Interreg

EUROPEAN UNION

Grande Région | Großregion

Robotix-Academy

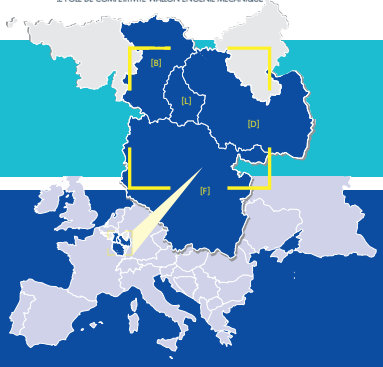
Fonds européen de développement régional | Europäischer Fonds für regionale Entwicklung

Axe prioritaire | Prioritätsachse 4
Compétitivité et attractivité
Wettbewerbsfähigkeit und Attraktivität

Robotix-Academy Summer School
September 03rd to 05th 2018
Université de Liège

Partenaires du projet | Projektpartner:

ZeMA · uni.lu Université du Luxembourg · Liège université · Université de Lorraine · Hochschule Trier Umwelt-Campus Birkenfeld · Pôle MecaTech



www.robotix.academy

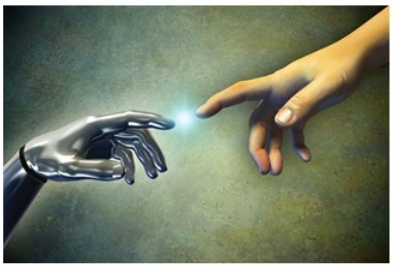
Vorwort

Die Robotix-Academy ist auf einem guten Weg ein internationaler Forschungscluster für industrielle Robotik und Mensch-Roboter-Kooperation zu werden. Das dritte Projektjahr wurde im Wesentlichen dazu genutzt, die Kooperationen mit der Industrie auszubauen und eine Verstetigung nach Projektende zu erreichen. Dafür wurden bereits mehrere neue Forschungsanträge eingereicht, u.a. ein MARIE SKŁODOWSKA-CURIE INNOVATIVE TRAINING NETWORKS, welches 2018 von allen Partnern gemeinsam erarbeitet worden ist.
Die Partner der Robotix-Academy haben ihre Portfolios mit Inhalten gefüllt und konnten die Academy über eine Vielzahl von Veranstaltungen in der Großregion einem breiten Publikum aus Industrie und Handwerk präsentieren. Die Marke „Robotix-Academy" konnte sich weiter etablieren, was die Präsenz auf den wichtigsten Fachmessen und Veranstaltungen 2018 zeigt. Die Veranstaltungsformate Wissenschaftskonferenz RACIR – Robotix-Academy Conference for Industrial Robotics – und Robotix-Academy Roadshow sind ausgereift und fester Bestandteil des Aktionsplans. Dieser ist mit Input der strategischen Partner sowie der Verbände und der Landesvertreter erstellt und sieht vor, die Bereiche Kommunikation, Forschung und Technologietransfer sowie den Aufbau des Beteiligungsnetzwerks in der Industrie-Robotik zu erweitern. Für den Bereich Kommunikation ist die Robotix-Academy hier bereits sehr erfolgreich: Im Herbst 2018 konnte sie sich auf der Interreg-Veranstaltung KapKomGR als Best Practice Beispiel präsentieren.
Die Projektpartner freuen sich über das wachsende Interesse an der Robotix-Academy und bedanken sich an dieser Stelle bei allen Beteiligten und Förderern für die gute Zusammenarbeit.

Préface

La Robotix-Academy est en voie de devenir un pôle de recherche international pour la robotique industrielle et la coopération homme-robot. La troisième année du projet a principalement servi à développer la coopération avec l'industrie et à assurer la continuité après la fin du projet. Plusieurs nouvelles propositions de recherche ont déjà été soumises, dont MARIE SKŁODOWSKA-CURIE INNOVATIVE TRAINING NETWORKS, qui a été développé conjointement par tous les partenaires.
Ceux-ci ont rempli leurs portefeuilles de contenu et ont pu présenter la Robotix-Academy à un large public de l'industrie et du commerce lors de nombreux événements dans la Grande Région. La marque «Robotix-Academy» a pu continuer à se positionner, comme en témoigne sa présence sur les salons et événements les plus importants en 2018. Les formats RACIR - Robotix-Academy Conference for Industrial Robotics - et Robotix-Academy Roadshow sont tous deux des événements matures qui font partie intégrante du plan d'action de l'Academy. Il a été élaboré avec la contribution de partenaires stratégiques, d'associations et de représentants nationaux et prévoit l'expansion des domaines de la communication, de la recherche et du transfert de technologie ainsi que la création d'un réseau de participation à la robotique industrielle. La Robotix-Academy connaît déjà un grand succès dans le domaine de la communication : En automne 2018, elle a pu se présenter comme un exemple de bonnes pratiques lors de l'événement Interreg CapComGR. Les partenaires du projet sont très contents de l'intérêt croissant apporté à la Robotix-Academy et voudraient saisir cette occasion pour remercier tous les participants et parrains pour cette excellente collaboration.

Inhaltsverzeichnis



2nd Robotix-Academy Summer School

Liège, September 03rd to 05th

Robotix-Academy Summer School

Die Robotix-Academy Summer School fand vom 03. bis 05. September 2018 an der Universität Lüttich (Belgien) statt und bot die Möglichkeit, Doktoranden und Masterstudenten der 5 Partneruniversitäten zusammenzubringen.
Die Universität Lüttich organisierte die zweite Summer School im Rahmen des Projekts Robotix-Academy. Anfang September trafen sich 16 Doktoranden und Masterstudenten der 5 Partneruniversitäten für drei Tage, um ihr technisches Wissen in verschiedenen Bereichen der Robotik, wie z.B. Human-Roboter-Kooperation oder Programmierung über ROS, auszutauschen.
Das Programm war sehr vielfältig und beinhaltete die Kalibrierung eines Lasermesssensors per Linie auf einem Roboterarm. Diese Veranstaltung bot auch eine Gelegenheit für die Teilnehmer, sich mit der Software „RobotStudio“ vertraut zu machen. Die Studenten konnten den Roboter „Yumi“ programmieren und modellieren; sie entdeckten so die Roboterprogrammierung von „ROS“ und „MoveIt“. Schließlich widmeten sie sich auch der computergestützten Bildverarbeitung für die Robotik und der Messung der menschlichen Bewegung.
Diese Veranstaltung bot auch Raum für Diskussionen und den Austausch und trug dazu bei, das Projektteam weiter zusammen zu schweißen.

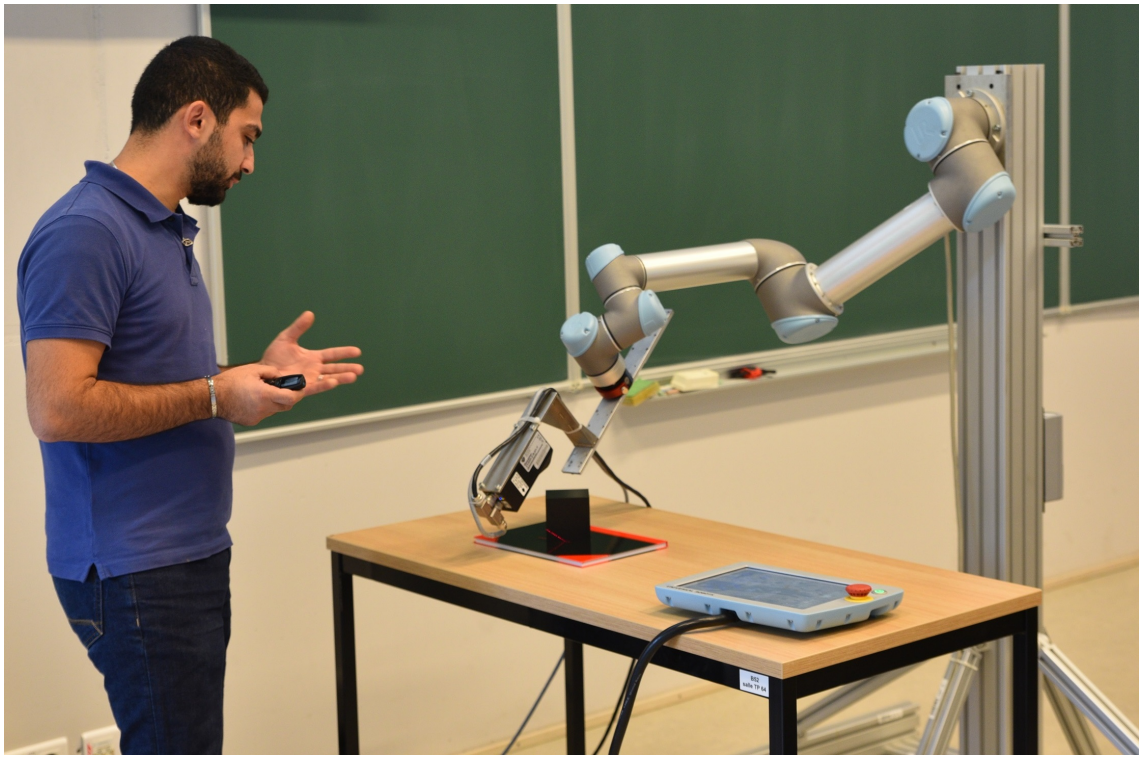
L'école d'été de la Robotix-Academy a eu lieu du 3 au 5 septembre à l'Université de Liège (Belgique) et a été l'occasion de réunir doctorants et étudiants de master des 5 universités partenaires.
L'université de Liège a organisé la deuxième école d'été du projet Robotix-Academy. Début septembre, 16 doctorants et étudiants de master des 5 universités partenaires se sont rassemblés durant 3 jours afin de partager leurs connaissances techniques dans divers domaines liés à la robotique, comme la collaboration humain-robot ou encore la programmation via ROS.
Le programme était très varié et comprenait notamment la calibration d'un capteur de mesure laser par ligne montée sur bras robotique. Ces journées ont également été l'occasion pour les participants de se familiariser avec le logiciel « RobotStudio ». Les étudiants ont pu programmer et modéliser le robot « Yumi » et ont découvert la programmation de robot par « ROS » et « MoveIt ». Enfin, ces journées étaient aussi dédiées à la vision par ordinateur pour la robotique et à la mesure du mouvement humain.
Cet événement était également un moment de discussion et d'échange qui a permis de souder un peu plus l'équipe du projet.

Interreg
EUROPEAN UNION
Grande Région | Großregion
Robotix-Academy
Fonds européen de développement régional | Europäischer Fonds für regionale Entwicklung

2nd Robotix-Academy Summer School at University of Liège, September 03rd to 05th 2018

	Monday	Tuesday	Wednesday	Thursday
8h	Welcome coffee & **Introduction**	**Robot programming using ROS and MoveIt.** 1/2 (Liège Université) *Robotic Lab*	**Human-Robot-Interaction** (Umwelt-campus Birkenfeld) *Robotic Lab*	To be determined
9h				
10h	**Position and orientation calibration of a 2D laser line sensor mounted on a robot flange** (ZeMA) *Robotic Lab*			
11h				
12h				
13h	Lunch	Lunch	Lunch	Lunch
14h	To be determined	**Robot programming using ROS and MoveIt.** 2/2 (Liège Université) *Robotic Lab*	Company visit (To be determined)	**Human motion measurement by Codamotion** (Liège Université) *Laboratoire d'analyse du mouvement humain*
15h				
16h				
17h				
18h				

ZeMA Zentrum für Mechatronik und Automatisierungstechnik · Université de Lorraine · Liège université · Pôle MecaTech · Hochschule Trier Umwelt-Campus Birkenfeld · Université du Luxembourg



Calibration of a 2D laser linesensor

Das ZeMA veranstaltete einen Workshop zum Thema „Kalibrierung eines 2D-Laserliniensensors“. Laserliniensensoren sind 2D-Sensoren, die nur Werte an zwei Koordinaten erfassen können. Während des Workshops wurde eine intuitive Kalibriermethode zur Kalibrierung eines 2D-Sensors mit einem Roboterarm vorgestellt.

Unter Berücksichtigung der vordefinierten Rahmenbedingungen wird aus dem Gleichungssystem, das an zwei Positionen mit dem Roboterarm in Verbindung steht, eine einzige Lösung berechnet. Um die Auswirkungen von Sensor- oder Umgebungsrauschen bei der Roboterapplikation zu minimieren, wird nach mehreren Messungen eine Lösung der kleinsten Fehlerquadrate bestimmt.

Kontakt:
ZeMA gGmbH
Ali Kanso
E-Mail: a.kanso@zema.de

Le centre ZeMA a organisé un atelier intitulé »Étalonnage d’un capteur laser linéaire 2D «. Les capteurs laser linéaire sont des capteurs 2D qui ne peuvent enregistrer des valeurs que sur deux coordonnées. Au cours de l’atelier, une méthode de calibration intuitive pour calibrer un capteur 2D par rapport à bras robotique a été présentée.

En tenant compte des conditions limites prédéfinies, une solution unique est calculée à partir du système d’équation liée à deux positions du bras robotique. Pour minimiser les effets du bruit du capteur ou du bruit ambiant dans l’environnement du robot, une solution des moindres carrés est déterminée après avoir effectué plusieurs mesures.

Contact:
ZeMA gGmbH
Ali Kanso
e-mail: a.kanso@zema.de

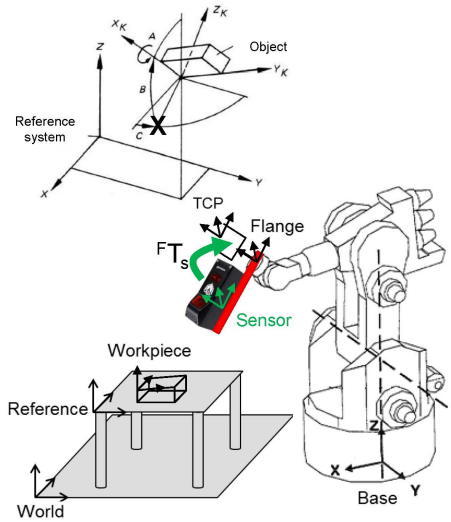
Project work – calibration of a 2D laser line sensor

Ali Kanso M.Sc.

ZeMA - Zentrum für Mechatronik und Automatisierungstechnik gemeinnützige GmbH

Liege, 03.09.2018 – 06.09.2018

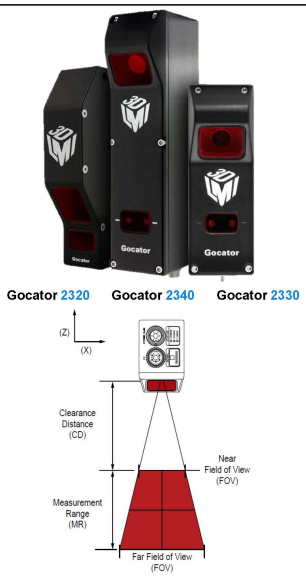
Introduction



Images: VDI 2681 - Blatt 1

- Coordinate systems for robot
  - World coordinate system
  - Reference coordinate system
  - Base coordinate system
  - Workpiece coordinate system
  - Flange coordinate system
  - End effector coordinate system (Tool Center Point - TCP)
- Transformation of the measured data
  - Laser line sensor provides 2D measured data with respect to the sensor coordinate system
  - The measured data must be transformed to the flange coordinate system
  - → expand the 2D measured data to 3D space
  - → The transformation matrix $^{F}T_{S}$ = **X** must be determined

Laser line sensor: LMI – Gocator 2300

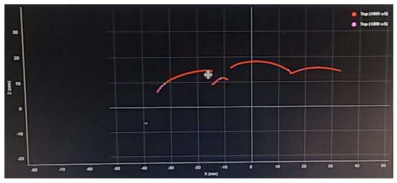


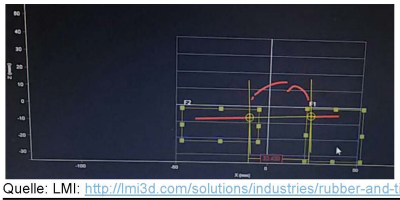
- Gocator 23XX, XX = 20, 30, 40, 50, 70 oder 80
  - Measuring range:
    - Xmin = 18 mm
    - Xmax = 1260 mm
    - Z = 25 mm … 800 mm
  - Accuracy: µm range
  - Price: ca. 8500 €
  - Function: Laser triangulation (Laser + CMOS Camera)
  - Provides raw data (1280 points/ profile)
- Provides raw data and teach-in algorithms are available

Quelle: LMI: http://lmi3d.com/solutions/industries/rubber-and-tire#rubber-tire

Laser line sensor: LMI – Gocator 2330

- The sensor provide 2D Data with respect to its internal coordinate system
- The coordinate system of the system must be calibrated with respected to flange coordinate system of the robot
- → the 2D measured data can be transformed to 3D data





Quelle: LMI: http://lmi3d.com/solutions/industries/rubber-and-tire#rubber-tire

Universal Robot – User-friendly HRC robot system for everybody

Universal Robot UR3 / UR5 / UR10

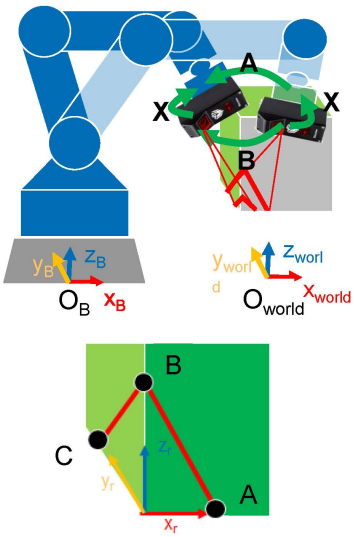
- 6 axis jointed arm robot
- Maximum range 500mm / 850mm / 1300mm
- Maximum load capacity 3 kg / 5 kg / 10 kg
- Own weight 11 kg / 18 kg / 29 kg
- Aluminum housing

Features:

- Very cheap and easy to handle
- User-friendly operation via Touch Pad
- Simple teaching of the positioning tasks by hand guiding
- Developed for small and medium-sized enterprises

Reference: Universal Robot Images: Universal Robot

Workshop application

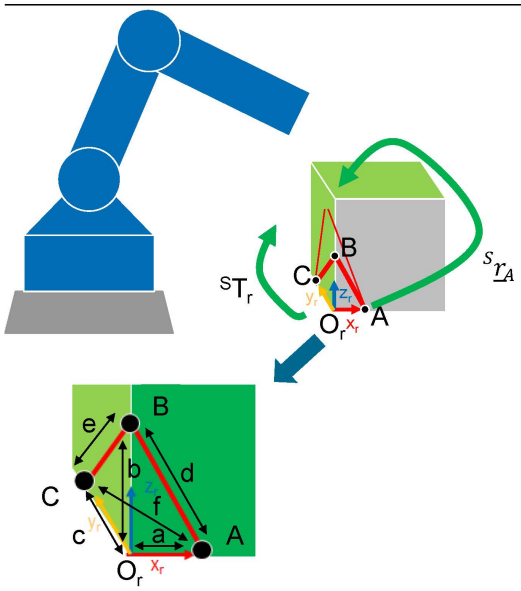


- The sensor must be mounted on the robot flange
- The corner of a precise fabricate cube is measured with the sensor
  - A coordinate system that is connected to the corner is measured with respected to internal coordinate system of the sensor
  - The cube represents a calibration target
  - The pose of the robot flange with respect to the base coordinate system is observed from the robot controller
  - → this event is repeated four times after moving the robot flange
- Let the calibration matrix of the sensor with respect to robot flange be **X**
- The transformation matrices of the robot flange and the sensor coordinate system with respect to the initial pose are **A** and **B** respectively
- → solve **AX=XB**

Workshop application

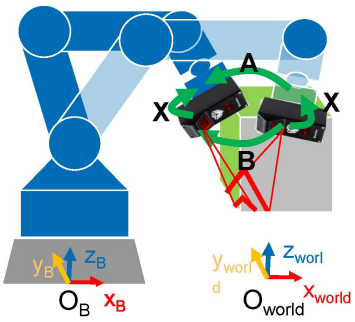
- *Edge detection*

Cube measurement



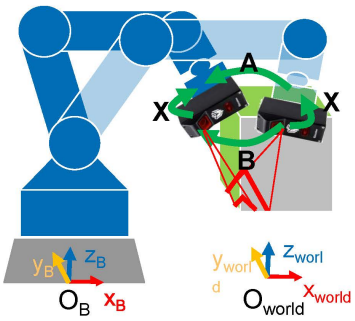
- A reference coordinate system is defined at a corner of a cube
- The corner is measured with the laser liner sensor
  - Three positions $^{s}\underline{r}_A$, $^{s}\underline{r}_B$ *and* $^{s}\underline{r}_C$ are measured with the sensor
  - A, B and C lies on X-, Y- und Z-axes of the reference coordinate system respectively
    - $^{r}\underline{r}_A = \begin{pmatrix} a \\ 0 \\ 0 \end{pmatrix}$; $^{r}\underline{r}_B = \begin{pmatrix} 0 \\ 0 \\ b \end{pmatrix}$; $^{r}\underline{r}_C = \begin{pmatrix} 0 \\ c \\ c \end{pmatrix}$;
- O_r is required to determine the transformation $^{s}T_r$

Solving the System AX=XB



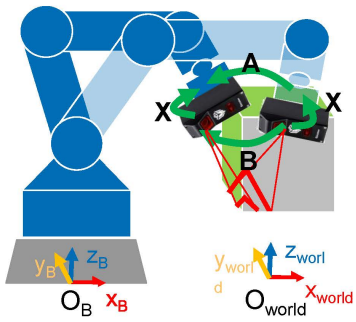
- **AX=XB**
- $\begin{pmatrix} D_A & \underline{r_A} \\ \underline{0}^T & 1 \end{pmatrix} \cdot \begin{pmatrix} D_X & \underline{r_X} \\ \underline{0}^T & 1 \end{pmatrix} = \begin{pmatrix} D_X & \underline{r_X} \\ \underline{0}^T & 1 \end{pmatrix} \cdot \begin{pmatrix} D_B & \underline{r_B} \\ \underline{0}^T & 1 \end{pmatrix}$
- Separation of the problem in position and orientation component
  - Orientation component
    - $D_A \cdot D_X = D_X \cdot D_B$
  - Position component
    - $D_A \cdot \underline{r_X} + \underline{r_A} = D_X \cdot \underline{r_B} + \underline{r_X}$
- Lie group theory transforms the orientation component into a linear system
  - Log(A)=$\frac{\theta}{2\sin(\theta)}(A - A^T)$
  - θ is the rotation matrix of A

Least square estimate – Orientation component



- mindestens drei Paare (A_i, B_i) sind notwendig, damit M regulär ist
  - → drei Bewegung ausgehend von der Erste Position
- Because of inaccuracies and noise:
  - $D_{Ai} \cdot D_X = D_X \cdot D_{Bi} <=> D_X \cdot \log(Ai) = \log(Bi)$
  - $sei\ \alpha_i = \log(Ai)\ und\ \beta_i = \log(Bi)$
  - $\rightarrow D_X \cdot \beta_i = \alpha_i$
- D_X is solvable by solving the equation :
  - $\min_{D_X} \sum_{i=1}^{n} \| D_X \cdot \beta_i - \alpha_i \|^2$
  - its solution in closed form can be effectively calculated as
    - $D_X = U \cdot V^{-1/2} \cdot U^{-1} \cdot M^T$
    - Wobei $M = \sum_{i=1}^{n} \beta_i \cdot \alpha_i{}^T$
    - $U \cdot V^{-1/2} \cdot U^{-1} = M^T \cdot M^{-1/2}$
    - $V = diag(\lambda_1, \lambda_2, \dots, \lambda_n)$
    - $\rightarrow V^{-1/2} = diag(1/\lambda_1, 1/\lambda_2, \dots, 1/n)$
    - $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigen values of the matrix $M^T \cdot M$
    - U is the matrix of the eigen values

Least square estimate – Position component



- $\underline{r_X}$ is solvable by solving the equation :
  - $\min_{\underline{r_X}} \sum_{i=1}^{n} \left\| (D_{A_i} - E)\underline{r_X} - D_X \cdot \underline{r_{B_i}} + \underline{r_{A_i}} \right\|^2$
  - Least squares method
    - → $\underline{r_X} = (C^T \cdot C)^{-1} \cdot C^T \cdot \underline{d}$
    - $C = \begin{bmatrix} E - D_{A_1} \\ E - D_{A_2} \\ \vdots \\ E - D_{A_n} \end{bmatrix}$
    - $\underline{d} = \begin{bmatrix} \underline{r_{A_1}} - D_X \cdot \underline{r_{B_1}} \\ \underline{r_{A_2}} - D_X \cdot \underline{r_{B_2}} \\ \vdots \\ \underline{r_{A_n}} - D_X \cdot \underline{r_{B_n}} \end{bmatrix}$

Robot programming using ROS & MoveIt

Die Universität Lüttich hat einen Workshop mit dem Titel „Programmierung von Robotern mit ROS & MoveIt“durchgeführt. ROS (Robotic Operating System) ist eine Computerumgebung zur Roboterprogrammierung und Sensorintegration. Diese Open-Source-Umgebung ist in Forschung und Industrie weit verbreitet und wird von Tag zu Tag umfangreicher. Der von der Universität Lüttich angebotene Workshop ist eine Einführung in zwei ROS-Tools: MoveIt und Gazebo. Während dieses Workshops werden die Teilnehmer zunächst angeleitet, ein Robotermodell (.urdf-Datei) in die ROS-Umgebung zu integrieren und mit dem Werkzeug Gazebo zu simulieren.
Der zweite Schritt besteht in der Programmierung des zuvor mit dem MoveIt Werkzeug simulierten Roboters. Schließlich ermöglicht der letzte Schritt die Integration einer Umgebung in die Robotersimulation.

Kontakt:
Universität Liège
Arthur Lismonde
E-Mail: alismond@ulg.ac.be

Robin Pellois
E-Mail: robin.pellois@ulg.ac.be

L'Université de Liège a proposé un atelier intitulé «Programmation de robots avec ROS & MoveIt». ROS (Robotic Operating System) est un environnement informatique pour la programmation de robot et l'intégration de capteur. Cet environnement open source est largement utilisé dans la recherche et dans l'industrie et s'enrichi de jour en jour. L'atelier proposé par l'Université de Liège est une introduction à deux outils de ROS : MoveIt et Gazebo. Lors de cet atelier, les participants sont d'abord amené à intégrer un model (fichier .urdf) de robot dans l'environnement de ROS et le simuler grâce à l'outil Gazebo. La seconde étape consiste en la programmation du robot simulé précédemment grâce à l'outil MoveIt. Enfin, la dernière étape permet d'intégrer un environnement dans la simulation du robot.

Contact:
Univérsité de Liège
Arthur Lismonde
e-mail:alismond@ulg.ac.be

Robin Pellois
e-mail:robin.pellois@ulg.ac.be

Interreg

EUROPEAN UNION

Grande Région | Großregion

Robotix-Academy

Fonds européen de développement régional | Europäischer Fonds für regionale Entwicklung

Robotix-Academy Summer-School 2018

September 3rd to 5th 2018

Robot programming using ROS & MoveIt

WorkShop presented by Mauricio Garcia, Arthur Lismonde and Robin Pellois

LIÈGE université

UNIVERSITÉ DE LORRAINE

UNIVERSITÉ DU LUXEMBOURG

PÔLE MECATECH

LE PÔLE DE COMPÉTITIVITÉ WALLON EN GÉNIE MÉCANIQUE

HOCHSCHULE TRIER Umwelt-Campus Birkenfeld

ZeMA

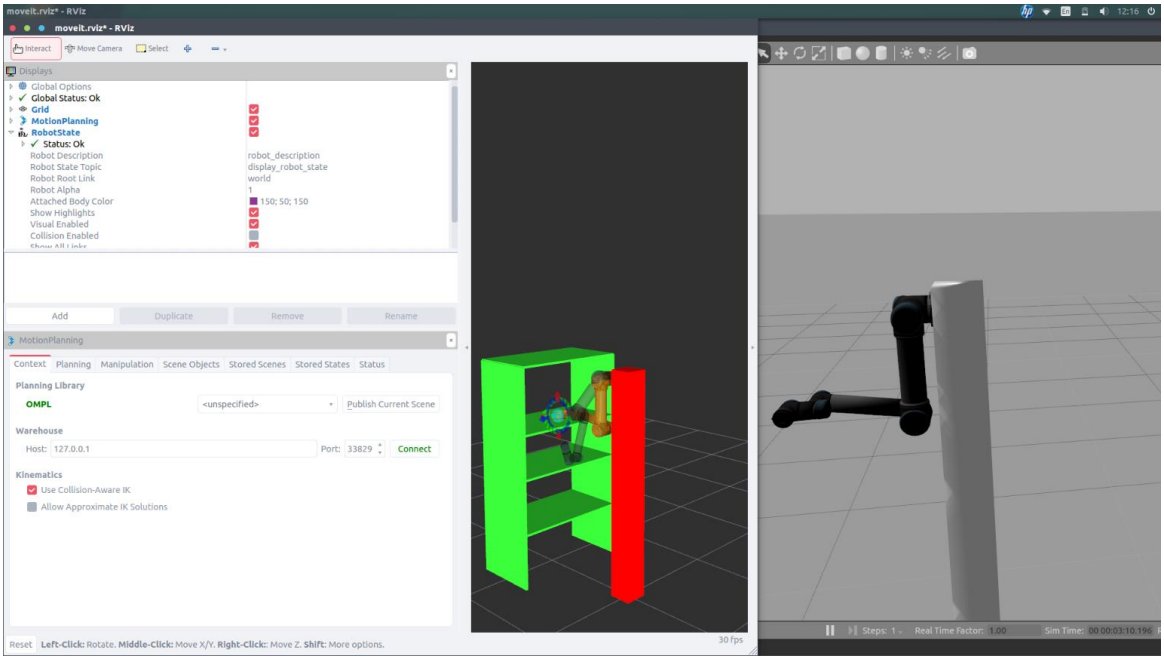
Table des matières

Introduction

Objectives of the complete tutorial

- Perform motion planning with ROS and MoveIt!
- Create a MoveIt configuration for a specific robot.
- Learn how to use .launch files.
- Cast a robot simulation in Gazebo (URDF- Xacro).

Illustration of the end of the workshop :



PART 1: GAZEBO

A - Creation of an virtual environnement based on URDF file

a - urdf file

The Unified Robot Description Format (URDF) is the standard ROS XML representation of a robot model (kinematics, dynamics, sensors) describing the robot characteristics in a ROS system.

FOR EXTRA INFO: http://wiki.ros.org/urdf

1) Create a workspace named **lab_workspace** in the home directory of the computer. To make ROS notice the existence of this package you have to modify a specific Ubuntu file. In a new terminal, in the home directory, write the following command:

gedit .bashrc

This will open a file. In that file, at the end of all the lines write the following, changing USER_XX with the name of your current Ubuntu user:

source ~/lab_workspace/devel/setup.bash

This is the standard procedure for making ROS to notice a user created package. It allows to avoid the requirement to source your workspace every single time you open a new terminal, which we will be doing several times in these tutorials.

2) Inside the **scr** folder of the **lab_workspace** folder create a **package** named **lab_ur5_pkg**

3) Inside the **lab_ur5_pkg** create a folder named **urdf**

4) Inside the folder named **urdf** create a file named **lab_ur5_joint_limited_robot.urdf** and write into the file the following:

4.1) The common headers: xml and robot name:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro"
       name="ur5" >

```

4.2) It is required to have a world link, which is the reference for the complete simulation. Also, create a “dummy” link that we will use in the future as follows:

```
<!-- original -->
<!-- links def-->
<link name="world" />

<link name="base_footprint_link" />

```

4.3) Create a pillar for attaching the robot, this will be used as the base for the robot. It is required to write the collision and the visual tags.

```
<link name="pillar">
    <inertial>
      <mass value="100.0"/>
      <inertia ixx="0.25" ixy="0.0" ixz="0.0" iyy="0.25" iyz="0.0" izz="0.2"/>
    </inertial>
    <visual>
      <geometry>
        <!-- TEST 2 -->
        <!-- <box size="0.5 0.5 0.05" /> -->
        <box size="0.2 0.2 2.0" />
      </geometry>
    </visual>
    <collision>
      <geometry>

        <box size="0.2 0.2 2.0" />
      </geometry>
    </collision>
</link>
```

4.4) Now we must connect the world with the pillar, and the pillar with the base of the robot. This is done by creating joints between the bodies. The location of the joints is measured relative to the geometric center of the bodies.

Add two joints to connect the pillar link to the world link. Also, define a joint for the “dummy” link.

```
  <!-- joints def-->
  <joint name="world_to_pillar_joint" type="fixed">
    <parent link="world" />
    <child link="pillar" />

    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
  </joint>

  <joint name="base_footprint_joint" type="fixed" >
   <parent link="world" />
   <child link="base_footprint_link" />
   <origin xyz="0.0 0.0 -1.0" rpy="0.0 0.0 0.0" />
  </joint>

</robot>
```

b - xacro file

Now, if the robot is a simple one, you could write all the links, joints, physical properties and visuals by hand. However, is a common practice for the robot manufacturers to provide their URDF robot description which can be launch in ROS.

1) Get the robot description of the UR5 as follows:

- copy the file : **/opt/ros/indigo/share/ur_description/urdf/ur5.urdf.xacro**
- paste it to **~/lab_workspace/src/lab_ur5_pkg/urdf** and renamed it as **UR51.urdf.xacro**

Xacro is a standard and very practical utility included with ROS. The package allows to use macros to construct XML files. With xacro, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions. This package is more useful when working with large XML documents, such as the .urdf robot descriptions required for ROS.

FOR EXTRA INFO: http://wiki.ros.org/xacro

So now, instead of writing or copying by hand all the robot description we will include it in the .urdf file with the use of xacro.

2) Change the name of your file from **lab_ur5_joint_limited_robot.urdf** to **lab_ur5_joint_limited_robot.urdf.xacro**

3) It is required to include the common plugins for the robot simulation in gazebo, which is achieved by:

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro"
       name="ur5" >

  <!-- common stuff -->
  <xacro:include filename="$(find ur_description)/urdf/common.gazebo.xacro" />

```

4) To include the actual robot in ROS, write the following lines right after the one you just added:

```xml
  <!-- ur5 -->
  <!-- This is a xacro macro containing the description file of the robot,
       this only includes the file, it does not load the file       -->
  <xacro:include filename="$(find lab_ur5_pkg)/urdf/UR51.urdf.xacro" />

  <!-- arm -->
  <!-- This is building the robot inside this URDF file-->
  <xacro:ur5_robot prefix="" joint_limited="true" />

```

(Note: We are using the "joint limited" version of the ur5 for the ROS simulation, for which the path planer requires smaller times to find trajectories).

5) Then, a joint is required to connect the robot with the pillar:

```xml
  <!-- joints def-->
  <joint name="world_to_pillar_joint" type="fixed">
    <parent link="world" />
    <child link="pillar" />
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
  </joint>

  <joint name="pillar_to_base_joint" type="fixed">
    <parent link="pillar" />
    <child link ="base_link" />
    <origin xyz="0.1 0.0 0.9" rpy="0.0 1.5708 0.0" />
  </joint>

  <joint name="base_footprint_joint" type="fixed" >
   <parent link="world" />
   <child link="base_footprint_link" />
   <origin xyz="0.0 0.0 -1.0" rpy="0.0 0.0 0.0" />
  </joint>

</robot>
```

B – Creation of the launch file

Now we want to launch the simulator and to load the robot on it, in order to visualize the setup designed in the previous steps. For this, we will use **roslaunch**, which is a default package of ROS. It allows to start a complete set of ROS nodes and load parameters in the ROS parameter server in a simple manner.

It includes options to automatically respawn processes that have already died. **roslaunch** takes one or more XML configuration files (with the **.launch** extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

FOR MORE INFO: http://wiki.ros.org/roslaunch

1) Make a new folder in the **lab_ur5_pkg** named **launch**. Inside that folder make a new empty file named **lab_robot.launch**, and write the following:

2) First, we write the default header and define some parameters that will be used

```
<?xml version="1.0"?>
<launch>

  <arg name="limited" default="true"/>
  <arg name="paused" default="true"/>
  <arg name="gui" default="true"/>

```

3) This is how an empty world simulation is launched, with gazebo showing the Graphic User Interphase and a paused simulation.

```
  <!-- startup simulated world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" default="worlds/empty.world"/>
    <arg name="paused" value="$(arg paused)"/>
    <arg name="gui" value="$(arg gui)"/>
  </include>

```

4) Now we have to send the .**urdf** file to the parameter server, in order to be able to refer the robot defined in the **.urdf.xacro** file.

```
  <!-- send robot urdf to param server -->
  <param unless="$(arg limited)"
         name="robot_description"
         command="$(find xacro)/xacro --inorder '$(find lab_ur5_pkg)/urdf/lab_ur5_robot.urdf.xacro'" />

  <param if="$(arg limited)"
         name="robot_description"
         command="$(find xacro)/xacro --inorder '$(find lab_ur5_pkg)/urdf/lab_ur5_joint_limited_robot.urdf.xacro'" />
```

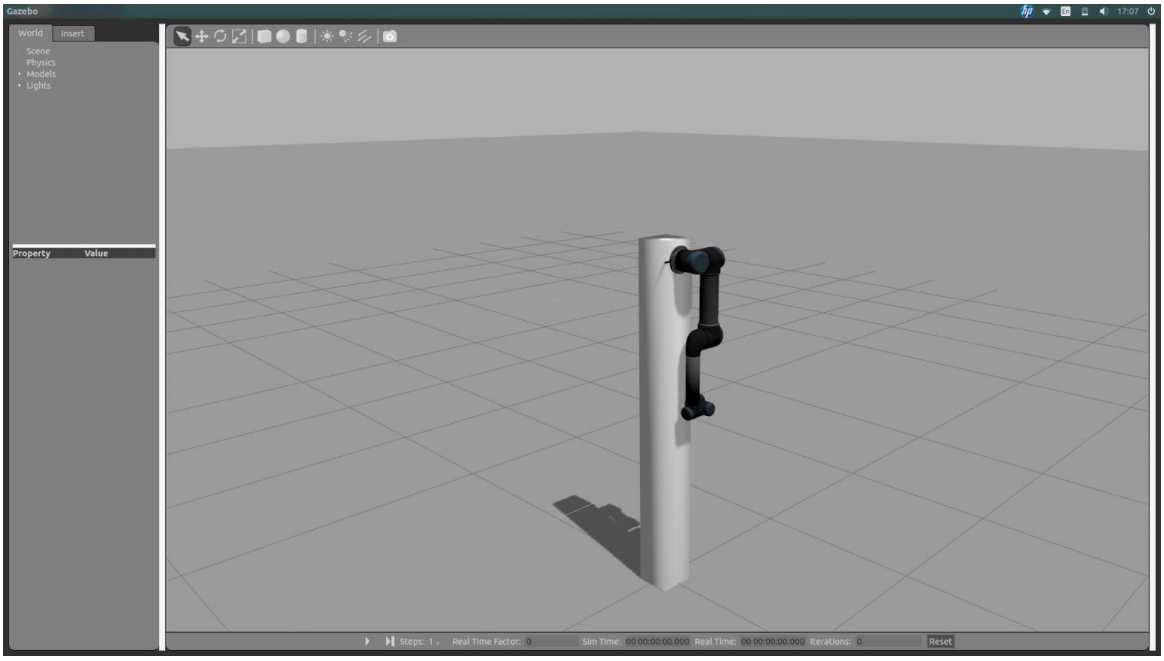
5) Last but not least, we have to cast the robot to the simulation, at the specified point, with a particular configuration for the joint wrist_1

```
<!-- push robot_description to factory and spawn robot in gazebo -->
<node name="spawn_gazebo_model"
      pkg="gazebo_ros"
      type="spawn_model"
      args="-urdf -param robot_description -model robot -x 1.0 -y 1.0 -z 1.0 -J wrist_1_joint 3.1415"
      respawn="false"
      output="screen" />

</launch>
```

7) Now you can see your work by launching the simulation in ROS by writing in a new terminal:

roslaunch lab_ur5_pkg lab_robot.launch

Now you know how to visualize customized robot description files in a Gazebo simulation.



PART 2: MOVEIT

MoveIt! is state of the art software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It provides an easy-to-use platform for developing advanced robotics applications, evaluating new robot designs and building integrated robotics products for industrial, commercial, R&D and other domains. Through

MoveIt! is the most widely used open-source software for manipulation and has been used on over 65 robots.

FOR MORE INFO: https://moveit.ros.org

Now we will use MoveIt! to generate, automatically, the required ROS configuration files by the UR5 to perform movement. The files will be able to command, whether a real or simulated robot, and the path planner that will generate the trajectories will be able to avoid obstacles in the workspace.

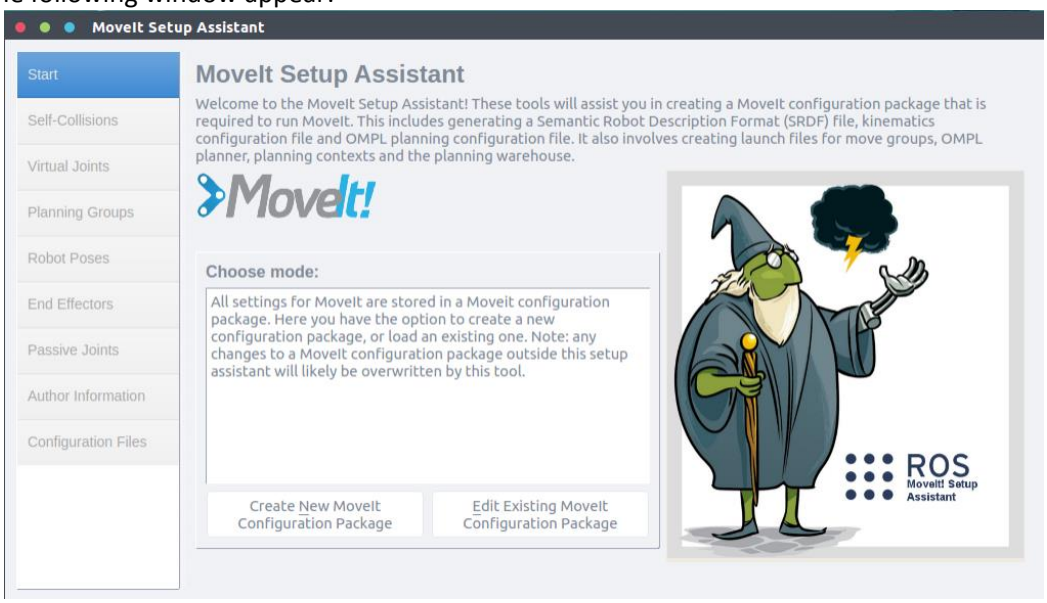
A – MoveIt Configuration

The MoveIt! Setup Assistant is a friendly graphical user interface for configuring any robot for use with MoveIt! We will use it to generate the configuration files for the UR5.

1) open it by typing in a the same terminal:

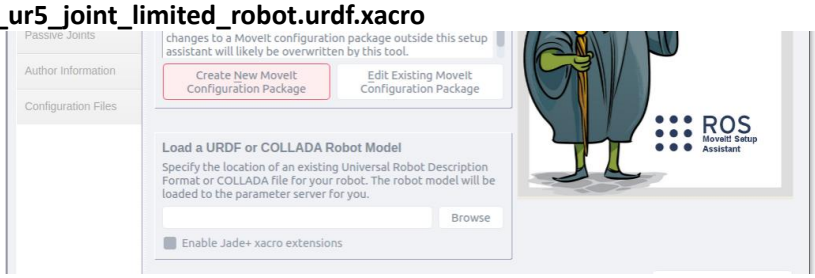
roslaunch moveit_setup_assistant setup_assistant.launch

The following window appear:



2) Click the button: **Create New MoiveIt Configuration Package**.

Then click the **browse** button and go to **~/lab_workspace/src/lab_ur5_pkg/urdf** and select the file **lab_ur5_joint_limited_robot.urdf.xacro**

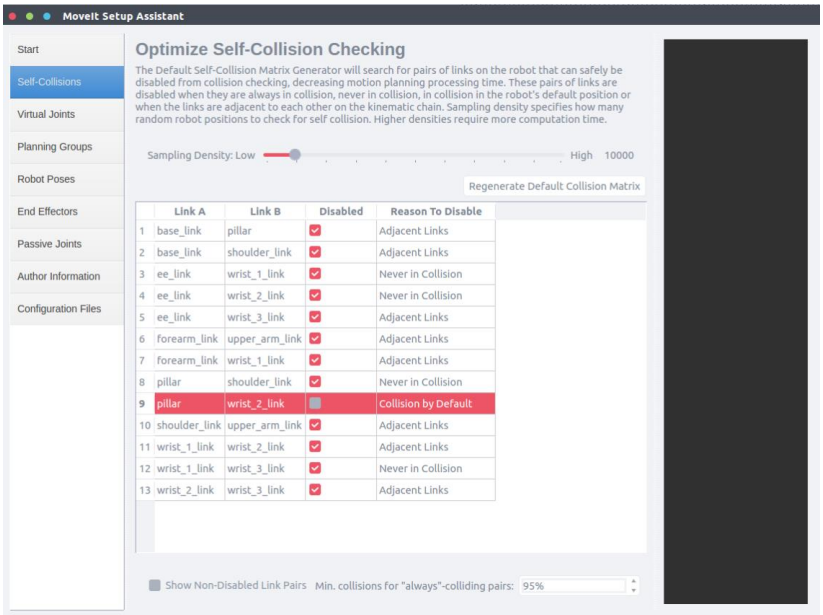


Now click the **Load Files** button.

3) Select the **Self-Collisions** tab in the left side of the window, and then click on the **Regenerate Default Collision Matrix** button.

This step detects the links that are one next to the other and deactivates the collision checking between them. The Default Self-Collision Matrix Generator searches for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time. These pairs of links are disabled when they are always in collision, never in collision, in collision in the robot's default position or when the links are adjacent to each other on the kinematic chain. Due to its initial position, the robot is inside the pillar. MoveIt detect it as "collision by default". This a problem that can be fixed easily as follow:

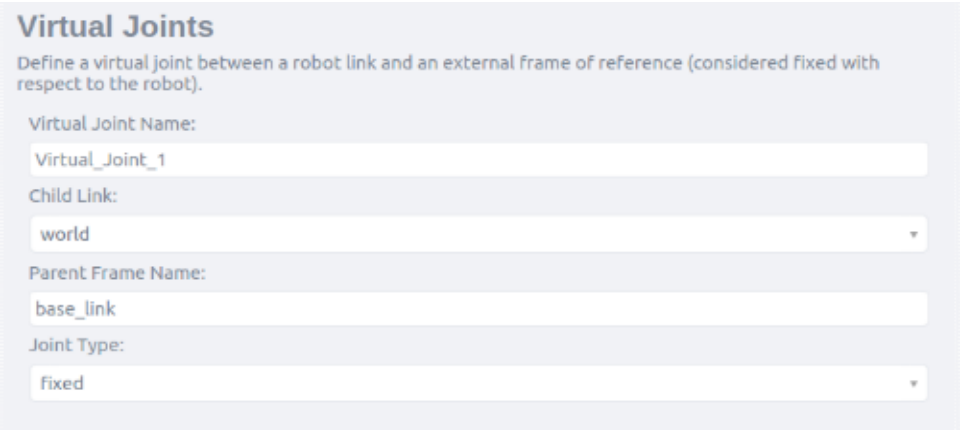
At this step, unclick row 9 tick, so your window looks exactly like this:



4) Go to the **Virtual Joints tab** and select **Add Virtual Joint**. In order to create a virtual

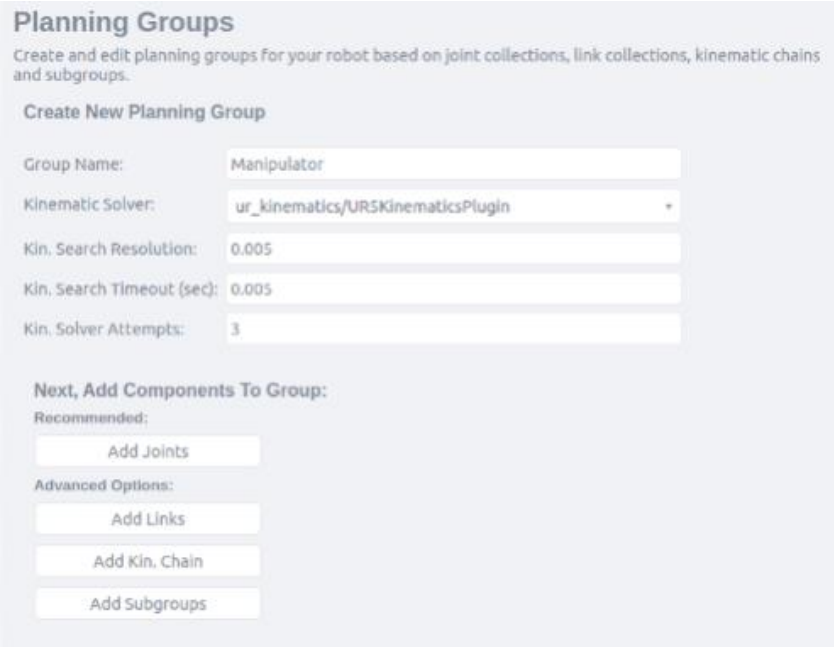
joint to connect the base of the robot with the world in the simulation, configure the following options:

Virtual Joint Name: **Virtual_Joint_1**, Child Link: **world**, Parent Frame Name: **base_link**, Joint Type: **fixed**



IMPORTANT: Make sure that the options are exactly as specified. Then click the Save button, now the robot should spawn, and you should see this window:

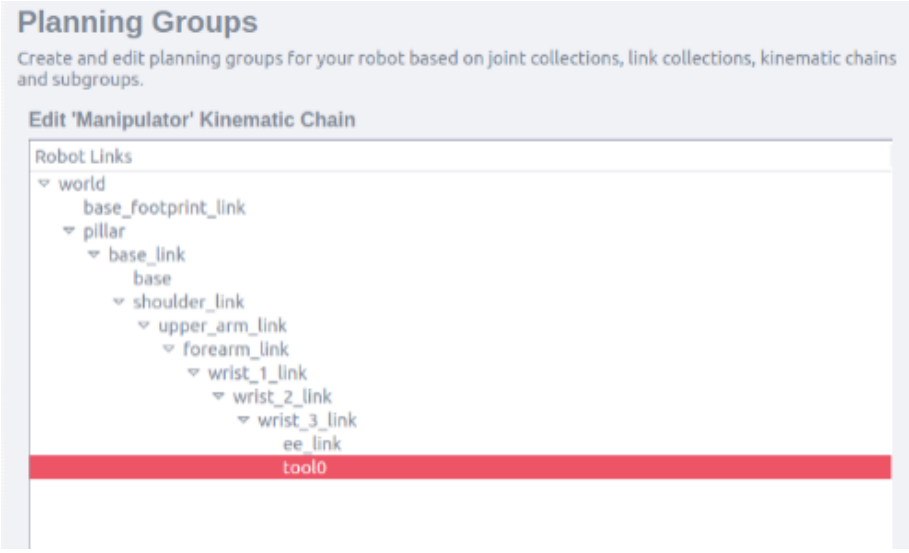
5) Now select the **Planning Groups tab**, click the **Add Group button** and write/select the following values. The Planning groups are used for semantically describing different parts of your robot, such as defining what an arm is, or an end effector. The set of links for doing the motion planning are being defined in this step.



6) In the current view click in the **Add Kin. Chain** button, select for the Base Link**:**

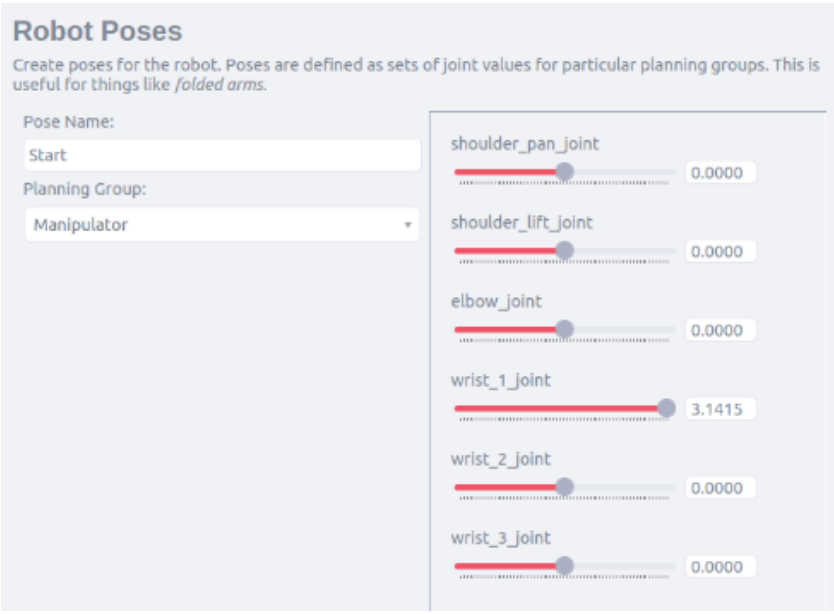
base_link and for the Tip Link**: tool0**

Now click save, your window should look **exactly** like this:



7) We can continue to the next tab, **Robot Poses**. The Setup Assistant allows you to add certain fixed poses into the configuration. This helps if, for example, you want to define a certain position of the robot as a Home/Start position.

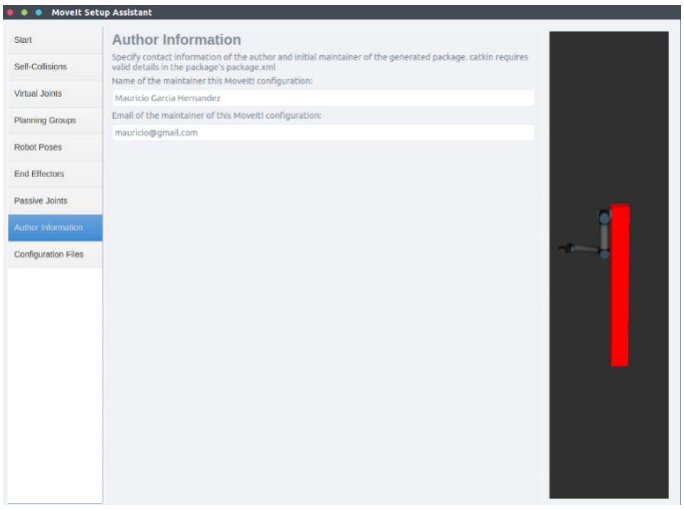
For this tutorial we need to define two poses. Click the **Add Pose** button, and use the sliders and text boxes to create a pose named **Start** with the following values, the joint values are expressed in radians:



Now create a new pose named **Middle**, with these values:

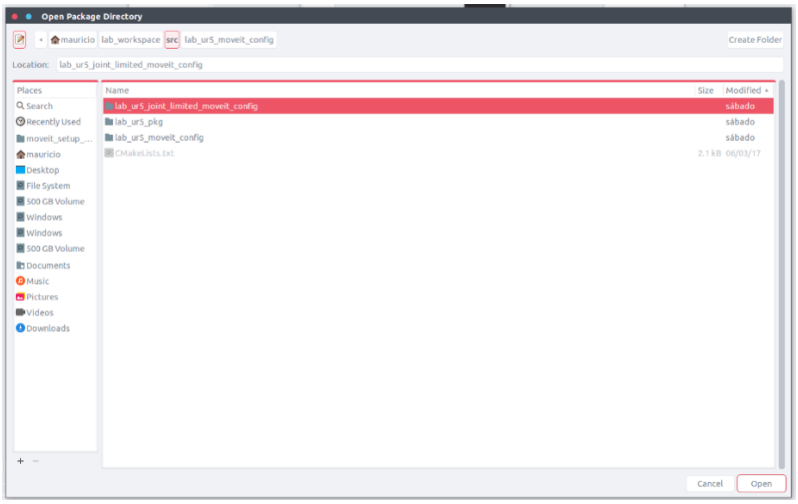
Later, these preconfigured poses will prove handy when we use the path planner in the simulation, as you will see.

8) Go to the **Author Information tab** in the left panel and fill in the information. You can fill it with fake information but it has as to be filled.

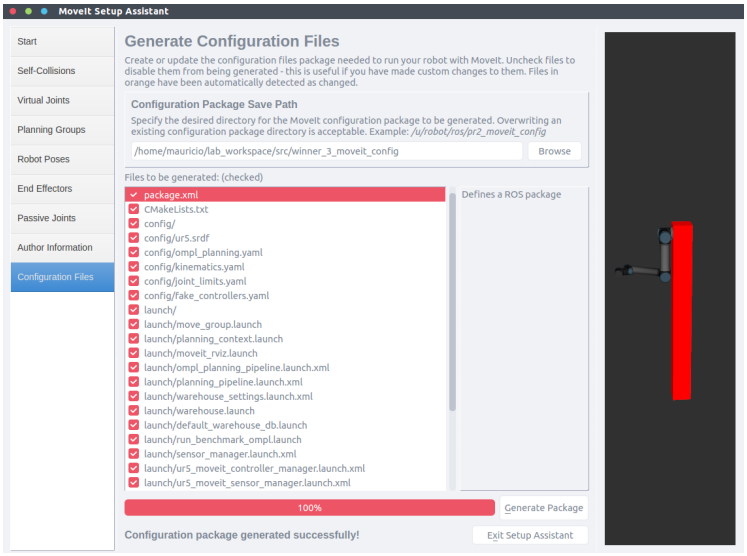


9) Now, without closing the **MoveIt** window, go to **~/lab_workspace/src/** and create a folder named **lab_ur5_joint_limited_moveit_config**

10) Go back to the MoveIt window and go to the **Configuration Files** tab in the left panel and click the **Browse** button to select the folder you just. Click on **open**.



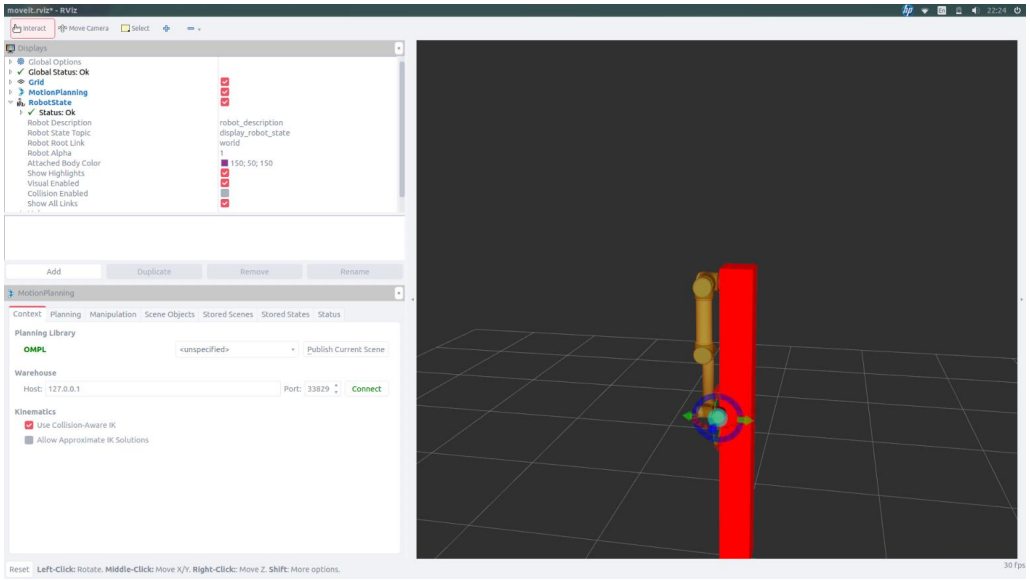
11) Now click the **Generate Package** button and **ok** in the pop-up window (only the gripper/tool should missed). When the 100% is reached the complete set of configuration files has been created.



12) Now you know how to create the configuration files required to move a robot using ROS using customized robot. If all went well you can launch a basic test demo of Moveit with your custom robot, write in a terminal:

roslaunch lab_ur5_joint_limited_moveit_config demo.launch

You should see this now in your computer:



13) rviz (ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. Using rviz, you can visualize the UR5 current configuration on a virtual model of the robot. You can also display live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more.

FOR MORE INFO: http://wiki.ros.org/rviz

However, you still have to connect your moveit configuration files with wether a real or simulated robot. Next, you will learn how to connect these files to do path planning and to use those trajectories for commanding your gazebo simulated robot.

B – Connection of the files with the real/simulated robot

The created package can provide the required ROS services and actions to generate and execute trajectories. This is achieved in the following way:

The objective of creating a MoveIt! configuration package is to control a real (or simulated robot). In the case of the simulated robot you have to include the controller of the robot in the simulation. In this way ROS, through the use of the created MoveIt! configuration package, can send information to move the simulated robot.

1) To include the controller in the gazebo simulated robot add the following lines at the end of your **lab_robot.launch** file (found in **~/lab_workspace/src/lab_ur5_pkg/urdf**) :

```
    <!-- INCLUDE the controller-->
    <include file="$(find ur_gazebo)/launch/controller_utils.launch"/>
    <!-- LOAD the controller-->
    <rosparam file="$(find ur_gazebo)/controller/arm_controller_ur5.yaml" command="load"/>
    <!-- LAUNCH the controller-->
    <node name="arm_controller_spawner"
          pkg="controller_manager"
          type="controller_manager"
          args="spawn arm_controller"
          respawn="false"
          output="screen"/>

</launch>
```

2) Now, to make the connection between your packages and the simulated robot you need to create some files. The first one specifies how the joints are going to be controlled. For this you need to know the name of the joint trajectory controller and the name space of the Action Server. Also, you need to know the names of the robot joints.

(Action server: In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request. This can currently be achieved via ROS services. In this case, the joint controller action server is in charge of receiving the movement messages from our MoveIt! configuration package and providing them to the simulated robot controller)

(Name space: A namespace is a ROS encapsulation technique, that allows to gather together a series of ROS items. In this way you could have, for example, two same UR5 robots in a simulation, sharing or not the information between them. In practice, you just write a prefix in most of your files, so instead of having something like: **/arm_controller/command** you could have, for the two robots, something similar to: **/robot_1/arm_controller/command and /robot_2/arm_controller/command**

However, a complete explanation about ROS namespaces is out of the scope of this tutorial. If you are interested, you can refer to: http://wiki.ros.org/Names)

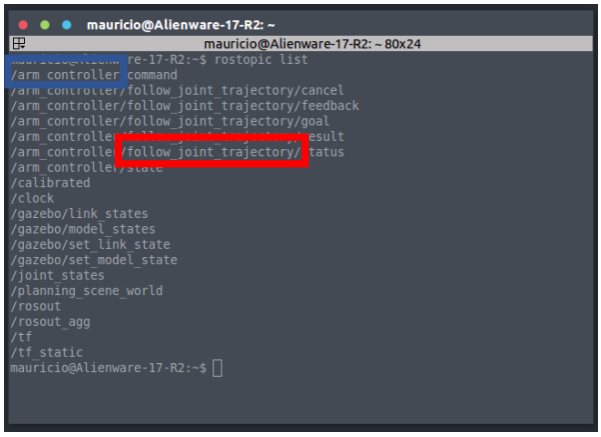
To get the name of the joint trajectory controller and the name space of the Action Server you have to launch in a terminal:

roslaunch lab_ur5_pkg lab_robot.launch

and click **play**, then in a new terminal check the topics currently active with the following command:

rostopic list

The names are found in the marked areas:



You recognize the name space of the **Action Server** from the cancel, feedback, goal, result and status end of lines.

To get the names of the joints you can review the **UR51.urdf.xacro** file.

2) Next, create a **controllers.yaml** file in the folder **lab_ur5_joint_limited_moveit_config/config**. Write in it the following code adapted to the names you found earlier:

```
controller_list:
  - name: arm_controller
    action_ns: "follow_joint_trajectory"
    type: FollowJointTrajectory
    joints: [shoulder_pan_joint, shoulder_lift_joint, elbow_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint]
```

3) Now, it is required to specify the names of the joints of the robot. You do this by creating the file **joint_names.yaml** in the same directory (**lab_ur5_joint_limited_moveit_config/config**) and writing the following:

```
controller_joint_names: [shoulder_pan_joint, shoulder_lift_joint, elbow_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint]
```

4) To load the controller (*specified in* **controllers.yaml**) in the ROS parameter server you have to use the **MoveItSimpleControllerManager** plugin (which connects MoveIt with the simulation). This allows to send the calculated plans from MoveIt to the simulated (or real) robot. The controller manager can be accessed by modifying the file named:

ur5_moveit_controller_manager.launch.xml
which can be found in the **launch** folder of the MoveIt configuration package **lab_ur5_joint_limited_moveit_config**. Add the following lines to this file:

```
<launch>
  <rosparam file="$(find lab_ur5_joint_limited_moveit_config)/config/controllers.yaml"/>
  <param name="use_controller_manager" value="false"/>
  <param name="trajectory_execution/execution_duration_monitoring" value="false"/>
  <param name="moveit_controller_manager" value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
</launch>
```

5) The last step is to create a launch file which will start the MoveIt environment. Inside the **launch** folder of the MoveIt configuration package:
~/lab_workspace/lab_ur5_joint_limited_moveit_config/launch
create a launch file named **demo_planning_execution.launch** and write in it the following code:

```
<launch>

  <rosparam command="load" file="$(find lab_ur5_joint_limited_moveit_config)/config/joint_names.yaml"/>

  <include file="$(find lab_ur5_joint_limited_moveit_config)/launch/planning_context.launch" >
    <arg name="load_robot_description" value="true" />
  </include>

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
    <param name="/use_gui" value="false"/>
    <rosparam param="/source_list">[/joint_states]</rosparam>
  </node>

  <include file="$(find lab_ur5_joint_limited_moveit_config)/launch/move_group.launch">
    <arg name="publish_monitored_planning_scene" value="true" />
  </include>

  <include file="$(find lab_ur5_joint_limited_moveit_config)/launch/moveit_rviz.launch">
    <arg name="config" value="true"/>
  </include>

</launch>
```

At this step we are launching the controller, and other launch files required by MoveIt, such as: the planning context, the move group and the MoveIt Rviz. This last **.launch** file were created by default when we created the MoveIt configuration package for our robot.

6) Often there is a common problem with MoveIt configuration packages, in which the path planner is not able to find collision free trajectories in the presence of obstacles. This is due to the density that the (selected) planner uses to confirm that the path is collision free.

So, in the **config** folder of the MoveIt! package we created (~/lab_workspace/lab_ur5_joint_limited_moveit_config/config) open the file **ompl_planning.yaml** and change the value (at the very end of the file) of the variable **longest_valid_segment_fraction** from **0.05** to **0.02**. Here you have an image of the file.

```
      - RRTConnectkConfigDefault
      - RRTstarkConfigDefault
      - TRRTkConfigDefault
      - PRMkConfigDefault
      - PRMstarkConfigDefault
    projection_evaluator: joints(shoulder_pan_joint,shoulder_lift_joint)
    longest_valid_segment_fraction: 0.02

src/lab_ur5_joint_limited_moveit_config/config/ompl_planning.yaml  0 0 0  1:1
```

(A brief explanation of this variable is the following: The **longest_valid_segment_fraction** defines the discretization of robot motions used for collision checking and greatly affects the performance and reliability of OMPL-based solutions. A motion in this context can be thought of as an edge between two nodes in a graph, where nodes are waypoints along a trajectory. The default motion collision checker in OMPL simply discretizes the edge into a number of sub-states to collision check. No continuous collision checking is currently available in OMPL/MoveIt! However, you can find the complete explanation of this variable in the MoveIt tutorial:http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/ompl_interface/ompl_interface_tutorial.html)

7) At this point you are ready to launch **rviz** with **MoveIt** in order to plan a trajectory and see how it is executed in the gazebo simulated robot.

First launch the Gazebo robot with:

roslaunch lab_ur5_pkg lab_robot.launch

Then launch the MoveIt package (in an other terminal) with the Rviz visualizer with:
roslaunch lab_ur5_robot_joint_limited_moveit_config demo_planning_execution.launch

All should be paused now, so you have to go to the Gazebo window and click play.

PART 3: Path Planning

It may be the case that with computer vision you detect an object you would like to pick, so you can define a pick point for the robot, or maybe with a laser sensor you detect an obstacle you would like to avoid. For such cases Rviz is not enough. Rviz is a great tool for visualizing the state of the robot. However, the standard practice when using ROS to command a robot is the use of code. In this way you can create a program that will move the robot in the working space to the points you desire automatically. You can do this using the Move Group Python Interface from MoveIt!

FOR MORE INFO:
http://docs.ros.org/indigo/api/moveit_tutorials/html/doc/pr2_tutorials/planning/scripts/doc/move_group_python_interface_tutorial.html

The interface provides functionality for most operations that a user may want to carry out, specifically setting joint or pose goals, creating motion plans, moving the robot, adding objects into the environment and attaching/detaching objects from the robot.

A - Obstacle avoidance

First, we will include a bookshelf in Rviz to try the path planning in a more realistic scenario. For the sake of simplicity, we will only add the bookshelf in Rviz (and not in Gazebo). However, in a real-world application the workspace is loaded in the Gazebo simulator by the creation of a custom .world file, such file includes all the URDF's of the robot environment (tables, walls, floor shapes, etc). Then, you would have to use a package such **gazebo2rviz** (http://wiki.ros.org/gazebo2rviz) to see your objects in **Rviz**. You could also add the bodies with a python node, or you could just code the **.scene** file in Rviz yourself.

1) In this case we will follow the third approach. In your **lab_ur5_pkg** create a folder called **scenes**. Inside the created folder make a new file named **bookshelve.scene**

2) The scene files stick to the following convention:

File	Explanation
box_scene * box1 1 box 0.1 0.2 0.3 1.0 1.0 1.0 0 0 0 1 0 1 0 1 .	#box_scene -> scene name #* box1 -> object name #1 -> number of shapes #box -> type of shape (box, cylinder, sphere) #0.1 0.2 0.3 -> xyz size of shape #1.0 1.0 1.0 -> position of shape #0 0 0 1 -> orientation of shape #0 1 0 1 -> RGBA color of shape #. -> necessary final dot

3) Now create **bookshelve.scene** file in the **scenes** folder with the following information.

```
bookshelves

* bookshelf_bottom
1
box
0.60 1.1 0.02
1.00 0 -0.5
0 0 0 1
0 0 0 0

* bookshelf_shelf1
1
box
0.60 1.1 0.02
1.00 0 0.0
0 0 0 1
0 0 0 0

* bookshelf_shelf2
1
box
0.60 1.1 0.02
1.00 0 0.4
0 0 0 1
0 0 0 0

* bookshelf_top
1
box
0.60 1.1 0.02
1.00 0 1.0
0 0 0 1
0 0 0 0

* bookshelf_left_wall
1
box
0.60 0.02 2.0
1.00 0.55 0.0
0 0 0 1
0 0 0 0

* bookshelf_right_wall
1
box
0.60 0.02 2.0
1.00 -0.55 0.0
0 0 0 1
0 0 0 0
.
```

4) Then, you have to add the bookshelve in Rviz, this is achieved by adding the following lines at the end of your **lab_robot.launch** file.

```
    <arg name="scene_file" default="$(find lab_ur5_pkg)/scenes/bookshelve.scene"/>
    <node name = "moveit_publish_scene_from_text"
          pkg= "moveit_ros_planning"
          type = "moveit_publish_scene_from_text"
          args= "$(arg scene_file)"/>
</launch>
```

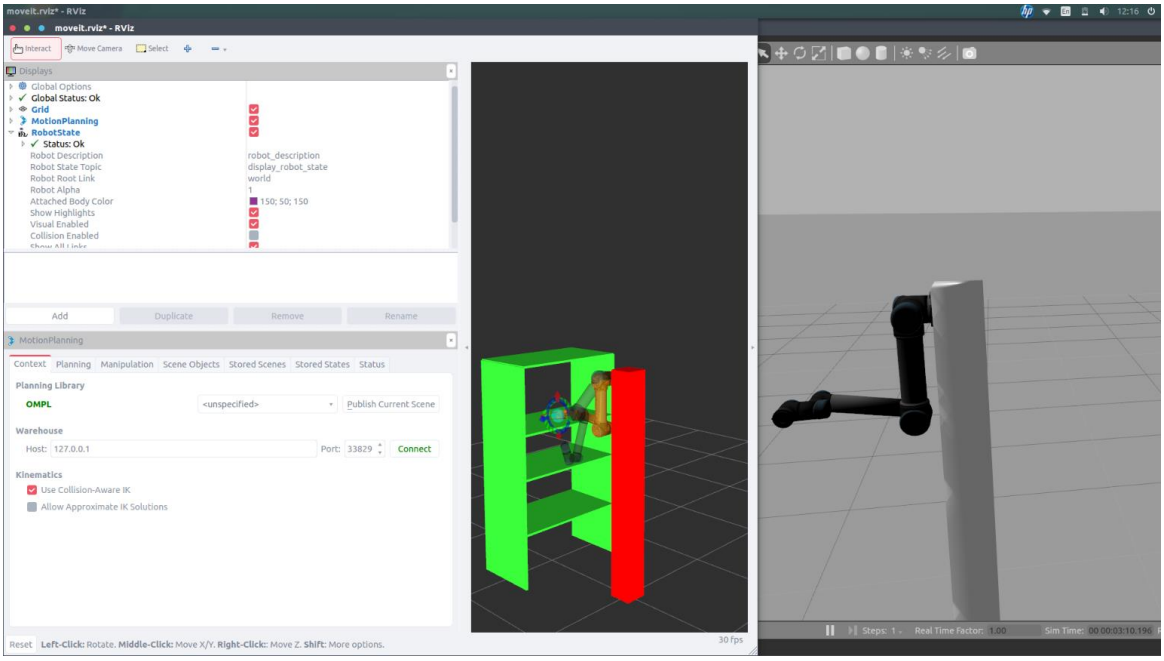
5) To see the effects of the last steps, Launch the Gazebo simulated robot with:

roslaunch lab_ur5_pkg lab_robot.launch

Then launch the MoveIt package with the Rviz visualizer with:

roslaunch lab_ur5_robot_joint_limited_moveit_config demo_planning_execution.launch

and click **play**. You should see something very similar to the following image.



B - Python code

Now we are ready to plan and execute a trajectory with python. First we will make a basic setup to use in the following steps.

1) Create a package named **lab_python_code_pkg** in the **lab_workspace/src** folder.

2) Inside the created folder (**lab_python_code_pkg/src**) create the following files:
move_to_point.py
joint_goal.py
give_me_data.py
give_me_the_pose.py

3) In a terminal go to the **lab_python_code_pkg/src** directory and allow for the files to be executable by typing in a terminal, once at a time:

chmod +x move_to_point.py
chmod +x joint_goal.py

chmod +x give_me_data.py
chmod +x give_me_the_pose.py

NB : Do not forget to build the **lab_workspace** after this (In a new terminal move to **lab_workspace** and then write **catkin_make**).

4) Now for the path planning and its execution you must add the following code in the file **move_to_point.py**

move_to_point.py — ~/lab_workspace — Atom

lab_robot.launch | move_to_point.py | joint_goal.py | give_me_data.py

```
1  #! /usr/bin/env python
2
3  import sys
4  import copy
5  import rospy
6  import moveit_commander
7  import moveit_msgs.msg
8  import geometry_msgs.msg
9
10 moveit_commander.roscpp_initialize(sys.argv)
11 rospy.init_node('move_group_python_interface_tutorial', anonymous=True)
12
13 robot = moveit_commander.RobotCommander()
14 scene = moveit_commander.PlanningSceneInterface()
15 group = moveit_commander.MoveGroupCommander("Manipulator")
16 display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path', moveit_msgs.msg.DisplayTrajectory)
17
18 group.set_planning_time(90.0)
19 group.set_planner_id("RRTConnectkConfigDefault")
20
21 pose_target = geometry_msgs.msg.Pose()
22
23 # Upper shelve pose
24 '''
25 pose_target.orientation.x = -0.49
26 pose_target.orientation.y = 0.51
27 pose_target.orientation.z = 0.51
28 pose_target.orientation.w = 0.49
29 pose_target.position.x = 1.0
30 pose_target.position.y = 0.2
31 pose_target.position.z = 0.6
32 group.set_pose_target(pose_target)
33 '''
34
35 # Near Start
36
37 pose_target.orientation.x = -0.49
38 pose_target.orientation.y = 0.51
39 pose_target.orientation.z = 0.51
40 pose_target.orientation.w = 0.49
41 pose_target.position.x = 0.67
42 pose_target.position.y = 0.19
43 pose_target.position.z = 0.52
44 group.set_pose_target(pose_target)
```

```
45
46
47 # Lower shelve pose
48 '''
49 pose_target.orientation.x = -0.49
50 pose_target.orientation.y = 0.51
51 pose_target.orientation.z = 0.51
52 pose_target.orientation.w = 0.49
53 pose_target.position.x = 0.72
54 pose_target.position.y = 0.19
55 pose_target.position.z = 0.26
56 group.set_pose_target(pose_target)
57 '''
58
59 plan1 = group.plan()
60
61 group.go(wait=True)
62
63 rospy.sleep(5)
64
65 moveit_commander.roscpp_shutdown()
```

The explanation of the code is the following:

Line 1-8: Import some common ROS modules, messages and the moveit_comander, that allows to communicate with MoveIt.
Line 10: Start the moveit_commander module.
Line 11: Starting the ROS node.
Line 13: Creation of a RobotCommander object to interface with the robot.
Line 14: Creation of a scene object to interface with the simulated world
Line 15: Creation of a group object to interface to the Manipulator group we created in the MoveIt configuration package, (remember?). By this we interact with the robot joints.
Line 16: Publisher that writes in the specified topic to see the planned motion in MoveIt
Line 18-19: Specification of the maximum time allowed for the path and selection of the path planner.
Line 21, 37-44: The goal is sent by the use of a pose object, which is the type of message that will be send as objective.

Line 59: We are telling the "Manipulator" group we created previously, to calculate the plan. If the plan is successfully computed, it will be displayed through MoveIt Rviz.
Line 61: We are telling the “Manipulator” group we created to execute the plan created.
Line 63: Pause the execution of the node for 5 seconds.
Line 65: Turr-off the moveit_comander

In order of appearance, the first position values are for placing the robot in the upper shelf, the second position is for placing the robot in the center out of the shelf, and the last one places the robot in a lower shelf.

5) Let us execute the created code. Launch the Gazebo simulated robot with:

roslaunch lab_ur5_pkg lab_robot.launch

Then launch the MoveIt package with the Rviz visualizer with:

roslaunch lab_ur5_robot_joint_limited_moveit_config demo_planning_execution.launch

and click play.

In a new terminal type: **rosrun lab_python_code_pkg move_to_point.py**

You should see the robot go to a position similar to the “Middle” one we created previously.

6) You can comment the last pose and chose (only) one of the others, the robot should move to the upper or lower shelf depending on your selection. Execute the code for going to the upper shelf. Later, comment that first pose section and uncomment the last pose, then execute the code again, now the robot should move to the lower shelf.

7) A simple way to define your custom poses would be to move the robot to wherever you want and check that pose in the terminal. Later, you could use that information to write those poses in your code. To do so, add the following code to the file **give_me_the_pose.py**

The code is much alike the one in the previous python file, the only difference in line 19-20 is that we print something in the terminal and then we print the pose of the end effector.

gime_me_the_pose.py — ~/lab_workspace — Atom

lab_robot.launch | move_to_point.py | joint_goal.py

```python
#! /usr/bin/env python

import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg

moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('info_node', anonymous=True)

robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
group = moveit_commander.MoveGroupCommander("Manipulator")
display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                               moveit_msgs.msg.DisplayTrajectory)

print "CURRENT POSE:"
print group.get_current_pose()

rospy.sleep(5)
```

8) If you have the Rviz and Gazebo windows launched, you can check the pose of the robot at any time by executing in a new terminal:

rosrun lab_python_code_pkg give_me_the_pose.py

9) You can also get some other relevant information by writing the following code in the file **give_me_data.py**

give_me_data.py — ~/lab_workspace — Atom

lab_robot.launch | move_to_point.py | joint_goal.py | give_me_data.py

```python
#! /usr/bin/env python

import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg

moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('info_node', anonymous=True)

robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
group = moveit_commander.MoveGroupCommander("Manipulator")
display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path', moveit_msgs.msg.DisplayTrajectory)

print "REFERENCE FRAME:"
print group.get_planning_frame()

print "CURRENT POSE:"
print group.get_current_pose()

print "CURRENT JOINT VALUES:"
print group.get_current_joint_values()

print "ROBOT STATE:"
print robot.get_current_state()

print "ROBOT GROUPS:"
print robot.get_group_names()


rospy.sleep(5)
```

10) You can launch this code as explained in point 5, if you have the Rviz and Gazebo windows launched, write in a new terminal:

rosrun lab_python_code_pkg give_me_data.py

11) Also, instead of doing path planning, if for example you get joint values from some external source you could just sent them to the robot with the following code. In the file **joint_goal.py** write:

joint_goal.py — ~/lab_workspace — Atom

lab_robot.launch | move_to_point.py | joint_goal.py

```python
#! /usr/bin/env python

import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg

moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('move_group_python_interface_tutorial', anonymous=True)

robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
group = moveit_commander.MoveGroupCommander("Manipulator")
display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                               moveit_msgs.msg.DisplayTrajectory)

group_variable_values = group.get_current_joint_values()

group_variable_values[0] = 0.8
group.set_joint_value_target(group_variable_values)

plan2 = group.plan()

group.go(wait=True)

rospy.sleep(5)

moveit_commander.roscpp_shutdown()
```

You should be familiar with this code by now, we get the current joint values and then we assign a specific value to joint 0. Later, we send it to the robot. Be aware that if the robot is obstructed when you sent this value the robot will not move. You can execute this code as explained in the previous steps.

Note: If you want to check the format of a ROS geometry message, write in a new terminal the following:

rosmsg show geometry_msgs/Pose

Congratulations, now you know how to simulate a robot, create a MoveIt configuration package, link the package with a simulated robot, create obstacles in Rviz and plan and execute trajectories.

Yumi: Modeling and programming

Im Rahmen der Robotix-Academy Summer School 2018 in Lüttich hat die Universität Lothringen einen Workshop für Studenten angeboten zum Thema „Yumi: Modellierung und Programmierung “. Ziel des Workshops war es, die Robotics Toolbox unter Matlab zu entdecken. Die Robotics Toolbox (RT) bietet viele nützliche und wichtige Funktionen in der Robotik (Kinematik, Dynamik, Bahnplanung etc.) und ermöglicht es, Versuchsergebnisse in der Handhabung mit realen Robotern zu simulieren und zu analysieren.

Diese Robotics Toolbox RT wurde anschließend für die geometrische und kinematische Modellierung und Bahnplanung des kollaborativen Roboters Yumi „irb 14.000“ von ABB verwendet.

Kontakt:
Universität Lothringen
Meryem Taghbalout
e-mail:meryem.taghbalout@univ-lorraine.fr

Dans le cadre de l'Ecole d'été Robotix Academy 2018 à Liège, l'Université de Lorraine a proposé aux étudiants un atelier sur »Yumi : modélisation et programmation«. Le but de l'atelier était de découvrir la Robotics Toolbox sous Matlab. La Robotics Toolbox (RT) fournit de nombreuses fonctions utiles et importantes en robotique (cinématique, dynamique, planification de trajectoire, etc.) et permet de simuler et d'analyser les résultats d'expérience avec de vrais robots.

Cette Robotics Toolbox RT a été utilisée par la suite pour la modélisation géométrique et cinématique et la génération de trajectoires du robot collaboratif Yumi »irb 14,000« d'ABB.

Contact:
Univérsité de Lorraine
Meryem Taghbalout
e-mail:meryem.taghbalout@univ-lorraine.fr

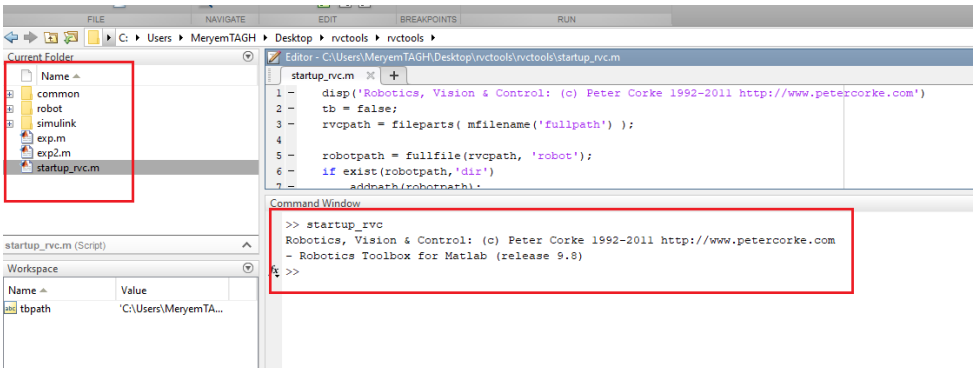
Tp: Yumi Modeling and Programming on Robotics Toolbox for Matlab

The goal of this workshop is to discover the Robotics Toolbox, created by Peter Corke (Queensland University of Technology, Australia), www.petercorke.com/Robotics_Toolbox.html , for the analysis of robots including the kinematic modeling of the Yumi irb 14000 robot from ABB.

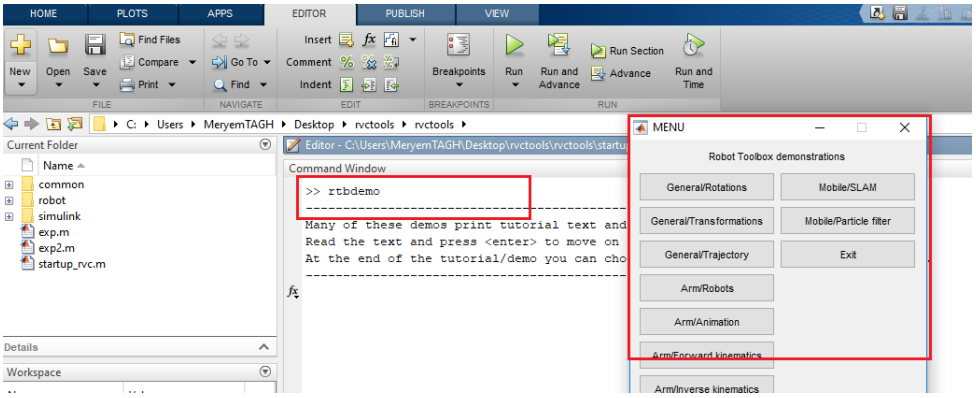
MATLAB is a powerful environment for linear algebra and graphical presentation, available on a very wide range of computer platforms. The basic functionality can be extended by application-specific toolboxes. The Robotics Toolbox provides many useful functions that are required in robotics (kinematics, dynamics, and trajectory generation…). The RT Toolbox is useful for simulating and analyzing the results of experiments with real robots, and can be a powerful tool for education.

Install Robotics Toolbox

- Unzip Folder rvctools that you will find in the following link: https://drive.google.com/open?id=1TDx1zTmGUPfeXlFpiV-4fLS3RdoGtezt
- Open Maltlab 2011 or newer versions
- Adjust your MATLAB PATH to include rvctools
- Execute the startup file vctools/startup_rvc.m



- Run the **rtbdemo** demo and discover the main features of the toolbox

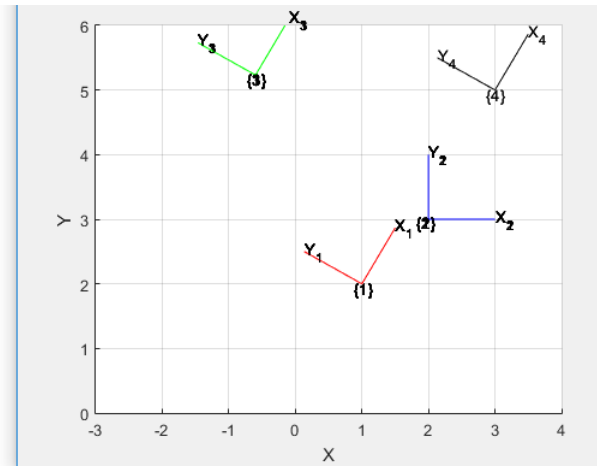


after the installation of the toolbox, we can start modeling.

Exercises:

Introduction: Representing positions in 2 Dim

```
T1=se2(1,2,60*pi/180);
T2=se2(2,3,0*pi/180);
T3=T1*T2;
T4=T2*T1;
figure(1)
trplot2(T1,'frame','1','color','r')
axis ([-3 4 0 6]);
hold on
trplot2(T2,'frame','2','color','b')
hold on
trplot2(T3,'frame','3','color','g')
hold on
trplot2(T4,'frame','4','color','k')
grid on
```



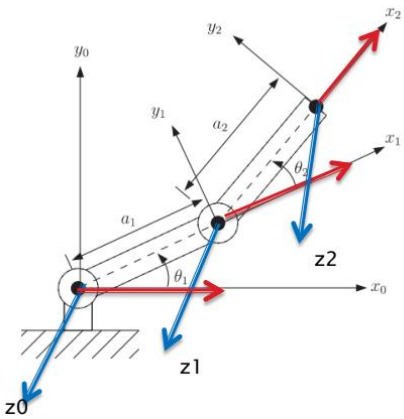
The first line creates a translation transformation [1, 2] and plane rotation of 60 °. After we multiplied the different matrices of transformation and we illustrate this in the figure beside with the function trplot2 which draws a 2D coordinate frame represented by the homogeneous transform T (3x3).

The arguments of the se2 object and other function can be found from >>help

```
Command Window
fx >> help se2
```

Exercise 1: Creating a new robot definition

In order to understand the use of RT basics, Let's take a simple example like the two-link planar manipulator which has the following Denavit-Hartenberg link parameters.



Link	α_i (rad)	d_i (mm)	a_i (mm)	θ_i (rad)
1	0	0	a_1	θ_1
2	0	0	a_2	θ_2

The Link toolbox class (with a capitalized L) is used to represent the segment of a simple open kinematic chain with the four parameters of Denavit-Hartenberg.

```
Command Window
>> help Link
 Link Robot manipulator Link class

  A Link object holds all information related to a robot link such as
  kinematics parameters, rigid-body inertial parameters, motor and
  transmission parameters.

  Methods::
   A             link transform matrix
   RP            joint type: 'R' or 'P'
   friction      friction force
   nofriction    Link object with friction parameters set to zero
   dyn           display link dynamic parameters
   islimit       test if joint exceeds soft limit
   isrevolute    test if joint is revolute
   isprismatic   test if joint is prismatic
   display       print the link parameters in human readable form
   char          convert to string

  Properties (read/write)::

   theta    kinematic: joint angle
   d        kinematic: link offset
   a        kinematic: link length
   alpha    kinematic: link twist
```

To define the robot via the Serial-link robot class with a1 = 0.6m and a2 = 0.8. You can follow the program below:

```
% DH-Paramters Using Peter Corke Robotics toolbox
a1 = 0.6;  a2 = 0.8;

% Create Link using this code
% L   = Link ( [ Th  d    a   alph])

  L(1)= Link ( [0   0    a1    0]);

  L(2)= Link ( [0   0    a2    0]);

  Rob = SerialLink (L);
  Rob.name = 'Robot 2ddl';
```

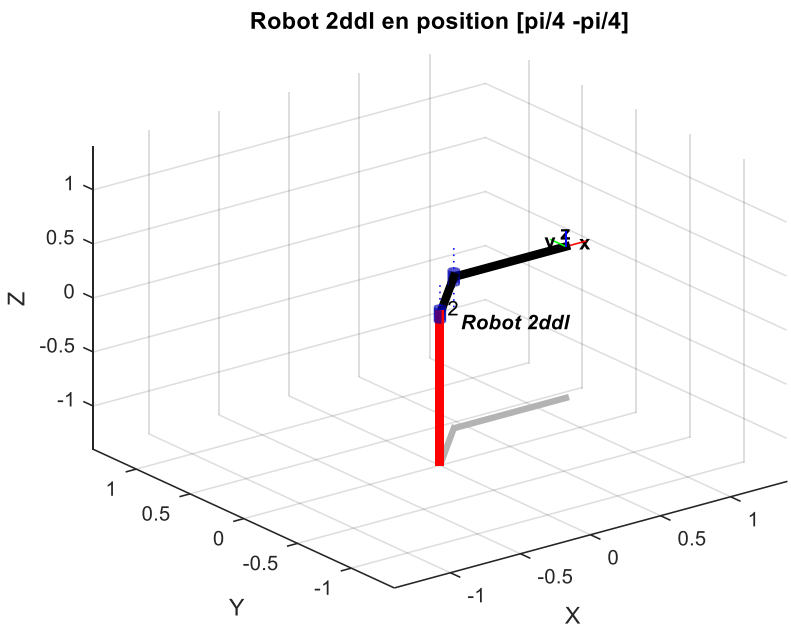
And to specify the different information that is returned, you just have to call the robot with its name.

```
>> Rob

Rob =

Robot 2ddl (2 axis, RR, stdDH)

+---+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |
+---+-----------+-----------+-----------+-----------+
|  1|         q1|          0|        0.6|          0|
|  2|         q2|          0|        0.8|          0|
+---+-----------+-----------+-----------+-----------+

grav =    0  base = 1  0  0  0   tool =  1  0  0  0
          0         0  1  0  0           0  1  0  0
       9.81         0  0  1  0           0  0  1  0
                    0  0  0  1           0  0  0  1
```

To calculate the forward kinematics of the robot we use the function "fkine" of the class Serial-link.

We take $[q_1, q_2]^T = [\pi / 4, -\pi / 4]^T$ and we visualize this robot position in 3D representation with the function plot in the same class Serial-link, see the Matlab program below.

```
>> q1 = pi/4 ; q2 = -pi/4 ;
 T= Rob.fkine([ q1 ,q2 ])
 figure(1)
 Rob.plot ([ q1 ,q2 ])
 title ('Robot 2ddl en position [0 0]')

T =

    1.0000    0.0000         0    1.2243
    0.0000    1.0000         0    0.4243
         0         0    1.0000         0
         0         0         0    1.0000
```



☺ The arguments to any function can be found from >> help.

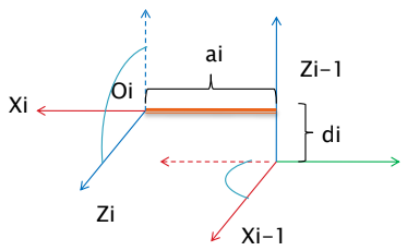
Exercise 2: Creating Yumi Model

Denavit Hartenberg parameters of a single arm of YUMI

The Manipulator serial-link robots comprising a set of bodies, called links, in a chain, connected by joints. Each joint has one degree of freedom, either translational or rotational. For a manipulator with n joints numbered from 1 to n, there are n 1 links, numbered from 0 to n. Link 0 is the base of the manipulator, generally fixed, and link n carries the end-effector. Joint i connects links i and i-1.

To facilitate describing the location of each link we affix a coordinate frame to it. Denavit and Hartenberg proposed a matrix method of systematically assigning coordinate systems to each link of an articulated chain. The link and joint parameters may be summarized as:

- a_i is distance from zi-1 to zi measured along xi
- α_i i is angle from zi-1 to zi measured about xi
- d_i is distance from xi-1 to xi measured along zi-1
- θ_i is angle from xi-1 to xi measured about zi-1

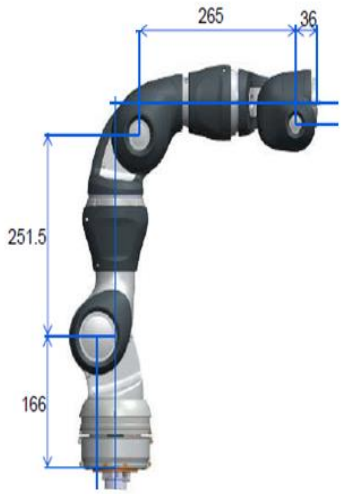


In this convention, each homogeneous transformation is represented as:

$$\left[\begin{array}{ccc|c} \cos\theta_n & -\sin\theta_n\cos\alpha_n & \sin\theta_n\sin\alpha_n & r_n\cos\theta_n \\ \sin\theta_n & \cos\theta_n\cos\alpha_n & -\cos\theta_n\sin\alpha_n & r_n\sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

The ABB YuMi is a two-arm collaborative robot that can be used for assembly operations. We consider the different parameters of Denavit-Hartenberg, defined in the table below to model a single arm of the robot.

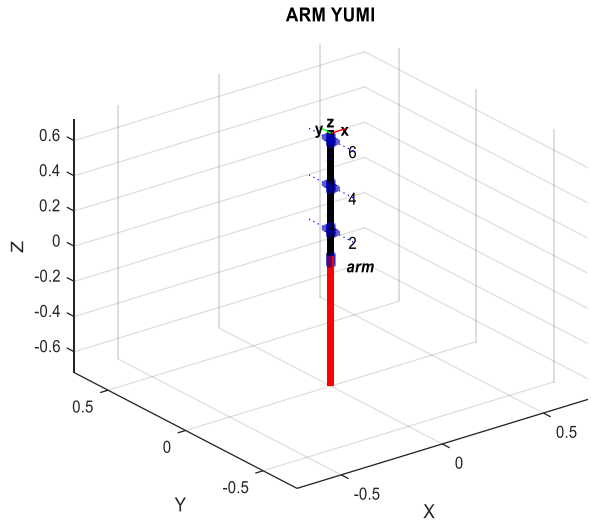
Link	α_i (rad)	d_i (mm)	a_i (mm)	θ_i (rad)
1	-π/2	0	166	q_1
2	π/2	0	0	q_2
3	-π/2	0	251.5	q_3
4	π/2	0	0	q_4
5	- π/2	0	265	q_5
6	π/2	0	0	q_6
7	0	0	36	q_7



Yumi's Single Arm Creation program is as follows:

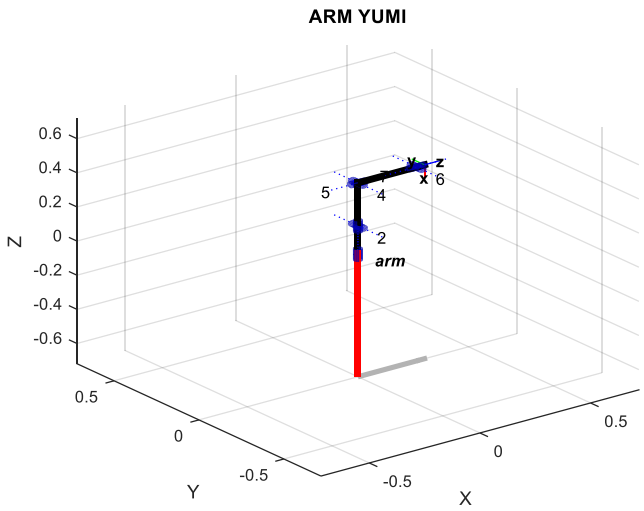
```
clear all
close all
clc
global  Alpha d r

% Desired position

q_l=[0 0 0 0*-pi/2 0 0 0 ];
q_r=[0 0 0 0*-pi/2 0 0 0 ];

%DH parameters

d1=0.166;d2=0;d3=0.2515;d4=0;d5=0.265;d6=0;d7=0.036;
r1=0;r2=0;r3=0;r4=0;r5=0;r6=0;r7=0;
Alpha1 =-pi/2;Alpha2 =pi/2; Alpha3 =-pi/2; Alpha4 =pi/2;
Alpha5 =-pi/2; Alpha6 =pi/2; Alpha7 =0;

Alpha=[Alpha1 Alpha2 Alpha3 Alpha4 Alpha5 Alpha6 Alpha7];
d=[d1;d2;d3;d4;d5;d6;d7];
r=[r1;r2;r3;r4;r5;r6;r7];
```

```
L(1)=Link ([0 d(1) r(1) Alpha(1) ]);
L(2)=Link ([0 d(2) r(2) Alpha(2) ]);
L(3)=Link ([0 d(3) r(3) Alpha(3) ]);
L(4)=Link ([0 d(4) r(4) Alpha(4) ]);
L(5)=Link ([0 d(5) r(5) Alpha(5) ]);
L(6)=Link ([0 d(6) r(6) Alpha(6) ]);
L(7)=Link ([0 d(7) r(7) Alpha(7) ]);


%robot
arm= SerialLink(L, 'name', 'arm');
arm.base=T0;
figure(3)
axis([-1 1 -1 1 -0.5 2]);
arm.plot(q_1);
title('ARM YUMI');
```



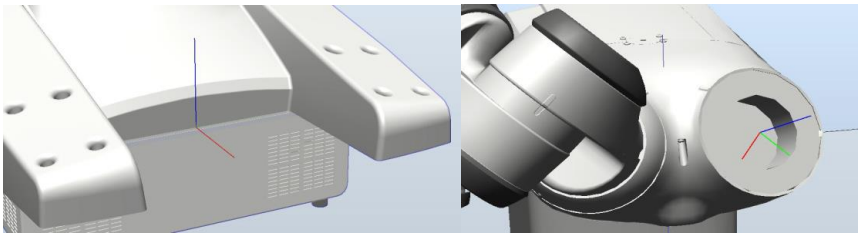
In position [0 0 0 0 0 0] the robot is not stretched we can add offsets on the robot with the option >> robot. offset
You can add these two lines in program to have the figure ..

```
offset=[0 0 0 pi/2 0 0 0];
arm.offset=offset;
```

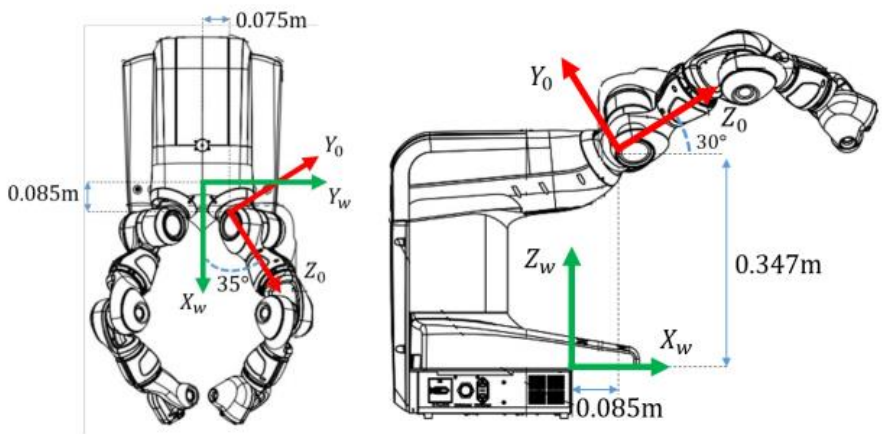


Definition of the base of the arm relative to the base of the robot:

The two arms of the robot are the same with a different basic transformation matrix



The x, y and z axes are represented respectively by the axes, red, green and blue base arm(left) base robot(right)



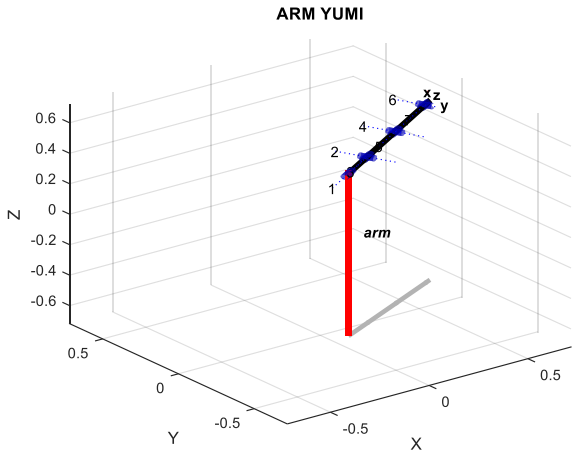
World frame (W) and base frame (0) for the left arm shown for the robot base. We have simplified the problem and eliminated the rotation about the X_W axis.

Based on the last figure, the basic transformation matrix that expresses the transformation matrix of the effector position in the robot's basic matrix is as following:

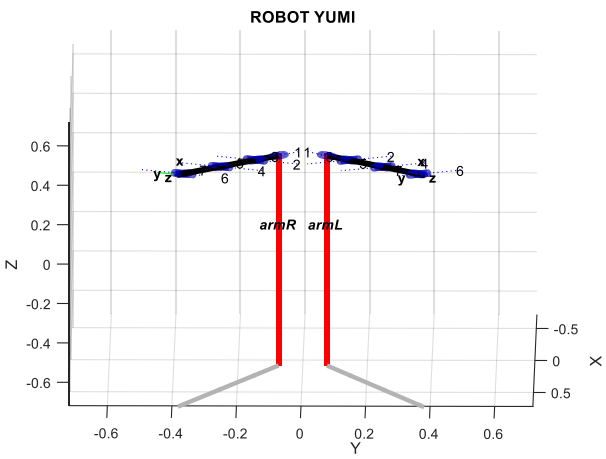
$$^{W}T_{Effector} = {}^{W}T_{0} \,.\, {}^{0}T_{Effector}$$

The following program allows us to add the basic information determined from last figure. We see that there is a rotation along the z axis of 35 ° and another of 30 ° along the y axis. For the datum point position, we have Pos = [0.085 0.075 0.347].

```
x0=0.085; y0=-0.075; z0=0.347;
T=rotz(35)*roty(30);
Pos=[x0; y0; z0];
T0=[T Pos;
    0 0 0 1];

%robot
arm= SerialLink(L, 'name', 'arm');
arm.base=T0;
figure(3)
axis([-1 1 -1 1 -0.5 2]);
arm.plot(q_1);
title('ARM YUMI');
```



The last program allows us to draw Yumi's two arms and to make simple movements of the arms.




```
x0=0.085; y0=0.075; z0=0.347;
T_l=rotz(35)*roty(30)
Pos_l=[x0;y0; z0]
T0_l=[T_l Pos_l;
    0 0 0 1]

T_r=rotz(-35)*roty(30)
Pos_r=[x0;-y0; z0]
T0_r=[T_r Pos_r;
    0 0 0 1]
```

```
for i=0:0.1:pi/2
q1=0;q2=0; q3=0;q4=0;q5=0;q6=0;q7=0;

armR.plot([q1+i q2 q3 q4 q5 q6 q7])
armL.plot([q1-i q2 q3 q4 q5 q6 q7])
end
```

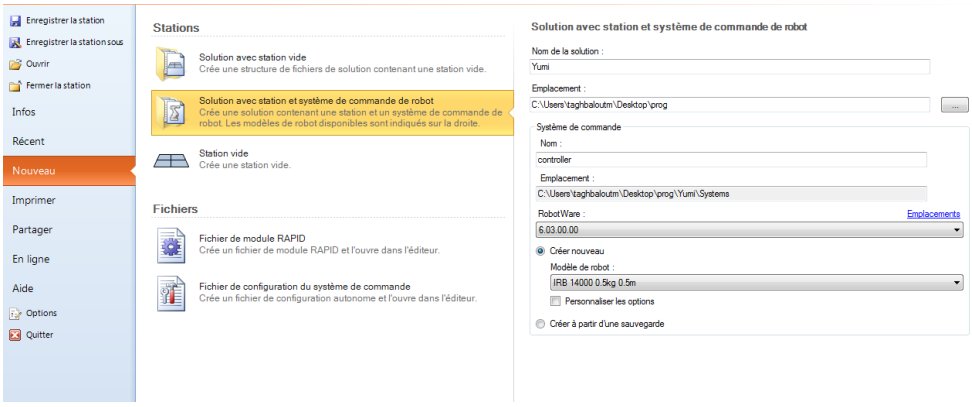
Tp : Yumi robot studio simple trajectory

Step1: Creat a new station

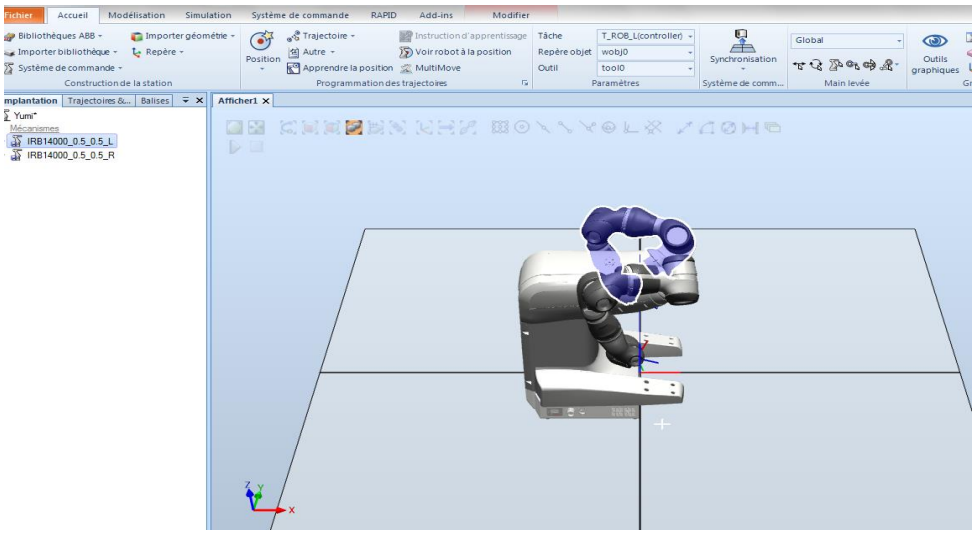
A new station can be created in three ways:

- Solution with Empty station
- Solution with station and robot controller
- Empty station

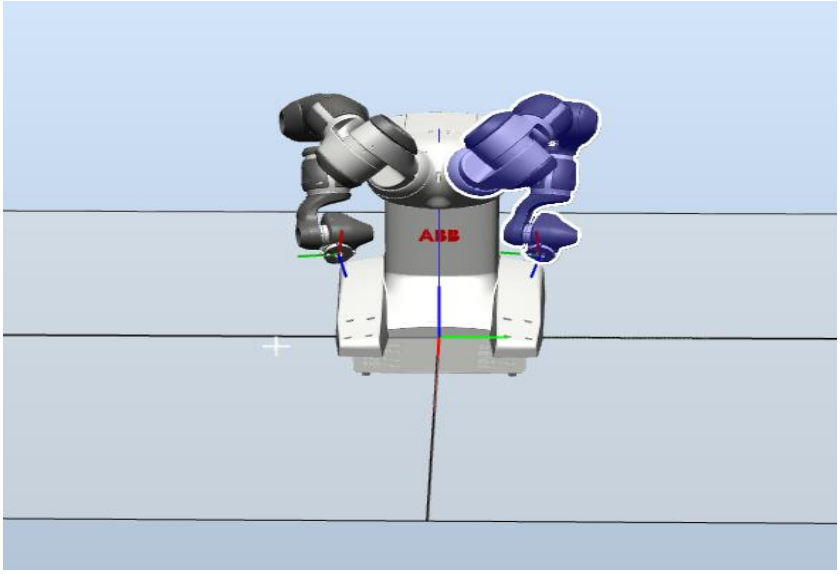
We will use the second alternative.



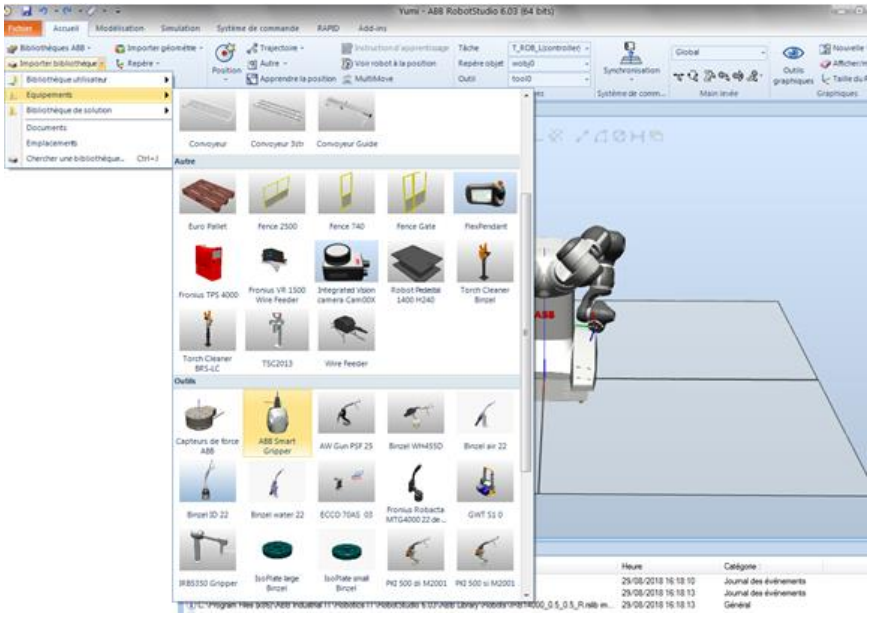
Step 2: Press create and wait until the robot appears

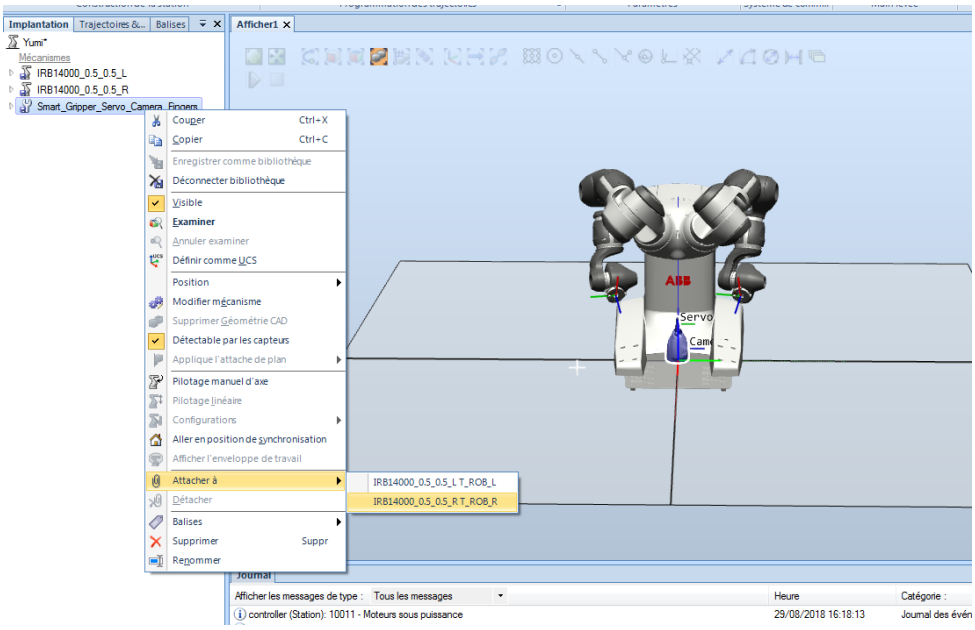


Step 3: Press CTRL +SHIFT+The left mouse button while dragging the mouse to rotate the station, press CTRL + The left mouse button to pan the station.

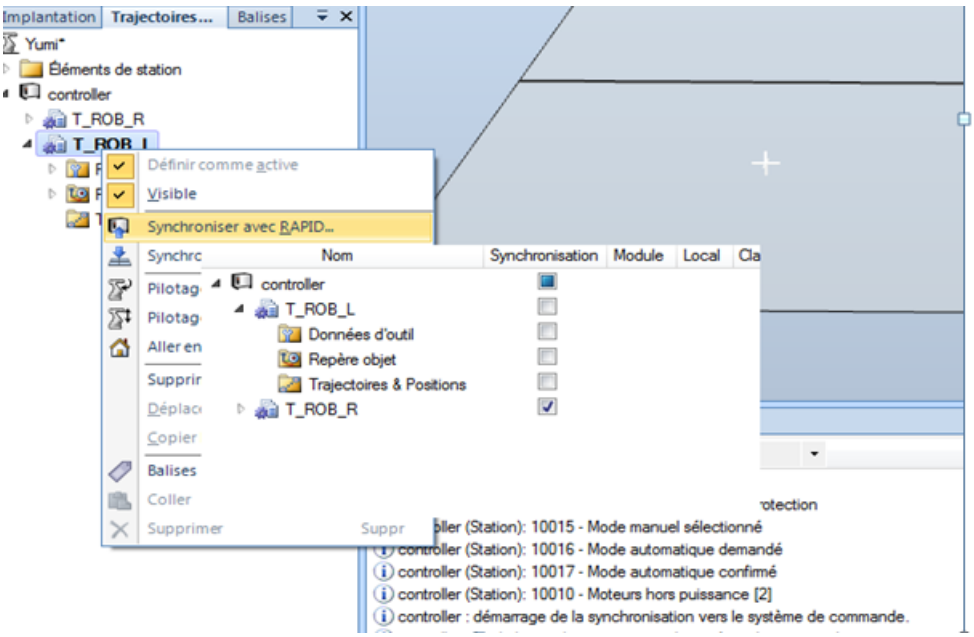


Step 4: Add the ABB smart gripper to the right arm of robot





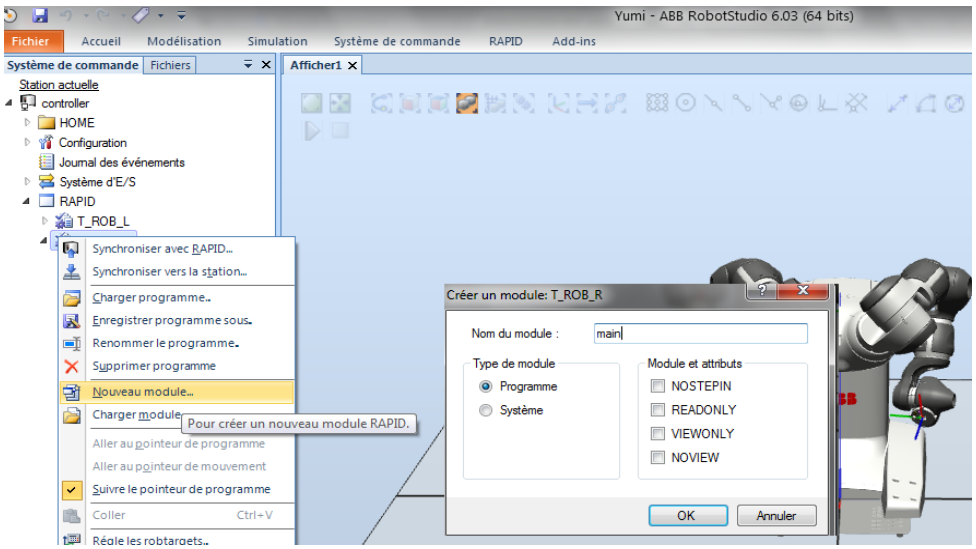
In the RAPID tab if you click the T_ROB_R you will find a created file called **Calib Data** where informations on the robot tool are presented (servo and camera).



Step 5: Repeat step 4 for the robot's left arm.

Step 6: Creating a RAPID program to make a simple trajectory to one of the arms.

There are a number of move instructions in RAPID. The most common are MoveL, MoveJ ,MoveAbsj and MoveC.



MoveJ is used to move the robot quickly from one point to another when that movement does not have to be in a straight line. Use MoveJ to move the robot to a point in the air close to where the robot will work.

MoveL instruction does not work if, for example, the robot base is between the current position and the programmed position, or if the tool reorientation is too large.

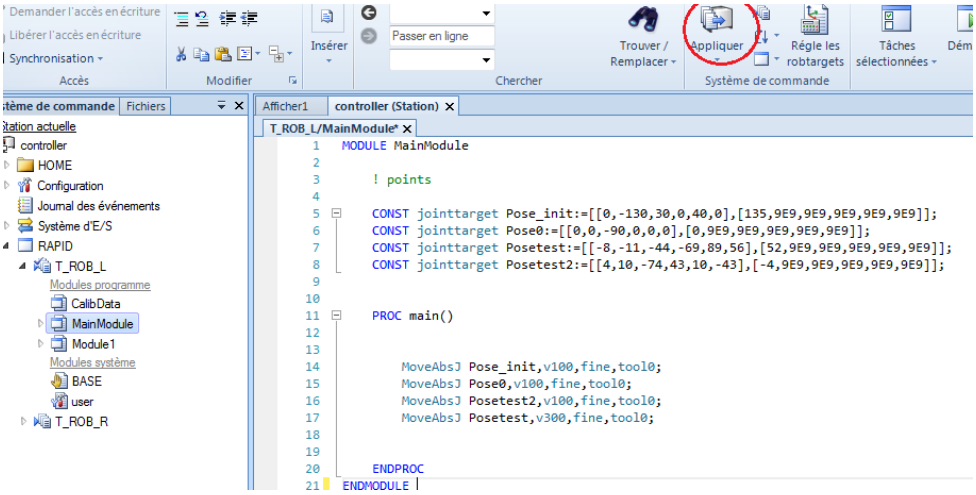
MoveC is used to move the robot circularly in an arc, MoveAbsJ is used to move the robot and external axes to an absolute position defined in axes positions.

- You can directly put the RAPID program as shown in the following step and after each modification it is necessary to press on "apply".

Example:

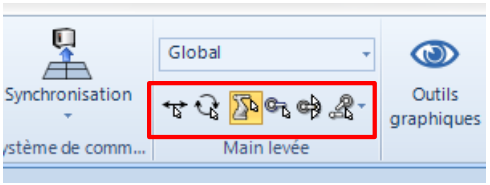
```
MODULE MainModule

  ! points

  CONST jointtarget Pose_init:=[[0,-130,30,0,40,0],[135,9E9,9E9,9E9,9E9,9E9]];
  CONST jointtarget Pose0:=[[0,0,-90,0,0,0],[0,9E9,9E9,9E9,9E9,9E9]];
  CONST jointtarget Posetest:=[[-8,-11,-44,-69,89,56],[52,9E9,9E9,9E9,9E9,9E9]];
```

```
CONST jointtarget Posetest2:=[[4,10,-74,43,10,-43],[-4,9E9,9E9,9E9,9E9,9E9]];

PROC main()

    MoveAbsJ Pose_init,v100,fine,tool0;
    MoveAbsJ Pose0,v100,fine,tool0;
    MoveAbsJ Posetest2,v100,fine,tool0;
    MoveAbsJ Posetest,v300,fine,tool0;

ENDPROC
ENDMODULE
```

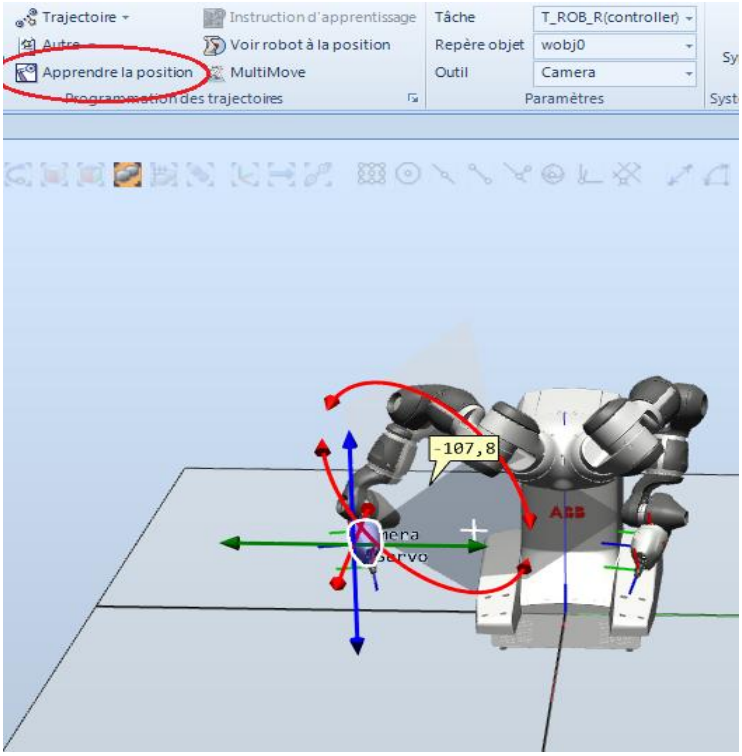


In the created module of the left arm, copy the example and start the simulation to see the robot movement.

- You can also create a RAPID program from the instructions of **Home**
  1. Move the robot arm with the following instructions

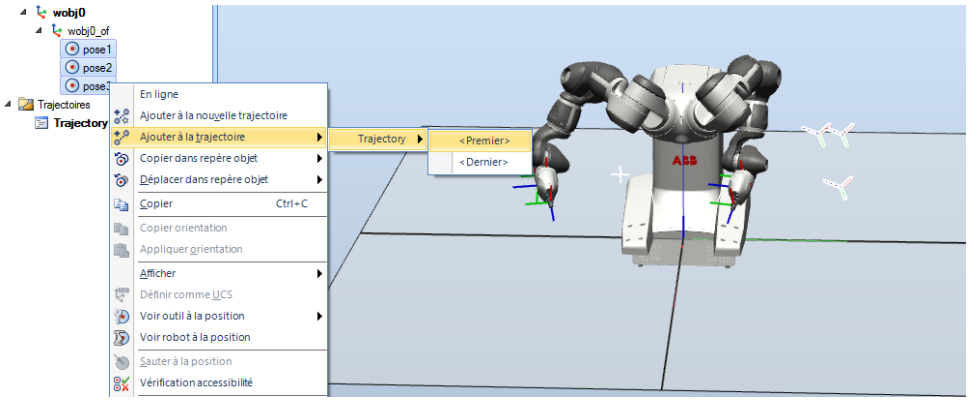


2. Click **Teach the instruction**

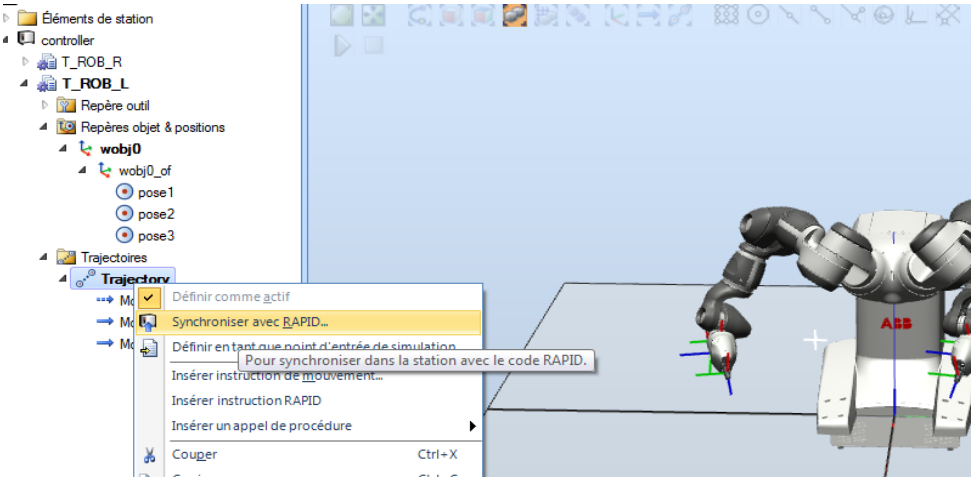


Now click the Target, select **Rename** and rename the position and do it again for different positions of your choices.

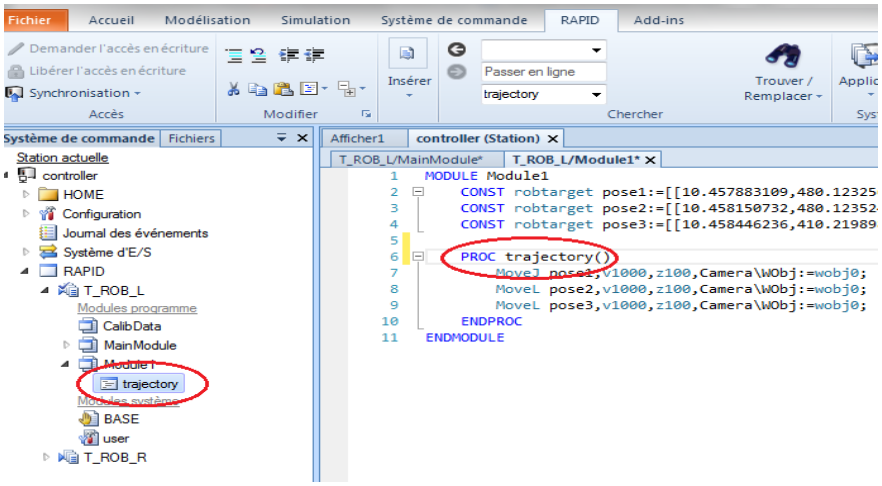
3. Adding the targets to a path
   - On the Home tab click Empty Path from drop-down menu, Path_10 is now created and displayed in the Paths&Target browser. You can rename it as you did for the target.
   - Select the positions you have created and add them to the trajectory as shown below.



- After creating the Path it only remains to synchronize it with RAPID program



- Now you find your program directly in **RAPID**



Computer vision for robotics

Der Umwelt-Campus Birkenfeld der Hochschule Trier bot in Lüttich einen Workshop an mit dem Titel „Computer Vision for Robotics“. Ziel dieses Workshops war es, an einem Beispiel die Einsatzmöglichkeiten von Bildverarbeitung in der Robotik kennen zu lernen. Hierzu wurde ein UR3 Roboter verwendet, welcher mit einem Kamerasystem und Servogreifer ausgestattet ist. Über eine in der Steuerung des UR3 Roboters integrierte Bildverarbeitungssoftware mussten die Teilnehmer eine Pick-and-Place Aufgabe realisieren. Des Weiteren wurde die Bildverarbeitungssoftware HALCON vorgestellt. In der dazugehörigen Übung lernten die Teilnehmer verschiedene Funktionalitäten der Software kennen.

Kontakt:
Umwelt-Campus Birkenfeld
Sebastian Groß
E-Mail: s.gross@umwelt-campus.de

Thomas Bartscherer
E-Mail: t.bartscherer@umwelt-campus.de

Le Campus environnemental de Birkenfeld a proposé à Liège un atelier intitulé «Vision par ordinateur pour la robotique». Le but de cet atelier était de découvrir les applications possibles du traitement d'images en robotique à l'aide d'exemples. D'une part, un robot Universal Robots UR3 a été utilisé, équipé d'un système de caméra et d'une pince servo. Les participants devaient réaliser une tâche de prise de vue à l'aide d'un logiciel de traitement d'images intégré à la commande du robot UR3. D'autre part, le logiciel de traitement d'images HALCON a été présenté. Dans cet exercice, les participants ont appris à connaître les différentes fonctionnalités du logiciel.

Contact:
Umwelt-Campus Birkenfeld
Sebastian Groß
e-mail: s.gross@umwelt-campus.de

Thomas Bartscherer
e-mail: t.bartscherer@umwelt-campus.de

Umwelt-Campus Birkenfeld | HOCHSCHULE TRIER

Robotix-Academy Summer School:

Computer Vision for Robotics

Interreg
EUROPEAN UNION
Grande Région | Großregion
Robotix-Academy
Fonds européen de développement régional | Europäischer Fonds für regionale Entwicklung

Directory

1 Getting Started

To install Halcon, just double click the Halcon Installer and follow the installation instructions. If you are asked for a license file, just point to the file that is given to you with the installer.
The getting Started guide is a shrunk version of the original guide by MVTec and the covered examples are delivered with the software library. If you are interested in the field, please feel encouraged to experiment with the software and especially the examples a bit after the summer school.
The license given to you is valid during September 2018.

In the following, it is assumed that HALCON has already been installed. Under Windows, HDevelop is usually started from the "Apps" screen or from the "Start" menu:

Start → Programs → MVTec HALCON → HDevelop

You can also start HDevelop from the Windows Command Prompt or from the `Start → Run…` menu, making it easy to pass optional command line switches:

```
hdevelop
```

Start Dialog

Figure 1.1: Start dialog.

The start dialog provides quick access to HDevelop programs (1), introductory material (2), and documentation (3). For the first introduction to HDevelop, try the links listed under "Getting Started". To close the start dialog without any further action, press Esc. If you have accidentally closed the start dialog, it can be re-opened from HDevelop's `Help` menu.

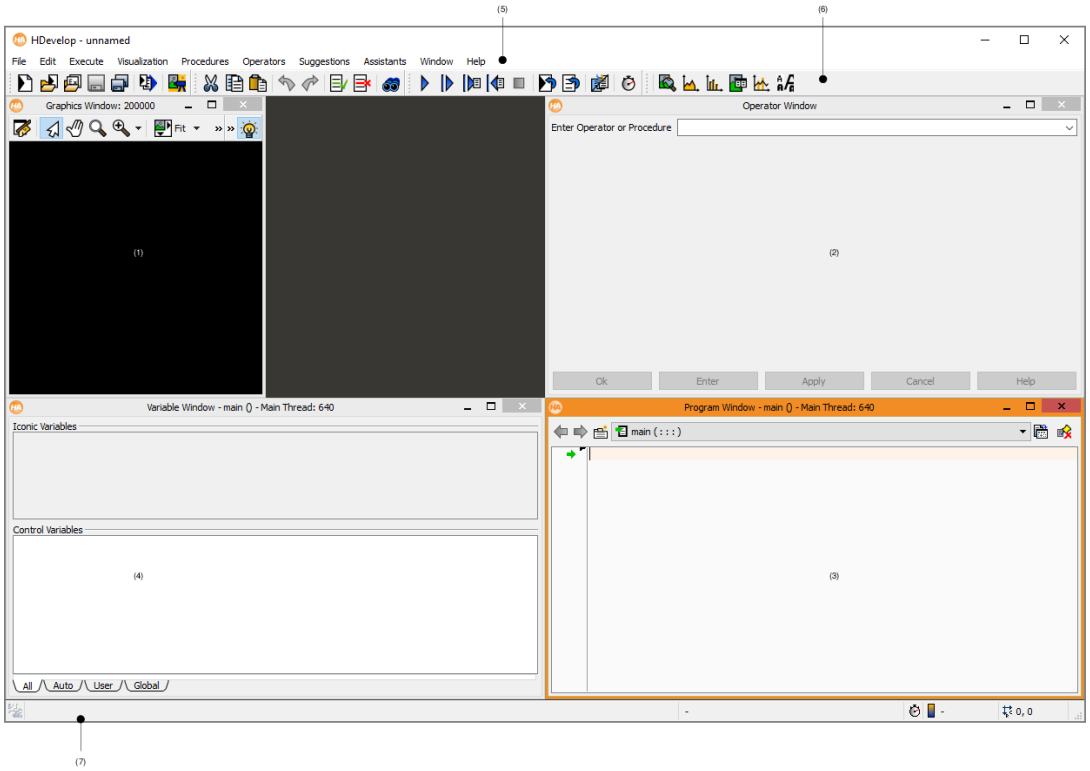


Figure 1.2: User interface: (1) graphics window, (2) operator window, (3) program window, (4) variable window, (5) menu, (6) toolbar, (7) status bar.

User Interface

When HDevelop is started for the first time, it looks similar to figure 1.2. The main window offers a menu (5) and a toolbar (6) for quick access to frequently used functions. The status bar (7) at the bottom of the window displays messages and image properties. In addition, the following windows are available by default:

1. Graphics window

   This window displays iconic data: images, regions, and XLDs. It provides its own toolbar to quickly zoom and pan the displayed image, and a context menu to adapt the visualization settings. HDevelop supports an arbitrary number of graphics windows.

2. Operator window

   You can select HALCON operators (and HDevelop procedures) in this window. The parameters of the selected operator can be specified, and the operator can be executed, entered in the current program, or both. You can also get online help for the selected operator from this window.

3. Program window

   This window displays the current program. It provides syntax highlighting with user-definable colors. The left column displays the program line numbers. The small black triangle is the insert

cursor, which is where new program lines will be added. In the following, it is referred to as IC. The green arrow is the program counter, which marks the next line to be executed. In the following, the program counter is referred to as PC. You can also add or remove breakpoints in the current program in this column. These will halt the program execution at user-defined places so that intermediate results may be examined.

When adding new lines or modifying existing lines, advanced autocompletion features help to speed up typing and to keep the program consistent. Program lines can also be modified by double-clicking them and editing them in the operator window.

4. Variable window

   Program variables can be watched in this window. It displays all variables of the current procedure and their current values. Iconic variables are displayed as thumbnails, whereas control variables are displayed as text. The layout of this window can be switched between horizontal and vertical splitting by double-clicking the separator. You can double-click iconic variables to display them in the active graphics window. Double-clicking control variables open an inspection window with a nicely formatted list of the current values and statistical data.

2 Running Example Programs

HALCON comes with a large number of HDevelop example programs from a variety of application areas. These range from simple programs that demonstrate a single aspect of HALCON or HDevelop to complete machine vision solutions. As an introduction to HDevelop, we recommend trying some of these programs to quickly get accustomed to the way HDevelop works.

The example program “Explore the Power of HALCON” demonstrates many different capabilities of HALCON in one program. It can be started from the start dialog, see figure 1.1 (2). Running this program is highly recommended to get a good overview of the many application areas of HALCON.

“Explore the Power of HALCON” starts up automatically when loaded from the start dialog. After loading it manually or loading one of the other example programs click `Run` (1) or press F5 to start it.

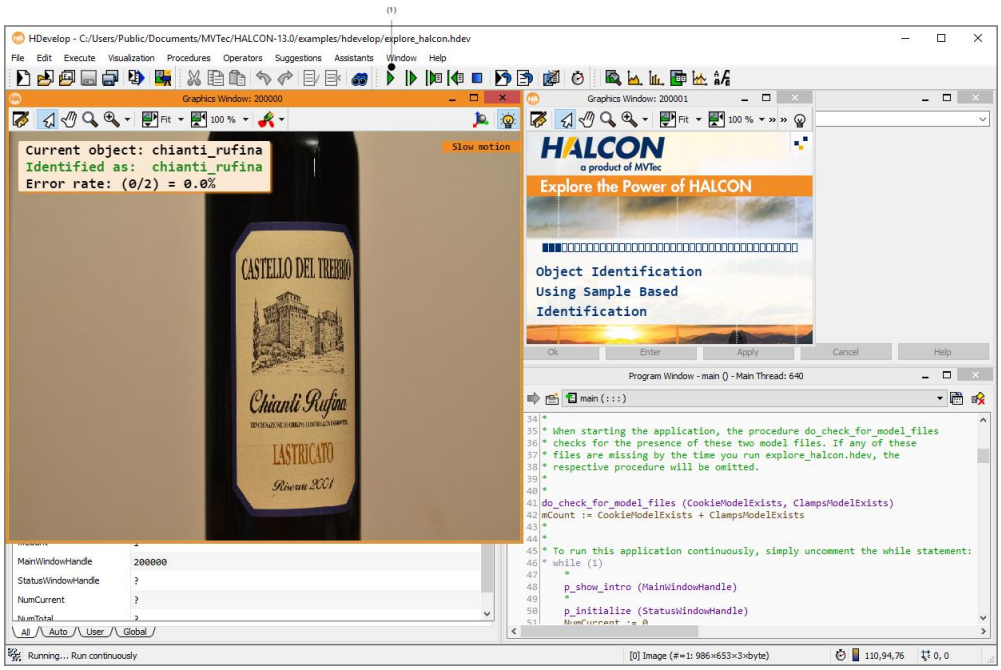


Figure 2.1: Explore the power of HALCON: Click (1) to run the program.

The example programs have been categorized by application area, industry, method, and operator usage. A special category “New in version” groups examples by their appearance in specific HALCON releases. Browsing these categories, you can quickly find example programs that cover image processing problems that you may wish to solve with HALCON. These programs may serve as a foundation for your own development projects.

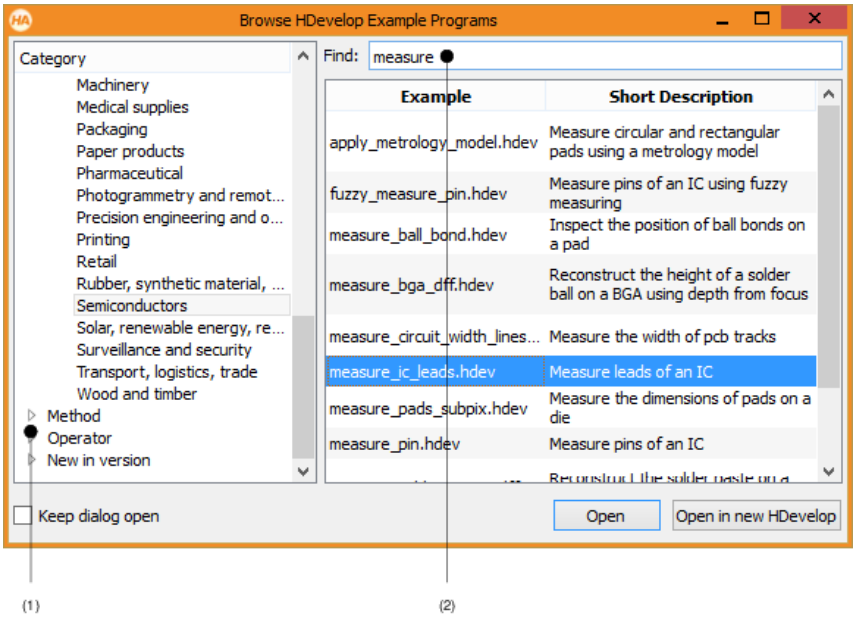


Figure 2.2: HDevelop program examples.

Browse and Load Example Programs

- Click `File` → `Browse HDevelop Program Examples...`

  This will open the example program browser (see figure 2.2). Similar to a file browser, it shows a tree of topics on the left and a list of example programs from the selected topics on the right. Subtopics can be toggled on or off by clicking the corresponding icon (1) or double-clicking the category name.

  Browse the categories: Click on a topic to select it and display its example programs. You can select multiple topics at once by holding the Ctrl key while clicking on the categories.

  Filter the example programs: To reduce the number of listed example programs, enter a word or substring into the `Find` text field (2). Subsequently, only example programs matching this substring in the file name or short description will be displayed.

  We pretend that you are looking for a measuring example from the semiconductor industry:

- Double-click on `Industry`.
- Click on the subtopic `Semiconductors`. The examples belonging to the semiconductor industry are listed on the right.
- Enter the word `measure` into the `Find` text field.

  Note how the list is updated as you type. Now, you have a short list of example programs to select from. You may need to resize the example browser to fully read the short descriptions of the listed programs.

- Select `measure_ic_leads.hdev` by clicking on it.

- Click `Open`. The selected example program is then loaded. Alternatively, you can load an example program by double-clicking on it. The example browser is closed unless `Keep dialog open` is checked.

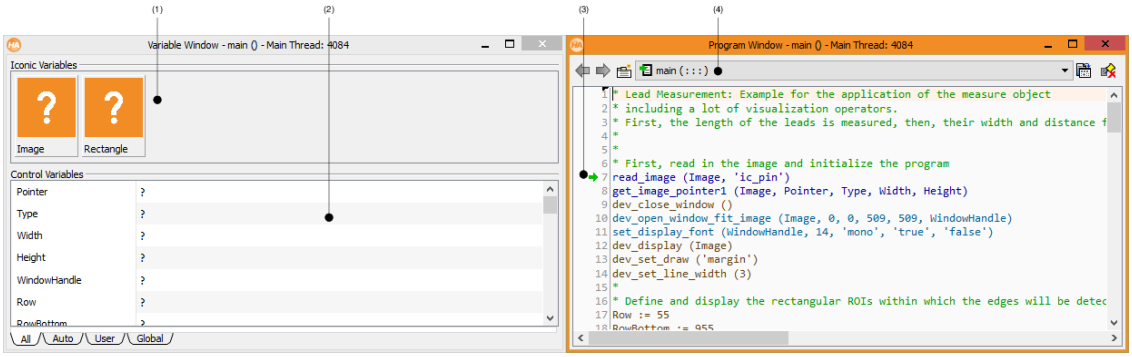


Figure 2.3: The variable and program window after loading the example program: (1) iconic variables, (2) control variables, (3) PC (program counter), (4) current procedure.

The program lines of the loaded example program are now displayed in the program window. The PC is set to the first executable line of the program (leading comments are ignored). The variable window is also updated: It lists the variables that are used in the main procedure, which is initially the current procedure. The variables are currently uninstantiated, i.e., their current value is undefined. This is indicated by the question mark (?). Both windows are displayed in figure 2.3.

Run Example Program

- Click `Execute` ⇒ `Run` or click the corresponding button (1) from the toolbar (see figure 2.4).

The program line next to the PC is executed, the PC is moved to the following line and so forth until the execution stops. There are four reasons for the program execution to stop: 1) the last program line has been executed, 2) a breakpoint has been reached, 3) the HDevelop instruction stop has been encountered as in this example, or 4) an error has occurred.

During execution, the graphics window is used for visualization. Changes to the variables are reflected in the variable window. When the program execution stops, the status bar displays the number of executed lines and the processing time.

To continue with the program execution, click `Execute` ⇒ `Run` again until the end of the program is reached.

- Click `Reset Program Execution` (4) to reset the program to its initial state. (see figure 2.6).
- Using the button `Step Over` (2) you can execute the program line by line and inspect the immediate effect of each instruction.

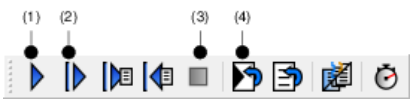


Figure 2.4: The basic execution buttons: (1) run continuously, (2) run step-by-step, (3) stop, (4) reset.

3 Image Handling

Image acquisition is crucial for machine vision applications. It will usually be an early if not the first step in your programming projects. This chapter explains how to load and view images within Hdevelop.

3.1 Reading Images From Files

Especially in the prototyping phase you often have a set of sample image files to work from. HDevelop (or rather the underlying HALCON library) supports a wealth of image formats that can be loaded directly (see read_image in the Reference Manual).

Drag-and-Drop

The easiest way to read an image is to simply drag it from a file browser to the HDevelop window and drop it there. When the file is dropped, HDevelop opens the dialog `Read Image` (see figure 3.1).

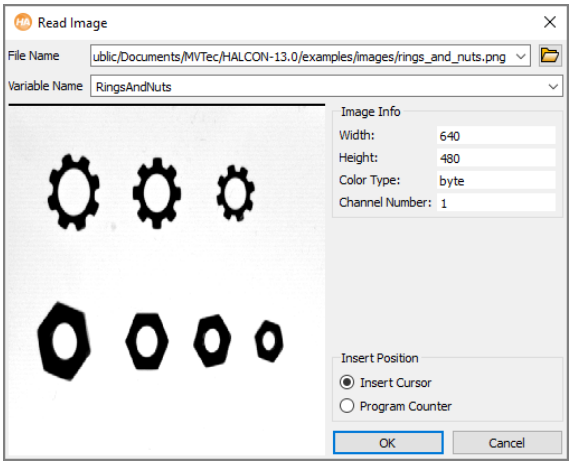


Figure 3.1: After dragging an image file onto the HDevelop window.

This dialog displays the full path of the image and automatically proposes a variable name derived from the file name. This name can be edited, or another iconic variable name from the current program may be selected from the drop-down list.

Furthermore, a preview of the image and basic image properties are displayed in the dialog (width, height, color type, and the number of channels). If you picked the wrong image, you can select another one from the same directory by pressing the button next to the file name. This will open a file browser native to the operating system, i.e., on Windows you may be able to switch to thumbnail view in this dialog. When another image is selected, the dialog is updated accordingly.

When you click the button `OK`, the instruction read_image is added to the current program. With the setting of `Insert Position`, you determine where the instruction will be put: At the IC or the PC. If you changed your mind about reading the selected image at all, click `Cancel`.

Drag-and-Drop of Multiple Images

You can also drag multiple images or directories containing multiple images to HDevelop. HDevelop will then open an image acquisition assistant with the images preselected.

Images from Selected Directories

Apart from dragging and dropping images, there is an equivalent method from within HDevelop: Select `File` ⇒ `Read Image...` to get the dialog described above. Browse to and select the desired image from this dialog, and click `OK` to add the selected image to your program.

3.2 Viewing Images

When images are read as described above, they are automatically displayed in the active graphics window. This is the default behavior in HDevelop, but the automatic display of images can be suppressed if desired, e.g., to speed up computationally intensive programs.

Initially, the loaded image fills the graphics window entirely. The window itself is not resized so the aspect ratio of the image might be skewed. Using the toolbox of the graphics window you can easily zoom the image, or change the window size with regard to the image.

We recommend adapting the window size to the size of the image because otherwise the display is slowed down. The image size, the window size and the displayed part of the image are set with the toolbar icons of the graphics window (see figure 3.2).

An iconic view of the loaded image is also displayed in the variable window. When the image is cleared in the graphics window, it can always be restored by double-clicking this icon.

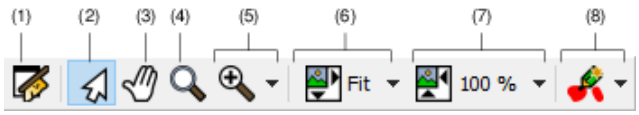


Figure 3.2: Tools in the graphics window: (1) clear window, (2) select, (3) pan image, (4) magnify, (5) zoom in/out, (6) adjust image size, (7) adjust window size, (8) ROI tools.

3.3 Image Acquisition Assistant

The image acquisition assistant is a powerful tool to acquire images from files (including AVI files), directories or image acquisition devices supported by HALCON through image acquisition interfaces. To use this assistant, select `Assistants` ⇒ `Open New Image Acquisition`. The window is displayed in figure 3.3. It features several tab cards that can be stepped through one after another. Ultimately, the assistant generates HDevelop code that can be inserted into the current program. Select the entry `Help` in the menu of the image acquisition assistant to open its online help.

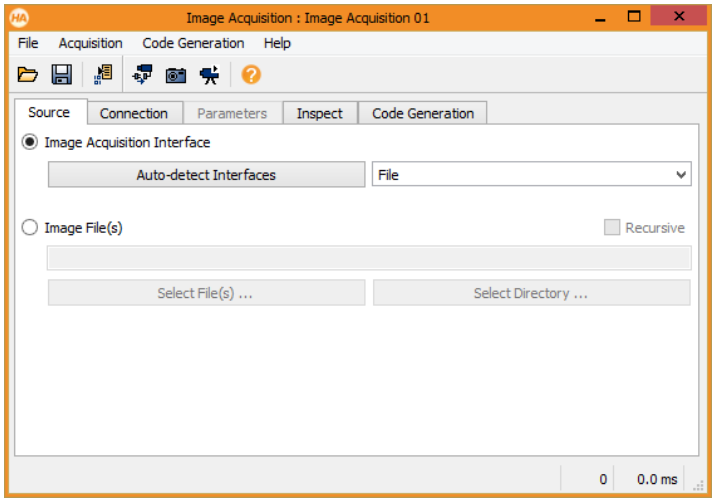


Figure 3.3: Image acquisition assistant.

The tab card `Source` determines the acquisition method and the image source. In the default setting images are acquired from files. This is described in the following section. Alternatively, images are acquired from an image acquisition device, e.g., a camera.

3.3.1 Acquiring Images from Files or Directories

You can specify a selection of image files or a directory to load images from. Make sure the radio button `Image File(s)` is selected in the tab card `Source`. You can directly enter image names or the name of a directory into the text field. Multiple image names are separated by a semicolon. Usually, it is more convenient to use one of the following buttons:

`Select File(s) ...`

HDevelop opens a file selection dialog in the current working directory, displaying the image files supported by HALCON. Multiple image files can be selected by holding down the Ctrl key while clicking additional image files. Click `Open` to confirm the selection. The first selected image is displayed in the active graphics window.

`Select Directory ...`

HDevelop opens a directory browser. It is not possible to select multiple directories. Confirm your selection by clicking `Open` or `OK`. The first image from the selected directory is displayed in the active graphics window. If the checkbox `Recursive` is ticked, all subdirectories of the specified directory are scanned for images as well.

View Images

You can single-step through the selected images by clicking the `Snap` button (see figure 3.4). Each time you click the button, the next image is displayed in the active graphics window. You can also loop through the images by clicking the button `Live`. This is especially useful for animations. Both functions are also available from the menu `Acquisition`.

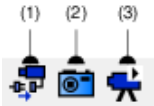


Figure 3.4: Browsing the selected images: (1) connect, (2) snap (single-step images), (3) live (continuous display).

Generate Code

Switch to the tab card `Code Generation`, and specify a variable name in the text field `Image Object`. You can later access the image in the program by this name. If multiple images or a directory were selected in the tab card `Source`, the image acquisition assistant will read the images in a loop. In this case, the following additional variable names need to be specified:

`Loop Counter`:

The name of the loop index variable. While looping over the images in the program, this variable will contain the object number of the current image.

`Image Files`:

The name of the variable that will contain the names of the selected images.

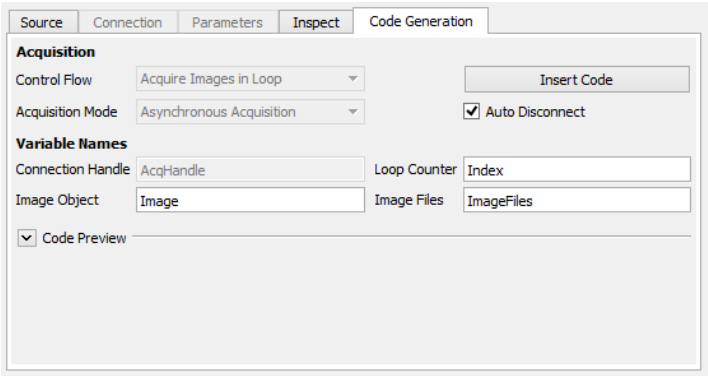


Figure 3.5: Specifying variable names for code generation.

Click `Code Preview` to inspect the code that would be generated from the currently specified parameters.

Click `Insert Code for Selection` to generate the code and insert it at the position of the IC in the current program.

The following piece of code is an example generated from three selected images. It is a self-contained HDevelop program that runs without alteration.

```
* Image Acquisition 01: Code generated by Image Acquisition 01
ImageFiles := []
ImageFiles[0] := 'W:/images/fin1.png'
ImageFiles[1] := 'W:/images/fin2.png'
ImageFiles[2] := 'W:/images/fin3.png'
```

```
for Index := 0 to |ImageFiles| - 1 by 1
   read_image (Image, ImageFiles[Index])
   * Image Acquisition 01: Do something
endfor
```

3.3.2 Modifying the Generated Code

After the generated code has been inserted into the program window, HDevelop internally keeps the code linked to the corresponding assistant. This link is kept until the assistant is quit using the menu entry `File` ⇒ `Exit Assistant`. If you close the assistant using the menu entry `File` ⇒ `Close Dialog` or using the close icon of the window, the assistant can be restored from the top of the menu `Assistants`.

You can change the settings inside the assistant and update the generated code accordingly. The code preview will show you exactly how the generated code lines will be updated. Furthermore, you can delete the generated code lines, or release them. When code lines are released, the internal links between the assistant and those lines are cut off. Afterward, the same assistant can generate additional code at a different place in the current program.

4 Programming HDevelop

This chapter explains how to use HDevelop to develop your own machine vision applications. It is meant to be actively followed in a running instance of HDevelop. In the following, it is assumed that the preferences of HDevelop are set to the default values. This is always the case after a fresh installation of HALCON. If you are uncertain about the current settings, you can always start HDevelop with the default settings by invoking it from the command line in the following way:

```
hdevelop -reset_preferences
```

This chapter deals with a simple example. Given is the image displayed in figure 4.1. The objective is to count the clips and determine their orientation.

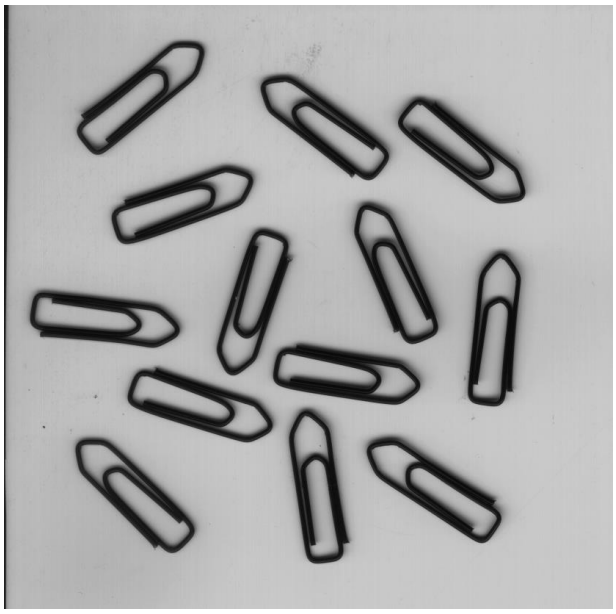


Figure 4.1: Paper clips.

4.1 Start a New Program

Start HDevelop or, if it is still running, click `File` ⇒ `New Program` to start a new program. HDevelop will notify you if there are unsaved changes in the current program. If it does, click `Discard` to throw away the changes and start anew. In case you rearranged the windows, click `Window` ⇒ `Organize Windows` to restore the default layout displayed in figure 1.2.

The first thing to do is read the image and store it in an iconic variable. From the last chapter, we know that we can simply drag an image to the HDevelop window. We also know that this inserts the operator read_image into the program. Therefore, we can just as well insert the operator directly.

4.2 Enter an Operator

Click into the text box of the operator window, type `read_image`, and press Return. You can also type any partial operator name and press Return. HDevelop will then open a list of operators matching that partial name. This way, you can easily select operators without having to type or even know the exact name. Selection is done with the mouse or using the arrow keys to highlight the desired operator and pressing Return. If you selected the wrong operator by accident, you can reopen the list by clicking on the drop-down arrow next to the operator name. When entering a partial name, operators commencing with that name appear at the top of the list.

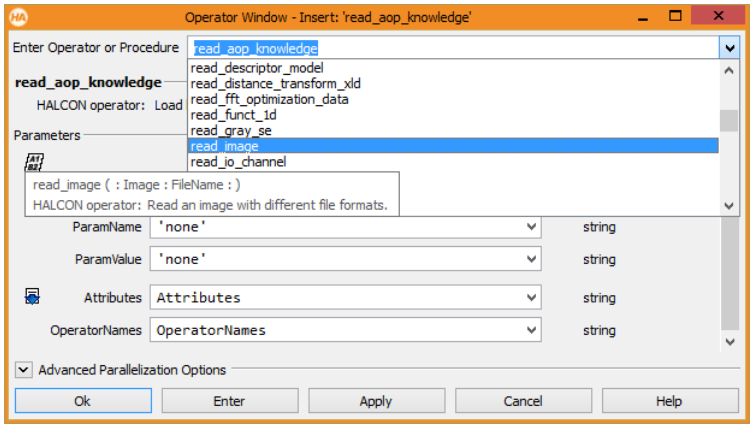


Figure 4.2: Matching operators after typing `read_` and pressing Return.

4.3 Specify Parameters

After selecting an operator, its parameters are displayed in the operator window. They are grouped by iconic and control parameters. The icons next to the parameter names denote the parameter type: Input (2) vs. output (1) (see figure 4.3). The semantic type is displayed to the right of the parameters. Parameters are specified in the text fields. The first parameter gets the input focus.

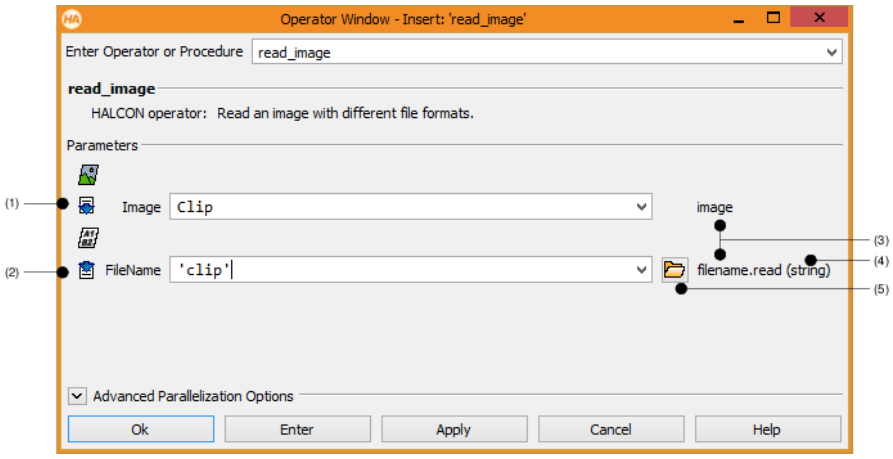


Figure 4.3: Specifying parameters: (1) iconic output parameter, (2) control input parameter, (3) semantic type, (4) data type, (5) file selection dialog.

Enter `Clip` into the text field `Image`. The image will be stored in this variable. Next, enter `'clip'` into the text field `FileName`. You can press Tab to go to the next input field. Pressing Shift+Tab takes you back to the previous field. This way you can enter all parameters without using the mouse.

Click `OK` to add the operator to the current program and execute it. This will do the following:

- An operator call is added as the first line of the current program.
- The IC is advanced so that additional lines will be added after the inserted line.
- The character `*` is added to the window title to indicate unsaved changes in the current program. The current procedure (main) is also marked with `*` in the program window.
- The program line is executed and the PC is advanced. To be more precise: All the lines from the PC to the IC are executed which makes a difference when adding program lines in larger programs.
- The image is displayed in the graphics window.
- The status bar is updated, i.e., the execution time of the operator read_image is displayed and the format of the loaded image is reported.
- The output variable `Clip` is created and displayed in the variable window.
- The operator window is cleared and ready for the insertion of the next operator.

4.4 Getting Help

You may be wondering where the clip image was loaded from since we did not specify any path or even a file extension. This is a detail that is related to the way the HALCON operator read_image works. HDevelop does not know anything about it. It just executes the operator with the parameters you supply. Accessing the operator documentation from within HDevelop is very easy.

Double-click the first program line in the program window. The operator is displayed in the operator window for editing. Now click `Help` to open the HDevelop online help window. It will automatically jump to the documentation of the displayed operator. The reference manual is completely cross-linked. The navigation at the left part of the window provides quick access to the documentation. The tab card `Contents` presents the hierarchical structure of the documentation. The tab card `Operators` lists all operators for direct access. Enter any desired substring into `Find` to quickly find an operator.

In the remainder of this chapter, try to use the online help as much as possible to get information about the used operators. The online help window is described in the section “Help Window”.

4.5 Add Additional Program Lines

Select the clips by thresholding

Now, we want to separate the clips from the background, i.e., select them. They clearly stand out from the background, thus a selection based on the gray value is appropriate. This operation is known as thresholding.

Enter `threshold` into the operator window. This is both the full name of an operator and part of other operator names. Thus, you get a list of matching operators with threshold pre-selected when you press Return. Press Return once more to confirm the selected operator and show its parameters.

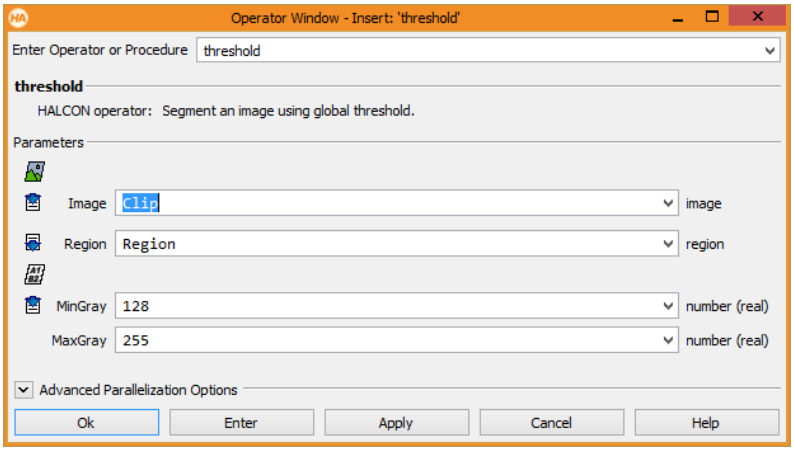


Figure 4.4: Parameter suggestions.

In figure 4.4 you can see that the input parameter Image is set to `Clip` automatically. For input variables with no default value, reasonable suggestions are inferred automatically by collecting previous output variables of the same type. Therefore, the name of the most recent matching output parameter will be suggested (most recent being the closest predecessor of the current program line). In this example, only `Clip` is available.

Set MinGray and MaxGray to `0` and `30`, respectively. This will select the dark pixels in the image.

Click `Apply`. This button executes the operator without adding it to the program. Additionally, it keeps the current parameters open for editing. This way, you can easily try different settings and immediately see the result. The selected pixels (the so-called *region*) are stored in the output variable `Region`, which is displayed in the variable window. The region is an image mask: White pixels are selected while black pixels are not.

The region is also displayed as an overlay in the graphics window. The selected pixels are displayed in red (unless you changed the default settings).

The selected threshold values are not perfect, but we will correct this later. For now, click `Enter` to add the operator to the program window. Contrary to clicking `Ok`, this does not execute the operator. Note that the variable `Region` keeps its value but is no longer displayed in the graphics window. Also, the PC is not advanced, indicating that the second line of the program is yet to be executed.

Adding program lines with `Enter` is especially useful if some of the input parameters use variable names that will be added to the program at a later time.

Successor
Click on the just inserted program line to select it. You can let HDevelop suggest operators based on the selected line. Open the menu `Suggestions` ⇒ `Successors`. This menu is filled dynamically to

show typical successors of the currently selected operator. We want to split the selected pixels into contiguous regions. Move the mouse pointer over the menu entries. The status bar displays a short description of the highlighted operator. Looking through the menu entries, the operator connection looks promising, so we click on it. Any operator selected through this menu is transferred to the operator window.

Again, the variable names suggested by HDevelop look reasonable, so click `Ok`. This time, two program lines are executed: The threshold operation and the connection operation. As noted above: Clicking `Ok` executes from the PC to the IC.

In the graphics window, the contiguous regions calculated by the operator connection are now displayed in alternating colors.

4.6 Understanding the Image Display

After having executed the three lines of our program, the graphics window actually displays three layers of iconic variables: the image `Clip`, the region `Region`, and the tuple of regions `ConnectedRegions` (from bottom to top). Place the mouse pointer over the icons in the variable window to obtain basic information about the variables.

The display properties of images and the topmost region can be adjusted from the context menu of the graphics window. For images, the look-up table (henceforth called LUT) and the display mode (referred to as “paint”) can be set. The LUT specifies gray value mappings. Experiment with different settings: Right-click in the graphics window and select some values from the menus `Lut` and `Paint`. Make sure, the menu entry `Apply Changes Immediately` is checked. Notice how the display of the image changes while the regions remain unchanged.

The menu entries `Colored`, `Color`, `Draw`, `Line Width`, and `Shape` change the display properties of the topmost region. Set `Draw` to `margin', `Color` to `cyan', and `Shape` to `ellipse'. The display of `ConnectedRegions` (which is the topmost layer) changes accordingly. The region `Region` is still displayed in filled red.

A more convenient way to set many display properties at once is available through the menu entry `Set Parameters`. It opens the settings window displayed in figure 4.5.

After trying some settings, click the button `Reset` to restore the default visualization settings.

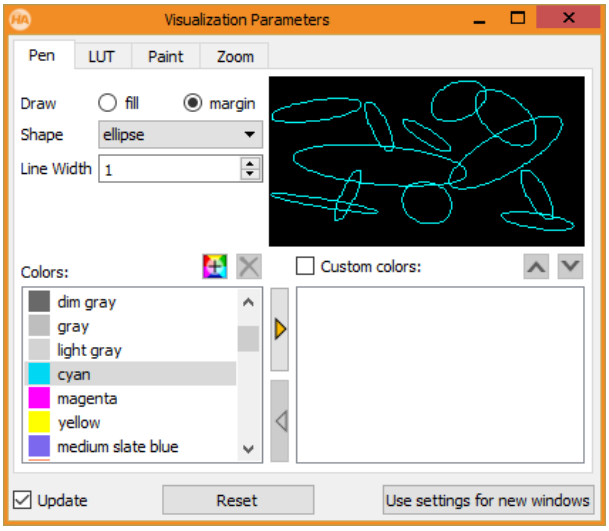


Figure 4.5: Changing the display parameters.

You cannot change the display properties of regions (or XLDs) other than the topmost. What you can do is rebuild the image stack in the graphics window manually by double-clicking iconic variables in the variable window and changing the properties each time another layer is added. The stack is cleared whenever an image is added that uses the full domain. To clear the stack (and thus the graphics window) manually, click the clear icon (see figure 3.2).

4.7 Inspecting Variables

When you move the mouse cursor over the variable `ConnectedRegions` you see that it contains 98 regions.

Right-click on the icon `ConnectedRegions` and select `Clear / Display` to display only the connected regions in the graphics window. Right-click again and select `Display Content` ⇒ `Select...`. This menu entry opens a variable inspection window which lists the contents of the variable `ConnectedRegions`. The selected region of this inspection window is displayed in the graphics window using the current visualization settings. Set `Draw` to `margin' and `Shape` to `ellipse' and select some regions from the list. An example is illustrated in figure 4.6.

For now, close the variable inspection window. The large number of regions is due to the coarse setting of the bounds of the threshold operator. In the following, we will use one of HDevelop's visual tools to find more appropriate settings interactively.

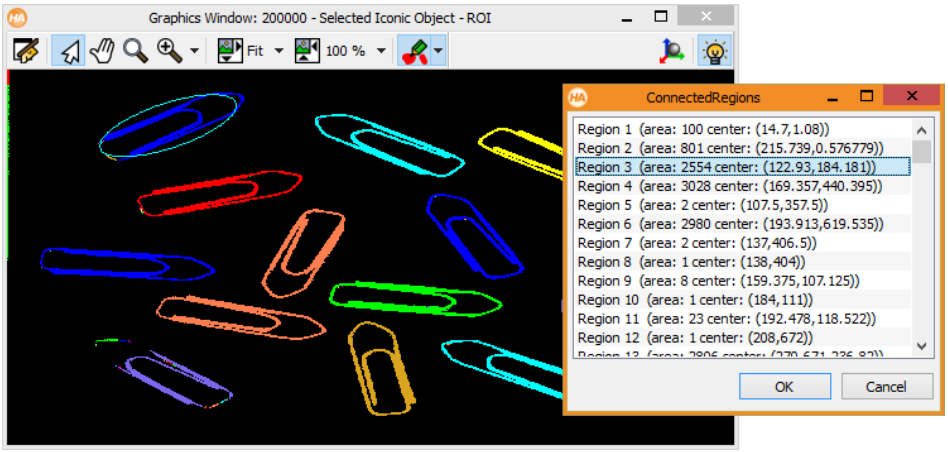


Figure 4.6: Interactive inspection of an iconic variable containing regions

4.8 Improving the Threshold Using the Gray Histogram

Click `Visualization/Tools` → `Gray Histogram` to open a tool for the inspection of gray value histograms. One of its uses is to determine threshold bounds visually. Because the graphics window currently displays only regions, the gray histogram is initially empty. Double-click the `Clip` icon in the variable window to re-display the original image and watch its gray histogram appear.

Make sure `Range Selection and Code Generation` is visible in the histogram window (1), see figure 4.7. Select `Threshold` in the column `Operation` of the gray histogram window, and click the icon next to `Threshold` (2) to visualize the operation. Now, you can try different threshold bounds by altering the values in `Min` and `Max` or by dragging the lines (3) in the histogram area. Any changes to these values are immediately visualized in the active graphics window. The values 0 and 56 seem suitable for the lower and upper bounds, respectively.

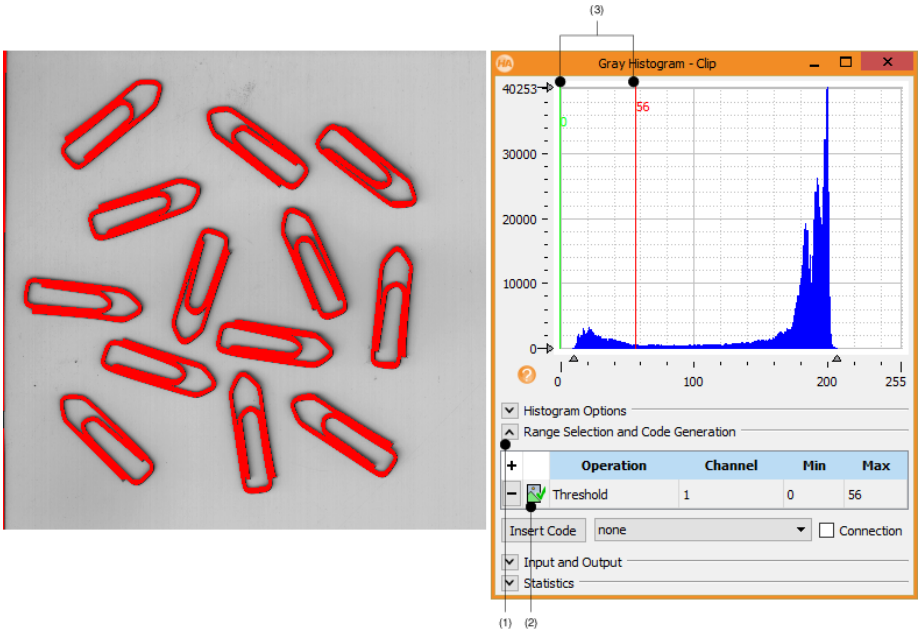


Figure 4.7: Determining threshold bounds interactively using the gray histogram.

4.9 Edit Lines

To edit a line in the operator window, double-click it in the program window. If you make changes to the parameters and click `Ok` or `Replace`, the original line in the program is updated. You can also edit the program directly in the program window (see section "Editing Programs").

Double-click the second line of the program to adjust the threshold operation. Replace the value 30 with 56 and click `Replace`. The program line is updated in the program window.

4.10 Re-execute the Program

The last editing step was just a tiny modification of the program. Often, after editing many lines in your program with perhaps many changes to the variables you want to reset your program to its initial state and run it again to see the changes.

Click `Execute` → `Reset Program Execution` to reset the program. Now, you can select `Execute` → `Run` to run the complete program, or click `Execute` → `Step Over` repeatedly to execute the program line by line.

4.11 Save the Program

Perhaps now is a good time to save your program. Select `File` → `Save` and specify a target directory and a file name for your program.

4.12 Selecting Regions Based on Features

Inspecting the variable `ConnectedRegions` after the changed threshold operation yields a much better result. Still, a contiguous area at the left edge of the image is returned (1), see figure 4.8. To obtain only the regions that coincide with the clips, we need to further reduce the found regions based on a common criterion. Analogous to the gray histogram tool, which helps to select regions based on common gray values, HDevelop provides a feature histogram tool, which helps to select regions based on common properties or features.

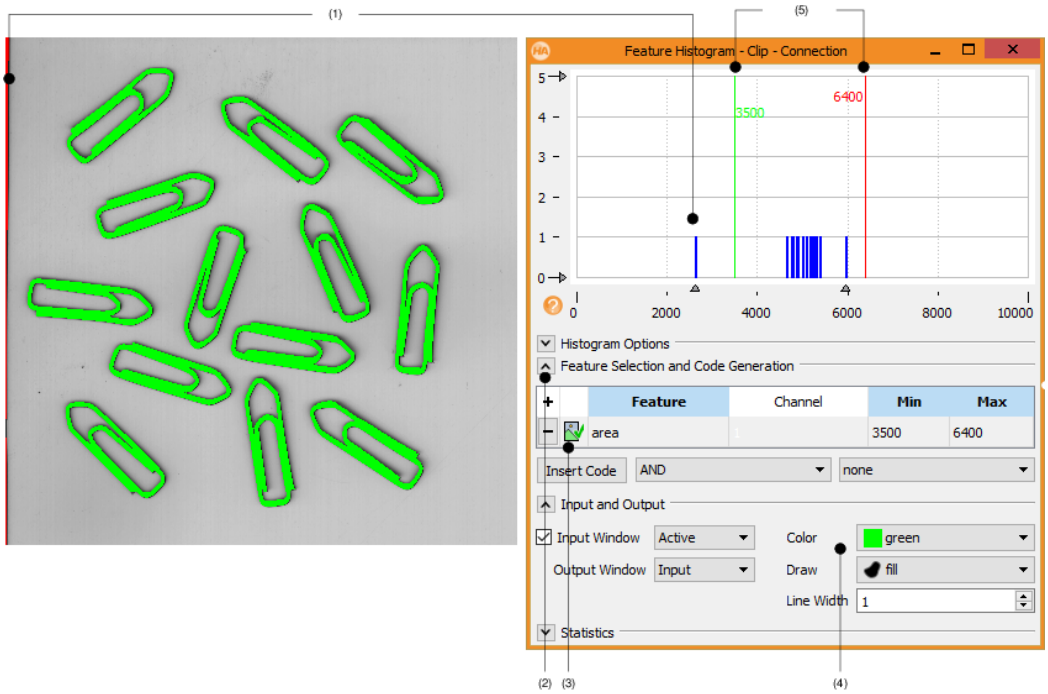


Figure 4.8: Selecting regions with a similar area in the feature histogram.

Click `Visualization/Tools` → `Feature Histogram` to open the tool. Make sure `Feature Selection and Code Generation` is visible in the histogram window (2). The column `Feature` allows selecting the feature that the region selection will be based on. The default feature is "area", which is adequate in this case: The actual clips are all the same size, thus the area of the regions is a common feature. In the feature histogram, the horizontal axis corresponds to the values of the selected feature. The vertical axis corresponds to the frequency of certain feature values.

Similar to the gray histogram window, you can visualize the selected regions, i.e., the regions whose area falls between the values `Min` and `Max`, which are represented by the green and red vertical lines, respectively. Click the icon next to the selected feature (`area`) (3) to enable the visualization.

Specify the parameters in the `Input and Output` section of the feature histogram window (4). Drag the green and red line (5) to see how this affects the selected regions. In the histogram, we can see that in order to cover all the clips, we can safely select regions whose area is between, say, 4100 and the maximum value in the histogram. When you are satisfied with the selection, click the button

`Insert Code`. The following line (with similar numeric values) will be added to your program at the position of the IC:

```
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 4100, 5964)
```

Run the program, and inspect the output variable `SelectedRegions`. The regions corresponding to the clips are now determined correctly. To obtain the orientation and the center of gravity of the clips, add the following operator calls to the program:

```
orientation_region (SelectedRegions, Phi)
area_center (SelectedRegions, Area, Row, Column)
```

The operator orientation_region returns a tuple of values: For each region in `SelectedRegions`, a corresponding orientation value in `Phi` is returned. The operator area_center, in the same way, returns the area, row and column of each input region as tuples. Again, run the program and inspect the calculated control variables. You can inspect multiple control variables in one inspection window. This is especially useful if the control variables all relate to each other as in this example. In the variable window select all control variables (hold down the Ctrl key), and right-click `Inspect` (see figure 4.9).

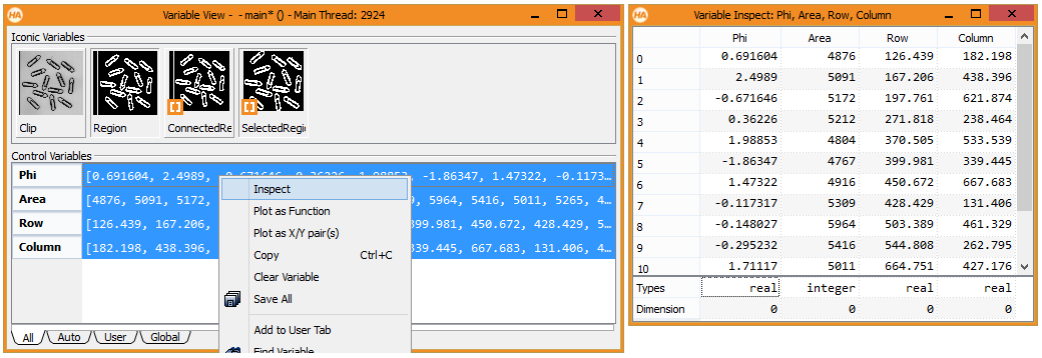


Figure 4.9: Inspecting control variables.

4.13 Open Graphics Window

Up until now, the visualization of iconic results relied on the fact that a graphics window is always opened by default when HDevelop starts. It is good practice to explicitly open a distinct number of graphics windows in your programs.

Add the following lines before the first line of your program:

```
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
```

The first line closes any open graphics windows to ensure a consistent state at the start of the program. The second line opens a single graphics window that can be referenced by the output variable `WindowHandle`. The window handle is a numeric value that is also displayed in the title bar of the corresponding graphics window.

4.14 Looping Over the Results

Being an integrated development environment, HDevelop provides features found in other programming languages as well: Variable assignment, expressions, and control flow. Variable assignment and control flow are implemented in terms of specific HDevelop operators. These operators can be selected from the menu `Operators` → `Control`. Expressions are implemented in terms of a specific HDevelop language which can be used in input control parameters of operator calls.

To iterate over the elements in `Phi`, we use a for loop which counts from zero (the index of the first element of a tuple) to the number of elements minus one. The for loop is entered just like a common HALCON operator: Enter `for` into the operator window and specify the parameters as in figure 4.10. Note that the closing endfor is entered automatically if the corresponding checkbox is ticked. Also, note that the IC is placed between the added lines so that the body of the loop can be entered.

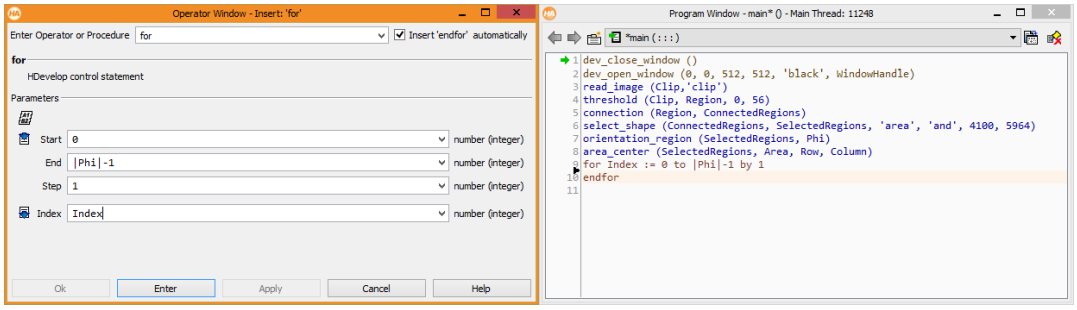


Figure 4.10: Entering a loop in HDevelop.

The notation `|Phi| - 1` is part of the HDevelop language. This operation calculates the number of elements in `Phi` minus one. When inserted in the program window, the operator for is displayed in a different format to make it more readable.

Add the following instruction to the program. It is automatically indented in the program window to highlight the nesting inside the for loop. The backslash at the end of the first line allows the program line to continue at the next line in the program window for readability. You can either enter the instruction as displayed or in a single long line without the backslash.

```
dev_disp_text (deg(Phi[Index]) + ' degrees', 'image', Row[Index], \
        Column[Index], 'black', [], [])
```

The operator dev_disp_text provides a convenient way to output text at specific positions in the activated graphics window. Press F1 to get more information about the meaning of the parameters. The notation `Phi[Index]` is another operation of the HDevelop language. It provides access to a single value of a tuple. The function `deg` is part of the HDevelop language. It converts its argument from radians to degrees. In this example, the operation `+` performs a string concatenation because the argument `' degrees'` is a `string` value. Before the two operands of `+` are concatenated, an automatic type conversion (`double` to `string`) of the numeric argument takes place.

Please note that the loop around dev_disp_text was chosen to illustrate how to access single elements of a tuple. In this specific example it is not really required because dev_disp_text is smart enough to directly operate on tuples. Instead of the loop, you could simply use the following call:

```
dev_disp_text (deg(Phi) + ' degrees', 'image', Row, Column, 'black', [], [])
```

4.15 Summary

This is basically the way to create programs in HDevelop. Select an operator, specify its parameters, try different settings using the button `Apply`, add a new program line using `Enter` or `Ok`, and edit it later by double-clicking it in the program window. Use the interactive tools provided by HDevelop to assist you, e.g., to find appropriate values for the operators.

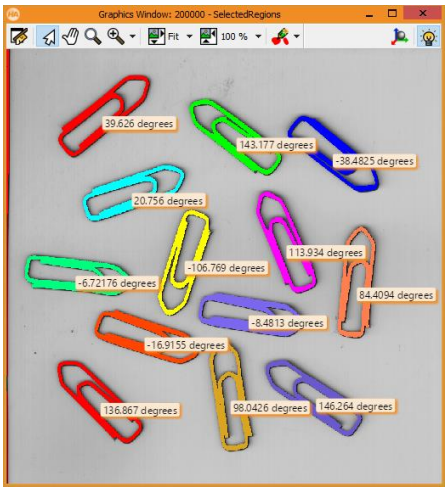


Figure 4.11: Final result of the example program.

5 Calibration and Location examples

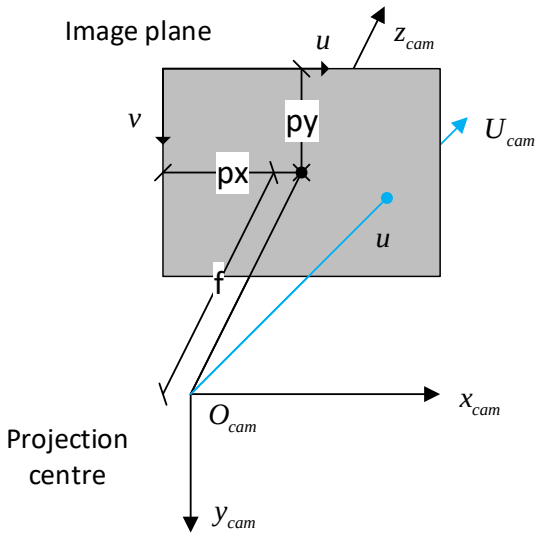
In order to use computer vision in combination with robots, your system has to be prepared. At first you need to calibrate your camera. In the field of computer vision, you have many tasks where you don't need a calibrated camera, but when you want to combine it with a robot, calibration is vital. After that can look for your object to grasp in your camera picture. So, this chapter is split in two topics: Calibrating the camera and Locating the desired object. Camera calibration

The camera calibration is also split in two smaller parts. The first one is the intrinsic calibration, where relation between the image plane and the camera coordinate system is described. The other one is the extrinsic calibration, that describe where the camera coordinate system is located in the world. If you want to know more about that topic, look for the help on the "calibrate_cameras" operator in the HDevelop help or in the common literature, for example "Robotics, Vision and Control" from Peter Corke.

5.1.1 Intrinsic calibration

The Intrinsic calibration describes the relation between the image plane and the camera coordinate system. It only depends on the combination of camera and lens and not on the positioning of the camera. It consists of (at least) five parameters:

- Camera constant (distance between projection centre and image plane)
- Principal point (x/y)
- Scale factor (CCD/CMOS-pixel cells per unit)
- Can be extended to consider distortion as well (also 5 parameters)



Example:

You can copy the following code to the program window. The first few lines just set the environment up

```
ImgPath := '3d_machine_vision/calib/'
dev_close_window ()
dev_open_window (0, 0, 652, 494, 'black', WindowHandle)
dev_update_off ()
dev_set_draw ('margin')
dev_set_line_width (3)
OpSystem := environment('OS')
set_display_font (WindowHandle, 14, 'mono', 'true', 'false')
```

The next few lines start a initial camera object and load the description file of the calibration plate. In there the shape and the number of calibration marks on the calibration object are denoted.

```
gen_cam_par_area_scan_division (0.016, 0, 0.0000074, 0.0000074, 326, 247, 652, 494, StartCamPar)
create_calib_data ('calibration_object', 1, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], StartCamPar)
set_calib_data_calib_object (CalibDataID, 0, 'caltab_30mm.descr')
```

Here we use 10 images to ensure to get a good calibration. Within each image the calibration pattern is searched and the found pattern is displayed

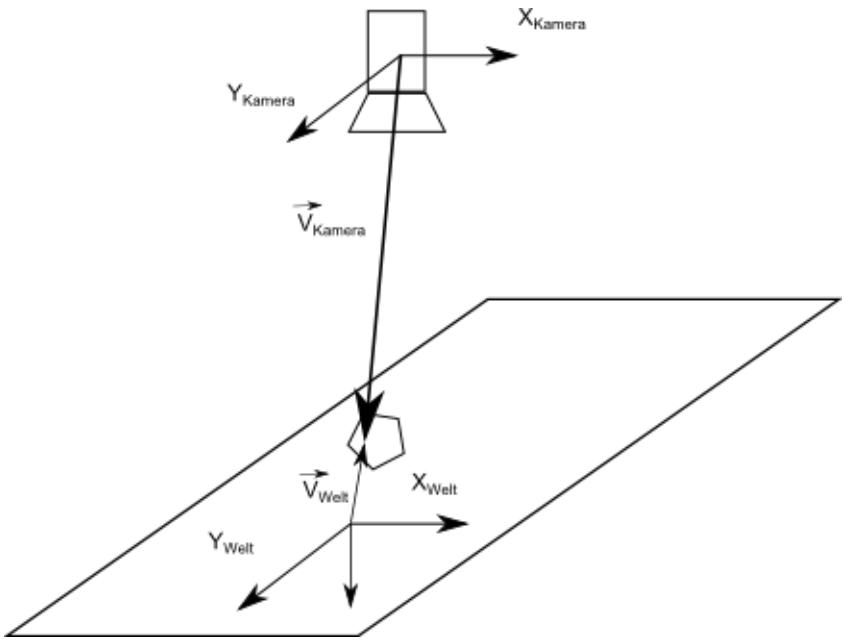
```
NumImages := 10
* Note, we do not use the image from which the pose of the measurement plane can be derived
for I := 1 to NumImages by 1
   read_image (Image, ImgPath + 'calib_' + I$'02d')
   dev_display (Image)
   find_calib_object (Image, CalibDataID, 0, 0, I, [], [])
   get_calib_data_observ_contours (Caltab, CalibDataID, 'caltab', 0, 0, I)
   dev_set_color ('green')
   dev_display (Caltab)
endfor
```

Now the calibration itself is triggered. After this step you can see the real intrinsic parameters in the “CamParam” variable. These parameters are saved to 'camera_parameters.dat', and the no longer needed Calibration Object gets disposed.

```
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
* Write the internal camera parameters to a file
write_cam_par (CamParam, 'camera_parameters.dat')
Message := 'Interior camera parameters have'
Message[1] := 'been written to file'
disp_message (WindowHandle, Message, 'window', 12, 12, 'red', 'false')
clear_calib_data (CalibDataID)
```

5.1.2 Extrinsic calibration

The extrinsic calibration describes the position and orientation of the camera coordinate system in relation to the world coordinate system. It only depends on the positioning of the camera. In this short course we only discuss the case when the camera is stationary to the scene.



Example:

```
ImgPath := '3d_machine_vision/calib/'
dev_close_window ()
dev_open_window (0, 0, 652, 494, 'black', WindowHandle)
dev_update_off ()
dev_set_draw ('margin')
dev_set_line_width (1)
set_display_font (WindowHandle, 14, 'mono', 'true', 'false')

* Read the internal camera parameters from file
try
   read_cam_par ('camera_parameters.dat', CamParam)
catch (Exception)
   * run 'camera_calibration_internal.hdev' first to generate camera
   * parameter file 'camera_parameters.dat'
   stop ()
endtry
*
* Determine the external camera parameters and world coodinates from image points
*
* The external camera parameters can be determined from an image, where the
* calibration plate is positioned directly on the measurement plane
read_image (Image, ImgPath + 'calib_11')
dev_display (Image)
```

```
CaltabName := 'caltab_30mm.descr'
create_calib_data ('calibration_object', 1, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], CamParam)
set_calib_data_calib_object (CalibDataID, 0, CaltabName)
find_calib_object (Image, CalibDataID, 0, 0, 1, [], [])
get_calib_data_observ_contours (Caltab, CalibDataID, 'caltab', 0, 0, 1)
get_calib_data_observ_points (CalibDataID, 0, 0, 1, RCoord, CCoord, Index, PoseForCalibrationPlate)
dev_set_color ('green')
dev_display (Caltab)
dev_set_color ('red')
disp_caltab (WindowHandle, CaltabName, CamParam, PoseForCalibrationPlate, 1)
dev_set_line_width (3)
disp_circle (WindowHandle, RCoord, CCoord, gen_tuple_const(|RCoord|,1.5))
* caltab_points (CaltabName, X, Y, Z)
* calibrate_cameras (CalibDataID, Error)
```

To take the thickness of the calibration plate into account, the z-value of the origin given by the camera pose has to be translated by the thickness of the calibration plate. Deactivate the following line if you do not want to add the correction.

```
set_origin_pose (PoseForCalibrationPlate, 0, 0, 0.00075, PoseForCalibrationPlate)
pose_to_hom_mat3d (PoseForCalibrationPlate, HomMat3D)
clear_calib_data (CalibDataID)
stop ()
```

5.2 Locating Objects

In this Chapter we look at two examples for searching an object in a picture. One example is a two-dimensional search for a pattern, the other one is a three-dimensional search for a deformed pattern. For this task, again, you can choose between a wide variety of possible algorithms, for example correlation-based, shape-based or perspective deformable matching algorithms. For the purpose of this short course we focus on a simple shape-based approach. The model is here obtained by defining a region of interest in the training image where the contours are well displayed. Within this region, the model shape is retrieved by edge-detecting algorithms. Afterwards this shape is searched for in the camera picture.

Example:

First set up the graphical environment

```
dev_update_pc ('off')
dev_update_window ('off')
dev_update_var ('off')
```

read the training image and prepare some annotations

```
read_image (Image, 'green-dot')
get_image_size (Image, Width, Height)
dev_close_window ()
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_set_color ('red')
dev_display (Image)
```

separate the logo from the background

```
threshold (Image, Region, 0, 128)
connection (Region, ConnectedRegions)
```

select the logo and create the region of interest (ROI) out of it

```
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 10000, 20000)
fill_up (SelectedRegions, RegionFillUp)
dilation_circle (RegionFillUp, RegionDilation, 5.5)
```

reduce the image to the size of the ROI

```
reduce_domain (Image, RegionDilation, ImageReduced)
```

create the model and move the shape model to the position of the ROI

```
create_scaled_shape_model (ImageReduced, 5, rad(-45), rad(90), 'auto', 0.8, 1.0, 'auto', 'none', 'ig-
nore_global_polarity', 40, 10, ModelID)
get_shape_model_contours (Model, ModelID, 1)
area_center (RegionFillUp, Area, RowRef, ColumnRef)
vector_angle_to_rigid (0, 0, 0, RowRef, ColumnRef, 0, HomMat2D)
affine_trans_contour_xld (Model, ModelTrans, HomMat2D)
```

Display the resulting model

```
dev_display (Image)
dev_display (ModelTrans)
```

Load the image and look for the green dots. After that display each of the found positions.

```
read_image (ImageSearch, 'green-dots')
dev_display (ImageSearch)
find_scaled_shape_model (ImageSearch, ModelID, rad(-45), rad(90), 0.8, 1.0, 0.5, 0, 0.5, 'least_squares', 5,
0.8, Row, Column, Angle, Scale, Score)
for I := 0 to |Score| - 1 by 1
   hom_mat2d_identity (HomMat2DIdentity)
   hom_mat2d_translate (HomMat2DIdentity, Row[I], Column[I], HomMat2DTranslate)
   hom_mat2d_rotate (HomMat2DTranslate, Angle[I], Row[I], Column[I], HomMat2DRotate)
   hom_mat2d_scale (HomMat2DRotate, Scale[I], Scale[I], Row[I], Column[I], HomMat2DScale)
   affine_trans_contour_xld (Model, ModelTrans, HomMat2DScale)
   dev_display (ModelTrans)
endfor
clear_shape_model (ModelID)
```

Example2:

This example shows an application case from the automotive industry. The task is to locate a car door put in different orientations and positions by using the calibrated deformable matching. First, a model is defined from a planar area of the car door, then a calibrated pose is calculated from the calibration plate put in the plane of the initial object. For all subsequent takes the pose can then be determined using the deformable matching. It is important to note that the perspective matching can only be used for planar areas of your object of interest that consist of well defined contours. Further a stable detection requires a perspective distortion of the object. Hence if the object is very far away or the camera has a long focal length, the projection is becoming affine making a stable pose estimation difficult. While the

translational part of the pose estimation is typically very stable, the estimation of the rotational part becomes difficult when looking exactly perpendicular at the object. In these cases, create_aniso_shape_model should be preferred.

```
dev_update_off ()
dev_close_window ()
read_image (Image, 'automotive/car_door_calib_plate')
get_image_size (Image, Width, Height)
dev_open_window_fit_size (0, 0, Width, Height, -1, -1, WindowHandle)
set_display_font (WindowHandle, 14, 'mono', 'true', 'false')
dev_display (Image)
```

Compute calibrated pose of object

```
CalTabDescrName := 'caltab_200mm.descr'
gen_cam_par_area_scan_division (0.0160522, -402.331, 9.30632e-006, 9.3e-006, 315.431, 273.525, 640, 512,
CamParam)
caltab_points (CalTabDescrName, X, Y, Z)
find_caltab (Image, Caltab, CalTabDescrName, 3, 112, 5)
find_marks_and_pose (Image, Caltab, CalTabDescrName, CamParam, 128, 10, 18, 0.9, 15, 100, RCoord,
CCoord, Pose)
gen_cross_contour_xld (Cross, RCoord, CCoord, 6, 0.785398)
dev_set_draw ('margin')
dev_set_line_width (1)
dev_set_color ('cyan')
dev_display (Cross)
dev_set_line_width (2)
dev_set_color ('green')
dev_display (Caltab)
disp_continue_message (WindowHandle, 'black', 'true')
stop ()
```

Create model

```
read_image (Image, 'automotive/car_door_init_pose')
dev_display (Image)
* Select a planar sub part of the car door as model region
read_region (ROI, 'automotive/car_door_region')
reduce_domain (Image, ROI, ImageReduced)
* The plane of the object is approximately the same as the calibration plate.
* However, there is the depth of the calibration plate and an additional displacement.
set_origin_pose (Pose, 0, 0, 0.03, PoseNewOrigin)
* We expect rotation and scale changes
create_planar_calib_deformable_model (ImageReduced, CamParam, PoseNewOrigin, 'auto', -0.1, 0.2, 'auto',
0.6, 1, 'auto', 0.6, 1, 'auto', 'none', 'use_polarity', 'auto', 'auto', [], [], ModelID)
* Get the model contours in world coordinates to be able to easily
* visualize the matching poses later
set_deformable_model_param (ModelID, 'get_deformable_model_contours_coord_system', 'world')
get_deformable_model_contours (ModelContours, ModelID, 1)
count_obj (ModelContours, NumberContour)
```

The 3D origin of the model is displayed here

```
get_deformable_model_params (ModelID, 'model_pose', ModelPose)
dev_set_colored (3)
```

```
disp_message (WindowHandle, 'Calibrated Pose of Object\n X: ' + ModelPose[0]$'.4f' + ' m\n Y: ' + Model-
Pose[1]$'.4f' + ' m\n Z: ' + ModelPose[2]$'.4f' + ' m', 'window', 12, 12, 'black', 'true')
disp_3d_coord_system (WindowHandle, CamParam, ModelPose, 0.15)
disp_continue_message (WindowHandle, 'black', 'true')
stop ()
```

Find pose of car door in subsequent images

```
gen_rectangle1 (Rectangle2, 200, 50, 420, 620)
for Index := 1 to 20 by 1
  read_image (ImageSearch, 'automotive/car_door_' + Index$'.02')
  * For speed reasons the search domain can be reduced
  reduce_domain (ImageSearch, Rectangle2, ImageReducedSearch)
  dev_display (ImageSearch)
  count_seconds (Seconds1)
  find_planar_calib_deformable_model (ImageReducedSearch, ModelID, -0.1, 0.2, 0.6, 1.0, 0.6, 1.0, 0.7, 1, 1,
0, 0.7, [], [], Pose, CovPose, Score)
  count_seconds (Seconds2)
  Time := Seconds2 - Seconds1
  *
*Visualisation of detected models
  for Index1 := 0 to |Score| - 1 by 1
    * Select respective match
    tuple_select_range (Pose, Index1 * 7, ((Index1 + 1) * 7) - 1, PoseSelected)
    pose_to_hom_mat3d (PoseSelected, HomMat3D)
    * Construct a Projection of the detected model with its 3D pose
    gen_empty_obj (FoundContour)
    for Index2 := 1 to NumberContour by 1
      select_obj (ModelContours, ObjectSelected, Index2)
      get_contour_xld (ObjectSelected, Y, X)
      Z := gen_tuple_const(|X|,0.0)
      * Transform the metric model into the world coordinate system
      affine_trans_point_3d (HomMat3D, X, Y, Z, Xc, Yc, Zc)
      * Project model from 3D world coordinate system into the camera
      project_3d_point (Xc, Yc, Zc, CamParam, R, C)
      gen_contour_polygon_xld (ModelWorld, R, C)
      concat_obj (FoundContour, ModelWorld, FoundContour)
    endfor
    * Display results
    disp_message (WindowHandle, 'Object found in ' + (Time * 1000)$'.4' + ' ms' + '\n X: ' + PoseSelect-
ed[0]$'.4f' + ' m\n Y: ' + PoseSelected[1]$'.4f' + ' m\n Z: ' + PoseSelected[2]$'.4f' + ' m', 'window', 12, 12,
'black', 'true')
    dev_set_color ('cyan')
    dev_display (FoundContour)
    dev_set_colored (3)
    disp_3d_coord_system (WindowHandle, CamParam, PoseSelected, 0.15)
  endfor
  if (Index != 20)
    disp_continue_message (WindowHandle, 'black', 'true')
  endif
  stop ()
endfor
clear_deformable_model (ModelID)
```

RobotStudio for programming ABB robots

Die Universität Luxemburg organisierte einen Workshop mit dem Titel „Einführung in RobotStudio zur Programmierung von ABB-Robotern“. Die RobotStudio-Software ist eine Offline-Programmiermethode für ABB-Roboter.

Während des Workshops haben wir ein Projekt simuliert, das mit einem Roboter synchronisiert werden kann. Zuerst richteten wir die virtuelle Arbeitsumgebung ein. Dann begannen wir mit der Simulation des Projekts mit virtuellen Elementen wie Roboter, einer Klemme, Tischen, etc.

Kontakt:
Universität Luxemburg
Abir Gallala
e-mail: abir.gallala@uni.lu

L'université de Luxembourg a organisé un atelier intitulé «Initiation à RobotStudio pour la programmation des robots ABB ». Le software Robotstudio est une méthode de programmation hors ligne pour les robots ABB.

Au cours de l'atelier, nous avons simulé un projet pouvant être synchronisé avec un robot. Tout d'abord, nous avons commencé par introduire l'environnement virtuel de travail. Nous avons ensuite démarré la simulation du projet en utilisant des éléments virtuels tels que le robot, une pince, des tables etc.

Contact:
Université du Luxembourg
Abir Gallala
e-mail: abir.gallala@uni.lu

Human motion measurement

Die Summer School endete mit einem originellen Workshop. Die Robotikforschung beschäftigt sich zunehmend mit der Mensch-Roboter-Interaktion. In diesem Bereich versuchen wir sicherzustellen, dass der Roboter auf die verschiedenen Reize des Menschen – Stimme, Gesichtsausdruck, Kontakt, etc. – reagieren kann. So kann die menschliche Bewegung ein Mittel zur Kommunikation mit Robotern sein.

Die Universität Lüttich verfügt über ein Labor für Bewegungsanalyse. Hier befassen sich mehrere Forschungsprojekte der Robotik mit der menschlichen Bewegung. Dies war die Gelegenheit, verschiedene Techniken zur Messung der menschlichen Bewegung vorzustellen. Cédric Schwarz präsentierte vor allem das Codamotion-System: Ein System, das auf Infrarotkameras und aktiven Sensoren am Menschen basiert.

In einem zweiten Schritt stellten Nathan Deom und Robin Pellois eher experimentelle Bewegungsmesssysteme vor, die jeweils auf Markererkennung und Datenverarbei-

L'Université d'été s'est terminée par un atelier un peu plus original. La recherche en robotique s'intéresse de plus en plus à l'interaction entre l'humain et le robot. Dans ce domaine on cherche à ce que le robot puisse réagir aux différents stimuli (voix, expressions faciales, contact ...) de l'humain. Ainsi, le mouvement humain peut être un moyen de communiquer avec les robots.

L'Université de Liège possède un laboratoire d'analyse de mouvement humain. Ici, plusieurs projets de recherche en robotique sont en lien avec le mouvement humain. C'était alors l'occasion de présenter différentes techniques de mesure de mouvement humain. Principalement, Cédric Schwarz a présenté le système Codamotion : Un système basé sur des caméras infrarouges et des capteurs actifs placés sur l'humain.

Dans un second temps Nathan Deom et Robin Pellois ont présenté des systèmes de mesure de mouvement plus expérimentaux basés respectivement sur la détection de

tung des Trägheitskraftwerks basieren. Dieser interaktive Workshop ermöglichte es, die jeweiligen Vor- und Nachteile der verschiedenen Techniken zu verstehen.

Kontakt:
Universität Lüttich
Arthur Lismonde
E-Mail: alismond@ulg.ac.be

Robin Pellois
E-Mail: robin.pellois@ulg.ac.be

marqueur et le traitement de donnée de centrales inertielles. Cet atelier interactif a permis d'appréhender les avantages et inconvénients respectifs des différentes techniques.

Contact:
Université de Liège
Arthur Lismonde
E-Mail: alismond@ulg.ac.be

Robin Pellois
E-Mail: robin.pellois@ulg.ac.be

Kontakt
Contact

Projektleitung
Direction du projet

Rainer Müller, Prof. Dr.-Ing.
Zentrum für Mechatronik und Automatisierungstechnik gGmbH
Telefon: +49 (0)681 857 87 15
E-Mail: rainer.mueller@zema.de
Webseite: www.zema.de

Matthias Vette-Steinkamp, Dr.-Ing. Dipl.-Wirt.-Ing. (FH)
Zentrum für Mechatronik und Automatisierungstechnik gGmbH
Telefon: +49 (0)681 857 87 531
E-Mail: matthias.vette@zema.de
Webseite: www.zema.de

Projektpartner
Operateurs du projet

Gabriel Abba, Prof. Dr.
Université de Lorraine
Telefon: +33(0)387 375 430
E-Mail: gabriel.abba@univ-lorraine.fr
Webseite: www.univ-lorraine.fr

Olivier Bruls, Prof.
Université de Liège
Telefon: +32 (0)4366-9184
E-Mail: o.bruls@ulg.ac.be
Webseite: www.ulg.ac.be

Jean Denoël
Pôle MecaTech
Telefon: +32 (0)488 833 464
E-Mail: jean.denoel@polemecatech.be
Webseite: www.polemecatech.be

Wolfgang Gerke, Prof. Dr.-Ing.
Hochschule Trier, Umwelt-Campus Birkenfeld
Telefon: +49 (0)6782 17-1113
E-Mail: w.gerke@umwelt-campus.de
Webseite: www.umwelt-campus.de

Peter Plapper, Prof. Dr.-Ing.
Université du Luxembourg
Telefon : +352 (0)466644-5804
E-mail: peter.plapper@uni.lu
Webseite: wwwde.uni.lu

Strategische Partner
Opérateurs méthodologiques

Régis Bigot
Manior Industries
Telefon: +33 (0)3 87 39 78
Webseite: www.manoir-industries.com

Frédérik Cambier
Technifutur
Webseite: www.technifutur.be

Julie Corouge
Universität der Großregion
Telefon: +49 (0)681 301 40801
E-Mail: julie.corouge@uni-gr.eu
Webseite: www.uni-gr.eu

Anja Höthker
Luxinnovation – National Agency for innovation and research
Telefon: +352 (0)43 62 63 – 858 E-Mail:
laurent.federspiel@luxinnovation.lu
Webseite: en.luxinnovation.lu

Amarilys Ben Attar
Institut de Soudure
Telefon: +33 (0)3 87 55 60 76
E-Mail: a.benattar@isgroupe.com
Webseite: www.isgroupe.com

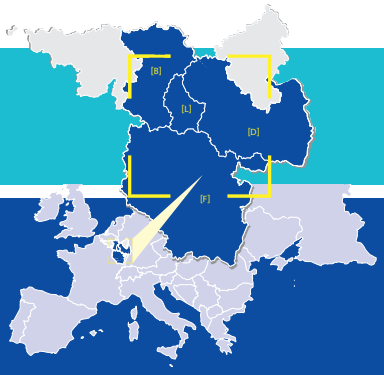
Christian Laurent
Automation & Robotics
Telefon: +32 (0)87 322 330
E-Mail: c.laurent@ar.be
Webseite: www.ar.be

Nigel Ramsden
FANUC Europe Corporation
Telefon: +352 (0)72 77 77 450
E-Mail: nigel.ramsden@fanuc.eu
Webseite: www.fanuc.eu

Sakina Seghir
MATERALIA – Pôle de Compétitivité MatériauxMaterial, Verfahren, Energie
Telefon: +33 (0)3 55 00 40 35
E-Mail: sakina.seghir@materalia.fr
Webseite: www.materalia.fr

Abdel Tazibt
CRITT TJFU
Telefon: +33 (0)3 29 79 96 72
E-Mail: a.tazibt@critt-tjfu.com
Webseite: www.critt-tjfu.com

Grégory Reichling
Citius Engineering
Telefon: +32 (0)4 240 14 25
E-Mail: gregory.reichling@citius-engineering.com
Webseite: www.citius-engineering.com



www.robotix.academy