Group Project: Image Guided Robotic Transcranial Magnetic Stimulation

Christopher Göthel, Gunnar Maerker, Fabian Sabljak,
Jan-Philipp Scheel, Aashik Udupi
Institute of Medical Technology at Hamburg University of Technology,
Am Schwarzenberg-Campus 3, 21073 Hamburg, Germany

***Abstract*—This report contains the results and approaches of the group work on the project of "Robotics and Navigation in Medicine". The task was to support transcranial magnetic stimulation (TMS) by a vision-guided robot which should compensate head movement.**

I. Introduction

Transcranial magnetic stimulation is used as a therapeutic and diagnostic technique. A magnetic coil needs to be placed for stimulation of a specific cortex. Therefore, compensation of patient movement is needed. During the group project, we developed a motion compensation system which is based on ROS and used both C++ and Python.

II. Tasks

In this report, we will follow the structure of the project. It was sectioned into four parts: The kinematics of the robot, the calibration of the camera, the head pose estimation and the head movement compensation.

A. Robot Kinematics - AU, GM

In transcranial magnetic simulation procedure, for effective therapy, the motion of the head must be compensated by moving the magnetic coil held by the robot end-effector. Thus, understanding the robot motion is crucial in solving such a task. This is possible by solving the direct kinematics and inverse kinematics of the robot.

1) Direct Kinematics: Direct Kinematics helps us in determining the position of the robot end-effector as a function of joint angles. The Direct Kinematics of the UR-3/UR-5 robot is calculated based on the DH-parameters of the given robot. The program is used to calculate the Transformation matrix ${}^{B}T_{E}$ from the robot base to the end-effector and the solution was validated by comparing it with solution provided by Hawkins [1]. The Direct Kinematics program is also used to convert the transformation matrix into a pose message.

2) Inverse Kinematics: In reality, it is desirable to calculate the joint angles which can place the end effector at a particular desired location. This calculation of joint angles happens through Inverse Kinematics. The joints of the UR-5 can be referred as shoulder pan (θ_1), shoulder lift (θ_2), elbow (θ_3), wrist 1 (θ_4), wrist 2 (θ_5), and wrist 3 (θ_6).

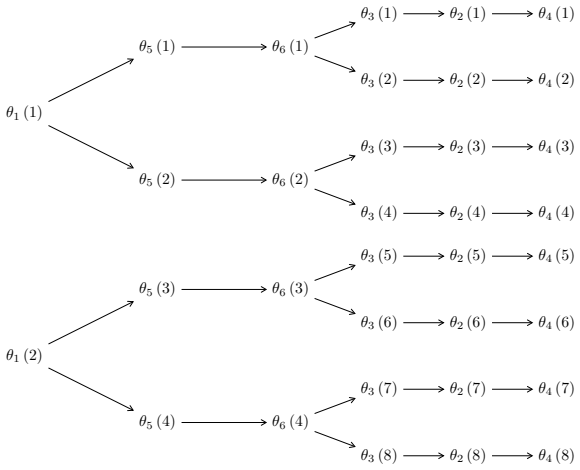

Fig. 1. 8 possible combinations of theta angles.

Due to the high flexibility of the UR robot, a point with a certain orientation can be reached by the robot's end-effector in up to 8 different joint angle combinations i.e. 8 different robot poses to reach a point and orientation in space. Through inverse kinematics, for every shoulder pan angle θ_1, 2 values of θ_5,θ_6 and 4 values of θ_3,θ_2,θ_4 can be calculated. Since there are two possible θ_1 angles, this results in 8 combinations of joint angles for the robot to reach a point in space. The 8 combinations have been graphically represented in Figure 1.

While the robot end effector continually moves to compensate for the movement of the patient, it receives input for the next desired pose from the Kinect camera. At every such instance, the inverse kinematics program calculates all the possible 8 joint angle combinations to reach the desired pose and then chooses the best possible set of joint angles to minimize the overall motion of the robot between the current position and the desired position.

The calculated desired joint angle combinations from inverse kinematics program must be modified before it can be used in further calculations as this could result in errors. This is accomplished by calculating three terms by adding factors 0, 2π, -2π to each desired joint angle. The

three terms are geometrically equivalent, but the robot would have to do one full rotation on this joint to get from one term to the next. Then the difference between the terms and the current robot position for each term is calculated. Afterwards, the factor resulting in the least difference is added to the desired angles. This procedure is repeated for all calculated desired joint angle values.
The best possible joint angles of the 8 possible combinations are chosen by calculating the overall absolute difference between each joint angle combination with the current pose of the robot. This overall absolute difference is calculated as the summation of the absolute values of the difference between each desired and the current joint angle.

$$\text{overall absolute difference for set } i = \sum_{j=1}^{6} \Delta\theta_j(i)$$
$$\Delta\theta_j(i) = |\theta_{j,current} - \theta_j(i)|$$

Then the set of joint angles resulting in the least overall difference is chosen. This results in a more efficient transition of the robot between poses.

3) Implementation in ROS: The overall task was implemented in ROS (Robot Operating System, a convenient middleware to control robots) since this brings many advantages: The overall program consist of multiple sub-programs called nodes, which can be programmed separately in different languages and communicate in a standardized way via ROS messages under certain so-called topics.
The implementation of kinematics was programmed in C++ and consists basically of two nodes, *Direct Kinematics* and *Inverse Kinematics*. There are principally two use cases: The Calibration (as needed for Task 2, marked in red) and the Head Movement Compensation (as needed for Task 4, marked in blue). Both use cases with their corresponding communication are illustrated in Figure 2. Nodes and objects which are needed in both use cases are marked in white. The Direct Kinematics are used within the Calibration to get the current pose from the current joint angles. The Inverse Kinematics are used to calculate the desired joint angles for a desired position, which is given by either the Calibration or the Head Motion Compensation. Furthermore, we added one node called *Move To Desired Angles*, which uses the result of the inverse kinematics and communicates this as a goal to the robot.
The nodes are built with this basic structure: In the main function an object of a class is created and a *start*-function of that object is called. In this function, the ROS publishers and subscribers are created. The other parts of a node happen in so-called *callback functions*. These functions are called whenever a new message is published to the relating topic.
For many nodes, the library *Eigen* (Version 3) [2] was used. Eigen is an elegant and sophisticated solution to use matrices, vectors and quaternions in C++ because it contains corresponding arithmetic operations as well as conversions to other forms (e.g. quaternion to rotation matrix) and to ROS messages.

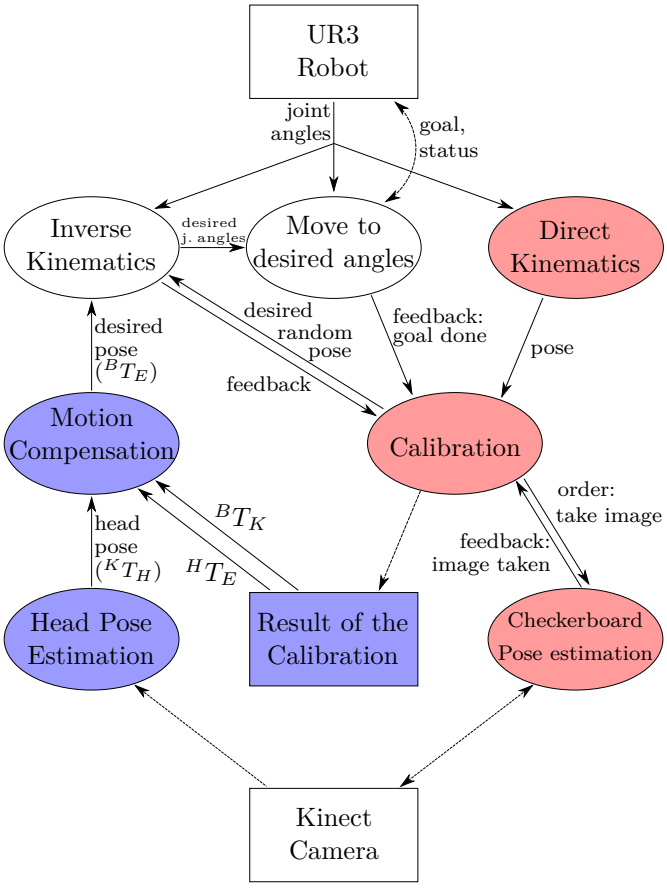

Fig. 2. Node Overview. Nodes are shown in ellipses, other elements in rectangles. Messages are shown with solid lines, other relations with dashed lines.

B. Calibrations - FS

In order to compensate the motion of a patient both the pose of the patient and the coil have to be known at any time. For this reason, a RGBD camera is used that provides a point cloud from where a 3D scene can be reconstructed. Besides, a calibration between the camera reference frame and the robot base is needed to navigate the robot based on images from the camera.

1) Camera Calibration - JS: Before the hand-eye calibration can be done, the camera itself needs to be calibrated. The goal of this calibration is to determine the parameters which are necessary to predict the pixels of the image of the objects in the field of view of the camera. Assuming the camera as a pinhole camera, the problem of calibration can be split up into two problems, the intrinsic and the extrinsic camera matrices.
The camera maps 3D points $M = (X, Y, Z)^T$ to 2D points $m = (u, v)^T$. The vectors are augmented to $\tilde{M} = (X, Y, Z, 1)^T$ and $\tilde{m} = (u, v, 1)^T$. The pinhole approach

leads to the following matrices.

$$\left.\begin{aligned} u &= -\tfrac{fY}{Z} \\ v &= -\tfrac{fX}{Z} \end{aligned}\right\} \quad \Longrightarrow \quad \left\{ \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \right.$$

The intrinsic camera matrix A describes the parameters intrinsic to the camera.

$$A = \begin{pmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{pmatrix}$$

This Matrix A considers the offset of the image coordinates (u_0, v_0), the scaling factors (α, β) and a potential skew of the camera sensor γ.
The extrinsic parameters $(R\ t) = (r_1\ r_2\ r_3\ t)$ are the rotational and translational parameters of the camera in regards to the world coordinate system. With the intrinsic matrix A and the extrinsic parameters *(R, t)* the projection is given by:

$$s\tilde{m} = A \begin{pmatrix} R & t \end{pmatrix} \tilde{M}$$

with s an scaling parameter.
Considering the planar calibration by Zhang [3], making the plane of the marker $Z = 0$ of the world coordinate frame system, this equation can be simplified to:

$$s \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = A \begin{pmatrix} r_1 & r_2 & r_3 & t \end{pmatrix} \begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} = A \begin{pmatrix} r_1 & r_2 & t \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}$$

Mutating the notation of M to $M = (X, Y)^T$ thus $\tilde{M} = (X, Y, 1)^T$ we can formulate the homography H:

$$s\tilde{m} = H\tilde{M} \ with \ H = \begin{pmatrix} r_1 & r_2 & t \end{pmatrix}$$

From an image with the two-dimensional marker, in this case a checkerboard, two constraints of the intrinsic parameters can be derived, as r_1 and r_2 of the homography H are orthonormal.

$$h_2^T A^{-T} A^{-1} h_2 = 0$$
$$h_1^T A^{-T} A^{-1} h_1 = h_2^T A^{-T} A^{-1} h_2$$

As the homography H has 8 degrees of freedom 2 constraints on the intrinsic parameters can be determined by one homography. As there are five parameters in the intrinsic camera matrix A, at least 3 images with different homographies are needed to determine all parameters. After calculating A one is able to calculate all rotational and translational matrices with the used marker. Zhang also implemented a maximum likelihood estimation, minimizing the error between each point and its projection. [3] In this project the in the IAI Kinect2, a toolkit for the use of the Kinect v2 in ROS, included calibration tool was used to calibrate the intrinsic parameters of the RGB camera, the intrinsic parameters of the infrared camera, the extrinsic parameters of both cameras in relation to each other and finally the resulting depth data.[4]

2) Hand-Eye Calibration - FS: The hand-eye calibration is used to get a relationship between the robot's reference frame and reference frame of the tracking device so that a navigation of the coil mounted at the robot's end effector based on images of calculated poses of the head is possible. For this reason we used an approach called *QR24* which makes use of the relation 3:

$$({}^R T_E)_i^E T_M = {}^R T_T ({}^T T_M)_i, \quad i = 1, \ldots, n$$

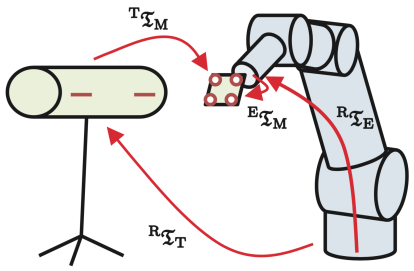

Fig. 3. Robot and optical tracking system [5]

The robot's end effector is moved to random positions within a certain range for n-times $({}^R T_E)_i$. At the end effector a checkerboard of known dimensions is mounted. For each of the the n poses an image is taken from which the pose of the checkerboard in relation to the camera reference frame is calculated $({}^T T_M)_i$. With the collected data the desired transformation matrix ${}^R T_T$ can be calculated using the least-squares approach [5]:

$$\sum_{i=1}^{n} \left| ({}^R T_E)_i^E T_M - {}^R T_T ({}^T T_M)_i \right|^2$$

a) Robot Poses - FS, GM, AU: The robot starts in an initial pose $({}^R T_E)_0$ manually set in which the checkerboard is both parallel and centered with respect to the tracking device. Out of this pose small changes in joint angles are made so that the end effector results in small changes in translation and rotation with respect to the robot base $({}^R T_E)_i$:

$$\begin{aligned} d_i &= d_0 + rand(\pm 10mm) && \text{(translation)} \\ \theta_i &= \theta_0 + rand(\pm 10^\circ) && \text{(angles)} \\ R_i &= R_x(\theta_i) R_y(\theta_i) R_z(\theta_i) && \text{(rotation)} \end{aligned}$$

$$\Rightarrow ({}^R T_E)_i = \begin{pmatrix} R_i(0,0) & R_i(0,1) & R_i(0,2) & x_i \\ R_i(1,0) & R_i(1,1) & R_i(1,2) & y_i \\ R_i(2,0) & R_i(2,1) & R_i(2,2) & z_i \end{pmatrix}$$

Here, $d = \begin{bmatrix} x & y & z \end{bmatrix}^T$ and $\theta = \begin{bmatrix} \theta_x & \theta_y & \theta_z \end{bmatrix}^T$ are vectors containing the translational part and the angles of $({}^R T_E)$, respectively. $R_.(\theta)$ are the 3x3 elementary rotation matrices in x, y and z direction. The function *rand(·)* creates different random values for each of the values in the

vectors d and θ within the given range. Small changes out of the initial position are important to maintain consistency with the tracking device, i.e. it is possible to calculate the pose of the checkerboard out of the images $({}^{T}T_{M})_i$ (see II-B2b). If it is not possible to get the pose of the checkerboard $({}^{T}T_{M})_i$ for a certain configuration $({}^{R}T_{E})_k$ of the robot, a new pose will be generated, the non-consistent pose will be overwritten and the algorithm to get the pose of the checkerboard will be told to try again. After n-iterations a matrix of $4nx4$ consistent transformation matrices has been generated.

b) Marker Positions - JS: The Marker Positions are recorded via a python script. A ROS node is implemented, which waits for the in figure 2 shown order to take images. As soon as the robot reaches its final pose the script tries to detect the used marker and takes numerous pictures of the current pose of the robot. The as the marker used checkerboard is detected via corner detection of the OpenCV library [6]. For the image the already rectified images with the in the camera calibration used distortion coefficients are used for the ease of post processing of the images. The to the topic /kinect2/sd/image_ir_rect and /kinect2/sd/image_depth_rect posted images are used, as the resolution of 512x424 pixel is the maximum output of the Kinects depth stream.[7]

The time synchronized pictures of the image stream are used and saved. After the desired number of pictures were taken, or for a prolonged time no checkerboard was detected, the script reports into a topic that it is done and the robot can move to a new pose and afterward the script tries to take the next series of images.

After the desired number of poses have been run through and the images are saved, the transformation matrices for the tracker to checkerboard matrices $({}^{T}T_{M})_i$ are to be calculated. Therefore again the pixels $u_i,\ v_i$ with the checkerboard corners are detected. To remove the signal noise in the depth image, averaging about the in a single pose taken depth images is done, the depth values d_i are, through the calibration, related to the pixels $u_i,\ v_i$ in the infrared image. The 3D pose of the checkerboard can now be estimated with the use of the intrinsic camera parameters A

$$X_i = \frac{u_i - u_0}{\alpha} * d_i$$
$$Y_i = \frac{v_i - v_0}{\beta} * d_i$$
$$Z_i = d_i$$

Finally the tracker to checkerboard matrices $({}^{T}T_{M})_i$ are calculated via least-square estimation by the Umeyama-Method.[8]

c) QR24-Algorithm - FS: After the corresponding transformation matrices of n configurations of the robot $({}^{R}T_{E}) = M$ and consistently to these poses n poses from the tracking system to the checkerboard $({}^{T}T_{M}) = N$ have been generated, it is possible to find the optimal entries of static transformation matrices from robot base to the tracking device ${}^{R}T_{T} = Y$ and from the end effector to the marker position ${}^{E}T_{M} = X$ in terms of minimizing the quadratic error:

$$\sum_{i=1}^{N} |M_i X - Y N_i|^2$$

These two matrices have both 12 non-trivial elements that can be combined into a vector $w = \left[x_{1,1}, x_{2,1}, \ldots, y_{1,1}, y_{2,1}, \ldots, y_{3,4}\right]^T \in \mathbb{R}^{24}$. Thus, at least 24 poses are needed to estimate the elements of w. The equation (II-B2c) can be combined to a system of linear equations $Aw = b$ to find the optimal solution of w[5].

For minimizing the error and thus getting a sufficient result more poses than 24 are eventually needed. However, there is a chance of overfitting as well, i.e. the estimated matrix has good results with the training data but has poor generalization abilities with new poses. For this reason, the data is split into training and test data, respectively. 80% of the data was used for training and the remaining 20% was used to test whether the error is small enough.

d) Evaluation QR24 - FS: The implementation was initially intended to be in Python which gets the matrices M_i and N_i to build A and b and calculates w from where X and $Y = {}^{R}T_{T}$ can be build. However, when it came to testing at the end of the project the Python solution was inconsistent with another estimation known to be good. From this perspective it was not clear whether the mistake was in the *QR24* algorithm, in the kinematics or in marker pose estimation.

To test the *QR24* algorithm a model of known dimensions and outcomes has been generated via Matlab. The model was constructed to be most similar to the real scenario, i.e. small changes in the robot parameters resulting in different robot configurations $({}^{R}T_{E})_i$ from where the corresponding transformation matrix from the tracking device to the marker position $({}^{T}T_{M})_i = ({}^{R}T_{T})^{-1}\,({}^{R}T_{E})_i\,{}^{E}T_{M}$ can be calculated. This is working since ${}^{R}T_{T}$ and ${}^{E}T_{M}$ are initially known in this model. Nevertheless, out of the generated data $({}^{R}T_{E})_i$ and $({}^{T}T_{M})_i$, $i = 1, \ldots, 1000$ the matrices ${}^{R}T_{T}$ and ${}^{E}T_{M}$ are estimated using the *QR24* approach and compared to the real target values.

Here, similar implementations of the algorithm has been realized in both Python and Matlab. After discovering and correcting a mistake in building the matrix A in the initial Python code the outcomes of the orthonormalized matrices ${}^{R}T_{T}$ where the following:

$${}^{R}T_{T_{\text{target}}} = \begin{bmatrix} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$${}^{R}T_{T_{\text{phyton}}} = \begin{bmatrix} 1 & 2.45{\cdot}10^{-8} & -2.54{\cdot}10^{-7} & 7.99 \\ -2.45{\cdot}10^{-8} & 1 & 3.27{\cdot}10^{-8} & -1.10{\cdot}10^{-6} \\ -2.54{\cdot}10^{-7} & -3.27{\cdot}10^{-8} & 1 & 8.36{\cdot}10^{-6} \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$${}^{R}T_{T_{\text{matlab}}} = \begin{bmatrix} 1 & -1.02{\cdot}10^{-15} & 5.07{\cdot}10^{-16} & 8.00 \\ 1.02{\cdot}10^{-15} & 1 & -3.05{\cdot}10^{-16} & -2.58{\cdot}10^{-15} \\ -5.07{\cdot}10^{-16} & 3.33{\cdot}10^{-16} & 1 & -4.51{\cdot}10^{-15} \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

The residual sum of squares (RRS) is the square sum of the differences between the target values and estimated values of each entry in the matrices which is used to measure the discrepancy between the data and the estimated model:

$$RSS = \sum_{i=1}^{4}\sum_{j=1}^{4}({}^{R}T_{T_{\text{tar}}}(i,j) - {}^{R}T_{T_{\text{est}}}(i,j))^2$$

For the modeled data the RSS are:

Python	$1.32 \cdot 10^{-10}$
Matlab	$3.30 \cdot 10^{-29}$

It can be seen that the algorithm works really well in Matlab but seems to have some issues by using python's environment *numpy* for solving the linear equation.
For the data we collected during the project the matrices are the following:

$${}^{R}T_{T_{\text{target}}} = \begin{bmatrix} 0.6940 & -0.1306 & 0.7079 & -1.0052 \\ -0.7192 & -0.0834 & 0.6897 & -0.5073 \\ -0.0310 & -0.9879 & -0.1518 & 0.4933 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^{R}T_{T_{\text{phyton}}} = \begin{bmatrix} 0.0268 & 0.0133 & 0.0254 & -0.3645 \\ -0.0216 & 0.0041 & 0.0182 & -0.0698 \\ 0.0150 & -0.0305 & 0.0016 & 0.50216 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^{R}T_{T_{\text{matlab}}} = \begin{bmatrix} 0.5542 & 0.7797 & 0.2912 & -0.0020 \\ 0.0991 & -0.4092 & 0.9070 & -0.0002 \\ 0.8264 & -0.4738 & -0.3041 & 0.0031 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The RSS results are:

Python	3.4212
Matlab	4.3716

The error with the collected data is still huge. In conclusion, there must be another error either in the calibrations or in the pose estimation of the target.

C. Head Tracking - CG

For motion compensation the head pose needs to be estimated. Therefore, the head first needs to be localized. This can be done by identifying the area of depth which contains a head. The authors [9] propose an adaptive detection filter where the size depends on the expected size of the head at various depths. They assume that the focal length is known and that only one head is present in the scene. Moreover, the subject is supposed to be at depth between d_m and d_M so that a binary mask $\epsilon(i,j) = d_m < d_0(i,j) < d_M$ can be used. The expected width $w(i,j)$ and height $h(i,j$ of the head are centered at (i,j). A score $s(i,j)$ is obtained by resizing the kernel and convolving it with the binary mask $\epsilon(i,j)$. The maximum scored pixel gives the center of the identified head and therefore the full head location by using width and height at that pixel. The authors choose a larger region of interst so that the head is fully contained in the process [9].

For the head pose estimation, the authors divide the process into a reference model, a cost function, optimization and morphable model fitting. For the reference model the provided head model can be used. The facial surface is chosen as set of 3d vertices. Mismatches are considered as weight factor for the surface. The cost function models the head pose in 6d consisting of position and orientation and quantifies the difference between the observed and hypothetical data by factoring out the effects of outliers. For optimization particle swarm optimization and iterative closest point are used. The trade-off between accuracy and computation time needs to be considered. In the last step, the shape and weights of the morphalbe model are updated and point correspondences are identified to compute a new set of coefficients of the morphable model [9].

In our project, we initially used an open source code based on the approach of Fanelli et al [10]. This approach estimates the head pose from given depth data and is based on discriminative random regression forests. That is training combinations of random trees by splitting the nodes so that the entropy of the label distribution and the variance of the head position and orientation are reduced [10]. Unfortunately, this algorithm did not work for our problem since the used point clouds have a different structure than the used data of our head model.

D. Motion Compensation

The last task consisted of three smaller subtasks which will be discussed individually.

1) Initial Path Planning: This task asked for path planning for the initial coil pose to avoid hitting the head. Since this was an optional task, our group decided to focus on the other tasks and rather get these working. Therefore, we did not include path planning.

2) Trajectory Planning - GM: Trajectory planning is about how fast a movement should happen. It describes the position, velocity and acceleration as a function of time. For this, there are many different approaches, such as cubic splines, LSPB or Bang-Bang Control (which requires the least amount of time). Since there is a post-processing of trajectories happening in the robot software, we decided not to include Trajectory Planning and set the corresponding variable *traj_time* to $0.01\,s$. Nevertheless, we considered using Trajectory Planning at first. Our general idea was to identify the slowest movement of the six joints and take this as the time in which all movements should happen.
The approach of this idea can be found in the move-node under the keyword *Approach for trajectory planning (not used)*[which is commented out since we did not use it], although the last command to assign *traj_time* to the value of *T_tmp* is missing.

3) Motion Compensation - FS, AU: The motion compensation system brings all individual parts together. First, the head pose is tracked by the Kinect camera to get ${}^{K}T_{H}$. With this information a new goal pose of the coil can be calculated since the matrices from end effector to head ${}^{E}T_{H}$ and from the robot base to the Kinect camera ${}^{B}T_{K}$ are known II-B.

$${}^{B}T_{E} = {}^{B}T_{K}\,{}^{K}T_{H}\,{}^{H}T_{E} \qquad \text{(new goal)} \qquad (1)$$

This matrix is sent to the inverse kinematics (see fig. 2). This is repeated during the whole TMS process.

III. Conclusion & Evaluation - GM

The overall task was to create a vision-guided robot system to achieve head movement compensation for transcranial magnetic stimulation. This was done in four tasks, namely the robot kinematics, the calibration of the camera, the head movement estimation and the combination of the latter three tasks to compensate head movement. To test the setup a styrofoam head was mounted on a separate robot and made unknown movements the motion compensation system needed to compensate. As mentioned before neither the hand-eye calibration nor the head tracking were functioning at this moment of time. Thus, these parts were replaced by given algorithms to test the remaining parts. One was able to see with the bare eye that there was a delay in the movement (guessing it is around $0.5\,s$). Due to the sophisticated structure of the evaluation data, we were not able to evaluate the results quantitatively. We tried to by using the rosbag package as well as the provided .py file, but it was not feasible for us to understand it well enough in time.

Acknowledgment

The authors would like to thank the Institude of Medical Technology for the great support during the group project. Especially, we would like to thank Omer Rajput and Mareike Wendebourg for the helpful consultations even out of our lab times.

References

[1] K. P. Hawkins, "Analytic inverse kinematics for the universal robots ur-5/ur-10 arms," Georgia Institute of Technology, Tech. Rep., 2013.

[2] (2018) The eigen website. [Online]. Available: http://eigen.tuxfamily.org/

[3] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, Nov 2000.

[4] T. Wiedemeyer, "IAI Kinect2," https://github.com/code-iai/iai_kinect2, Institute for Artificial Intelligence, University Bremen, 2014 – 2018, accessed September 1, 2018.

[5] F. Ernst, L. Richter, L. Matthäus, V. Martens, R. Bruder, A. Schlaefer, and A. Schweikard, "Non-orthogonal tool/flange and robot/world calibration," *The International Journal of Medical Robotics and Computer Assisted Surgery*, vol. 8, no. 4, pp. 407–420, 2012.

[6] OpenCVteam, "Open source computer vision library," https://opencv.org/, 2018.

[7] O. Wasenmüller and D. Stricker, "Comparison of kinect v1 and v2 depth images in terms of accuracy and precision," in *Computer Vision – ACCV 2016 Workshops*. Springer International Publishing, 2017, pp. 34–45.

[8] S. Umeyama, "Least-squares estimation of transformation parameters between two point patterns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 4, pp. 376–380, April 1991.

[9] G. P. Meyer, S. Gupta, I. Frosio, D. Reddy, and J. Kautz, "Robust model-based 3d head pose estimation," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 3649–3657.

[10] G. Fanelli, T. Weise, J. Gall, and L. V. Gool, "Real time head pose estimation from consumer depth cameras," in *Proceedings of the 33rd International Conference on Pattern Recognition*, ser. DAGM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 101–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=2039976.2039988