 or if dfiltF2 is negative and beta is at least 0.9, a default case occurs. This case is similar to if dfiltF2 is larger than 0.1. In addition to the impedance relation that exist in the case where dfiltF2 is larger than 0.1, a case that reduce the output velocity if beta is larger than 0.9 and F2 is small enough, is present.

The last thing that is done in the function that calculates dv is that a function is called that limits the rate dv is changing if there is a value on F2. The reason for this is a safety precaution to keep the velocity of the robot from changing too much when run by F2. After calcdv(...) is called the new velocities are also limited in their size to avoid to large outputs.

The last thing that is performed in the discrete update function is that a check of if f_switch is turned off is done. If this would be true, the current position of the TCP in the base frame is saved into the discrete states that specify where the center of the safety zone is located. The rest of the discrete states are set to zero. After the f_switch stage, the values of all variables are saved away in the discrete states for next update and to be read of the function that sets the output of the control block.

4.2.3 The safety zone block

After the velocity outputs are calculated in the s-function controller block, these signals are filtered through a safety zone block. The objective of this block is to only let the robot move in a certain zone in space and to also make sure that the output velocity only is in the controlled directions (in case less than three control directions are used). A note that should be made is that the block assumes the inputs to only contain velocities in the controlled directions.

The code for the safety zone block can be found in Appendix C.2 and will not be explained here. The main idea however, is to first remove the components of the input velocity that would make the robot move out of the safety zone. After that is done, the new velocity is adjusted to only give an output in the controlled directions. If that wouldn't be possible the output will be zero.

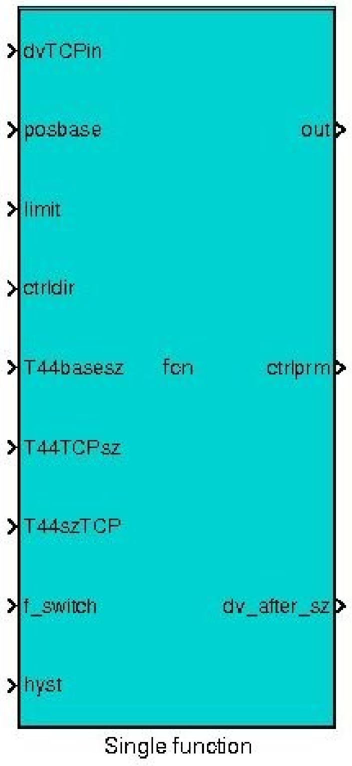


Fig 4.5: The safety zone block. Some outputs are mainly used for analytical reasons.

To achieve its purpose, the block has a number of inputs (see Fig 4.5). The first input is the velocity vector as expressed in the TCP frame. Next are the controlled directions and a value for the limits of the safety zone. The safety zone is defined as a cube with its center in the initial position of the TCP as expressed in the base frame when f_switch was set to one. The distance from the safety zone center that a side of the box is, is defined by the limit value. The orientation of the safety zone frame doesn't necessarily need to be the same as the base frame. Hence, a T44 matrix that describes how the safety zone is rotated relative the base frame also need to be sent in to the Simulink block. Note that the code in the block only use the rotational part of the T44 in this case and that the position part does not change the position of the safety zone (which was decided by the initial position of the TCP when f_switch was set to one as stated above). T44s for describing the rotations from the TCP to the safety zone and vice versa also need to be sent in to the block.

The relative position of the TCP from its initial position when f_switch was set to one, as expressed in the base frame, is sent into the block so it can decide when to perform the safety zone restrictions. To avoid possible chattering problems while moving along a surface of the safety zone, a hysteresis effect is implemented. Since this effect needs some memory functions to work, a couple of variables are declared as persistent (this is done in Matlab code, its similar to static in C). To make sure that these variables don't contain old values, they are reset when f_switch is set to one, why f_switch needs to be sent into the safety zone block. Also a parameter that sets how much the hysteresis effect should be must be sent in to the block.

These are all the inputs to the block. On the other side of the block are the outputs. Mainly this is only one output which is the filtered TCP velocity vector. The other two outputs are just there to make it possible to monitor internal parameters and signals.

4.3 Simulations

Before any tests have been performed on the robot, the controller that will be used has been simulated through various situations to make sure it doesn't behave unexpectedly or bad. Of course, this has also been done for this final version of the code and the different simulation cases and results will now be explained. For the graphs that belong to these simulations, please see Appendix C.3 and C.4.

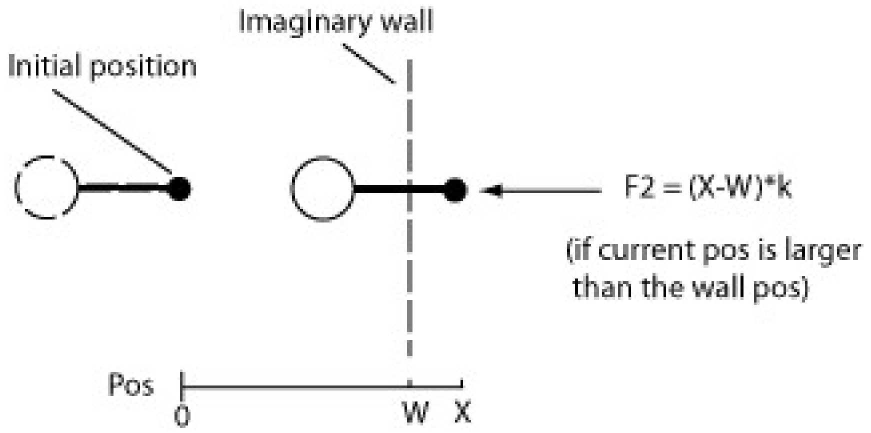


Fig 4.6: How the wall is defined in simulations 1-6

4.3.1 Simulations of the controller

Simulations 1-6
The first six simulations are all wall crash scenarios. Basically what has been done is that a wall has been defined to be located at a certain distance from the initial position. The value of F2 has then been set to increase linearly with the distance

the TCP is beyond the start position of the wall, see Fig 4.6. In other words, the wall pretty much works as a spring with a certain stiffness.

The first three simulations are with a wall with the stiffness 10N/mm. F1, that is used to move the robot towards the wall, has the values 1N, 15N and 50N depending on which simulation that is run, hence accelerating the robot to different velocities at the impact with the wall. The next three simulations are pretty much the same with the difference of the stiffness being set to 500N/mm instead.

If one looks at the graphs from the simulations, one can see that beta become one and the control is transferred to F2 at a crash into a wall. The output velocity changes direction to stop the robot and to move it back to reduce F2 to a low value. Here the robot stops since F1 (run by the imaginary programmer of the robot) still implies that the robot should be moved in the direction that increases the measurements on F2.

A thing that can be noted is that the force on F2 still becomes quite large before it starts to be reduced. This is not really a good thing since it would mean that the forces on the tool, which should be protected, still become large. The large forces are due to the fact that the walls have been set to have quite high stiffness. If the impact with a wall occur right after a sample has been made, it would take one sample until the controller actually detects the impact. Next, it also takes some time for the controller to calculate the new output and, moreover, it takes some extra time before the actuators actually affect the output. If the sample time as in this case is 4 ms and the velocity of the robot at the impact is 100 mm/s (the maximum allowed velocity) and the stiffness is 500N/mm, it quite easily can be calculated that the increase of force for each sample delay is 200N. In other words it is hard to reduce this force buildup in the controller. The best thing to do would be to reduce the sample time and the system time delay. Also one should build in some compliance in the robot tool if it is possible to avoid large forces that are built up at crashes into stiff objects. The positive thing is that this force is being reduced relatively fast after the impact.

Simulations 7 and 8
These two tests only have force inputs on one of the force sensors, F1 in simulation 7 and F2 in simulation 8. The force input is a sequence with the force either being one, zero or minus one. The purpose is to see how the controller behaves when only run by one force sensor and to see how similar the output is when the input comes on F1 or on F2. As can be seen they are pretty much the same and it is not much more to say about it.

Simulations 9 and 10
In these tests it is looked upon how a temporarily input on F2 under the same time as F1 is active affects the output. The purpose is to see how the output changes when the controlling input goes from F1 to F2 and then back again. F2 has opposite sign in the different tests to see the difference of when F1 and F2 has the same direction and when they do not.

Simulation 11
This simulation has a flat input on F1 and then gets a disturbance on F2. The purpose is of course to see how much disturbances, from for example g-forces on F2, affects the output. As can be seen dv gets a dip when the disturbance occurs. This dip should ideally be less than what it is. However it is hard to find a good balance of when considering an input to be a disturbance or, for example, the

result of an impact with a stiff object. In the latter case it is important to have fast reactions of the controller to avoid large force build-ups as discussed above. In the first case it is also important to have some disturbance rejection in order to not get a behavior that is too jerky. The results of this simulation thus have to be seen as a compromise.

Simulations 12 and 13

These two simulations merely check that the controller works as intended when f_switch is turned off. F_switch is turned off at t = 15s. In one of the cases F1 then has an input of one and in the other case it does not. It's not much to say about these tests. They are mostly done as a safety precaution to see that nothing unexpected happens when f_switch is turned off.

4.3.2 Simulations of the safety zone

Since the safety zone was located in a separate block some special tests were run for that. In these tests only F1 was used as a control input since the main objective was to create a velocity output that could be filtered through the safety zone block. It has turned out that it is a little hard to set the initial orientation and position of the TCP to an exact value in the simulations (for various reasons). Hence to get the TCP frame to exactly coincide with for example the base frames rotation has been a little cumbersome and the rotation of the TCP in the simulations can thus not be seen as something exact. The graphs belonging to these simulations can be found in Appendix C.3.

Safety zone simulation 1
This first test used all three control directions. The safety zone was set to have the same rotation as the base frame and so was the TCP. F1 had forces in all three control directions which resulted in velocities in all of the control directions. Notice that the velocities in the TCP are cut off one by one as the robot reaches the walls in the safety zone since the TCP frame and the safety zone frame are aligned. In other words, the components that are cut away from the velocity to achieve the safety zone limitation are equivalent to the TCP components.

Safety zone simulation 2
Same as for test one but with the safety zone frame now being rotated 20 degrees about the z-axis of the base frame and the TCP frame having an angle of about 35 degrees about the x-axis relative to the base frame. F1 only has forces in one TCP direction in this case. The purpose is to slide along the surfaces of the safety zone, which it does. Notice how the velocities in the TCP differ from the original ones sent into the block when the sliding occurs. Also, notice how velocity components are set to zero in the normal direction to a safety zone surface that is reached and how the safety zone surfaces are reached one by one.

Safety zone simulation 3
Same as simulation 2 but with only one control direction being used. Notice how the output become zero in all directions as soon as any surface is reached.

Safety zone simulation 4
As the previous cases have seemed to work, the turn now comes to testing the case of two controlled directions. In this simulation the rest of the settings are as in simulation 2. When the first safety zone surface is reached, a sliding behavior is

still achieved. However, it is different from the sliding in simulation 2. Notice how the intermediate output velocity, that is obtained after the components directed out of the safety zone has been removed, still has a component in the non-controlled direction. After the adaptations to the controlled directions has been made there no longer are components in this non-controlled direction however and there are neither any component directed out from the safety zone. When the second safety zone surface is reached, the velocity should only be allowed in one direction. If this wouldn't be in the control plane the output should be set to zero, which is what is done in this test.

4.4 Test results on the real robot

After the simulations had been performed and analyzed, the Simulink model was considered safe to test on the real robot. The purpose was of course to see that the model worked in the same way on the real robot as in the simulations. The first tests that were done considered the safety zone and checked that it worked as intended since this feature was very important. During the tests, the signals were logged and the results of these tests can be found in Appendix C.5. Next the controller was tested with inputs on both F1 and F2 (see Fig 4.7 and Fig 4.8) to see that it worked as intended. F1 was not mounted on the robot. Instead it was put on the table in the control room. Among the tests was one where the robot was run into a piece of Styrofoam to simulate a crash into a wall (see Fig 4.9). The results of the tests of the controllers can be found in Appendix C.6. The different tests will now be explained one-by-one.

4.4.1 Test of the safety zone

The first test of the safety zone that was performed used all three control directions. The safety zone frame was set to be rotated 30 degrees around the z-axis of the base frame and the TCP was also rotated slightly to get a similar situation as for the previous simulations. As can be seen the safety zone seems to be working since the relative position of the TCP, as expressed in the safety zone coordinates, never gets outside the zone (position limited to be +/-100mm). In the test, the robot is moved around randomly in the zone.

In the second test, only one control direction is allowed. In other words the robot is only allowed to move along a line. When it reaches the first safety zone surface it stops completely as intended. It is not much more to say about this test.

In the third test, the control was set to be performed in a plane. It can be seen that there are no velocity outputs in the non-controlled direction at any point which is good. When the control only is limited by one safety zone surface the robot can still get a sliding behavior which can be seen in the logged signals (look at the TCP velocities in the lower figure of test three in Appendix C.5 before and after they have been filtered by the safety zone block).

4.4.2 Tests of the controller

The first test of the controller was merely to move the robot around with F1 and F2 separately. In these cases the robot should behave more or less like the original impedance controller that was only run by one force sensor. When the controller is run, it has a quite nice feeling. A thing that can be notice is that beta sometimes becomes one which causes the behavior to be restrictive. This is due to the fact that beta will be set to one if the sign of F1 and F2 are not the same and |F2| at the same time is larger than Fmax. Since the case where F1 is zero is considered to have a different sign than F2 if it is non-zero beta in this case become one. The idea behind this is to always make sure that the control is completely left to F2 when |F2| is larger than Fmax. Of course it can be questionable if the case where F1 is zero should be treated like this.

The next simulation used simultaneous inputs on F1 and F2 where F1 and F2 are directed in opposite directions as well as in the same directions. It was noticed during the tests that if F1 and F2 were directed in opposite directions and if F2 then had forced the robot to stop and later was removed, the control went back to F1 again. If F2 soon became active again it instantly took over the control and stopped the robot.

The last test was a wall crash scenario as described above. Only one control direction was used to make it a little more simple to run the tests. When the robot got in contact with the Styrofoam, the forces on F2 increased and the control was taken over of this sensor. The robot stopped and kept the contact with the Styrofoam until F1 again indicated to move away from the surface.

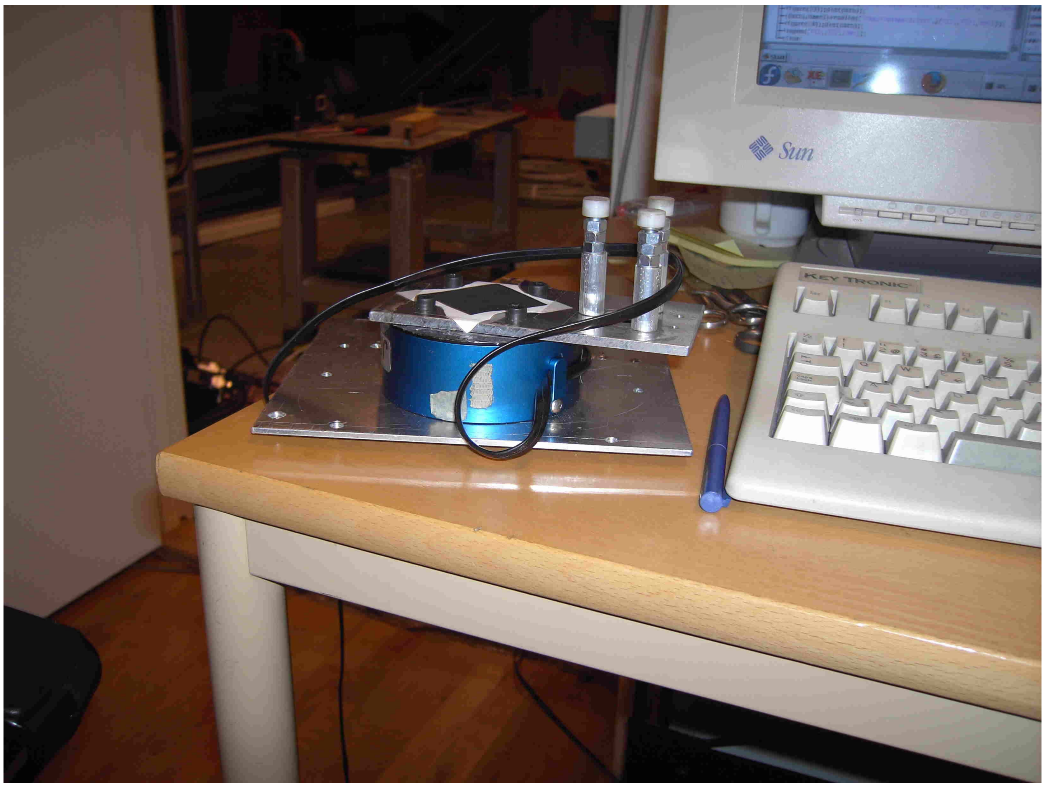


Fig 4.7: The operator sensor, F1 (the blue cylinder), placed in the control room.

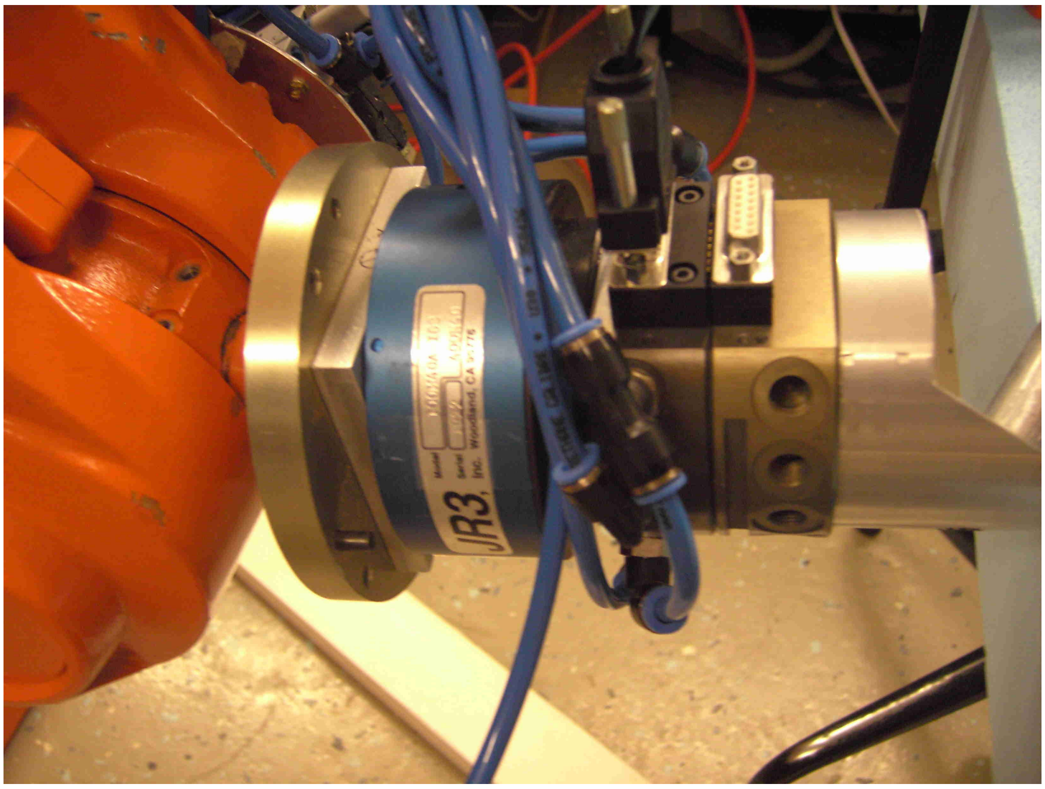


Fig 4.8: The tool protection sensor, F2, mounted directly after the flange of the robot.

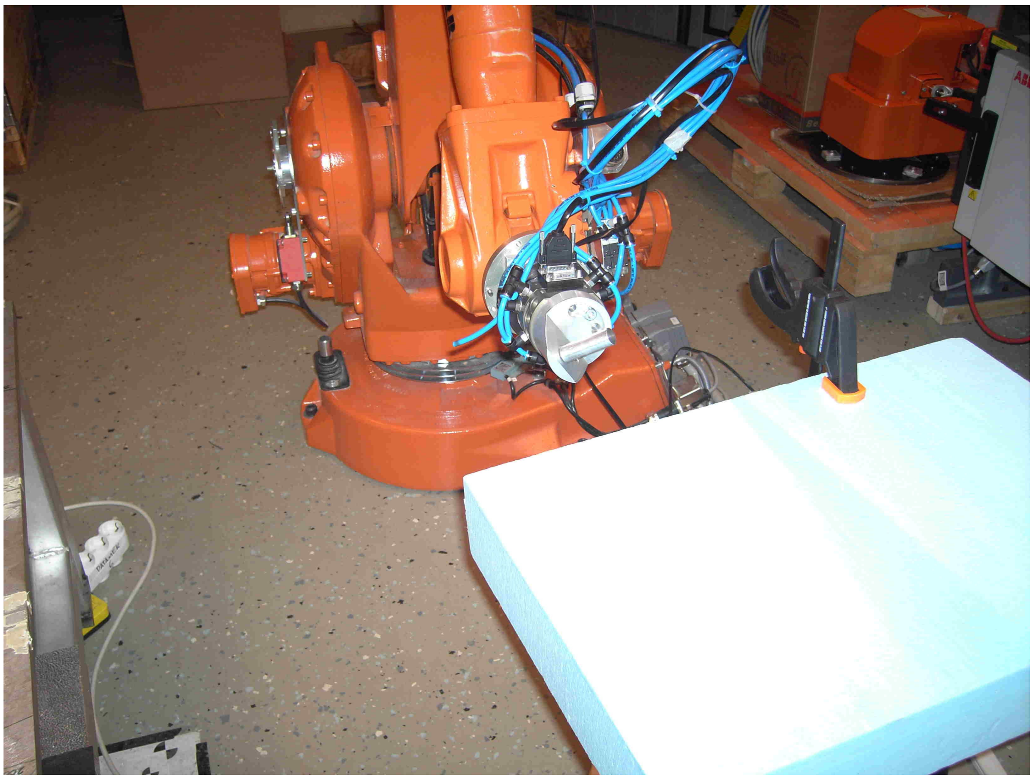


Fig 4.9: The experiment environment with the Styrofoam mounted on a chair, acting as an obstacle in the working cell.

5. Summary, conclusions and future work

The main strategy for the controller that has been developed in this thesis, is to take two separate force sensor readings and combine them through a weight function into one force value. This force is then used as an input to a function that calculates an output signal for the controller. The main idea behind this function is that it, in most cases, should work as an impedance controller.

The longer the time that F1 and F2 disagrees on what that should be done, the more the control is transferred to be run by F2. How long F1 and F2 have been disagreeing is represented by a value called beta. When beta is larger than a certain level it means that F1 and F2 have been disagreeing on what should be done relatively long. When this occurs the controller enters a more restrictive mode and the main task of the controller becomes to reduce the force on F2 to a small value and remain there until F1 (the user) stops disagreeing with F2 and moves the robot in a direction that doesn't increase F2. When the user starts agreeing with F2, beta starts decreasing once again and the controller goes back to its normal state.

The strategy described above works out quite well for the situation where the robot is run into an object. The control is then taken over by F2 and reduces the force to a small value. This means that the robot will stay in contact with the object until it is moved away from it (or until it is removed).

During the development stage of the current controller, some conclusions have been drawn. First, it has been noted that just using a weight function for F1 and F2 and then using the new force reading as input to an impedance controller has not been enough. The reason for this is mainly that it easily lead to that the control is transferred to F2 when contact with an object has been reached. F2 then starts moving the robot away from the object but quite easily loses contact with it. When this is done the user might try to push it back towards the object where the same procedure happens again. In other words, it appears to be hard to stay in contact with an object. Another reason for not considering this strategy sufficient is that it is rather slow, hence not reacting very fast to large forces on F2.

This leads us into the next conclusion, that one needs a way of discovering when the robot has collided with an object. The main reason for this is to be able to decide when to change the behavior of the controller to be more restrictive and thus avoiding large forces on the tool. To be able to stay in contact with the object one also needs a faster way of slowing down than just damping when there still is an active force on F2. This part creates needs for deciding when F2 actually starts to decrease. Since F2 has been proved to be noisy, this has been a bit tricky to do while still keeping the reaction times of the controller.

Another conclusion that has been drawn during the development is that it is hard to handle the cases with the environment being very stiff and to still avoid large forces. However this was not unexpected.

A thing that can be kept in mind is that the final solution presented in this report is build on the so called *Fmod* solution. However, the *impx2* solution can still be considered as an alternative solution if some more effort is put into that solution. Also other possible alternatives could be considered in future work.

The final version of the controller that is presented in Chapter 4 currently seems to be working fine on the robot. There are no known bugs but it should be kept in mind when using the controller that undetected bugs still can be present, why a sense of care should be taken when using the controller. This is a lot due to the fact that it is hard to find ways of validating the control for all possible situations. The strategy for this has been to use simulations and to try to test worst-case scenarios. This works quite well, but of course it is hard to decide all the worst case scenarios, why it is also hard to guarantee the control behavior in all possible situations.

A note that should be made is that the main effort has been put on getting a working control in one control direction. When the control is implemented in three control directions, that is basically done by just utilizing three separate controllers, one for each orthogonal control direction. Since the controller sometimes change its behavior this strategy might give a little weird behavior when the robot is not controlled just along the main control directions. For example, it might be hard to slide with the robot over a surface. Hence, there is a need to put some more effort into how the control should be implemented in three directions.

As previously mentioned the current controller has not got functionality for rotating the TCP, hence this is a topic for future work on the controller. A similar approach as the present one could probably be used. However, care has to be taken to how the current outputs are handled. Notice that the rotational part should be implemented in order to use the controller for lead-through programming functionality since the lack of it limits the usage quite a lot.

Another modification that could be made to the current controller is to increase the generality of the safety zone. The way it is implemented at the moment, it is always defined as a cube. By changing the definition of the safety zone to just be defined by a list of surfaces to form an arbitrary safety zone and on which side of the surface the robot is allowed to be, this generality could be accomplished.

Due to all the current limits of the controller, one has to consider this project to have been more of a study of possible ways of how a second force sensor could be connected to the control and different problems that might occur while doing this.

6. Bibliography

[1] "Robot Dynamics And Control" by Mark W. Spong and M.Vidyasagar, 1989, ISBN 0-471-61243-X
[2] "Introduction To Robotics" (2nd edition) by John J. Craig, 1989, ISBN 0-201-09528-9
[3] "Robot Force Control" by Bruno Siciliano and Luigi Villani, 1999, ISBN 0-7923-7733-8
[4] "Force Control Interface for ABB S4" by Isolde Dressler, Department of Automatic Control, Lund University, 2007
[5] "Introduction To Robot Kinematics And Dynamics" by Magnus Olsson, Mikael Fridenfalk, Per Cederberg, former Division of Robotics, Lund University
[6] http://www.abb.se (2008)

A Denavit-Hartenberg

A.1 – How the frames are attached to the robot and how the link parameters are determined in the Denavit-Hartenberg convention

Attachment of the frames

1. Put the z-axes z_0 to z_{n-1} of the frames through the joints (compare Fig 2.2).
2. Put the origin of the base frame anywhere on the z_0 axis with the x- and y-axes in a suitable way to form a right-hand frame.

   For i=1 to i = n-1repeat the next three steps:
3. Put the origin of frame i where a common normal to z_i and z_{i-1} intersect z_i. If z_i and z_{i-1} would happen to intersect each other the origin should be located at this point and if they would happen to be parallel with each other the origin should be located at the joint i.
4. Put the x-axis of frame i along the common normal to z_i and z_{i-1} or if z_i and z_{i-1} happen to intersect, in the normal to the plane that is made up of z_i and z_{i-1}.
5. Complete the frame with the y-axis so that it become a right-hand frame.
6. The last step in the attachment of the frames to the robot is to locate the n:th frame, the end-effector frame. Let the origin of frame n be coincident with frame n-1. The transformation from this frame to the TCP-frame can most often be done as a static transformation.

Determine the link-parameters

a_i = The distance along the x_i -axis from the origin of frame i to the point where x_i and z_{i-1} intersect.

d_i = Let this be the distance from the i-1 frame origin along the z_{i-1} axis to the point where x_i and z_{i-1} intersect.

α_i =Let this be the angle between the z_i and z_{i-1} axes as measured about the i:th x-axis.

θ_i =Let this be the angle between the x_i and x_{i-1} axes as measured about the z_{i-1} axis.

Notice that the variable of each joint either is d (in the case of a prismatic joint) or θ (in case of a revolute joint). Also notice that the method does not always have the same type of numbering even though it still is referred to as Denavit-Hartenberg. The numbering above is according to course material handouts in Robot Technology, MMT 150, LTH.

B Simulations of the initial exercise

B.1 Results from simulations of the modified border region

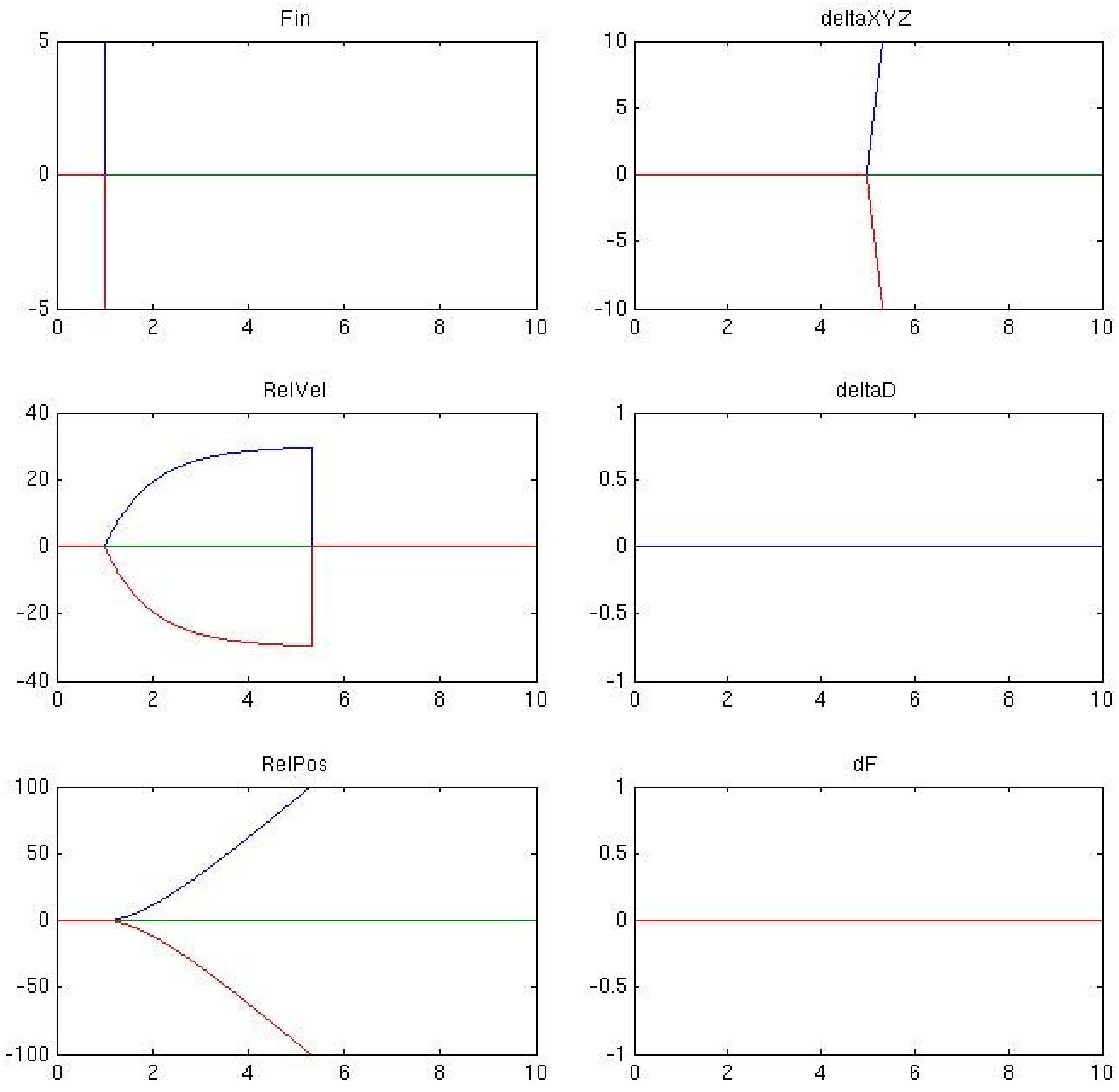


Figure 1 For comments, see discussion in Chapter three.

B.1 Results of the simulations of the modified border region

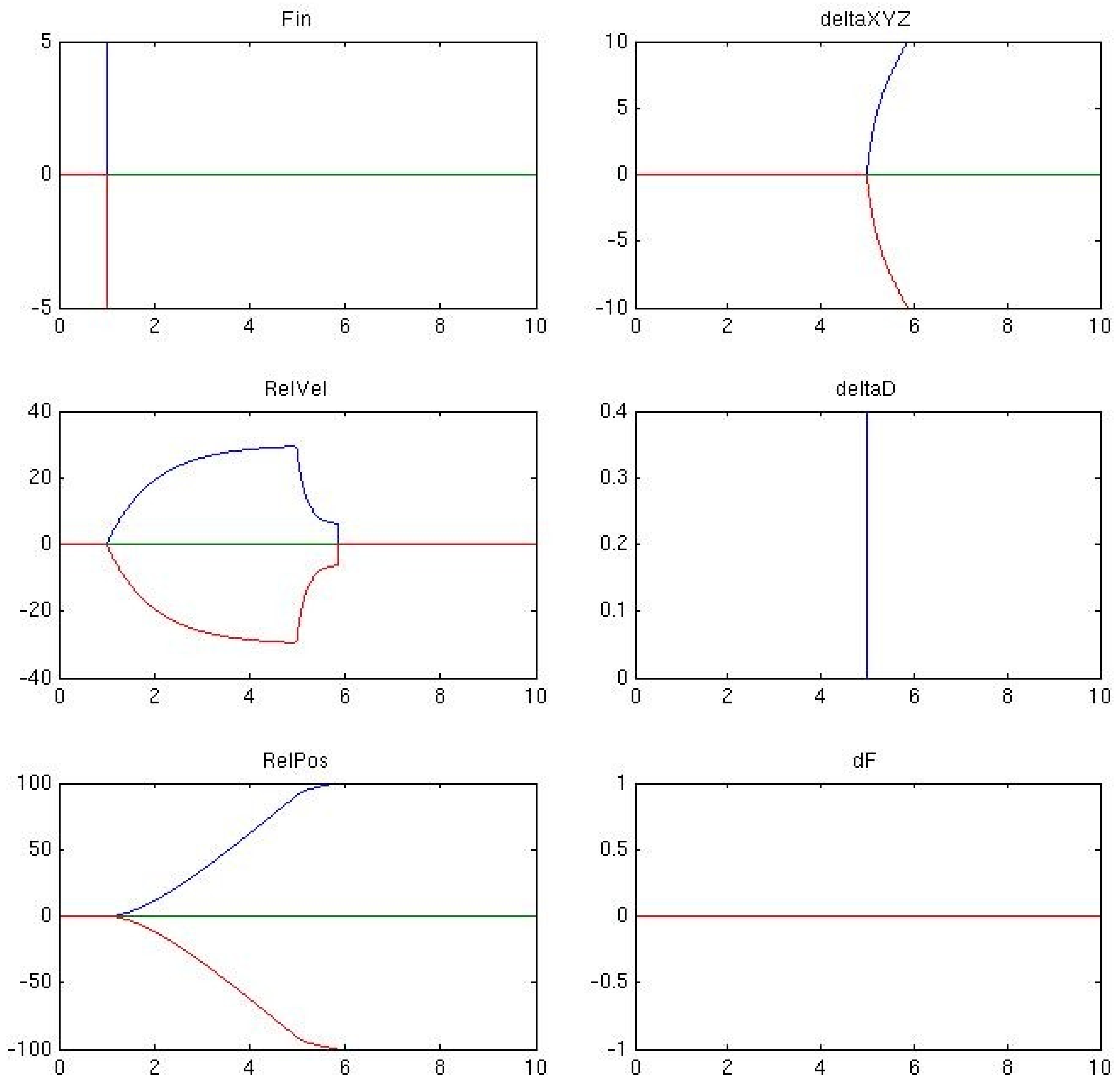


Figure 2

B.1 Results of the simulations of the modified border region

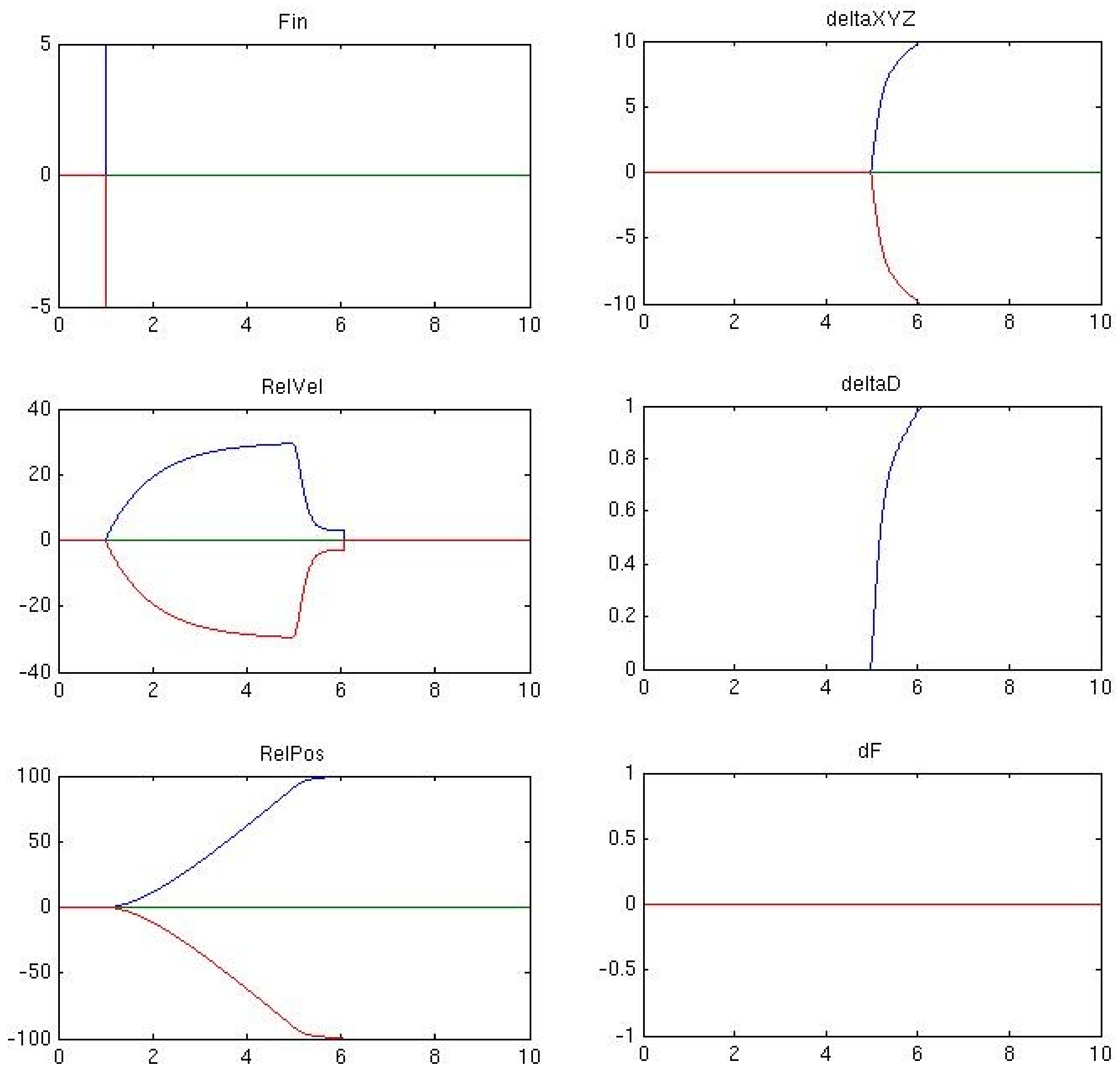


Figure 3

B.1 Results of the simulations of the modified border region

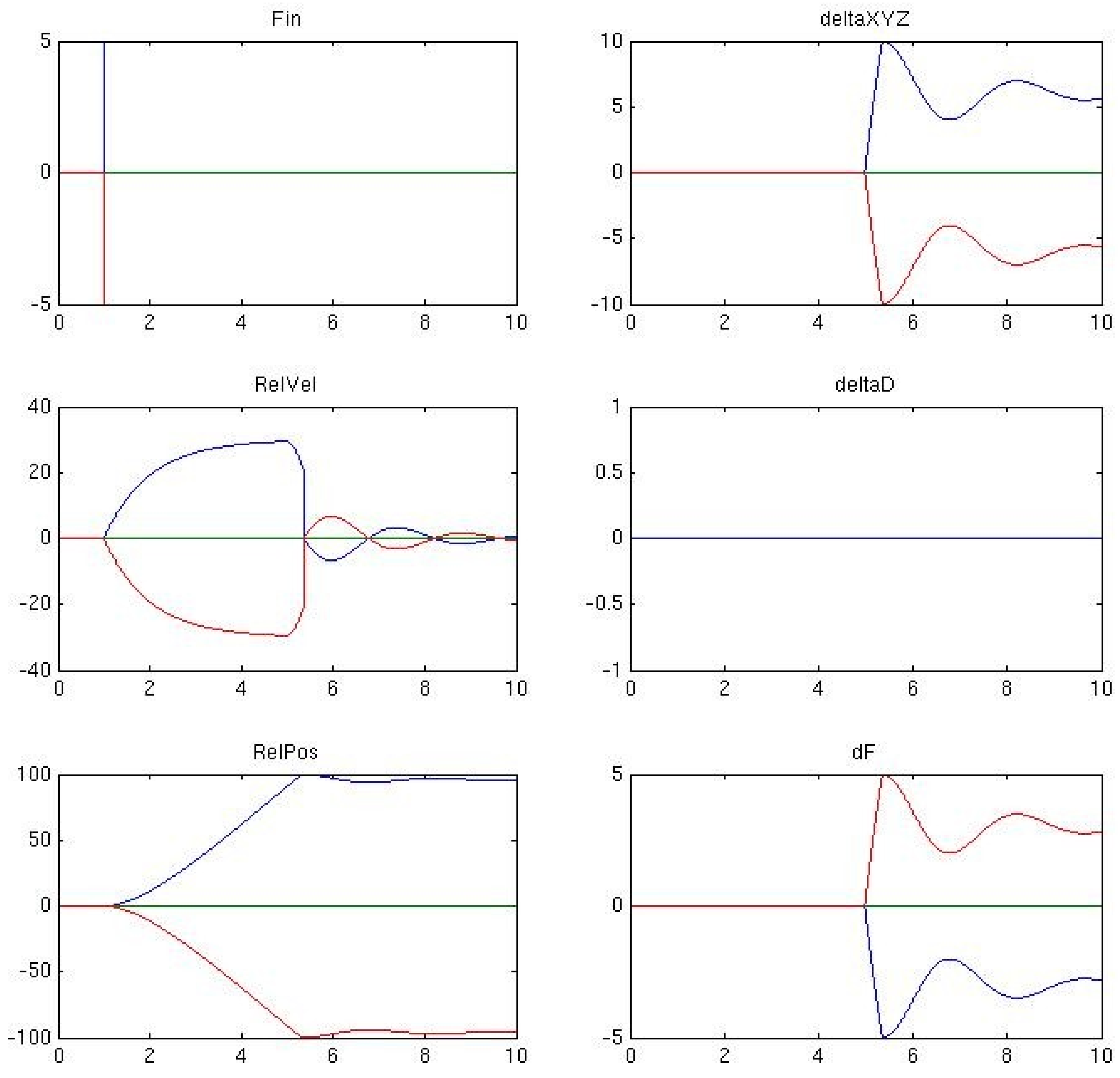


Figure 4

B.1 Results of the simulations of the modified border region

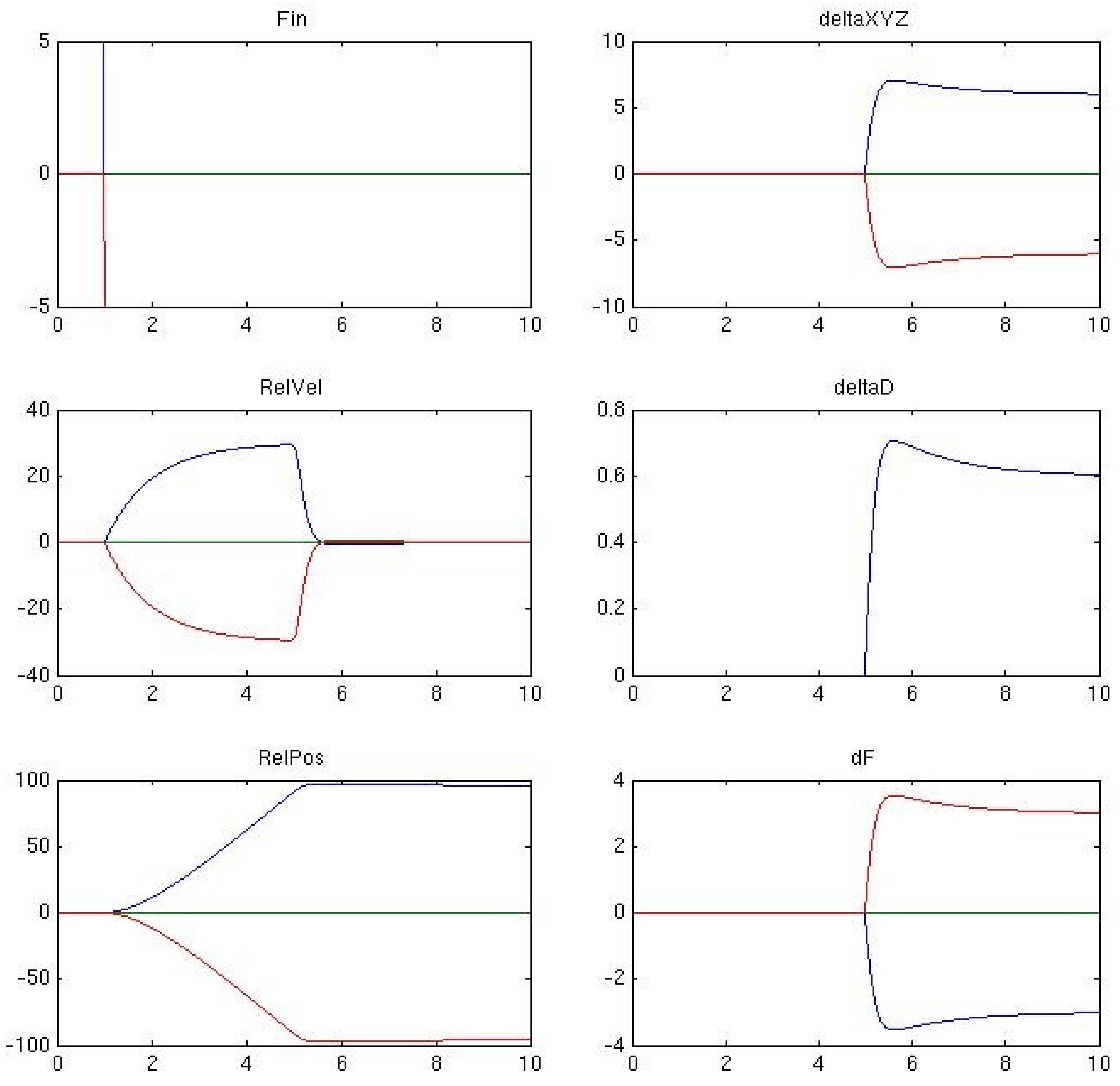


Figure 5

B.1 Results of the simulations of the modified border region

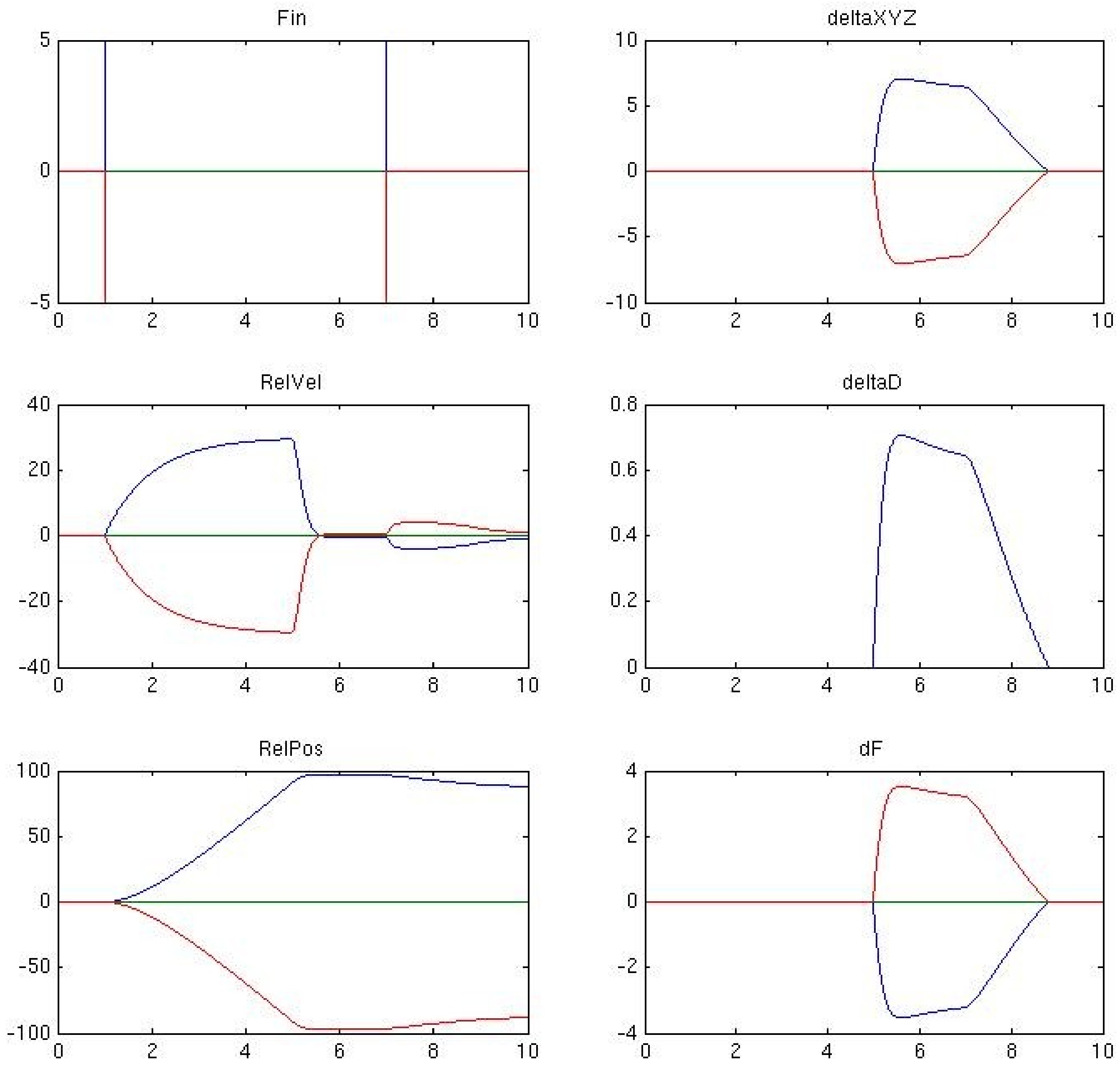


Figure 6a

B.1 Results of the simulations of the modified border region

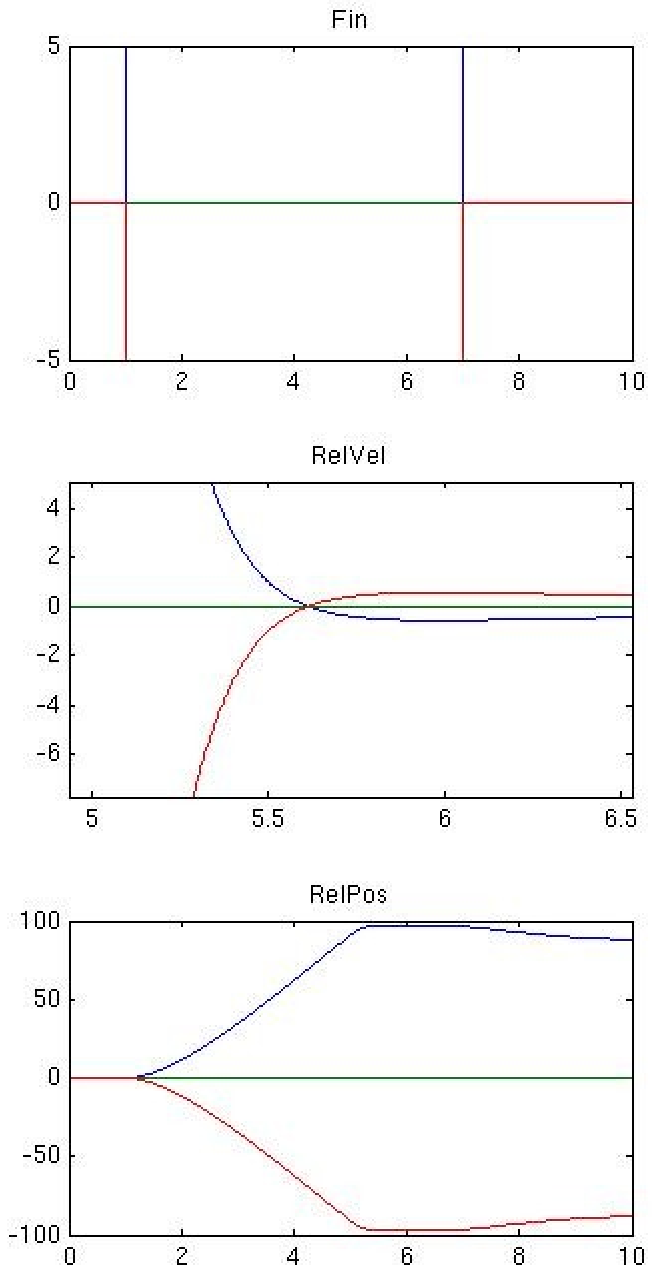


Figure 6b

B.2 Results from simulations of turning f_switch off

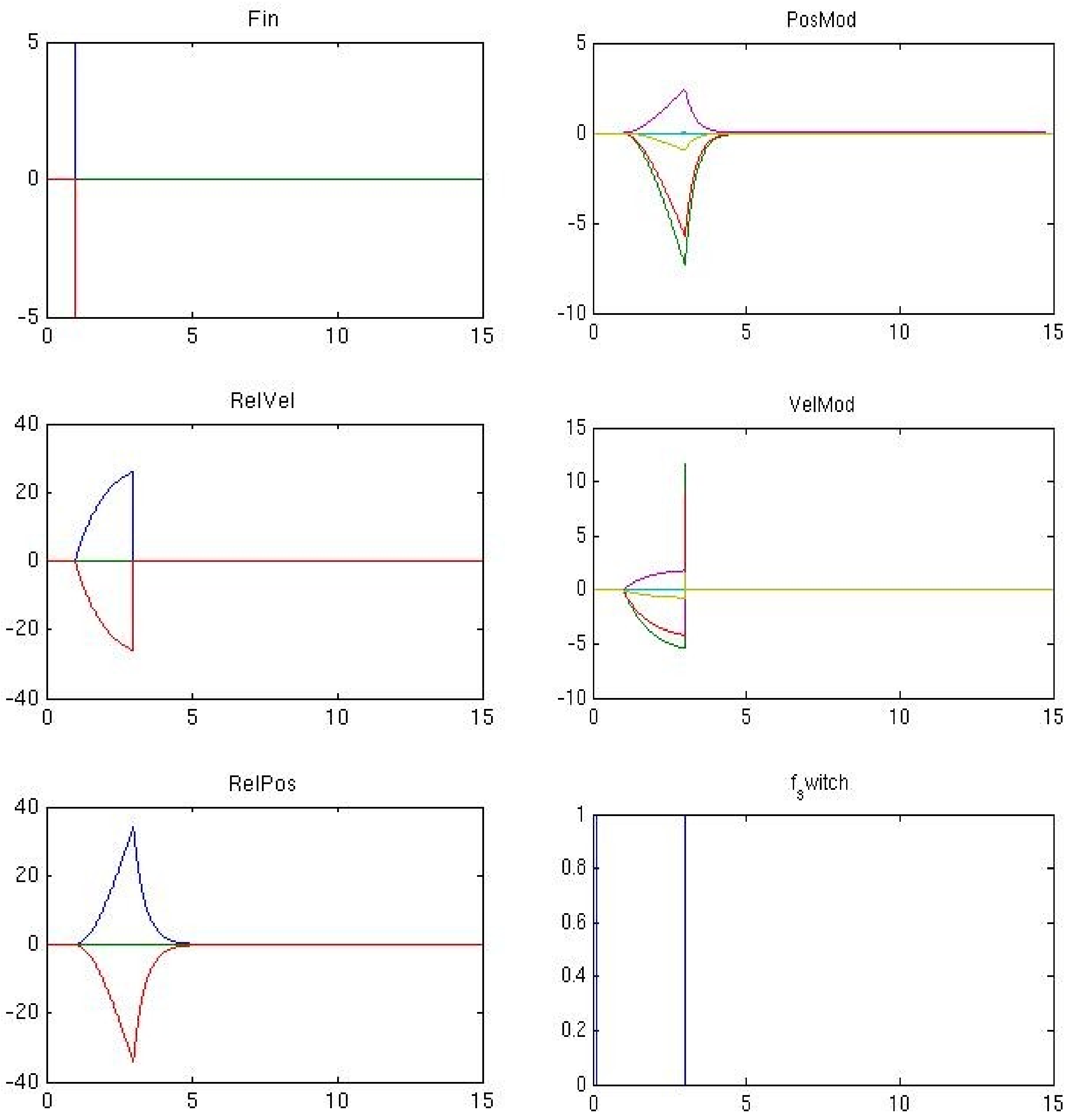


Figure 1 – Turning f_switch off after 3 sec while arm is being accelerated. PosMod and VelMod are the reference signals for the position and the velocity respectively that are sent to the axis controllers of the robot.

B2 Results from simulations of turning f_switch off

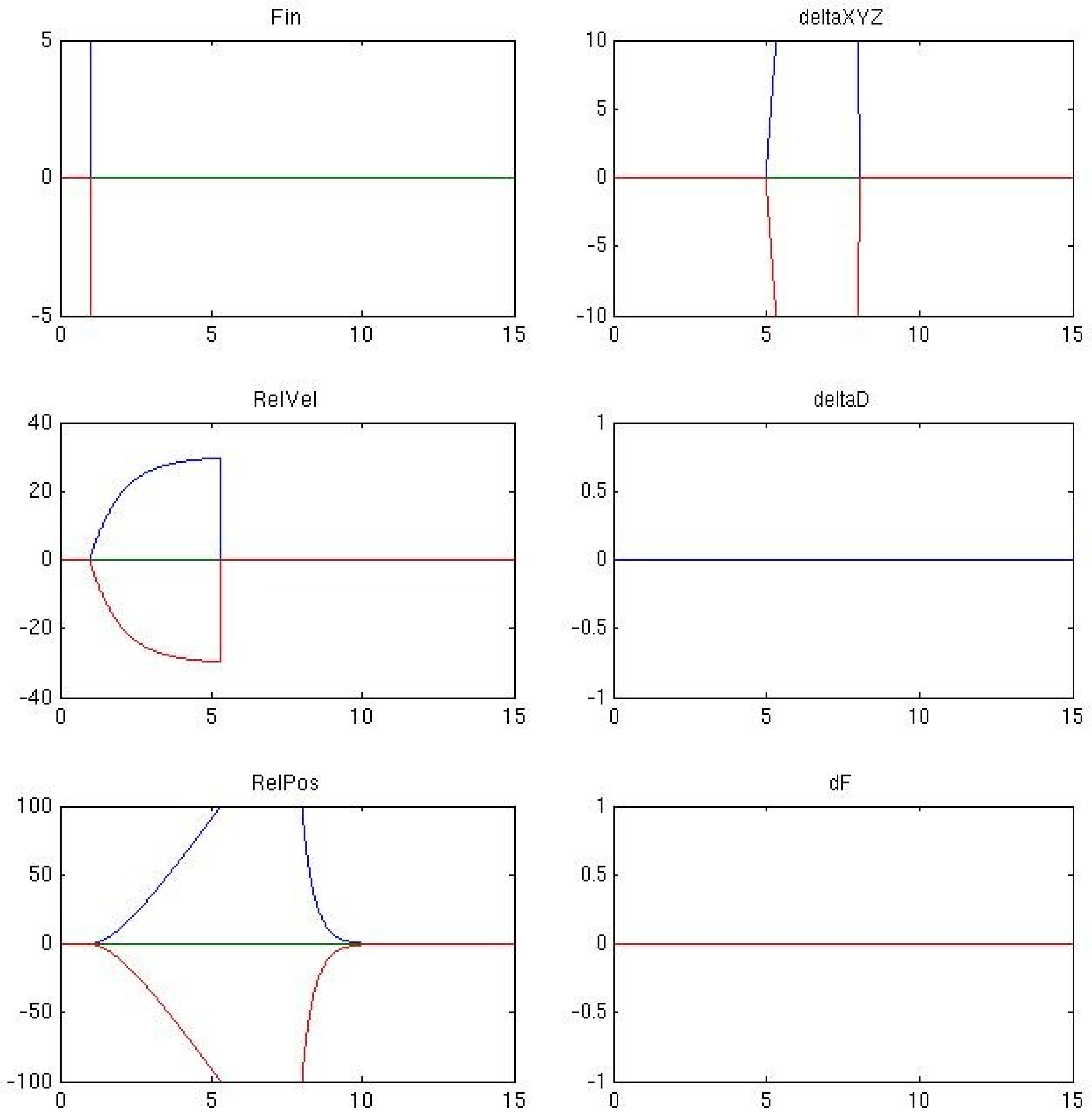


Figure 2a – Turning f_switch off after 8 sec when manipulator has reached safety zone limit

B2 Results from simulations of turning f_switch off

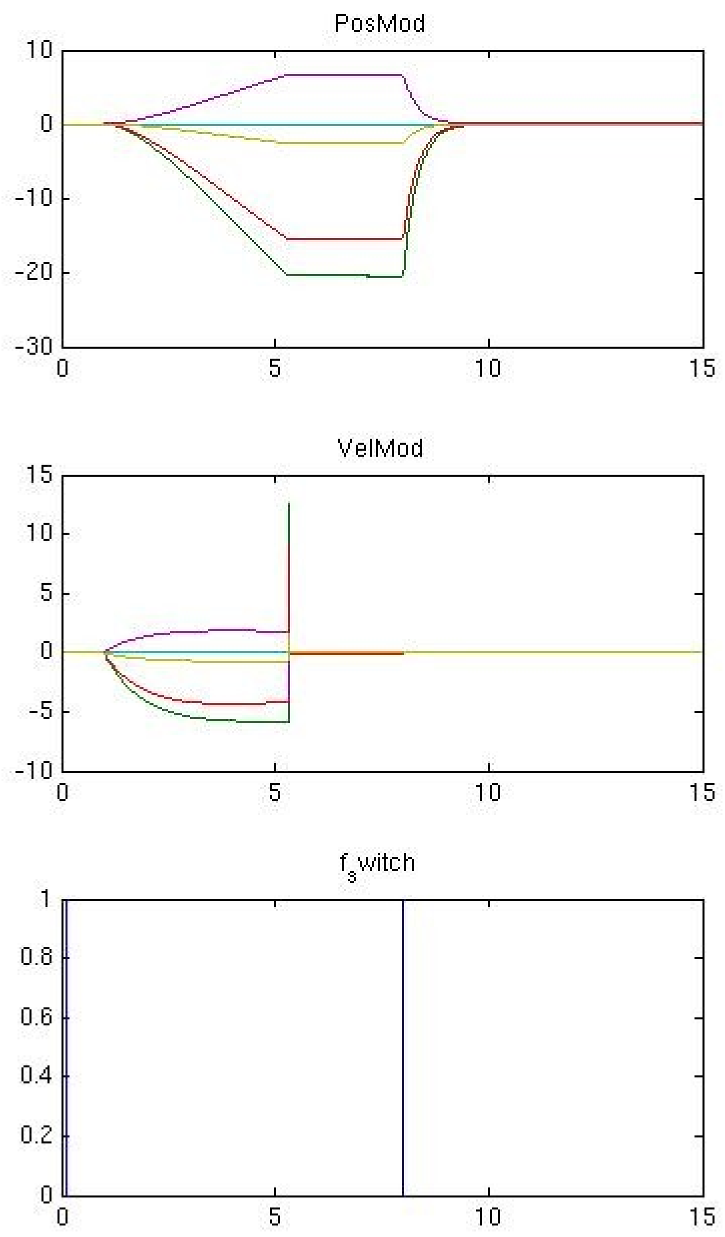


Figure 2b – belongs to simulation in fig 2a

B2 Results from simulations of turning f_switch off

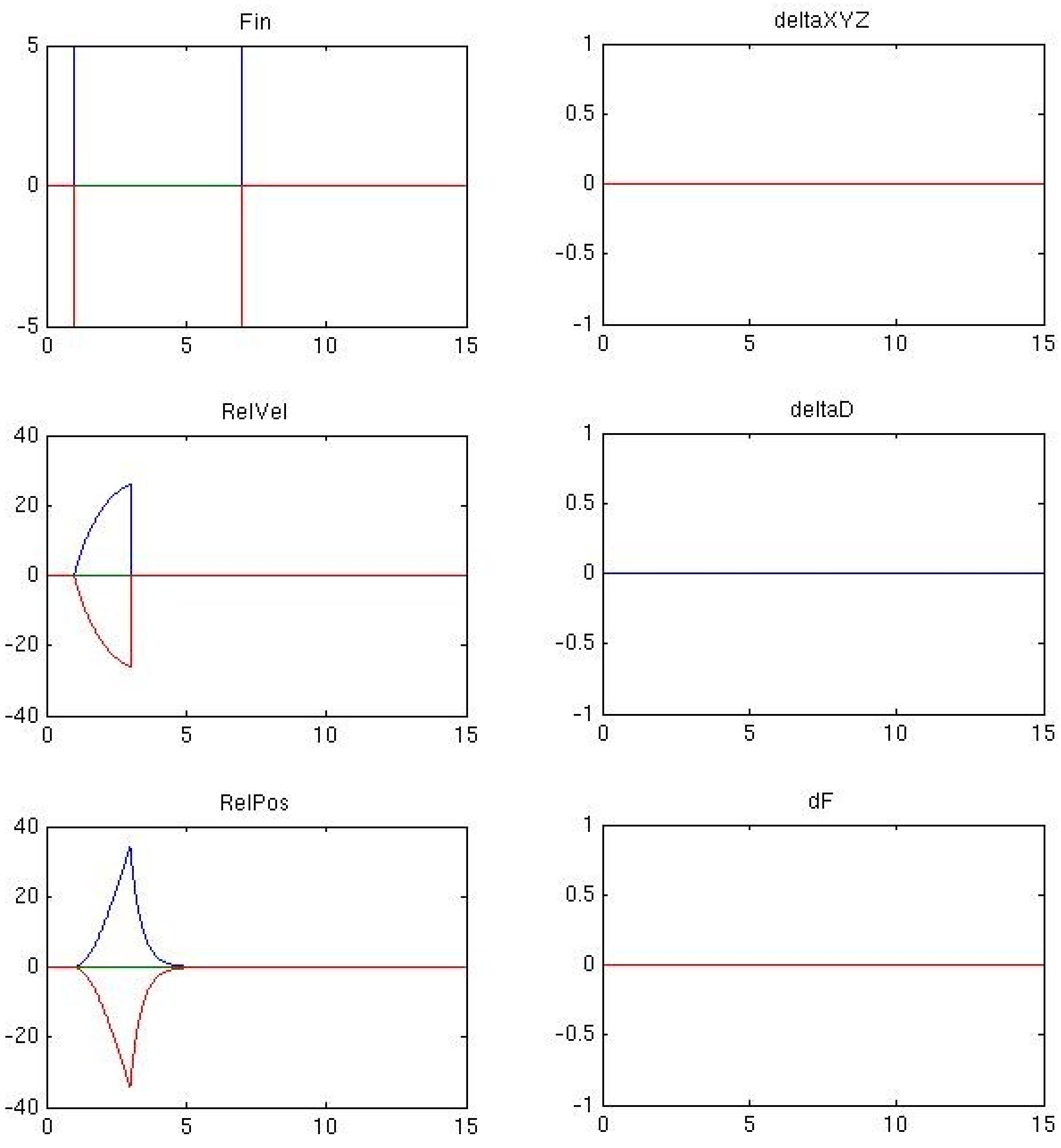


Figure 3a - As in figure 1 but with K_mode and D_mode activated

B2 Results from simulations of turning f_switch off

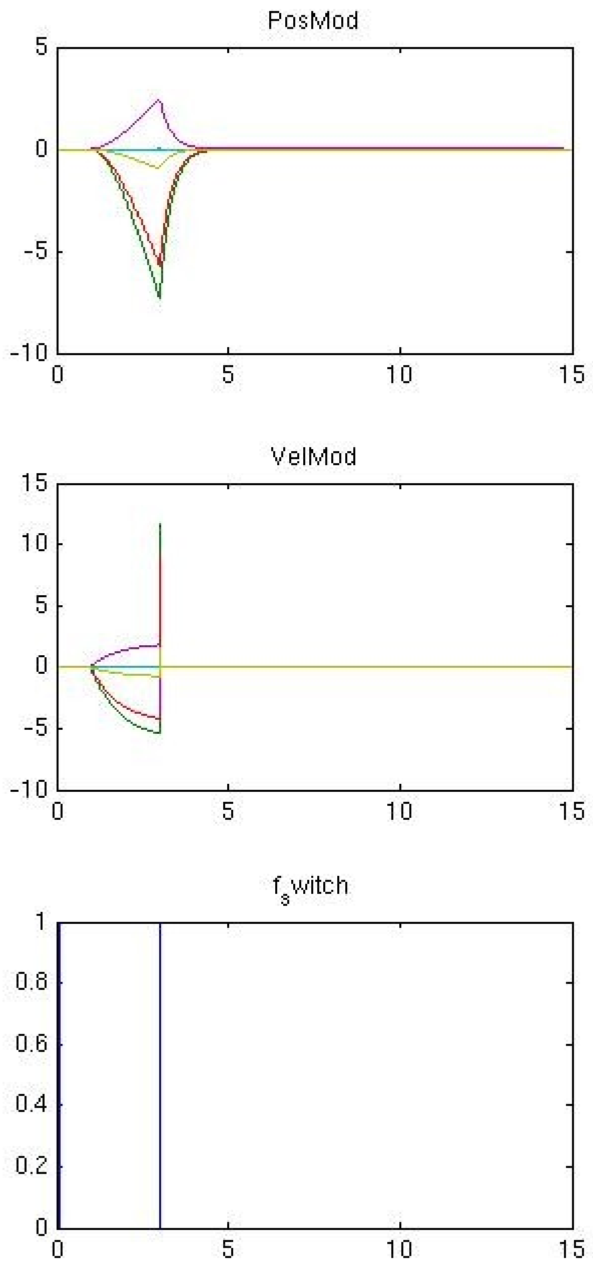


Figure 3b - Belongs to simulation in previous figure

B2 Results from simulations of turning f_switch off

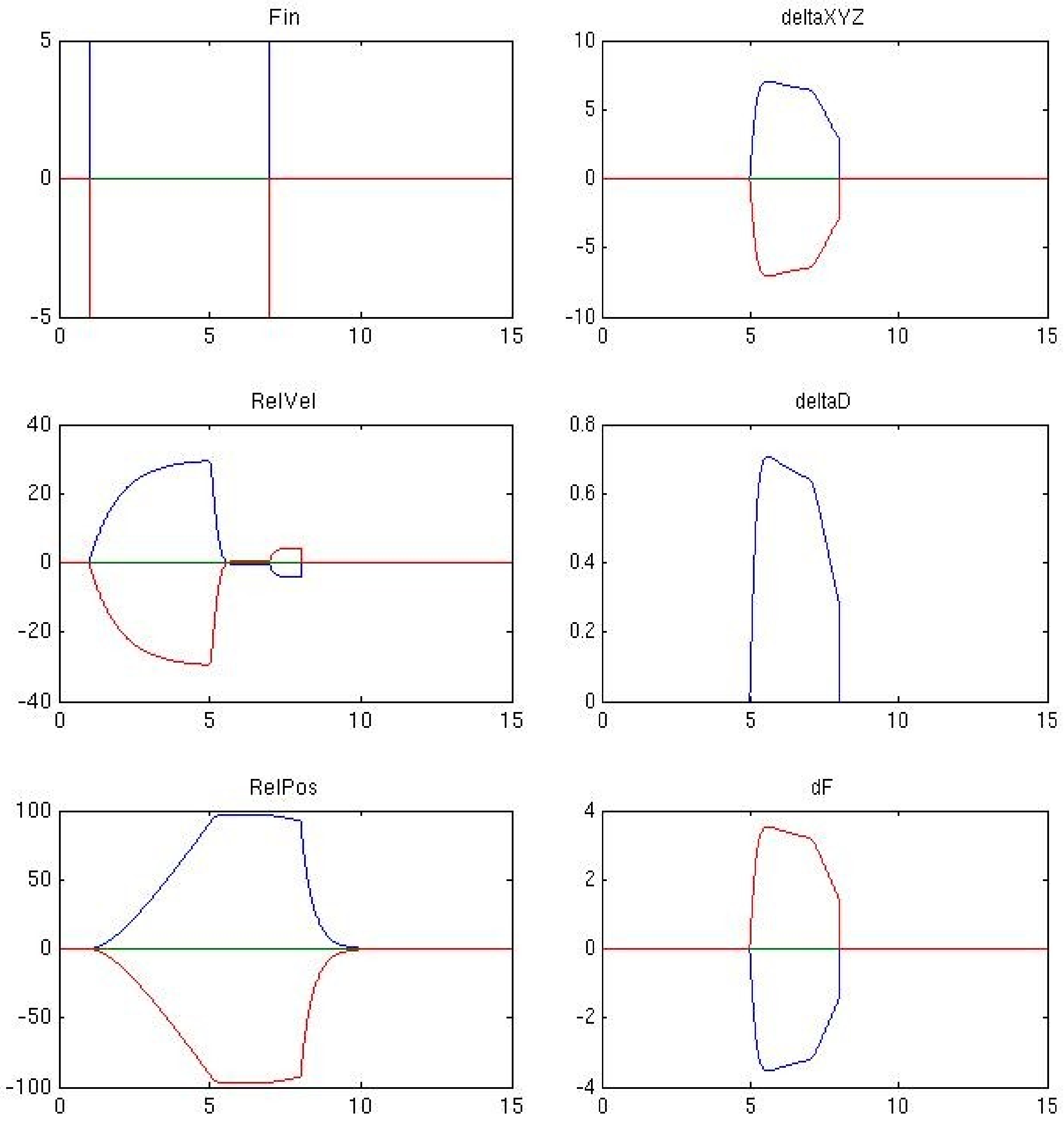


Figure 4a – Turning force input off after 7 sec and turning f_switch off after 8 sec with D_mode and K_mode activated.

B2 Results from simulations of turning f_switch off

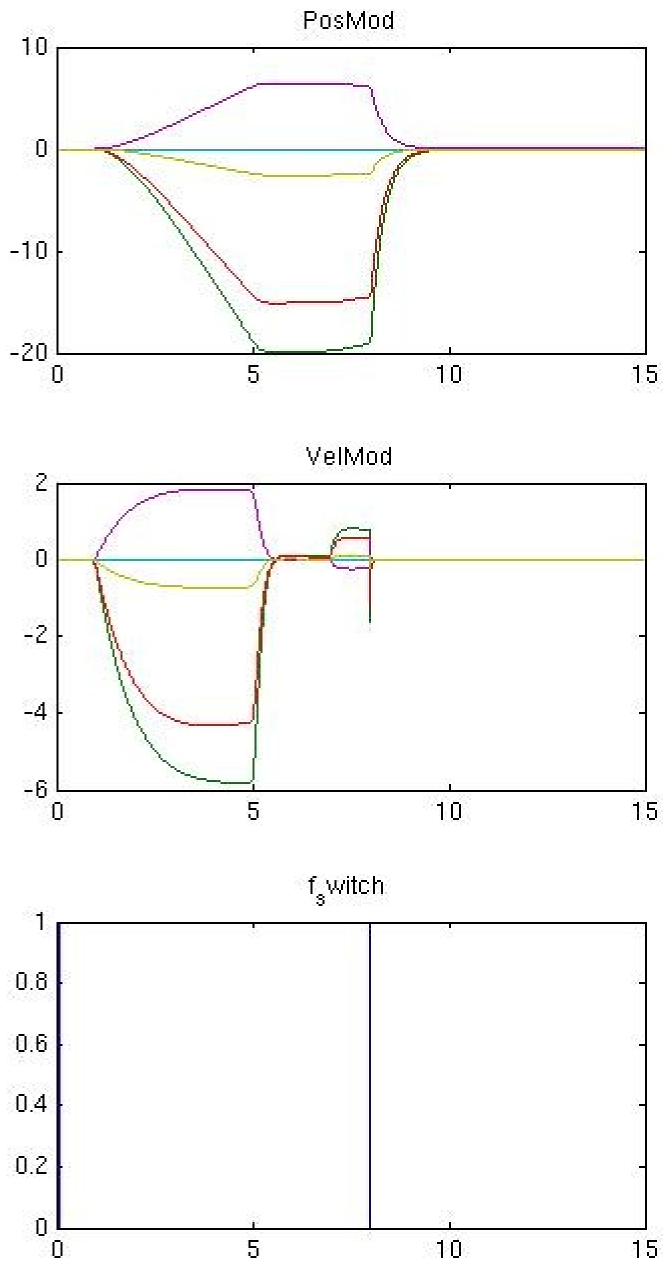


Figure 4b - Belongs to simulation in previous figure

B2 Results from simulations of turning f_switch off

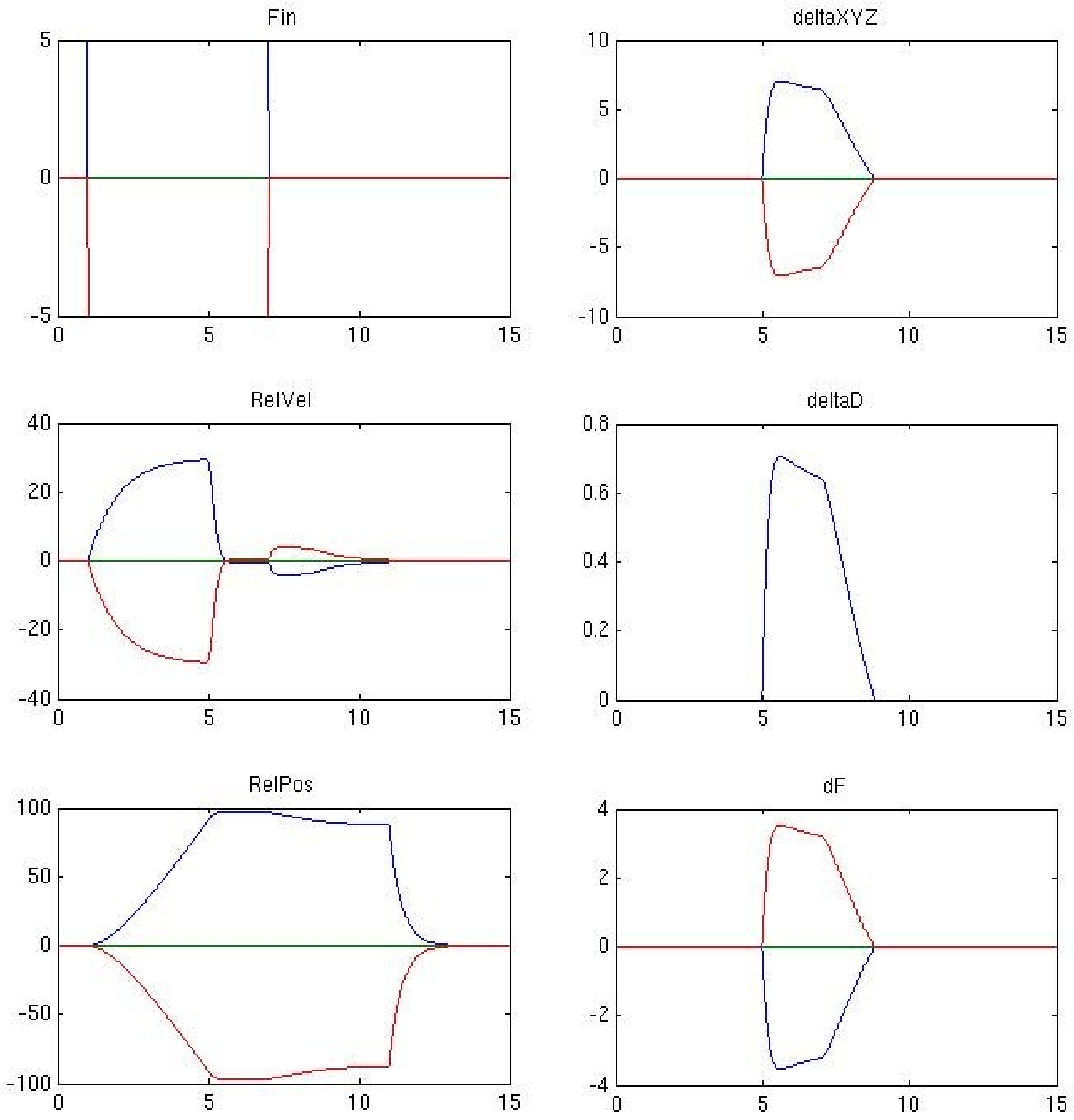


Figure 5a – Turning force input off after 7 sec and turning f_switch off after 12 sec when RelVel ~= 0.

B2 Results from simulations of turning f_switch off

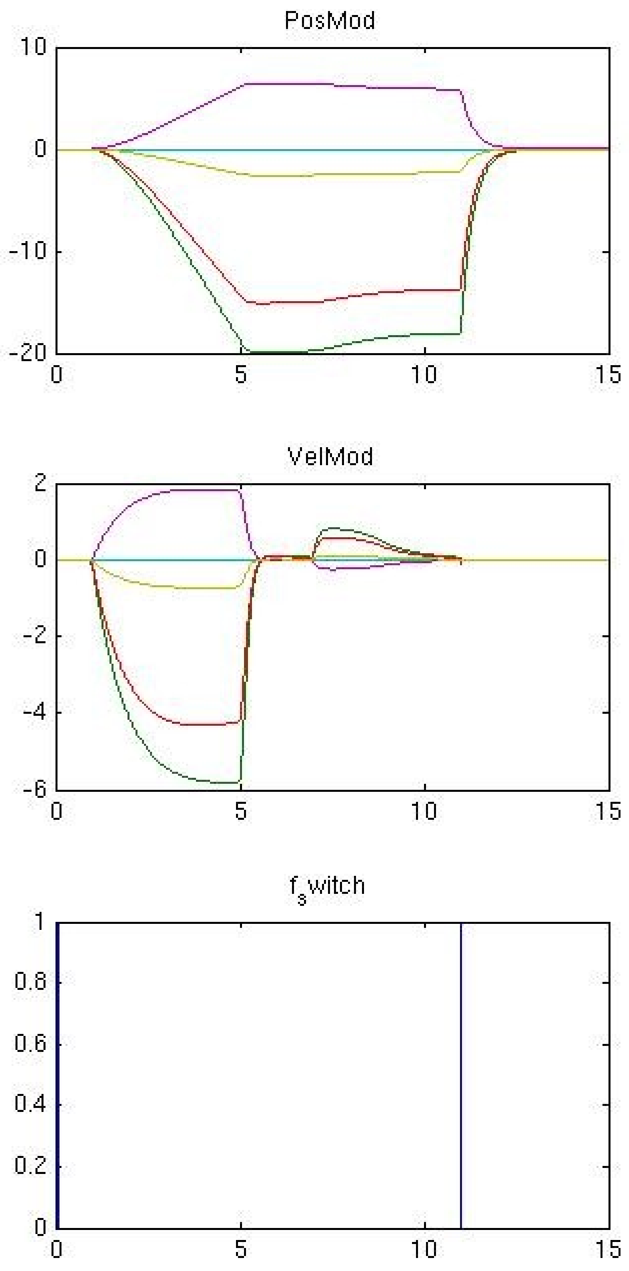


Figure 5b - Belongs to simulation in previous figure

C Final controller

C.1 Code for the discrete update part

This is the final code for the discrete update and the belonging help functions in the controller S-function Simulink block (written in C).

Discrete update

```
/* Let the discrete states be as follows:
 * xD[0] = Center of the safety zone cube x. When f_switch is turned off (=0)
 *        every update resets this position to the current one
 * xD[1] = Center of the safety zone cube y.
 * xD[2] = Center of the safety zone cube z.
 * xD[3] = The velocity dv x (mm/s)
 * xD[4] = The velocity dv y (mm/s)
 * xD[5] = The velocity dv z (mm/s)
 * xD[6] = beta x
 * xD[7] = beta y
 * xD[8] = beta z
 * xD[9] = F2oldx
 * xD[10] = F2oldy
 * xD[11] = F2oldz
 * xD[12] = Fmodx
 * xD[13] = Fmody
 * xD[14] = Fmodz
 * xD[15] = filtered F2 x
 * xD[16] = filtered F2 y
 * xD[17] = filtered F2 z
 * xD[18] = derivative of the filtered F2 x
 * xD[19] = derivative of the filtered F2 y
 * xD[20] = derivative of the filtered F2 z
 * xD[21] = topdv x
 * xD[22] = topdv y
 * xD[23] = topdv z
 * xD[24] = topF2 x
 * xD[25] = topF2 y
 * xD[26] = topF2 z
 * xD[27] = relative position of TCP x
 * xD[28] = relative position of TCP y
 * xD[29] = relative position of TCP z
 * xD[30] = boost x
 * xD[31] = boost y
 * xD[32] = boost z
 */

/*Initiating variables*/
double t44[16];
int i;
for(i = 0; i < 16; i++){
   t44[i] = t44tcp[i];
}

double Ts = sampleTime[0];  /*sampleTime[0] is a parameter to the block*/
double nbrsamples = 500;
double mdvc = Ts * max_dv_change[0];
double Tfilter = 50*Ts;
```

C1 Code for discrete update part

```
double Fmax = Fmax_in[0];
double dvmax = 100;

double pos[3] = {(t44[3] - xD[0]), (t44[7] - xD[1]), (t44[11] - xD[2])};
double dv[3] = {xD[3], xD[4], xD[5]};
double beta[3] = {xD[6], xD[7], xD[8]};
double F2old[3] = {xD[9], xD[10], xD[11]};
double Fmod[3] = {xD[12], xD[13], xD[14]};
double filtF2[3] = {xD[15], xD[16], xD[17]};
double dfiltF2[3] = {xD[18], xD[19], xD[20]};
double topdv[3] = {xD[21], xD[22], xD[23]};
double topF2[3] = {xD[24], xD[25], xD[26]};
double boost[3] = {xD[30], xD[31], xD[32]};

double F1[3] = {F1_in[0], F1_in[1], F1_in[2]};
double F2[3] = {F2_in[0], F2_in[1], F2_in[2]};

/*Do the calculations for each direction*/
for(i = 0; i < 3; i++){

   /*Reset filter if F2 inactive*/
   if(F2[i] == 0.0){
      filtF2[i] = 0.0;
   }

   /*Calculate beta and filter F2*/
   beta[i] = calbeta(F1[i], F2[i], nbrsamples, beta[i]);
   filtF2[i] = filterF2(F2[i], F2old[i], filtF2[i], &(dfiltF2[i]),
                 Fmax, Ts, Tfilter, &(topF2[i]), &(topdv[i]), dv[i]);

   /*If F2 > Fmax some exceptions must be handled*/
   if(absd(F2[i]) >= Fmax){
      if(signd2(F2[i]) != signd2(F1[i])){
         beta[i] = 1.0;
      }
      if(signd2(dv[i]) != signd2(F2[i])){
         dv[i]= 0;
      }
   }

   /*Boost beta*/
   betabooster(&boost[i], beta[i], nbrsamples);
   if(boost[i] > 0 && (absd(F2[i]) > 0.0 && absd(F1[i]) > 0.0 &&
                                                            signd2(F1[i]) != signd2(F2[i]))){
      beta[i] = 1;
   }

   /*calculate Fmod*/
   Fmod[i] = calcftot(F1[i],F2[i],Fmax, beta[i]);

   /*calculate dv, k-part of impedance makes no sense to use, therefor pos = 0*/
   dv[i] = calcdv(Fmod[i], F1[i], F2[i], F2old[i], Fmax, MDK[0], MDK[2], MDK[4],
            dv[i], 0, Ts, mdvc, beta[i], &topdv[i], &topF2[i], &filtF2[i], &dfiltF2[i]);

   if(dv[i] > dvmax){
      dv[i] = dvmax;
   }else if (dv[i] < -dvmax){
      dv[i] = -dvmax;
```

```
    }
}

/*If f_switch = 0, reset the states*/
if(f_switch[0] < 0.01){
    xD[0] = t44[3];
    xD[1] = t44[7];
    xD[2] = t44[11];

    for(i = 0; i < 3; i++){
        dv[i] = 0;
        dvout[i] = 0;
        beta[i] = 0;
        Fmod[i] = 0;
        topdv[i] = 0;
        topF2[i] = 0;
        F2[i] = 0;
        filtF2[i] = 0;
        dfiltF2[i] = 0;
        boost[i] = 0;
        pos[i] = 0;
    }
}

xD[3] = dv[0];
xD[4] = dv[1];
xD[5] = dv[2];
xD[6] = beta[0];
xD[7] = beta[1];
xD[8] = beta[2];
xD[9] = F2[0];
xD[10] = F2[1];
xD[11] = F2[2];
xD[12] = Fmod[0];
xD[13] = Fmod[1];
xD[14] = Fmod[2];
xD[15] = filtF2[0];
xD[16] = filtF2[1];
xD[17] = filtF2[2];
xD[18] = dfiltF2[0];
xD[19] = dfiltF2[1];
xD[20] = dfiltF2[2];
xD[21] = topdv[0];
xD[22] = topdv[1];
xD[23] = topdv[2];
xD[24] = topF2[0];
xD[25] = topF2[1];
xD[26] = topF2[2];
xD[27] = pos[0];
xD[28] = pos[1];
xD[29] = pos[2];
xD[30] = boost[0];
xD[31] = boost[1];
xD[32] = boost[2];
```

C1 Code for discrete update part

Help functions (fct_fmod2.h)

```
/*This function gives the absolute value of a double*/
double absd(double d);

/*This function returns the sign of a double*/
double signd2(double d);

/*This function calculates the weighted value of F1 & F2*/
double calcftot(double F1, double F2,double Fmax, double beta);

/*This function calculates dv*/
double calcdv(double Ftot, double F1, double F2, double F2old, double Fmax, double m,
double d, double k, double dv, double pos, double Ts, double mdvc, double beta, double*
  topdv, double* topF2, double* filtF2, double* dfiltF2);

/*This is a helpfunction for limiting dv*/
double limitdv(double dv, double dvold, double F2, double mdvc, double Fmax);

/*This function calculates beta*/
double calbeta(double F1, double F2, double nsmp, double beta);

/*This function calculates a filtered value of F2 and its derivative*/
double filterF2(double F2, double F2old, double F2filter, double* dfiltF2, double Fmax,
  double Ts, double Tf, double* topF2, double* topdv, double dv);

/*Boost beta*/
void betabooster(double* boost, double beta, double nbrsamples);
```

C1 Code for discrete update part

Help functions (fct_fmod2.c)

```
#include "fct_fmod2.h"

/*This function gives the absolute value of a double*/
double absd(double d){
  if(d < 0){
    return -d;
  }else{
    return d;
  }
}

/*This function returns the sign of a double*/
double signd2(double d){
    if(d < 0){
        return -1.0;
    }else if(d == 0.0){
        return 0;
    }else{
        return 1.0;
    }
}

/*This function calculates the weighted value of F1 & F2*/
double calcftot(double F1, double F2, double Fmax, double beta){
    if(absd(F2) < Fmax){
        /*Purpose here is to if possible not reduce importance of F2*/
        if((signd2(F2) == signd2(F1) || absd(F1) < 0.5) && absd(F2) > absd(F1)){
            return F2;
        }else{
            return ((absd(F2)/Fmax+beta*(1-absd(F2)/Fmax))*F2 +
                                                        (1-beta)*(1-absd(F2)/Fmax)*F1);
        }
    }else{
        return F2;
    }
}

/*This function calculates dv*/
double calcdv(double Ftot, double F1, double F2, double F2old, double Fmax, double m,
double d, double k, double dv, double pos, double Ts, double mdvc, double beta, double*
   topdv, double* topF2, double* filtF2, double* dfiltF2){
    if(m==0.0){
        return 0;
        if(d==0){
            return 0.0;
        }else{
            return (Ftot/d-k/d*pos);
        }
    }else{
        double res = 0;
        double F2lim = 4;
        double F2lim2 = 8;
        double stoprate = 0.25;
        double stoprate2 = 0.5;
        double difflim = 2;
        double breakrate = 0.75;
        double breaklim = 200;
```

C1 Code for discrete update part

```
double nslim = 50;
if(absd(F2) > 50){
   difflim = absd(F2)/20;
}

if(F2<0){
   if(((absd(F2old)-absd(F2)) > difflim) && beta > 0.9){
      if(absd(F2) < F2lim2){
         res = stoprate2*dv;
      }else{
         *topF2 = F2old;
         *topdv = dv;
         *filtF2 = F2;
         double ns = absd(F2)/(absd(F2old)-absd(F2));
         if(ns > nslim){
            breakrate = 1;
         }else{
            breakrate = ns/nslim;
         }
         if(absd(F2)<breaklim){
            res = breakrate*dv;
         }else{
            res = dv;
         }
      }
   }else{
      if(((*dfiltF2) > 0) && beta > 0.9){
         if(absd(F2) < F2lim){
            res = stoprate*dv;
         }else{
            if((*dfiltF2)>0.05){
               res = (*topdv)*absd(F2)/absd(*topF2);
            }else{
               double fct = 1 - (*dfiltF2)/0.05;
               res = (dv+fct*Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
               *topdv = dv;
               *topF2 = F2old;
            }
         }
      }else if((*dfiltF2) < -0.1){
         res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
      }else{
         if(absd(F2) < F2lim && beta > 0.9){
            res = stoprate*dv;
         }else{
            res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
         }
      }
   }
}else if(F2>0){
   if(((absd(F2old)-absd(F2)) > difflim) && beta > 0.9){
      if(absd(F2) < F2lim2){
         res = stoprate2*dv;
      }else{
         *topF2 = F2old;
         *topdv = dv;
         *filtF2 = F2;
         double ns = absd(F2)/(absd(F2old)-absd(F2));
```

C1 Code for discrete update part

```
                    if(ns > nslim){
                        breakrate = 1;
                    }else{
                        breakrate = ns/nslim;
                    }
                    if(absd(F2)<breaklim){
                        res = breakrate*dv;
                    }else{
                        res = dv;
                    }
                }
            }else{
                if((*dfiltF2) < 0 && beta > 0.9){
                    if(absd(F2) < F2lim){
                        res = stoprate*dv;
                    }else{
                        if((*dfiltF2)<-0.05){
                            res = (*topdv)*absd(F2)/absd(*topF2);
                        }else{
                            double fct = 1 - absd(*dfiltF2)/0.05;
                            res = (dv+fct*Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
                            *topdv = dv;
                            *topF2 = F2old;
                        }
                    }
                }else if((*dfiltF2) > 0.1){
                    res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
                }else{
                    if(absd(F2) < F2lim && beta > 0.9){
                        res = stoprate*dv;
                    }else{
                        res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
                    }
                }
            }
        }else{
            if(beta > 0.9){
                res = stoprate*dv;
            }else{
                res = (dv+Ts*((Ftot/m)-(k/m*pos)-(d/m*dv)));
            }
        }
        res = limitdv(res, dv, F2, mdvc, Fmax);
        return res;
    }
}

/*This is a helpfunction for limiting dv. It has harder limits when F2 > Fmax.*/
double limitdv(double dv, double dvold, double F2, double max_dv_change, double
  Fmax){
    if(0.1 < absd(F2)){
        double mdvc = max_dv_change;
        if(dv < 0.0){
            if((dv < dvold) && (dv < (dvold - mdvc))){
                return dvold - mdvc;
            }else{
                return dv;
            }
        }else if(dv == 0.0){
            return dv;
        }else{
            if((dvold < dv) && ((dvold + mdvc) < dv)){
                return dvold + mdvc;
            }else{
                return dv;
            }
        }
    }else{
        return dv;
    }
}



/*This function calculates beta*/
double calbeta(double F1, double F2, double nsmp, double beta){
    double res = beta;
    if(absd(F2) > 0.0 && absd(F1) > 0.0 && signd2(F1) != signd2(F2)){
        res = beta + absd(F2)/nsmp;
        if(res > 1.0){
            res = 1.0;
        }
    }else{
        if(0.0 < beta){
            res = beta - 1.0/nsmp;
        }
        if(res < 0.0){
            res= 0.0;
        }
    }
    return res;
}

/*This function calculates a filtered value of F2 and its derivative*/
double filterF2(double F2, double F2old, double F2filter, double* dfiltF2, double Fmax,
   double Ts, double Tf, double* topF2, double* topdv, double dv){
    double newfilt = F2filter*(1-Ts/Tf)+F2*Ts/Tf;
    double dfiltF2old = *dfiltF2;
    *dfiltF2 = (newfilt-F2filter);
    if(signd2(*dfiltF2) != signd2(dfiltF2old)){
        *topdv = dv;
        *topF2 = F2old;
    }
    return newfilt;
}

/*Boost beta*/
void betabooster(double* boost, double beta, double nbrsamples){
    double res = 0;
    if(beta == 1){
        res = nbrsamples;
    }else{
        res = *boost-1;
        if(res < 0){
            res = 0;
        }
    }
    *boost = res;
}
```

C.2 Code for the safety zone

The code for the safety zone block (written as a function in Matlab code)

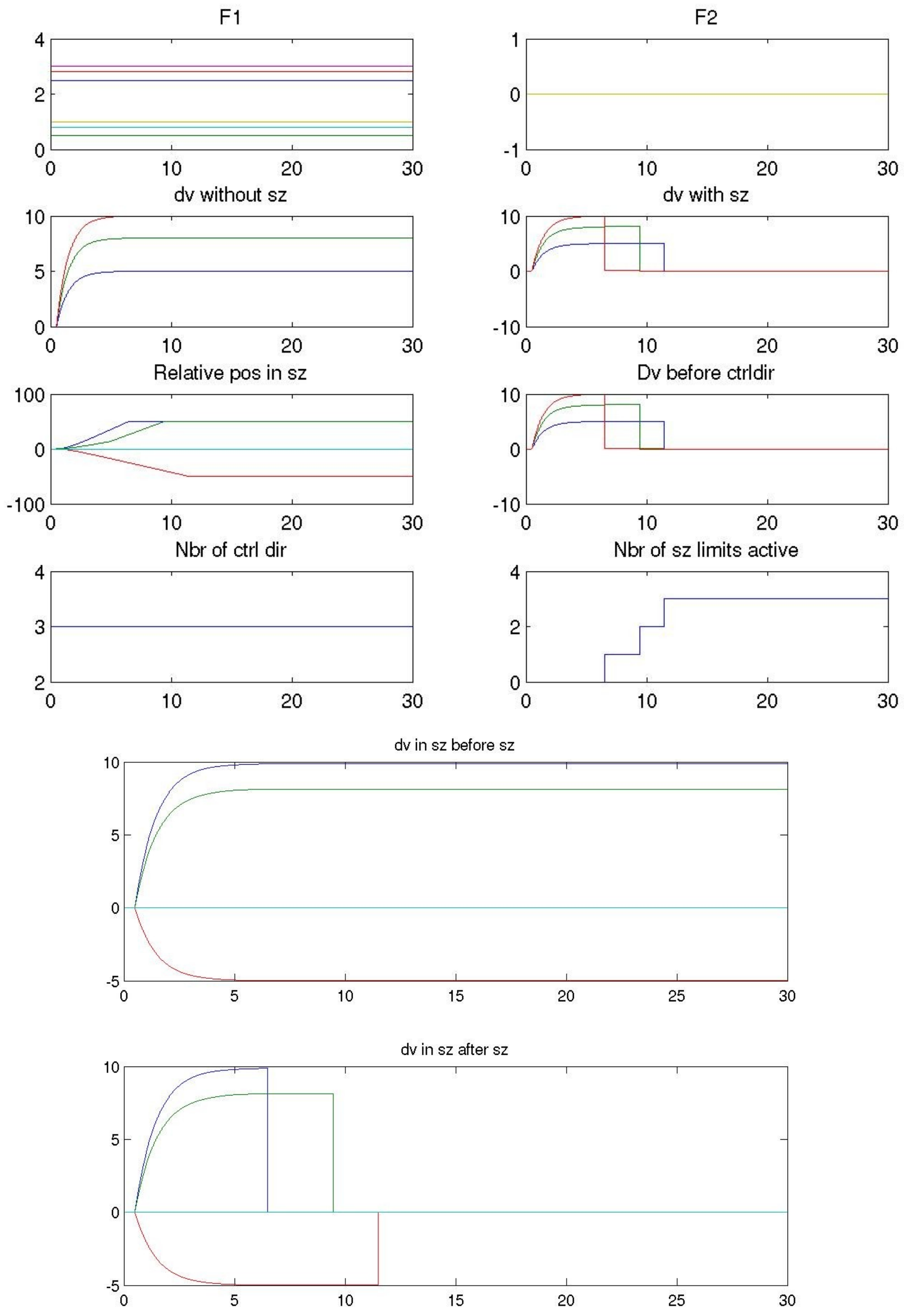
```
function [out, ctrlprm, dv_after_sz] =fcn(dvTCPin, posbase, limit,
      ctrldir, T44basesz, T44TCPsz, T44szTCP, f_switch, hyst)

     persistent rtlim;
     if(isempty(rtlim) || f_switch < 0.5)
         rtlim=[limit, limit, limit, -limit, -limit, -limit];
     end

     %Decide how many control directions there are
     sv = 0.0001;
     ncdir = 0;
     for i = 1:3
         if(ctrldir(i) < (1+sv) && ctrldir(i) > (1-sv))
             ncdir = ncdir+1;
         end
     end

     %rotate dv in tcp to dv in sz
     dvsz = [0 0 0];
     dvsz(1)  = T44TCPsz(1)*dvTCPin(1) + T44TCPsz(2)*dvTCPin(2) +
                                           T44TCPsz(3)*dvTCPin(3);
     dvsz(2)  = T44TCPsz(5)*dvTCPin(1) + T44TCPsz(6)*dvTCPin(2) +
                                           T44TCPsz(7)*dvTCPin(3);
     dvsz(3)  = T44TCPsz(9)*dvTCPin(1) + T44TCPsz(10)*dvTCPin(2) +
                                           T44TCPsz(11)*dvTCPin(3);

     %rotate relpos in base to relpos in sz
     possz = [0 0 0];
     possz(1)  = T44basesz(1)*posbase(1)  + T44basesz(2)*posbase(2)
                                     + T44basesz(3)*posbase(3);
     possz(2)  = T44basesz(5)*posbase(1)  + T44basesz(6)*posbase(2)
                                     + T44basesz(7)*posbase(3);
     possz(3)  = T44basesz(9)*posbase(1)  + T44basesz(10)*posbase(2)
                                     + T44basesz(11)*posbase(3);

     iout = [0 0 0]; lim = [0 0 0]; nszlim = 0;
     for i = 1:3
         %Create the hysterisis effect
         if(possz(i) >= rtlim(i))
             rtlim(i)  = limit - hyst;
         else
             rtlim(i)  = limit;
         end
         if(possz(i) <= rtlim(i+3))
             rtlim(i+3)  = -limit + hyst;
         else
             rtlim(i+3)  = -limit;
         end
         %Create the sz
         if((possz(i) >= rtlim(i) && dvsz(i) > 0.0) ||
                         (possz(i) <= rtlim(i+3) && dvsz(i) < 0.0))
             dvsz(i) = 0.0;
             lim(i) = 1;
             nszlim = nszlim + 1;
```

```
        end
    end
```

C2 Code for the safety zone

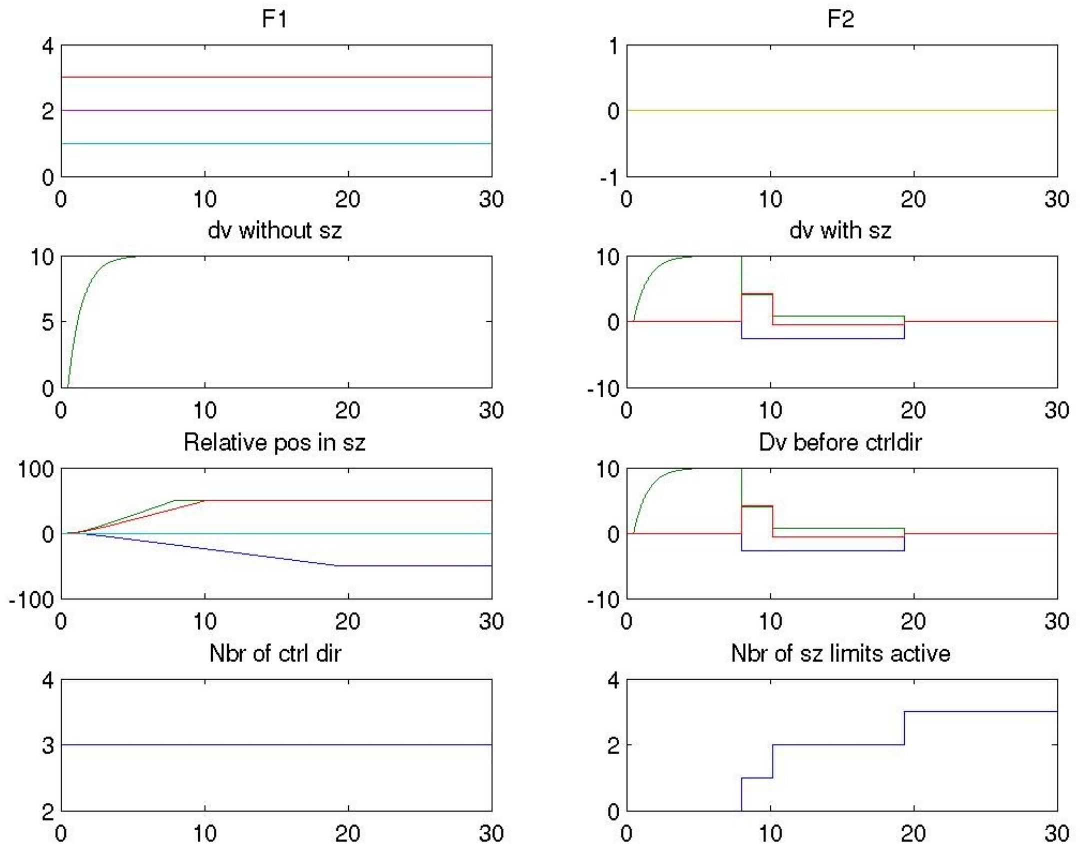
```
%rotate from safety zone to TCP
iout(1)  = T44szTCP(1)*dvsz(1)  + T44szTCP(2)*dvsz(2)  +
                                      T44szTCP(3)*dvsz(3);
iout(2)  = T44szTCP(5)*dvsz(1)  + T44szTCP(6)*dvsz(2)  +
                                      T44szTCP(7)*dvsz(3);
iout(3)  = T44szTCP(9)*dvsz(1)  + T44szTCP(10)*dvsz(2)  +
                                      T44szTCP(11)*dvsz(3);

dv_after_sz = iout';

%Make some more restrictions depending on what the controlled
 directions are
if(ncdir <  (1+sv)  && ncdir >  (1-sv))
    if(nszlim <sv  && nszlim > -sv)
        iout = dvTCPin';
    else
        iout = [0  0  0];
    end
elseif(ncdir <  (3+sv)  && ncdir >  (3-sv))
    if(nszlim <sv  && nszlim > -sv)
        iout = dvTCPin';
    end
elseif(ncdir <  (2+sv)  && ncdir >  (2-sv))
    if(nszlim <sv  && nszlim > -sv)
        iout = dvTCPin';
    elseif(nszlim <(1+sv)  && nszlim >  (1-sv))
        cpn = [0  0  0];
        for i = 1:3
            if(ctrldir(i)  < sv  && ctrldir(i)  > -sv)
                cpn(i)  = 1;
            end
        end
        %rotate from safety zone to tcp
        szpn = [0  0  0];
        szpn(1)  = T44szTCP(1)*lim(1)  + T44szTCP(2)*lim(2)  +
                                      T44szTCP(3)*lim(3);
        szpn(2)  = T44szTCP(5)*lim(1)  + T44szTCP(6)*lim(2)  +
                                      T44szTCP(7)*lim(3);
        szpn(3)  = T44szTCP(9)*lim(1)  + T44szTCP(10)*lim(2)  +
                                      T44szTCP(11)*lim(3);

        %Project on line
        %n1 x n2 ->
        n1 = cpn; n2 = szpn;
        projdir = [0  0  0]; t= 0; n= 0;
        projdir = [(n1(2)*n2(3)-n2(2)*n1(3)),
                    (n1(3)*n2(1)-n2(3)*n1(1)),
                    (n1(1)*n2(2)-n2(1)*n1(2))];

        %       dv*projdir
        % Y = -----------2- * projdir
        %        |projdir|

        t = iout(1)*projdir(1)+iout(2)*projdir(2)       +
                                      iout(3)*projdir(3);
```

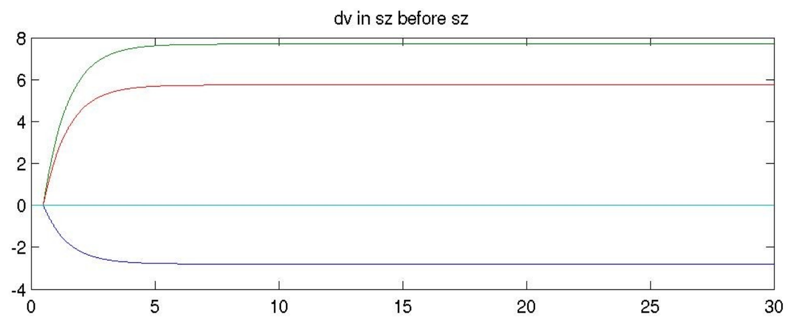
C.3 Simulations of the safety zone

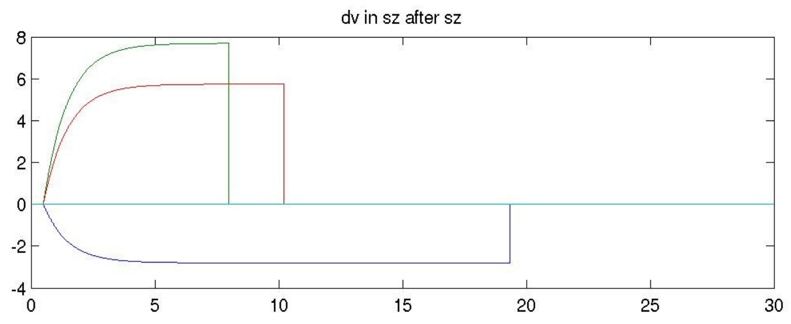


Simulation one – All three control directions used

C3 Simulations of the safetyzone

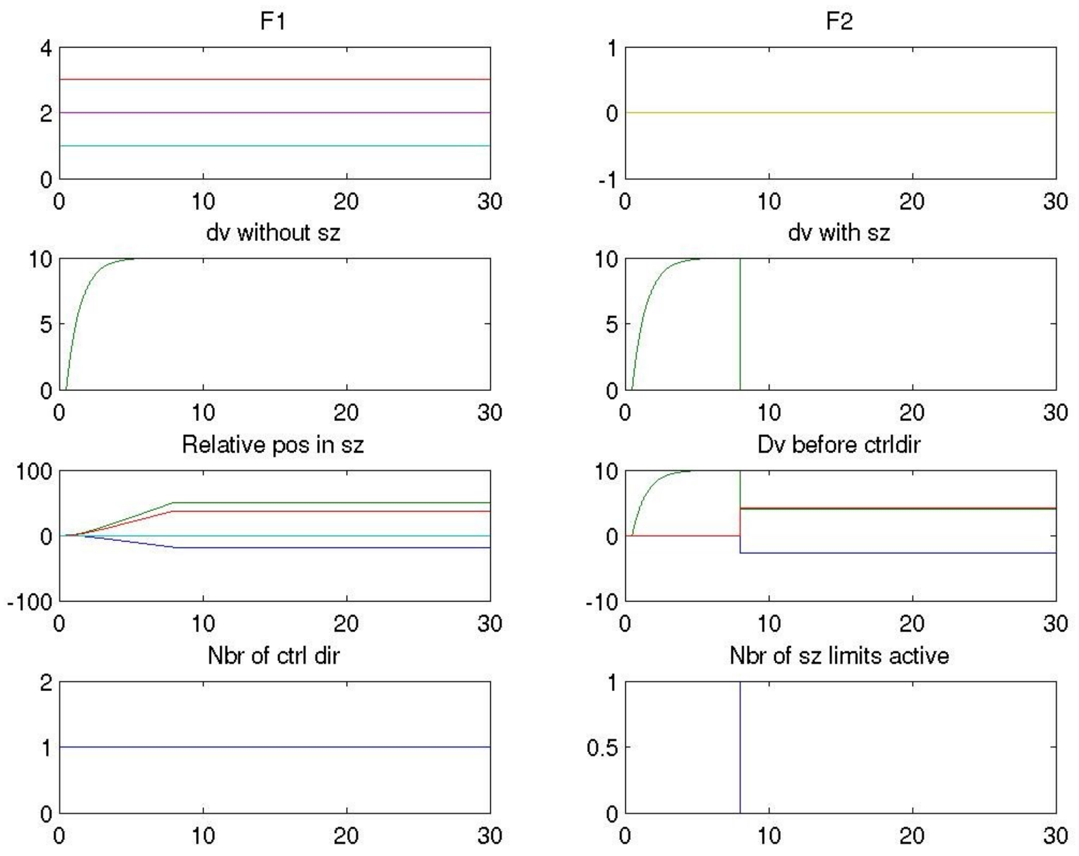


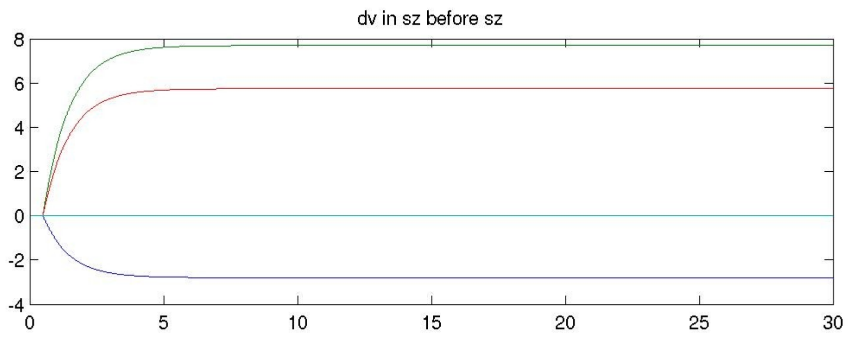


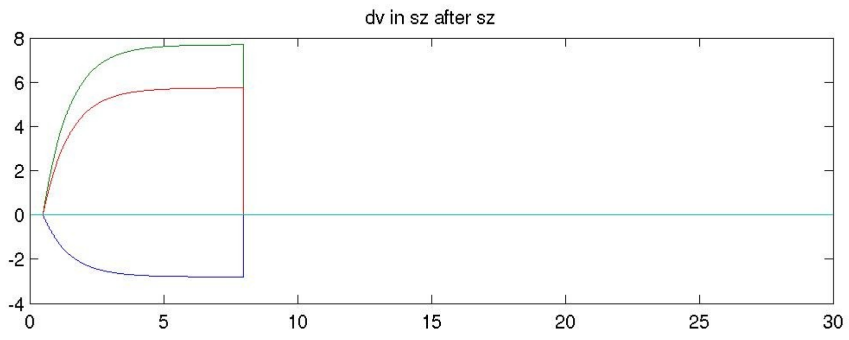


Simulation two – All three control directions in use but only output on one.

C3 Simulations of the safetyzone

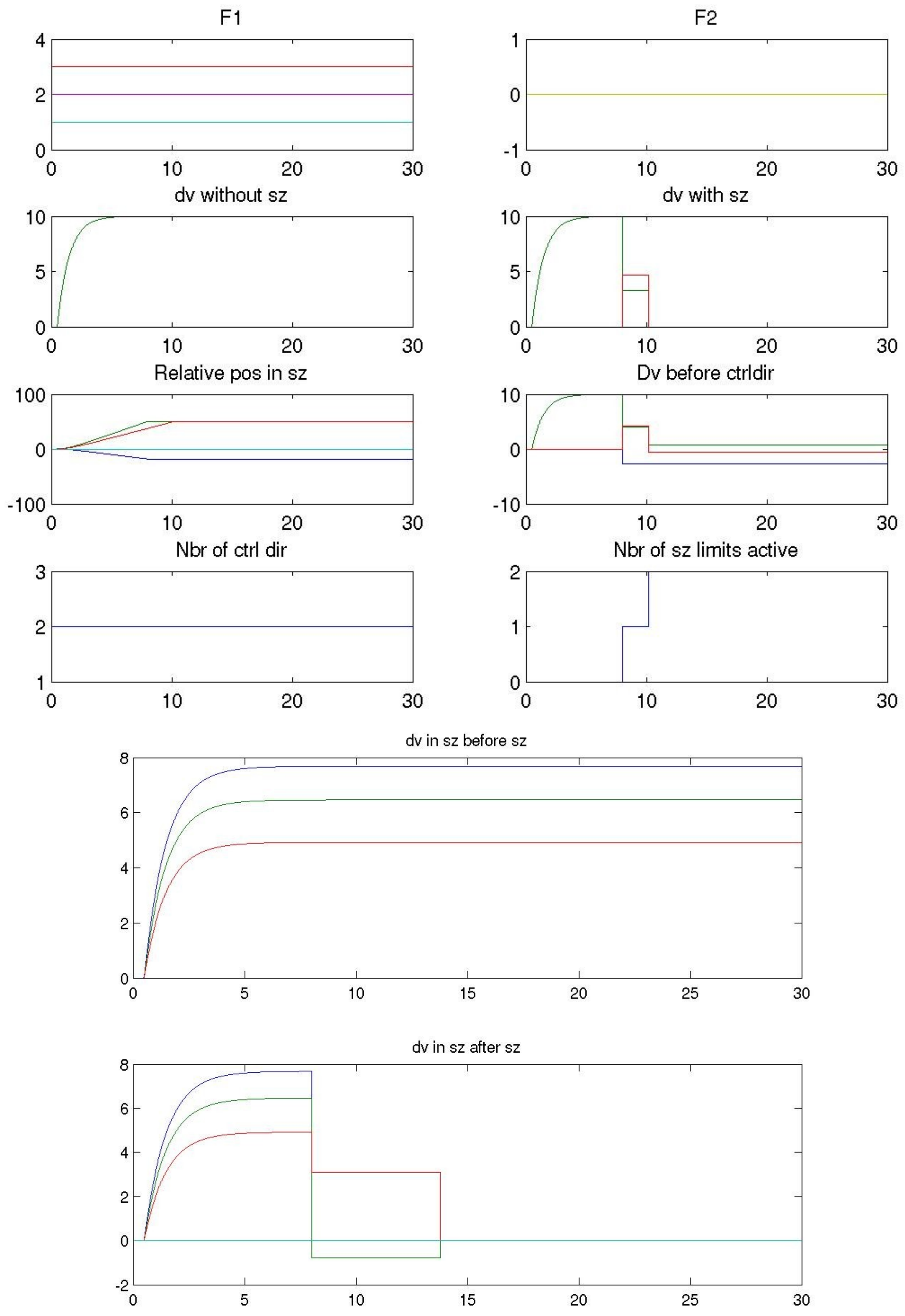






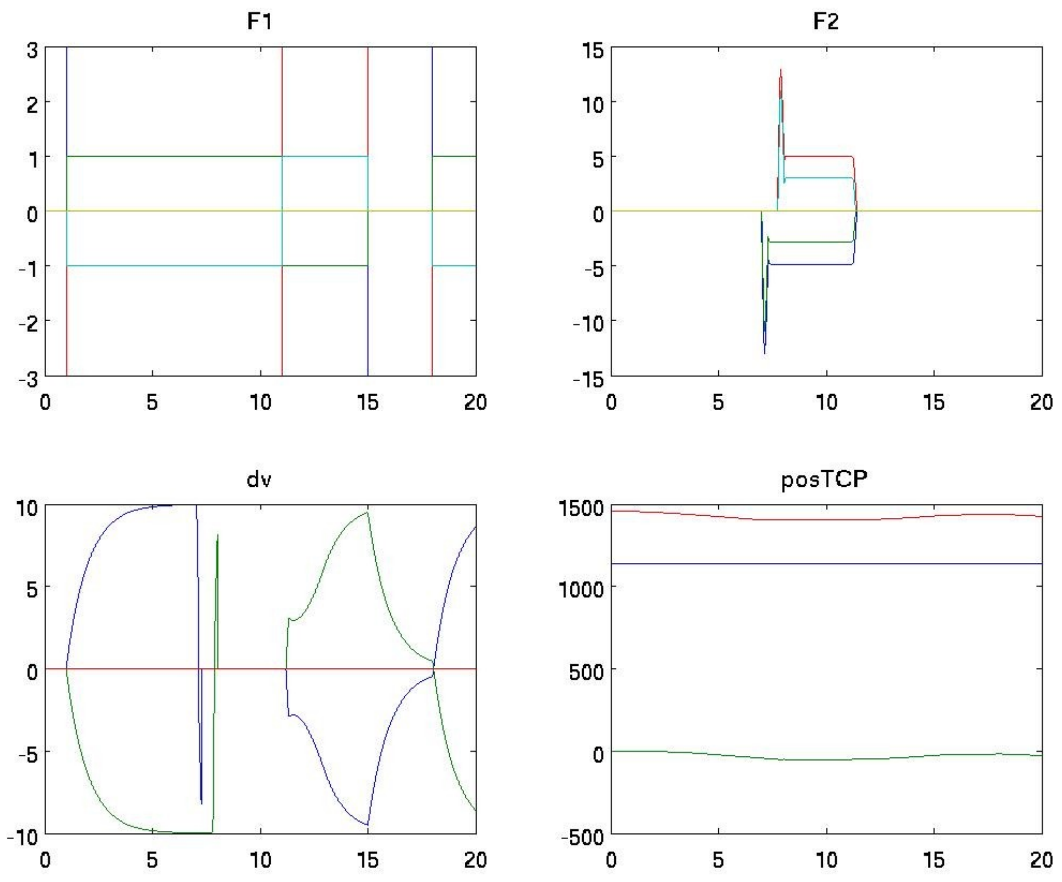
Simulation three – Only one control direction in use.

C3 Simulations of the safetyzone



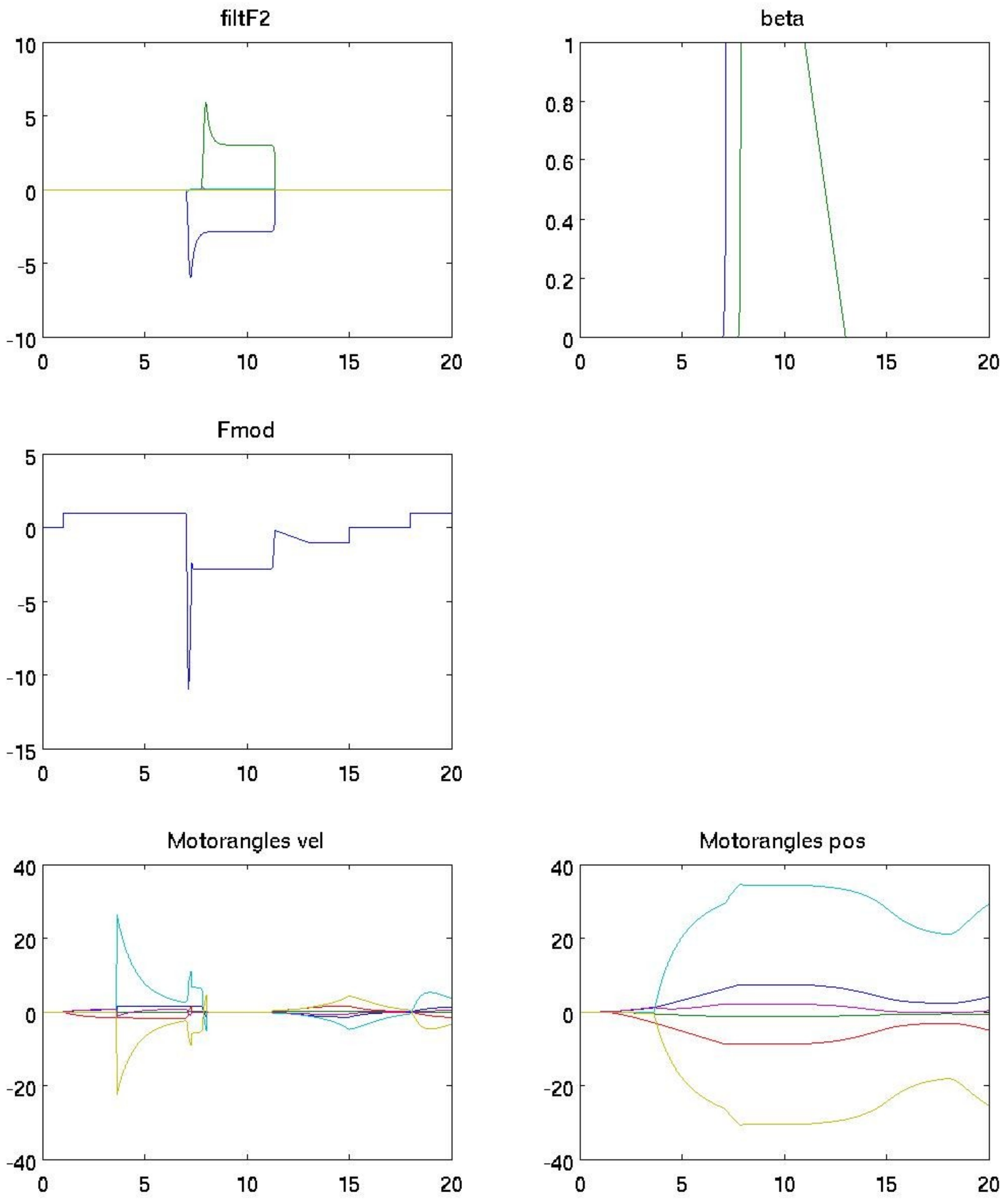
Simulation four – Controlling in a plane.

C.4 Simulations of the Fmod rc7 controller



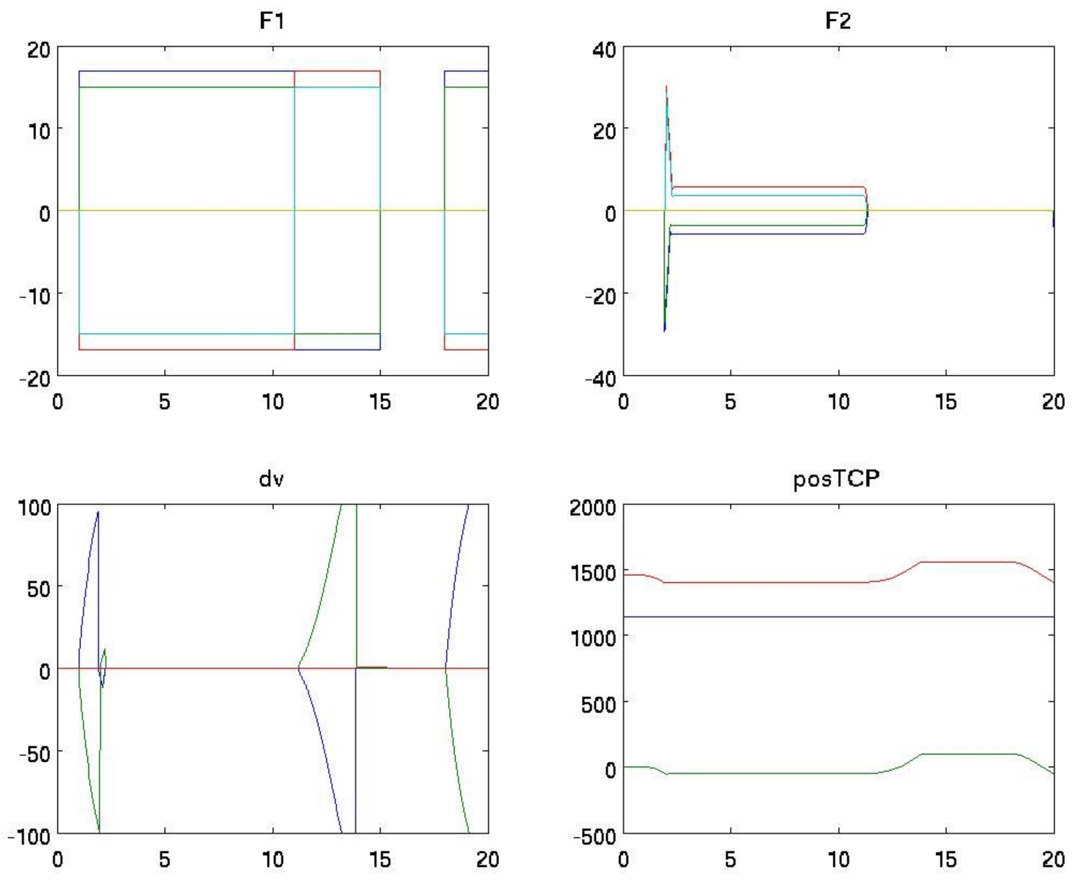
Simulation 1 – Main signals

C4 Simulation of the Fmod rc7 controller



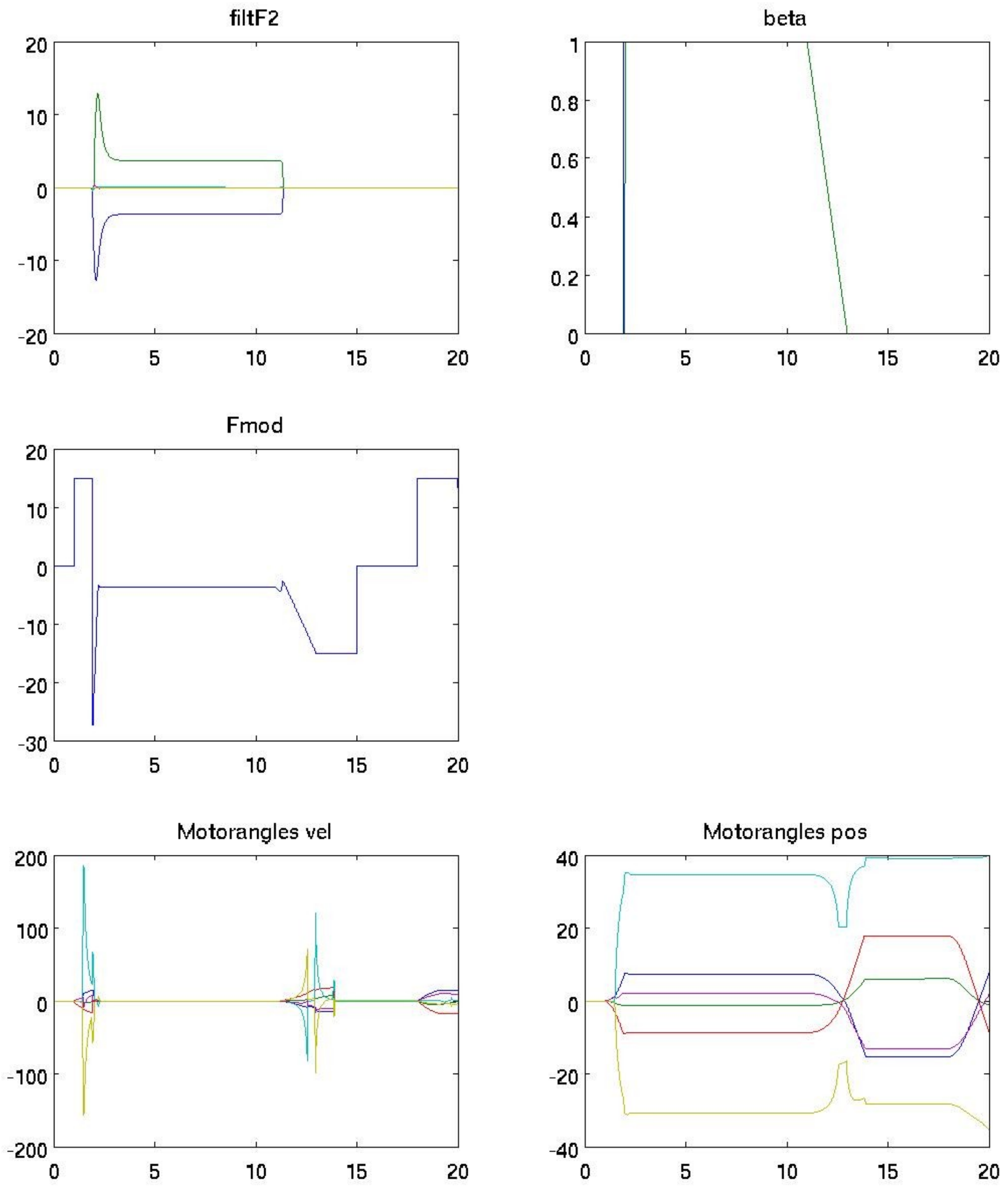
Simulation 1 – Control signals

C4 Simulation of the Fmod rc7 controller



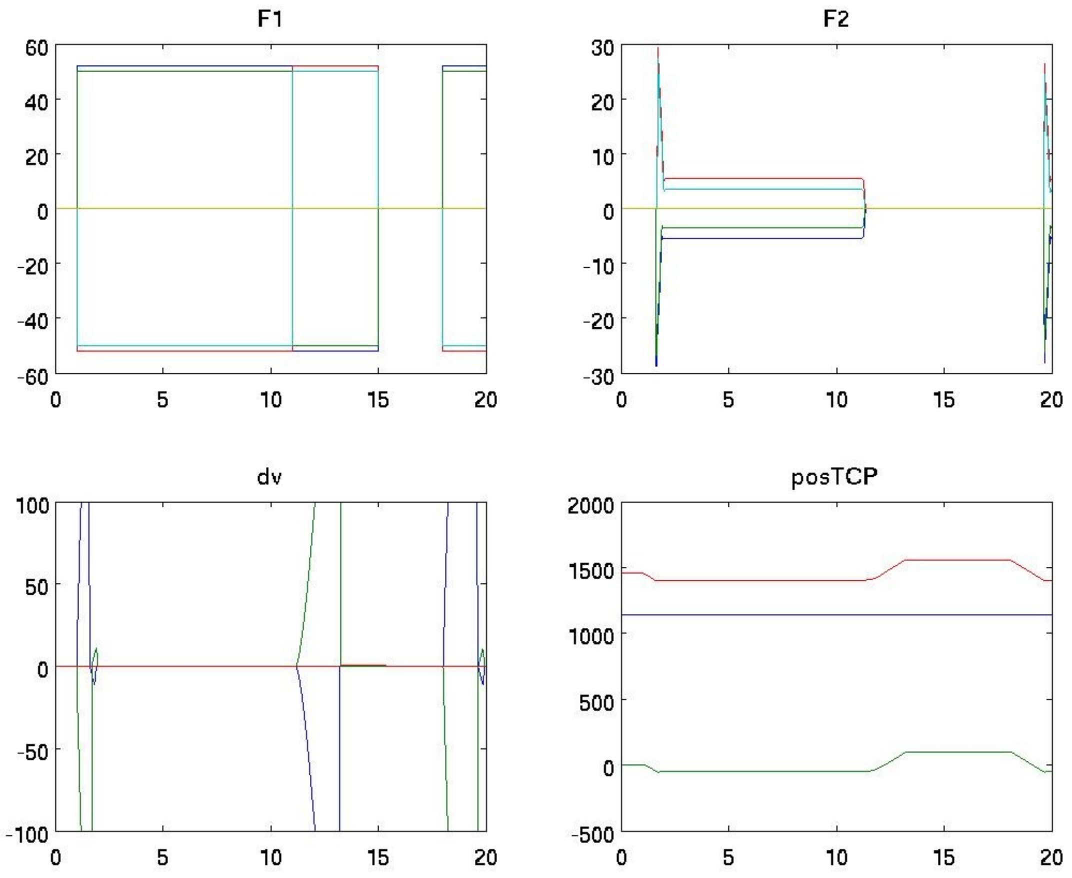
Simulation 2 – Main signals

C4 Simulation of the Fmod rc7 controller



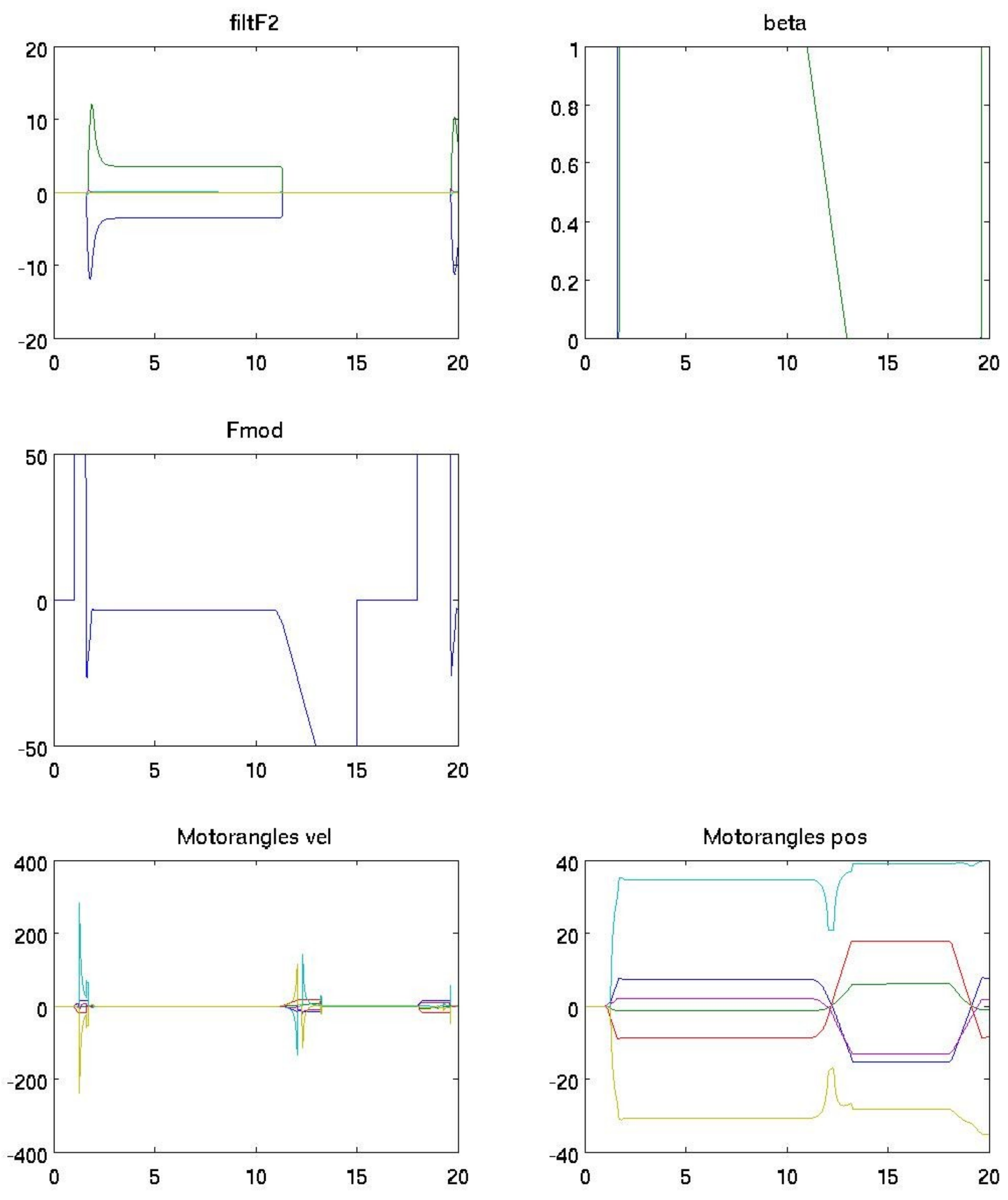
Simulation 2 – Control signals

C4 Simulation of the Fmod rc7 controller



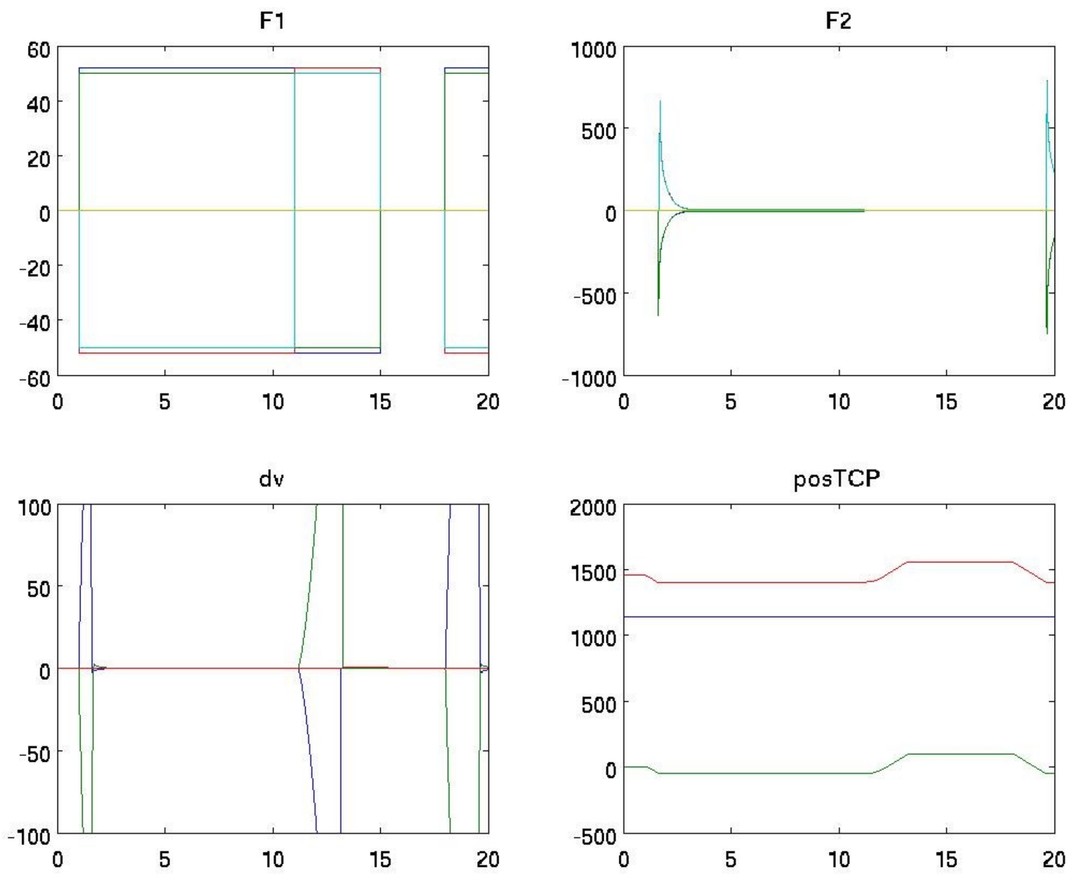
Simulation 3 – Main signals

C4 Simulation of the Fmod rc7 controller



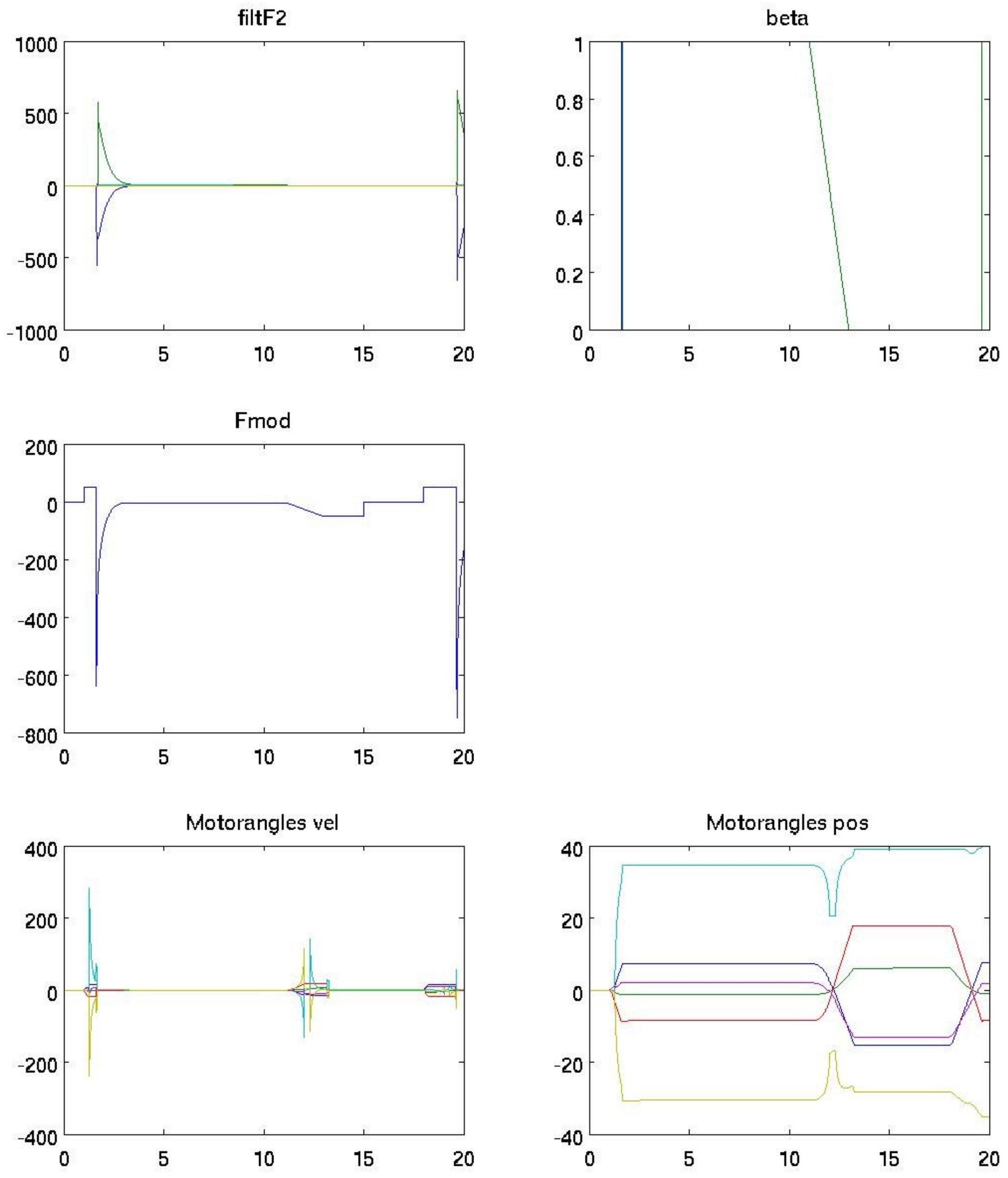
Simulation 3 – Control signals

C4 Simulation of the Fmod rc7 controller

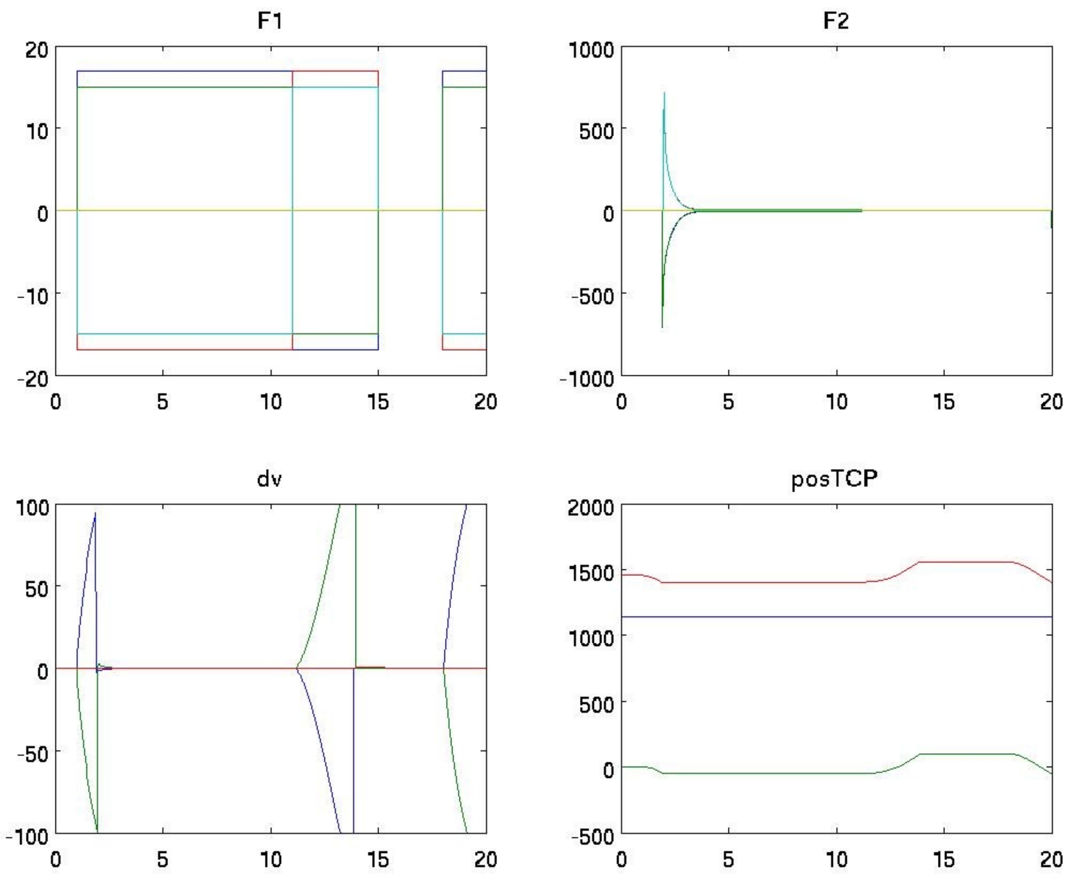


Simulation 4 – Main signals

C4 Simulation of the Fmod rc7 controller

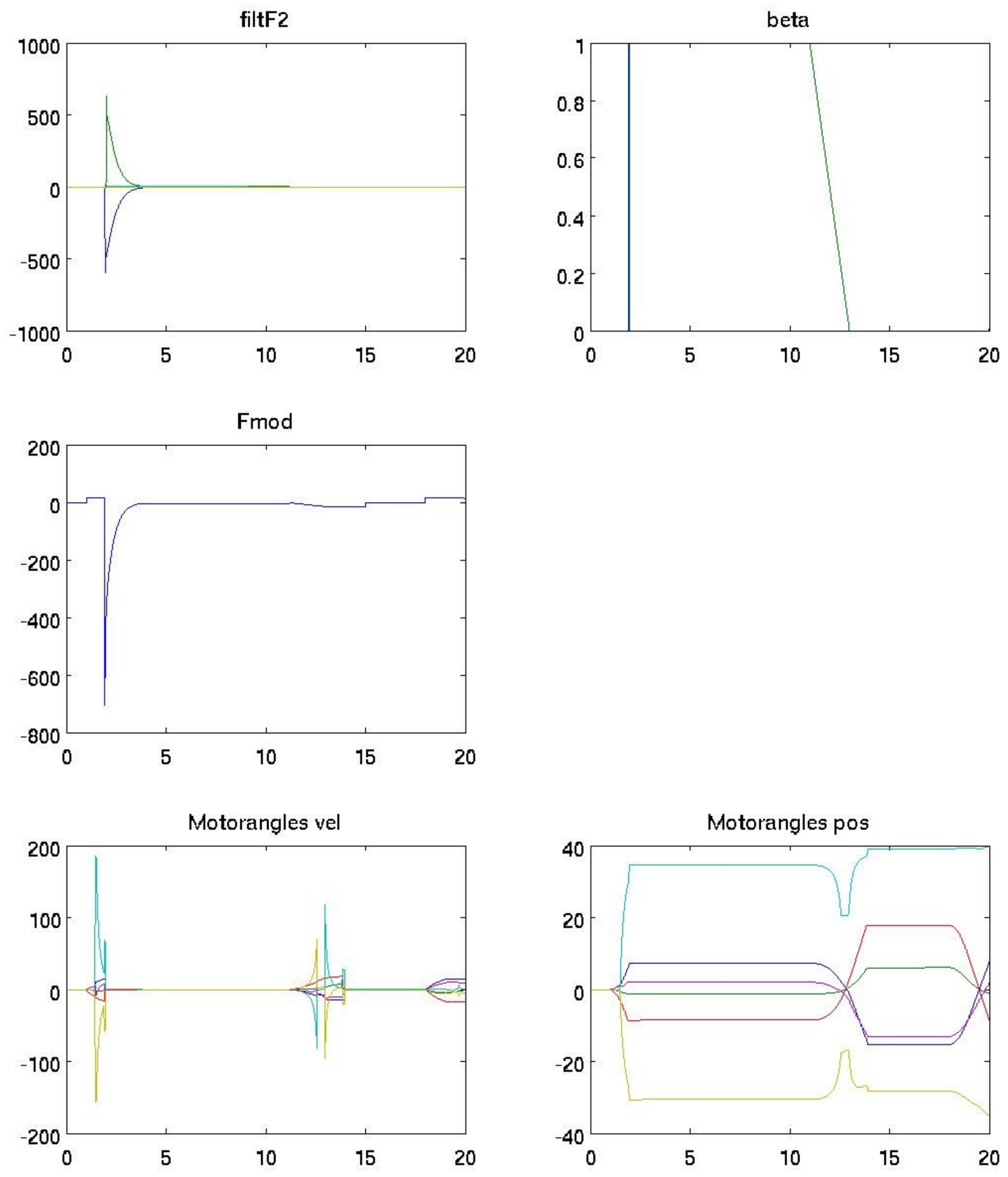


Simulation 4 – Control signals



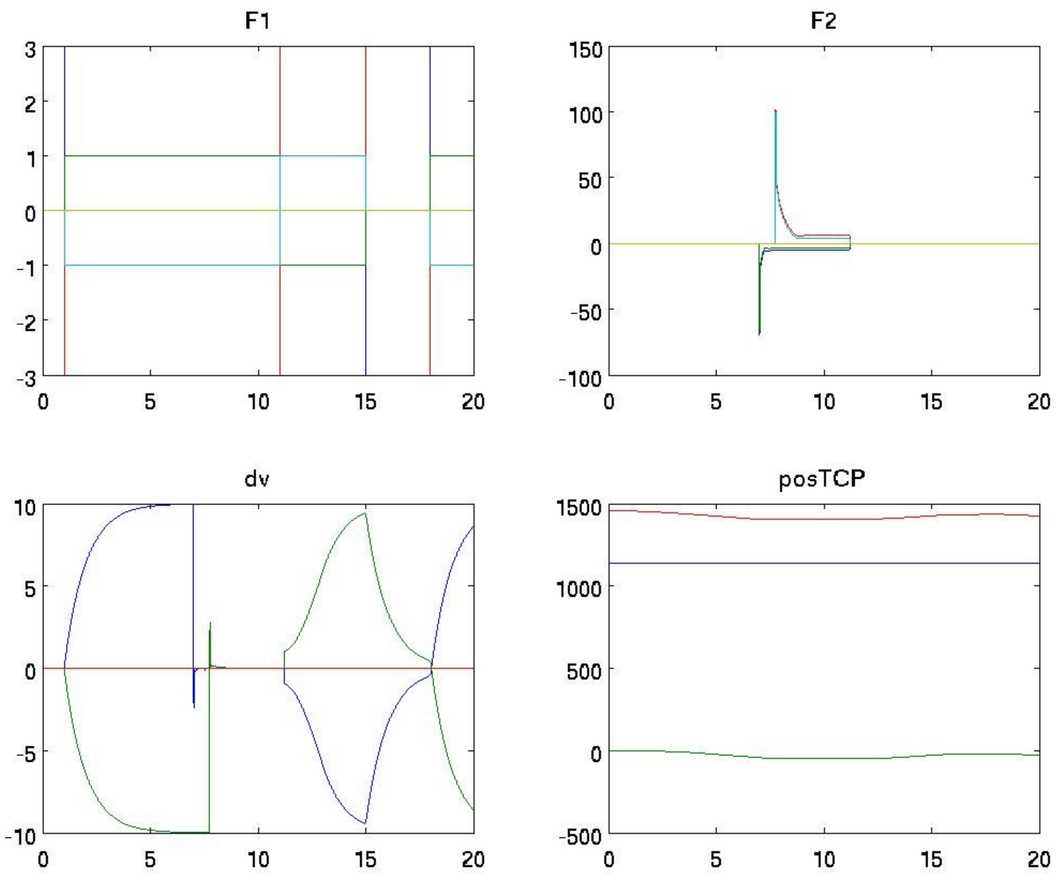
Simulation 5 – Main signals

C4 Simulation of the Fmod rc7 controller



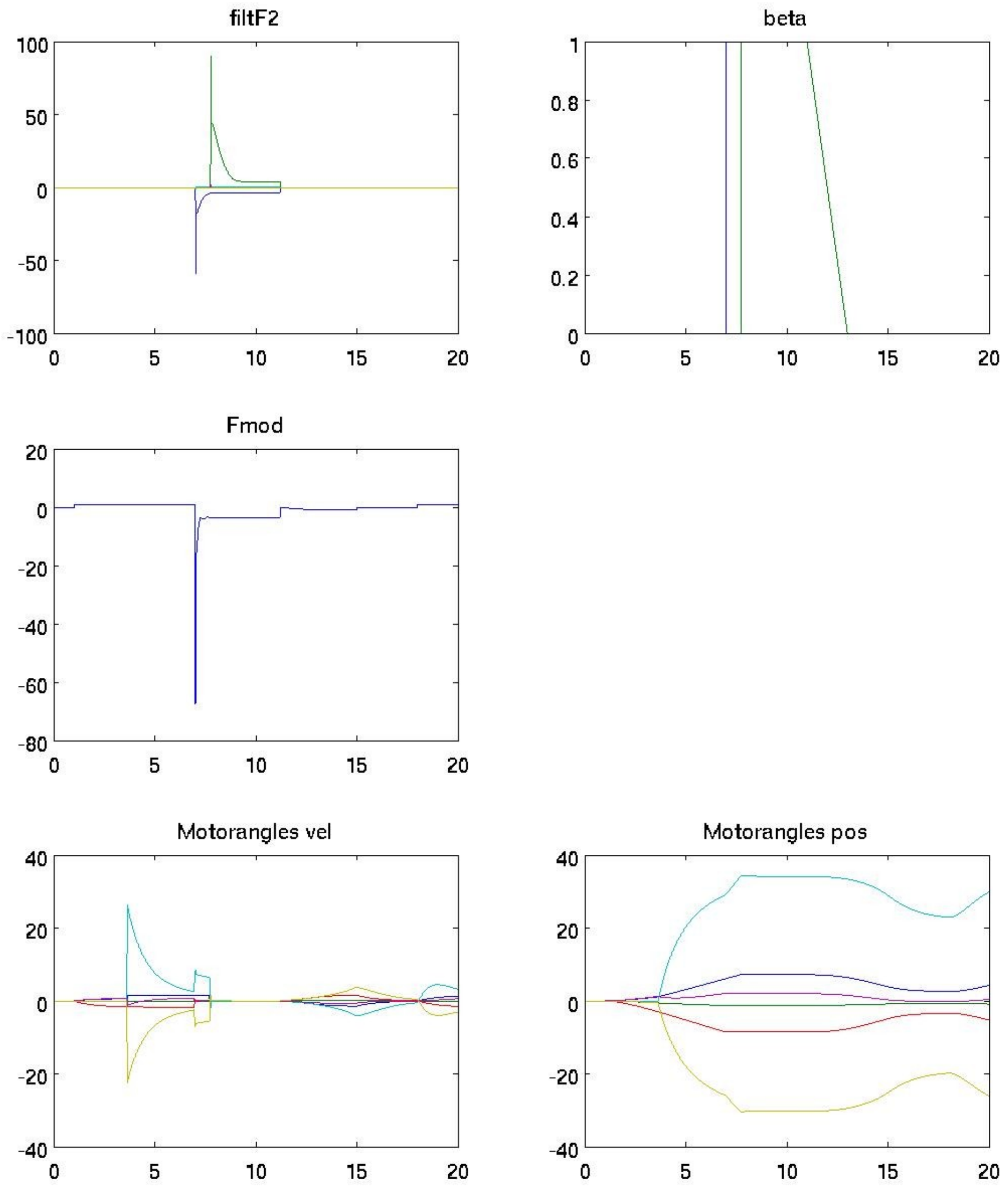
Simulation 5 – Control signals

C4 Simulation of the Fmod rc7 controller



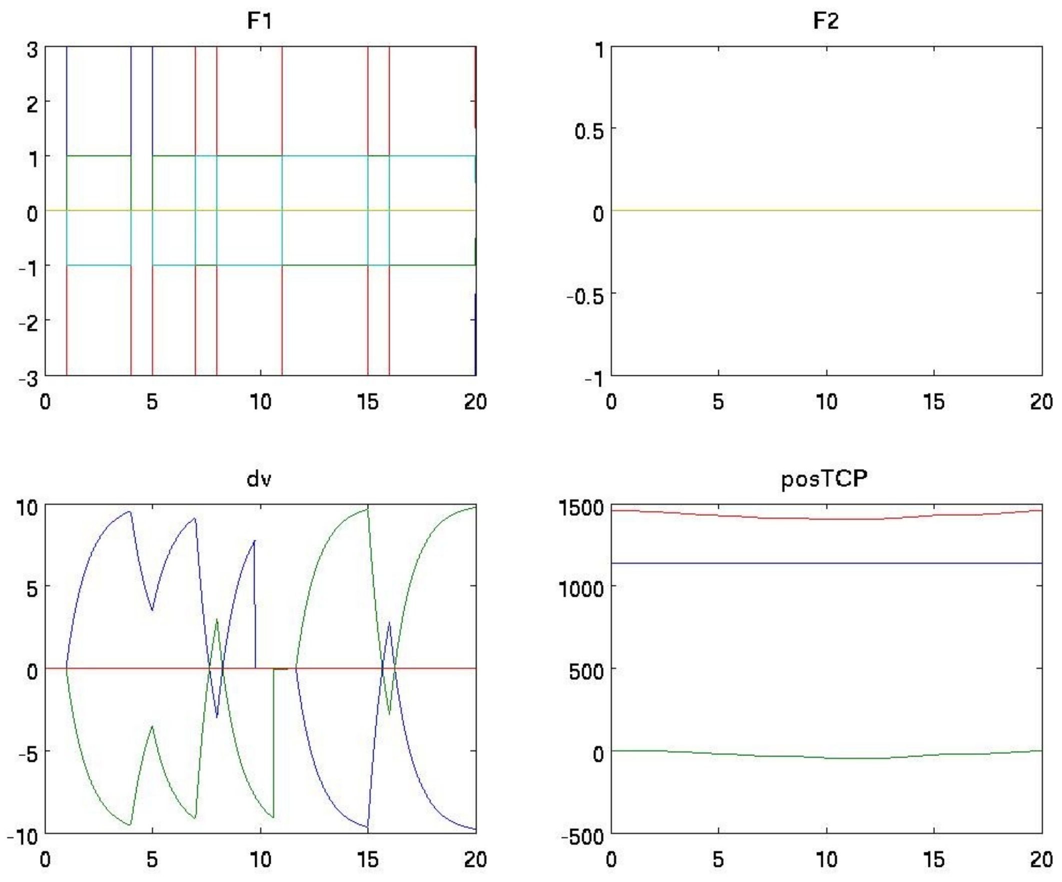
Simulation 6 – Main signals

C4 Simulation of the Fmod rc7 controller



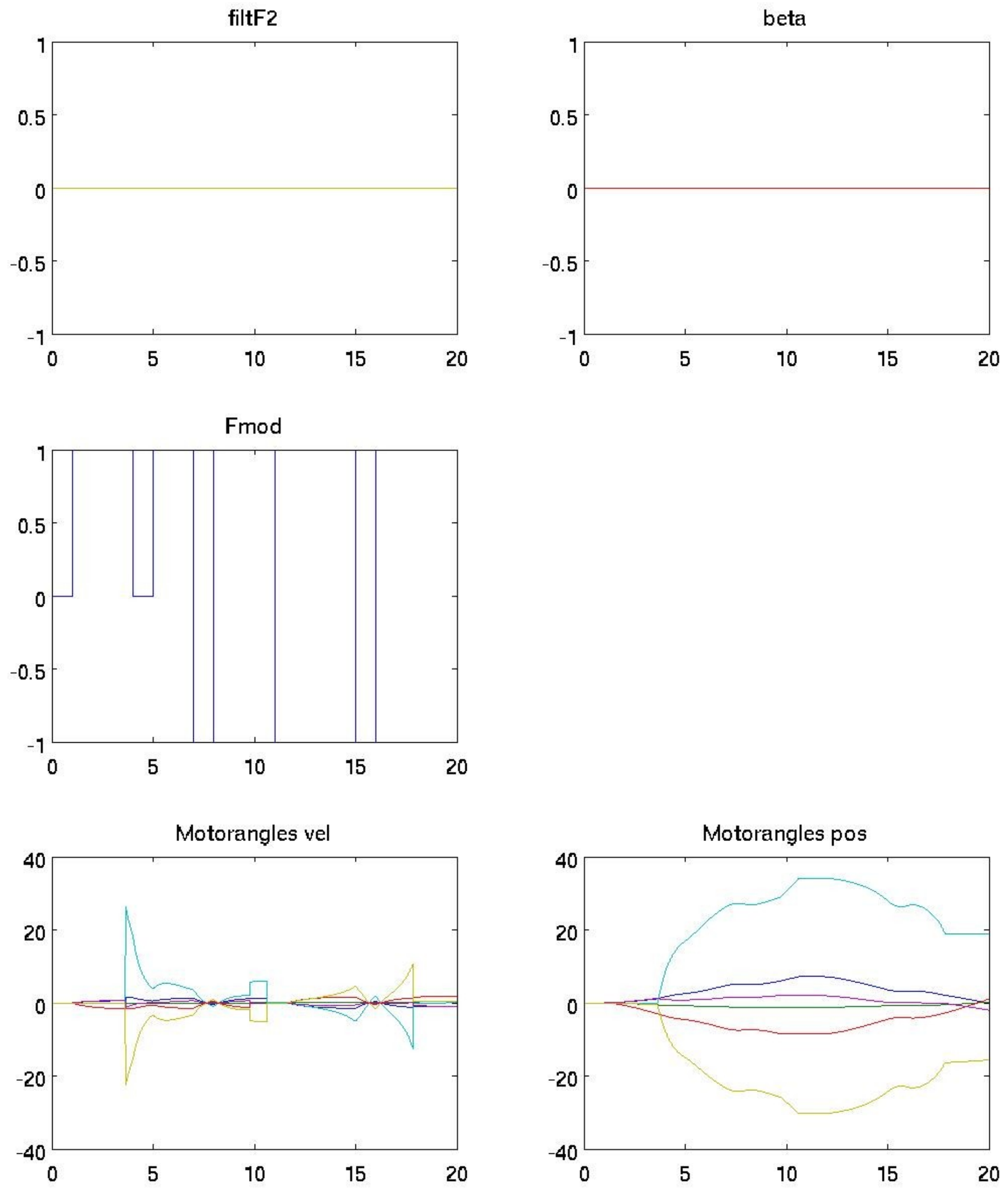
Simulation 6 – Control signals

C4 Simulation of the Fmod rc7 controller



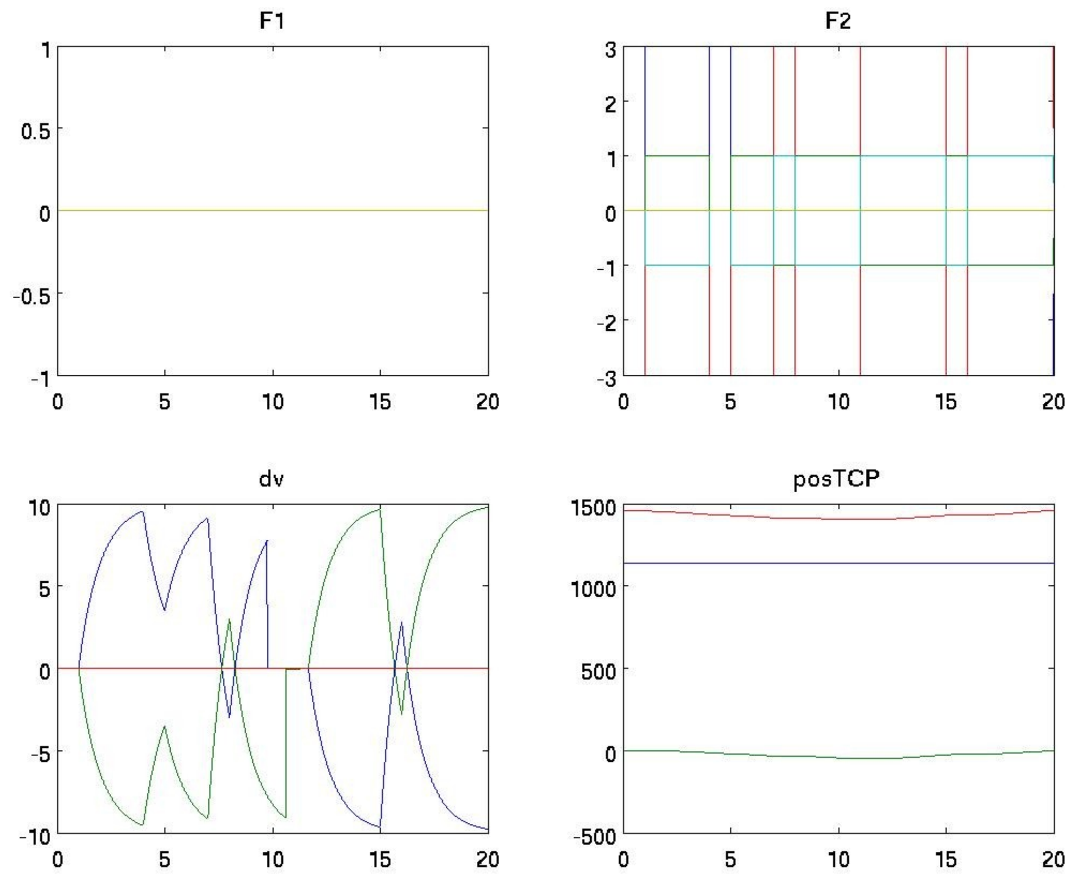
Simulation 7 – Main signals

C4 Simulation of the Fmod rc7 controller



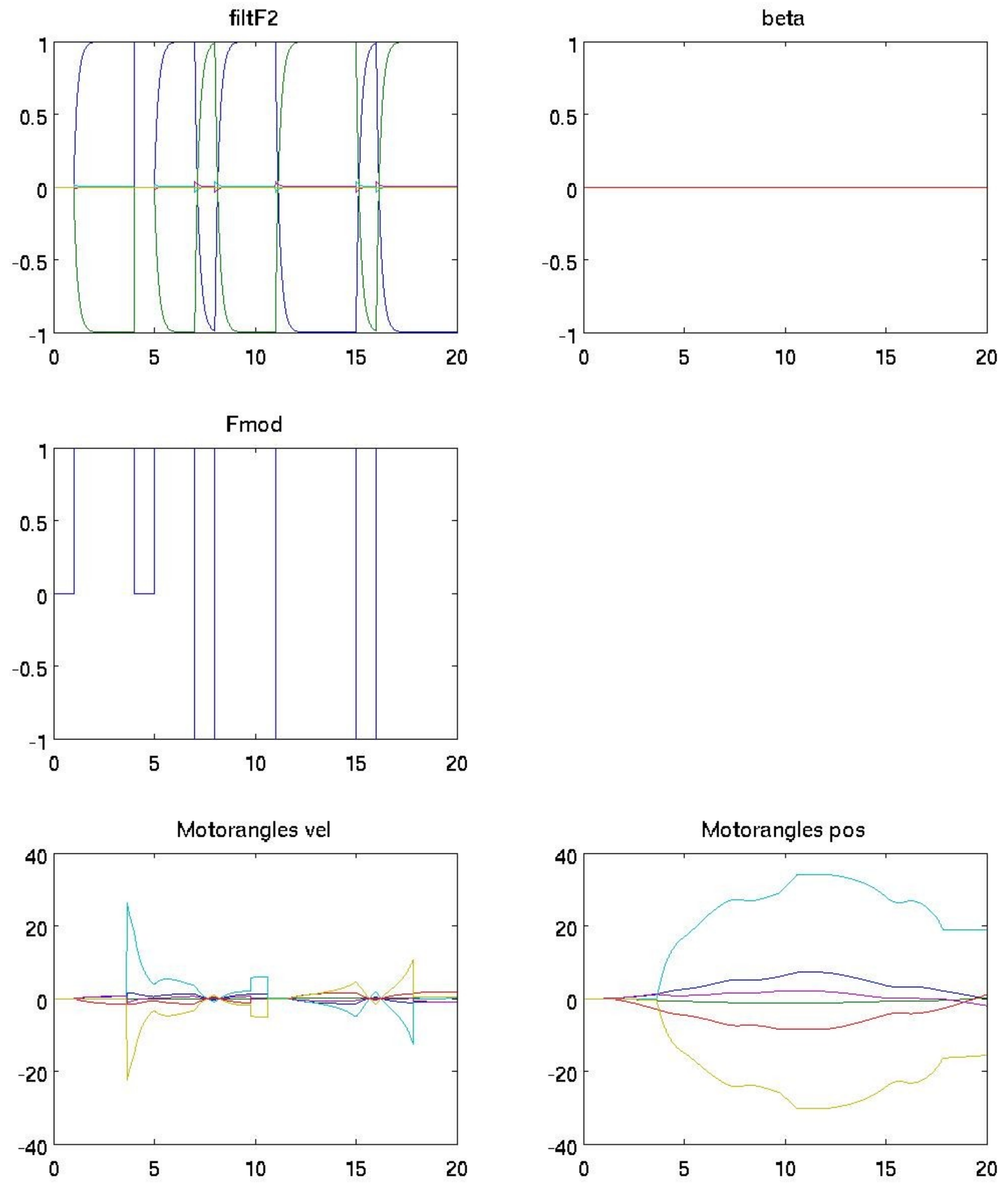
Simulation 7 – Control signals

C4 Simulation of the Fmod rc7 controller



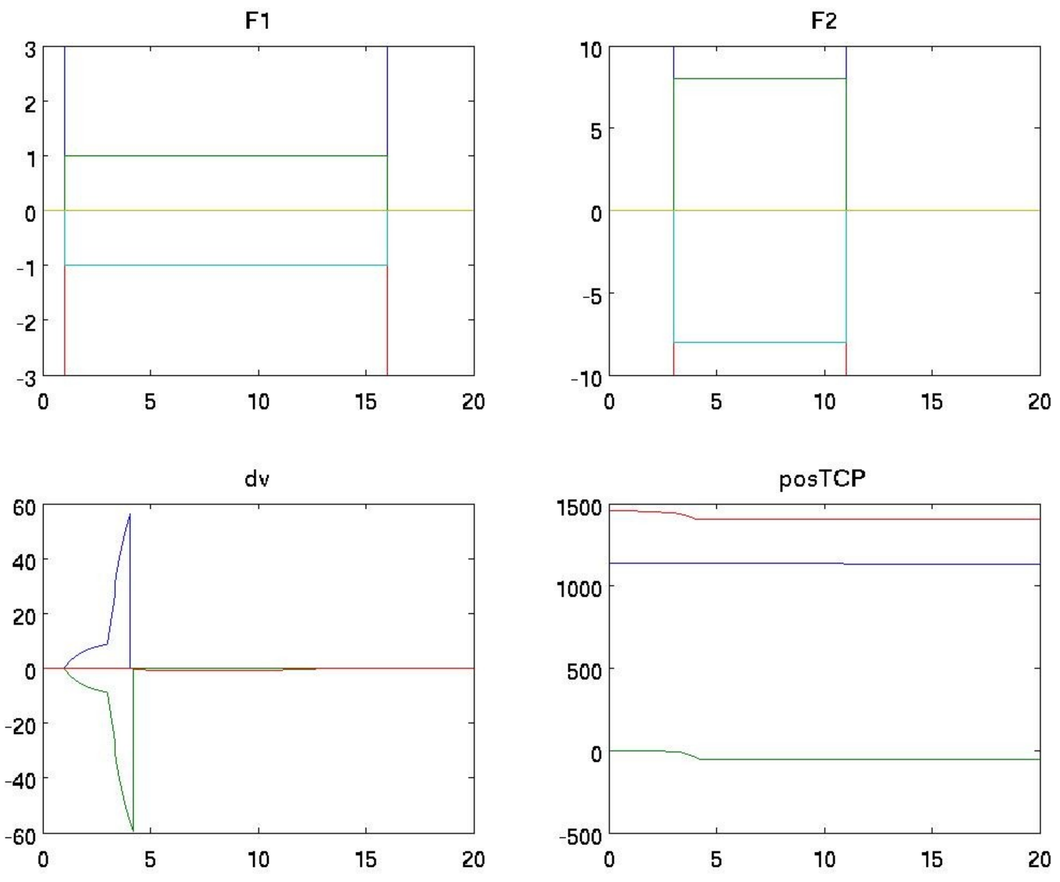
Simulation 8 – Main signals

C4 Simulation of the Fmod rc7 controller



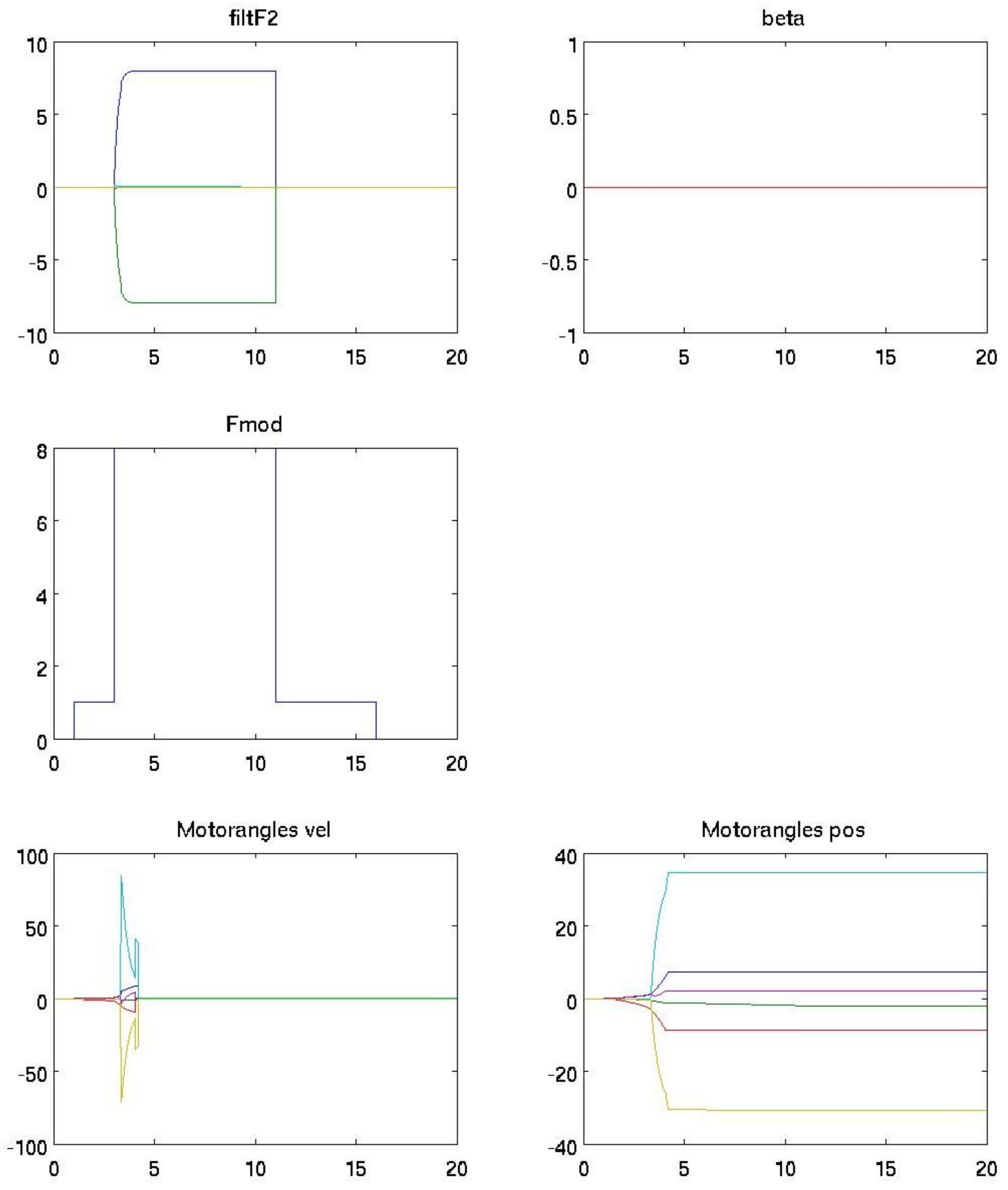
Simulation 8 – Control signals

C4 Simulation of the Fmod rc7 controller



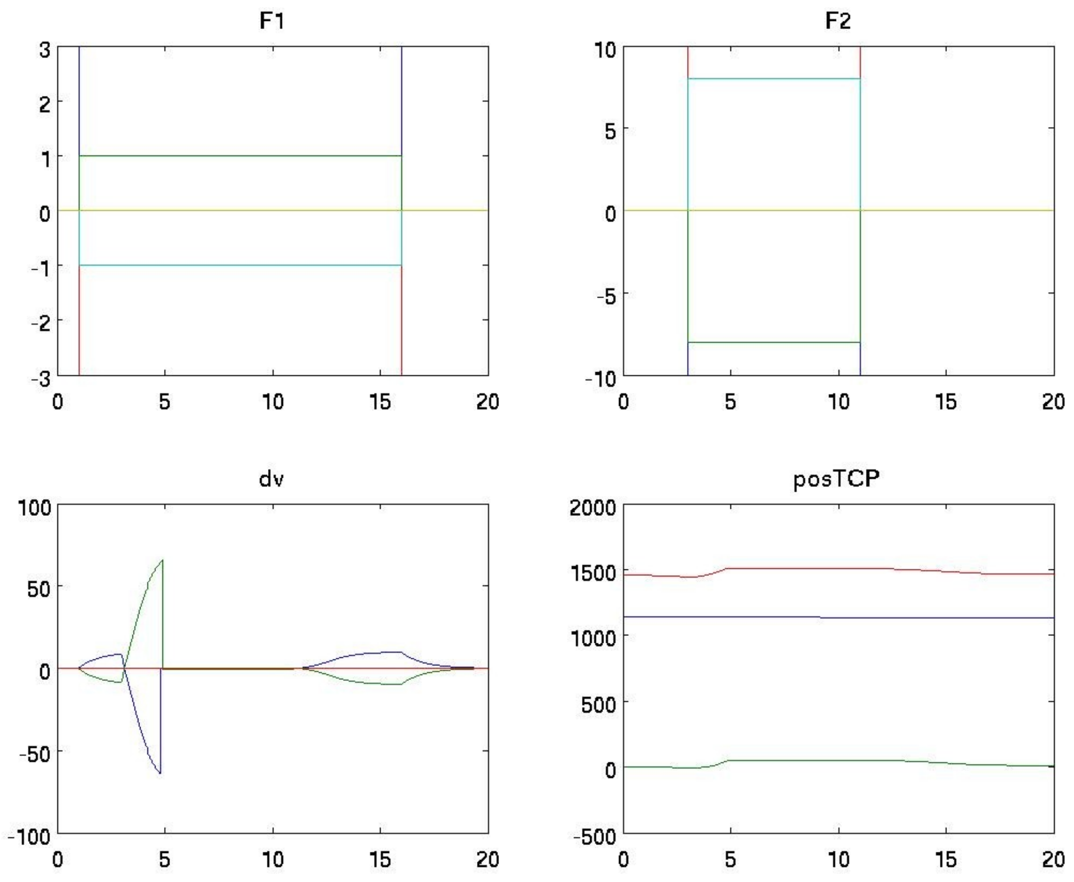
Simulation 9 – Main signals

C4 Simulation of the Fmod rc7 controller



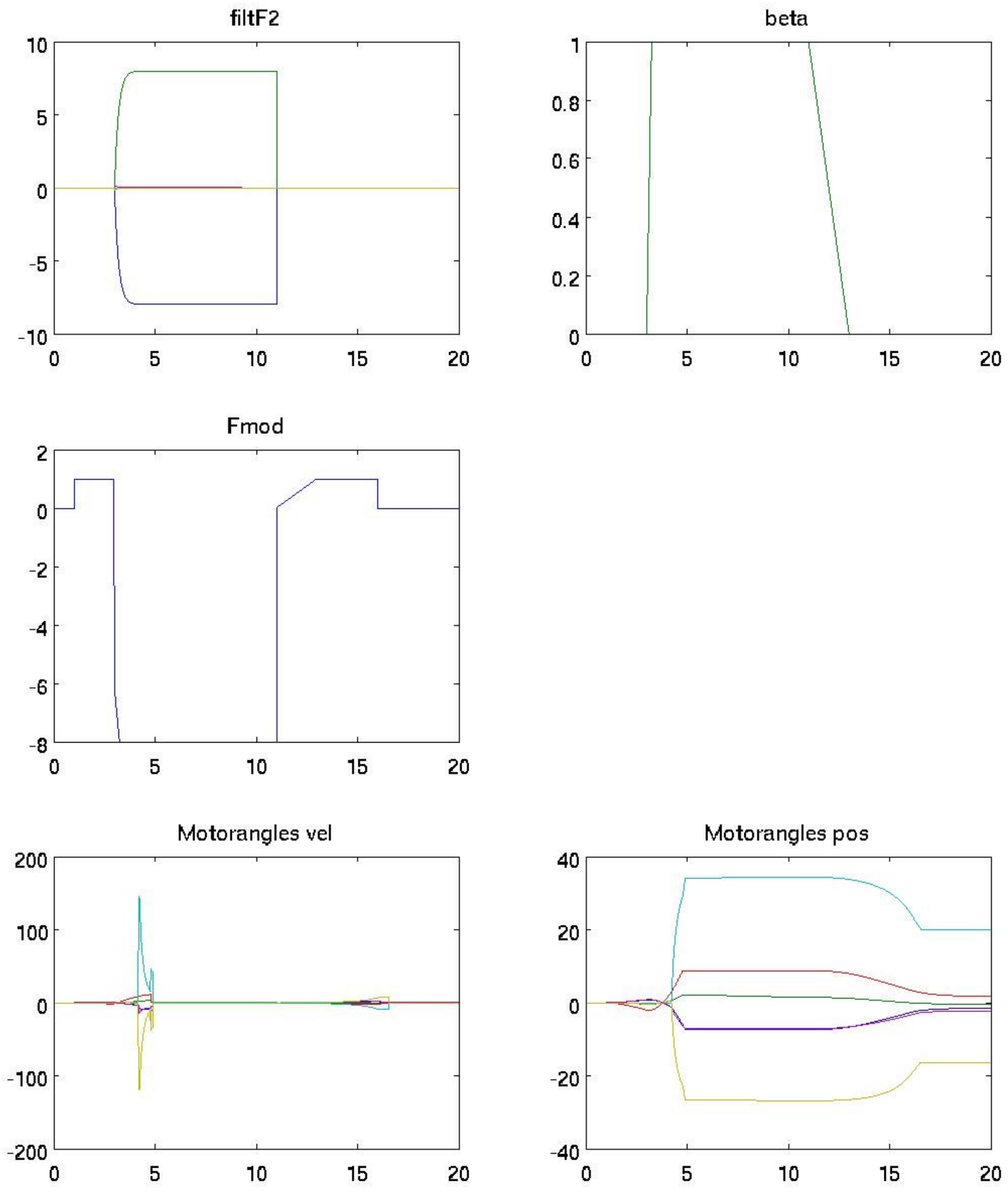
Simulation 9 – Control signals

C4 Simulation of the Fmod rc7 controller

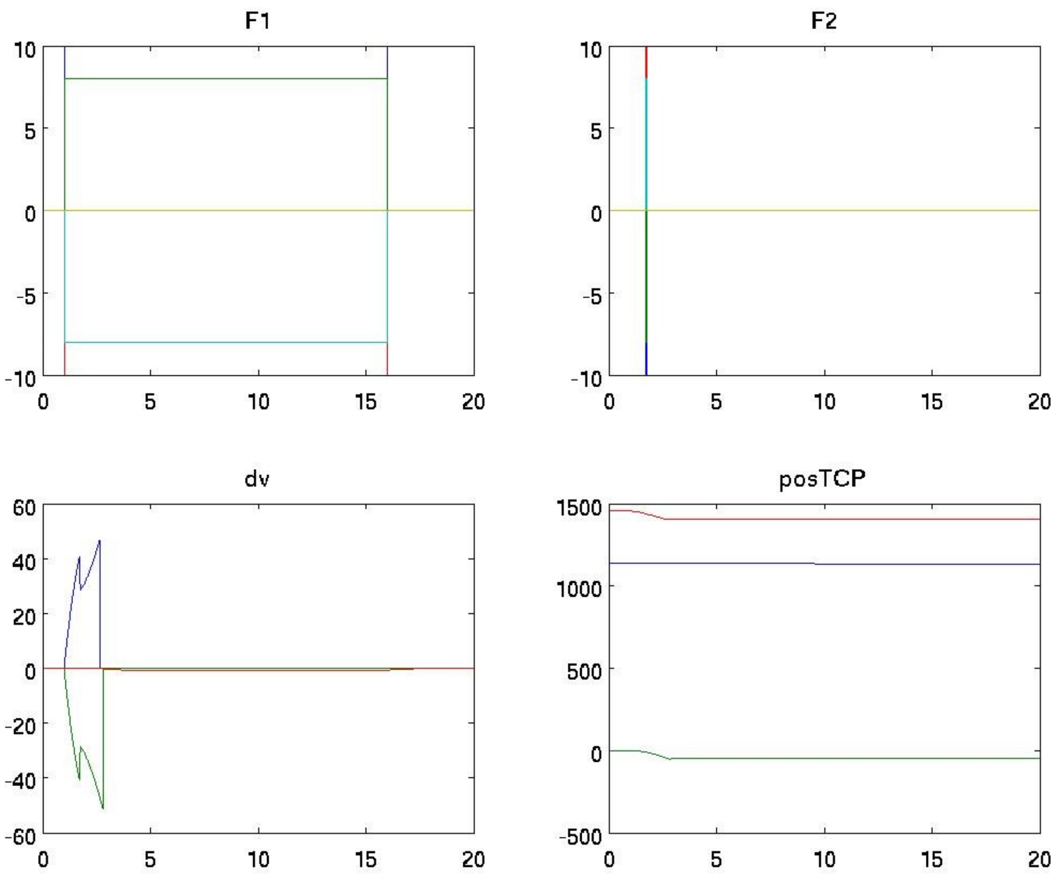


Simulation 10 – Main signals

C4 Simulation of the Fmod rc7 controller

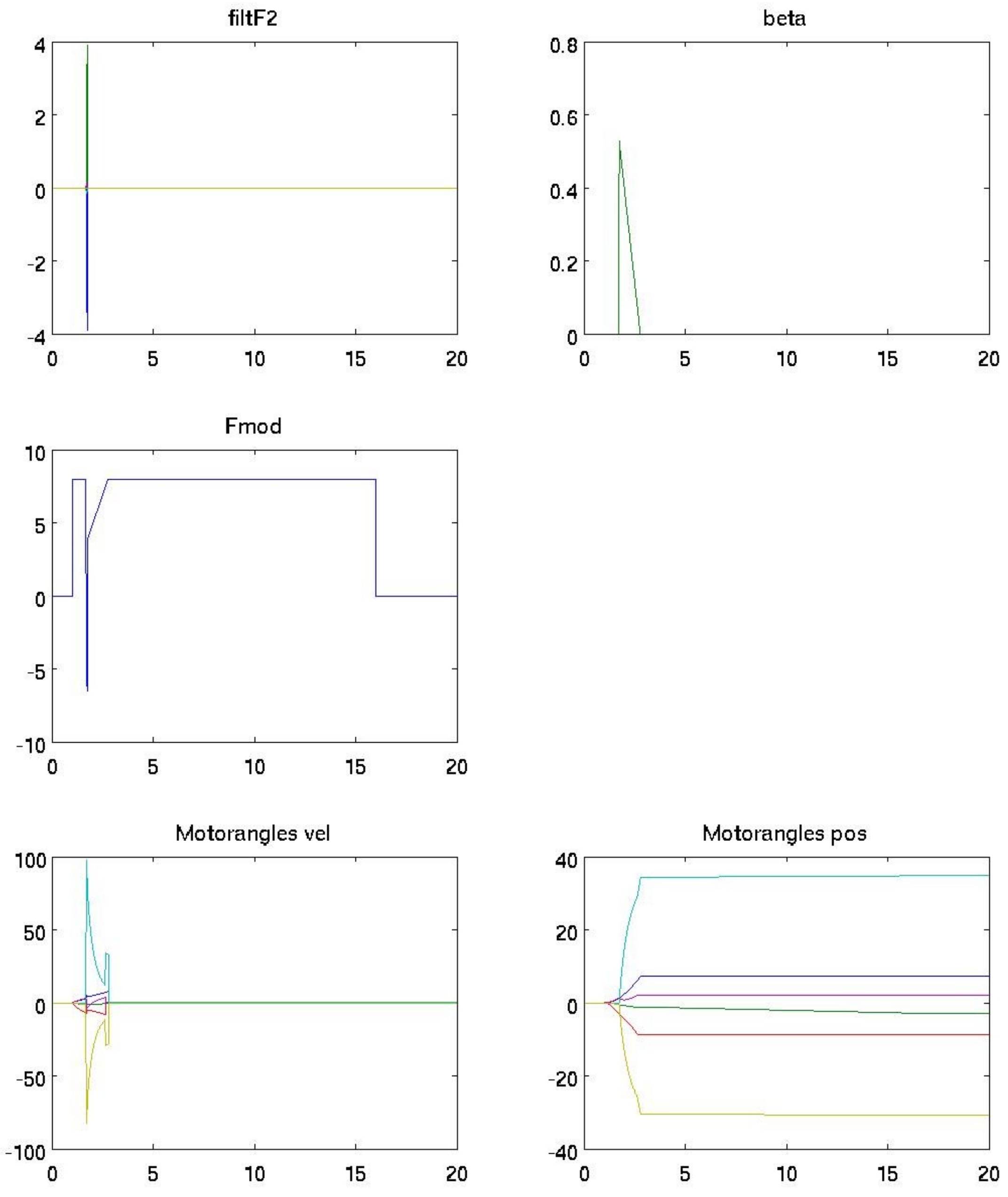


Simulation 10 – Control signals

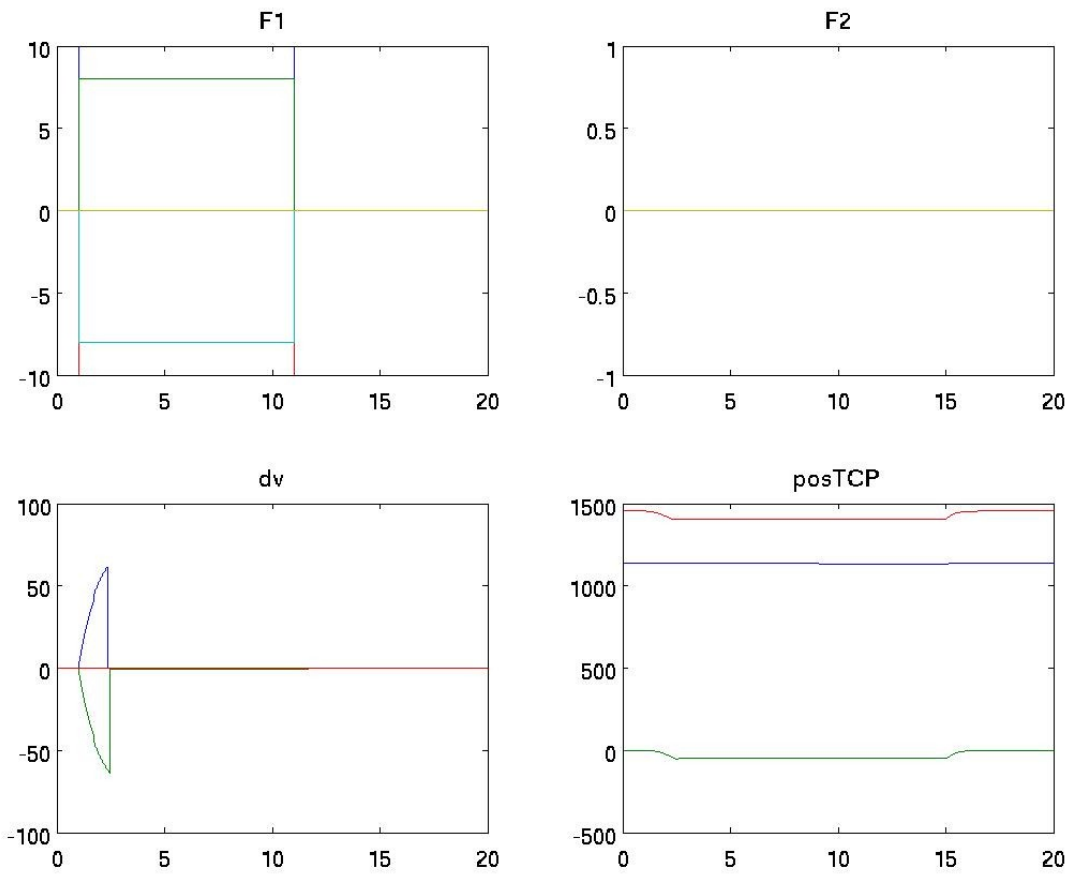


Simulation 11 – Main signals

C4 Simulation of the Fmod rc7 controller

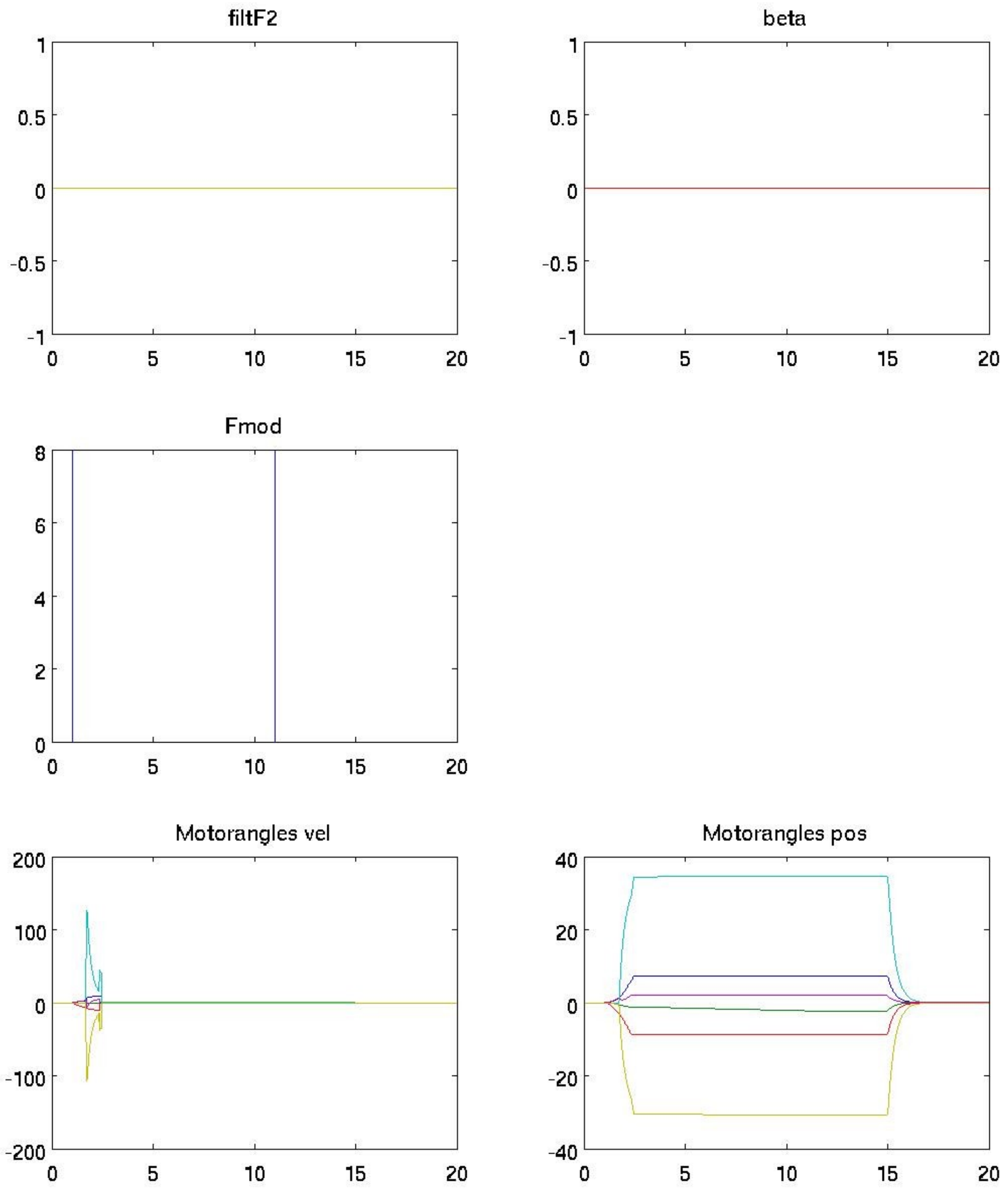


Simulation 11 – Control signals

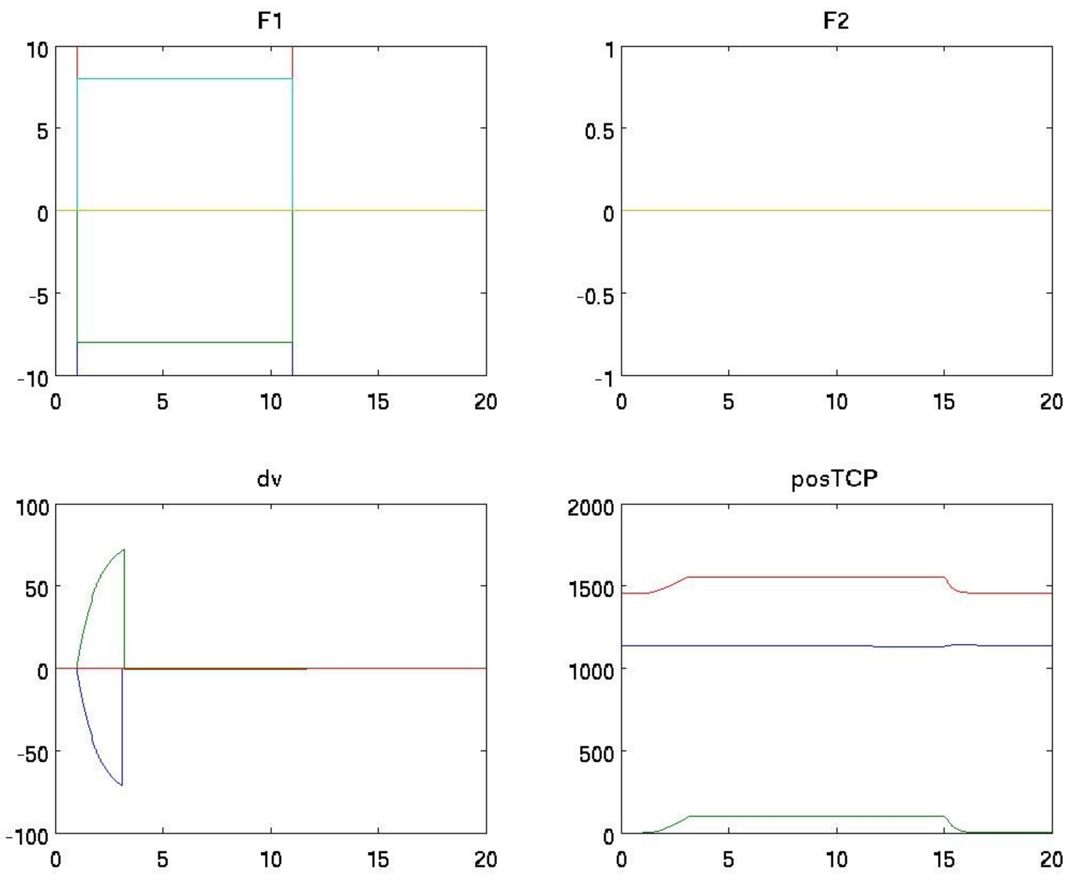


Simulation 12 – Main signals

C4 Simulation of the Fmod rc7 controller

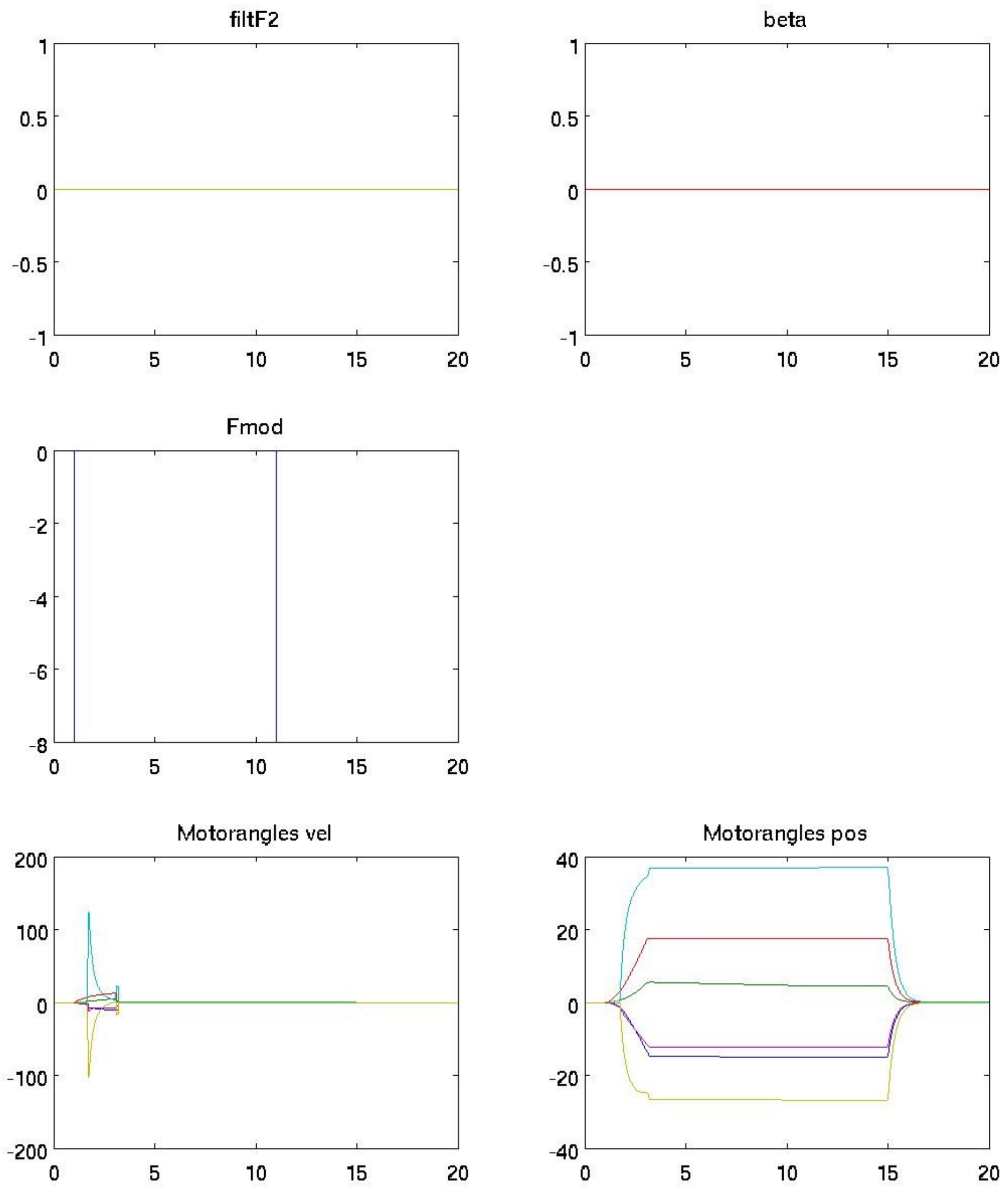


Simulation 12 – Control signals



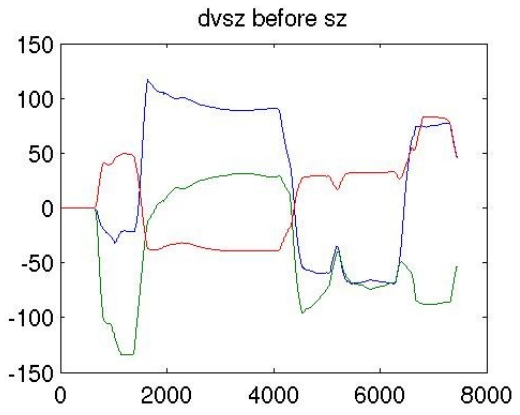
Simulation 13 – Main signals

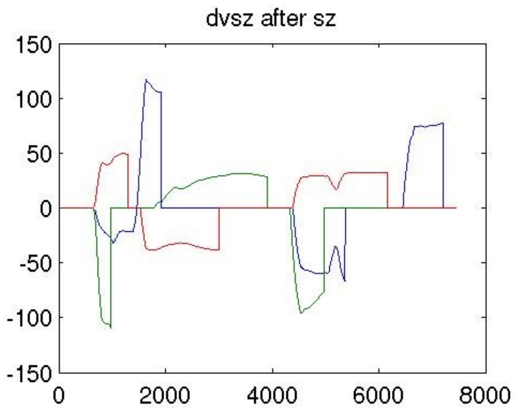
C4 Simulation of the Fmod rc7 controller

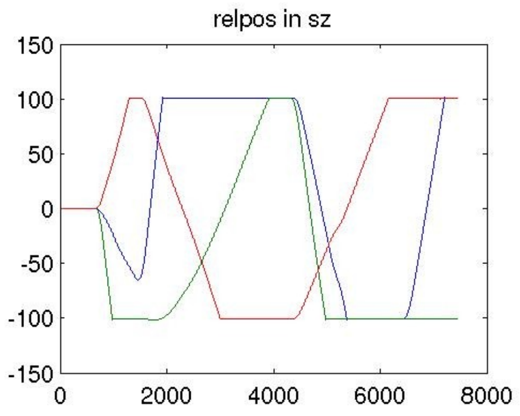


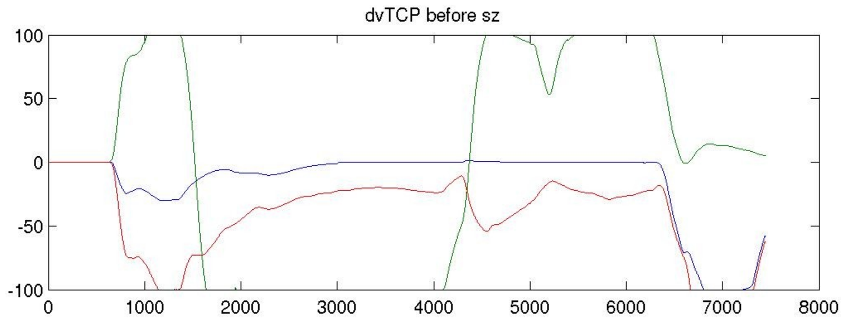
Simulation 13 – Control signals

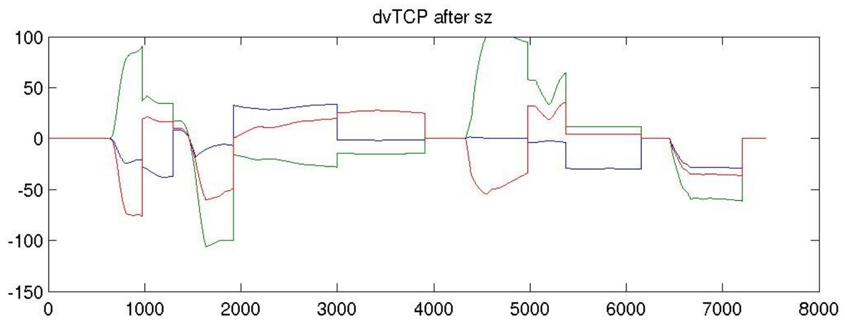
C.5 Results of tests of the safety zone on the robot





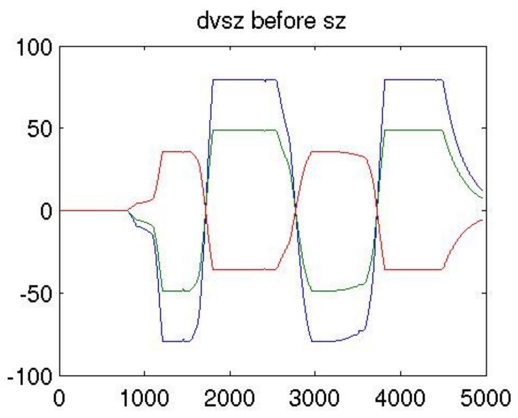


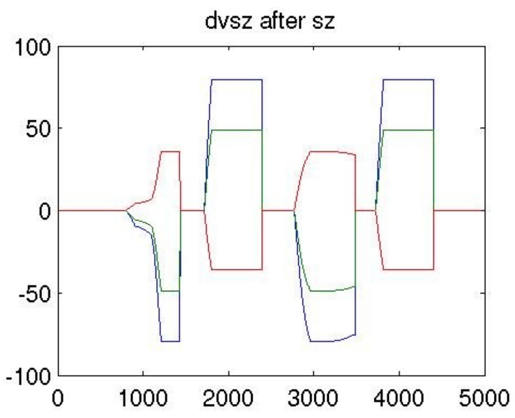


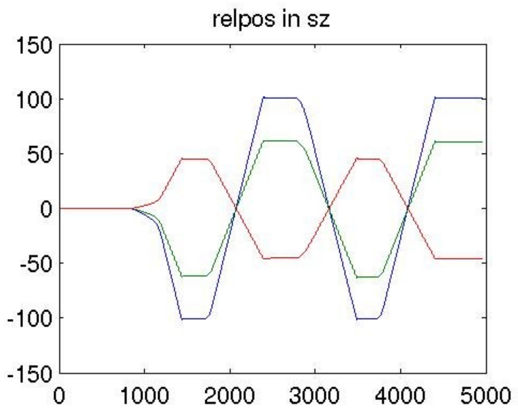


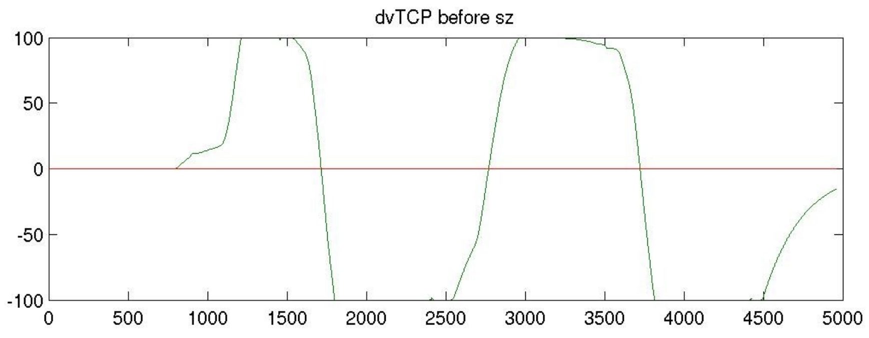
Test one – Using three control directions

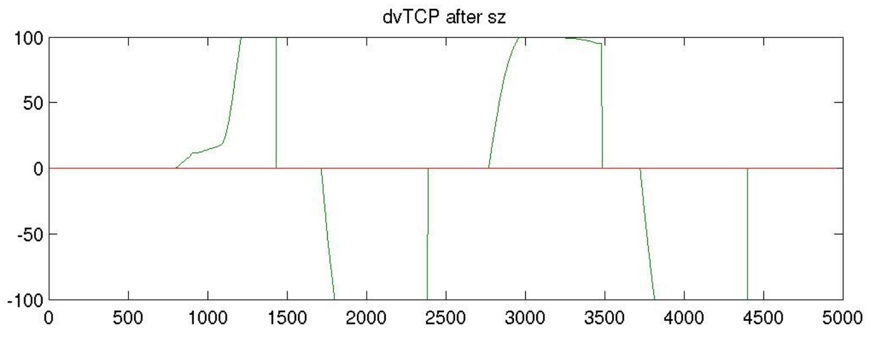
C.5 Results of tests of the safety zone on the real robot





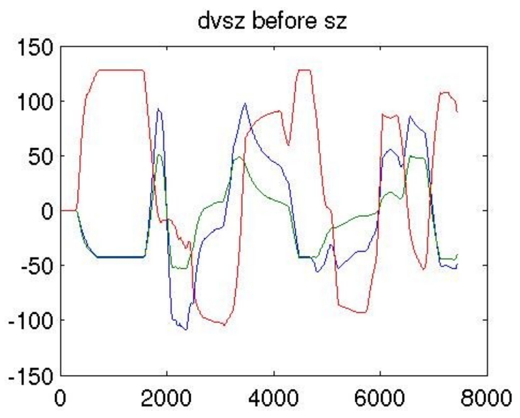


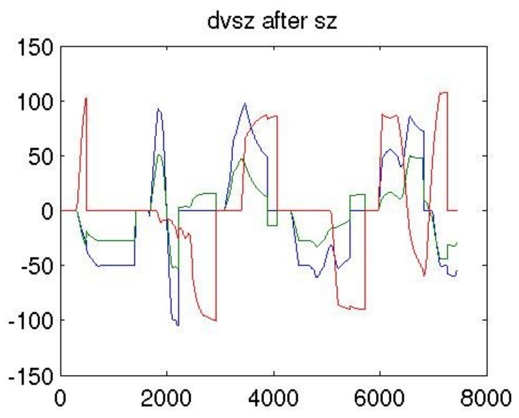


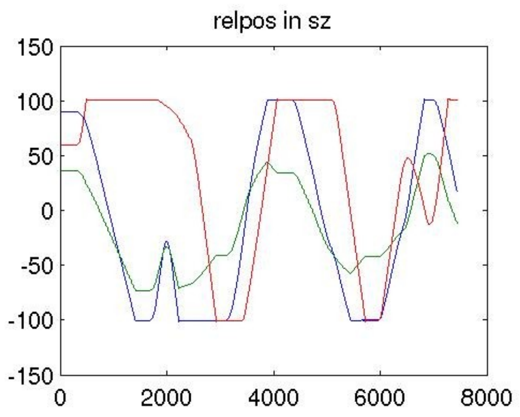


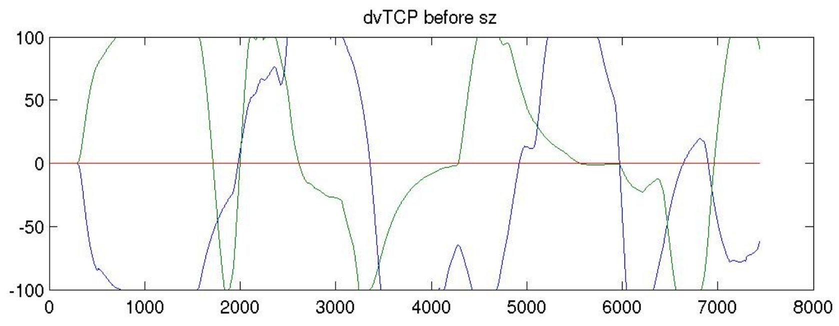
Test two – Only using one control direction

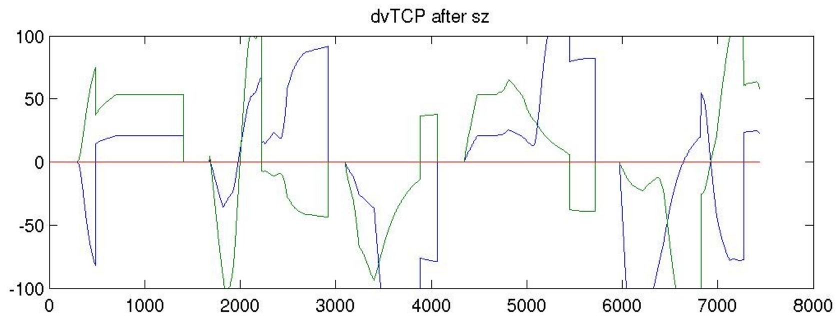
C.5 Results of tests of the safety zone on the real robot











Test three – Using two control directions

C.6 Results of tests of the controller on the robot

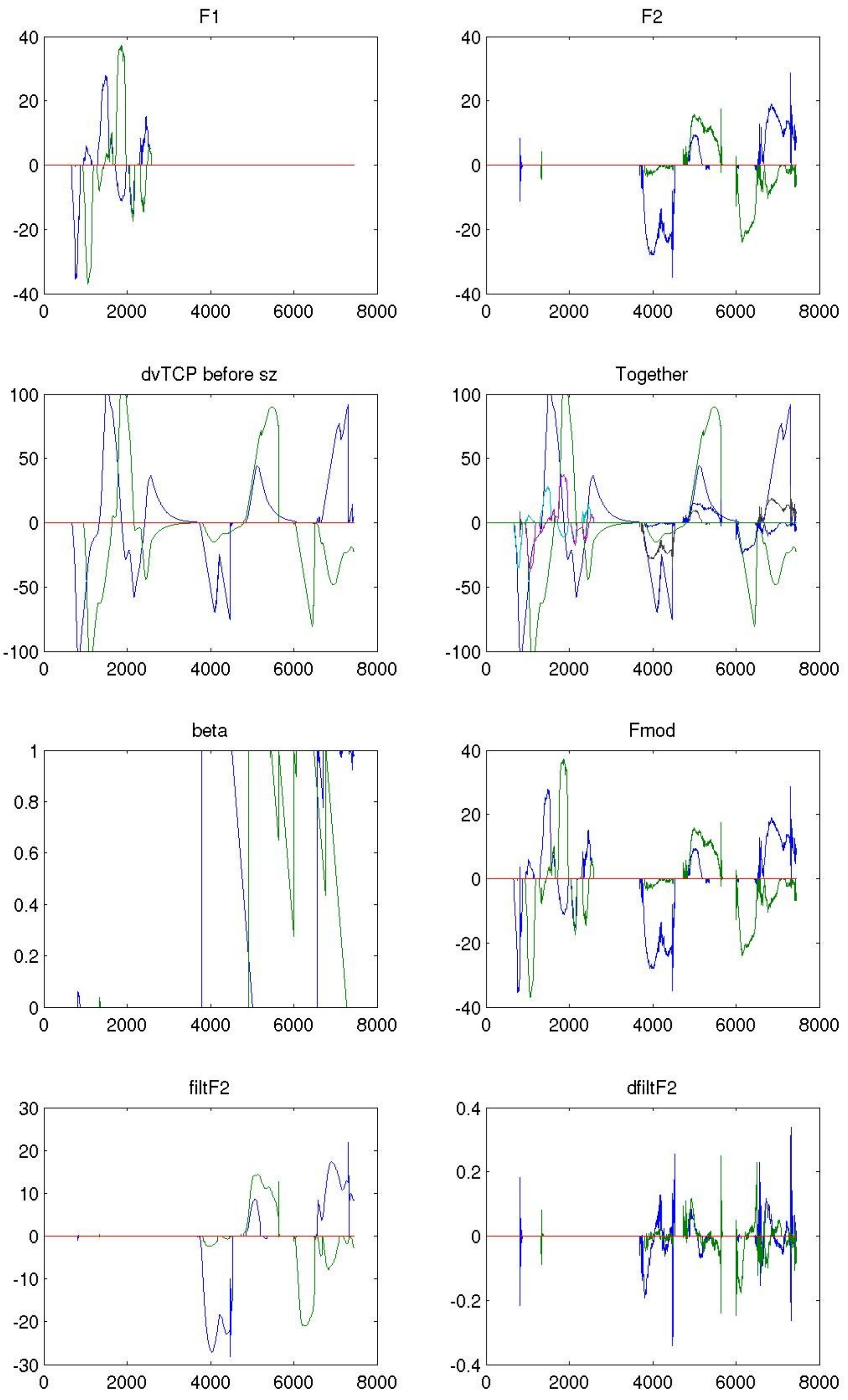


Fig 1 – In this test the robot was moved around with first F1 and then F2 (separately that is).The graphs in the lower half of the figure represents the internal variables of the controller.

C.6 Results of tests of the controller on the robot

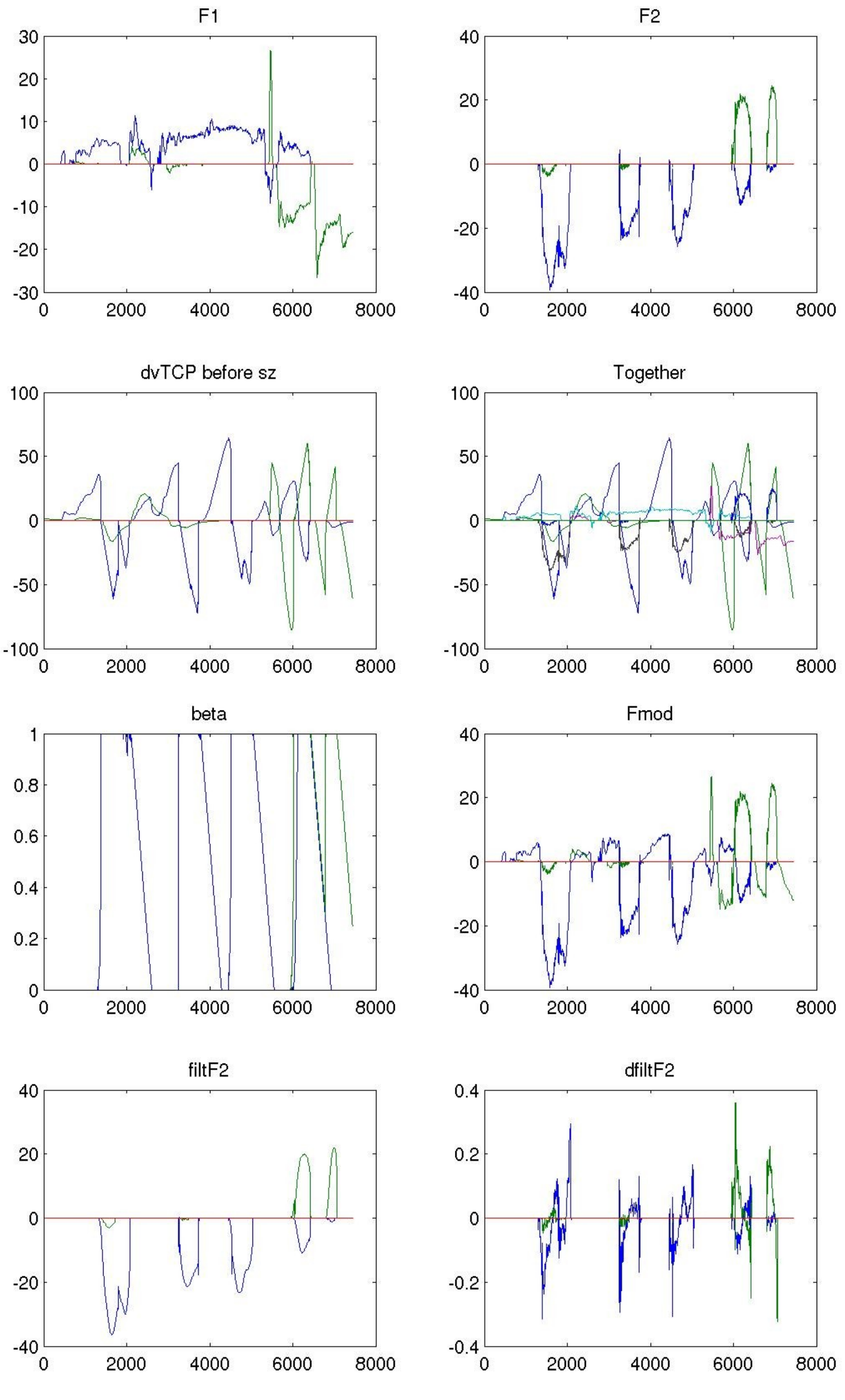


Fig 2 – In this test the robot is stimulated with both F1 and F2 at the same time. The purpose is to see how controller behaves in this case. It can be noticed how F2 changes the direction of the velocity when F1 and F2 are directed in an opposite way.

C.6 Results of tests of the controller on the robot

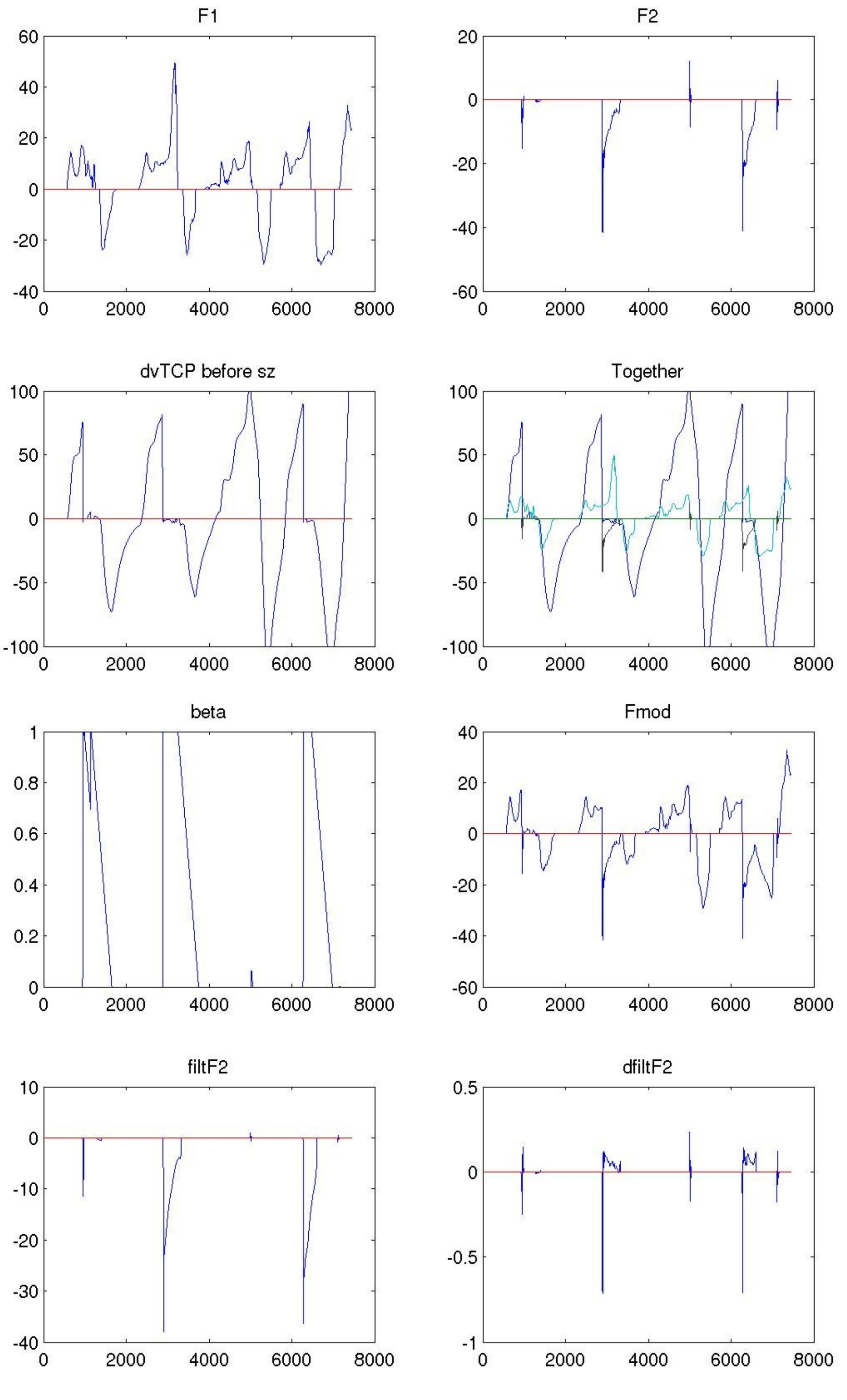


Fig 3 – This test is a crash into a Styrofoam wall. Notice how F2 is decreased. The disturbance at the sample around 5000 on F2 is due to a slight touch of the wall when the robot was passing it.