Autonome mobiele robots
Assignment 1

Maarten de Jonge
Inge Becht
Duncan ten Velthuis

November 9, 2012

1. For the differential drive robot:
   We want to find the ξ_I at each Δt interval. To do this we can use the formula:

$$\dot{\xi_I} = R(q)^{-1} J_1^{-1} * J_2 * \varphi(t)$$

   where in the case of the differential drive:

$$R = \begin{bmatrix} cos(q) & sin(q) & 0 \\ -sin(q) & cos(q) & 0 \\ 0 & 0 & 1 \end{bmatrix} J_1 = \begin{bmatrix} sin(\alpha_1 + \beta_1) & -cos(\alpha_1 + \beta_1) & (-l)cos(\beta_1) \\ sin(\alpha_2 + \beta_2) & -cos(\alpha_2 + \beta_2) & (-l)cos(\beta_2) \\ cos(\alpha_1 + \beta_1) & sin(\alpha + \beta) & lsin(\beta) \end{bmatrix}$$

$$J2 = \begin{bmatrix} r_1 & 0 & 0 \\ 0 & r_2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \varphi(t) = \begin{bmatrix} \varphi_1(t) \\ \varphi_2(t) \\ 0 \end{bmatrix}$$

   The values inserted in J_1 are those of the constraints, with the first two rows being the rolling constraint for both wheels and the third column the sliding constraint for the robot. The value for $\alpha_1 = \frac{-\pi}{2}$ and $\beta_1 = \pi$. For $\alpha_2 = \frac{\pi}{2}$ and $\beta_2 = 0$. J_2 consists of the radii of the wheels on the diagonal. Filling in these values gives:

$$\dot{\xi_I} = R(q)^{-1} \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2l} & \frac{1}{2l} & 0 \end{bmatrix} \begin{bmatrix} r_1\varphi_1(t) \\ r_2\varphi_2(t) \\ 0 \end{bmatrix}$$

   Because with this formula only the velocity in the x, y and q direction is given the new position (at time $t + \Delta t$) can be calculated with:

$$\xi_I^{t+\Delta t} = \xi_I^t + (\dot{\xi}_I^t * \Delta t)$$

2. a

   Given without the constraints:

$$\begin{bmatrix} x \\ y \\ q \end{bmatrix}^{t+\Delta t} = \begin{bmatrix} x \\ y \\ q \end{bmatrix}^{t} + (R(q)^{-1} \begin{bmatrix} \frac{r_1\dot{\varphi_1}}{2} + \frac{r_2\dot{\varphi_2}}{2} \\ 0 \\ \frac{r_1\dot{\varphi_1}}{2l} + -\frac{r_2\dot{\varphi_2}}{2l} \end{bmatrix}^{t}) * \Delta t$$

   (b) $\dot{w}(t) = \frac{r_1\dot{\varphi_1}(t)}{2l} + -\frac{r_2\dot{\varphi_2}(t)}{2l}$
   $\dot{v}(t) = \frac{r_1\dot{\varphi_1}(t)}{2} + \frac{r_2\dot{\varphi_2}(t)}{2}$

3. This exercise was made in Python using the nxt-python library. To control the robot we created a simple movement API (see `movement.py`) in which we distinguish between two possible motions; moving forward and rotating. For moving forward, the motor power and wheel rotation (in degrees) have to be specified, and the robot will move by rotating both wheels at the given rotation angle.

   There's two ways to do a rotation; leaving one wheel stationary and rotating the other one (thus rotating around the stationary wheel), or spinning both wheels in the opposite direction, rotating around the point between the two wheels. In practise, the first method doesn't perform as well due to increased friction on the stationary wheel. The rotation function in our API takes a rotation for the robot to perform along with motor power. Positive motor power will turn counterclockwise, negative power will turn clockwise.

To construct a path for the robot to drive, a list of forward movements and rotations can be specified, which will be executed sequentially.

For both movement types (rotation and straight ahead) we tested both braking and not breaking after each movement. brakes and not using brakes. Using brakes works much better than not using brakes for the odometrie model as the specified number of rotations will be closer to the retrieved number of rotations from the sensor data.

We created two odometry models. The first model checked after every movement of the robot (so after each transition from moving straight ahead and rotating) if the specified amount of driven milimeters was the same as the actually driven amount of milimeters. This is done by reading the rotation count from the wheels after every movement. See `odometrie_tester.py` for the functions that enable this odometrie model and in function `path` of `movement.py` the interaction between movement modules and the rotation counts is shown. To apply this odometrie run `movement.py`

The file `odometrie_position_tester.py` is the second odometrie model in which the kinematics model is constructed in the same way as is given in question 1. The idea here is to keep track of the robot's position (x, y and q) in a global reference frame with a given sampling rate, taking the initial pose of the robot as the origin. This odometrie model starts when running `movement_position_odometrie.py` and works as a background thread, every 100 milliseconds updating the environmental position.

A simple PID controller has been implemented for correcting the robot's trajectory when driving straight. It works by measuring the rotational speeds of both wheels and taking their difference as the error function (the more equal the wheel speeds are, the straighter you're driving). The nxt-python library only offers one way to influence the power of already rotating wheels, which wouldn't work well together with our earlier motion functions. For this reason the PID controller is implemented in a seperate test-function (in the file `pid.py`). It actuates by keeping the power of the left wheel constant (at 80), and converting the output of the controller to a value that will be added to the power of the right wheel, in an attempt to balance the wheels' powers such that they both rotate equally fast.