Objectives

This lab introduces the Mathworks Robotics System Toolbox. Proficiency in Matlab is assumed, please revisit the assignments on Matlab Basics and Simulink in the course Scientific Programming in Matlab in case you need to refresh your knowledge. The Robotics System toolbox provides an interface between MATLAB and Simulink and the Robot Operating System (ROS) that enables you to test and verify applications on ROS-enabled robots and robot simulators such as Gazebo or Stage.

This lab deals with representations of spatial transformations between coordinate frames in the three dimensional Euclidean space. Handling these general 3D transformations is somewhat simplified as the mobile robot in our course resides in a planar two dimensional world. You will learn to inspect the ROS transformation tree and to utilize transforms to map point cloud data between frames.

The purpose of this assignment is to get familiar with representations of spatial transformations. A recommended reading is the section on position and orientation representation in the chapter on kinematics in the Springer Handbook of Robotics [3] or chapters $2.1-2.7$ in the book by Siciliano et al [5]. Please carefully read these sections prior to the lab.

The lab discusses the ROS transformation tree that provides access to the coordinate transformations that are available on the ROS network. These transformations are applied to geometric entities such as points, point clouds or poses to represent them w.r.t. different coordinate frames. The three most relevant frames for mobile navigation are the static map and odom frame and the robocentric base link frame that moves with the robot. The transform between odom and base link is estimated from odometry information The transform between map and odom is corrected by adaptive Monte Carlo localization to account for the odometric drift.

This lab also introduces scan matching as a technique to recover the relative pose between two scans at nearby robot poses. Scan matching provides an alternative to Monte Carlo localization to compensate for the odometry drift. Please read the paper on scan matching with the normal distribution transform by Biber et at [4].

Representations of Spatial Transformations

The purpose of this assignment is to get familiar with representations of spatial transformations. A recommended reading is the section on position and orientation representation in the chapter on kinematics in the Springer Handbook of Robotics [3].

In case you already attended the lab on spatial transformations in the course Modeling and Control of Robotic Manipulators, you can skip the first part and directly move on to the topic **Robotics System Toolbox Transformation Tree**.

Spatial Transformations

Robot kinematics establishes the transformations among various coordinate frames that capture the positions and orientations of end-effector, links or rigid bodies. The mathematics of spatial transformations that describe rigid motions in Euclidean space is of fundamental importance to robotic manipulation.

ROS and the Matlab Robotic toolbox support multiple representations of rotations and spatial transformations:

- homogeneous transformations (4x4 matrix)
- rotation matrix (3x3 matrix)
- quaternions
- angle and vector orientation
- Euler angles
- roll, pitch, yaw angles

Translations

A translation in Euclidean space is represented by the components of an ordinary 3D vector $p = [p_x\ p_y\ p_z]$ in which the vector $\mathbf{p} = p_x\mathbf{e_x} + p_y\mathbf{e_y} + p_z\mathbf{e_z}$. $\mathbf{e_x}, \mathbf{e_y}, \mathbf{e_z}$ are the unit vectors of the frame axes of an orthogonal frame $O - xyz$. Successive translations are obtained by mere vector addition $\mathbf{p}_{02} = \mathbf{p}_{01} + \mathbf{p}_{12}$.

Matlab employs the abbreviation `trvec` for a translation vector which is represented in 3-D Euclidean space as Cartesian coordinates. Its numeric Representation is a 1-by-3 vector. For example, a translation by 3 units along the x -axis and 2.5 units along the z -axis is expressed as:

```
trvec = [3 0 2.5];
```

Rotation Matrices

A rotation matrix performs a rotation of a vector in Euclidean space, e.g. the matrix:

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

rotates vectors counter-clockwise through an angle θ about the z-axis of the Cartesian coordinate system. To perform the rotation using a rotation matrix $\mathbf{R}$, the position of a point in space is represented by a 1-by-3 column vector $\mathbf{v} = \begin{bmatrix} x & y & z \end{bmatrix}^T$, that contains the coordinates of the point w.r.t. an $X - Y - Z$ frame. A rotated vector is obtained by the matrix multiplication $\mathbf{Rv}$. Coordinate rotations are a natural way to express the orientation of a robot end effector, robot link or camera relative to a reference frame e.g. the robot base frame or a world frame. Once XYZ axes of a local frame are expressed numerically as three direction vectors w.r.t. a global world frame, they together comprise the columns of the rotation matrix $\mathbf{R}$ that transforms vectors in the reference frame into equivalent vectors expressed in the coordinates of the local frame. The inverse of rotation matrix coincides with its transpose, namely $\mathbf{R}^{-1} = \mathbf{R}^T$. Rotation matrices are square matrices that obey the orthogonality constraints: $\mathbf{R}^T = \mathbf{R}^{-1}$ and $\det(\mathbf{R}) = 1$.

A basic or elemental rotation is a rotation about one of the axes of a Cartesian coordinate frame (see figure 1). The three basic rotation matrices rotate vectors by an angle θ about the x, y, or z axis in three dimensions using the right hand rule.

$$\begin{aligned} \mathbf{R}_x(\theta) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \\ \mathbf{R}_y(\theta) &= \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \\ \mathbf{R}_z(\theta) &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Matlab employs the abbreviation `rotm` for a rotation matrix. It is a square, orthonormal matrix with a determinant of 1. Its numeric representation is a 3-by-3 matrix. For example, elemental rotations of $\pi/2$ radians along x, y, z are generated with the commands:

```
theta = pi/2;
% A rotation about the x-axis
rotmx = [1 0 0; 0 cos(theta) -sin(theta); 0 sin(theta) cos(theta)];
```

[1] source: By Jan Boddez - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3450270

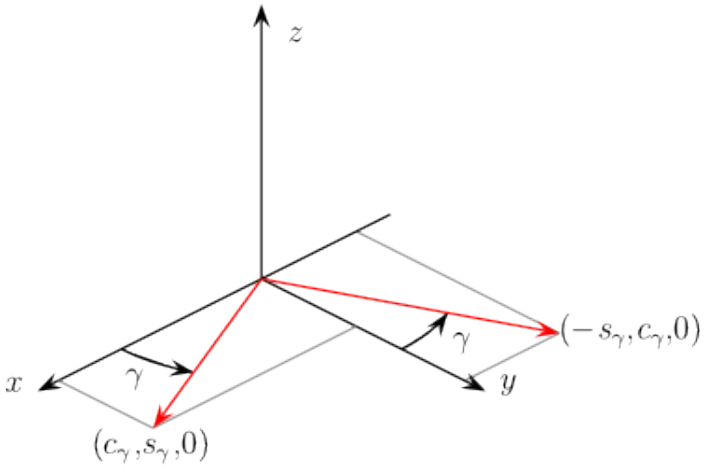

Figure 1: Rotation by an angle γ along the z-axis [1]

```
% A rotation about the y-axis
rotmy = [cos(theta) 0 sin(theta); 0 1 0; -sin(theta) 0 cos(theta)];
% A rotation about the z-axis
rotmz = [cos(theta) -sin(theta) 0; sin(theta) cos(theta) 0; 0 0 1];
```

General rotations are obtained from these three using matrix multiplication. For example, the product:

$$\mathbf{R} = \mathbf{R}_z(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_x(\gamma) \tag{2}$$

represents an intrinsic rotation whose yaw, pitch, and roll angles are α, β and γ.

In general, mobile robots are considered to operate in a planar 2D world, and their spatial extension along the z-axis is often ignored. That implies that the basic mobile robot has two translational degrees of freedom (x-y-plane) and a single rotational degree of freedom (rotation along the z-axis). In particular, this assumption simplifies the description of consecutive rotations, as all rotations share the common z-axis. Therefore it suffices to simply add the corresponding angles of rotation along z, rather than to consider the general case of matrix multiplication of rotation matrices.

Homogeneous Transformations

Homogeneous transformations are important as they provide the basis for defining robot direct (forward) kinematics. The relationships between frames, e.g. robot base frame and end effector frame, are defined by homogeneous transforms.

In the field of robotics there are many possible ways of representing positions and orientations, but the homogeneous transformation is well matched to MATLABs powerful tools for matrix manipulation. Homogeneous transformations combine the two operations of rotation and translation into a single matrix multiplication.

Homogeneous transformations are 4×4 matrices that describe the relationships between Cartesian coordinate frames in terms of translation and orientation. They are composed of a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$.

$$\mathbf{H} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Assume the frame $x_1y_1z_1$ is obtained from frame $x_0y_0z_0$ by applying a rotation $\mathbf{R}_1^0$ followed by a translation with a vector $\mathbf{t}_1^0$. The coordinates of the point $\mathbf{p}^0$ with respect to frame $x_0y_0z_0$ are computed according to:

$$\mathbf{p}^0 = \mathbf{R}_1^0\mathbf{p}^1 + \mathbf{t}_1^0 \tag{4}$$

Assume another rotation and translation to obtain frame $x_2y_2z_2$ from frame $x_1y_1z_1$ by a rotation $\mathbf{R}_2^1$ followed by a translation $\mathbf{t}_2^1$.

$$\mathbf{p}^1 = \mathbf{R}_2^1\mathbf{p}^2 + \mathbf{t}_2^1 \tag{5}$$

The composition of these consecutive rigid motions is given by:

$$\mathbf{p}^0 = \mathbf{R}_1^0\mathbf{R}_2^1\mathbf{p}^2 + \mathbf{R}_1^0\mathbf{t}_2^1 + \mathbf{t}_1^0 \tag{6}$$

thus

$$\begin{aligned} \mathbf{R}_2^0 &= \mathbf{R}_1^0\mathbf{R}_2^1 \\ \mathbf{t}_2^0 &= \mathbf{R}_1^0\mathbf{t}_2^1 + \mathbf{t}_1^0 \end{aligned}$$

This affine representation of rotation and translation has the drawback that rotational and translational components get mixed, e.g. with terms such as $\mathbf{R}_1^0\mathbf{t}_2^1$. A homogeneous representation replaces the matrix multiplication and vector addition in three dimensions by matrix multiplication in four dimensions. For that purpose the Euclidean vectors $\mathbf{p}^0, \mathbf{p}^1$ are augmented by a fourth component (constant of 1) to obtain the homogeneous vectors:

$$\begin{aligned} \mathbf{P}^0 &= \begin{bmatrix} \mathbf{p}^0 \\ 1 \end{bmatrix} \\ \mathbf{P}^1 &= \begin{bmatrix} \mathbf{p}^1 \\ 1 \end{bmatrix} \end{aligned}$$

The transformation in equation (4) is equivalent to the homogeneous matrix transformation

$$\mathbf{P}^0 = \mathbf{H}_1^0 \mathbf{P}^1 \tag{7}$$

with the 4×4 homogeneous transformation

$$\mathbf{H}_1^0 = \begin{bmatrix} \mathbf{R}_1^0 & \mathbf{t}_1^0 \\ \mathbf{0} & 1 \end{bmatrix} \tag{8}$$

The same applies to

$$\mathbf{P}^1 = \mathbf{H}_2^1 \mathbf{P}^2 \tag{9}$$

The overall transformation in equation (6) is obtained from

$$\mathbf{P}^0 = \mathbf{H}_1^0 \mathbf{H}_2^1 \mathbf{P}^2 \tag{10}$$

such that

$$\mathbf{H}_2^0 = \mathbf{H}_1^0 \mathbf{H}_2^1 \tag{11}$$

is obtained from 4×4 matrix multiplication of the individual transformations.

Matlab employs the abbreviation `tform` for a homogeneous transformation matrix that combines a translation and rotation into one matrix. Its numeric representation is a 4-by-4 matrix. For example, a rotation of angle `theta` around the x-axis and a translation of `dy` units along the y-axis is expressed as:

```
theta = pi/2;
tform = [1 0 0 0; 0 cos(theta) -sin(theta) dy; 0 sin(theta) cos(theta) 0; 0 0 0 1];
```

You should pre-multiply your transformation matrix with your homogeneous coordinates, which are represented as a matrix of row vectors (n-by-4 matrix of points). Utilize the transpose (') to rotate your points for matrix multiplication to a 4-by-n) matrix of column vectors. For example:

```
points = rand(100,4);
tformPoints = (tform*points')';
```

The Robotics System Toolbox provides conversion functions for transformation representations:

```
tform=trvec2tform(trvec);
```

generates the homogeneous transformation matrix `tform` that corresponds to a translation vector `trvec` with components $\mathbf{t} = [t_x, t_y, t_z]$.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
tform=rotm2tform(rotm);
```

generates the homogeneous transformation matrix `tform` ($\mathbf{H}$) that corresponds to a rotation matrix `rotm` ($\mathbf{R}$) with components r_{ij}.

$$\mathbf{H} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Euler Angles

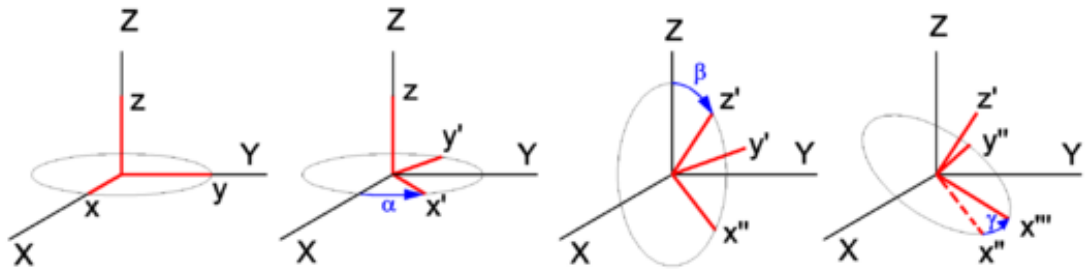

Figure 2: ZYZ - Euler angles [2]

Since a rotation matrix only possess three independent components (orthogonality constraint), one seeks a minimal representation of rotations in term of three independent quantities, in this case, angles of three consecutive elemental rotations denoted by Euler angles. The overall rotation is composed of three successive rotations around the designated axis: First, rotate about the z-axis by the angle ϕ, then rotate about the current y-axis by the angle θ and finally rotate about the current z-axis by the angle ψ. In the figure 2 the angles are denoted by α, β, γ instead of ϕ, θ, ψ. This particular order of rotation axes gives rise to the convention ZYZ-Euler angles. There are other types of combinations, e.g. ZXZ-Euler angles, but ZYZ-Euler angles are the most common ones. The overall rotation is obtained by postmultiplication of rotation matrices

$$\mathbf{R}_{\phi,\theta,\psi} = \mathbf{R}_{z,\phi}\mathbf{R}_{y,\theta}\mathbf{R}_{z,\psi} \tag{12}$$

[2] Citenzendium Euler angles `http://en.citizendium.org/images/thumb/7/7c/Euler_angles.png/550px-Euler_angles.png`

Matlab supports the 'ZYZ' as well as the 'ZYX' axis order of Euler angles which is denoted as Roll Pitch Yaw (rpy). Knowing which axis order you use is important for applying the rotation to points and in converting to other representations. The numeric representation is an ordinary 1-by-3 vector of scalar angles For example, a rotation around the y -axis of π is expressed as:

```
eul = [0 pi 0]
```

The conversions `tform2eul, eul2tform, rotm2eul, eul2rotm` provide changes of representation between Euler angles and rotation matrices and homogeneous transformations. Unfortunately, the default order for these conversions is 'ZYX' corresponding to roll pitch yaw. Therefore you need to specify the sequence 'ZYZ' as an extra argument.

```
% ZYZ Euler angles
rotmzyz = eul2rotm([phi theta pis],'ZYZ');
```

Roll Pitch Yaw Angles

Roll, pitch and yaw angles are an alternative to ZYZ Euler angles, they denote rotations along the axes of a fixed frame rather than the current frame. A rotation matrix $\mathbf{R}$ is described as a product of successive rotations about the principal coordinate axes x, y, z taken in a specific order. These rotations define the roll, pitch and yaw angles usually denoted by ϕ, θ and ψ. The overall rotation with respect to a fixed frame (extrinsic rotation) occurs by an angle ψ along the x-axis (roll), an angle θ along the y-axis (pitch) and an angle ϕ along the z-axis (yaw) resulting from the premultiplication:

$$\mathbf{R}_{\phi,\theta,\psi} = \mathbf{R}_{z,\phi}\mathbf{R}_{y,\theta}\mathbf{R}_{x,\psi} \tag{13}$$

Notice, that instead of roll-pitch-yaw (XYZ) relative to the fixed frames the transformation $\mathbf{R}_{\phi,\theta,\psi}$ can be interpreted as yaw-pitch-roll (ZYX) w.r.t. to the current frame.

Roll Pitch Yaw angles are also denoted as Tait-Bryan angles. They are common in automotive and aerospace applications, so that zero degrees elevation represents the horizontal attitude. They represent the orientation of a vehicle or aircraft with respect to the world frame as shown in figure 3.

Matlab supports the 'ZYX' axis order of Euler angles which is denoted as Roll Pitch Yaw (rpy).

```
% roll pitch yaw
rotmrpy = eul2rotm([phi theta psi],'ZYX');
```

Therefore 'ZYX' Euler angles w.r.t. current frame are equivalent to 'XYZ' roll pitch yaw angles w.r.t. fixed frame. For robotics manipulators 'ZYZ' Euler angles are common whereas in mobile robotics 'XYZ' roll pitch yaw angles are common.

[3]source: By Juansempere - copied from previous workPreviously published: 2011 in wikipedia, CC BY-SA 3.0, `https://en.wikipedia.org/w/index.php?curid=39309443`

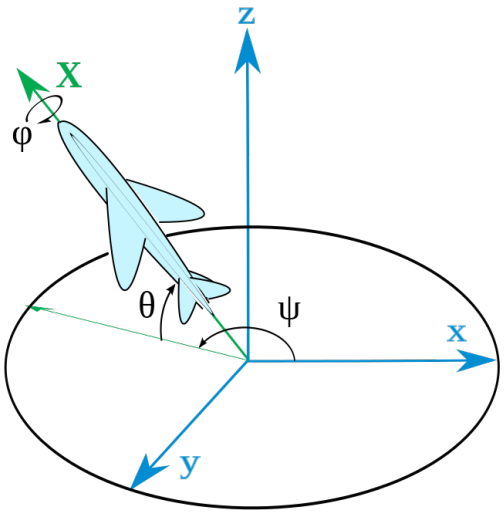

Figure 3: Roll Pitch Yaw angles [3]

Axis - Angle

Euler's rotation theorem postulates that any rotation in 3D can be expressed as a single rotation about a designated axis which itself remains unchanged by the rotation. The magnitude of the angle is also unique, with its sign being determined by the sign of the rotation axis.

Rotations are not necessarily described by consecutive rotations w.r.t principal coordinate axes but instead in terms of a single rotation along an arbitrary axis in space. Let $\mathbf{w} = [w_x, w_y, w_z]^T$ be a unit vector that defines an axis of rotation expressed w.r.t. to the reference frame The pair $\mathbf{w}, \theta$ with unit vector $\mathbf{w} = [w_x, w_y, w_z]^T$ and scalar θ defines a rotation by θ along $\mathbf{w}$. Since the axis is normalized, it has only two degrees of freedom. The angle adds the third degree of freedom to this rotation representation.

If the rotation angle θ is zero, the axis is not uniquely defined. The composition of rotations represented by Euler angles or axis-angle, is not straightforward as the axis vectors do not satisfy the law of vector addition. Typically, one converts from Euler angle / angle-axis to rotation matrix, performs the composition as a matrix product and then transforms back to Euler angle / angle-axis representation.

Matlab employs the abbreviation `axang` for the angle axis representation of a scalar rotation around a fixed axis defined by a vector. The numeric representation is a 1-by-3 unit vector and a scalar angle combined into a1-by-4 vector. For example, a rotation of $\pi/2$ radians around the y-axis is defined by the vector `axang = [0 1 0 pi/2]`.

Unit Quaternions

[4]source:By DF Malan - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1354297

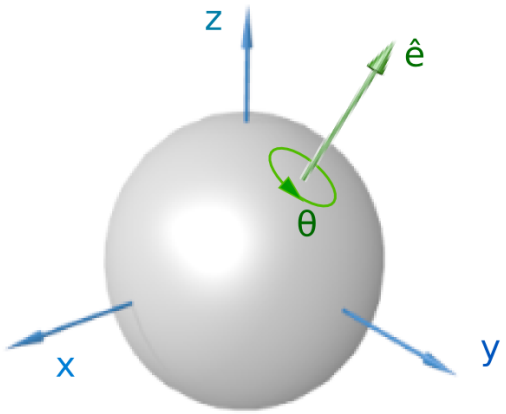

Figure 4: Rotation represented by an Euler axis $\hat{e}$ and angle Θ [4]

Unit quaternions are another representation of orientation. They constitute a compromise between the advantages and disadvantages of rotation matrices and Euler angle sets. Compared to Euler angles they are simpler to compose and avoid the problem of gimbal lock. Compared to rotation matrices they are more numerically stable and may be more efficient. A quaternion $q \in \mathbb{H}$ with:

$$q = \eta + \epsilon_x \mathbf{i} + \epsilon_y \mathbf{j} + \epsilon_z \mathbf{k} \tag{14}$$

is composed of a **scalar part** $\mathrm{Re}(q) := \eta$ and an **imaginary part** (sometimes referred to as **vector part**) with three components $\mathrm{Im}(q) := \epsilon_x \mathbf{i} + \epsilon_y \mathbf{j} + \epsilon_z \mathbf{k}$. The imaginary part of a quaternion behaves like a vector in three dimension vector space, and the real part η a behaves like a scalar in $\mathbb{R}$. The abstract symbols $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are similar to the imaginary component of complex numbers and satisfy:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1 \tag{15}$$

For some operations it is useful to express the quaternion as a 4D vector, composed of the scalar and the vector part (note, we write $\mathbf{q}$ in boldface for this representation):

$$\mathbf{q} = [\eta \ \boldsymbol{\epsilon}]^T = [\eta \ \epsilon_x \ \epsilon_y \ \epsilon_z]^T \tag{16}$$

Unit quaternions are constrained by:

$$\|\mathbf{q}\| = \sqrt{\eta^2 + \boldsymbol{\epsilon}^T \boldsymbol{\epsilon}} = 1 \tag{17}$$

$$\iff \eta^2 + \boldsymbol{\epsilon}^T \boldsymbol{\epsilon} = 1 \tag{18}$$

The inverse of a quaternion is defined by:

$$\mathbf{q}^{-1} = [\eta \ -\boldsymbol{\epsilon}]^T = [\eta \ -\epsilon_x \ -\epsilon_y \ -\epsilon_z]^T \tag{19}$$

The composition of rotations $q^{(1)}$ and $q^{(2)}$ (not squared!) is achieved by the quaternion product (Hamilton product):

$$\begin{aligned}(\eta^{(1)} + \epsilon_x^{(1)}\mathbf{i} + \epsilon_y^{(1)}\mathbf{j} + \epsilon_z^{(1)}\mathbf{k})(\eta^{(2)} + \epsilon_x^{(2)}\mathbf{i} + \epsilon_y^{(2)}\mathbf{j} + \epsilon_z^{(2)}\mathbf{k}) \ &= \\ (\eta^{(1)}\eta^{(2)} - \epsilon_x^{(1)}\epsilon_x^{(2)} - \epsilon_y^{(1)}\epsilon_y^{(2)} - \epsilon_z^{(1)}\epsilon_z^{(2)}) \ &+ \\ (\eta^{(1)}\epsilon_x^{(2)} + \epsilon_x^{(1)}\eta^{(2)} + \epsilon_y^{(1)}\epsilon_z^{(2)} - \epsilon_z^{(1)}\epsilon_y^{(2)})\mathbf{i} \ &+ \\ (\eta^{(1)}\epsilon_y^{(2)} - \epsilon_x^{(1)}\epsilon_z^{(2)} + \epsilon_y^{(1)}\eta^{(2)} + \epsilon_z^{(1)}\epsilon_x^{(2)})\mathbf{j} \ &+ \\ (\eta^{(1)}\epsilon_z^{(2)} + \epsilon_x^{(1)}\epsilon_y^{(2)} - \epsilon_y^{(1)}\epsilon_x^{(2)} + \epsilon_z^{(1)}\eta^{(2)})\mathbf{k}\end{aligned} \tag{20}$$

A unit quaternion resembles the angle axis representation as it can be interpreted as a rotation about the axis denoted by $\frac{\boldsymbol{\epsilon}}{\|\boldsymbol{\epsilon}\|}$ by an angle θ.

$$q = e^{\frac{\theta}{2}(\epsilon_x\mathbf{i}+\epsilon_y\mathbf{j}+\epsilon_z\mathbf{k})} = \cos\frac{\theta}{2} + (\epsilon_x\mathbf{i} + \epsilon_y\mathbf{j} + \epsilon_z\mathbf{k})\sin\frac{\theta}{2} \tag{21}$$

The rotation of an ordinary 3D vector $[p_x, p_y, p_z]^T \in \mathbb{R}^3$ with a unit quaternion $\mathbf{q}$ is achieved by embedding the vector into $\mathbb{H}$ with $\mathbf{p} = [0, p_x, p_y, p_z]^T$ and applying the conjugation of $\mathbf{p}$ by $\mathbf{q}$:

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \tag{22}$$

using the quaternion product defined in equation (20). The position vector $\mathbf{p}'$ denotes the coordinates of the point $\mathbf{p}$ after the rotation. The quaternion $\mathbf{p} = [\eta \ \boldsymbol{\epsilon}]^T$ is composed of the vector part $\boldsymbol{\epsilon}$ that is equal to the 3D vector $[p_x, p_y, p_z]^T$ and a real part $\eta = 0$ that equals zero. The vector part $\boldsymbol{\epsilon}'$ of the resulting quaternion $\mathbf{p}'$ is the desired 3D vector obtained by the quaternion product of $\mathbf{q}$, $\mathbf{p}$ and $\mathbf{q}^{-1}$.

The Robotics System toolbox unfortunately does not (yet) support the quaternion (Hamilton) product. (Remark: The aerospace toolbox provides a function `quatmultiply`) It represents quaternions as an ordinary four-element vector with a scalar rotation and 3-element vector. The first element, $\eta = \cos\theta/2$, is a scalar to normalize the vector with the three other values, $[\epsilon_x \ \epsilon_y \ \epsilon_z] = [w_x \ w_y \ w_z]\sin\theta/2$ define the axis of rotation. The Robotics System toolbox employs the abbrevation `quat` for quaternions with the usual conversions such as `tform2quat, quat2tform, rotm2quat, quat2rotm`

Robotics System Toolbox Transformation Tree

The ROS transformation tree object provides access to the tf coordinate transformations that are shared on the ROS network. You can receive transformations and apply them

to objects such as points, point clouds or poses. You can also send transformations and share them with the rest of the ROS network. ROS uses the tf transform library to keep track of the relationship between multiple coordinate frames. The relative transformations between these coordinate frames are maintained in a tree structure. Querying this tree lets you transform entities like poses and points between any two coordinate frames.

```
tfTree = rostf;
tfTree.AvailableFrames
```

creates a ROS transformation tree object and displays the available frames.

```
canTransform(tftree,targetframe, sourceframe)
```

checks whether the transform between the `targetframe` and `sourceframe` is available. If that is the case:

```
tf = getTransform(tftree,targetframe,sourceframe);
```

determines the current transformation in terms of translation vector and quaternion between the `targetframe` and `sourceframe`.

```
tform = getTransform(tftree,'base_link','map');
```

This code defines a message of type `PointStamped` w.r.t. the 'map' frame.

```
pt = rosmessage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'map';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;

ptbaselink = transform(tftree,'base_link',pt);
```

transforms the ROS message from the message frame `pt.Header.FrameId` to the `'base_link'` frame.

```
tfpt2 = apply(tform,pt);
```

applies the previously stored transformation `tform` to the `PointStamped` message.

```
tfentity = transform(tftree,targetframe,entity,'msgtime');
```

uses the timestamp in the header of the message, entity, as the source time to retrieve and apply the transformation. This option is useful to compensate for discrepancies between the timestamp of the message and the time instance at which the transform is computed.

```
ptbaselink = transform(tftree,'base_link',pt,'msgtime');
```

interpolates the transform to frame `'base_link'` to the timestamp in the header of the message `pt.Header.Stamp` rather than referring to the current rostime.

ROS Frames in Mobile Robot Navigation

The frame `base_link` is rigidly attached to the mobile robot base. For mobile robots the origin of `base_link` is typically at the center of the robot, the x-axis is along the direction of motion, and the z-axis is in vertical direction.

The frame `odom` is a world-fixed frame. The `odom` frame is computed based on an odometry source, such as wheel encoders, visual odometry or an inertial measurement unit. The pose of the mobile robot in the `odom` frame drifts due to the uncertainty of encoder information. Odometry errors accumulate over time without any bounds. Therefore the `odom` frame is useful for short term predictions of the actual robot motion but does not provide a long-term global reference. The pose of a robot in the `odom` evolves in a continuous fashion without discrete jumps. The frame `odom` is corrected by localization methods such as Monte Carlo Localization that utilize range sensor data to correct the robot pose w.r.t. to a map. The origin of the `odom` frame corresponds to the initial pose of the robot, thus does not necessarily coincide with the origin of the `map` frame.

The `map` is a world fixed frame, with a z-axis pointing upwards. The pose of a mobile platform, relative to the map frame, should not significantly drift over time. The map frame is not continuous, as the robot pose in the map frame changes in discrete jumps during correction steps.

A localization scheme, such as Adaptive Monte Carlo Localization (AMCL) constantly corrects the robot pose in the map frame based on sensor observations and a map of the environment, therefore eliminating the drift of odometry errors, but causing discrete jumps whenever novel sensor data is incorporated into the robot pose estimate. The `map` frame provides a long-term global reference for navigation.

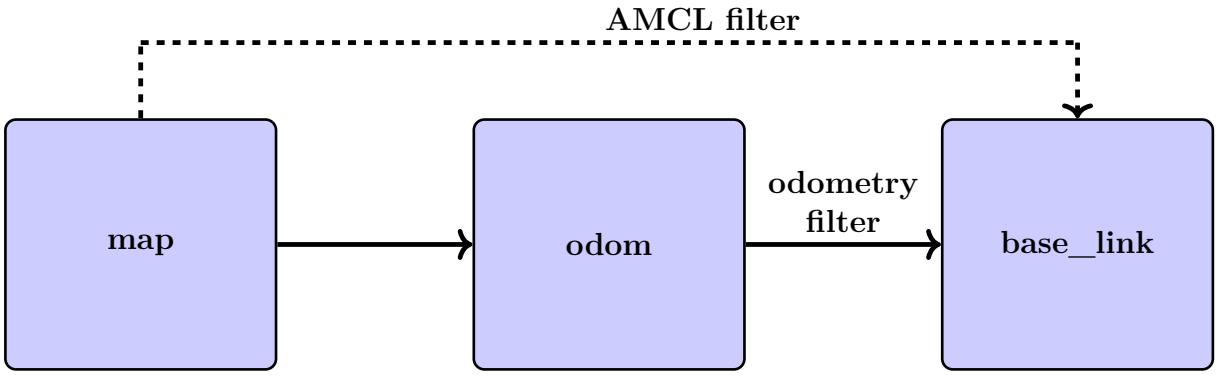

Figure 5: Relationship between frames `map, odom and base_link` [5]

Figure 5 illustrates the relationship between `map, odom` and `base_link` frame. Each coordinate frame has one parent coordinate frame, and any number of children coordinate frames. Notice, that the dashed line indicates that the transform between `map` and`base_link` is purely internal to the ACML filter and not published.

[5] source: RST

The `map` frame is the parent of `odom`, and `odom` is the parent of `base_link`. Although intuition suggests that both `map` and `odom` should be attached to `base_link`, this is not feasible since each frame only has one parent.

The transform from `odom` to `base_link` is estimated and broadcast by the odometry filter. Notice, that the estimated transform diverges from the true robot pose in the world as odometry errors accumulate over time.

In order to compensate for odometry errors the internal transform from `map` to `base_link` is computed by Adaptive Monte Carlo Localisation (AMCL). However, AMCL in itself does not broadcast the transform from `map` to `base_link` directly. Instead, it first receives the transform from `odom` to `base_link`, and then utilizes this information to broadcast the transform from `map` to `odom`. In other words, the localization scheme corrects the origin of the `odom` frame w.r.t. to its parent frame `map` such that the consecutive transform `map` to `base_link` via `odom` is compliant with the sensor observations and the map of the environment.

The `map` and `odom` are both global frames, so in an ideal world, the robot pose would remain identical in both frames. The `odom` and `map` filter create transforms from `odom` to `base_link` based on odometry information. The localization scheme determines the transform from `map` to `base_link` based on sensor and map data. As the frame `base_link` can only have one parent it can not directly correct `map` to `base_link` which does not exist as a parent child transform on the ROS transformation tree. Instead the map instance of AMCL looks up the transform from `odom` to `base_link` of the odometry filter and uses that transform, along with its own internal `map` to base_link transform, to obtain the corresponding `map` to `odom` transform.

The following assignment investigates the evolution of the transform `odom` to `base_link` according to odometry along with the corrections between `map` and `odom` due to localization.

1) Restart the Gazebo Simulation with the `world` environment and launch the navigation node.

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
roslaunch turtlebot3_navigation turtlebot3_navigation.launch
```

   Set the initial pose of the robot in RViz according to the location in the Gazebo simulation using the button "2D Pose Estimate".

2) Restart the global Matlab ROS node with `rosinit`

3) Retrieve the ROS transformation tree and check whether the frames `map, odom` and `base_link` are available.

4) Instantiate subscribers for the topics `' /move_base_simple/goal'` and `'/clicked_point'`. The topic `'/clicked_point'` contains the most recent published point in RVIZ. If

the "Publish Point" button is not active in toolbar, select the "+" button and add the publish point to tools.

5) Receive the published point message from the topic `'/clicked_point'` and create a rated while loop at a rate of 10 Hz.

6) Within the loop, transform the published point message into the frames `map, odom` and `base_link`. Subsequently, convert the message into its ordinary components with the helper function `PointStampedMsg2Point` .

7) Plot of x- and y- components of the published point w.r.t. the three different frames while setting `hold on` to superimpose the points of every loop cycle. Observe the evolution of the published point w.r.t. the three frames as the robot is moving.

8) Publish a point in RVIZ in the `map` frame and start the navigation in RVIZ by setting a navigation goal.

9) Repeat the experiment but this time publish the point in the `odom` frame. Select the frame in RViz under Displays -> Global Options -> Fixed Frame.

10) Repeat the experiment but this time publish the point in the `base_link` frame.

Point Cloud Mapping with Known Poses

The lab `Robotics System Toolbox I` was concerned with superposition of laser scan range data. As the laser range readings are associated with a robocentric frame the direct superposition is not consistent across scans. Robotic map building is related to cartography. The objective is to utilize a mobile robot to explore and construct a map of the environment. Map learning is inherently intertwined with localization, and a difficulty arises when errors in localization are incorporated into the map. This problem is commonly referred to as Simultaneous Localization and Mapping (SLAM) and is covered in a later lab. In this assignment, we resort to a technique known as mapping from known poses. It is assumed that the robot poses, in our case the transform between `map` and `/base_scan` frame are error free. The point cloud data of the laser range sensor is transformed from the laser frame `/base_scan` onto to the global map frame and superimposed with previous scans.

This assignment processes the range data as a point cloud on the topic `/camera/depth/points`. The point cloud message `sensor_msgs/PointCloud2.msg` contains the range readings as an array of 3D points with coordinates $[xyz]$. ROS has a dedicated Point Cloud Library (or PCL) for point cloud processing. The PCL framework contains numerous state-of-the art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation.

```
xyz = readXYZ(pcloud);
```

extracts the $[x\ y\ z]$ coordinates from all points in the point cloud object, `pcloud`, and returns them as an n-by-3 matrix of n 3-D point coordinates.

11) Inspect ROS topics and check for a topic `/camera/depth/points`. This topic contains message of type `sensor_msgs/PointCloud2` that contains the laser scan range data as a point cloud in the laser base frame `/camera_rgb_optical_frame` .

12) Inspect the particular topic `/camera/depth/points` and inspect information about the message type `sensor_msgs/PointCloud2`.

13) Instantiate a subscriber for the topic `/camera/depth/points` and receive a message. Check the field `Header.FrameId` to figure out the frame to which the point cloud data is associated.

14) Inspect the the available frames in the ROS transformation tree and check for the `/map` and `/camera_rgb_optical_frame` frames.

15) Verify that the transform between `/camera_rgb_optical_frame` and `/map` is available with `canTransform`.

16) Transform the point cloud message from its current frame to the `/map` frame with `transform`.

17) Extract the x-y-z-coordinates of the point cloud in the `/map` frame with . Convert the x-y-z array to a `pointCloud` object and randomly downsample the point cloud to 1% of the original one. Finally, visualize the x-y-z coordinates in a figure with `pcshow`.

18) Create a rated while loop such that it receives, transforms and downsamples the laser point cloud w.r.t. the global static `map` frame and accumulates the latest downsampled point cloud in an array in every loop cycle. To access the x-y-z coordinates of the points from a `pointCloud` object, use its `Location` attribute. Visualize the mapping process using the `pcplayer`. For this, initialize it before the loop using

```
player = pcplayer([-5, 5], [-5, 5], [0, 2])
```

In every loop cycle, update player with the current accumulated point cloud using the method `view`. Run the loop and set a navigation goal for the robot in RViz.

19) Improve the accuracy of mapping by applying the transform w.r.t. the time stamp of the point cloud message.

20) Investigate the quality of the superimposed point clouds if you transform the point cloud to the `odom` rather than the `map` frame.

Scan Matching with the Normal Distribution Transform

You might have noticed a misalignment of the point cloud data between consecutive scans. This error is attributed to the uncertainty of the pose estimate and to possible discrepancies between the time instance at which the laser scan is taken and the instance at which the transform is computed. Scan matching provides an alternative to Monte Carlo localization to compensate for the odometry drift.

In this assignment, you create a map of the environment using scan matching. The goal of scan matching is to find the transform between the two robot poses at which the scans are taken. The scans are aligned based on the shapes of their overlapping features.

To estimate this pose, the Normal Distributions Transform (NDT) algorithm [4] subdivides the laser scan into 2D cells and each cell is assigned a corresponding normal distribution according to algorithm (1). The distribution $N(\mu_{jk}, \sigma_{jk})$ represents the probability density of points in cell c_{jk}.

Algorithm 1 Normal distribution transform

Input: Set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ of particles scan points $\mathbf{x}_i$, grid of cells c_{jk} in the robots local environment $\mathcal{C} = \cup c_{jk}$

1: **for** each cell c_{jk} **do**
2: collect all points $\mathbf{x}_i \in c_{jk}$
3: calculate the mean $\boldsymbol{\mu}_{jk} = \frac{1}{n} \sum_{\mathbf{x}_i \in c_{jk}} \mathbf{x}_i$
4: calculate the covariance matrix $\boldsymbol{\Sigma}_{jk} = \frac{1}{n} \sum_{\mathbf{x}_i \in c_{jk}} (\mathbf{x}_i - \boldsymbol{\mu}_{jk})(\mathbf{x}_i - \boldsymbol{\mu}_{jk})^T$
return normal distribution transforms $N(\boldsymbol{\mu}_{jk}, \boldsymbol{\Sigma}_{jk})$;

The probability of observing the 2D-point $\mathbf{x}$ in cell c_{jk} is given by the normal distribution.

$$p(\mathbf{x}) \sim e^{-\frac{(\mathbf{x}-\boldsymbol{\mu}_{jk})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu}_{jk})}{2}} \tag{23}$$

Normal distribution transforms share some feature with probability occupancy grids. The occupancy grid represents the general probability of a cell being occupied, whereas the normal distribution transform reflects the probability of a sample at a particular location $\mathbf{x}$ within the cell.

Once the probability density is calculated, an optimization method estimates the relative pose between the current laser scan and the reference laser scan. To speed up the convergence of the method, an initial guess of the pose, typically obtained from odometry, is provided.

The objective of the scan alignment is to recover the relative transform between overlapping scans taken at two nearby robot poses. Given two scans taken at two different poses the algorithm in recovers the relative transform $\Delta x, \Delta y, \Delta\theta$ between the two poses. The

Algorithm 2 Scan matching with normal distribution transform

Input: Two scans $\mathcal{X}^{(1)} = \{\mathbf{x}_1^{(1)}, \ldots, \mathbf{x}_N^{(1)}\}$, $\mathcal{X}^{(2)} = \{\mathbf{x}_1^{(2)}, \ldots, \mathbf{x}_M^{(2)}\}$

1: calculate normal distribution transform $N(\boldsymbol{\mu}_{jk}, \boldsymbol{\Sigma}_{jk})$ for $\mathcal{X}^{(1)}$
2: estimate initial transform $\Delta x, \Delta y, \Delta\theta$ from odometry or set it to zero;
3: **repeat**
4: map each sample $\mathbf{x}_i^{(2)} \in \mathcal{X}^{(2)}$ to the corresponding point $\hat{\mathbf{x}}_i^{(2)} = T(\mathbf{x}_i^{(2)}, \Delta x, \Delta y, \Delta\theta)$ in the first frame according to $\Delta x, \Delta y, \Delta\theta$
5: determine the corresponding normal distribution $N(\boldsymbol{\mu}_{jk}, \boldsymbol{\Sigma}_{jk})$ for which $\hat{x}_i^{(2)} \in c_{jk}$.
6: calculate the probability $p(\hat{\mathbf{x}}_i^{(2)})$ of the mapped point $\hat{\mathbf{x}}_i^{(2)} \in c_{jk}$ according to $p(\hat{\mathbf{x}}_i^{(2)}) \sim e^{-\frac{(\hat{\mathbf{x}}_i^{(2)} - \boldsymbol{\mu}_{jk})^T \boldsymbol{\Sigma}^{-1} (\hat{\mathbf{x}}_i^{(2)} - \boldsymbol{\mu}_{jk})}{2}}$;
7: calculate the score of the transform $\Delta x, \Delta y, \Delta\theta$ as the sum of probabilities $\mathbf{score}(\Delta x, \Delta y, \Delta\theta) = \sum_i^M p(\hat{\mathbf{x}}_i^{(2)})$
8: calculate an improved transform by numerical optimization of $\mathbf{score}(\Delta x, \Delta y, \Delta\theta)$
9: **until** convergence of transform $\Delta x, \Delta y, \Delta\theta$;
return transform $\Delta x, \Delta y, \Delta\theta$;

first four steps are straightforward: The construction the normal distribution transform in step 1 is done by the algorithm 1. The initial estimate of the transform $\Delta x, \Delta y, \Delta\theta$ is obtained from odometry. In case odometry is not available the initial transform is set to zero. The second scan $\mathcal{X}^{(2)}$ is mapped according to $\hat{\mathbf{x}}_i^{(2)} = T(\mathbf{x}_i^{(2)}, \Delta x, \Delta y, \Delta\theta)$ given by

$$\begin{bmatrix} \hat{x}_i \\ \hat{y}_i \end{bmatrix} = \begin{bmatrix} \cos\Delta\theta & -\sin\Delta\theta \\ \sin\Delta\theta & \cos\Delta\theta \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{24}$$

The corresponding normal distribution $N(\boldsymbol{\mu}_{jk}, \boldsymbol{\Sigma}_{jk})$ is a simple lookup of $\hat{\mathbf{x}}_i^{(2)}$ in the grid of the first scan. The negative **score** is the objective subject to optimization by a numerical optimization scheme such as Newton or Quasi-Newton methods. Actually it is straight forward to calculate the gradient and Hessian of $\mathbf{score}(\Delta x, \Delta y, \Delta\theta)$ w.r.t. the parameter vector $\Delta x, \Delta y, \Delta\theta$ in a closed analytical form.

The scan match algorithm allows it to track the current robot pose from an initial start pose. For that purpose the relative transforms $T(\Delta x(t_i), \Delta y(t_i), \Delta\theta(t_i))$ at time instance t_i are simply chained by postmultiplication to obtain the absolute pose at time t_n w.r.t. to the initial frame at t_0.

$$T = T(\Delta x(t_1), \Delta y(t_1), \Delta\theta(t_1))\, T(\Delta x(t_2), \Delta y(t_2), \Delta\theta(t_2)) \ldots T(\Delta x(t_n), \Delta y(t_n), \Delta\theta(t_n)) \tag{25}$$

If you apply scan matching to a sequence of scans, it generates a rough map of the environment that the robot traverses. Scan matching plays a crucial role in robotic tasks such as position tracking and Simultaneous Localization and Mapping (SLAM).

```
[pose, stats] = matchScans(currScanRanges, currScanAngles, refScanRanges,...
    refScanAngles, 'SolverAlgorithm', 'fminunc', 'InitialPose', pose);
```

estimates the relative `pose` between a reference laser scan and current laser scan using the normal distributions transform (NDT) [4]. Specify the optimization solver algorithm as `'fminunc'`. The `'InitialPose'` determines the initial guess of the current pose relative to the reference laser scan. The output argument `stats` provides additional statistics about the scan matching result. Scan matching replaces odometry if one integrates the relative poses over time. This is accomplished by mapping the pose vector $[x\ y\ \theta]$ into a corresponding homogeneous transformation matrix between consecutive poses:

```
relativeTform = trvec2tform([pose(1) pose(2) 0])*eul2tform([pose(3) 0 0]);
```

The absolute transform is initialized with the 4-by-4 identity matrix

```
absoluteTform = eye(4);
```

The relative poses are integrated into the absolute transform by post-multiplication of the homogeneous transformation matrices.

```
absoluteTform = absoluteTform*relativeTform;
```

Create a loop for processing the scans and mapping the area. The laser scans are processed in pairs. Define the previous scan as reference scan and the latest scan as current scan. The two scans are then passed to the scan matching algorithm and the relative pose between the two scans is computed with `matchScans`.

21) Instantiate a subscriber `scanSub` for the laser scan topic `/scan` and assign the received a scan message to the variable `referenceScanMsg`.

22) Extract the `scanAngles` from the message with `readScanAngles`. Notice that the `scanAngles` remain the same throughout the loop as only the range readings change but noot the angular resolution of the scanner. Initialize the `relativePose` with `[0 0 0]`. Initialize the `absoluteTform` with 4×4 identity matrix.

23) Within a while loop, receive the current scan message `currentScanMsg`.

24) Extract the ranges `refScanRanges` and `currScanRanges` from the `Ranges` field of the scan messages.

25) Determine the `relativePose` between the two scans with `matchScans`. In each loop iteration, provide the `relativePose` as `'InitialPose'` to the `matchScans` function. Skip the following steps, if the scan matching result is poor which is indicated by a low matching score.

```
[relativePose, stats] = matchScans(currScanRanges, scanAngles, refScanRanges
 , scanAngles,  'SolverAlgorithm', 'fminunc', 'MaxIterations', 500, '
 InitialPose', relativePose);

if stats.Score / numel(currScanRanges) < 0.5
   continue;
end
```

26) Determine the true range readings that are below the maximum range reading

```
inRange=find(currScanRanges < double(currentScanMsg.RangeMax));
```

27) Convert the current scan readings in range from polar into Cartesian coordinates `scanX`, `scanY` with `pol2cart`.

28) Determine the homogeneous transform of the relative pose and integrate with the absolute transform by postmultiplication of the two homogeneous transformation matrices.

```
relativeTform = trvec2tform([relativePose(1) relativePose(2) 0])*eul2tform
([relativePose(3) 0 0]);
absoluteTform=absoluteTform * relativeTform;
```

29) Apply the absolute transform to the Cartesian coordinates of the scan points `scanX`, `scanY`.

30) Plot the transformed scan points while setting `hold on` to superimpose the scans of every loop cycle.

31) The current scan message becomes the reference scan message of the next iteration of the loop.

32) Run the loop and set a navigation goal for the robot in RViz. Inspect the created laser scan matching map.

Occupancy Grid Mapping

Occupancy grids represent a robot workspace as a discrete grid. In mapping applications, they enable the integration of sensor range data in a discrete map. Path planning rests upon occupancy grid to plan collision-free path from start to goal. Localization schemes employ occupancy grids to match sensor readings with the map of the environment.

Binary occupancy grid of objects `robotics.BinaryOccupancyGrid` represent the occupied workspace (obstacles) by true and the free workspace by false values.

A probability occupancy grid specified as a `robotics.OccupancyGrid` object assigns probabilities of occupancy to cells. Each cell in the occupancy grid has a value representing

the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell is occupied by an obstacle. Values close to 0 indicate certainty that the cell is unoccupied. Laser range finders, cameras are commonly used to detect obstacles in the environment.

```
map = robotics.OccupancyGrid(15, 15, 20);
map.GridLocationInWorld = [-7.5 -7.5];
show(map);
```

instantiates an object of a probability occupancy grid of width and height 15 meter and a resolution of 20 cells per meter along each dimension. The property `GridLocationInWorld` determines the grid location in world coordinates, that way the world coordinate $(0, 0)$ corresponds to the grid center at $(7.5, 7.5)$ meter. `show` visualizes the occupancy grid map such that white areas denote free space, dark areas correspond to occupied cells, and gray areas are regions that are not charted yet.

```
insertRay(map,pose,ranges,angles,maxrange);
```

inserts one or more range sensor observations in the occupancy grid, map using the input ranges and angles to get ray endpoints. The ray endpoints are considered free space if the input ranges are below maxrange. Cells observed as occupied are updated with an observation of 0.7. All other points along the ray are treated as obstacle free and updated with an observation of 0.4. The `pose` is specified as an $[x\ y\ \theta]$ vector. One of the upcoming lab gets back to issue of simultaneous localization and map building (SLAM) with occupancy grids.

The remaining tasks on occupancy grid mapping are optional.

33) Instantiate a probabilistic occupancy grid of object class `OccupancyGrid` centered at $(7.5, 7.5)$ meter.

34) Convert the homogeneous transformation matrix `absoluteTform` back into an ordinary pose vector `absolutePose` in terms of $[x\ y\ \theta]$ with `tform2tvec` and `tform2eul`.

35) Within the while loop insert the laser range readings with the estimated absolute pose into the occupancy grid with `insertRay`.

```
insertRay(map, absolutePose, currScanRanges, scanAngles, double(currentScanMsg.
    RangeMax));
```

36) Visualize the occupancy grid map with `show`.

37) Vary the maximum linear and angular robot velocity, when moving in the map, and observe the quality of the created occupancy grid map. To set a maximum velocity for the turtlebot use the rqt reconfigure gui:

```
rosrun rqt_reconfigure rqt_reconfigure
```

The maximum linear and angular velocity can be changed in the DWA tab.

References

[1] ROS Tutorials, `http://wiki.ros.org/ROS/Tutorials`, 2016

[2] Jason M. O'Kane, A Gentle Introduction to ROS, `https://cse.sc.edu/~jokane/agitr/`, 2015

[3] K. Waldron, J. Schmiedeler, Kinematics, Springer Handbook of Robotics, Springer, `http://link.springer.com/referenceworkentry/10.1007/978-3-540-30301-5_2`, 2015

[4] P. Biber, W. Strasser, The normal distributions transform: A new approach to laser scan matching, Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2743-2748, 2003

[5] B. Siciliano et al, Robotics: Modelling, Planning and Control, Springer, 2008