```
%==========================
%  Name:                hw5_1.m
%
%  Author:              Kairi Kozuma
%
%==========================

% Points in the world
qWpts = [2,8,3,4,6,5,5;7,6,2,8,8,5,4;5,4,4,4,2,2,2;1,1,1,1,1,1,1];

% Left camera points
rpL = [384,189,22,358,313,215,165;137,169,287,172,243,298,316;1,1,1,1,1,1,1];

% Right camera points
rpR = [579,433,253,597,580,457,408;178,250,375,220,309,379,399;1,1,1,1,1,1,1];

% psi matrix
psi = [500,0,320;0,-500,240;0,0,1];
psi = [psi,zeros(3,1)];

% 1) R, T pairs for each camera, with respect to the world frame
disp('1) R, T pairs for each camera, with respect to the world frame');
[R_LW, T_LW] = extrinsicCalib(rpL, qWpts, psi);
[R_RW, T_RW] = extrinsicCalib(rpR, qWpts, psi);

disp('Left camera R with respect to world frame:');
scaleL = nthroot(det(R_LW),3);
R_LW = R_LW ./ scaleL;
disp(R_LW);

disp('Left camera T with respect to world frame:');
T_LW = T_LW ./ scaleL;
disp(T_LW);

disp('Right camera R with respect to world frame:');
scaleR = nthroot(det(R_RW),3);
R_RW = R_RW ./ scaleR;
disp(R_RW);

disp('Right camera T with respect to world frame:');
T_RW = T_RW ./ scaleR;
disp(T_RW);

% Check if transformation matrix is correct
G_LW = [R_LW, T_LW; 0,0,0,1];
G_RW = [R_RW, T_RW; 0,0,0,1];

% rpL2 = psi * G_LW * qWpts;
% rpL2 = round(rpL2 ./ rpL2(3,:));
% disp(rpL);
% disp(rpL2);
% rpR2 = psi * G_RW * qWpts;
% rpR2 = round(rpR2 ./ rpR2(3,:));
% disp(rpR);
% disp(rpR2);

% 2) R, T pair for the right camera relative to the left camera
disp('2) R, T pair for the right camera relative to the left camera');
```

```
% Obtain G_LR
G_LR = G_LW * inv(G_RW);

% Extract R and T
R_LR = G_LR(1:3,1:3);
T_LR = G_LR(1:3,end);

disp('Right camera R with respect to left camera frame');
disp(R_LR);

disp('Right camera T with respect to left camera frame');
disp(T_LR);

%============================ extrinsicCalib ============================
%
%
%  Given a set of image points plus the world coordinates that they came
%  from, and thirdly the intrinsic camera matrix, solve for the extrinsic
%  parameters associated to the camera rig.
%
%
%  function [R_CW, T_CW] = extrinsicCalib(rp, qWpts, psi)
%
%  Inputs:
%    rp     - Points in the image as homogeneous rays.
%    qWpts  - Points in the world in homogeneous form.
%    psi    - psi matrix
%
%  Outputs:
%    R_CW      - The rotation of the camera frame relative to the world frame.
%    T_CW      - The translation of the camera frame relative to the world frame.
%
%============================ extrinsicCalib ============================
function [R_CW, T_CW] = extrinsicCalib(rp, qWpts, psi)

% Convert image pts (in pixels) to rays (in world length units)
sz = size(qWpts);
masterMatrix = zeros(2*sz(2),12);

for index = 1:sz(2)
    rmat = makeRMat(rp(:,index));
    Qmat = makeQMat(qWpts(:,index));
    mat = rmat * Qmat;
    index2 = 2*index;
    masterMatrix(index2:index2 + 1,:) = mat(1:2,:);
end

% % Run for each column in rp and qWpts to construct 12 x 12 matrix.
% for i = 1:size(rp,2)
% end

% Perform SVD and reconstruct extrinsic parameters (scale by nth root of det)

[UU SS VV] = svd(masterMatrix);
szvv = size(VV);
M = VV(:,szvv(2));
M = (reshape(M, 4, 3))';

G_CW = inv(psi(1:3,1:3)) * M;

R_CW = G_CW(1:3,1:3);
```

```
T_CW = G_CW(1:3,end);

end


%============================== makeRMat ==============================
%
%  function mat = makeRMat(vector)
%
%
%  INPUT:
%    vector - 3 x 1 vector
%
%============================== makeRMat ==============================
function mat = makeRMat(vector)
mat = [0, vector(3), -vector(2); -vector(3), 0, vector(1); vector(2), -vector(1), 0];
end


%============================== makeQMat ==============================
%
%  function mat = makeQMat(vector)
%
%
%  INPUT:
%    vector - 4 x 1 vector
%
%============================== makeQMat ==============================
function mat = makeQMat(qPts)
mat = [qPts',zeros(1,8); zeros(1,4), qPts', zeros(1,4);zeros(1,8), qPts'];
end
```

```
1) R, T pairs for each camera, with respect to the world frame
Left camera R with respect to world frame:
   -0.5632    0.8209   -0.1026
    0.2305    0.2799    0.9365
    0.7827    0.5062   -0.3482

Left camera T with respect to world frame:
   -3.1524
   -5.5655
    4.0902

Right camera R with respect to world frame:
    0.0069    0.9341   -0.3888
    0.0470    0.3825    0.9309
    0.9795   -0.0147   -0.0546

Right camera T with respect to world frame:
   -1.3297
   -6.6396
    4.7487

2) R, T pair for the right camera relative to the left camera
Right camera R with respect to left camera frame
    0.7932    0.1864   -0.5895
   -0.0973    0.9765    0.1892
    0.5879   -0.0817    0.7989
```

```
Right camera T with respect to left camera frame
    1.9397
   -0.1099
    0.5357
```

Published with MATLAB® R2016b

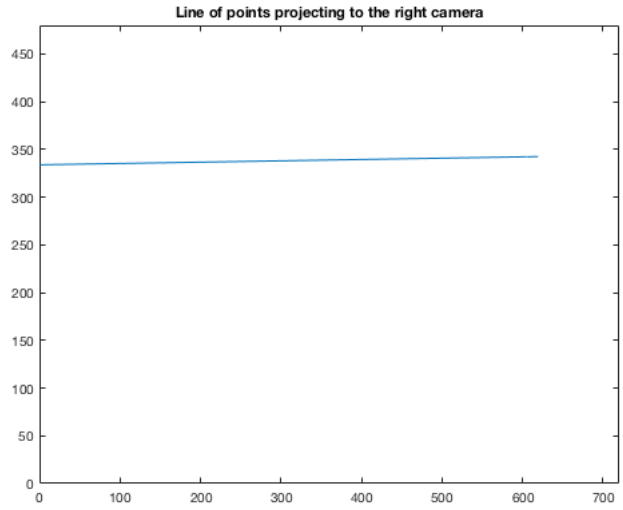
```
%==========================
%  Name:                hw5_2.m
%
%  Author:              Kairi Kozuma
%
%==========================

% Points in the world
pwpts = [13.1800000000000,24.7500000000000,14.4700000000000,15.8300000000000,22.5100000000000,12.5500000000000;-3.36200000000000,-11.7900000000000,-9.
sz = size(pwpts);
qwpts = [pwpts ; ones(1,sz(2))];

% Points in the image
rpts = [157,92,416,3,91,167;376,139,97,26,387,238;1,1,1,1,1,1];

% Get M matrix
M = calibrateM(qwpts, rpts);
fprintf('M matrix:\n');
disp(M);

% Test M matrix to see if original results obtained
test = (M * qwpts);
test = test./test(3,:);
fprintf('M * qwpts:\n');
disp(round(test(1:2,:)));

% F matrix
F = fliplr(eye(3));

% QR factorization
[Q U] = qr(F * (M(1:3,1:3))' * F);

% Obtain S matrix
S = diag(sign(diag(U)));

% Obtain R and K matrices
R_CW = (F * Q * S * F)';
K = (F * S * U * F)';

%trueK = K / K(3,3);

R_WC = R_CW';
N0 = M(1:3,end);
T_WC = -inv(M(1:3,1:3))* N0;
G_WC = [R_WC, T_WC; 0,0,0,1];
G_CW = inv(G_WC);
T_CW = G_CW(1:3,end);
psi = K;

disp('Psi matrix:');
disp(psi);

disp('D matrix R:');
disp(R_CW);

disp('D matrix T:');
disp(T_CW);

disp('Reconstruct M matrixM:');
M_new = [psi * R_CW, psi * T_CW];
disp(M_new);
disp(M);

%============================== calibrateM ==============================
%
%  function [M] = calibrateM(qPts, rPts)
%
%
%  INPUT:
%    qPts   - The points in world coordinates.
%    rPts   - The image points (in ray form).
%
%============================== calibrateM ==============================
function [M] = calibrateM(qPts, rPts)


%--(1) For each world point and image point pair, create the 2-row matrix,
%      and use them to create a master matrix.

sz = size(qPts);
masterMatrix = zeros(2*sz(2),2*sz(2));

for index = 1:sz(2)
    rmat = makeRMat(rPts(:,index));
    Qmat = makeQMat(qPts(:,index));
    mat = rmat * Qmat;
    index2 = 2*index;
    masterMatrix(index2:index2 + 1,:) = mat(1:2,:);
end
```

```
%--(2) Perform SVD using the master matrix and extract the projection
%      matrix.  Be careful about rows versus columns ...

[UU SS VV] = svd(masterMatrix);
szvv = size(VV);

M = VV(:,szvv(2));

M = (reshape(M, 4, 3))';

end


%============================== makeRMat ==============================
%
%  function mat = makeRMat(vector)
%
%
%  INPUT:
%    vector - 3 x 1 vector
%
%============================== makeRMat ==============================
function mat = makeRMat(vector)
mat = [0, vector(3), -vector(2); -vector(3), 0, vector(1); vector(2), -vector(1), 0];
end


%============================== makeQMat ==============================
%
%  function mat = makeQMat(vector)
%
%
%  INPUT:
%    vector - 4 x 1 vector
%
%============================== makeQMat ==============================
function mat = makeQMat(qPts)
mat = [qPts',zeros(1,8); zeros(1,4), qPts', zeros(1,4);zeros(1,8), qPts'];
end
```

```
M matrix:
   -0.0331    0.0003    0.0937   -0.9567
   -0.0519   -0.0730    0.0127   -0.2581
   -0.0002    0.0000    0.0001    0.0002

M * qwpts:
   157    92   416     3    91   167
   376   139    97    26   387   238

Psi matrix:
    0.0776    0.0000    0.0621
         0    0.0776    0.0466
         0         0    0.0002

D matrix R:
    0.3255   -0.0669    0.9432
   -0.1056   -0.9938   -0.0340
   -0.9396    0.0886    0.3306

D matrix T:
  -13.2727
   -4.0327
    1.1779

Reconstruct M matrixM:
   -0.0331    0.0003    0.0937   -0.9567
   -0.0519   -0.0730    0.0127   -0.2581
   -0.0002    0.0000    0.0001    0.0002

   -0.0331    0.0003    0.0937   -0.9567
   -0.0519   -0.0730    0.0127   -0.2581
   -0.0002    0.0000    0.0001    0.0002
```

Published with MATLAB® R2016b

```
%==========================
%  Name:                hw5_3.m
%
%  Author:              Kairi Kozuma
%
%==========================

% Psi matrix
psi = [400,0,360;0,-400,240;0,0,1];

% Visible point
pL = [16.595;-15.538;63.748];
rL = psi * pL;
pW = [33.1;-39.1;38.5];

scale_vec = [0.5:0.5:100];
pL_vec = pL .* scale_vec;

% Transformation matrix
R_WL = [0.913545457642601,-0.063627629171822,0.401729040058774;0.287606238475951,0.799453749866612,-0.527405302792764;-0.287606238475951,0.597348496800
R_WR = [0.994521895368273,-0.016351854232753,0.103241544429788;0.073912785203567,0.808411029059454,-0.583959337863936;-0.073912785203567,0.588391217607
T_WL = [-8.659258262890683;2.169872981077807;4.830127018922193];
T_WR = [10.659258262890683;5.830127018922193;1.169872981077807];

% G matrices
G_WL = [R_WL, T_WL;0,0,0,1];
G_WR = [R_WR, T_WR;0,0,0,1];
G_RL = inv(G_WR) * G_WL;

l_vec = [0:.1:1000];
for i = 1:size(l_vec,2)
    rR(:,i) = [psi,zeros(3,1)] * G_RL * [pL .* l_vec(i) ;1];
end

% Rescale to 1
rR = rR ./ rR(3,:);

% Plot the line
figure(1);
plot(rR(1,:),rR(2,:));
axis([0,720,0,480]);
title('Line of points projecting to the right camera');
```



Published with MATLAB® R2016b

Vision-Based Autonomous Vehicle Navigation of a Robot: Week 2

Members: Xiaoyi Jeremy Cai, Kairi Kozuma

TurtleBot: TB-02

PART 1: Basic Movements

Procedure

- Clone the Git repository into the ECE4580jc_kk directory located in the desktop of the turtlebot user with the following command:

```
git clone https://github.com/TurtlebotAdventures/turtlebot.git
```

- Inspect the code for controlling the robot in `goforward.py` and `draw_a_square.py`. Write down the specific commands to move the robot forward at a certain velocity, forward a certain distance, and turn a certain angle.
- Run the code by first setting up the ROS server on the turtlebot:

```
roslaunch turtlebot_bringup minimal.launch
```

- Run the python code:

`python goforwad.py` and `python draw_a_square.py`

- Observe the robot movement.

Investigation

1. How to command the turtle bot to move its wheels at predefined velocities:

```
move_cmd = Twist()                  # assign the movement vector to move_cmd

move_cmd.linear.x = SomeSpeed       # assign an x-direction speed

self.cmd_vel.publish(move_cmd)      # publish to move_cmd to move the turtlebot
```

2. How to command them to move the robot a certain distance:

```
r = rospy.Rate(5)    # Set the frequency to 5 Hz (or some other frequency)
```

```
for x in range(0,5): # Repeat 5 times, since 5 * 5 Hz = 1 second

  self.cmd_vel.publish(move_cmd) # Move forward at (move_cmd.linear.x) m/s for 1s

  r.sleep() # Sleep for (1 / frequency) seconds before sending same command

# The above code moves the robot 1 second forward at move_cmd.linear.x m/s

# The distance covered by the robot is (move_cmd.linear.x) meters
```

3. **How to command them to rotate for a certain amount (rather than at a certain speed):**

```
turn_cmd = Twist()            # initialize a turn command

turn_cmd.linear.x = 0         # set the x-direction speed to 0

turn_cmd.angular.z = radians(45)     # set angular speed to be 45 degree/s

                                     # radians() converts degree to radians

for x in range (1,10):

       self.cmd_vel.publish(turn_cmd)       # publish turning command to subscribers

       r.sleep()                            # sleep for 1/f seconds before next iteration
```

Adventure

1. **Do they agree with your background investigation?**

Yes, the code in both goforward.py and draw_a_square.py agree with the background investigation. The code in both uses a rate of 5Hz and a time interval of 2 seconds (by using a for loop that executes 10 times, with a (1/5) second sleep in between.

Explore

1. **Run the programs many times. How well does the robot execute the commands?**

When we ran "draw_a_square.py" on the tutlebot on the carpet floor, the turtlebot didn't turn exactly 90 degree for the first few attempts. However, after four to five attempts, the turtlebot started to turn correctly.

2. **If you work out the math for them, is it following the theoretical movements?**

The robot is close to the theoretical movements, but the turning angle is slightly off from what it should be, because the python "sleep()" does not provide any real time guarantee that it will return in the specified time interval.

3. **Try out different surfaces. How do they impact the robot movement?**

On the carpet, the robot did not turn as much as it needed to trace a square. There was minimal difference between the carpet surface and the hallway surface.

4. **What are the topics that are published to in order to send the commands? Do they agree with what you found?**

The topic that is published is "cmd_vel_mux/input/navi". The name is slightly different from the "cmd_vel" topic that was found in other code, but the comment in the goforward.py indicates that this may be necessary just for the turtlebot 2.

5. **Did you find any extra? If so, how do they relate to the ones here?**

There were no extra topics in the two programs that were inspected.

PART 2: Sensing

Procedure

- **Use the following command to find the location of the minimal launch file:**
  `sudo find . -name minimal.launch`
- **The file is located in** `/opt/ros/hydro/share/turtlebot_bringup/launch`
- `cd` **into that directory and list all files**
  `3dsensor.launch`
  `minimal.launch`
  `paired_public.launch`
  `minmimal_with_appmanager.launch`
- **Launch** 3dsensor.launch **file using:**

```
roslaunch turtlebot_bringup 3dsensor.launch
```

Investigation

1. **What are the ROS topics that must be subscribed to in order to get these sensor measurements? Identify the main two that provide the proper raw data associated to the image stream and to the depth stream for the kinect camera.**

- **/camera/depth_registered/image_raw**
- **/camera/rgb/image_raw**

2. **Name a few of the different image sensor topics and explain how they differ from the raw versions.**

/camera/rgb/image_mono: Monochrome unrectified image

/camera/rgb/image_color: Color unrectified image

/camera/rgb/image_rect_mono: Monochrome rectified image

Adventure

1. **Properly launch the necessary services to have kinect sensor data be published (beyond the linescan). Show that this worked by doing a `rostopic list` which should then output the kinect sensor messages.**

This is a screenshot of the list of topics in 3dsensor.

```
turtlebot@turtlebot:~$ rostopic list
/camera/camera_nodelet_manager/bond
/camera/debayer/parameter_descriptions
/camera/debayer/parameter_updates
/camera/depth/camera_info
/camera/depth/disparity
/camera/depth/image
/camera/depth/image/compressed
/camera/depth/image/compressed/parameter_descriptions
/camera/depth/image/compressed/parameter_updates
/camera/depth/image/compressedDepth
/camera/depth/image/compressedDepth/parameter_descriptions
/camera/depth/image/compressedDepth/parameter_updates
/camera/depth/image/theora
/camera/depth/image/theora/parameter_descriptions
/camera/depth/image/theora/parameter_updates
/camera/depth/image_raw
/camera/depth/image_raw/compressed
/camera/depth/image_raw/compressed/parameter_descriptions
/camera/depth/image_raw/compressed/parameter_updates
/camera/depth/image_raw/compressedDepth
/camera/depth/image_raw/compressedDepth/parameter_descriptions
/camera/depth/image_raw/compressedDepth/parameter_updates
/camera/depth/image_raw/theora
/camera/depth/image_raw/theora/parameter_descriptions
/camera/depth/image_raw/theora/parameter_updates
/camera/depth/image_rect
/camera/depth/image_rect/compressed
/camera/depth/image_rect/compressed/parameter_descriptions
/camera/depth/image_rect/compressed/parameter_updates
/camera/depth/image_rect/compressedDepth
/camera/depth/image_rect/compressedDepth/parameter_descriptions
/camera/depth/image_rect/compressedDepth/parameter_updates
/camera/depth/image_rect/theora
```

Here's a screenshot of the information published by "camera/rgb/camera_info"

```
header:
  seq: 3053
  stamp:
    secs: 1487180666
    nsecs: 717637290
  frame_id: /camera_rgb_optical_frame
height: 480
width: 640
distortion_model: plumb_bob
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [525.0, 0.0, 319.5, 0.0, 525.0, 239.5, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [525.0, 0.0, 319.5, 0.0, 0.0, 525.0, 239.5, 0.0, 0.0, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
```

2. **Python and ROS fortunately have harnessed the power of OpenCV. Using the OpenCV bridge code for python, demonstrate the ability to subscribe to the sensor data and display it. The libraries to import are ``cv2`` and ``cv_bridge``. The first one gives access to OpenCV code. The second one lets ROS and OpenCV work together (usually through conversion routines that translate data forms between the two system and their standards). Code up a python class for subscribing and displaying the kinect image streams. When run, this should just display to separate windows the color image data and the depth data. The ``cv2`` function for displaying an image is called ``imshow`` which expects the data to be in a specific**

format (either from 0 to 1, or from o to 255, I forget). The depth data may have to be normalized for proper display.

Where we found how to normalize depth_registered/image_raw:

http://answers.ros.org/question/58902/how-to-store-the-depth-data-from-kinectcameradepth_registeredimage_raw-as-gray-scale-image/

Module was demoed to Fu-Jen in the lab.