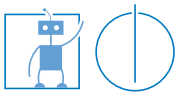

Robotics and Embedded Systems
Department of Informatics
Technische Universität München

Surgical Robot Path Planning & Simulation Based on VREP

Gheorghita, Daniel	**03683030**	**daniel0392@gmail.com**
Paulis, Bogdan	**03674059**	**bogdan_ paulis@yahoo.com**
Swazinna, Phillip	**03686497**	**p.swazinn@tum.de**
Löhr, Timo	**03686312**	**timo.loehr@tum.de**
Deutrich, Matthias	**03653725**	**matthias-deutrich@t-online.de**
Neves, Miguel	**03680468**	**chukas.spam@gmail.com**

Seminar *Robot-assisted Surgery in Clinics* SS 2017

Advisor:	Mingchuan Zhou
Supervisor:	Prof. Dr.-Ing. Alois Knoll
Submission:	2. August 2017

Everyone:

we had consistency issues regarding the labeling of figures / tables... so what i marked in green isn't necessarily wrong, but it needs to be changed so it will be consitent throughout the entire document

Controlling an Ophthalmic Surgery Robot with Matlab & Simulink

***Abstract*— The problem of controlling kinematic chains (robot manipulators) becomes more difficult with the number and strictness of the constraints involved. The present document will describe a method of controlling a surgical robot developed at the Technical University of Munich and provide a solution for simulating the device in the V-REP environment. It is clear that the safety standards for this type of device are above the ones for regular industrial robots.**

I. INTRODUCTION

This paper aims to present a solution for controlling an ophthalmic surgery robot (currently developed at the Technical University of Munich), using Matlab/Simulink and V-REP. More precisely, the aim of this document is to describe an implementation for path planning the robot needle inside the human eye. However, the problem of controlling robot dynamics is not addressed. spaces

The surgical robot (Figure 1) is a kinematic chain composed exclusively out of prismatic joints (linear piezo actuators). Component A (left side) is a Parallel Coupled Joint Mechanism (PCJM) which allows translational and rotational movement by differential displacement of two prismatic joints [3]. Component B is also a PCJM which is perpendicular to the first one. The final component (C) is a prismatic joint on which the needle is attached.

comma

to

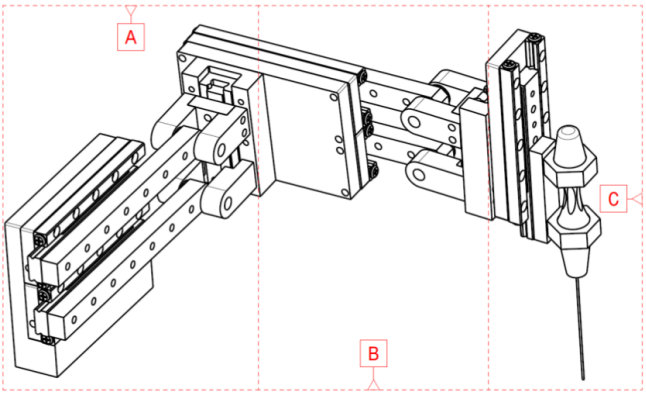

Fig. 1: Opthalmic surgical robot

The paper is organized in 8 sections. The first one offers an overview over the project that was tackled, section II describes the steps required in order to control the V-REP simulation environment from Matlab and Simulink. The third section describes the kinematics of the robot, section IV ilustrates an implementation for a GUI directly in Simulink. The actual control issues are described in the final 3 sections: section V handles generation of point-wise shapes to be used as trajectory descriptors, section VI provides trajectory planning that guides the needle to the entry point of the eye.Section VII presents the control loops that were used in order to ensure that the robot follows the desired paths and the final section will present our conclusions.

architecture

capital & missing article

while

missing space

II. CONNECTING VREP WITH MATLAB/SIMULINK (DANIEL & MIGUEL)

V-REP and Matlab

Straight forward connection from a Matlab script to the V-REP simulation environment can be done by using the files available in the V-REP installation folder. However, a few steps have to be performed. At first, the user has to select an object from the environment which will be present there at all times (not an auxiliary scene object that might be deleted at some point). A child script has to be added to the selected object by right clicking the object → Add → Associated child script → Non threaded. In the initialization call, the following line should be inserted: `simExtRemoteApiStart(19999)` . The figure below is an example of how this step should look at the end. The script and the scene should be saved before doing the configuration on the Matlab side.

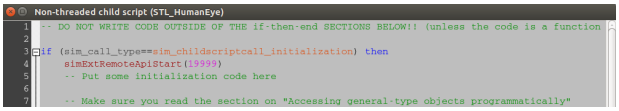

Fig. 2: V-REP side connection setup

The library files offered by V-REP should be copied in the current working folder or added to the Matlab path. The library files can be found in `../VREP/programming/remoteApiBindings/matlab/matlab` and `../VREP/programming/ remoteApiBindings/matlab/lib/64Bit` (or `../32Bit` folder, depending on the operating system). Next, the V-REP library should be included in the Matlab script, all previous connections (if any) should be closed and a new connection should be established:

space

comma

```
vrep=remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
vrep.simxFinish(-1); % just in case, close all opened connections
clientID=vrep.simxStart('127.0.0.1', 19999, true, true, 5000, 5);
```

Fig. 3: Matlab side communication setup

If the connection is successful, the `clientID` variable should contain a non-negative integer (it can be ”0”). It is recommended to check the connection before executing the desired code. At the end of the code execution, the connection should be closed.

why string?

```
if (clientID>-1)
    disp('Connected');

    % CODE START

    % CODE END

    % Close connection
    vrep.simxFinish(-1);

end
```

Fig. 4: Matlab code loop

A good practice rule is to delete the `vrep` object from the workspace when the connection is closed.

```
% Delete vrep object
vrep.delete();
```

Fig. 5: Delete VREP object

In order to connect Matlab to V-REP after performing the above steps, the V-REP simulation should be started. Running the Matlab barebone script completely described above should be enough to connect and disconnect from V-REP.

Controlling V-REP objects from a Matlab script is done through object handles. The `simxGetObjectHandle()` can be used as in the example:

```
[returnCode, prismaticJointS22] = vrep.simxGetObjectHandle(clientID,
                                  'Prismatic_joint_S22', vrep.simx_opmode_blocking);
```

Fig. 6: Get VREP object handle

In order to obtain information about objects (i.e., position) for which handles are already available, one must first use:

```
[returnCode, needleTipPos] = vrep.simxGetObjectPosition(clientID,
                             needleTip, -1, vrep.simx_opmode_streaming);
```

Fig. 7: Get VREP object position (first call)

Further requests of the same information should be done using the method call (similar to the previous one, except the last parameter):

```
[returnCode, needleTipPos] = vrep.simxGetObjectPosition(clientID,
                             needleTip, -1, vrep.simx_opmode_buffer);
```

Fig. 8: Get VREP object position (following calls)

For manipulating object properties (i.e., position), one can call a setter method such as:

```
joint1 = 0.01; % desired position
[returnCode] = vrep.simxSetJointTargetPosition(clientID, prismaticJointS22,
                                               joint1, vrep.simx_opmode_blocking);
```

Fig. 9: Set VREP object position

By adding some of these method calls in the previous script, the user can interact with the V-REP simulation environment from Matlab. A full description of the available methods in Matlab can be found in the V-REP Documentation [1]. For debugging purposes, each method call outputs a value in `returnCode`. The bits of this variable can be interpreted as flags for errors [2].

V-REP and Simulink

"Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. It is integrated with MATLAB" [4]. It also provides numerous Solvers with fixed time-step for "Real-time" simulation and a plug-in (Real-Time Workshop plug-in) that enables the deployment into real-time operating Systems. With the above mentioned benefits in mind a communication solution between V-Rep and Simulink was searched for.

V-REP and Simulink using the Matlab API: The aforementioned V-REP Matlab API can also be used from within Simulink with the following adjustments:confusion: which one now?
From within Matlab: --> maybe second one "changes in matlab"

- the `remoteApi` should be initilized in a script
- the `vrep` object should be declared as global

From within Simulink: might want to adjust this too then

- `Interpreted Matlab Function` block must be used to interact with the V-REP Matlab API.

The need for `Interpreted Matlab Function` blocks proved to be a strong disadvantage as such block only supports a single scalar value as its input, a single scalar value as its output and needs to behave as an isolated script. This means that a new connection thread needs to be created every time an `Interpreted Matlab Function` is called, the single scalar input sent, the single output scalar read and finally the connection thread needs to be closed. The result of such was that each one of the six robot joints needed to be addressed by an individual `Interpreted Matlab Function` block, resulting in a deadly propagational motion delay between each one of the Robot's Joints. This propagation delay led to absolute accuracy positional errors that were too large to be accepted.

The steps described above are enough for a simple control of the V-REP simulation environment from Matlab/Simulink. These can successfully handle control tasks when there are no strict time or accuracy constraints. For the case of the surgical robot, the simple script can take the needle from the current position to a desired final position. However, in this case there are strict accuracy conditions which have to ensure that the needle point does not only converge to a desired position, but it has to follow an extremely precise trajectory. This can be difficult to achieve due to sequentially sending joint target positions. The kinematic chain will accumulate offsets due to the delays between joint commands. This will result in a imprecise or jerky trajectory of the needle tip. Another disadvantage of using plain setter and getter methods for controlling simulations can be observed when the Matlab side has a large volume of computations to process (i.e., video processing). In the case of high volume of computations required in the Matlab

side, one could manually control the V-REP simulation steps. The communication mode should be set to synchronous by calling `simxSynchronous(clientID,1)` in Matlab before calling the command for starting the communication. Next, control methods can be called as before. However, the V-REP simulation will only advance after calling `simxSynchronousTrigger(clientID)`. Basically, V-REP waits for the trigger from Matlab and only then executes the commands.

V-REP and Simulink using Shared Memory: If a system controller is to be realized within Simulink a communication solution that enables the simultaneous sending and receiving of data arrays (to send all of the robot's joint position at once) was needed.Since the previously mentioned Matlab API did not suffice a better solution was needed.

In [6] a Shared Memory communication library was introduced. In [5] Coppelia Robotics states that "The next version of the remote API (V-REP 3.4.1) on the other hand will also support shared memory (instead of socket communication)."[5]. This is because shared memory was supported in the previous V-REP versions up to 3.2.XX and was no longer supported at the time of writing in the current V-REP version (3.3.XX). As Coppelia Robotics, the makers of V-REP stated that the version 3.4.XX will once again support a shared memory communication protocol, we knew that this was a solution with usability in the future. It is worth noting that the current Simulink V-REP shared memory communication solution was realized using V-REP version 3.2.3 and a simulink version of choice. In V-REP a Lua Script with the following modifications was added:

in the section:

```
if (sim_call_type==sim_childscriptcall_initialization) then
id=tonumber(simGetScriptSimulationParameter(sim_handle_self,
"id EyeRob"))
-- Get the shared memory ID number manually defined and equal
in both V-REP and Simulink for the communication to be
established

        q_in={0.0,0.0,0.0,0.0,0.0,0.0}
        q_out={0.0,0.0,0.0,0.0,0.0,0.0}
    --define the format of the data to be exchanged

    result,floatMem_IN  =simExtShareMemoryOpen
    ("EyeRob_entrada_"..id,6*4)
    --Start the shared memory thread using the ID number
    defined. this thread is for data in of the Robot
    s position in joint space. Its format is
    (6 float * 4 bytes per float)

  result,floatMem_OUT  =simExtShareMemoryOpen
  ("EyeRob_salida_"..id ,6*4)
  --Start the shared memory thread using the ID number
    defined. this thread is for data out with the current
    Robots position in joint space
    (6 float * 4 bytes per float)
```

in the section:

```
if (sim_call_type==sim_childscriptcall_actuation) then
result,data=simExtShareMemoryRead(floatMem_IN)
--Read data
q_in=simUnpackFloats(data,0,6)
--unpack data into floats to be read
data_out=simPackFloats(q_out,0,6)
--Pack data into floats to be sent
result=simExtShareMemoryWrite(floatMem_OUT,data_out)
--send data
```

in the section:

```
in if (sim_call_type==sim_childscriptcall_cleanup) then
simExtShareMemoryClose();
--destructor
```

In Simulink a new library needed to be compiled. Once the source code was downloaded from [6] it was then compiled and added to Simulink by following the guide in [7]. The Resulting `Generic Shared Memory Block`

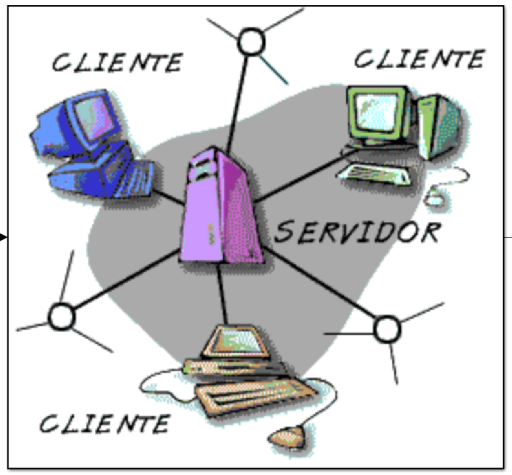

Fig. 10: Generic Shared Memory Block

could now be used to communicate with V-REP from within Simulink.It needs to be configured in the following way for the communication to succeed:

- **Base Name:** 'EyeRob' - Same as Declared in V-REP
- **Device Number:** Same as Declared in V-REP
- **Sizes Input Vector:** [6] - Same size as as q_{in} in V-REP
- **Sizes Output Vector:** [6] - Same size as as q_{out} in V-REP
- **Data Types Inputs:** ['f']
- **Data Types Outputs:** ['f'] - Attention is required here. Matlab is currently a 64bit application and V-REP 3.2.x is a 32bit application.As a result a single in Matlab (32 bit) is a float in V-REP, and should be declared as a float ('f') in the data inputs and outputs in the simulink block.

Forcing Simulink to run in Real-time: Once the described procedures were followed, Simulink's simulation using the `Generic Shared Memory Block` was running faster than Real-time.As a result no Real-time control of the robot was possible. To Overcome this issue a `Simulink Block for Real Time Execution` from [8] was used.

III. Robot Kinematics(Miguel)

In order to control the robot in Task Space, the Kinematics of the system are needed for computing the transformations

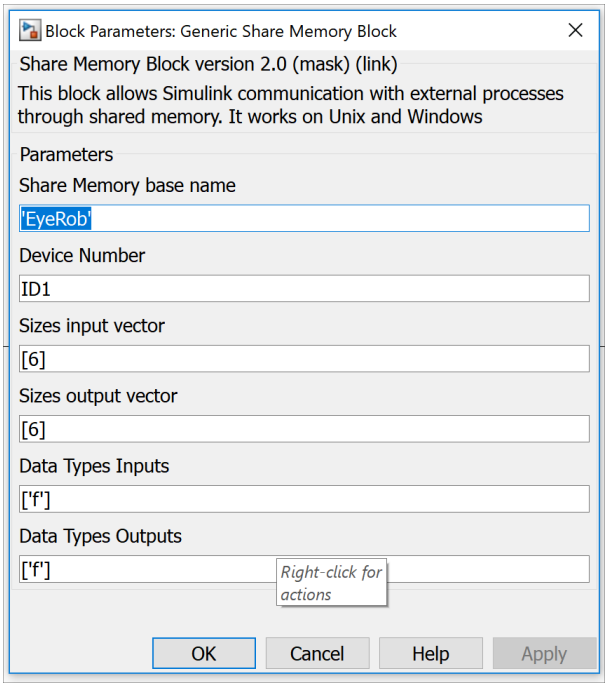

Fig. 11: Generic Shared Memory Block Configuration

small

again...

to and from the Joint Space of the robot. This section provides a short description of the Direct and Inverse Kinematics of the Opthalmic Surgical Robot with needle.It then proceeds to establish a relationship between Task Space velocities and Joint Space velocities, for both restricted and unrestricted motions of the robot's end-effector.

comma As shown previously in Fig.1 the Opthalmic Surgical Robot addressed in this paper consists of "five prismatic piezo actuators arranged as two Parallel Coupled Joint Mechanism (PCJM) elements and one prismatic element. An optional revolute joint in the end effector allows 6 degrees of freedom of movement of the surgical instrument."[3] Each PCJM "allows translational as well as rotational movement by differential displacement of the two prismatic joints."[3]

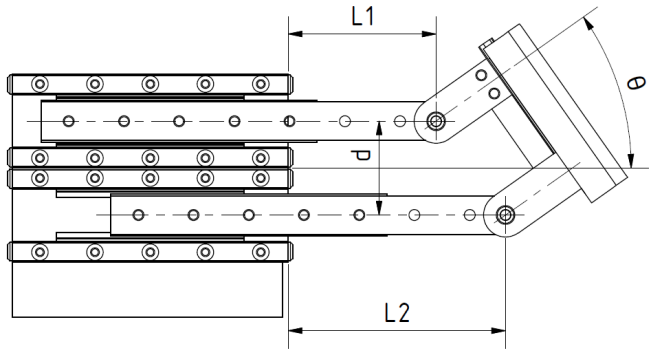

Fig. 12: Parallel Coupled Joint Mechanism (PCJM) elements

small

The rotational movement that results can be calculated by:

$$\theta = arctan\left(\frac{L_2 - L_1}{d}\right) \quad (1)$$

Where d = 17mm is the distance between the center of each prismatic joint, L_1 and L_2 are the linear displacements of each prismatic joint.

capital + comma from [3] the following simplified serial kinematic for the Opthalmic Surgical Robot was used as a reference:

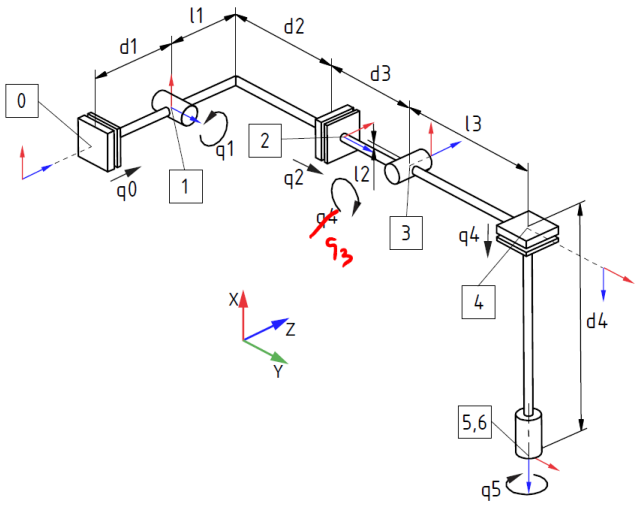

Fig. 13: Simplified serial kinematics for the Opthalmic Surgical Robot

space not sure if this is the beginning of a new sentence (then capital) or if there is a part missing?

for which the Denavit-Hartenberg parameters were specified [3]:

Link	ϕ_i	d_i	a_i	α_i
1	0	$d_1 + q_0$	0	$-\frac{\pi}{2}$
2	$q_1 - \frac{\pi}{2}$	d_2	l_1	0
3	$\frac{\pi}{2}$	$d_3 + q_2$	l_2	$\frac{\pi}{2}$
4	$q_3 + \frac{\pi}{2}$	0	l_3	$-\frac{\pi}{2}$
5	0	$d_4 + q_4$	0	0
6	q_5	0	0	0

Offsets	mm
l_1	32.5
l_2	1.33
l_3	40.5
d_1	22
d_2	20
d_3	22
d_4	55.35

small

Fig. 14: DH parameters for Simplified serial kinematics for the Opthalmic Surgical Robot

small small small? + comma

In Order to control the Robots end-effector in Task Space both its forward and inverse kinematics were derived from [3]

Forward Kinematics:

$$T_0^6 = \begin{bmatrix} -s_1s_5 - c_1c_5s_3 & c_1s_3s_5 - c_5s_1 & c_1c_3 & l_2c_1 + l_1s_1 - c_1c_3(d_4+q_4) - l_3c_1s_3 \\ c_3c_5 & -c_3s_5 & -s_3 & d_2 + d_3 + q_2 - s_3(d_4+q_4) + l_3c_3 \\ c_5s_1s_3 - c_1s_5 & -c_1c_5 - s_1s_3s_5 & c_3s_1 & d_1 + q_0 + l_1c_1 - l_2s_1 + c_3s_1(d_4+q_4) + l_3s_1s_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2)$$

where $s_i := sin(q_i)$, $c_i := cos(q_i)$

Inverse Kinematics:

$$\begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{pmatrix} = \begin{pmatrix} z - d_1 - l_1 c_1 + l_2 s_1 - c_3 s_1 (d_4 + q_4) - l_3 s_1 s_3 \\ atan2(c_3 s_1, c_3 c_1) \\ y - d_3 - d_2 + s_3 (d_4 + q_4) - l_3 c_3 \\ atan2(s_3, c_3) \\ -(x - l_2 c_1 - l_1 s_1 + d_4 c_1 c_3 + l_3 c_1 s_3)/(c_1 c_3) \\ atan2(c_3 s_5, c_3 c_5) \end{pmatrix} \tag{3}$$

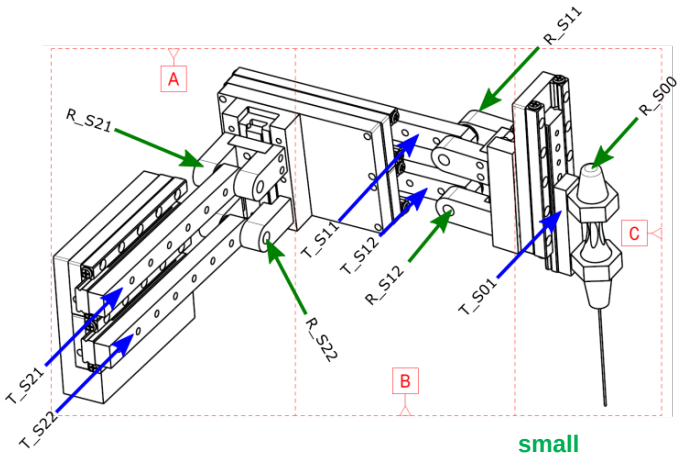

Fig. 15: the Opthalmic Surgical Robot V-REP Structure definition

Conversion from Simple kinematics and PCJM: On Fig. 15 the V-REP Joint structure for the Opthalmic Surgical Robot can be seen, where the prefix T_ corresponds to a translational joint and the prefix R_ to a rotational joint.

Using (1) it is possible to convert a PCJM into a Translational joint and a Rotational Joint. The Relationship between Fig.15 and Fig.14 then becomes:

TABLE I: Conversion between simple kinematics and the V-REP kinematic structure

Joint	V-REP Joint
q_o	T_S21
q_1	R_S21
q_2	T_S11
q_3	R_S11
q_4	T_S01
q_5	R_S00

Since all Rotational joints are purely passive they need to be actuated using Formula(1).

IV. Simulink Architecture (Phillip)

Given that it was our task to control the robot using matlab, simulink was deemed to be the perfect environment to create an architecture including a GUI for this project. In this section we will briefly describe the major parts of the final GUI, as well as further explain why we chose to implement it in this particular fashion.

Figure 16 shows the top level of the built architecture. It consists of four subsystems, each responsible for different tasks:

- PathPlanner: Used to Choose Designated Shape & Output its Datapoints
- ManualIncrementalControl: Manual Generation of Delta's to be Followed by the Robot
- GetDelta: Used to Calculate Delta's from Datapoints
- Controller: Control Robot According to Delta

Generally, the flow of information works as follows: At the start, one sets the Manual Switch just before the Controller block to either the data coming from the PathPlanner (via GetDelta and possibly with an added offset) block or the ManualIncrementalControl block. When the Controller is done executing the desired position change, it will set its output 'request_next_sample' to True, in order to indicate it is finished and now awaits the next position change. This output is then caught on by the PathPlanner block, which will output the next sample to the controller if the switch isn't set to manual control. Otherwise, one has to manually choose when to send the next signal with the ManualIncrementalControl block. The subsequent sections will further explain each of the four previously mentioned building blocks.

A. PathPlanner

Figure 18 shows the PathPlanner subsystem. Its core is a Multiport Switch block displayed on the far right, thanks to which the user may choose a shape to be drawn by the robot. Choosing is done by altering the value of the "Control" input of the switch to the number corresponding to the shape one wishes to draw. The switch has a one based-index, so 1 corresponds to the square, 2 to the heart, and 3 to the circle shape. The user may also choose to enter a 4 as a constant, enabling him to create a shape by himself. Right next to the switch, there is a Gain block, which is set to 0.01, because the robot is supposed to be (and also only capable of) drawing small shapes.

The inputs to the switch are located directly left of it. Each of the three subsystems outputs the data points for one of the three specified shapes as (x,y,z)-triplets. The subsystems are all comprised of three "Triggered Signal From Workspace" blocks, loading the X, Y, and Z points for the selected shape, and a Mux block fusing the three together to a single point (see how the data points are generated in section five).

The triggered version of the "From Workspace" block was

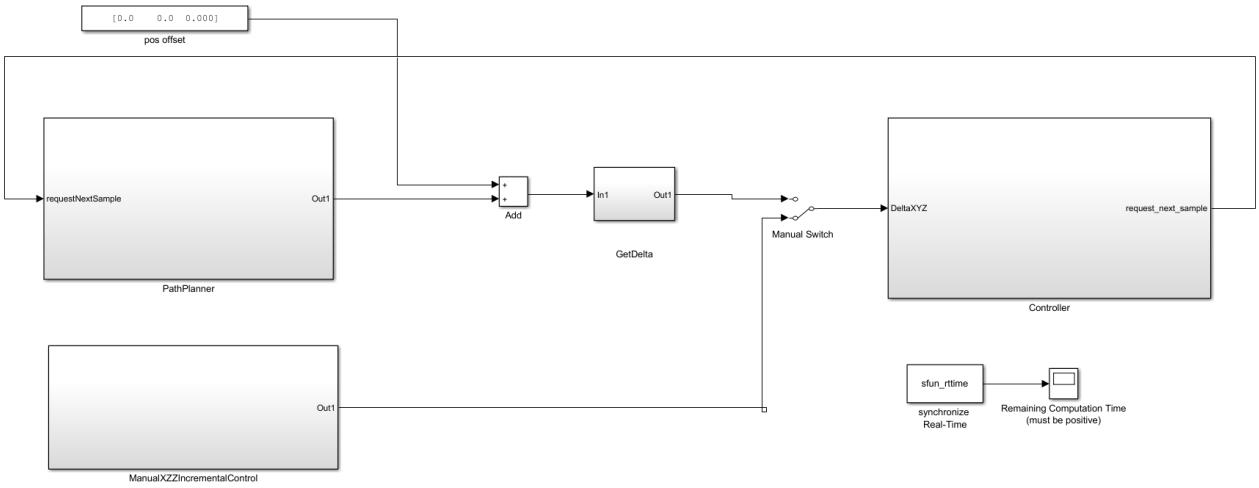

Fig. 16: Simulink Architecture for Robot Control

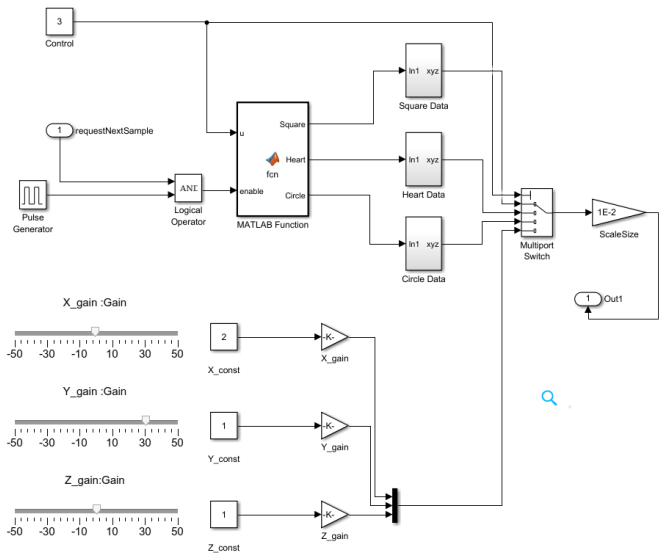

Fig. 17: PathPlanner subsystem

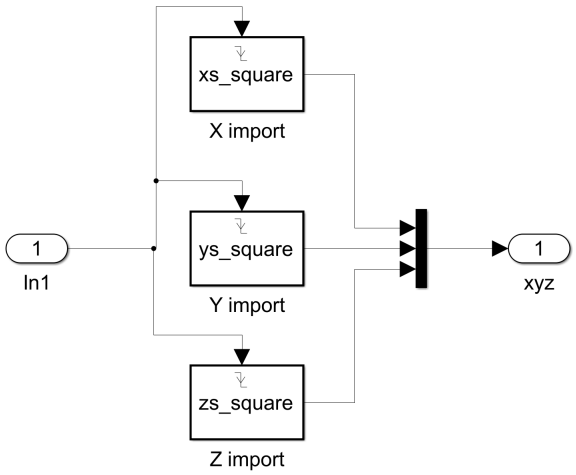

Fig. 18: Example of a Data Loading Block

comma

used in order to be able to choose when to start drawing the designated shape. On the left of the three subsystems for the shapes, one therefore finds a construct consisting of a constant, a pulse generator, a Logic Operator block which is set to simulate the AND function, and a Matlab Function block. The pulse generator is only used to supply a rising edge. Its outgoing signal is then input to the AND block, with the other input being the subsystem's input labeled "RequestNextSample" . This results in the AND block only passing on the pulse if the input is set to True, so only when the controller says it is ready for another sample. The output of the AND block is then fed in to the Matlab Function block together with the constant for choosing which shape to draw. The function then outputs three signals, which are either all set to 0 (if the AND block passed on a 0) or the one associated with the choosing constant is set to 1 (if the AND block passed on a 1).

As previously mentioned, the user has a fourth choice when deciding for a shape, which is doing it completely manually. This option is implemented using three sliders, constants, and gains, as shown in the lower half of the subsystem. All three constants are set to 1 and are input to one of the three Gain blocks, which multiply their inputs by a number determined by the three corresponding sliders on the left. Finally, the outputs of the three Gain blocks are again multiplexed together to a single triplet.

B. ManualIncrementalControl

The ManualIncrementalControl subsystem contains three constants, as well as two manual switches for each coordinate axis. Using these, the user can manually construct a delta to be sent to the robot, which was mainly used for testing purposes. The three deltas are in the end multiplexed together and directly sent to the controller (if the manual switch in the global architecture permits it).

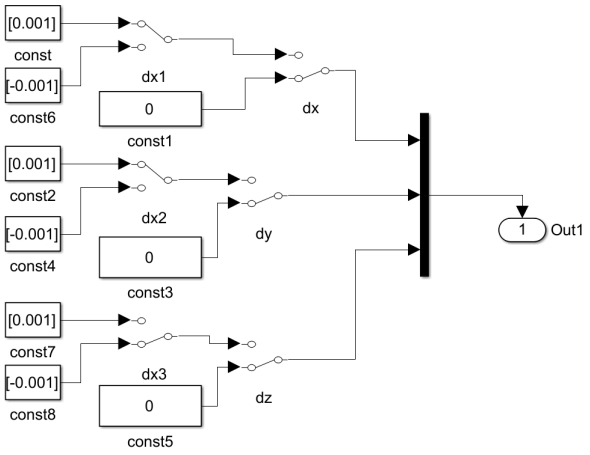

Fig. 19: ManualIncrementalControl Subsystem

C. GetDelta

In the middle of the subsystem, one can find a subsystem called GetDelta. This subsystem is comprised of a Demux block, splitting it into the separate X, Y, and Z parts of the points, and three Difference blocks, which calculate the difference between the current and the previous value (or simply put: they calculate the change / delta relative to the last point). The final output of this subsystem is again multiplexed together and passed on.

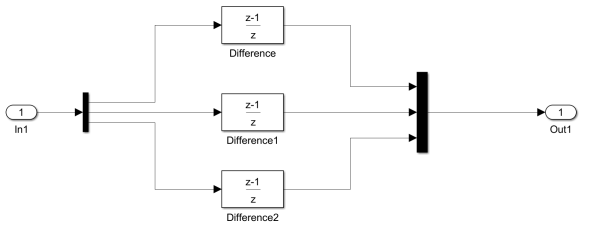

Fig. 20: GetDelta Subsystem

D. Controller

Figure 21 shows the inside of the Controller subsystem which will be detailed in section VII. One thing to note is the calculation of the error between the desired position and the current position. This error is used for generating the 'request_next_sample'signal in order to solve timing issues that would send the next motion command to V-Rep even though the robot has not reached the desired position. When the desired position is reached, the error function outputs True in order to request a new sample from the PathPlanning subsystem.

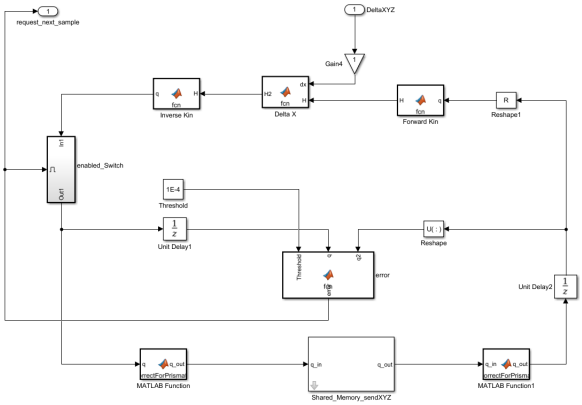

Fig. 21: Controller Subsystem

V. Creation of Shapes (Timo)

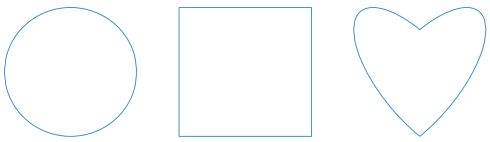

Fig. 22: Circle, square and heart shape

This section describes how different shapes were created to test the simulation of the robot. The idea is that the robot draws those shapes with the needle while moving the needle only inside the eye. For testing the simulation three different shapes (a square, a circle and a heart (see Figure 22)) which should be drawn by the robot were created.
Each shape is centered in the coordinate origin (0,0,0) and consists of many single points represented by coordinates in a x-y-z-coordinate system. To compute the single coordinates each shape is described by a two-dimensional function.

$$\text{circle: } f(x) = \sqrt{r^2 - x^2}$$

$$\text{heart: } f(x) = |x| + \sqrt{1 - x^2}$$

Those functions are evaluated in four steps. In the first step, all points from the top right quarter are calculated, in the second all points from the bottom right, in the third from the bottom left and in the last from the top left quarter. While moving with a given step size over the x range the values are inserted in the function to get the complete

x-y-coordinates. The resulting x-y-coordinates are extended by a constant z-coordinate (0), because the robot draws a two-dimensional shape in a three-dimensional coordinate system, so the z-coordinate is irrelevant for the coordinates. The final points are saved in a text file with the name of the shape.

remove

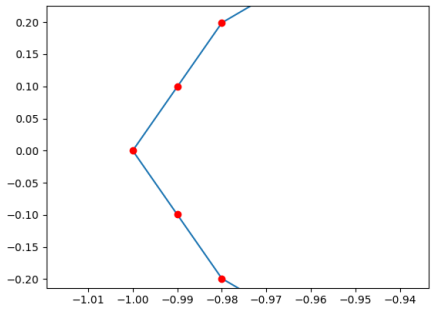

Fig. 23: Left border of the circle shape

In a following step, the created points are optimized for a better movement of the robot. The robot had problems with shapes if there is a rotation in every single point (like for the circle). To avoid point sequences where each point implies a rotation, we add one point between every two points. This new point is located on the straight line between the neighboring points (see fig 23). As a result, the circle for example consists of many very small lines instead of only points. The resulting points are saved back in the text file. To use the points at runtime for the robot, a MATLAB script (importdata.m) is responsible for loading all the text files, reading the coordinate values of the points and making them accessible in the general workspace for the Simulink GUI.

true in final version?

VI. Entering the eye (Matthias)

Since the robot's starting position is outside the eye, the first order of business is entering it. While this could be achieved using the same kinematics algorithm we used for the rest of the movement, the simple nature of the robot allows for a different, less complex approach.

The solution to the entry-problem are the following steps:

1) Adjust the angle of the needle to be equal to the line through the two entry points
2) While keeping the angle, move the needle in the correct position for entry
3) Extend the needle to enter the eye

Fortunately, these tasks are made significantly easier by V-reps functionality to automatically compute all coordinates with respect to a chosen reference frame, instead of their absolute value. As a result, by choosing the right reference frame at the base of the robot - in this case the part imported_sub_38 - we can simplify the computation, since now every joint of the robot moves parallel to one axis of the reference frame. Thus, when searching for the right positions for the joints, every pair of joints is responsible for the translation along and the rotation around one axis. The way the coordinate system for the chosen base is oriented means that the first pair of joints (the one closer to the base, namely q0 and q1) moves only in direction of the z axis and can facilitate a rotation around the x axis. The second pair of joints (q2 and q3) will always move along the x axis of the base, though the axis of rotation will change corresponding to the rotation caused by the first pair. While this approach would also be possible without the aforementioned V-rep functionality, one would have to manually translate every coordinate with respect to the chosen reference frame.

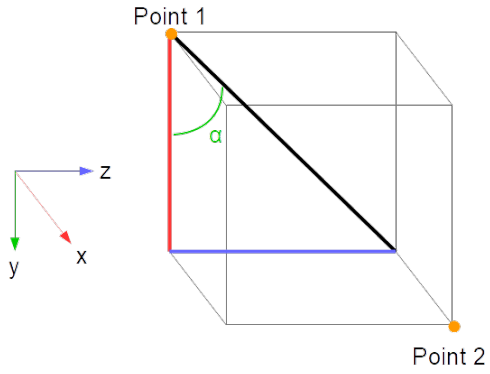

Fig. 24: Angle of first pair

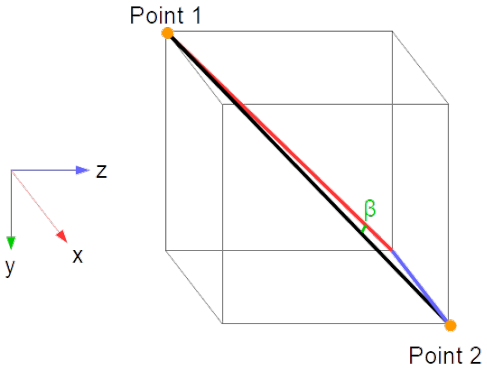

Fig. 25: Angle of second pair

To get the correct angle for the entry, one possible way would be to extract it from the trocar. However, the more precise way is to compute the angle of the line through the two entry points.

Let $(x_1, y_1, z1)$ be the coordinates of one entry point and $(x_2, y_2, z2)$ the coordinates of the other, with both being in the coordinate frame of the base of the robot.
Then the target angle ϕ_1 for the first pair of joints will be

$$\phi_1 = arctan(\frac{z_2 - z_1}{y_2 - y_1})$$

, while the target angle ϕ_2 for the second pair will be

$$\phi_2 = arctan(\frac{x_2 - x_1}{\sqrt{(y_2 - y_1)^2 + (z_2 - z_1)^2}})$$

At any time, the angle θ of the needle around any one axis of the base of the robot is given as

$$\theta = arctan(\frac{L_2 - L_1}{17mm})$$

As a result, the required distance between the joints of each pair is

$$d_1 = \frac{z_2 - z_1}{y_2 - y_1}$$

and

$$d_2 = \frac{x_2 - x_1}{\sqrt{(y_2 - y_1)^2 + (z_2 - z_1)^2}}$$

respectively. Now we can move the joints to the appropriate distance. One thing to note here is that in V-rep, the intrinsic joint position at the start of the simulation will be 0, even if the joints are not in the same position. Hence we first have to check the actual current positions and adjust the distance by the computed target distance plus the current distance.

Now that we have the correct orientation of the needle, we only need to do translations, hence we can treat each pair of joints as a single joint and move both joints by the same amounts. Before computing the correct position for the needle, we extend it as far as we can without colliding with the trocar. This way we reduce errors caused by a slightly incorrect angle - which could happen due to rounding. Afterwards, we can compute the target point as the intersection of the line through the entry points and the plane through the needle perpendicular to the y-axis of the base (since this is the direction in which neither pair of joints can move). In other words, we need to solve

$$p = e_1 + t \cdot (e_2 - e_1)$$

where p is the target point and e_1 and e_2 are the entry points. Since we know the y value of p, this is easily solved. We then merely need to move both joints of the first pair by the z value and both joints of the second pair by the x value to get the needle to the desired point.

Finally, all we need to do is extend the needle to enter the eye through the trocar. Now that we are inside the eye and centered through the entry points, our kinematics algorithms can be applied.

multiple?

VII. Different Algorithm for Robot control (Bogdan)

This section provides more explanations on the methods used in order to control the robot motions. Multiple control schemes were experimented with and had to be adapted in order to compensate for the limitations of communicating with the simulated robot in V-Rep.

The first Controller was presented in Figure 21 and it small produces the control signal based on delta values computed by the GetDelta block. This type of control is useful when trying to execute predefined trajectories around a new center point or when using manual control, where a surgeon might clearly see how much the needle needs to be moved and in what direction, but he remove lacks the knowledge of the world coordinates for the desired point. These delta values are offsets in Cartesian coordinates that are desired relative to the current position of the robot needle.

The control loop begins at the Shared Memory block, where the current position of the robot in joint space is read. These coordinates are transformed in the next block in order to correct for the errors in the simulated model. The coordinates for the virtual rotational joint are also computed in the same block ensuring that the output signal contains comma the correct coordinates that are expected by the kinematics capital (you capitalized all previous blocks) blocks.

The loop contains a few Delay blocks meant to solve any ambiguous execution order errors and also contains a few Reshape blocks that transform multidimensional signals so that they are consistent with the inputs of the defined Matlab functions.

The desired control commands can be computed with the proposed inverse kinematic control scheme by using the desired offset, the joint coordinated and kinematic transformations.

comma

The current Cartesian coordinates of the robot end effector small can be computed using the forward kinematics. By adding the desired delta to the current Cartesian small coordinates the goal position is obtained, but it must be transformed back into Joint small space in order to be sent to the simulator. The inverse kinematics block transforms the coordinates into the desired joint position and can be used because of the simple robot model which does not include singularities.

The next block on the loop ensures that a new desired position is not used or sent forward until the robot reaches the last position that was sent to V-Rep. The initial conditions of the signals enables the communication for the first command and afterwards this block will ensure the synchronization between the controller and the simulator.

The simulator sensors are not ideal so the desired point comma is considered to be reached when the error is bellow a designated threshold.

The goal position must be transformed once more to reverse the effects of the initial transformation by transforms transfor ming? the coordinates of the virtual rotational joints back into coordinates for the translational joints of the robot and also adjusts for the errors on the simulated robot model. The

in / of

position is then sent to V-Rep via the Shared Memory block generating the new motion and closing the loop.

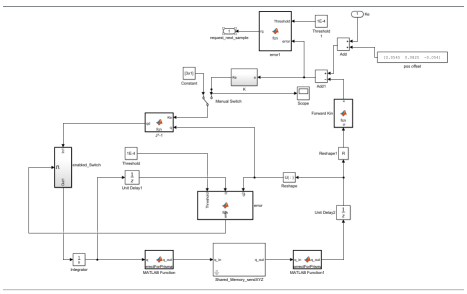

Fig. 26: Controller Subsystem 2

The second approach for controlling the robot is the Inverse Jacobian approach, which uses a desired position in the world's Cartesian coordinates as an input and based on the error between the current and the desired positions generates a desired velocity for the robot.

The desired velocity can be transformed into joint coordinates with the help of the Jacobian Matrix.

The main advantage of using this Jacobian approach is that system can realize speed control, but also if the Jacobian has any redundancies, the remaining degrees of freedom can be used to respect constraints or to execute nullspace motions.

The simulator does not have the capability to control the speed of the motion, so the computed desired speed must be integrated, generating a desired joint position. Based on the weight of the error the system can be critically damped, overdamped or can generate overshooting. In any case however, the system will need to generate a steps until it reaches the desired position. The steps are larger at the beginning because of the larger error, but will become smaller, stabilizing the motion, the closer the robot gets to the goal position.

As mentioned before this approach benefits speed control and since the internal position controller of the simulator does not cause any overshooting problems, generating more steps only slows the simulation down through synchronization delays.

The reason for using the Jacobian approach is motivated by the possibility of extending the Jacobian matrix in order for the motion to respect constrains.

In eye surgery is it critical that the robot does not move in any direction that would cause damage around the entry point on the eye.

VIII. Conclusions

Conclusions on the quality of motions.

Abbreviations and Acronyms

Define abbreviations and acronyms - not sure if we need this?

APPENDIX

Appendixes should appear before the acknowledgment - keeping this section for god knows what...

ACKNOWLEDGMENT

Same...

I put the Master's Thesis in the bibliography, wasn't sure whether we needed anything else - maybe the Algorithm (that I think) Miguel worked on.

References

[1] Coppelia Robotics - VREP Remote API Functions (Matlab), July 2017 http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsMatlab.htm

[2] Coppelia Robotics - Remote API Constants, July 2017 http://www.coppeliarobotics.com/helpFiles/en/remoteApiConstants.htm

[3] P. Gschirr, Control and Simulation of a Robotic Setup for Assisting Ophthalmic Surgery, Master Thesis at the Computer Science Department of the Technical University of Munich. Munich: 2014

[4] Simulation and Model-Based Design, July 2017 https://nl.mathworks.com/products/simulink.html

[5] Simulink and V-REP using Shared Memory, July 2017 http://www.forum.coppeliarobotics.com/viewtopic.php?f=9

[6] Simulink and V-REP Shared Memory Source, July 2017 http://www.forum.coppeliarobotics.com/viewtopic.php?f=9

[7] Simulink Add Libraries to the Library Browser, July 2017 https://de.mathworks.com/help/simulink/ug/adding-librar

[8] Simulink Block for Real Time Execution, July 2017 https://nl.mathworks.com/matlabcentral/fileexchange/309