# 3D Sensor Pose Calibration using Least Squares

Elective in Robotics 2013/14 - Least Squares & SLAM final project

Federico Nardi

July 21, 2014

# Contents

# 1 Introduction

Robot navigation refers to the set of techniques that provide a mobile robot with the ability to reach a user-defined goal, to accomplish this task the robot must be capable of exploring the surrounding environment, building an exact representation of it and determine its position inside this representation.To solve these problems it's necessary to use mathematicals tools and, in particular, optimization methods adapted to a probabilistic framework. In my project I faced the problem of calibrating the pose of a 3D sensor mounted on a moving platform using the so-called Least Squares estimation method. This is a standard technique and in the following I'll show in detail the steps involved to implement it.

# 2 Motivation

Of course, one could solve this problem by using very precise measurement tools (like laser interferometers), but these solutions tend to be very expensive and time consuming. For these reasons, it's better to use an estimation technique that can be implemented via software and can be executed any time it's needed. The Least Squares method is an iterative technique, based on the minimization of an error defined between the actual measurement and its prediction based on the current estimate of the system state. At each step a correction, proportional to the error, is applied to the estimate until no substantial change is done: which means that the estimation reflects the measurement outcomes.

# 3 Problem Formulation

In this case, the system state is the pose of the 3D sensor mounted on the robot, and the error is defined between the camera motion obtained with high accuracy through a *Visual Odometry* technique and the camera motion predicted by applying the sensor transformation to the robot odometry. Having said this, let's move to a more formal definition of the problem.

As already mentioned, the system state is the position and the orientation of the 3D sensor, so in mathematical terms this is represented by a matrix $K \in SE(3)$. This representation doesn't lie in an Euclidean Space, so it's necessary to define the increments and the operator that allows to sum them to the current estimate. To do this, it's a good choice to represent the increments in a minimal form:

$$\Delta k = \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix} \tag{1}$$

where $x$, $y$ and $z$ are the position coordinates and $\phi$, $\theta$ and $\psi$ are the *Euler Angles* associated to the $ZYX$ convention. This vector can be added to $K$ by means of the $\boxplus$ operator, defined in the following way:

$$K' = K \boxplus \Delta k = K \cdot v2t(\Delta k) \tag{2}$$

where $v2t(\cdot)$ is the function that maps $\Delta k$ to its extended representation. In this problem, the measurements are the sensor poses acquired through VO so, again, they are represented by an isometry $Z \in SE(3)$; this must be compared with the measurement prediction based on the current system state estimate, which is computed as follows:

$$P = K^{-1} \cdot O \cdot K \tag{3}$$

where $O \in SE(3)$ is the robot odometry computed from encoder ticks. After that, it's possible to define the *error function* that we want to minimize in order to fit the system state with the measurement:

$$e = P \boxminus Z = t2v(Z^{-1} \cdot P) \tag{4}$$

where $t2v(\cdot)$ maps a matrix belonging to $SE(3)$ to its minimal representation.

# 4 Implementation

```
%------------------------------------------------------------
%Least Squares Sensor Pose Calibration
%------------------------------------------------------------

clear;clc;

%initial guess of the state vector which is made of the
%sensor pose [Xk;Yk;Zk;Rk;Pk;Yk]
x = [0.102259;-0.0115757;0.4;-1.5617;0.0120;-1.5606];

%measurement covariance
Omega = eye(6);

%linear system matrices for solving Least Squares
H=zeros(6);
b=zeros(6,1);

%default increment for computing numeric Jacobian
eps = 1.e-5;

%increments vector
deltaX=zeros(6,1);

%utility variables
count = 0;
iterate=2;
th = 0.0002;
max_iterations = 100;
l = 1000;

%load data file
load output.txt
[m, n] = size(output);

if m ~= 0
    fprintf('[INFO]: output.txt file has %d lines\n',m)
        ;
    disp('Press any key to continue...');
```

```
38      pause
39  end
40
41  while (iterate > 1)
42
43      err_norm = 0;
44      count = count + 1;
45      disp('Iteration: ')
46      disp(count)
47      for i = 1:m
48
49          %acquire data from file
50          measurement = transpose(output(i,1:6));
51          odometry = transpose(output(i,7:n));
52
53          X = v2t(x);
54          M = v2t(measurement);
55          O = v2t(odometry);
56
57          %predict sensor pose from robot pose and
                transform
58          P = prediction(O,X);
59
60          %compute difference between prediction and "
                ground truth"
61          e = error_function(P,M);
62          err_norm = err_norm + norm(e);
63
64          %Compute numeric Jacobian
65          J = zeros(6,6);
66          x_up = x;
67
68          for j = 1:6
69              x_up(j) = x_up(j) + eps;
70              perturbedP = prediction(O, v2t(x_up));
71              e_up = error_function(perturbedP,M);
72              J(:,j) = (e_up - e)/eps;
73              x_up(j) = x(j);
74          end
75
76          %update the linear system matrices H and b
```

```
77          H = H + (J')*Omega*J;
78          b = b + (J')*Omega*e;
79
80      end
81
82      err_norm
83      H = H + l * eye(6);
84      deltaX = -H\b ;
85      DeltaX = v2t(deltaX);
86      x = t2v(X*DeltaX);
87      current_x = x'
88      deltaX_norm = norm(deltaX)
89
90      if ( deltaX_norm < th || count > max_iterations )
91          iterate = 0;
92          x
93          disp('Stop iterating!!!')
94      end
95
96  end
```

## v2t.m

```
1   function T = v2t(v)
2
3           [m, n] = size(v);
4
5           if ( m == 6 && n == 1 )
6                   T = eye(4);
7                   T(1:3,4) = v(1:3);
8
9                   cb = cos(euler(1));
10                  ch = cos(euler(2));
11                  ca = cos(euler(3));
12                  sb = sin(euler(1));
13                  sh = sin(euler(2));
14                  sa = sin(euler(3));
```

```
15
16 T(1:3,1:3) = [ch*ca;-ch*sa*cb + sh*sb;ch*sa*sb + sh*cb;
17                sa;ca*cb;-ca*sb;
18                -sh*ca;sh*sa*cb + ch*sb;-sh*sa*sb + ch*cb
                      ];
19         end
20 end
```

### t2v.m

```
1  function v = t2v(T)
2
3          [m n] = size(T);
4
5          if ( m == 4 && n == 4 )
6                  v = zeros(6,1);
7                  v(1:3) = T(1:3,4);
8
9                  if (T(2,1) > 0.998)
10                         heading = atan2(T(1,3),T(3,3));
11                         attitude = pi/2;
12                         bank = 0;
13                 end
14
15                 if (T(2,1) < -0.998)
16                         heading = atan2(T(1,3),T(3,3));
17                         attitude = pi/2;
18                         bank = 0;
19                 end
20
21                 heading = atan2(-T(3,1),T(1,1));
22                 bank = atan2(-T(2,3),T(2,2));
23                 attitude = asin(T(2,1));
24
25                 v(4:6) = [bank;heading;attitude];
26         end
27 end
```