Lisp Tutorial

Jan Winkler

Institute for Artificial Intelligence
Universität Bremen

May 19, 2014
Summer Term

Today's Schedule

- Coordinate Transformations
  - Important Coordinate Systems in Robotics
  - The Homogeneous Transformation
  - Coordinate Transformations in Lisp: `cl-transforms`, `cl-tf`
- Localization
  - Base knowledge: A Map
  - Sensor Information for Localization
  - Pitfall: Kidnapped Robot
- Odometry
  - Blind Localization
  - Wheel Slippage
- Guidelines for configuring the NXT ROS node

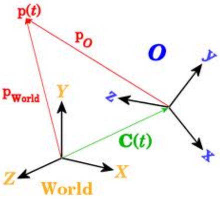
Coordinate Transformations

What is a Coordinate Transformation

The position and orientation of a given body, with respect to a reference coordinate system.

Coordinate systems can be *nested*:

- Head relative to the feet
- Feet relative to earth coordinates (e.g. GPS)
- Earth's coordinates relative to the sun
- Sun's coordinates relative to the milkyway's center
- ...



Coordinate Transformations

Important Coordinate Transformations in Robotics

Some coordinate transformations are especially interesting in robotics:

- Position and orientation (**Pose**) of a robot ...
  - ... in the world
  - ... relative to other robots
  - ... relative to an object of interest
- Pose of the robot's hand relative to its camera
- Difference of the robot's **believed** pose vs. it's **real** pose

Coordinate Transformations

The Homogeneous Transformation

$$\left[\begin{array}{ccc|c} & R & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \quad , \quad R \in \mathbb{R}^{3\times 3}, T \in \mathbb{R}^3 \tag{1}$$

R: Rotation Matrix
This is the rotational part of the transformation

T: Translation Vector
This is the translational part of the transformation

$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$: Shearing Vector
This describes how much the transformation *shears*

$\begin{bmatrix} 1 \end{bmatrix}$: Scale Factor
This signifies how much the transform *scales*

Coordinate Transformations
Shearing

Shearing or *skewing*:

Shearing is a **non-affine transformation**, i.e. **breaks** the transformed object. The same accounts for **scaling**.

Translating and/or rotating an object does not break it, therefore being **affine** transformations.

Coordinate Transformations

Applying the Homogeneous Transformation

General formulation of applying the homogeneous transformation:

$$\vec{v}_2 = \left[\begin{array}{ccc|c} & R & & T \\ & & & \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \cdot \vec{v}_1 \tag{2}$$

In vectors:

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ * \end{bmatrix} = \left[\begin{array}{ccc|c} & R & & T \\ & & & \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \tag{3}$$

Coordinate Transformations

The Identity Transformation

The identity transformation just returns the same vector:

$$\vec{v}_1 = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \cdot \vec{v}_1 \tag{4}$$

Coordinate transformations therefore only contain two interesting values for us:

- The **Rotation** (3D Rotation Matrix, Angles, Quaternion)
- The **Translation** (3D Vector)

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms`

Load the package `cl_transforms`, and the included system `cl-transforms`. Switch to the package `cl-transforms`.

REPL Prompt in the cl-transforms package

```
CL-TRANSFORMS>
```

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms`

Creating the identity pose in the REPL:

`(make-identity-pose)`

Identity pose

```
CL-TRANSFORMS> (make-identity-pose)
#<POSE
   #<3D-VECTOR (0.0d0 0.0d0 0.0d0)>
   #<QUATERNION (0.0d0 0.0d0 0.0d0 1.0d0)>>
```

A **quaternion** is a mathematical formulation of a rotation. It is equivalent to rotation matrices, axis/angle pairs, and euler angles.

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms`

Creating a custom pose:

```
(make-pose origin orientation)
```

Custom pose

```
CL-TRANSFORMS> (make-pose
                (make-3d-vector 1.0 0.0 0.5)
                (make-identity-rotation))
#<POSE
   #<3D-VECTOR (1.0d0 0.0d0 0.5d0)>
   #<QUATERNION (0.0d0 0.0d0 0.0d0 1.0d0)>>
```

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms`

Using Euler angles:

```
(euler->quaternion &key (ax 0.0) (ay 0.0) (az 0.0))
```

With

- `ax`: Rotation around `x` axis
- `ay`: Rotation around `y` axis
- `az`: Rotation around `z` axis

Custom pose

```
CL-TRANSFORMS> (make-pose
                 (make-3d-vector 1.0 0.0 0.5)
                 (euler->quaternion :ax pi))
#<POSE
   #<3D-VECTOR (1.0d0 0.0d0 0.5d0)>
   #<QUATERNION (1.0d0 0.0d0 0.0d0 6.12323399d-17)>>
```

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms`

Transformations vs. Poses:

- Syntactically equivalent (i.e. *they have the same values*)
- Semantically different:
  - Poses are **current positions and orientations** of a given body
  - Transformations are **relative changes** to a given body's current pose

Identity Transformation

```
CL-TRANSFORMS> (make-identity-transform)
#<TRANSFORM
   #<3D-VECTOR (0.0d0 0.0d0 0.0d0)>
   #<QUATERNION (0.0d0 0.0d0 0.0d0 1.0d0)>>
```

Other that that, technically the same as poses.

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms`

Applying a transformation to a pose:

Pose transformation

```
CL-TRANSFORMS> (let ((pose-1 (make-pose
                                (make-3d-vector 1.0 0.0 0.0)
                                (make-identity-rotation)))
                     (traf-1 (make-transform
                                (make-3d-vector 0.0 0.0 0.5)
                                (make-identity-rotation))))
                 (transform traf-1 pose-1))

#<POSE
   #<3D-VECTOR (1.0d0 0.0d0 0.5d0)>
   #<QUATERNION (0.0d0 0.0d0 0.0d0 1.0d0)>>
```

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms`

Accessing values of poses:

- Get the **origin of a pose**: `(origin pose)`
- Get the **orientation of a pose**: `(orientation pose)`

Accessing values of transforms:

- Get the **translation of a transform**: `(translation transform)`
- Get the **rotation of a transform**: `(rotation transform)`

Accessing values of vectors and quaternions:

- Get the **x, y, z** coordinate: `(x vector)`, `(y vector)`, `(z vector)`
- Get the **w** coordinate of quaternions: `(w quaternion)`

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-tf`

New types:

- `pose-stamped`:

```
(make-pose-stamped frame-id stamp translation
                   rotation)
```

- `transform-stamped`

```
(make-stamped-transform frame-id child-frame-id stamp
                        translation rotation)
```

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-tf`

Meaning of the new fields:

- `frame-id`: Reference system for this pose/transform
- `child-frame-id`: Coordinate system's coordinates described by this transform
- `stamp`: Validity timestamp of this pose/transform (UNIX timestamp)

`cl-tf` does automatic transformation from one coordinate system to another, as long as they are part of the same **TF tree**.

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-tf`

Pose transformation in `cl-tf`:

```
(transform-pose tf &key pose target-frame)
```

Underlying thought: Use the TF instance *tf* to generate a pose given in the coordinate system *target-frame* based on the pose *pose*.

Example:

Transforming Poses using cl-tf

```
CL-TF> (transform-pose *tf* pose-of-object-in-camera-frame "/map")
```

For us, pose **coordinate systems are frames**.

Coordinate Transformations

Coordinate Transformations in Lisp: `cl-transforms` vs. `cl-tf`

Both packages work together, `cl-transforms` being a basis for `cl-tf`.

`cl-transforms`:

- Mathematical basis
- Accessor functions for structures
- Offline

`cl-tf`:

- Buffering of TF tree
- Transform over tree systems
- Online, based on ROS
- Uses the global `/tf` topic

For now, we will only use `cl-transforms`, as the structure of robot coordinate frames strongly depends on it's physical structure.

More info on **tf**: `http://wiki.ros.org/tf/`

Localization

Why Localize?

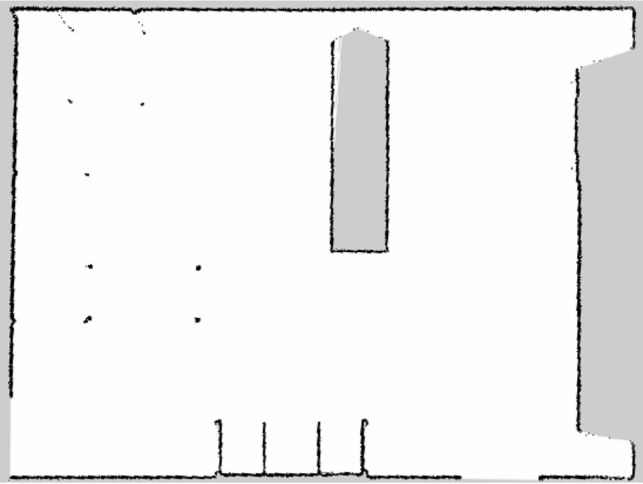
Localization means:

- Properly determining the pose of a robot in it's environment
- Having knowledge about static obstacles relative to the **map**

Why you should care:

- Basis for **path planning**
- Moving to a predefined position relative to map
- Properly store locations of **detected objects** in the environment

Localization
Map



Localization
Sensor Information

Important, common sensor information used for localization:

- Distance measurement
  - Laser scanners (1D, 2D, 3D: Kinect)
  - Ultrasonic sensors (point distance)
- (Color) Images (landmark fitting)

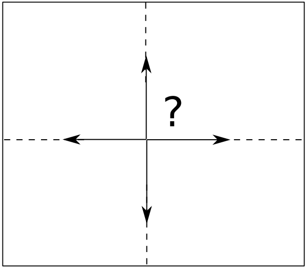
A whole set of research topics builds on top of this:

- ICP
- SLAM(-RGB/D)
- SIFT/SURF Features
- ...

Localization

Kidnapped Robot

Ambiguities in the map can lead to robot confusion:



Place your robot in a position that is ambiguous in the map. Chances are that it will localize itself wrongly (seen from a **global** perspective), because from a **local** perspective the environment matches.

Odometry
Blind Localization

Localization can be done completely based on . . .

- . . . a well-defined initial pose, plus . . .
- . . . tracking of wheel encoder values

Pro:

- No continuous tracking of external features (distances, landmarks)

Con:

- Wheel slippage falsifies results (encoder values are only heuristics)

Guidelines for NXT ROS

Components of your NXT ROS package

Inside the virtual machine, open a terminal and type

```
roscd nxt_project
```

Important files in this package:

- `nxt.launch`: Launch file for starting (custom) NXT interface
- `robot.yaml`: Custom configuration of ROS/NXT interface
  - Sensors, Actuators
  - Parameters

Guidelines for NXT ROS

Configuring your own `robot.yaml`

Connectable Sensors we are going to use:

- `touch sensor`
- `intensity sensor`
- `ultrasonic sensor`

Connectable Actuators:

- `servo`

Guidelines for NXT ROS

robot.yaml: touch sensor

Touch sensors act as bumber buttons.

Touch Sensor Definition in robot.yaml

```
- type: touch
  name: touch_sensor
  port: PORT_1
  desired_frequency: 10.0
```

Guidelines for NXT ROS

`robot.yaml`: intensity sensor

Intensity sensors react to different sources of light.

Intensity Sensor Definition in robot.yaml

```
- type: intensity
  name: intensity_sensor
  port: PORT_1
  color_r: 1.0
  color_g: 0.0
  color_b: 0.0
  desired_frequency: 10.0
```

The RGB channel can be restricted to the color the sensor "*sees*".

Guidelines for NXT ROS

`robot.yaml`: ultrasonic sensor

Ultrasonic sensors measure the distance to the next obstacle.

Ultrasonic Sensor Definition in robot.yaml

```
- type: ultrasonic
  name: ultrasonic_sensor
  port: PORT_1
  spread_angle: 0.2
  min_range: 0.01
  max_range: 2.5
  desired_frequency: 10.0
```

Ultrasonic sensors have a variable spread angle, minimum, and maximum measuring distance.

Guidelines for NXT ROS

`robot.yaml`: **servo**

Servos are actuators that can be controlled from ROS.

Servo Definition in robot.yaml

```
- type: motor
  name: l_motor_joint
  port: PORT_A
  desired_frequency: 10.0
```

Servos take an *effort* as input (i.e. force exerted on their axis). This is proportionally correlated to their *angular velocity*.

From here on for today: Practical Lessons

Assembling an NXT robot

Assemble an NXT robot of your flavour that has the following characteristics:

- Is mobile (use servos)
- Has at least one sensor attached

Keep in mind:

- The robot should solve a simple problem of autonomous behaviour
  - Navigation in unknown environments
  - Search for specific sensor signals
  - Transport of objects
  - . . . come up with your own, individual idea!
- The programming has to be done in **Lisp**.

From here on for today: Practical Lessons

Starting the ROS node

Start the `nxt_project` node:

```
$ roslaunch nxt_project nxt.launch
```

This launches the USB interface node between the ROS master and the NXT brick.

Remember: `roslaunch` starts an implicit `roscore` if you haven't done so already. So don't bother starting it yourself.

From here on for today: Practical Lessons

Testing the Connection

To set the effort for a motor at port A to 10.0NM, do the following:

Intensity Sensor Definition in robot.yaml

```
$ rostopic pub /joint_command nxt_msgs/JointCommand
    {name: 'motor_A', effort: 10.0}
```

To switch it back off, set effort to zero:

Intensity Sensor Definition in robot.yaml

```
$ rostopic pub /joint_command nxt_msgs/JointCommand
    "{name: 'motor_A', effort: 0.0}"
```

From here on for today: Practical Lessons
Basic NXT Lisp Setup Repository

A sample NXT Lisp package can be acquired here:

`https://github.com/ai-seminar/nxt_lisp.git`

The base package performs two actions:

- `NXT-LISP> (startup)`
  Start ROSLisp ROS node and connect it to the ROS master
- `NXT-LISP> (set-motor-effort name effort)`
  Set the effort `effort` for the motor `motor`.

You can use this package as a template for your Lisp implementation.

From here on for today: Practical Lessons

Troubleshooting

If `nxt_project` complains about not finding `nxt.locator` (Python Exception) and you are NOT using the Virtual Machine, you have to add the following line to your `~/.bashrc`:

```
export PYTHONPATH=
$PYTHONPATH:~/ros_ws/src/nxt_tutorial/nxt/nxt_python/scripts
```

In case your NXT Python checkout (NOT `nxt_project`, see earlier slides) resides in `nxt_tutorial` and your ROS Catkin Workspace is in `ros_ws`.