Elena Morara **July 2016**

Documentation for ODOMETRY AND MAPPING INSIDE PIPES

This document is aimed at reproducing results of Master Thesis "Odometry and Mapping inside Pipes", by Elena Morara.
Please contact the author elena294@gmail.com for any question and concern.

Summary of contents

1. Experimental setup

Hardware:
- SEA snake with 18 modules
- Go Pro Hero Session 4, fixed with tape at the last module
- Small flash light ThorFire KL02 LED 120 lumen
- PVC pipes whose internal walls are not completely white
- PVC straight junctions, forks and bends (45, 90, 180)

Software:
- MATLAB (2015A)
- C++ (g++, C++ 11)
- OpenCV for C++ (3.1)
- GoProStudio

Software packages from this work:
- C++: *visual_odometry*
- MATLAB: *snake_in_pipes*
- MATLAB: *odometry_and_mapping*

2. Overview of the procedure

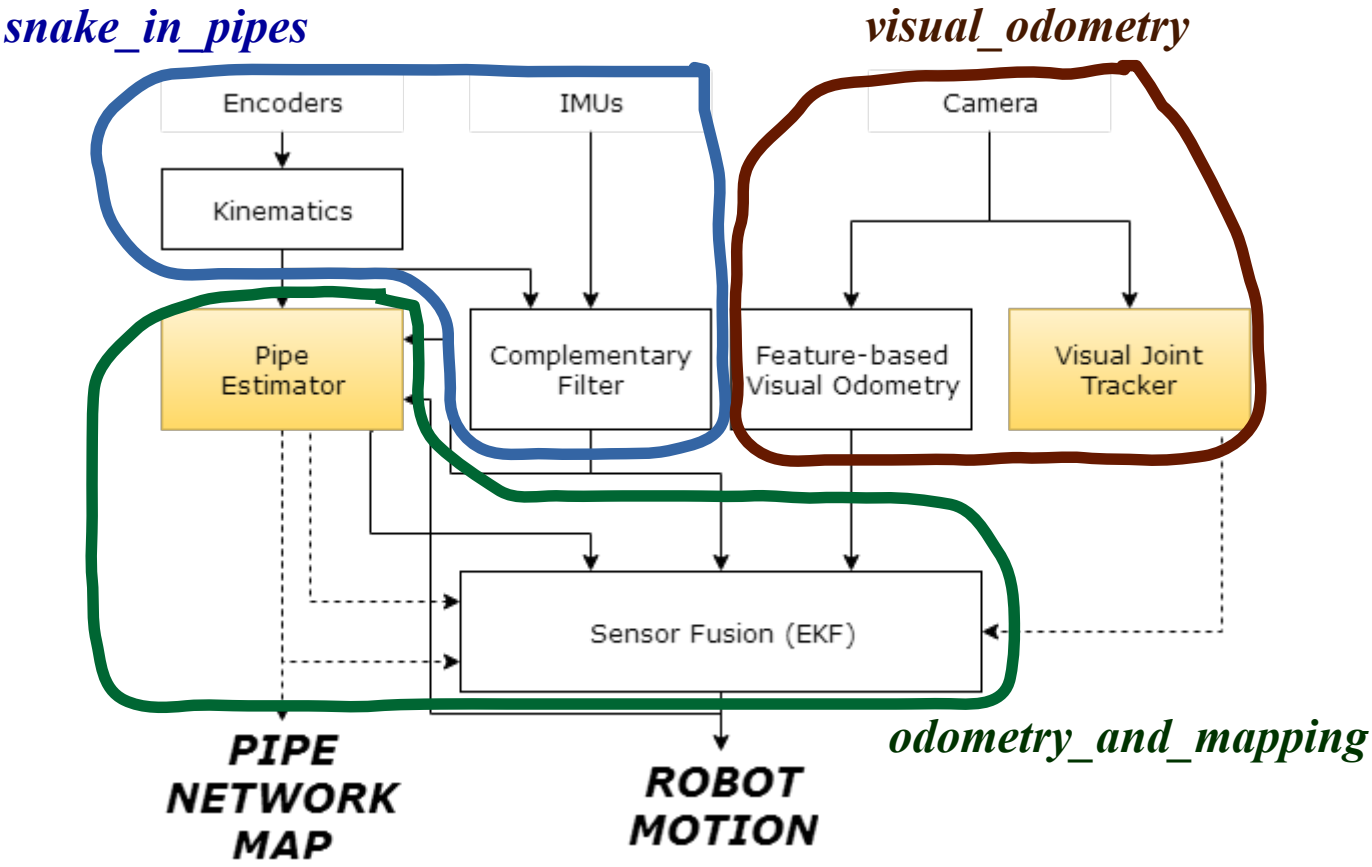
First thing to do is to run an experiment.
The experiment consists in
- making the snake crawl inside a pipe while running the Complementary Filter (use s*nake_in_pipes*)
- recording a video from the Go Pro attached to its head.

The software s*nake_in_pipes* returns an experiment log in the form of a matlab structure.

Second thing to do, is to use the Go Pro video to run the visual estimators (feature-based visual

odometry and visual joint tracker). In order to do that, first undistort the video using the Go Pro Studio software and trim it so that it is synchronized with the experiment log. Then feed it to *visual_odometry.*
It returns two text files with the estimates of the visual estimators.

Finally, feed the experiment log and the text files from visual estimators to *odometry_and_mapping*
It will return the motion estimate and the map estimate.



3 Recording data

Set the Go Pro options as: resolution: 1080, fps: 30, fov: wide
Mount the Go Pro on the head of the snake. I usually stick it to it with bi-sided tape to the last module cap. The correct orientation of the Go Pro is checked by the code later.
Tape the flashlight on its lower side, i.e. the one opposite to the red button.
[If you mount it on the sides of the Go Pro, the flashlight shows up in the video and you cannot run visual odometry]

Open `snake_in_pipes_main.m` and change the Options section to your preference.
Then run the code. The code will ask you to go through some initial checks about the snake hardware. If the checks fail, it will suggest how to fix them. Fix them and then re-run the code, it will guide you through every step to make the snake crawl inside the pipe.

3.1 Options

As a first thing, you have to decide whether to rectify the head of the snake. Normally, when the snake uses the rolling gate to crawl inside pipe, the head is not pointing forward. The performance of the visual odometry is very enhanced if you can make the head point forward and approximately aligned with the pipe main axis.

Secondly, you should specify which accelerometers and gyros do you trust. The gyros and accelerometers of some modules are unreliable, so you wanna set their trustability to 0. In order to find out which one are trustable, you'll have to go through the initial checks (see section CHECKS).

Then, specify the name of any of the modules which are currently mounted on the snake. (Type *HebiLookup* to find out).

Finally, specify the parameters of the rolling gait according to the pipe you wanna crawl into.
I used an helix with radius 0.03 and pitch 0.06 to roll inside pipes of 0.1 meters in diameter. The bigger radius is because you need to push against the walls of the pipe to locomote fforward, so you can change it according to how much friction you have inside pipe walls.
I usually used a gait speed of 0.2 or 0.1.

Then, in order to rectify the head, we load the function describing the gait of the last three modules. The file is specific for the case of an helix with p=0.06 and r = 0.05. If you wanna change the helix shape you should generate a new file (I did a dense search on the snake nominal gait) or use an on line inverse-kinematics-based rectification (see section FUTURE WORK).

3.2. Initialization

Matlab connects to the module group containing the module whose name you choose in the options. Then it sets the gains for the wanted control strategy.
Finally, it loads the offsets of the accelerometers and gyros of the modules which are currently on the snake, which are saved in the files `retrieveAccelOffsets` and `retrieveGyroOffsets`. If the offsets for the current modules are not contained in those files, you will get a warning for each of the unknown-offset-modules. If you wanna compute the unknown offsets, use `compute_gyro_offsets` or compute accelerometer offsets (see section RELATED FILES).

3.3 Checks

First, you are asked to check if the motors are fine, since modules with broken actuators don't assume the commanded position (angle).
It commands the snake to be straight, so you should see the snake go straight. If it is slightly twisted may be because of friction with ground, so make sure that the snake is almost straight and it can move kind of freely.

Then, you are asked to check if the accelerometers are giving good readings. Since the snake is still, accelerometers should be reading gravity.
You will see a visualization of the snake with the accelerometer reading shown as a vector. Ideally, all vectors are black and of the same length and orientation (vertical). The modules which you set as un-trustable in the OPTIONS section are colored in blue/purple.

You should roll the snake on itself. If some modules show strange readings, i.e. too short, too long or twisted vectors, you should kill the program and set their accelerometer trustability to 0 in the Options section. See Illustration 1.

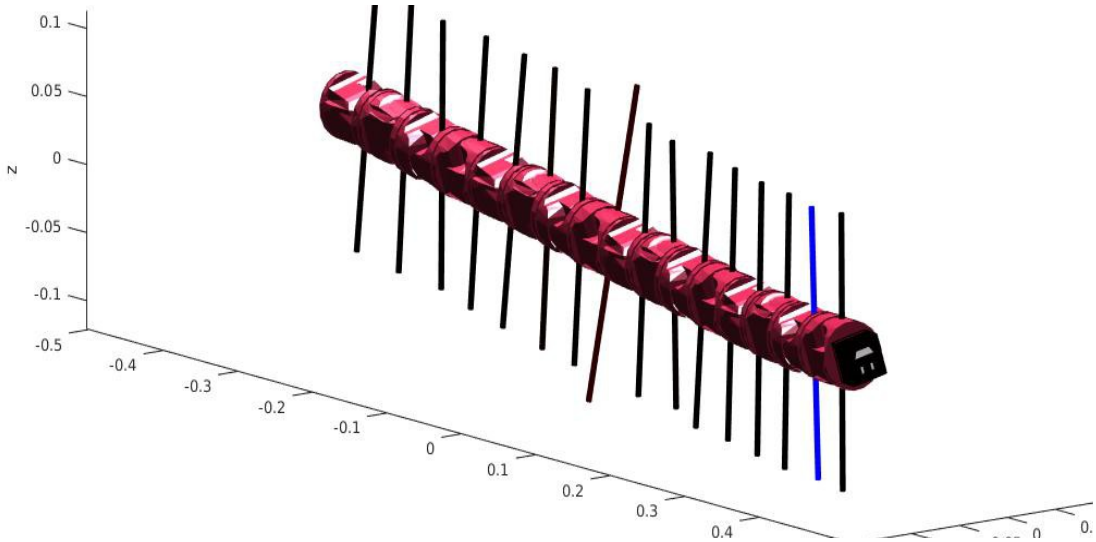


Illustration 1: This is not a good case. The gravity vector estimated by accelerometer number 8 is wrong (twisted and longer), its trustability should be set to 0 (and then it will show up as blue). Also, the accelerometer of module 2 seems to behave well, so its trustability can be set to 1

Then you should check that the Go Pro agrees with the smiley face on the visualization of the go pro in the snake. The go pro red button is supposed to be the Go Pro head top, i.e. the flashlight is below the chin.

Then you should check that the gyro reading are reasonable.
Leave the snake still on the floor. The gyros should read null angular velocity.
Two figure will show up.
One shows how much does the estimated gyro offsets are different with respect to the loaded one. If the difference is beyond 10% you get a warning and you should recompute the problematic offsets using `compute_gyro_offsets` (see section RELATED FILES). See Illustration 2.

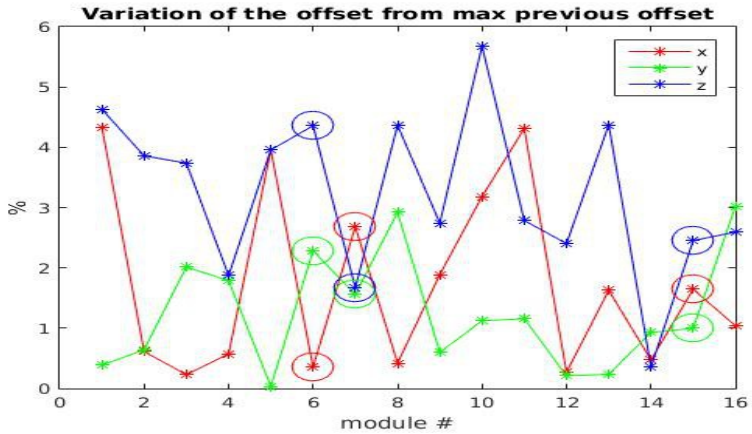


Illustration 2: This is a good case, because all offset variation percentage (on the y axis) is below 10% for each of the modules (on the x axis)

The other figure shows the mean and the standard deviation of the gyro-readings once they are compensated for the loaded offsets. Ideally, the means are 0 and the standard deviations are below 0.01 rad/sec. If the std is beyond that threshold you get a warning, because it means that the gyro reading of those modules are very noisy. You should kill the program and set their gyro testability to 0 in the Options section. See Illustration 3.

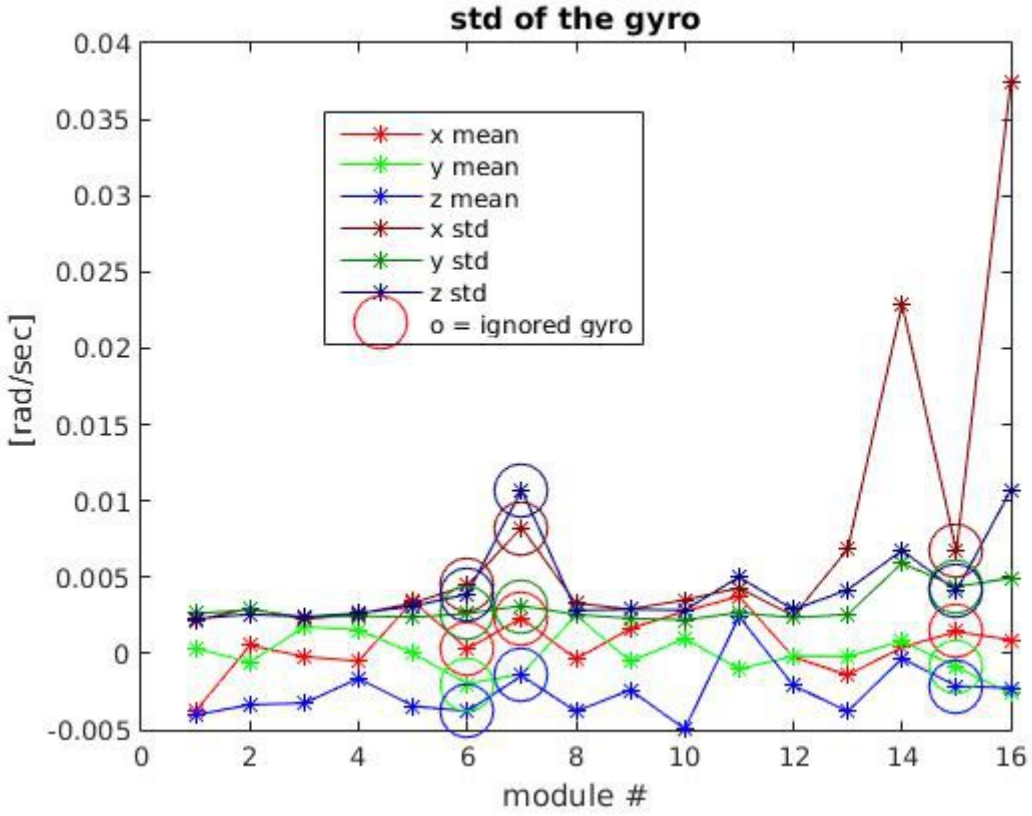


Illustration 3: This is not a good case. The mean of the gyro readings are around zeros, so they are OK (light red, green and blue). But the standard deviation of the readings is high for the modules 7, 14 and 16. While module 7 is appropriately been labeled as non-trustable (you can see it because its values are circled), 14 and 16 trustablity should be set to 0 too. You probably also wanna set the trustability of 6 and 15 to 1, because they look pretty good

Note that while the compensation for the offset in the accelerometers is fairly irrelevant, using bad gyro readings will compromise heavily the performance of the estimate from the Complementary Filter.

Then you are reminded to turn on the Go Pro when it is facing up, i.e. when the red button is facing upwards. This is crucial because the internal Go Pro software flips the video in case the video recording is started when the Go Pro orientation is downwards-ish. The flipped video is incompatible with the rest of the algorithm.

Finally, you are reminded to turn on the flashlight and to insert the snake inside the pipe.

3.4 Complementary Filter Initialization

The Complementary Filter is started. It starts updating the pose estimate of the snake.
The Complementary Filter in principle does not need changes. If for some reasons it gives bas estimates, I would suggest playing with the following parameters in ComplementaryFilter.updateFilter:

- `accelWeight`, which is a number from 0 to 1: if 0 you don't trust the accelerometers at all, with 1 you don't trust gyros at all except in the angle around the gravity vector (since accelerometers provide no estimate about it). I used 0.5
- `accelThresh`, [m/s.^2] threshold for the trustability of current accelerometer readings. In case the average reading from the accelerometers differs from gravity magnitude (9.81 m/s.^2) more than the threshold, accelerometers information is discarded.

3.5 Gait Initialization

It loads the rolling gait parameters and command the snake (slowly) into an helical shape.

3.6 Run

At every loop, the snake position is commanded, the Complementary Filter estimate is updated and the current snake state is saved in the log.

The log contains the current time, the angles of the modules and head pose according to the Complementary Filter, as well as others data which are not really used nowhere else but just-in-case: commanded angles, accelerometer readings, gyro readings, the pose of VC frame and of every module frame, and the magnitude of the main axis of inertia of the body.

The snake starts moving after two seconds (may do a bit of a jerky motion when it starts moving depending on which phase of the helix you are commanding).

Once the snake has finished locomoting inside the pipe network you should abort the program with Control+C.

NOW PLEASE REMEMBER TO SAVE THE LOG OF THE RUN.
I used something like `save('run_log_20160719_1037', 'run_log')` , meaning the run was done at 10:37 of 07/19/2016.

4. Running visual odometry

4.1 Process the video

Take the video from the Go Pro and import it into the Go Pro Studio software. Make sure that “Advanced settings/ Remove fish-eye” is selected and then convert the video.

Since my Ubuntu can not the read the video files which were the output of the Go Pro Studio, I used to import the video in Windows Movie Maker and save it with these customized options: frame size 1920x1080, speed 8000 kps and fps: 29,97.

You should also trim the video so that it is synchronized with the run log. Approximately figure out from the video when the snake starts moving and crop the video 2 seconds before that moment (as seen in 3.1.5, the snake starts moving 2 seconds after starting logging data).

It does not need to be super precise, we will check the synchronization of the video and the log in a later stage.

4.2 Run visual odometry

Open `visual_odometry.cpp`.
In main() you should specify the path of the Go Pro file, select the frame step and the time of the video when you would like to start processing it.

The frame step defines how many frames are skipped between two processed frames and should be chosen according to the speed of the gait.
The reason for not using every frame is that the camera has moved very little between consecutive frames, hence the Feature-Based Visual Odometry performance is impaired. Also the Visual Joint Odometry is impaired because noise(shadow)-generated circle are more likely to be tracked as pipe-joint circles. I chose the frame step empirically to 5.

I used 2 seconds as a starting time because that's when the snake starts moving.
The relative scaling of the translation does not work if you start Feature-based visual odometry with the robot being still!

For every (non-skipped) frame, the Feature-Based Visual Odometry (`appearanceOdometry`) and the Visual Joint Tracker (`circleOdometry`) are updated. Their estimates are written in comma-separated text files `circle_odometry.csv` and `appearance_odometry.csv`
For every frame the following data are written on the files

- time of the frame [milliseconds]
- estimated translation: dx, dy, dz [-]
- uncertainty of the translation estimates: sigma_dx, sigma_dy, sigma_dz [-]
- estimated differential rotation in Euler angles: droll, dpitch, dyaw [rad]
- uncertainty of the rotation estimates: sigma_droll, sigma_dpitch, sigma_dyaw [-]
- estimated differential rotation, in the form of rotation matrix: R(1,1), R(1,2), R(1,3), R(2,1), R(2,2), R(2,3), R(3,1), R(3,2), R(3,3) [-]
- boolean to say if the estimate is available at all [0/1]
- estimated joint position: x,y, z [m]

4.3 Details on visual odometry

The visual odometry runs two independent estimators. The Feature-Based Visual Odometry (`appearanceOdometryEstimator`) and the Visual joint Tracker (`circleOdometryEstimator`). They are briefly explained in the following (see Master Thesis for details on the theory behind it).

4.3.1 Appearance Odometry Estimator

In `appearanceOdometryEstimator.cpp` there is the code of the implementation.
At every iteration:

- Feature Extraction

Then salient visual features are extracted.
To do that, I first mask out of the frame the darker regions, so that features are not extracted in the far away non illuminated part of the pipe. I did this because often times the region between shadow and illumination generated many feature which then could not be matched in the next frame due to illumination changes.
Then I search for features in all subregions of the frame with OpenCV **`goodFeaturesToTrack`**. I

start searching for features at a given `MIN_FEATURE_QUALITY`, then if I don't find enough good features I lower the threshold till I find enough features or the threshold is too low. I did this because a evenly distribution of the features over the frame improves very much the motion estimation quality, so I wanna encourage extraction of features in every part of the frame, even if their global quality would not be enough for them to be extracted if I was searching in the whole frame at the same time. I previously tried to do the same with `MIN_FEATURE_QUALITY` but it didn't have such a useful impact. Note that extracting feature from the whole image at once is much faster. If you have on line implementation concerns, you can try just running **`goodFeaturesToTrack`** on the whole frame.

- Feature matching

If it is the first iteration, there are no features from previous iteration.
Otherwise, I match the features from previous frame in current frame. Again, I mask out darkest regions. Then I run OpenCV **`calcOpticalFlowPyrLK`** I played with Termination Criteria, winSize and maxLevel and chose for the current values. I use feature with smaller eigenvalues rather than Harris corner because performance was better. I prune out of the matched features the ones whose matching error is bigger than 1.5 the median error of the matches in this iteration for robustness.

- Essential matrix

I simply use OpenCV **`findEssentialMat`** with RANSAC and prune out the outliers
Note that this uses the Camera Matrix of the Go Pro, which I am not 100% sure about. Actually I am very confident about the focal length because I did extensive checks, but the principal point could be computed more carefully. Hence, a possible easy improvement could come from performing again the calibration of the Go Pro. Note that the Camera Matrix I use was computed from wide-angle undistorted frames, if you change the settings of the Go Pro it will heavily change.

- Relative pose estimate

I simply use OpenCV **`recoverPose.`** Same reasoning as Essential Matrix, it could benefit of recomputing the Go Pro principal point.

- Triangulate features

If there are feature matches, assumed that camera has moved of the motion estimated by previous step and triangulate 2D-points with **`triangulatePoints`**. Feature pruning according to chierality check. I also compute mean re projection error.

- Scale the translation

If there are at least two couples of 3D points from previous and current iteration, compute the relative scaling of the translation by comparing the distance of the 3D points (see Scaramuzza, Tutorial to Visual Odometry).

- Propagate the features

4.3.2 Circle Odometry Estimator

The implementation code is in `circleOdometryEstimator.cpp` and `circleTracker.cpp`
At every iteration:

- Image processing and Hough circle detection

I use a combination of Gaussian blurring and Laplacian filter to transform the image and make the Hough circle detector much more effective. The idea comes from reference
All the parameters were tested very carefully. In case the pipes will look very different from the

white PVC pipes which I used, they will need to be re-tuned. You can use the manually annotated circles (see section RELATED FILES) in order to find the optimal parameters (`sharpeningWeight`, `laplaccianScale`, `houghResolution`, `cannyThreshold`, `accumulatorThreshold`).
In case the speed of the algorithm is a concern, limiting the range between minradius and maxradius in `HoughCircles` is of high effectiveness. Also, one other parameter was very influent check

- Circle tracker

The circle tracker is made of several independent tracks of the circle status, defined as [c dot(c)], with c = [x_cc y_cc r], with (x_cc, y_cc) the coordinates of the circle center and r its radius [pixels]. Dot(c) is the derivative of c.
The tracks are implemented as Probabilistic Data Association Filters.
At every iteration, a new filter is initialized with the current detection of the Hough circle detector and every existing filter is updated. Tracks are erased if they are too similar to a better track. A track is better than another if it is the currently active one, or if it is supported and the other is not, or if it has a lower number of missed validation, in this order of importance.
The threshold for the Battachayara distance among tracks should be tuned according to how much the camera wanders around during the gait. I think it may use a better tuning also with the current setup. Same holds for the covariance matrices of the PDAFs.

- Pinhole camera model

Use pinhole camera model and knowledge of real radius of the pipe to compute distance and position of the joint in the camera frame

- Pinhole camera model uncertainty

Use uncertainty propagation rules to convert the uncertainty on the position of the circle to retrieve the uncertainty on the position of the joint

- Pose estimate

If the previous position of the joint is known from previous iteration, compute the differential motion. Actually, this is not used, because I realized that the position of the joint should be more heavily filtered before using it. In the future, a better tuning of the PDAF may allow the joint position estimate to be reliable enough to output of the estimator the differential motion of the camera.
Hence, the only used output of the circle odometry is the position of the joint in the camera frame.

General comments. I am not sure that the current implementation of the pipe joint tracker is the optimal one.

5 Running Sensor Fusion

Open `odometry_and_mapping_in_pipes_main.m`
You should specify the names of the previously generated files `run_log.m`, `circle_odometry.csv` and `appearance_odometry.csv` and have them somewhere in your MATLAB path.

Firstly, the estimates are transformed into measurement to be fed to the kalman filter. Here, the Pipe Estimator is created too, since it relies on other estimates.

Then, the Extended Kalman Filter is run.

The output of the Extended Kalman Filter contains the pose of the camera for every point in time, as well as the pose of the pipe where the head is located over time.

5.1 Details on estimates to measurements

In `estimates_to_measurements.m` a few plots are created. Please check how they look like for the sample data which are provided with the code as a reference for future experiments.
Among the others, you should check that the head and tail orientation are properly filtered to estimate the bend angles, since this depends on the speed of the gait. See Illustration 4 as an example of good filter.

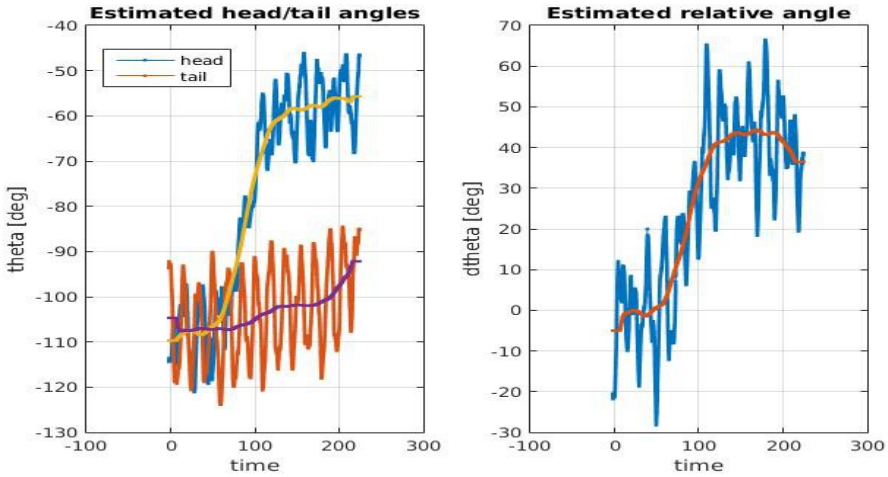


Illustration 4: Example of good filtering inside pipe estimator

You should also check that the motion of the joint in the camera frame is properly filtered too. Again, this depends on the snake speed, and the goal is that it looks like in Illustration 5.

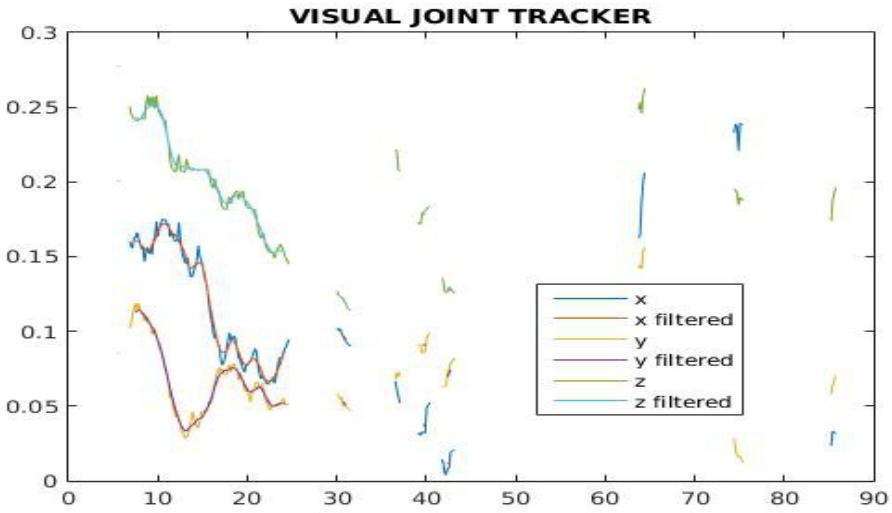


Illustration 5: Example of good joint motion filtering. This picture also shows that the joint was successfully tracked by the Visual Joint Tracker only between 5 and 25 seconds of the experiment. A better tuning of the PDAFs (see Section 4.3.2.) would improve it

Finally, check that the video and the run log were correctly synchronized by comparing the Complementary Filter and Feature-based visual odometry estimates. The rotation estimate is usually pretty good so it is easy to spot if there is a delay between the two. See Illustration 6 for an example in which the two estimates (blue line and red line) are nicely synchronized.

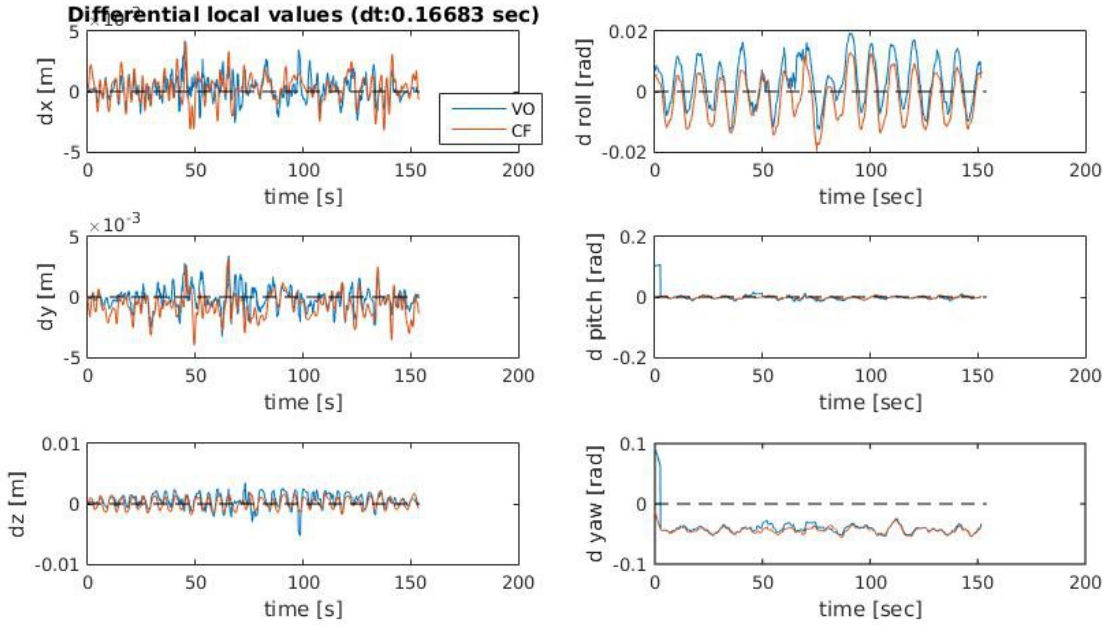


Illustration 6: Example of good video and run log synchronization: the Complementary Filter and the Visual Odometry estimates are not delayed with respect to each other

5.3. Details on EKF

The equations are shown in the Thesis.
The measurements are considered independent, while they are very not. This can be easily taken in `define_covariances`, where trustability of different estimates can be tuned accordingly to the changes to the previous steps of the work.

At the end of the EKF the odometry and the reconstructed map are shown.

6. Future work

Undistortion of the video
Since I am running the odometry off line, I now use the Go Pro Studio software to undistort the video (which is heavily affected by the fish-eye lens). This cannot be used for on line implementation. Hence you should spend some time finding the distortion parameters. I tried before but my results where worse than the ones from the Go Pro Studio. Nevertheless, I believe that spending more time on it may lead to good results.

Rectification of the head
The current implementation of the rectification of the head is very specific for the dimension of the used pipes. A smart and pretty easy improvement would come from solving the inverse kinematics problem on line, i.e. at every loop while the snake is moving, choose the angles of the last three or more modules such that the final module is near the center of the helix and pointing forward.

Dynamically change frame step
In current implementation, I update the video-based odometry at a fixed time frequency. The performance would be improved if you checked at every frame if the camera has moved significantly enough to perform good visual odometry (use distance of triangulated points). Also,

this would allow the snake to stop and start during the run without impairing the Visual Joint tracker (which would consider shadow-generated circles as pipe joint-generated circles,, since they are the same at every frame).

On line implementation
The only reason why I don't process the video while the snake is crawling inside the pipe is that the Go Pro does not stream its video to my computer. Theoretically, there is nothing against it.
The *visual_odometry* code may use some optimization to become faster. A critical lengthy step is the Hough circle detection, whose timing heavily depends on the parameters. Tune them according to the trade off between accuracy and speed of detection you need.

Dependency of EKF measurments on time and other measurements
]The EKF relies on several simplifications. Maybe tackling them will make the EKF perform better, but I am not sure. But at least it will be more theoretically solid.
Firstly, the Pipe Estimator measurement is considered independent from the Complementary Filter Estimate and from the current state estimate itself, while it is not true. It will be painful to find the proper relationship and linearize it, because heavy time-filtering and strange signals manipulations happen inside the Pipe Estimator block.
Secondly, the signals of the other blocks are filtered over time before being fed as measurements in the EKF. This in theory is not allowed, but including all the past measurements inside the EKF to do the time-filtering while running the EKF was too heavy of a solution. Maybe this violation is not too important to deal with.

Indirect EKF
I used a direct EKF and I subtract quaternions to compute the residual between predicted and measured orientations. This is not a theoretically sound approach, even if TUM suggests it in its tutorial to attitude estimation (http://campar.in.tum.de/Chair/KalmanFilter). The most common approach when dealing with quaternions and Kalman Filters is to use an indirect approach, where the orientation mismatch is approximated with a 3D angle (pretty well explained in http://www-users.cs.umn.edu/~trawny/Publications/Quaternions_3D.pdf). Many visual-inertial odometry approaches use it. I failed in implementing it because it was too computationally heavy without some theoretically complex simplifications. I am not sure if the performance of the filter would improve, but it would be more theoretically solid.

Re computation of Go Pro camera matrix
I think the current value used as the principal point of the Go Pro is a little off. The visual odometry part would very benefit from a more accurate estimate.

7. Related work

Compute gyro offsets
The file `compute_gyro_offsets` is provided to compute the offsets in the gyro readings of the modules currently connected to the snake.
It just averages the gyro readings and shows some statistics about them. Copy the output in `retrieveGyroOffsets`

Compute accelerometer offsets
The accelerometer on the SEA snake are quite cheap and they show constant-over-time axes-dependent offsets. This did not prove to be super influent in the performance of the Complementary Filter, but it is still expected to improve it.
In order to check them, slowly roll the snake in place while recording of plotting the accelerometer

readings. Ideally, the readings of the x and y axis accelerometers should go from -g to g in one full turn. Note down the offset in the curve and add it in `retrieveAccelOffsets`. See Illustration 7

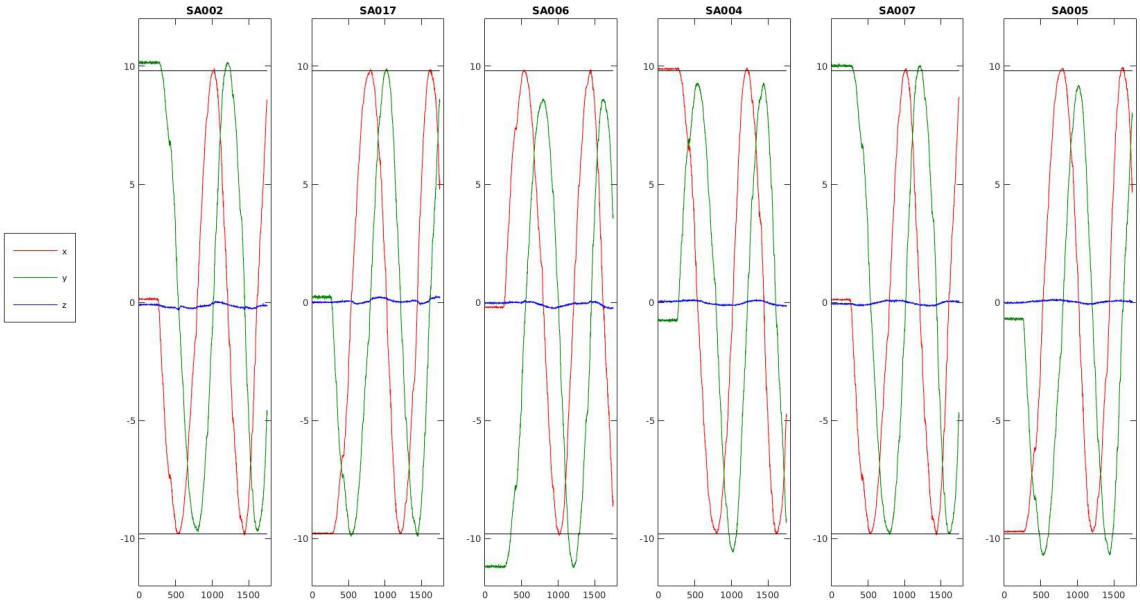


Illustration 7: Sample reading from the accelerometers when slowling rolling the straight snake around its main axis. The module SA006 and SA005 have a clearly visible offsets in the y-axis of their accelerometers

Circle annotator

The files `circleAnnotator.cpp/.h` are provided with the visual odometry software.
It provides the functionality of annotating reference circles of the pipe joints on the frames, which can be useful for testing the performance of the Visual Joint Tracker and to tune the hough circle detector parameters. See next paragraph.
To use it, open `circleAnnotator.cpp`, change the name of **`main`**() into something else and rename **`add_reference_circles`**`()` as **`main`**().
You will be guided trough a procedure for annotating reference circles (take a look to both the console and the popping up figure).

Camera calibration

I used the files made available by http://docs.opencv.org/3.1.0/d4/d94/tutorial_camera_calibration.html#gsc.tab=0.