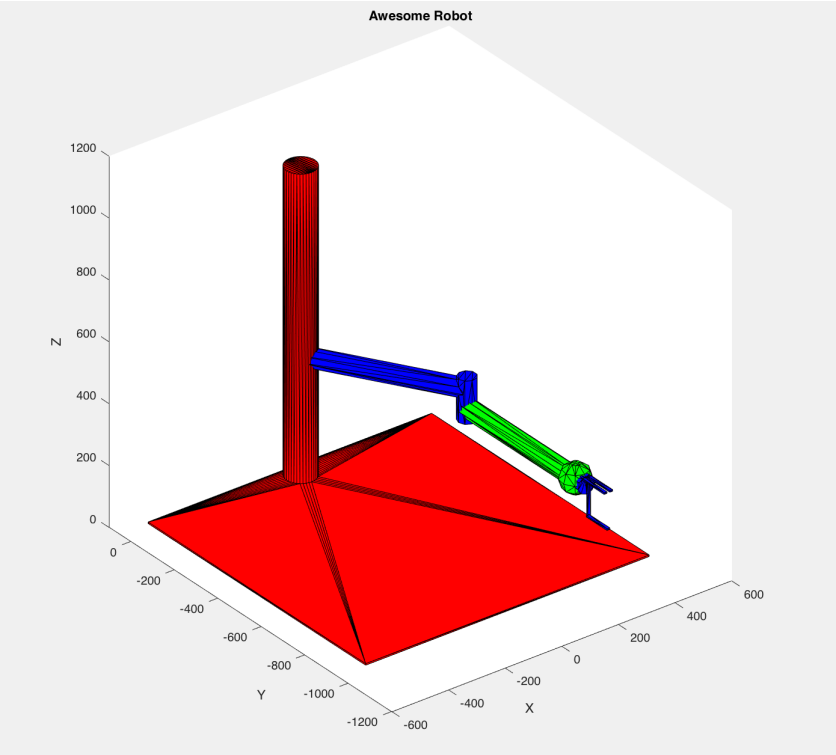



# Robotic Arm Simulation

## Simulation of a robot arm

### Object oriented geometry

This project is about creating a simulation of a robotic arm created with a CAD software and simulated with matlab/octave. Goal is that the arm can grab a cylinder on the floor and re-position it.


Studiengang: Micro- and Medicaltechnology

Autoren: Daniel Tschupp, Lukas Studer


Datum: 06. 01. 2016

# Inhaltsverzeichnis

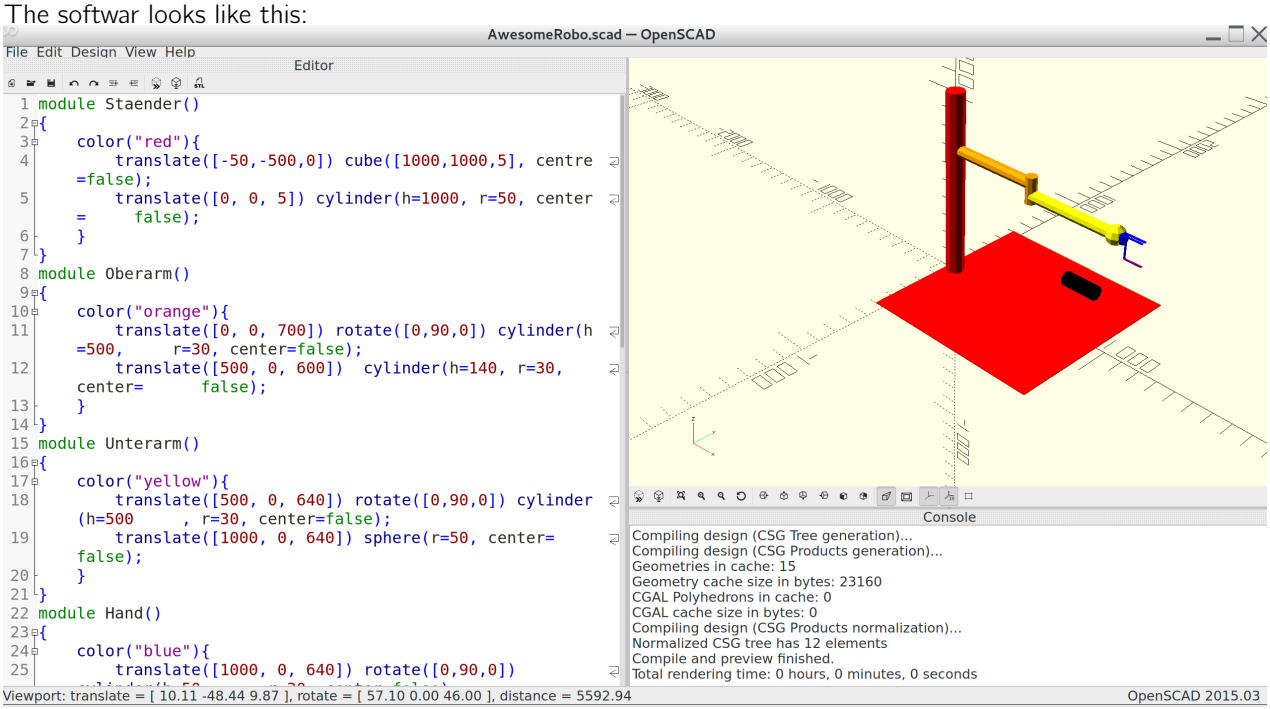
# 1 Introduction

Goal of the project is to simulate an robotic arm. The robotic arm is drawn in a CAD program called OpenSCAD. This is a sript based CAD program. All the different parts of the robotic arm are exported as a .stl file and then read by the matlab/octave software. That gives us matrizes with pointobjects of all the parts of the arm which can be easily calculated with.
The whole project includes five different big parts. First is to draw the robotic arm wish the OpenSCAD software and export the stl files. Second is to include the .stl files into matlab/octave. The third part is to write the code to move all the parts of the robotic arm. The fourth is to dynamically calculate the path to get from point A to point B. And the last part is to test the whole simulation.

# 2 Creating Robotic Arm

## 2.1 OpenSCAD

OpenSCAD is an open source sriped based CAD program with which we decided to create our model of the robotic arm.
The softwar looks like this:




We collored all the different parts of the robotic arm in different collors for a better overview. All this parts are discribed in the following chapters.
As can be seen in the picture with OpenSCAD you can create modules seperately and place them where you want them. So you can add additional modules easily.

## 2.2 Description of the Robotic Arm

Our robotic arm is built up of 5 pieces. The pillar, the upper arm, the forearm, an hand with two fingers and a separate finger to squeeze. Each of this part has their own movement possibilities:
- pillar: no movement
- upper arm: just z movement
- forearm: just rotate around z-achsis
- hand: rotate around all three achsises
- finger: just z movement (but it depends on the posision of the hand)

## 2.3 Exporting .stl-files

Creating an .stl-file is a very easy task with the OpenSCAD software and the way we built our robotic arm. Just draw and render the modules separately. Then click File-¿Export-¿Export STL... Give it a name and click Save and that's it.

# 3 Import .stl-files into Matlab/Octave

The .stl-files created by the OpenSCAD software have to be imported into Matlab/Octave. We found a function written by Don Riley on the mathworks forum which did exactly what we needed. You can give the path to your .stl-file as a parameter and it returns matrix with homogeneous coordinates of every point in the .stl-file. This matlab/octave function is also included in this project and it's name is: loadCAD.m.

# 4 Movement of the Robotic Arm

## 4.1 Comparison of methods

There are different ways to perform the mathematical Operations to move a pointobject in a coordinate system. And that's basically what we do here. We have matrices with lots of points in it which built up the modules of our robotic arm and the arm itself.
Four of the possible ways to handle it are explained below together with their advantages and disadvatages.

### 4.1.1 3d Rotation Matrizes and separate Translations

With this method we have three 3x3 matrices to perform rotational movements. Translational movements needs to be taken care of separately.

Advantages:
- small matrices which doesen't need a lot of cpu power.
- easy to use matrices.

Disadvantages:
- Not very clearly arranged
- If there are lots alternating rotations and translations it consumes a lot cpu power to calculate all the matrices for all the points.

### 4.1.2 Principe of Homogen Matrizes

This method is very similar to the first method. The difference is that you have three 4x4 matrices. This additional dimension allows you to calculate the translation movement the same way as the rotational movement which makes it a lot more intuitive and much faster if you have alternating translation and rotation movements.

Advantages:
- intuitive
- fast when calculation many movements for one point

Disadvantages:
- 4x4 matrices which is more calculating work for the computer

### 4.1.3 Rodrigues rotation formula

This is a very easy to use mathematical formula to perform rotations around an allocatable axis. But it only allows rotation movements.

Advantages:
- Very fast
- Comprehensive for rotations

Disatvantages:
- Not intuitive because Translational movements must be treated separately

### 4.1.4 Quaternions

Quaternions are an extention of the Gauss number plane created by Sir William Rowan Hamilton. It to allows to perform rotational movements too. But no translations.

Advantages:
- very fast operations

Disadvantages:
- Not intuitive because translations have to be treated seperately.
- Quaternions are a pain to calculate
- Not supported on matlab, just octave

### 4.1.5 Decision

We decided to build our simulation on the second method (Homogen Matrices) because it's the most intuitive one.

## 4.2 Movement of Arm

In general one movement persists of at least 5 part movements:
- move the object to the world coordinate origin
- turn the object to the world coordinate direction
- make the movement you want to
- turn the object back to the coordinate direction they where befor turning them to the world coordinate directon
- move the object to it's original position

Sometimes we could shorten this general sequence. How we could shorten it will be explained in each case.

Here's needed to be said, that we never made any sanity checks for movements.

The following matrices are the one everything builts on:

Homogeneous Translation Matrix:

$$tl(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Rotation Matrix rotation around x-axis:

$$rx(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Rotation Matrix rotation around y-axis:

$$ry(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Rotation Matrix rotation around z-axis:

$$rz(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### 4.2.1 Movement of upper arm

The upper arm movement is relatively easy because we defined that it's only movement possibilities is up and down. So we just need the homogeneous translation matrix and set $t_x = 0$ and $t_y = 0$ $t_z$ =relative movement.

### 4.2.2 Movement of forearm

The forearm is a bit more complex, but still not to complex. It's origin isn't the origin of the world coordinate system, so fist step is to move it to the world coordinate system origin.
The only movement the forearm can perform is a rotation around the z-axis because of the way it's built up. So next step is to rotate it and after that we can move it back to the starting position.

$Rot_{matrix} = T(x, y, z) * Rz(\theta) * T(-x, -y, -z)$

### 4.2.3 Movement of hand

The hand is a general movement as described on top of the chapter.
We name the matrix which turns the roboter coordinate system to the world coordinate system: $rot_{sys}$ (How this matrix is created you can see in the section: Matching coordinate systems)
And we name the relativ rotation matrix: $rot_{rel}$ (This is the movment we acually want to perform.)
$Rot_{matrix} = T(x, y, z) * inv(rot_{sys}) * rot_{rel} * rot_{sys} * T(-x, -y, -z)$
We've written three function with which we can independently rotate around x-axis, y-axis or z-axis. When using them, the user has to make sure to not fall into a Gimbal-Lock(see section Gimbal-Lock). There are two additional function to perform combined rotations around xyz-axises. One of it with the Gimbal-Lock z-axis direction(turning sequence = zxy), the other with the Gimbal Lock in x-axis direction (turning sequence = xyz).

### 4.2.4 Movement of finger

The finger movement is very similar to the hand movement, though a bit easier because there is no rotation movement to perform, just a translation movement. This resolves the whole Gimbal-Lock problem. What remains is, that we have to match the coordinate systems because the movement of the hand affects the finger.
So the movement matrix looks like:
$Move_{matrix} = T(x, y, z) * inv(rot_{sys}) * T_{0,0,z} * rot_{sys} * T(-x, -y, -z)$

### 4.2.5 Matching coordinate systems

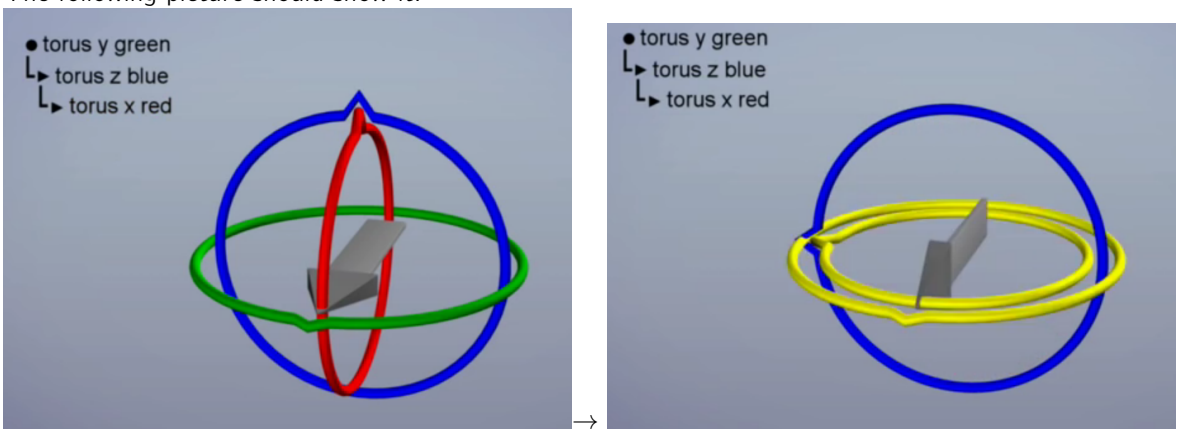
To match two coordinate systems you have to move first the two origins together. Then you've to find the rotation matrix which turns one into the other. That's why we decided that every part of the robotic arm has it's own coordinate system together with a vector which points from the origin of the world coordinate system to the origin of the roboter coordinate system, which is turned and moved together with the corresponding part. This allows us to match the origin easily with just one translation move. The adjusting of the coordinate systems is a bit more tricky. But we know the following thing:
Definitions:
$e_x$: unit vector in world coordinate x-axis direction
$e_y$: unit vector in world coordinate y-axis direction
$e_z$: unit vector in world coordinate z-axis direction
$e'_x$: unit vector in roboter coordinate x-axis direction
$e'_x$: unit vector in roboter coordinate y-axis direction
$e'_x$: unit vector in roboter coordinate z-axis direction
We know one rotation matrix M must comply with all three rotations. so we get the following three equations:

$$
\begin{aligned}
M * \vec{e_x} &= \vec{e'_x} \\
M * \vec{e_y} &= \vec{e'_y} \\
M * \vec{e_z} &= \vec{e'_z} \\
\rightarrow \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} ex_x \\ ex_y \\ ex_z \end{bmatrix} &= \begin{bmatrix} ex'_x \\ ex'_y \\ ex'_z \end{bmatrix} \\
\rightarrow \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} ey_x \\ ey_y \\ ey_z \end{bmatrix} &= \begin{bmatrix} ey'_x \\ ey'_y \\ ey'_z \end{bmatrix} \\
\rightarrow \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} ez_x \\ ez_y \\ ez_z \end{bmatrix} &= \begin{bmatrix} ez'_x \\ ez'_y \\ ez'_z \end{bmatrix}
\end{aligned}
$$

These are 9 unknown variables and 9 linear equations which can be solved easily with matlab/octave.

### 4.2.6 Gimbal-Lock

Gimbal-Lock is a problem which occurs when rotating an object with euler angles. It occurs because you always have to decide for an order how to rotate the object. For example first rotate around x-axis, then y-axis then z-axis(xyz-order). In this case Gimbal-Lock would occur when the y-axis is 90° to the x-axis and z-axis. It's always the middle axis which is the problem, because it can move the other both in one plane if the first axis isn't moved. The following picture should show it:




*(Pictures out of the following you tube video: https://www.youtube.com/watch?v=zc8b2Jo7mno)*

### 4.2.7 Calculation of rotation angles

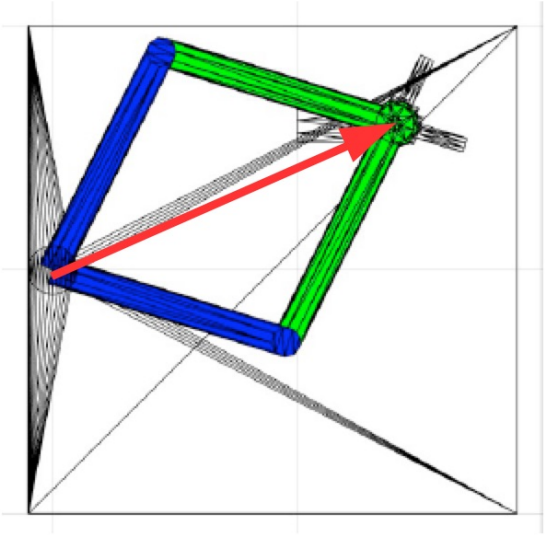
In order to get the arm to move to a certain position, we have to calculate the required angles of rotation of each segment of the arm. We do this with a vector based approach which leads to following system of equations. Alternative approaches would have involved *the law of cosins* or *complex numbers.* We used vectors due to their simplicity. Following equations are the basis of our calculations:

$$\begin{aligned} \vec{a} + \vec{b} &= \vec{x} \\ |\vec{a}| &= A \\ |\vec{b}| &= B \end{aligned}$$

$\vec{a}$ being the vector representing the upper arm, $\vec{b}$ representing the fore arm, and $\vec{x}$ being the position vector of the destination. $A$ and $B$ are the lengths of each segment. Using the components of the vectors we get following equations:

$$\begin{aligned} a_x + b_x &= x_x \\ a_y + b_y &= x_y \\ \sqrt{a_x^2 + a_y^2} &= A \\ \sqrt{b_x^2 + b_y^2} &= B \end{aligned}$$

Solving the system for $a_x$, $a_y$, $b_x$, and $b_y$ gives us two sets of components for $\vec{a}$ and $\vec{b}$. The second solution is the same as the first but mirrored along the position vector of the destination. We can therefor just pick a solution.



*Showing the the solutions for moving the robot to a given position*

Calculating the angles of the vectors and considering the rotation of the first segment for the second one, give us the rotation angle of each segment. Subtracting the current angles from the result will give us the relative rotation angle to get from one position to another.

$$\alpha = \arctan(\frac{a_y}{a_x})$$

$$\beta = \arctan(\frac{b_y}{b_x}) - alpha$$

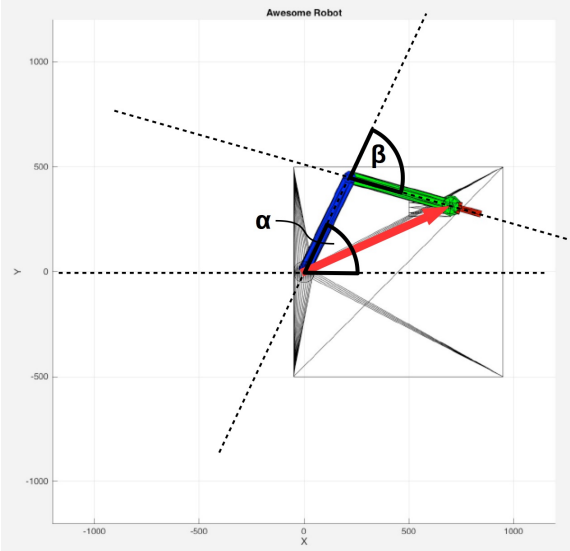


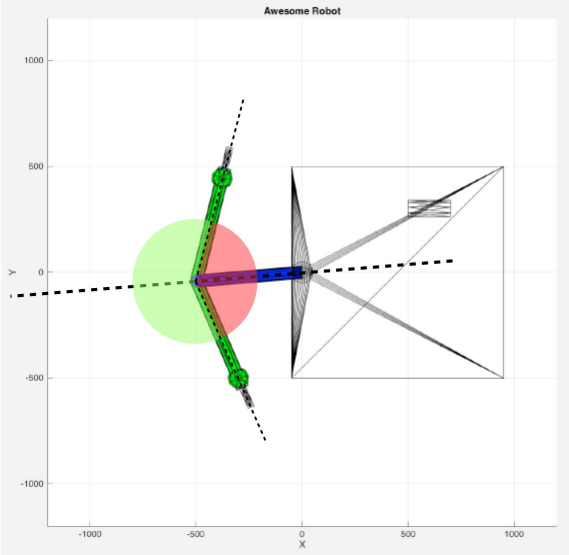

*Illustration of the problem*

### 4.2.8 Preventing the arm from passing through itself

A big challange of calculating the rotation angle is to prevent the arm or rather the hand from passing through the pillar. The problem mostly occures when doing more than one rotation. Solving the problem proved to be quite difficult, as we couldn't pedict which of the two solutions we end up with when picking the *first*. To eprevent the phenomenon, we can formulate following simple rules:

*The fore arm must never rotate more than 360 degrees. If the fore arm is left of the upper arm, or has a positive angle, the fore arm should, in general, turn right using a negaive rotation angle and if the fore arm is right of the upper arm, or has a negative angle, the fore arm should, in general, turn left using a positive rotation angle.*

These rules lead to the fact that the shortes way is not always the correct way. Following graph shows why:




*Red shows the bad rotation angle and green the correct rotation angle*

Following code in *MatLab* does the trick:

```
dest_angles = calcArmAngles(dest_pos, robo.upperArm_len, robo.foreArm_len, robo.hand_len);
angles = dest_angles - robo.pos_angles;

% making sure that the arm does not pass through itself
angles = rem(angles, 360);
if robo.pos_angles(2) <= 0
        if angles(2) > 0 & abs(angles(2)) > abs(180 - robo.pos_angles(2))
                angles(2) = angles(2) - 360;
        elseif angles(2) < 0 & abs(angles(2)) > abs(180 + robo.pos_angles(2))
                angles(2) = angles(2) + 360;
        end
elseif robo.pos_angles(2) > 0
        if angles(2) > 0 & abs(angles(2)) > abs(180 - robo.pos_angles(2))
                angles(2) = 360 - angles(2);
        elseif angles(2) < 0 & abs(angles(2)) > abs(180 + robo.pos_angles(2))
                angles(2) = angles(2) + 360;
        end
end
```

At first we calculate the relative rotation angles to get to the destination. We then make sure that we don't turn more than 360 degrees by calculating the remainder *'rem'* from a division with 360. We then basically apply the rule above: if the fore arm is to the left, turn right and if the fore arm is to the right, turn left. But of course, we also allow small rotations to the left if the fore arm is to the left and the other way around. But all under the restriction that the fore arm never turns more than *180 degrees plus his current rotation* or *180 degrees minus his current rotation*, depending on his position and the destination. This prevents the hand from passing through the pillar.