Diploma Work

Grasping a Book from a Table with a 7DOF Manipulator

Submitted by:
Bãbãu Mircea Alexandru

August 2007

Supervisors:

Prof. Dr.-Ing. Axel Gräser
Dipl. Ing. Darko Ojdanić
Prof. Dr.-Ing. Ivan Bogdanov

Contents

List of figures

1.Introduction

The aim of this work is to enable disabled people to work again. One of the proposed scenarios is the library task". Within this scenario, user should be able to do several manipulative actions, such as: moving the robot arm towards a book, taking the book from a shelf, scanning the book etc.

This diploma work deals with only a part of the library scenario, like approaching the book, moving and scanning the book. For the implementation of these skills the FRIEND II system is used.

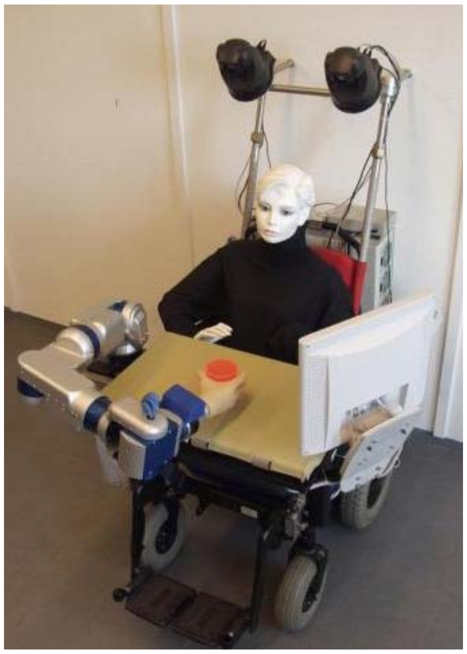


Fig.1 FRIEND II system

The rehabilitation robot FRIEND II (**F**unctional **R**obot with dexterous arm and user-fr**IEN**dly interface for **D**isabled people) is the successor of the FRIEND I system. The basic component of FRIEND II is the commercial wheelchair with a mounted lightweight robot arm. Additionally the system is provided with a tactile skin and a scale (both combined in the wheelchair tray), a stereo camera system, a computer system, a force torque sensor and a hand prosthesis. The latter ones are mounted to the last joint of the robot arm.

The long-term goal of the overall project is the support of disabled people at activities of daily living (ADL). To reach this goal a software framework was developed that introduces a modified hybrid multilayer-architecture to process semi-autonomous tasks.

1.1 Functional Robot with dexterous arm and user-frIEndly interface for Disabled people

To act safely and effectively in a real unstructured and clustered environment, rehabilitation robots require dexterous manipulators with at least 7 joints (degrees of freedom, or DoFs) like a human arm. FRIEND II is equipped with such a dexterous lightweight robot arm with 7 joints. This electrically driven robot arm was developed by AMTEC robotics GmbH (Berlin) with the functional specification given by the IAT. It has a humanlike kinematics: the arm is composed of a series of turn- and pan-joints with perpendicular axes respectively.

The combination of a turn-pan-turn-joint is cinematically equivalent to a spherical joint like the human shoulder or the wrist joint, and the middle (the 4th) pan-joint corresponds to the elbow. The arm is mounted on a linear axis which allows it to drive in a specific home position and reduce visibility if it is not in use. At the wrist a multi-axis force/torque sensor, model Gamma, from ATI-Industrial Automation (NC, USA) is integrated.

This compact, light and robust monolithic transducer uses silicon strain gauges, providing high noise immunity, to sense forces and torques from all three directions (x, y and z) of the tool frame. To process the strain

gauge information into digital CAN-Bus signals a compact wrist mounted electronics unit has been developed.

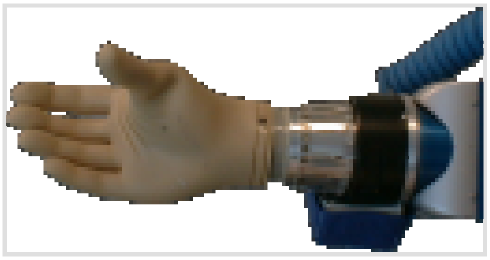


Fig.2 Hand-like gripper

The robot arm is equipped with an Otto Bock Sensor-Hand, used as gripper. The necessary mechanical as well as electrical adaptations were made in agreement with the Otto Bock Health Care (Duderstadt, Germany). A gripper force and a slip control mode which will be activated from the FRIEND II automation system are integrated in the Sensor-Hand.

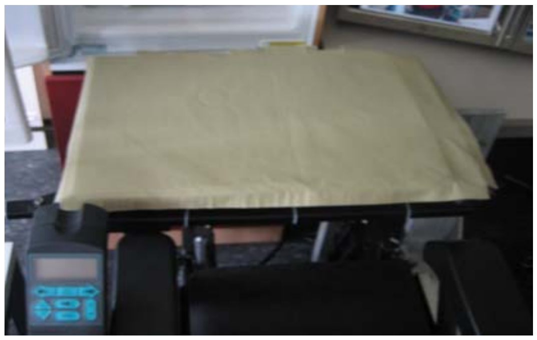


Fig.3 Intelligent tray

FRIEND II is equipped with an “intelligent” tray as a kind of a smart device. The term ‘smart’ expresses its ability to measure the weight of objects placed on the tray and to provide the position information about the placement of the objects relative to the tray coordinate system. The tray can be divided into two subsystems: A scale for the measurement of weight changes of objects placed on the tray and an artificial skin (touchpad) for the detection of object positions. The scale consists of an off-the-shelf digital scale with a measuring precision of ±1g that is connected to the main system PC. The position detection is realized by a touchpad sensor that was developed at the IAT. The touchpad consists of a 48x30 matrix, where each

matrix element has binary output. Binary 1 denotes the presence of a weight greater then 5g per element, 0 indicates that there is no load on the corresponding matrix element. Hence, the result can be treated as a binary image and known image processing methods can be used for object segmentation.

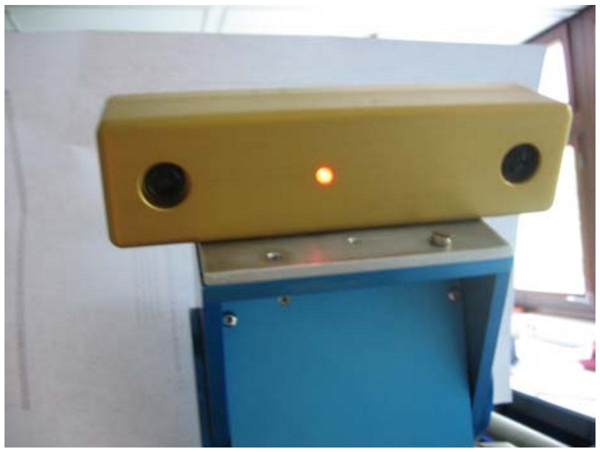


Fig.4 Imagine capturing system

The vision system of the FRIEND II system consists currently of two pan-tilt-Zoom cameras by Sony. They are combined to a stereo camera system and the image capturing is done by two PCI capture devices. The overall system is controlled by a 3-PC system, each with a Pentium 4 processor at 3 GHz. The first system is used for the image processing, the second for the manipulator motion planning and the third for task planning processes and the Human-Machine-Interface (HMI). The interconnection between these computer systems is realized using the free CORBA implementation ACE/TAO. As graphical interface to the user a 14" LCD Display is mounted to the tray.

1.2 MASSiVE - Multilayer Architecture for Semi-Autonomous Service-Robots with Verified Task-Execution

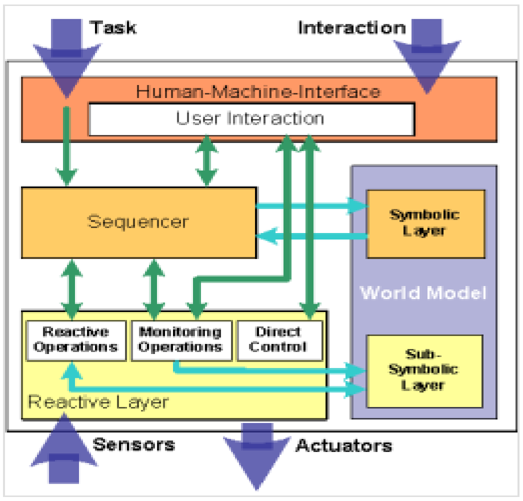


Fig.5 Control architecture

A necessary prerequisite for a service robotic system, acting in mostly unstructured and clustered environment, is a carefully designed concept of a software framework. The base of the software framework used for the Friend II system are two principles: Semi-structuring of tasks and close integration of the user’s cognitive capabilities during the execution of a task. Without a structured approach and restrictions towards feasibility, the service robotic systems’ complexity leads to very high costs and low efficiency and the realization is estimated to be a rather long-term goal.

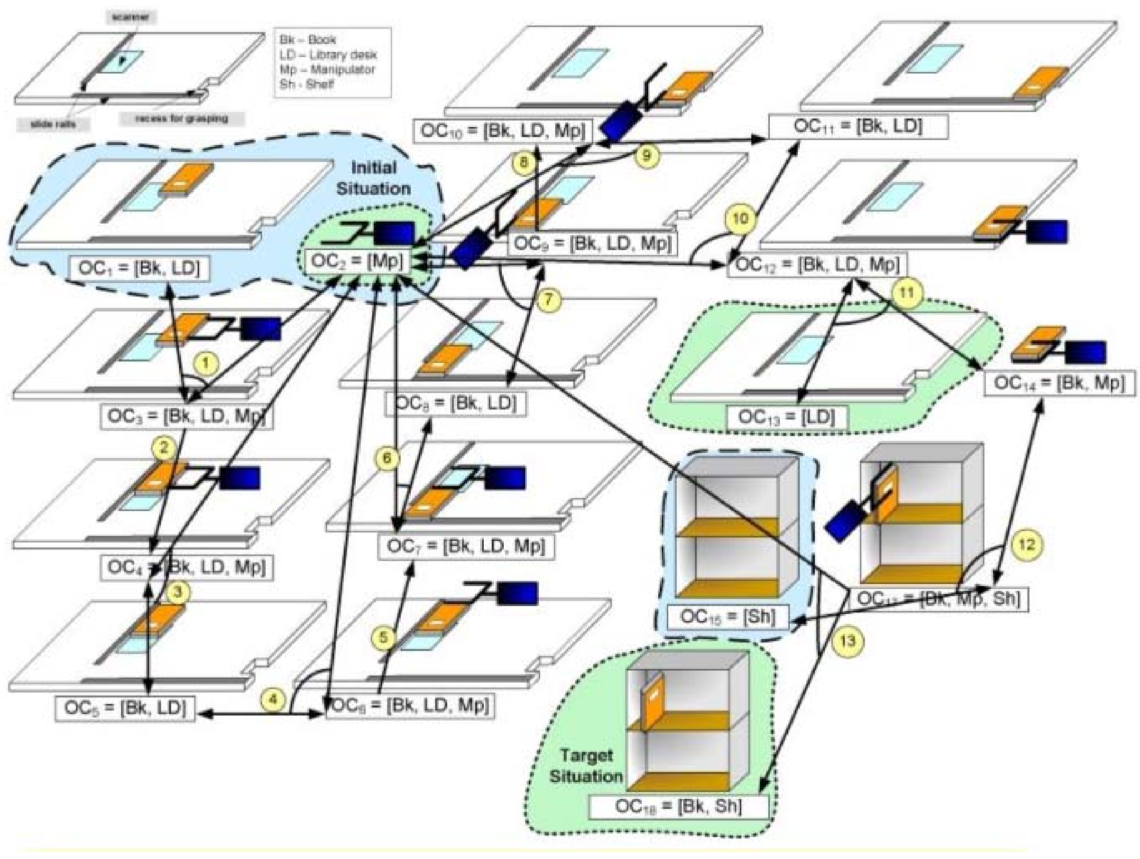


Fig.6 Library “book” scenario

To fulfill the requirements of semi-autonomous task execution, the top layer of MASSiVE, usually being the deliberator, has been replaced with a human-machine-interface (HMI). This HMI specially satisfies the needs in the field of rehabilitation robotics but also provides good advantages for general service robots, such as independence from the input device hardware or controlled direct access to actuators. Subsequently, the ability of deliberation has been moved to sequencer which coordinates command-requests, human-machine interactions as well as autonomously executed reactive operations on base of so called process-structures. These process-structures are used as input for task planning and execution in the sequencer and subsumes predefined semi-structured task knowledge. The output of the sequencer is an generated plan for task execution that consists of autonomously processable sub-tasks.

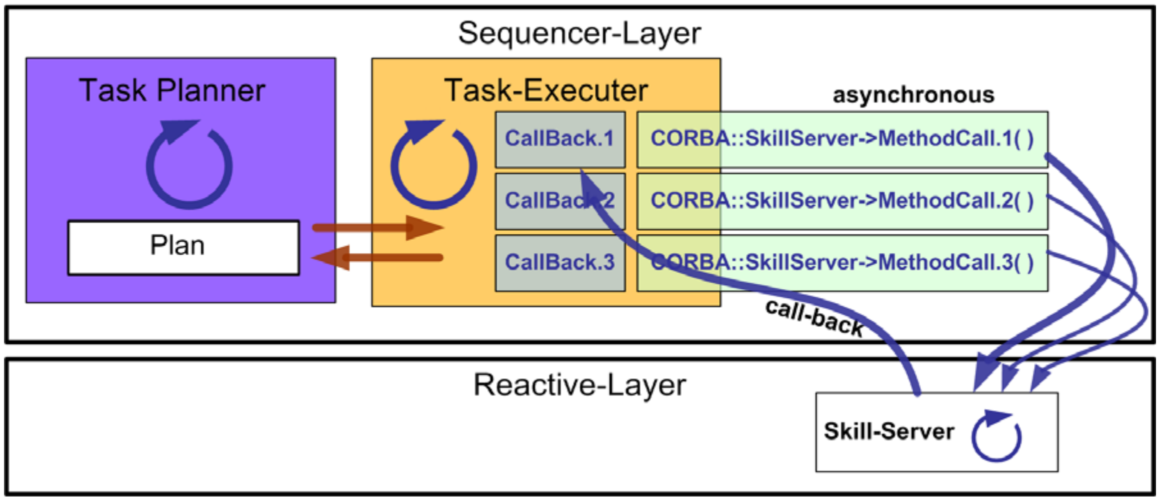


Fig.7 The sequence of calling a skill

On base of the generated plan the task execution takes place in the reactive layer or in case of user interaction in the HMI. MASSiVE provides for task execution a CORBA-based servant network (powered by the free CORBA implementation ACE/TAO). So-called skill-servers offer basic system skills, which are algorithms that operate on base of sensor input and control the system's actuators. In most cases the skills will be realized as closed control loops and thus realize reactive system behavior.

The advantages of CORBA are mainly the location transparency of modules as well as the opportunity for asynchronous skill execution. The first aspect means that the physical relocating of modules does not affect the system design whereas the latter aspect is mandatory for the effective and safe operation of a multi-sensor system including the ability to abort the execution at any time. The skill-servers in the reactive layer group system hardware according to functionality and they are responsible for the management of the hardware-servers associated to them

1.3 Outline of the following chapters

The concern is to grasp a book from the table using a manipulator with seven degrees of freedom. This is the task that this project had dealt with, as shown in the next chapters.

The second chapter introduces the mathematical basic needed to complete the task from the notion about matrices and vectors to trajectory planning and robot control.

The third chapter deals with the problem itself, explains the structure of the program and provides further information about the skills created for this task.

The forth chapter states the conclusion of this project and some guide lines for future work.

2.Mathemathical Background

The study of robot manipulation is concerned with the relationship between objects as well as between objects and manipulators. Since the description of these relationships is based on the use of vectors, transformation matrices and coordinate systems, the goal of this section is to establish notation and to review mathematics.

2.1. Matrices and Vectors

Matrices are denoted by uppercase bold letters like: **R**, **T**, …

A matrix **R** of dimensions (m x n), with m and n positive integers, is an array of elements r_{ij} arranged into m rows and n columns:

$$\mathbf{R} = \left[r_{ij}\right]_{\substack{i=1,\dots,m \\ j=1,\dots,n}} = \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ r_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & & \vdots \\ r_{m1} & r_{m2} & \cdots & r_{mn} \end{bmatrix}.$$

If m = n, the matrix is said to be square. An (n x n) square matrix **R** is said to be *diagonal* if $r_{ij} = 0$ for $i \neq j$. If an (n x n)

diagonal matrix has all unit elements on the diagonal ($r_{ii} = 1$), the matrix is said to be *identity matrix* and is denoted by **I**n.

$$\mathbf{R} = \begin{bmatrix} r_{11} & 0 & \dots & 0 \\ 0 & r_{22} & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & r_{nn} \end{bmatrix}$$

Vectors are denoted by lowercase bold letters like: **v**, **u**,…

In contrast to scalar quantity characterized by magnitude only, vector are characterized by its direction as well as by its magnitude. Usually, a vector is represented graphically by a directed line segment whose length and direction correspond to the magnitude and direction of the vector.

The *scalar product* (*dot product* or *inner product*) of two given vectors **v** and **u** is defined as:

$$\mathbf{v} \cdot \mathbf{u} = v_1 u_1 + v_2 u_2 + v_3 u_3,$$

or

$$\mathbf{v} \cdot \mathbf{u} = |\mathbf{v}| \cdot |\mathbf{u}| \cdot \cos\theta,$$

where θ is the angle between the two vectors.

2.2. Cartesian coordinate systems

If there is a subset of linearly independent vectors { $\mathbf{x_0}$,$\mathbf{y_0}$,$\mathbf{z_0}$} in three-dimensional vector space V and a set of scalars { v_1, v_2,v_3 } such that every vector **v** in V can be expressed as:

$$\mathbf{v} = v_1 \mathbf{x_0} + v_2 \mathbf{y_0} + v_3 \mathbf{z_0}$$

then it is said that **v** is linear combination of the vectors {$\mathbf{x_0}$,$\mathbf{y_0}$,$\mathbf{z_0}$} which represent the basis vectors for a vector space V.

If a set of basis vectors {$\mathbf{x_0}$,$\mathbf{y_0}$,$\mathbf{z_0}$} are all drawn from a common origin 0 and are orthogonal to each other, that is, if they intersect at right angles at the origin 0, then they form a *rectangular* or *Cartesian* coordinate system. Furthermore, if each of the basis vectors is of unit length, the coordinate system is called *orthonormal*. Basis vectors of an orthonormal coordinate system (orthonormal vectors) satisfy the following equations:

$$\mathbf{x_0} \cdot \mathbf{y_0} = 0;\ \mathbf{x_0} \cdot \mathbf{z_0} = 0;\ \mathbf{y_0} \cdot \mathbf{z_0} = 0;\ ,$$

$$|\mathbf{x_0}| = |\mathbf{y_0}| = |\mathbf{z_0}| = 1.$$

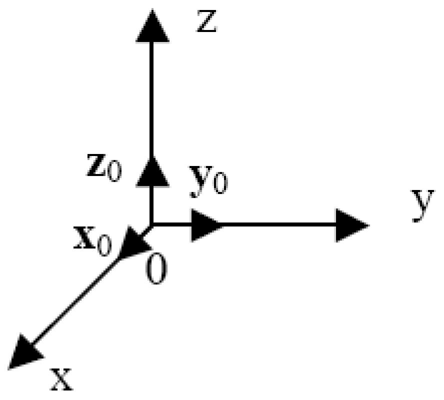


Fig.8. Cartesian coordinate system

2.3. Coordinate transformations

Consider two coordinate systems shown in figure below. Let {A} be the orthonormal reference frame and $\{\mathbf{x_A}, \mathbf{y_A}, \mathbf{z_A}\}$ be the unit vectors of the frame axis. The coordinate frame {B} is completely described with respect to {A} (its position and orientation is completely defined related to {A}) by the following vectors:

${}^{A}\mathbf{v}_{A0\rightarrow B0}$- vector from the origin of {A} to the origin of {B} expressed related to {A}

${}^{A}\mathbf{z_B}, {}^{A}\mathbf{y_B}, {}^{A}\mathbf{x_B}$ - orthonormal vectors of {B} expressed with respect to {A}

where for the sake of notation simplicity the following notation is adopted:

leading superscript denoting the reference system$\rightarrow {}^{A}\mathbf{v}_{B} \leftarrow$*subscript denoting the related system*

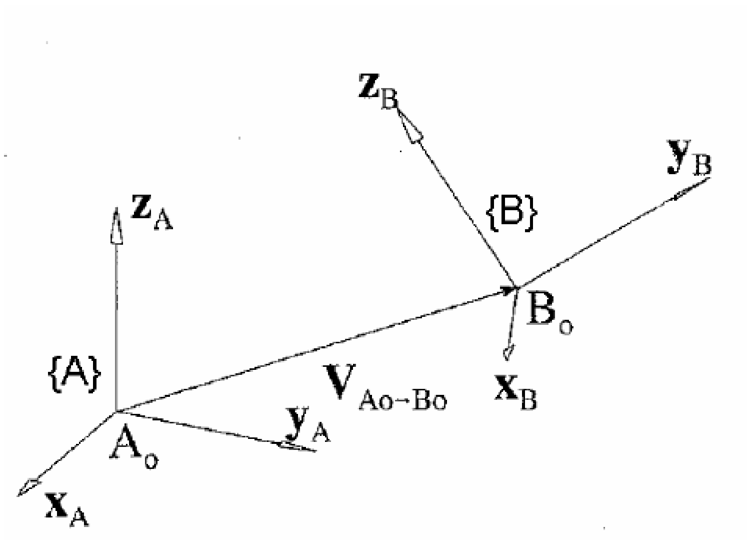


Fig.9 Relationship between two coordinate systems

The orthonormal vectors of related coordinate system {B} with respect to {B} are expressed by:

$$ {}^{B}\mathbf{x}_{B} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix};\ {}^{B}\mathbf{y}_{B} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix};\ {}^{B}\mathbf{z}_{B} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} $$

while related to {A} these vectors can be written in the form:

$$ {}^{A}\mathbf{x}_{B} = \begin{bmatrix} x_{a_1} \\ x_{a_2} \\ x_{a_3} \end{bmatrix};\ {}^{A}\mathbf{y}_{B} = \begin{bmatrix} y_{a_1} \\ y_{a_2} \\ y_{a_3} \end{bmatrix};\ {}^{A}\mathbf{z}_{B} = \begin{bmatrix} z_{a_1} \\ z_{a_2} \\ z_{a_3} \end{bmatrix} $$

2.4. Homogenous transformations matrices and Inverse transformations

Homogenous coordinates play also an important role in projective geometry. Homogenous coordinates embed three-dimensional space R^3 into P^3, the three-dimensional projective space, which is R^4.As a result, inversions or combinations of linear transformations are simplified to inversion or multiplication of the corresponding matrices.

Matrix ${}^{A}\mathbf{T}_{B}$ is so-called transformation matrix of the coordinate system {B} to coordinate system {A}. Using homogenous transformation matrices we can write in the case of

pure rotation of the coordinate system {B} with respect to {A}matrix ${}^{A}\mathbf{T}_{B}$ is of the form:

$$ {}^{A}_{B}\mathbf{T} = \begin{bmatrix} {}^{A}_{B}\mathbf{R} & \mathbf{0}_{3\times1} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix} $$

and in the case of *pure translation* of the coordinate system {B} with respect to {A} matrix has the following form:

$$ {}^{A}_{B}\mathbf{T} = \begin{bmatrix} \mathbf{I}_{3} & {}^{A}\mathbf{p}_{B_0} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix} $$

Where the notation have the following meaning:

${}^{I}\mathbf{p}_{j}$ - point vector defining the coordinates of a point j with respect to the coordinate system{I};

$^{A}\mathbf{R}_{B} = [\,^{A}\mathbf{x}_{B}\ ^{A}\mathbf{y}_{B}\ ^{A}\mathbf{z}_{B}\,]$- (3 x 3) matrix defining the rotation of the coordinate system {B} with respect to {A};

Inverse transformations is the idea starting from given transformation matrix of the coordinate system {B} to reference coordinate system {A}, $^{A}\mathbf{T}_{B}$, to determine the inverse.

In general, given a transformation matrix

$$\mathbf{T} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the inverse is

$$\mathbf{T}^{-1} = \begin{bmatrix} n_x & n_y & n_z & -\mathbf{p}\cdot\mathbf{n} \\ o_x & o_y & o_z & -\mathbf{p}\cdot\mathbf{o} \\ a_x & a_y & a_z & -\mathbf{p}\cdot\mathbf{a} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where **p**, **n**, **o** and **a** are column vectors of the transformation matrix **T** and "." represents the vector dot (scalar) product.

2.5. Compound transformations

The ability to perform matrix multiplication to yield compound transformations is the primary reason for the introduction of the (4x4) homogenous transformation notation. The described ability is particularly useful in robotics since it enables the combination of mathematical and graphical description of a manipulator as *open-chain* constituted by n+1 links connected by n joints where a coordinate frame is attached to each link. The direction of arrow, pointing from one origin to another origin, indicates which way the frames are defined as it is shown in the figure in the case of an open-chain constituting of 5 links.

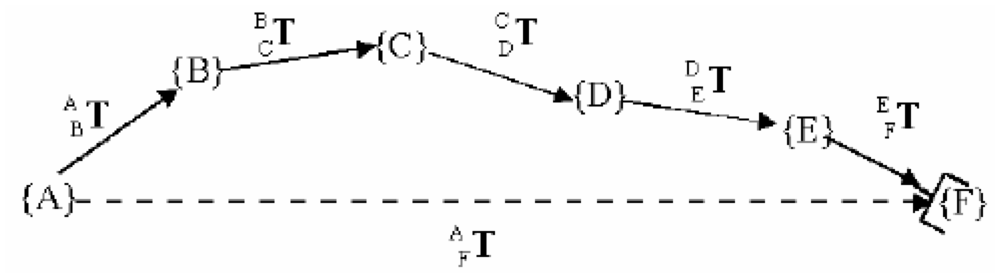


Fig. 10. Coordinate transformations in an open-chain manipulator

Then, the coordinate transformation describing the position and orientation of the coordinate frame {F} attached to the end-effector (*end-effector frame*) with respect to *base frame* {A} is given by the following transfer equation:

$$^{A}_{F}\mathbf{T}={}^{A}_{B}\mathbf{T}\cdot{}^{B}_{C}\mathbf{T}\cdot{}^{C}_{D}\mathbf{T}\cdot{}^{D}_{E}\mathbf{T}\cdot{}^{E}_{F}\mathbf{T}$$

2.6. Standard transformations

In many problems, the relationship between coordinate systems will be defined in terms of rotations about the x, y or z axes. Using the (4 x 4) homogenous coordinate transformation matrix notation, the transformation matrix $^{A}\mathbf{T}_{B}$ representing the rotation of the coordinate system {B} by φ degrees about the z axis with respect to {A} is of the form:

$$\mathrm{Rot}(z,\varphi) = \begin{bmatrix} {}^{A}_{B}\mathbf{R} & \mathbf{0}_{3x1} \\ \mathbf{0}_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

According to the same procedure as above, the transformation matrices representing the rotation of the coordinate system {B} by α degrees about the x and by β degrees about the y axes with respect to {A} are of the form:

$$\mathrm{Rot}(x,\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad \mathrm{Rot}(y,\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

2.7. Euler angles

The nine elements in a general (3x3) rotation matrix, defining the rotation of one coordinate system with respect to another, are not independent quantities but related by six constraints due to the orthogonality conditions. This implies that *three parameters* (three different independent rotations around frame axes) are sufficient to describe orientation of a rigid body in space. Frequently used way to specify an arbitrary rotation matrix in terms of only three independent quantities is to use the so-called *Euler angles.* In the following three sets of Euler angles are analyzed: the so-called ZYZ, ZYX and *Roll, Pitch, Yaw* angles.

The rotation described by Euler angles, known as ZYZ angles, is obtained as composition of the following elementary rotations: rotation by φ degrees about the z axis, then rotation by θ degrees about the new y axis (y'), and, finally, rotation by ψ degrees about the new z axis (z").

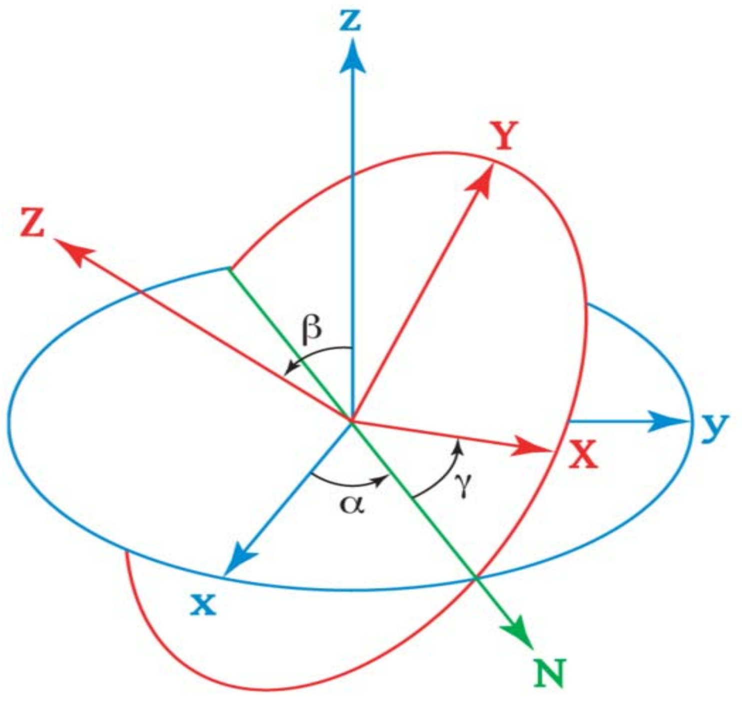


Fig. 11. Euler angles

The Euler transformation, Euler(φ, θ, ψ), can be computed by multiplying of the matrices of above mentioned elementary rotations, made with respect to the current frame (transformed coordinate system):

$$\mathrm{Euler}(\phi,\theta,\psi) = \mathrm{Rot}(z,\phi)\,\mathrm{Rot}(y',\theta)\,\mathrm{Rot}(z'',\psi)$$

The Euler angles are known as ZYX angles since they correspond to rotation by φ degrees about the z axis, then rotation by θ degrees about the new y axis (y'), and, finally, rotation by ψ degrees about the new x axis (x"). In this case, the Euler transformation can be computed as follows:

$$\mathrm{Euler}(\phi,\theta,\psi) = \mathrm{Rot}(z,\phi)\,\mathrm{Rot}(y',\theta)\,\mathrm{Rot}(x'',\psi)$$

Another set of Euler angles originates from a representation of orientation in the (aero)nautical field. These are so-called *Roll*, *Pitch* and *Yaw* angles (RPY). These angles denote the typical motions of an (air)craft or a ship. Roll corresponds to a rotation by φ about the z axis, pitch corresponds to a rotation by θ about the y axis while yaw corresponds to a rotation by ψ about the x axis. In the case of the manipulator end-effector the angles φ, θ and ψ represent rotations defined with respect to a fixed frame attached to the centre of the end-effector as shown in figure:

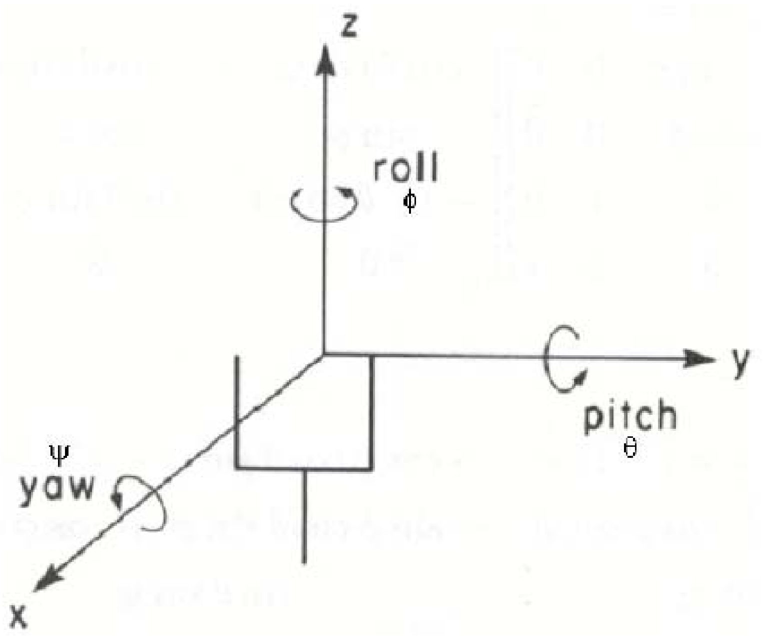


Fig.12. Roll, pitch and yaw angles for a manipulator

The resulting frame orientation is obtained by composition of rotations with respect to the fixed (reference) frame where the rotation about x axis is followed by a rotation about reference *y* axis and, finally, followed by rotation about the reference z axis. The corresponding rotation matrix can be computed via multi-plication of the matrices of elementary rotation as follows:

$$\mathrm{RPY}(\phi,\theta,\psi) = \mathrm{Rot}(z,\phi)\,\mathrm{Rot}(y,\theta)\,\mathrm{Rot}(x,\psi)$$

2.8. Manipulator kinematics

There are two problems regarding the manipulator kinematics.

The *direct* or *forward kinematics* problem is concerned with the relationship between the individual joints of the robot manipulator and the position and orientation of the tool or end-effector. Stated more formally, the forward kinematics problem is to determine the position and orientation of the end-effector frame, given the values for the joint variables of the robot, relative to the robot base frame. Sometimes, the direct kinematics problem is stated as changing the representation of manipulator position from a *joint space* description into a *Cartesian space* description. In contrast to forward kinematics problem, the *inverse kinematics* problem can be stated as follows: given a desired position and orientation for the end-effector of the robot, determine a set of joint variables that achieve the desired position and orientation.

2.8.1. Direct kinematics and Denavit-Hartenberg convention

For the general spatial case, the solution for the Direct kinematics is not so trivial as in the case of simple planar robot for example. This is because the joint angles do not simply add as they do in the planar case. Using of Denavit-Hartenberg kinematics parameters provide a systematic, general method for the solution of direct kinematics problem.

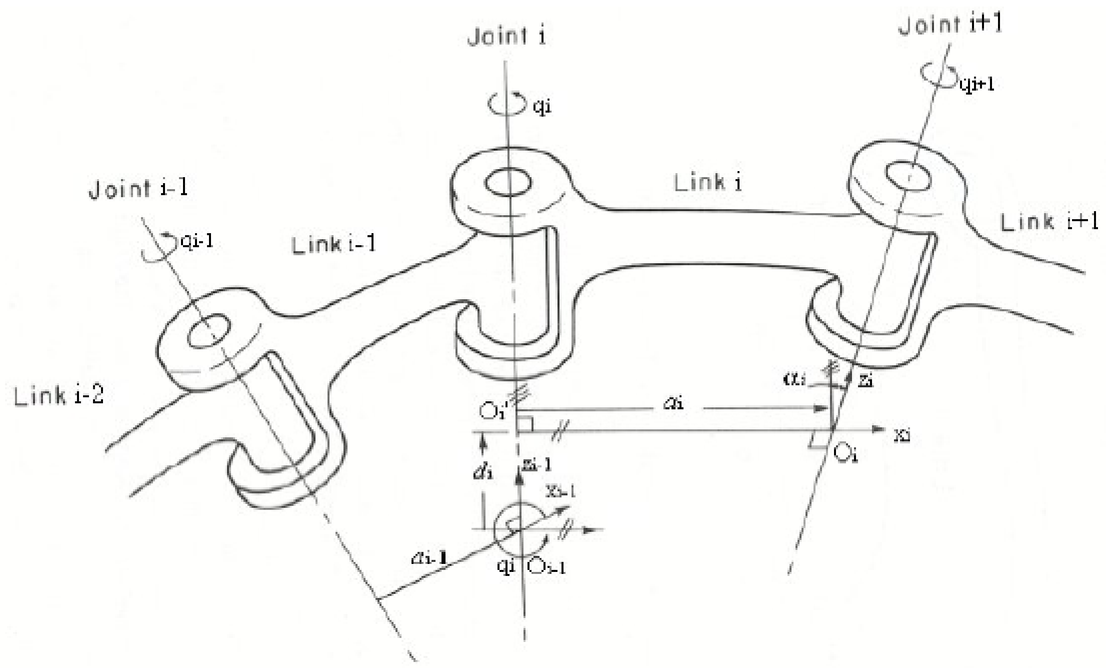


Fig.13 Denavit-Hartenberg kinematics parameters for revolute joints

According to the so-called *Denavit-Hartenberg convention* the coordinate frame of link *i* is defined as follows:

• Choose axis z_i along the axis of joint i + 1.

• Locate the origin O_i at the intersection of axis z_i with the common normal to axes z_i-1 and z_i. Also, locate O_i ' at the intersection of the common normal with axis z_i-1.

• Choose axis x_i along the common normal to axis z_i-1 and z_i with direction from joint i to
joint i + 1.

The Denavit-Hartenberg convention gives a nonunique definition of the link frame in the following cases:

• For frame 0 only the direction of axis z_0 is specified. The origin O_0 and the axis x_0 can be arbitrarily chosen.

• For frame n, since there is no joint n + 1, z_n is not uniquely defined while x_n has to be normal to axis z_n-1. Typically, joint n is revolute, and thus z_n is to be aligned with the direction of z_n-1.

• When two consecutive axes are parallel, the common normal between them is not uniquely defined.

• When two consecutive axes intersect, the direction of x_i is arbitrary.

• When joint i is prismatic, the direction of z_i-1 is arbitrary.

In all such cases, the indeterminacy can be exploited to simplify the procedure. For instance, the axes of consecutive frames can be made parallel. Once the link frames have been established, the Denavit-Hartenberg parameters (DH parameters) that completely specified the position and orientation of frame i with respect to frame i-1 can be defined as:

a_i – distance between O_i and O_i ',
d_i – coordinate of O_i ' along zi-1.
α_i – angle between z_i-1 and z_i axes about axis x_i.
q_i – angle between axes xi-1 and xi about axis z_i-1.

The parameters a_i and α_i are always constant and depend only on the geometry of connection between consecutive joints established by link I From the remaining two parameters of the above four only one is variable depending on the type of joint that connects link i-1 to link i.

2.9. Motion planning

Motion planning algorithms can be implemented in either Cartesian space (operational, W-space) or configuration space (C-space). Cartesian space represents a Euclidean space where the robot moves. C-space is the set of all possible configurations of a manipulator. The configuration of the manipulator is given in terms of joint variables – one for each joint. The dimension of the C-space is the number of parameters required to fully specify a configuration of the robot, which usually corresponds with the number of joints .

Most commonly used algorithms for motion planning are: cell decomposition, roadmaps and potential fields. In cell decomposition approach, the entire space is divided into a number of non-overlapping cells. The space is searched by using graph based searching algorithms. The nodes of the graph represent each cell and only neighbour cells can generate connected nodes. Cell decomposition can be exact, where the cells do not have to be the same size but contain only free space, or approximate, in which case the entire region is divided into equal sized cells that can be marked as either free or occupied. Most commonly used graph-based searching algorithms are: depth-first search, breadth-first search, best-first search, A* search, randomized search (like simulated annealing and GA) . Which method should be used is not easy to recommend since they have different characteristics regarding complexity (running-time), optimality of solution, guarantee of giving a solution if one exists, etc. In some cases better results can be obtained with approaches like backward or bidirectional search.

The motion planning problem is defined here as: finding an obstacle free trajectory in a given environment, where the goal is given as a location in Cartesian space. By "location" it is meant that the position and orientation of the gripper are defined. It is assumed that the environment is dynamic but its description and obstacles' displacement are completely known during the planning. This means that the distance calculation between manipulator and obstacles are always corresponding to the actual situation. The manipulator's motion will be planed step by step, so that in each step the manipulator has to choose between several incrementally generated TCP-positions. In that way, the manipulator will have the freedom to move through the free space and each choice will refer to the current robot configuration and state of environment. It has to be noticed that the ambition of motion planning is not only to reach the goal location but also to have a well configured motion. In each step, the distances between robot links and obstacles are observed and taken into account for the selection of the next configuration.

Additional TCP positions for the next step can be generated by a small variation of the main direction, which is computed as a straight line from the current TCP to the goal. Hence, in general, the goal directed movement of the end-effector (goalseeking) is guaranteed.

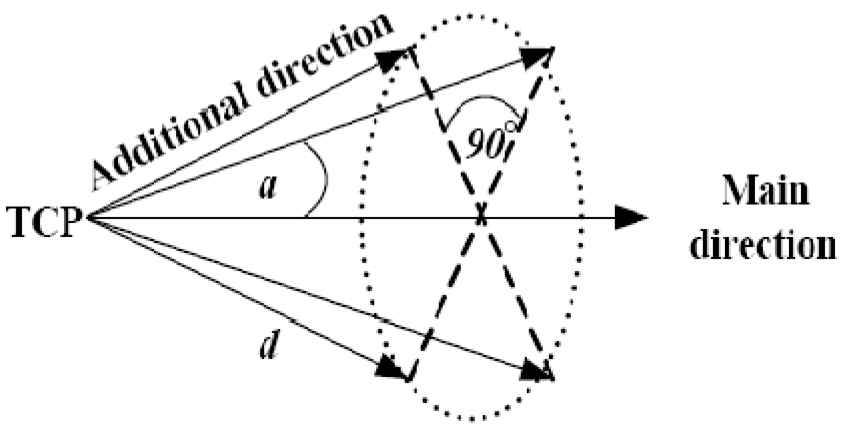


Fig.14 Geometry of the main and additional directions

Let *k* be the number of all directions, including the main one. The manipulator will pass in one step only a discrete distance *d* along the given direction. This means that in each step *k* points will be offered to be the next TCP position. These points will be called local goals in the following text. From figure …… can be seen that local goals lie on a sphere with the centre in the TCP. In a planar case it would be a circle. For making a decision in which direction to move, robot postures (configurations) for all *k* possible TCP positions have to be calculated. In order to calculate needed inverse kinematics solutions, a gripper orientation for each local goal has to be defined. The orientation will be specified in a way that the gripper is gradually governed from its current orientation toward its goal orientation.

The technique with two rotations as proposed in [Paul R. P., “Robot Manipulators: Mathematics, Programming and Control”, MIT Press, 1981.] is used here. The first rotation will serve to align the gripper in the required final direction, and the second rotation will control the orientation of the gripper about its axes. The alignment ratio between the current and the goal gripper orientation is proportional to $r = d/D$, where *D* is the distance from the gripper position to the goal position. This will insure that the gripper gradually, through the motion, achieve its final orientation. After the inverse kinematics calculations, *k* resulting configurations are available. Among these set of configurations one will be selected for the next step. If the manipulator is redundant, several (*m*) inverse kinematics solutions could be found for one local goal. In that case, the best configuration will be chosen

among $k \oplus m$ configurations. These m inverse kinematics solutions are determined by choosing m different postures in the nearness of the current robot posture. For the manipulator with 7DOF humanlike kinematics, m solutions can be resolved by using a redundant circle. Each solution,
differing in elbow position, will correspond to one sample on the redundant circle. A fast inverse kinematics solution can be obtained by using the concept "Kinematic Configuration Control" (KCC) .
Once the configuration is selected, the manipulator will execute the motion and the previously explained procedure will be repeated.

2.10 Robot control

The control problem for robot manipulators is the problem of determining the time history of joint inputs required to cause the end-effector to execute a commanded motion. The joint inputs may be joint forces and torques, or they may be inputs to the actuators, for example, voltage inputs to the motors, depending on the model used for controller design. The commanded motion is typically
specified either as a sequence of end-effector positions and orientations, or as a continuous path. The command task may regard either the execution of specified motions for a manipulator operating in free space, or the execution of specified motions and contact forces for a manipulator whose end-effector is constrained by the environment.

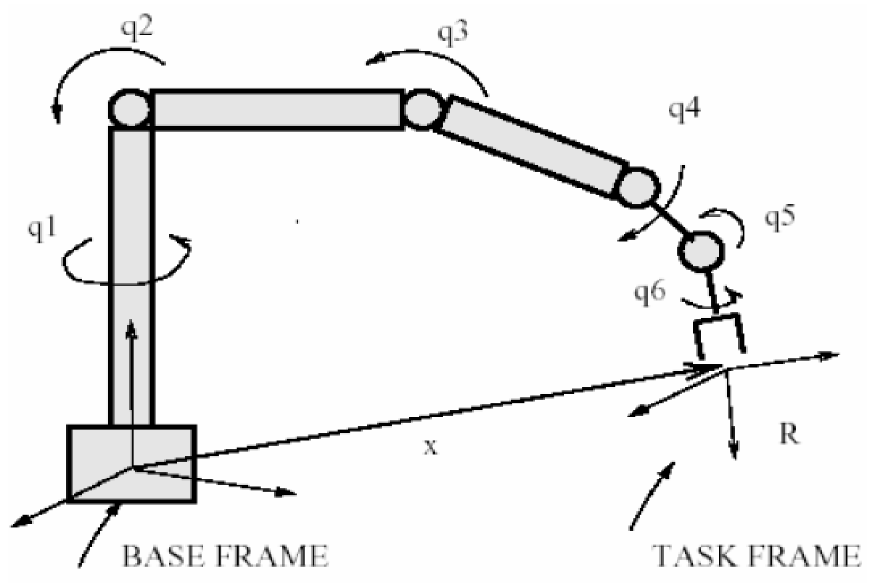


Fig.15 A link manipulator with the attached base frame, task frame, and configuration variables

There are many control techniques that can be applied to the control of manipulators. The fact that task specification is usually carried out in the operational space, whereas control actions are performed in the joint space leads to considering two types of general control schemes: *configuration space control* and *Cartesian (operational) space control*.

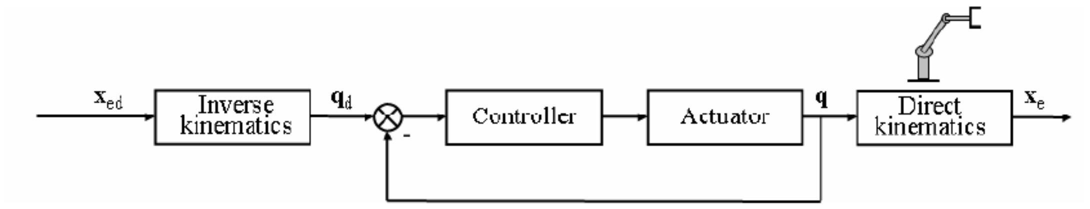


Fig.16. Standard robot control structure in Cartesian space

The disadvantage of solution shown above is that the operational space variables are controlled in an open-loop through the manipulator

mechanical structure (open-loop control with respect to the TCP-Tool Center Point). The conceptual advantage of the *closed-loop sian space control (closed-loop control with respect to the TCP)*, regards the possibility of acting directly on operational space variables. However, it is only a potential advantage since measurement of operational space variable is often performed not directly, but through a direct kinematics starting from measured joint space variables.

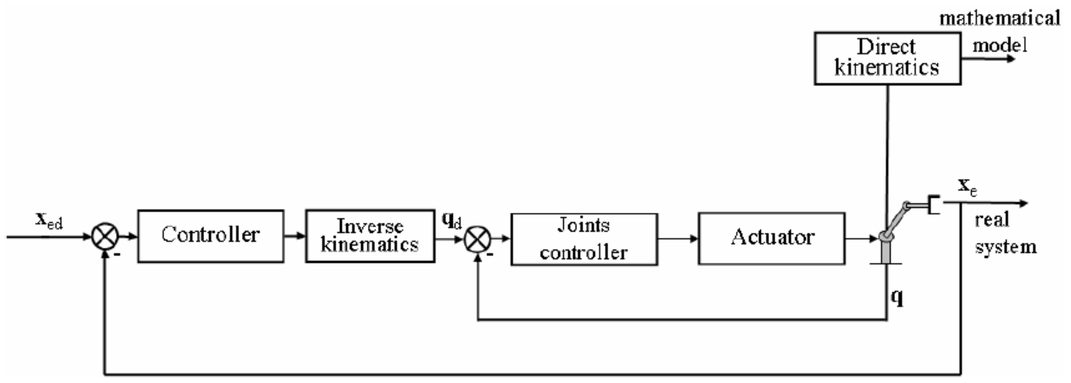


Fig. 17. Closed-loop control with respect to tool center point (TCP)

3.Grasping a Book

3.1 The Task

In order to scan a book or to place it in the shelf, it has to be grasped firstly. It is assumed that the book will lay on the table (or shelf). Due to the limitation of the gripper, grasping can not be directly implemented on the table. The book should probably be moved toward the end of the table, and than grasped. But this has to be further investigated. Available force-torque sensor should be used for feedback information during the process, which will improve the robustness.

It can be assumed that the location of the book comes from the cameras, but other local sensor may also be used (hand camera, tactile skin etc). Although other researchers will do image-processing part, a close cooperation is needed.

3.2 The layout of the experiment environment

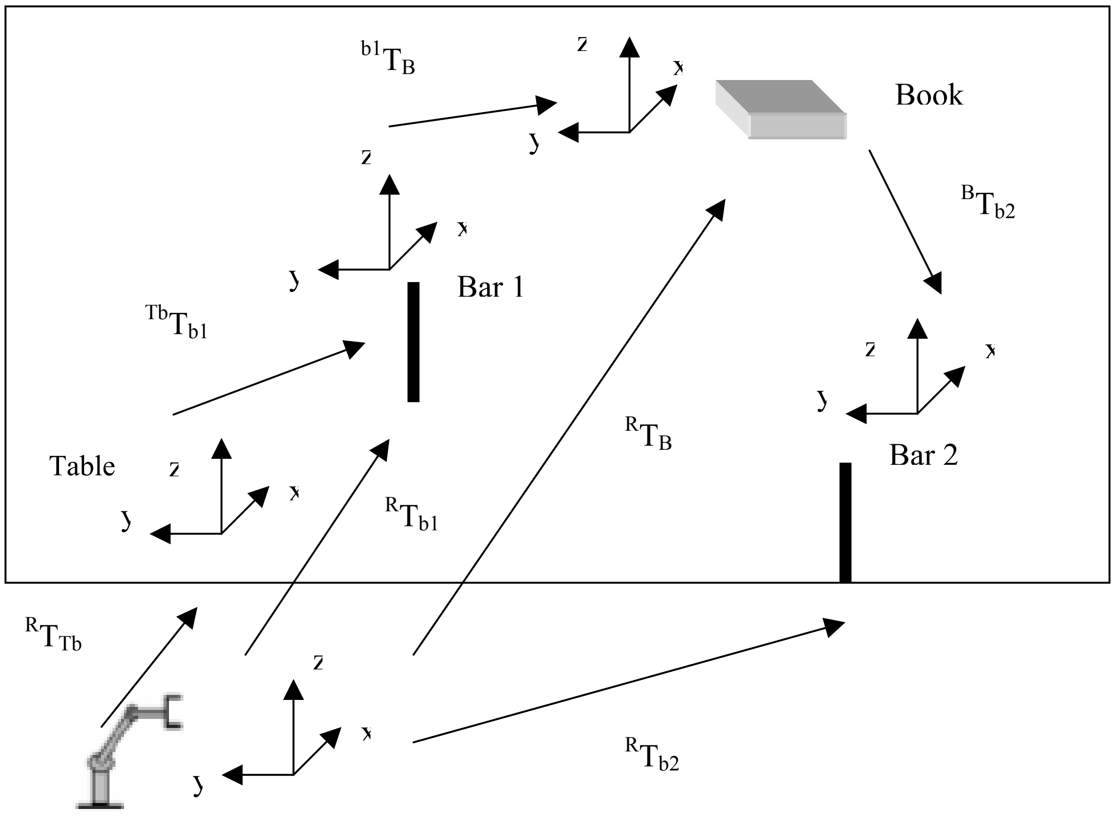


Fig.18 Table layout

On the table, as shown in the figure above, there are two bars that define specific action. The book will be move to the first bar where its bar code will be scanned, and then moved to the second bar where it will be position for grasping.

To move the manipulator towards the book we need the next sequence of matrix multiplication:

$$^{R}T_B = {}^{R}T_{Tb} * {}^{Tb}T_{b1} * {}^{b1}T_B$$

or we can get the direct location of the book using the imagine capturing system.

The next motion in the skill is moving the book to Bar1. There are several solution to get the location of the bar, one of then is:

$$^{R}T_{b1}={}^{R}T_{Tb}*{}^{Tb}T_{b1}$$

When the bar is reached the scanning process will begging, from the manipulator point of view the book is moved down slowly on the negative ox.

The last motion is to move the book to Bar2 where is it will set in the acceptable position for grasping. The location of the bar is obtain as:

$$^{R}T_{b2}={}^{R}T_{Tb}*{}^{Tb}T_{b2}$$

or using the imagine capturing system.

There is one more situation that must be taken into account as shown in the figure below:

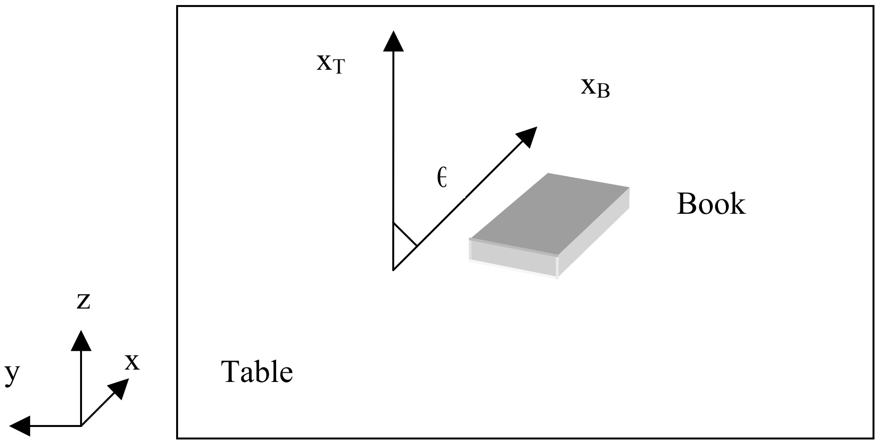


Fig.19 Book alignment

The book is not aligned with the table, which means that there is an angle θ between the ox axis of the table and the ox axis of the book. Therefore we need to align the book to the table, which means we need to rotate the book around the z-axis towards the ox axis of the table with θ:

$$^{Tb}T_{B}'= Rot\ (z,\theta)*\ {}^{Tb}T_{B}$$

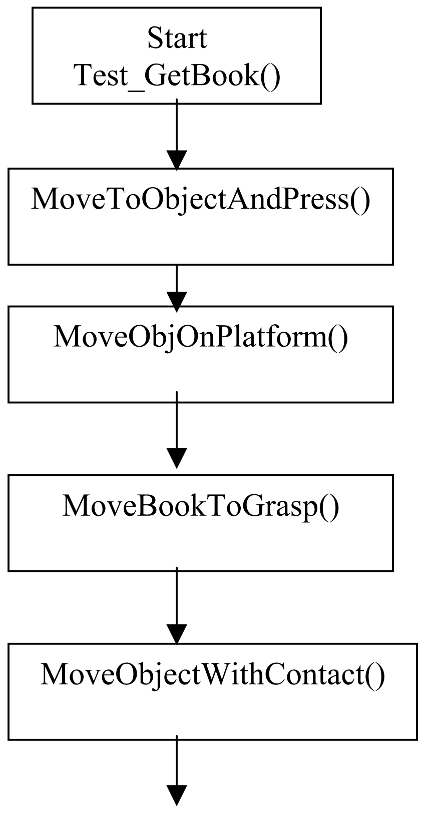
3.3 The structure of the program

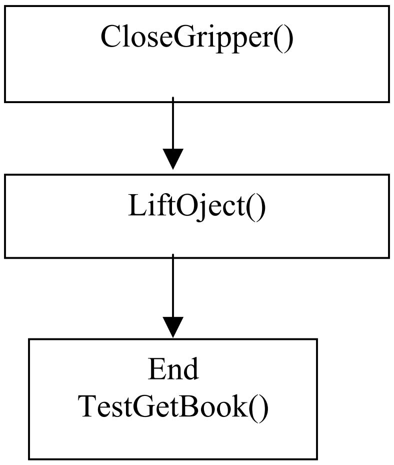
The program is written in C++ using ACE/TAO and COBRA libraries and is divided in three skills (or subprograms, functions) that are part of a specific program that deals with the manipulator actions, named ManipulatorSkillServer.cpp .

As I mention above the task is structured in 3 skills:

- MoveToObjectAndPress
- MoveObjOnPlatform
- MoveBookToGrasp

For testing it was necessary to write a small program, named Test_GetBook() in ManipulatorSkillServerTesApp.cpp that would call in chronological order the skills, for the task to be completed. The logical diagram of the program is shown below:





The last three function are skills or helper skills that already exist and can be used.

3.4 The skills

In this subchapter we will explain in detailed the main skills of this task.

3.4.1 MoveToObjectAndPress()

The parameters of this skill are:

- *pManipulator* is the name of the manipulator that will be used for grasping (here to derive gripper geometry).
- *pPlatfLocation* is the ID for the target object's location data in the World Model server.
- *pPlatfsize* is the ID for the target object's size data in the WorldModel server. (not yet used)
- *pObjLoccation* is the ID for the target object's location data in the WorldModel server.

- *pObjSize* is the ID for the target object's location data in the WorldModel server. (not yet used)

- *pCallBack* is the pointer to the proxy of the callback object. See above for possible callback messages.

- *bSimulative* is the flag for the sequencer skill to be executed only simulative.

Sends a CM_EXECUTION_SUCCESS, if execution was successful. If an error occurs CM_EXECUTION_FAILURE.
If SM_STOP_SKILL is sent, the initial condition before skill execution will be restored and CM_TERMINATION_SUCCESS is sent back. On a SM_KILL_SKILL a CM_KILL_SUCCESS follows directly with no more actions.

The parameters *pPlatsize* and *pObjsize* are not used in this program but they are declared in case of further development of the skill. The *pCallBack* is used for transmitting messages incase of failure or success of

the skills; Also using the parameter *pSimulative* to differentiate between a skill or a helper skill.

At the begging of each skill it is mandatory to initialize the servers (robot-arm sever, force torque sensor, manipulator server, virtual reality model server), depending of witch you use, through the next section:

```
// convert hardware server to robot arm hardware server

      CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::Co
      nvert(
                 pManipulator,
                 m_HardwareServers[pManipulator],
                 pRobotarmHardwareServer
            );

// manage state of robot arm hardware server

CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::ManageSt
ate(pManipulator, pRobotarmHardwareServer);
```

```
   // convert hardware server to FT sensor hardware server

      CManageHardwareServer<HardwareLayer::CFTSensorHardwareServer>::Co
nvert(pFTSensor,
      m_HardwareServers[pFTSensor],
      pFTSensorHardwareServer
      );

// manage state of FT sensor hardware server

      CManageHardwareServer<HardwareLayer::CFTSensorHardwareServer>::Ma
nageState(pFTSensor, pFTSensorHardwareServer);


// convert skill server to manipulator skill server
CManageSkillServer<SkillLayer::CManipulatorSkillServer>::Convert
 (SkillServerNames::MANIPULATOR,
                  m_SkillServers[SkillServerNames::MANIPULATOR],
                  pManipulatorSkillServer

);

// convert data server to MVR server
 ManageDataServer<SubSymbolicLayer::CMVRServer>::Convert(pManipulator,

                                          m_DataServers[pManipulator],

                                                      pMVRServer);
```

After the initialization of the servers we get the location, meaning position and orientation of the object (in our case the book) using the next line, which is then given to a location variable:

```
      TempLocation=ReactiveLayer::CWorldModelExtractorComplexType
      <SubSymbolicLayer::CLocation>::GetData(m_WorldModelServer,
      pObjLocation);
```

Then after setting the orientation of the gripper we create a temporary location 7 cm above the book, which is then set in the World Model server:

```
      m_WorldModelServer->SetData("PreFrame.EEL.Frame.Temp",
                                  SubSymbolicData);
```

This location is used as a parameter in the instruction PlanAndMoveGripperToLocation(), instruction that will move the manipulator to a certain location, which is given as an input parameter. Other parameters that this instruction has can determine the speed of the manipulator (normal, slow) or the type of movement (direct line). All of the above will be exemplify in the instruction below:

```
pManipulatorSkillServer->PlanAndMoveGripperToLocation(pManipulator,
                        "PreFrame.EEL.Frame.Temp",
                    "NormalSlow.EEL.Par.SDB",CallBackHelperProxy);
```

For the object, in our case the book, not to be considered an obstacle we need to set it to approach mode, which is done with this instruction:

```
pMVRServer->SetApproach(iLink, pObjectName);
```

where *iLink* represents with which link we touch the object, in our case 7 (this depending on the number of links of the robot). And the other parameter represents the object name.

To approach the book we use the instruction *PlaceOnPlatform(),* which uses the force torque sensor. The motion ends only if one the next two condition is satisfied, we reach the desired location or a certain threshold of the pressing force is reached. The threshold is set by this constant:

```
const double            nFORCE_THRESHOLD_WORLD(4);  // value in Newton
```

which is then compared with the real force obtained from the force torque sensor.

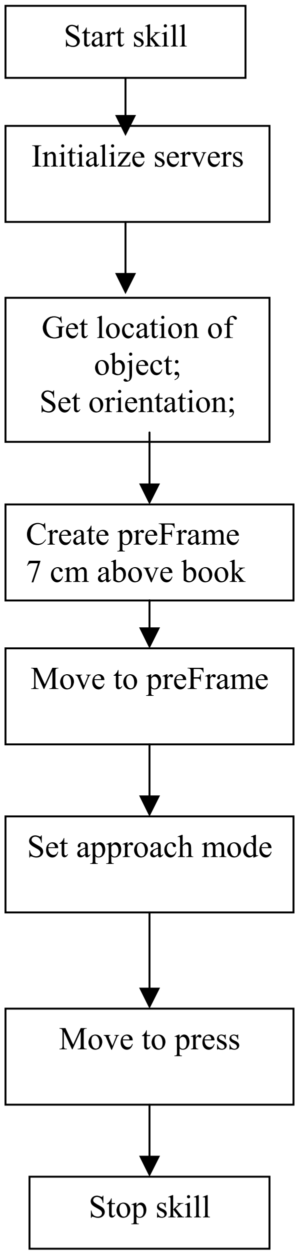
After calling this helper skill, we wait for its completion and then reset the approach mode for the book and delete any temporary data from the World model server that we created. In our case for example:

```
pMVRServer->ResetApproach();

m_WorldModelServer->DeleteData("PreFrame.EEL.Frame.Temp");
```

Before we end this skill we deal with the main errors throughout the skill using the coupled instruction : *throw {}….catch{}*

The logical diagram of this skill is:



3.4.2 MoveObjOnPLatform()

The parameters of this skill are:

- *pManipulator* is the name of the manipulator that will be used for grasping (here to derive gripper geometry).

- *pLocation* is the ID for the target object's location data in the World Model server.

- *pCallBack* is the pointer to the proxy of the callback object. See above for possible callback messages.

- *bSimulative* is the flag for the sequencer skill to be executed only simulative.

Sends a CM_EXECUTION_SUCCESS, if execution was successful. If an error occurs CM_EXECUTION_FAILURE.

If SM_STOP_SKILL is sent, the initial condition before skill execution will be restored and CM_TERMINATION_SUCCESS is sent back. On a SM_KILL_SKILL a CM_KILL_SUCCESS follows directly with no more actions.

What this skill does is to move the book to the first bar and then move slowly down, to scan the book bar code.

After we initialize the servers, using the same procedure, we get the location of the bar. The location of the bar is acquired with the help of the camera and imagine processing. For the testing par we use a location stored in the database: `"Book1.EEL.Loc.SDB".`

Before we move the book to the bar we must take into account that the book may be not in the correct position. The ‘ox’ and ‘oy’ of the book and the table are not parallel. This will affect the way we grasp the book.

To solve this we calculate the angle between the 'ox' axes of the book and the table and with the helper skill *CartesianDiret Control()* we rotate around the 'oz' axes with the angle in world coordinate system. For this to work we need to create a Cartesian command:

```
// Create Cartesian command
CartesianCommand.m_iTransX = 0; // +1, 0, or -1
CartesianCommand.m_iTransY = 0;
CartesianCommand.m_iTransZ = 0;
CartesianCommand.m_iRotX = 0;
CartesianCommand.m_iRotY = 0;
CartesianCommand.m_iRotZ = static_cast <long> ( sign(angle) );
CartesianCommand.m_dTransStep = 0.02;
//CartesianCommand.m_dRotStep = angle;
CartesianCommand.m_dJointSpeed=5;
CartesianCommand.m_bCoordinateSystem = false; //world coordinate system

// Add the data to the WorldModel
SubSymbolicData <<= CartesianCommand;
m_WorldModelServer->SetData("TEMP.CartesianControl.EEL.Cart.Temp",
SubSymbolicData);

// Call the skill
ManipulatorSkillServer>CartesianDirectControl("Robotarm","TEMP.Cartesia
nControl.EEL.Cart.Temp", CallBackHelperProxy);
```

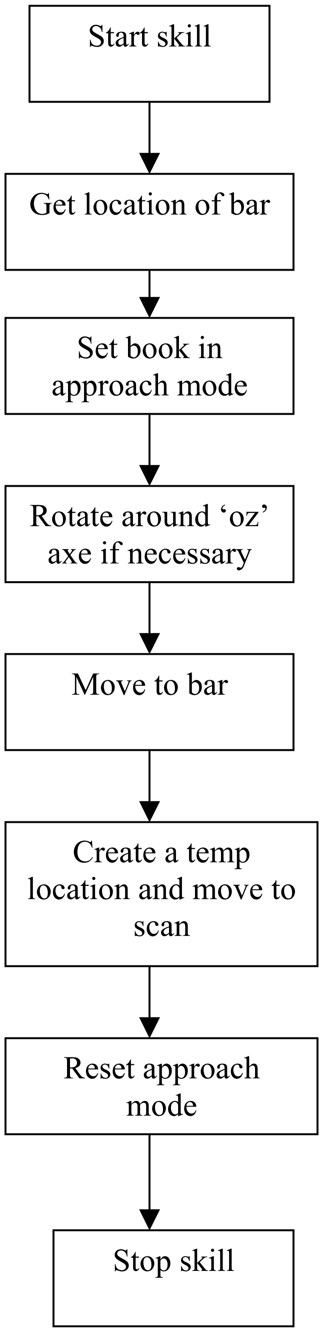
This idea was tested, but the results were not conclusive. So I encourage further development or to find another solution to solve the problem.

The next step is to move the book to the bar location using for this case a direct line movement:

```
pManipulatorSkillServer->PlanAndMoveGripperToLocation(pManipulator,
pLocation,"DirectLine.EEL.Par.SDB", CallBackHelperProxy);
```

After the location of the bar was reached, we create a temporary location to witch we move slowly thus scanning the code bar of the book.

The logic diagram of this skill is:


Start skill
Get location of bar
Set book in approach mode
Rotate around 'oz' axe if necessary
Move to bar
Create a temp location and move to scan
Reset approach mode
Stop skill

3.4.3 MoveBookToGrasp()

The parameters of this skill are:

- *pManipulator* is the name of the manipulator that will be used for grasping (here to derive gripper geometry).

- *pLocation* is the ID for the target object's location data in the World Model server.

- *pObjSize* is the ID for the target object's location data in the World Model server. (not yet used)

- *pCallBack* is the pointer to the proxy of the callback object. See above for possible callback messages.

- *bSimulative* is the flag for the sequencer skill to be executed only simulative.

Sends a CM_EXECUTION_SUCCESS, if execution was successful. If an error occurs CM_EXECUTION_FAILURE. If SM_STOP_SKILL is sent, the initial condition before skill execution will be restored and CM_TERMINATION_SUCCESS is sent back. On a SM_KILL_SKILL a CM_KILL_SUCCESS follows directly with no more actions.

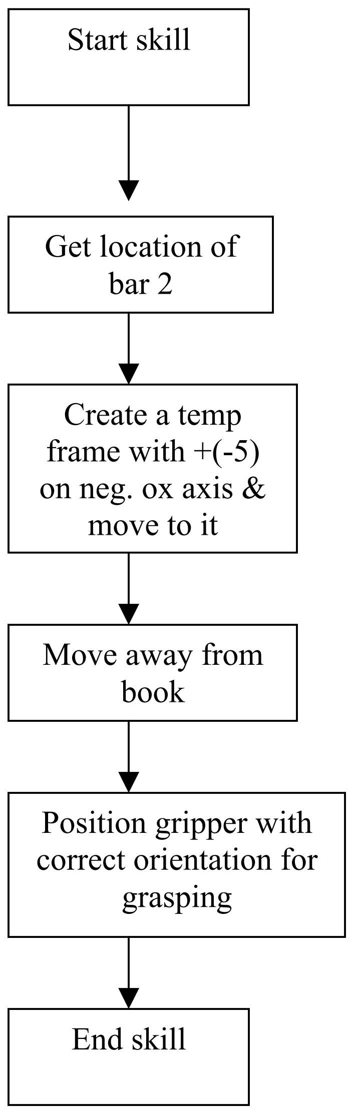
The sequence of event is almost the same as in the previous skills. After the initialization of the servers, we acquire the location of the second bar.

Using the same helper skill *PlanAndMoveGrippeToLocation( )* we move the book towards the edge of the table. Actually the temporary frame is with -5 cm more on the negative ‘ox’ axe, making the book ready for grasping.

After positioning the book the gripper, using created temporary frames, away from the book and with the correct orientation for grasping.

Before the skill is over we cleaned the World Model server of all the temporary data created and dealt with the major error that may occur using the *throw….catch* protective programming.

The logical diagram of this skill is:


Start skill
Get location of bar 2
Create a temp frame with +(-5) on neg. ox axis & move to it
Move away from book
Position gripper with correct orientation for grasping
End skill

3.5 MVR picture

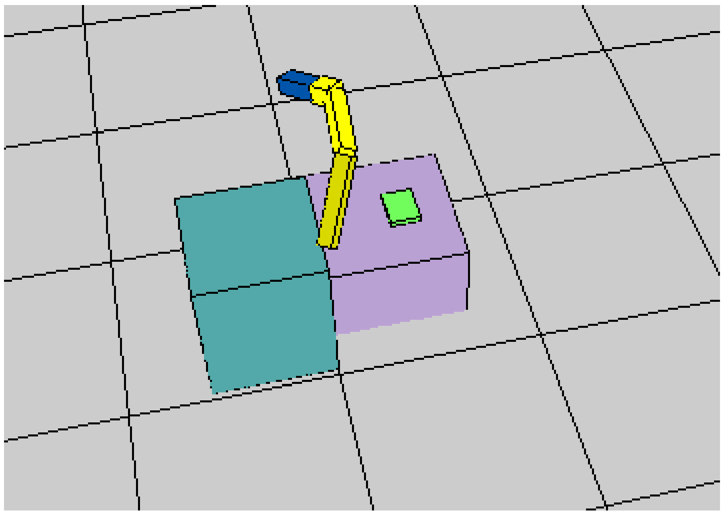


Fig. 20 MVR picture of the robot

In the picture above an MVR model of the robot and the experiment layout is presented. The motion planning is here, in the MVR, first planned and tested, and only if the configuration of the robot arm found is acceptable, then the robot arm moves. An acceptable configuration means that the point reached is not a singularity, the joints are not at their limits and no collision is detected.

The reasons why the singularities of a manipulator must be avoided are listed below:

• Singularities represent configurations at which mobility of the structure is reduced;

• When the structure is at a singularity, infinite solutions to inverse kinematics problem may exist;

• In the neighborhood of a singularity, small velocities in the operational space may cause large velocities in the joint space.

4. Conclusion and Future Work

After completing the experimental phase several conclusions have been reached. Firstly, the surface of the table should be laminated or have a slippery surface, as long as it offers less friction. Furthermore, the surface should be clean. Any debris will attach itself to the book cover, thus preventing a smooth motion.

Secondly, the book should have a hard cover. During the experiment we observed that if we grasp a book with a soft cover, the book tends to be unstable. Also when we press and try to move the book the cover wrinkles and there is a possibility that it would tear up.

As for future work I recommend further improvements of the library scenario by adding other actions and the part that deals with the alignment of the book to the table, so that the axes oy and ox will be parallel.

REFERENCES

[1] Prof. Dr.-Ing. A. Gräser ,“Robotics I: Lecture notes”, IAT, Summer semester 2007

[2] D. Ojdanic, O. Ivlev, A. Gräser: "A New Fast Motion Planning Approach for Dexterous Manipulators in 3D-Cartesian Space"; ISR-Robotik Joint conference on robotics, May 15-17, Munich, Germany; 2006

[3] www.iat.uni-bremen.de

[4] www.wikipedia.com

[5] www2.iat.uni-bremen.de/~friend2/iatWiki

APPENDIX A

(programs in ManipulatorSkillServer.cpp)

```
void CManipulatorSkillServer_impl::MoveToObjectAndPress(const char*
pManipulator,

            const char*        pFTSensor,
                                                       const char*
pPlatfLocation,
                                                       const char*
pPlatfSize,
                                                       const char*
pObjLocation,
                                                       const char*
pObjSize,
                                                       CCallBack_ptr
pCallBack,
                                                       CORBA::Boolean
bSimulative
                                                       ) throw
(CORBA::SystemException)

    {
        CCallBack_impl                        CallBackHelperServant;
        CCallBack_var                        CallBackHelperProxy;
        CORBA::Any                            SubSymbolicData;
        SubSymbolicLayer::CFrame             TempPreFrame;
        //SubSymbolicLayer::CFrame             TableFrame;
        SubSymbolicLayer::CFrame                oFrame;
        SubSymbolicLayer::COrientation         GFrame_ori;
        CKMatrix                        oGripperFrame(4, 4);
        //SubSymbolicLayer::CPosition             TempPosition;
        SubSymbolicLayer::CLocation            TempLocation;
        //CKMatrix                            CKTable(4,4);
        CKMatrix                             CKPreFrame(4,4);
        CKMatrix                             CKBook(4,4);
        CKMatrix                             CKGoal(4,4);
        const char*                             pObjectName;
        std::string
      ErrorMessage;
        std::string                             TempString;
        long                                     iLink;
        bool                                 bError(false);
        //const double
nMOVE_DOWN_DISTANCE(0.03);     // value in meter
        HardwareLayer::CRobotarmHardwareServer_ptr
pRobotarmHardwareServer;
        SkillLayer::CManipulatorSkillServer_ptr
pManipulatorSkillServer;
        HardwareLayer::CFTSensorHardwareServer_ptr
      pFTSensorHardwareServer;
        SubSymbolicLayer::CMVRServer_ptr            pMVRServer;
        //double                                            dAngle;
        //bool
      bForceReached = false;

        LogEvent("Skill MoveToObjectAndPress() started");

      try
        {
            // if true, then execute in simulation mode
            if (bSimulative)
            {
                //SimulateSkill(pManipulator, pCallBack);
                // Can we simulate something useful?
            }
            else
            {
                // EXECUTION
                // ----------------------------------------------------
--------
                // check, if robot world model exists

                // convert hardware server to robot arm hardware server

CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::Convert(
pManipulator,

m_HardwareServers[pManipulator],

pRobotarmHardwareServer

);
         // manage state of robot arm hardware server

CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::ManageSt
ate(pManipulator, pRobotarmHardwareServer);

                        LogEvent("Convertion of pFTSensorHardwareServer
started");

                // convert hardware server to FT sensor hardware server

      CManageHardwareServer<HardwareLayer::CFTSensorHardwareServer>::Co
nvert(pFTSensor,

m_HardwareServers[pFTSensor],

pFTSensorHardwareServer);

LogEvent("Convertion of pFTSensorHardwareServer finished");

// manage state of FT sensor hardware server
      CManageHardwareServer<HardwareLayer::CFTSensorHardwareServer>::Ma
nageState(pFTSensor, pFTSensorHardwareServer);
```

```
// convert skill server to manipulator skill server
CManageSkillServer<SkillLayer::CManipulatorSkillServer>::Convert(SkillS
erverNames::MANIPULATOR,

m_SkillServers[SkillServerNames::MANIPULATOR],

pManipulatorSkillServer

);

 // convert data server to MVR server
 CManageDataServer<SubSymbolicLayer::CMVRServer>::Convert(pManipulator,

m_DataServers[pManipulator],

pMVRServer);

                if (CORBA::is_nil(m_WorldModelServer))
                {
                    // throw internal exception
                    throw (CM_EXECUTION_FAILURE);

                } // if (CORBA::is_nil(m_pWorldModel))

              //Get location from world model (Book)

TempLocation=ReactiveLayer::CWorldModelExtractorComplexType<SubSymbolic
Layer::CLocation>::GetData(m_WorldModelServer, pObjLocation);

                // create orientation of the gripper
                GFrame_ori.m_dRotationX = 0;
                        GFrame_ori.m_dRotationY = 140.0;
                        GFrame_ori.m_dRotationZ = 0;

                TempLocation.m_Orientation = GFrame_ori;

                //Conv. location to frame
                TempPreFrame <<= TempLocation;

                // Change z-axis
                TempPreFrame[2][3]  = TempPreFrame[2][3]  + 0.07;

                //Conv CFrame to CKMatrix
                CKBook <<= TempPreFrame;

                        LogEvent("PreFrame.EEL.Frame.Temp: ");
                        LogFrame(CKPreFrame);

                        LogEvent("CKBook frame: ");
                        LogFrame(CKBook);

                        // Calc. preframe matrix
                CKPreFrame=CKBook;

                //Conv CKMatrix to Frame
                TempPreFrame<<=CKPreFrame;
```

```
                        LogEvent("PreFrame.EEL.Frame.Temp: ");
                        LogFrame(CKPreFrame);

                // Add this frame in the WorldModel
                SubSymbolicData <<= TempPreFrame;
                m_WorldModelServer->SetData("PreFrame.EEL.Frame.Temp",
SubSymbolicData);

                  // start the helper call back servant
                        ActivateServant(CallBackHelperServant,
CallBackHelperProxy);

                // Call helper skill
                pManipulatorSkillServer-
>PlanAndMoveGripperToLocation(pManipulator, "PreFrame.EEL.Frame.Temp",
                             "NormalSlow.EEL.Par.SDB",
CallBackHelperProxy);

                  WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);

                // Get the name of the object
                TempString = SubSymbolicLayer::GetNameString(pObjSize);
                pObjectName = TempString.c_str();

                // getting the actual numbers of links
                 iLink = 7 ;//GetNumberOfLinks();

                // Set ApproachMode() for the platform
                pMVRServer->SetApproach(iLink, pObjectName);

            // start the helper call back servant
                        ActivateServant(CallBackHelperServant,
CallBackHelperProxy);

                // Call skill to place an object on platform with FTS
                        pManipulatorSkillServer-
>PlacеOnPlatform(pManipulator, pFTSensor, "Book1.EEL.SCub.Temp",
                                        pObjLocation,
CallBackHelperProxy, false);

                  WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);
                //................end movement ..above the
book.........
                // Reset MVR from Approach mode
                  pMVRServer->ResetApproach();

                  // Delete data used for Frame
                  //m_WorldModelServer-
>DeleteData("TEMP.GripperFrame.EEL.Frame.Temp");
                 m_WorldModelServer-
>DeleteData("PreFrame.EEL.Frame.Temp");
```

```
            }//else (bSimulative)

        }//try
        catch (const SubSymbolicLayer::CMVRServerError& Error)
            {
                  // Create and log error message
                  ErrorMessage = "MVR Error: ";
                  ErrorMessage.append(Error.m_pErrorMessage);
                  LogEvent(ErrorMessage);

                  // send callback error message via internal method
                  SendCallBack(pCallBack, CM_EXECUTION_FAILURE);

                  // set error flag to true
                  bError = true;
            }
        catch (const INTERNAL_EXCEPTION_T& Error)
            {
                  // log error message
                  LogEvent(Error.second);

                  // send callback error message via internal method
                  SendCallBack(pCallBack, Error.first);

                  // set error flag to true
                  bError = true;

            } // catch (const INTERNAL_EXCEPTION_T& Error)
        catch (const ReactiveLayer::CWorldModelExchange::CException&
Exception)
            {
                  // log error message
                  ErrorMessage  = "Error in world model exchange in
MoveToObjectAndPress(): ";
                  ErrorMessage += Exception.m_eException;
                  ErrorMessage += " with ID: ";
                  ErrorMessage += Exception.m_ID;
                  LogEvent(ErrorMessage, LoggingLayer::ERROR_MSG);

                  // throw internal exception
                  SendCallBack(pCallBack, CM_EXECUTION_FAILURE);

            } // catch (const
ReactiveLayer::CWorldModelExchange::CException& Exception)

        catch (const HardwareLayer::CFTSensorHardwareError&)
        {
            LogEvent("CFTSensorHardwareError error in
MoveToObjectAndPress()");

            // send callback error message
            SendCallBack(pCallBack, CM_EXECUTION_FAILURE);
        }
        catch (const CORBA::SystemException&)
        {
                  // send callback error message
            SendCallBack(pCallBack, CM_EXECUTION_FAILURE);
        }
        catch (const std::string& cMessage)
        {
            // send callback error message via internal method
            SendCallBack(pCallBack, cMessage);

            // set error flag to true
            bError = true;
        }

        if( !bSimulative )
        {

            // Check, if error occurred.
            if (!bError) {
                // Send "Success" callback message via internal method.
                SendCallBack(pCallBack, CM_EXECUTION_SUCCESS);
            }
            else {
                // Send "Failure" callback message via internal method.
                SendCallBack(pCallBack, CM_EXECUTION_FAILURE);
            }
        }

      LogEvent("Skill MoveToObjectAndPress() terminated");

   } // // void CManipulatorSkillServer_impl::MoveToObjectAndPress(...)


    void CManipulatorSkillServer_impl::MoveObjOnPlatform(const char*
pManipulator,
                                                       const char*
pLocation,
                                                       CCallBack_ptr
pCallBack,
                                                       CORBA::Boolean
bSimulative
                                                       ) throw
(CORBA::SystemException)
   {
        bool                                            bError(false);
        CCallBack_impl
CallBackHelperServant;
        CCallBack_var
CallBackHelperProxy;
        CORBA::Any
SubSymbolicData;
        SubSymbolicLayer::CFrame
TempPreFrame;
        SubSymbolicLayer::CLocation
TempLocation;
        SubSymbolicLayer::CLocation
TempLocation1;
        //SubSymbolicLayer::COrientation              GFrame_ori;
        CKMatrix
CKBar(4,4);
        CKMatrix                                              CKNxFrame(4,4);
        std::string
ErrorMessage;
        HardwareLayer::CRobotarmHardwareServer_ptr
pRobotarmHardwareServer;
        SkillLayer::CManipulatorSkillServer_ptr
pManipulatorSkillServer;
        SubSymbolicLayer::CMVRServer_ptr                    pMVRServer;
        const char*                                           pObjectName;
        std::string                                         TempString;
        long                                                  iLink;
        double
angle,angle1,angle2;
        SubSymbolicLayer::CCartesianCommand     CartesianCommand;
        SubSymbolicLayer::COrientation          tmp_ori;

        LogEvent("Skill MoveObjOnPlatform() started");

     try
        {
            // if true, then execute in simulation mode
            if (bSimulative)
            {
                //SimulateSkill(pManipulator, pCallBack);
                // Can we simulate something useful?
            }
            else
            {

                // EXECUTION
                // ----------------------------------------------------
--------
                // ------------ check, if needed servers are avaiable -
------------
                // convert hardware server to robot arm hardware server

CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::Convert(
pManipulator,

m_HardwareServers[pManipulator],

pRobotarmHardwareServer

);

                // manage state of robot arm hardware server

CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::ManageSt
ate(pManipulator, pRobotarmHardwareServer);

                // convert skill server to manipulator skill server

CManageSkillServer<SkillLayer::CManipulatorSkillServer>::Convert(SkillS
erverNames::MANIPULATOR,
m_SkillServers[SkillServerNames::MANIPULATOR],

pManipulatorSkillServer

);

 // convert data server to MVR server
 CManageDataServer<SubSymbolicLayer::CMVRServer>::Convert(pManipulator,

m_DataServers[pManipulator],

pMVRServer

);

                // check, if robot world model exists
                if (CORBA::is_nil(m_WorldModelServer))
                {
                    // throw internal exception
                    throw (CM_EXECUTION_FAILURE);

                } // if (CORBA::is_nil(m_pWorldModel))

               //Get location from world model (Book)
                TempLocation =
ReactiveLayer::CWorldModelExtractorComplexType<SubSymbolicLayer::CLocat
ion>::GetData(m_WorldModelServer, pLocation);

               tmp_ori=TempLocation.m_Orientation;
               angle1=tmp_ori.m_dRotationX;

               // Get the name of the object
                TempString =
SubSymbolicLayer::GetNameString("Book1.EEL.Loc.SDB");
                pObjectName = TempString.c_str();

                // getting the actual numbers of links
                 iLink = 7 ;//GetNumberOfLinks();

                // Set ApproachMode() for the platform
                pMVRServer->SetApproach(iLink, pObjectName);

                // Get location of platform (Table)
                  TempLocation1 =
ReactiveLayer::CWorldModelExtractorComplexType<SubSymbolicLayer::CLocat
ion>::GetData(m_WorldModelServer, "Platform1.EEL.Loc.SDB");

               tmp_ori=TempLocation1.m_Orientation;
               angle2 =tmp_ori.m_dRotationX;

               angle = angle2 - angle1;

                 //angle = TempLocation1.m_Orientation.m_dRotationX -
TempLocation.m_Orientation.m_dRotationX;
```

```
                cout << "The orientation differece is: "<<angle <<
std::endl;

                if (angle !=0)
                  {
                        if(angle > 90)
                         {
                              CartesianCommand.m_dRotStep = abs(angle);
                         }
                         else
                      {
                        CartesianCommand.m_dRotStep = angle;
                           }
                        // Create Cartesian command
                              CartesianCommand.m_iTransX = 0; // +1, 0,
or -1
                              CartesianCommand.m_iTransY = 0;
                              CartesianCommand.m_iTransZ = 0;
                              CartesianCommand.m_iRotX = 0;
                              CartesianCommand.m_iRotY = 0;
                              CartesianCommand.m_iRotZ = static_cast
<long> ( sign(angle) );
                              CartesianCommand.m_dTransStep = 0.02;
                              //CartesianCommand.m_dRotStep = angle;
                              CartesianCommand.m_dJointSpeed = 5;
                              CartesianCommand.m_bCoordinateSystem =
false; // world coordinate system

                              // Add the data to the WorldModel
                              SubSymbolicData <<= CartesianCommand;
                              m_WorldModelServer-
>SetData("TEMP.CartesianControl.EEL.Cart.Temp", SubSymbolicData);

                               // Start the helper call back servant
                  ActivateServant(CallBackHelperServant,
CallBackHelperProxy);

// Call the skill
  pManipulatorSkillServer->
CartesianDirectControl("Robotarm","TEMP.CartesianControl.EEL.Cart.Temp"
, CallBackHelperProxy);

                   // Wait on the helper skill
                  WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);
                  }

                // Start the helper call back servant
                ActivateServant(CallBackHelperServant,
CallBackHelperProxy);

                // Call helper skill
                  pManipulatorSkillServer-
>PlanAndMoveGripperToLocation(pManipulator, pLocation,
                               "DirectLine.EEL.Par.SDB",
CallBackHelperProxy);
```

```
                // Wait on the helper skill
                WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);

                //Convert loc. to frame
                TempPreFrame <<= TempLocation;

                //Conv CFrame to CKMatrix
                CKBar <<= TempPreFrame;

                LogEvent("CKBar _1_1: ");
                        LogFrame(CKBar);

                // Change x-axis
                TempPreFrame[0][3] = TempPreFrame[0][3] - 0.05;
                  TempPreFrame[2][3] = TempPreFrame[2][3] + 0.02;

                //Conv CFrame to CKMatrix
                CKNxFrame <<= TempPreFrame;

                LogEvent("CKNxFrame _2_2: ");
                        LogFrame(CKNxFrame);

                //Conv. CKMatrix to CFrame
                TempPreFrame<<= CKNxFrame;


                // Add this frame in the WorldModel
                SubSymbolicData <<= TempPreFrame;
                m_WorldModelServer->SetData("PreFrame.EEL.Frame.Temp",
SubSymbolicData);


                // Move to next location
                pManipulatorSkillServer-
>PlanAndMoveGripperToLocation(pManipulator,

"PreFrame.EEL.Frame.Temp","NormalSlow2.EEL.Par.SDB",CallBackHelperProxy
);

              // Wait on the helper skill
                WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);

             }//else (bSimulative)

                  // reset approch for the book
                  pMVRServer->ResetApproach();

            // Delete data used for Frame
                  m_WorldModelServer->
DeleteData("PreFrame.EEL.Frame.Temp");

          }//try
          catch (const INTERNAL_EXCEPTION_T& Error)
            {
                  // log error message
                  LogEvent(Error.second);

                  // send callback error message via internal method
                  SendCallBack(pCallBack, Error.first);

                  // set error flag to true
                  bError = true;

            } // catch (const INTERNAL_EXCEPTION_T& Error)

        catch (const ReactiveLayer::CWorldModelExchange::CException&
Exception)
            {
                  // log error message
                  ErrorMessage  = "Error in world model exchange in
MoveToObjectAndPress(): ";
                  ErrorMessage += Exception.m_eException;
                  ErrorMessage += " with ID: ";
                  ErrorMessage += Exception.m_ID;
                  LogEvent(ErrorMessage, LoggingLayer::ERROR_MSG);

                  // throw internal exception
                  SendCallBack(pCallBack, CM_EXECUTION_FAILURE);

            } // catch (const
ReactiveLayer::CWorldModelExchange::CException& Exception)

        catch (const std::string& Error)
        {
            LogEvent("Internal exception in MoveObjOnPlatform()");

            // send callback error message via internal method
            SendCallBack(pCallBack, Error);

            // set error flag to true
            bError = true;

        } // catch (const SkillError& Error)

        // check, if error occurred
        if (!bError)
        {
            // send success callback message via internal method
            SendCallBack(pCallBack, CM_EXECUTION_SUCCESS);
        }

        LogEvent("Skill MoveObjOnPlatform() terminated");

   }//void CManipulatorSkillServer_impl::MoveObjOnPlatform()
   void CManipulatorSkillServer_impl::MoveBookToGrasp(const char*
pManipulator,
                                                       const char*
pLocation,
                                                       const char*
pObjSize,
                                                       CCallBack_ptr
pCallBack,
                                                       CORBA::Boolean
bSimulative
                                                       ) throw
(CORBA::SystemException)
   {
        bool                                    bError(false);
        CCallBack_impl                            CallBackHelperServant;
        CCallBack_var                            CallBackHelperProxy;
        CORBA::Any                                SubSymbolicData;
        SubSymbolicLayer::CFrame                TempPreFrame;
        SubSymbolicLayer::CLocation                 TempLocation;
        //SubSymbolicLayer::CLocation               RLoc1 ;
        //SubSymbolicLayer::CLocation              RLoc2 ;
        //SubSymbolicLayer::CLocation              RLoc3 ;
        //SubSymbolicLayer::CPosition                     tmp_pos;
            //SubSymbolicLayer::COrientation              tmp_ori;
        SubSymbolicLayer::COrientation            G_ori;
        //CKMatrix                                  CKRotx(4,4);
        //CKMatrix                                  CKRoty(4,4);
        //CKMatrix                                  CKRotz(4,4);
        CKMatrix                                CKFrame1(4,4);
        std::string
      ErrorMessage;
        //std::string                                 TempString;
        const ACE_Time_Value                   oCHECK_CALLBACK_CYCLE(0, 100
* 1000); //100ms
        HardwareLayer::CRobotarmHardwareServer_ptr
pRobotarmHardwareServer;
        SkillLayer::CManipulatorSkillServer_ptr
pManipulatorSkillServer;
        SubSymbolicLayer::CMVRServer_ptr               pMVRServer;
        const char*                                       pObjectName;
        std::string                                     TempString;
        long                                              iLink;

        LogEvent("Skill MoveBookToGrasp() started");

      try
        {
            // if true, then execute in simulation mode
            if (bSimulative)
            {
                SimulateSkill(pManipulator, pCallBack);
                // Can we simulate something useful?
            }
            else
            {
                // EXECUTION
                // --------------------------------------------------------
--------
          // ------------ check, if needed servers are available ------
-------
                // convert hardware server to robot arm hardware server

CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::Convert(
pManipulator,
m_HardwareServers[pManipulator],
pRobotarmHardwareServer);

                // manage state of robot arm hardware server
CManageHardwareServer<HardwareLayer::CRobotarmHardwareServer>::ManageSt
ate(pManipulator, pRobotarmHardwareServer);

                // convert skill server to manipulator skill server
CManageSkillServer<SkillLayer::CManipulatorSkillServer>::Convert(SkillS
erverNames::MANIPULATOR,
m_SkillServers[SkillServerNames::MANIPULATOR],
pManipulatorSkillServer);

                // convert data server to MVR server
CManageDataServer<SubSymbolicLayer::CMVRServer>::Convert(pManipulator,
m_DataServers[pManipulator],
pMVRServer);
                // check, if robot world model exists
                if (CORBA::is_nil(m_WorldModelServer))
                {
                    // throw internal exception
                    throw (CM_EXECUTION_FAILURE);

                } // if (CORBA::is_nil(m_pWorldModel))

                // Get the name of the object
                TempString =
SubSymbolicLayer::GetNameString("Book1.EEL.Loc.SDB");
                pObjectName = TempString.c_str();

                // getting the actual numbers of links
                 iLink = 7 ;//GetNumberOfLinks();

                // Set ApproachMode() for the platform
                pMVRServer->SetApproach(iLink, pObjectName);

                //Get location from world model (Bar2)
TempLocation=ReactiveLayer::CWorldModelExtractorComplexType<SubSymbolic
Layer::CLocation>::GetData(m_WorldModelServer, pLocation);

                //give orientation
                G_ori.m_dRotationX = 0;
                G_ori.m_dRotationY = 140.0;
                G_ori.m_dRotationZ = 0;

                TempLocation.m_Orientation = G_ori;

                //Convert loc. to frame
                TempPreFrame <<= TempLocation;

                // Change x-axis
                TempPreFrame[0][3] = TempPreFrame[0][3] - 0.06;
                TempPreFrame[2][3] = TempPreFrame[2][3] + 0.012;

                //Conv CFrame to CKMatrix
                CKFrame1 <<= TempPreFrame;

                //Conv. CKMatrix to CFrame
                TempPreFrame<<=CKFrame1;

                // Add this frame in the WorldModel
                SubSymbolicData <<= TempPreFrame;
                m_WorldModelServer->SetData("PreFrame.EEL.Frame.Temp",
SubSymbolicData);

                        // Start the helper call back servant
                        ActivateServant(CallBackHelperServant,
CallBackHelperProxy);

                // Call helper skill
                pManipulatorSkillServer-
>PlanAndMoveGripperToLocation(pManipulator, "PreFrame.EEL.Frame.Temp",
                               "Directline.EEL.Par.SDB",
CallBackHelperProxy);

               // Wait on the helper skill
                WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);

               LogEvent("!!!!!move to bar 2 succes!!!!1 ");

                //1

                //give orientation
                G_ori.m_dRotationX = 0;
                        G_ori.m_dRotationY = 140.0;
                        G_ori.m_dRotationZ = 0;

                TempLocation.m_Orientation = G_ori;

                //Convert loc. to frame
                TempPreFrame <<= TempLocation;

                // Change z-axis(move gripper above the book)--bar
coord

                TempPreFrame[0][3] = TempPreFrame[0][3] - 0.17;
                TempPreFrame[2][3] = TempPreFrame[2][3] + 0.14;

                //Conv CFrame to CKMatrix
                CKFrame1 <<= TempPreFrame;

                //Conv. CKMatrix to CFrame
                TempPreFrame<<=CKFrame1;
                // Add this frame in the WorldModel
                SubSymbolicData <<= TempPreFrame;
                m_WorldModelServer->SetData("PreFrame1.EEL.Frame.Temp",
SubSymbolicData);

                // Call helper skill
               pManipulatorSkillServer-
>PlanAndMoveGripperToLocation(pManipulator, "PreFrame1.EEL.Frame.Temp",
                                "NormalSlow.EEL.Par.SDB",
CallBackHelperProxy);

                        // Wait on the helper skill
                WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);

               LogEvent("!!!!move up succes!!!!1 ");

                         // reset approch for the book
                        pMVRServer->ResetApproach();

                //3

              /*
                        tmp_pos.m_dPositionX = 0;
                        tmp_pos.m_dPositionY = 0;
                        tmp_pos.m_dPositionZ = 0;

                        tmp_ori.m_dRotationX = -20;
                        tmp_ori.m_dRotationY = 0;
                        tmp_ori.m_dRotationZ = 0;

                        RLoc1.m_Position = tmp_pos;
                        RLoc1.m_Orientation = tmp_ori;

                //Convert loc. to frame
                TempPreFrame <<= RLoc1;

                //Conv CFrame to CKMatrix
                CKRotx <<= TempPreFrame;

                LogEvent("!!!!  CKRotx   _!!!: ");
                        LogFrame(CKRotx);

                tmp_pos.m_dPositionX = 0;
                        tmp_pos.m_dPositionY = 0;
                        tmp_pos.m_dPositionZ = 0;

                        tmp_ori.m_dRotationX =0 ;
                        tmp_ori.m_dRotationY =-10;
                        tmp_ori.m_dRotationZ = 0;

                RLoc2.m_Position = tmp_pos;
                        RLoc2.m_Orientation = tmp_ori;

                //Convert loc. to frame
                TempPreFrame <<= RLoc2;
                //Conv CFrame to CKMatrix
                CKRoty <<= TempPreFrame;

                tmp_pos.m_dPositionX = 0;
                        tmp_pos.m_dPositionY = 0;
                        tmp_pos.m_dPositionZ = 0;

                        tmp_ori.m_dRotationX =0 ;
                        tmp_ori.m_dRotationY =0;
                        tmp_ori.m_dRotationZ =-20;

                RLoc3.m_Position = tmp_pos;
                        RLoc3.m_Orientation = tmp_ori;

                //Convert loc. to frame
                TempPreFrame <<= RLoc3;

                //Conv CFrame to CKMatrix
                CKRotz <<= TempPreFrame;
                */

                //give orientation
                G_ori.m_dRotationX = -40;
                        G_ori.m_dRotationY = 120.0;
                        G_ori.m_dRotationZ = -40;

                TempLocation.m_Orientation = G_ori;

                 //Convert loc. to frame
                TempPreFrame <<= TempLocation;

                 /*
                 // Get actual GripperFrame
                  //  SubSymbolicLayer::CFrame GFrame;
                    TempPreFrame= GetGripperFrame(pManipulator,
                                                 oCHECK_CALLBACK_CYCLE);

                         // Convert CFrame into CKMatrix.
                 CKMatrix oGripperFrame(4, 4);
                 oGripperFrame <<= TempPreFrame;

                 LogEvent("!!!!  TempPreFrame   _1_1: ");
                         LogFrame(oGripperFrame);
                          */

                // Change x-axis( move gripper to a help point --
down)--bar coord
                //TempPreFrame <<= TempLocation;
                TempPreFrame[0][3]  = TempPreFrame[0][3]  - 0.23;
                TempPreFrame[2][3]  = TempPreFrame[2][3]  - 0.07;

                //Conv CFrame to CKMatrix
                CKFrame1 <<= TempPreFrame;

                  LogEvent("!!!! CKFrame1 !!  _2_2: ");
```

```
                        LogFrame(CKFrame1);

                // matrix  position for grasping
                // CKFrame1= CKFrame1*CKRotx*CKRoty*CKRotz;

               LogEvent("!!!! CKFrame1 final !!  _3_3: ");
                        LogFrame(CKFrame1);

                //Conv. CKMatrix to CFrame
                TempPreFrame<<=CKFrame1;

                // Add this frame in the WorldModel
                SubSymbolicData <<= TempPreFrame;
                m_WorldModelServer->SetData("PreFrame2.EEL.Frame.Temp",
SubSymbolicData);

                // Call helper skill
                pManipulatorSkillServer-
>PlanAndMoveGripperToLocation(pManipulator, "PreFrame2.EEL.Frame.Temp",
                               "NormalSlow.EEL.Par.SDB",
CallBackHelperProxy);

               // Wait on the helper skill
                WaitForHelperSkill(pCallBack, CallBackHelperServant,
CallBackHelperProxy);

                  }//else (bSimulative)

              // Delete data used for Frame
              m_WorldModelServer->
DeleteData("PreFrame.EEL.Frame.Temp");
              m_WorldModelServer->
DeleteData("PreFrame1.EEL.Frame.Temp");
              m_WorldModelServer->
DeleteData("PreFrame2.EEL.Frame.Temp");

                  // reset approch for the book
                  //pMVRServer->ResetApproach();

            }//try

            catch (const INTERNAL_EXCEPTION_T& Error)
            {
                  // log error message
                  LogEvent(Error.second);

                  // send callback error message via internal method
                  SendCallBack(pCallBack, Error.first);

                  // set error flag to true
                  bError = true;

            } // catch (const INTERNAL_EXCEPTION_T& Error)

            catch (const
ReactiveLayer::CWorldModelExchange::CException& Exception)
```

```
            {
                  // log error message
                  ErrorMessage  = "Error in world model exchange in
MoveToObjectAndPress(): ";
                  ErrorMessage += Exception.m_eException;
                  ErrorMessage += " with ID: ";
                  ErrorMessage += Exception.m_ID;
                  LogEvent(ErrorMessage, LoggingLayer::ERROR_MSG);

                  // throw internal exception
                  SendCallBack(pCallBack, CM_EXECUTION_FAILURE);

            } // catch (const
ReactiveLayer::CWorldModelExchange::CException& Exception)

        catch (const std::string& Error)
        {
            LogEvent("Internal exception in MoveBookToGrasp()");

            // send callback error message via internal method
            SendCallBack(pCallBack, Error);

            // set error flag to true
            bError = true;

        } // catch (const SkillError& Error)

        // check, if error occurred
        if (!bError)
        {
            // send success callback message via internal method
            SendCallBack(pCallBack, CM_EXECUTION_SUCCESS);

        }

        LogEvent("Skill MoveBookToGrasp() terminated");

         }//void CManipulatorSkillServer_impl::MoveBookToGrasp()
```

APPENDIX B

(program in ManipulatorSkillServerTestApp.cpp)

```
void TestGetBook()

{
    CORBA::Any                                          SymbolicData;
    CORBA::String_var                                   CallBackMessage;
    SubSymbolicLayer::CLocation                           TableLocation;
    SubSymbolicLayer::CSizeCuboid                       TableSize;
    SubSymbolicLayer::CLocation                             BookLocation;
    SubSymbolicLayer::CSizeCuboid                       BookSize;
    const ACE_Time_Value                           oCHECK_CALLBACK_CYCLE(0, 100 *
1000); //100ms
      long                                                      iLink;

    std::cout << "TestGetBook() started" << std::endl;

      // Location and size of the table and book into WorldModel
      //..

      // Goal orientation
      TableLocation.m_Orientation.m_dRotationX = 0;
      TableLocation.m_Orientation.m_dRotationY = 0;
      TableLocation.m_Orientation.m_dRotationZ = 0;

      TableLocation.m_Position.m_dPositionX = 0.6;
      TableLocation.m_Position.m_dPositionY = -0.30;
      TableLocation.m_Position.m_dPositionZ = -0.67;

      SymbolicData <<= TableLocation;
      WorldModel->SetData("Table.EEL.Loc.Temp", SymbolicData);

      // The size will allways be the same
      TableSize.m_dHeight = 0.5;
      TableSize.m_dDeepth = 0.70;
      TableSize.m_dWidth  = 0.80;

      SymbolicData <<= TableSize;
      WorldModel->SetData("Table.EEL.SCub.Temp", SymbolicData);

      //...

      // Goal orientation
      BookLocation.m_Orientation.m_dRotationX = 0;
      BookLocation.m_Orientation.m_dRotationY = 0;
      BookLocation.m_Orientation.m_dRotationZ = 0;

      BookLocation.m_Position.m_dPositionX = 0.6;
      BookLocation.m_Position.m_dPositionY = -0.45;
      BookLocation.m_Position.m_dPositionZ = -0.40;
      SymbolicData <<= BookLocation;
      WorldModel->SetData("Book1.EEL.Loc.Temp", SymbolicData);

      // The size will allways be the same
      BookSize.m_dHeight = 0.02;
      BookSize.m_dDeepth = 0.2;
      BookSize.m_dWidth  = 0.15;

      SymbolicData <<= BookSize;
      WorldModel->SetData("Book1.EEL.SCub.Temp", SymbolicData);

      // Call skill CalculateDynamicObstacles()
      // Call the skill
      ManipulatorSkillServer->CalculateDynamicObstacles("Robotarm",
CallBackProxy, false);

      // Wait for the termination
      while (!CallBackProxy->IsCallBackUpdated())
      {
            ACE_OS::sleep(ACE_Time_Value(0, 100000));
      }

      // get call back helper message
      CallBackProxy->GetCallBackValue(CallBackMessage);

      cout << CallBackMessage << endl;


      std::cout << "Test_CalculateDynamicObstacles() finished" <<
std::endl;

    // Call the skill
    ManipulatorSkillServer->MoveToObjectAndPress("Robotarm",
"FTSensor","Platform1.EEL.Loc.SDB","sdd",

      "Book1.EEL.Loc.SDB","Book1.EEL.SCub.Temp", CallBackProxy, false);

    // Wait for the termination
    while (!CallBackProxy->IsCallBackUpdated())
    {
        ACE_OS::sleep(ACE_Time_Value(0, 100000));
    }

    // get call back helper message
    CallBackProxy->GetCallBackValue(CallBackMessage);

    cout << "MoveToObjectAndPress() finished" << CallBackMessage <<
endl;

    if (CallBackMessage == SkillLayer::CM_EXECUTION_FAILURE)
    {
        cout << "TestGetBook finished" << std::endl;
        return;
    }
```

```
// Call the skill
    ManipulatorSkillServer-
>MoveObjOnPlatform("Robotarm","Bar1.EEL.Loc.SDB", CallBackProxy,
false);

    // Wait for the termination
    while (!CallBackProxy->IsCallBackUpdated())
    {
        ACE_OS::sleep(ACE_Time_Value(0, 100000));
    }

    // get call back helper message
    CallBackProxy->GetCallBackValue(CallBackMessage);

    cout << "MoveObjOnPlatform(): " << CallBackMessage << endl;

    if (CallBackMessage == SkillLayer::CM_EXECUTION_FAILURE)
    {
        cout << "TestGetBook finished" << std::endl;
        return;
    }

// Call the skill
    ManipulatorSkillServer-
>MoveBookToGrasp("Robotarm","Bar2.EEL.Loc.SDB","gfgf", CallBackProxy,
false);

    // Wait for the termination
    while (!CallBackProxy->IsCallBackUpdated())
    {
        ACE_OS::sleep(ACE_Time_Value(0, 100000));
    }

    // get call back helper message
    CallBackProxy->GetCallBackValue(CallBackMessage);

    cout << "MoveBookToGrasp(): " << CallBackMessage << endl;

    if (CallBackMessage == SkillLayer::CM_EXECUTION_FAILURE)
    {
        cout << "TestGetBook finished" << std::endl;
        return;
          }

      }// TestGetBook()
```