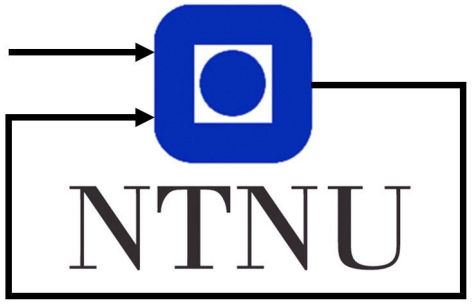
Embedded system for maze-mapping



Author:
Marcus Steffensen Vormdal

Supervisor:
Tor Onshus

Specialization project - TTK4550
Department of Engineering Cybernetics
Norwegian University of Science and Technology

January 19, 2023

Problem description

In relation to the overarching project ”co-operating robots” efforts are being made to integrate an Unmanned Aerial Vehicle (UAV) into the existing network of ground-based robots. The intended purpose of the UAV is to gather, process and distribute information about a maze that is to be mapped by the full network of unmanned vehicles. This requires the development of several different modules, such as drone control, position estimation, communication and mapping. This project thesis aims to implement the latter. The final product is intended to be a functioning embedded system for the mapping of a maze, using components fit as a payload for an UAV. The system should take into consideration the limiting factor of weight, as well as the corresponding computational limitations. If possible the system should integrate a nRF52840 System-on-Chip (SoC) from Nordic Semiconductors. This due to much of the previous work on the overarching project having been implemented specifically with this component in mind.

Moments of the thesis:

- Determine if the nRF52840 is capable of image processing
- Choose hardware to use in the embedded system
- Implement or repurpose a image based line segment finder for use in mapping
- Implement or repurpose a mapping algorithm
- Implement or repurpose a communication protocol for use between components
- Merge these modules in to an embedded system
- Test how the system performs in a controlled environment

Acknowledgements

I would like to thank my supervisor Tor Onshus for guiding me through the process of writing this project thesis as well as giving valuable feedback. Thanks to Simon Lexau for the helpful suggestions regarding academic writing.

Abstract

This project thesis describes the implementation and testing of an embedded system for mapping a maze, intended to be the payload in a UAV. It was undertaken in an effort to reduce the weight and computational requirements of already developed solutions, as well as integrate the System-On-Chip nRF58240.

Hardware

The hardware chosen for the system was the openMV CAM M7 camera module and the nRF52840, as well as a wired SPI-connection between the boards. The former was chosen due to its existing optimization of image processing, which was not trivial to recreate on the nRF52840. It was found that although it should theoretically be possible for the nRF52840 to handle light image processing, the effort might be better spent by working on hardware meant for the task.

Mapping

Extraction of maze-edges was achieved by an already implemented line segment extractor in conjunction with a merging and matching algorithm. The results showed that while the lines accurately represented the maze some artefacts remained and differences in lighting could impact the results.

Global coordinates

The length of each line segment in real world coordinates were calculated using the GSD as described in Bjoernsen [2017]. The calculated lengths consistently overshot the real-world lengths and had to be accounted for. After tuning for the GSD error the image coordinates were transformed to an arbitrary global coordinate system using a homogeneous transformation matrix. Results with different camera angles and maze positions were accurate, with the largest error during testing coming in at 3.5 %.

Embedded system

To integrate the nRF52840 a SPI-framework was developed, which did the intended job of transferring relevant information between the components. A final dynamic test was performed to ensure that the modules worked together over consecutive images. The results showed that it was possible to match lines between images taken at different orientation and that the corresponding error between these lines were in the single digits.

Sammendrag

Denne prosjektoppgaven beskriver implementasjonen og testingen av et innvevd system for kartleggingen av en labyrint, tiltenkt som nyttelasten i en UAV. Prosjektet ble gjennomført for å prøve å redusere vekten og prosesseringskraften mot tidligere løsninger, samt innlemme nRF52840, et system på en integrert krets.

Komponenter

Komponentene valgt for systemet var kamera-modulen openMV CAM M7 og nRF52840, koblet sammen gjennom en SPI-tilkobling. Førstnevnte ble valgt på grunn av den allerede eksisterende optimaliseringen av bildeprosessering, noe som ikke var trivielt å gjenskape på nRF52840. Teoretisk sett burde nRF52840 kunne prosessere bilder på egenhånd, men det ble vurdert til at innsatsen i oppgaven heller burde rettes mot komponenter ment for oppgaven.

Kartlegging

Uthenting av labyrint-kanter ble utført av en allerede implementert linjestykke-detektor i samarbeid med en sammenlignings og sammenslåings-algoritme. Resultatene viste at linjene presist representerte labyrinten, men at noen uforutsette linjer også kom med. Endringer i lysforhold påvirket også resultatene.

Globale koordinater

Lengden på hver linje i reelle koordinater ble utregnet basert på GSD-metoden beskrevet i Bjoernsen [2017]. De utregnede linjene var konsistent for lange og måtte bli tatt høyde for. Etter å ha skalert for feilen ble linjene transformer til globale koordinater via en homogen transformasjons-matrise. Resultatene med forskjellige posisjoner og vinkler var presise, med den største observerte feilen på 3.5 %.

Innvevd system

For å integrere nRF52840 ble det implementert et SPI-rammeverk. Dette hadde ansvaret for å kommunisere relevant informasjon mellom komponentene. En siste dynamisk test ble gjennomført for å teste at modulene jobbet sammen over et sett av bilder. Resultatene viste at det var mulig å pare linjer mellom forskjellige bilder tatt ved forskjellige vinkler og at feilen mellom dem var relativt liten.

Table of Contents

List of Tables

List of Figures

Abbreviations

Abbreviation	Description
SPI	Serial Peripheral Interface
M7	openMV CAM M7
nRF52	nRF52840 DK
MISO	Master-In-Slave-Out
MOSI	Master-Out-Slave-In
SCK	Clock signal
CS	Chip Select
GPIO	General Purpose input/output
VS Code	Visual Studio code
IDE	Integrated Development Environment
RPC	Remote Procedure Calls
SoC	Socket-on-Chip
UAV	Unmanned Aerial Vehicle
LSD	Line Segment Detector
RAM	Random access Memory
CPU	Central Processing Unit
GSD	Ground Sample Distance

1

Introduction

1.1 Background

The background for this project thesis is the academic project ”Co-operating robots”, supervised by Tor Onshus. The system has originally consisted of ground based robots mapping a maze and a central server to receive, process and distribute data. In recent years there has also been an intention to integrate a Unmanned Aerial Vehicle (UAV) into the mesh. An UAV can maneuver above the maze and capture image data, which after processing can be distributed to network giving valuable data about the structure of the maze. This also opens up several new avenues for better estimates, such as merging sensor data through sensor fusion or refining the position estimate of all the units in the mesh. Several projects have already laid the foundation for the implementation of an UAV to map a maze. The two most relevant to this project thesis is the line segment mapping implemented in Bjoernsen [2017] and the ultrasound-based position estimator implemented in Bjerke [2022]. The focus in both was on a modular level and explored the viability of the respective methods as a standalone solution. Both the position estimate system and the implemented line segment extractor showed that an implementation with these methods as foundation had merit. A natural next step in the overarching project is therefore to shift the focus from viability to integration of the modules and the required hardware. The latter is important as the size, mass and computational requirements of such a system should be as small as possible. UAVs have limited carrying capacity, and fully loading them can have severe impacts on the flight duration and control performance. Several of the projects that have been done have been based on the nRF52840 System-on-Chip (SoC) and its bluetooth capabilities (e.g. Blom [2020], Gilje [2022]). An additional aspect under consideration is

therefore if the nRF52840 can take on some role in the final product, which would ease integration in the overarching system.

1.2 Report structure

A brief description of each chapter is presented below:

- Chapter 2: Hardware specification
  Describes the hardware used in the system, the technical specifications of each component as well as some of the main features.
- Chapter 3: Line segment detection
  Describes the theory needed to detect line segments in an image, the setup for testing of the implemented solution and corresponding results.
- Chapter 4: Global coordinates
  Describes the theory needed to convert line segments from local to global coordinates, the setup for testing of the implemented solution and corresponding results.
- Chapter 5: Serial Peripheral Interface
  Presents the Serial Peripheral Interface protocol, the setup for testing of the implemented solution and corresponding results.
- Chapter 6: Embedded system
  Describes the planned embedded system, the setup for testing of the implemented solution and corresponding results.
- Chapter 7: Discussion
  Discussion about the performance of the system in light of the test results.
- Chapter 8: Conclusion
  A summary of the most important results and the performance of the system.
- Chapter 9: Further work
  Suggested ways of improving on or expanding the work done in this project thesis.
- Appendix A:
  Describes how to set up the environments to develop for the openmv CAM M7 and the nRF52840 DK. From the TTK8 project Vormdal [2022]
- Appendix B:
  Presents the files and functions in the software implementation.

2

Hardware specification

2.1 OpenMV CAM M7

Description

The openMV module is an integrated solution for machine vision on an embedded system where the main processing unit is a microcontroller (OpenMV [b]). The main components are the image sensor OV7725 and the microcontroller STM32F765VI.

Specification

- CPU: 216 MHz
- RAM: 512 KB
- Flash: 2 MB
- SPI transfer rate: 54 Mbps
- Max resolution: 640x480
- Voltage supply: 3.3 V
- Weight: 16 g

Features

The M7 features a rich out-of-the-box library of common image processing operations, optimized to work on the processing limited hardware. Development can be done in a

custom IDE using a embedded version of python built on C called micropython, with the open-source code of the camera module also available if it is necessary to do changes in the base functionality. Enables communication with SPI, UART, I2C and USB. A micro-SD slot enables the system to store data while operating.

2.2 nRF52840 DK

Description

The nRF52840 DK is a development kit for the nRF52840 System-On-Chip (SoC) produced by Nordic semiconductor (Nordic [b]). The chip itself is a low-cost and lightweight, and comes with support for bluetooth, while the DK enables easy access to the different pins and easy flashing of the hardware.

Specification

- CPU: 64 MHZ
- RAM: 256 kB
- Flash: 1 MB
- SPI transfer rate: 32 MHz
- Voltage supply: 3.3 V

Features

Since the nRF52840 DK is a general purpose development kit there are a multitude of features. The ones relevant for this project is the SPI compatibility, the bluetooth tranceiver as well as the software stack that includes basic examples of code implementation.

3

nRF52840 as an image processor

Before implementing the embedded system the choice of components had to be evaluated. As the nRF52840 already was an integral part of previous work it was the first to be considered for the role of image processing. This chapter presents some of the theory and considerations that laid the groundwork for pivoting away from this choice and rather choosing the openMV CAM M7.

3.1 Theory

Image size

Digital images are built up of discrete pixel elements. For each pixel there is associated data that sets the color according to a color model. One of the most common models, RGB (Red, Green, Blue), contains data that represents the intensity of each color. By mixing the colors it is possible to recreate many of the naturally occurring colors, and using arrays of pixels it is possible to represent images with discrete values which can be used by computers (Miano [2000], pg. 1-5). Two of the main ways a computer program can access and do operations on image data is to either load it in Random Access Memory (RAM) or read/write directly to a storage medium such as flash memory. Due to the significantly faster transfer speed between RAM and CPU the first method is preferred, as the second leads to bottlenecks in processing speeds. The minimal size requirement of the image in memory can be described by Equation 3.1:

$$KB = \frac{width * height * bit\_depth}{8 * 1024} \tag{3.1}$$

where *width* and *height* is the width and height of the image array in pixels and *bit_depth* is the amount of bits used to represent the color of one pixel. Due to the normally limited hardware on embedded systems it can be necessary to minimize the size. The equation gives three main ways to do this. *width* and *height* can be reduced by choosing a worse resolution or by cropping the image. *bit_depth* can be selected depending on what the intended purpose of the image is. The smallest bit depth, 1 bit monochrome, only represent pixels as black or white. This presents several challenges in image processing. In addition to requiring an image sensor that can provide this format, it also makes techniques that requires derivatives unusable, as the pixel gradient will be the unit step function for transitions between black and white pixels. 2-bit and 4-bit also exists, but much information is still lost. Only at 8-bit and above it becomes hard to discern the difference for the human eye, and in scientific applications this is often used as a minimum (pho [2021]). Unless it is critical the minimum bit depth should therefore be at this level at a minimum. For 8 bit one can choose between grayscale with higher resolution, or color where each channel occupies a subset of the total bitdepth (3,3,2 for 8-bit RGB).

3.2 nRF52840 viability

Image size

Based on the theory in section 3.1 it is possible to calculate some base requirements needed for image processing. Assuming conventional image resolution standards (QVGA at 320x240, VGA at 640x480) and a minimum of 8-bit depth, the storage space needed can be calculated as follows:

$$QVGA = \frac{320 * 240 * 8}{8 * 1024} = 75KB \tag{3.2}$$

$$VGA = \frac{640 * 480 * 8}{8 * 1024} = 300KB \tag{3.3}$$

From section 2.2 it can be seen that the nRF52840 has 256KB of RAM. This means that a full VGA image cannot be loaded or processed without some workaround. The next step down in conventional standards is QVGA, which is small enough to fit in RAM. However, the lower resolution the lower the accuracy of the results. Depending on the application this can become a severe source of error. For the system implemented in this thesis it is necessary to calculate real life distances based on a relationship between the resolution, height and sensor intrinsic. As the resolution decreases the distance each pixel represents increases. This has two implications. The first is that the error even with a ”pixel-perfect”

match to a real world line is upper limited to half of this distance. The second is that for every pixel that the line segment is off by adds this distance as error. The base error of QVGA is double that of VGA, which again is double that of HD (1280x720).

Processing

Another challenge that presents itself if trying to use the nRF52840 for image processing is the computational demand. Extracting information from an image is usually an iterative process that runs over every pixel, often more than once. The amount of pixels also scales in a quadratic fashion with the resolution. The nRF52840 is neither optimized for this type of processing or intended for the task. If implementing these types of algorithms it is necessary to account for the limited hardware. If not handled correctly the result could be that the nRF52840 is overloaded and stuck on processing the image, rather than doing other scheduled tasks. The openMV CAM M7, designed with the exclusive purpose of processing images and optimized to run on the hardware, barely manages to run some of its more complex examples in real time. It should be expected that this is a best case scenario for the nRF52840. How critical this is depends on what other tasks the SoC is intended for. If it can be dedicated to image processing without any timing constraint or other responsibilities then the only thing that needs to be ensured is that the processing actually finishes. If controlling any critical component the robustness required might demand that the image processing is reduced in complexity to such a degree that the results suffer.

Software overhead

In addition to memory and processing limitations some challenges related to implementing software have to be considered. The first is that although the QVGA standard fits on the nRF52840, several of the procedures used in image processing usually requires memory overhead. When calculating the gradient of an image this information has to be stored somewhere, meaning it is necessary to cast another array of equal size to the image. Furthermore one might want to store extracted data, further filling the memory. Considering that other tasks might also need RAM this might mean that even at QVGA running out of memory is a real possibility that needs to be accounted for. All of these are problems that can be worked around, but require either destructive behaviour (overwriting) or more complex solutions, e.g. working on parts of the image at the time, merging results as one goes.

3.3 Viability conclusion

Based on the findings in this chapter it should theoretically be possible to implement the necessary image processing algorithms to extract line segments on the nRF52840. The implementation would however require limitations that might have severe impact on the performance, both with regards to computing and accuracy of extracted line segments. The nRF52840 should probably also be dedicated to the task, as the demanding computation might lock out other scheduled tasks in periods. This would be critical if these tasks related to the control of the UAV. Furthermore the overhead in development could also end up being substantial to work around the limitations. It was therefore decided that the nRF52840 would not take the role as the image processor in this project thesis. The openMV CAM M7 was chosen instead, as solutions to many of the challenges described in this section were already implemented out of the box.

4

Line segment detection

4.1 Theory

Computer vision is a wide field with many applications, from detecting simple features such as a corner to recreating the structure of an object in 3D. As the processing power of computers have increased, the viability of extracting relevant information from an image has increased correspondingly. In recent years much research and development has been done to improve performance and find new use-cases which can solve real-life problems. One of the most basic operations in computer vision is to extract simple features such as edges and lines in an image. These are also some of the fundamental building blocks for more advanced computer vision methods. This chapter presents the underlying theory needed to implement the line segment extractor used in this thesis. It also presents the algorithms implemented and the resulting accuracy of implementation.

Line segment extraction

After a picture is taken the first processing step that needs to be done is to extract line segments. Several methods for line extraction exists, e.g. by using an edge extractor such as Canny Edge and a Hough transformation such as described in Bjoernsen [2017]. In this project the method used was the Line Segment Detector (LSD), as this was already implemented and optimized out-of-the-box in the openMV CAM M7. In the case where the LSD only processes a section of the image at the time a line segment might be split into parts. Further processing is needed to identify these cases and reconnect them. This can be done in two steps; Matching line segments that should be merged and then merging these

lines. This process is shown in Figure 4.2. Since the result of merging two lines will be a line with altered characteristics (e.g. an angle reflecting that there is a slight discrepancy between the two lines to be merged), new matches might be found by running the process several times.

Line Segment Detector - LSD

This section gives an introduction to the Line Segment Detector as presented in Morel et al. [2012]. The Line Segment Detector algorithm (LSD) takes a a grayscale image as input and returns line segments corresponding to locally straight contours (edges) in the image. The LSD can be broken down in to three main steps; gradient computation to build a *level-line field*, building *line support regions* and finally *rectangle fitting*. The process is vizualied in Figure 4.1

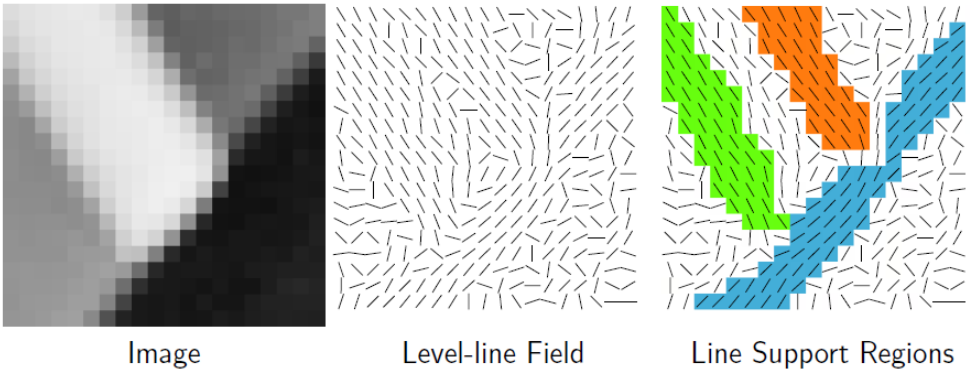


Figure 4.1: Grayscale image converted to Level-line Field and the corresponding Line Support Regions. From Morel et al. [2012], Figure 2

The gradient computation is done by applying the mask Table 4.1 to each pixel and calculating the gradient with Equation 4.1 and 4.2

i(x,y)	i(x+1,y)
i(x,y+1)	i(x+1,y+1)

Table 4.1: Gradient mask applied to each pixel.
i(x,y) corresponds to the pixel being considered

$$g_x(x,y) = \frac{i(x+1,y) + i(x+1,y+1) - i(x,y) - i(x,y+1)}{2} \tag{4.1}$$

$$g_y(x,y) = \frac{i(x,y+1) + i(x+1,y+1) - i(x,y) - i(x+1,y)}{2} \tag{4.2}$$

With the gradient computed it is possible to compute the angle (level-line angle) and magnitude of the gradient. This is given by Equation 4.3 and 4.4. The results are stored in the level-line field.

$$g_{angle} = arctan\left(\frac{g_x(x,y)}{-g_y(x,y)}\right) \tag{4.3}$$

$$G(x,y) = \sqrt{{g_x}^2(x,y) + {g_y}^2(x,y)} \tag{4.4}$$

When the line level field is complete the pixels are sorted in bins representing discrete intervals of gradient magnitude. Starting at the bin with largest magnitude the next step, *Region growing*, takes the pixels as seed values. The region is grown by comparing the seed pixel to the adjacent pixels in the 8-connected neighborhood. The adjacent pixels are first compared to a magnitude threshold, rejecting low-magnitude pixels. The gradient line angle is then compared to the seed gradient angle and the difference is again compared to a threshold. If within threshold the pixel is added to the line support region, the region angle is calculated according to 4.5 and the search continues by comparing the adjacent pixels to the one added. As pixels are added they are considered ”used” and are removed from the bins. The search runs until no more pixels can be added.

$$region\_angle = arctan\left(\frac{\Sigma_j sin(level\_line\_angle_j)}{\Sigma_j cos(level\_line\_angle_j)}\right) \tag{4.5}$$

After the line support regions are found the next step of the LSD is to build rectangles out of the line-support regions. This is done by interpreting a region as a solid in 2 dimensions, setting the gradient magnitude as point mass for each pixel. It is then possible to calculate the ”centre of mass” of the region and the angle as the first inertia axis of the solid. The width and length of the rectangle corresponds to the smallest values fully covering the region. This fully represents each line segment found. As a final step the rectangles are validated by calculating several statistical metrics and improved by varying some of the rectangles configurations.

Line segment matching

The LSD makes it possible to extract full line segments from an image. In a scenario where the LSD is applied on parts of an image at the time line segments might end up being split apart. During implementation it became clear that the LSD had to be applied in this manner

to not run out of Random Access Memory (RAM). To work around this a matching and merging algorithm was required. Algorithm 1 presents the overarching algorithm to match and merge all lines found by the LSD in an iterative fashion. It runs through the list of line segments, matching and merging lines within thresholds. It dynamically removes merged segments from the working lists. Returns when no more matches can be found.

Algorithm 1 Line merging algorithm

```
found_matches ← True
lines ← LSD(image)
merged_lines ← []
while found_matches == True do
    org_size ← len(lines)
    for i in range(len(lines)) do
        for j in range(i+1, len(lines)) do
            match, vertical ← is_match(lines[i], lines[j])
            if match == True then
                merged_line ← merge(vertical, lines[i], lines[j])
                del lines[j]
            end if
        end for
        merged_lines.append(merged_line)
    end for
    lines ← merged_lines
    if len(merged_lines == org_size) then
        found_matches ← False
    end if
end while
Return merged_lines
```

Line segment matching

To determine if any lines are split it is necessary to match segments that share similar properties. One way to do this is by comparing properties of each segment and then extrapolating the expected placement of a matching segment. The matching and merging process is shown in Figure 4.2 and presented as pseudo-code in Algorithm 2. Step one is to compare the angles of two lines relative to some common reference point. This is shown as the angles θ_1 and θ_2 in **Step 1** of Figure 4.2. If these angles are within a threshold the potential match transitions to the next step. If not the process terminates and tries with a new pair of lines. Step 2 is split in two parts depending on the orientation, as the extrapolated slope tends towards infinity when the lines are close to vertical. The pseudo-code for normal lines is presented in Algorithm 4 and for vertical lines in Algorithm 3.

Algorithm 2 Matching algorithm

$end\_points \leftarrow (None, None)$
$match \leftarrow False$
$\theta_{diff} \leftarrow abs(L1_\theta - L2_\theta)$
$L1_{x-disp} \leftarrow abs(L1_{x_2} - L1_{x_1})$
$L2_{x-disp} \leftarrow abs(L2_{x_2} - L2_{x_1})$
if $\theta_{diff} < \theta_{thres}$ **then**
  if $L1_{x-disp} > X_{thres}$ and $L2_{x-disp} > X_{thres}$ **then**
    $match \leftarrow regular\_match(L1, L2)$
    if $match = True$ **then**
      $break$
    end if
  else if $L1_{x-disp} < X_{thres}$ and $L2_{x-disp} < X_{thres}$ **then**
    $match, end\_points \leftarrow vertical\_match(L1, L2)$
  end if
end if
Return *match, end_points*

For lines not close to vertical the matching the procedure is shown in **Normal step 2** of Figure 4.2 and is done as follows: Calculate the x and y-displacement between end point of L1 and start point L2. If both are within a threshold calculate the slope of the leftmost line (L1). Extrapolate the y-coordinate when the x-value of the end point corresponds with the start point of the candidate line being considered (L2 or L3). Create an outer limit by adding a user-defined threshold above and below the extrapolated y-coordinate. If the y-coordinate of the start point of the considered line is within bounds a match is considered found. In the presented example of Figure 4.2 L2 will be considered a match while L3 is rejected.

Algorithm 3 Normal matching algorithm

$match \leftarrow False$
$x_{dist} \leftarrow abs(L2_{x_2} - L1_{x_1})$
$y_{dist} \leftarrow abs(L2_{y_2} - L1_{y_1})$
if $x_{dist} < x_{thres}$ and $y_{dist} < y_{thres}$ **then**
  slope = $(L1_{y2} - L1_{y1})/(L1_{x2} - L1_{x1})$
  $upper_b \leftarrow L1_{y2} + (slope * x_{dist}) + u_{bound})$
  $lower_b \leftarrow L1_{y2} + (slope * x_{dist}) - l_{bound})$
  if $L2_{y1} < upper_b$ and $L1_{y2} > lower_b$ **then**
    $match \leftarrow True$
  end if
end if
Return *match*

Vertical step 2 of Figure 4.2 is done in the following manner: The first step for matching vertical lines is to ensure that both considered lines are sufficiently vertical. This is done by calculating the absolute displacement in the x-axis of both lines and rejecting matches outside a user-defined threshold. The line-pairs within the limits can then be matched by calculating if two points of the lines are close enough spatially to be considered on top of each other. This gives four distinct scenarios corresponding to the four permutations of orientations, shown in the left column of Table 4.2. The distances are calculated and again compared to a user-defined threshold, which results in a reject or a match.

Algorithm 4 Vertical matching algorithm

$end\_points \leftarrow (0, 0)$
$line\_len \leftarrow 0$
$match \leftarrow False$
$p_1 \leftarrow [abs(L2_{x_1} - L1_{x_1}), abs(L2_{y_1} - L1_{y_1}), p1_{ext}]$
$p_2 \leftarrow [abs(L2_{x_2} - L1_{x_1}), abs(L2_{y_2} - L1_{y_1}), p2_{ext}]$
$p_3 \leftarrow [abs(L2_{x_1} - L1_{x_2}), abs(L2_{y_1} - L1_{y_2}), p3_{ext}]$
$p_4 \leftarrow [abs(L2_{x_2} - L1_{x_2}), abs(L2_{y_2} - L1_{y_2}), p4_{ext}]$
$permutations \leftarrow [p_1, p_2, p_3, p_4]$
for p in permutations **do**
  if $p[0] < x_{thr}$ **then**
    if $p[1] < y_{thr}$ **then**
      $new\_line\_len \leftarrow len(p_{ext})$
      if $new\_line\_len > line\_len$ **then**
        $end\_points \leftarrow p[2]$
        $line\_len \leftarrow new\_line\_len$
      end if
    end if
  end if
end for
Return *match, end_points*

L1 Orientation	L2 Orientation	End points
Up	Up	$(L1_S, L2_E)$
Up	Down	$(L1_S, L2_S)$
Down	Up	$(L1_E, L2_E)$
Down	Down	$(L1_E, L2_S)$

Table 4.2: Vertical line permutations and corresponding end points of merged line. Subscript S denotes the start coordinate and E the end coordinate.

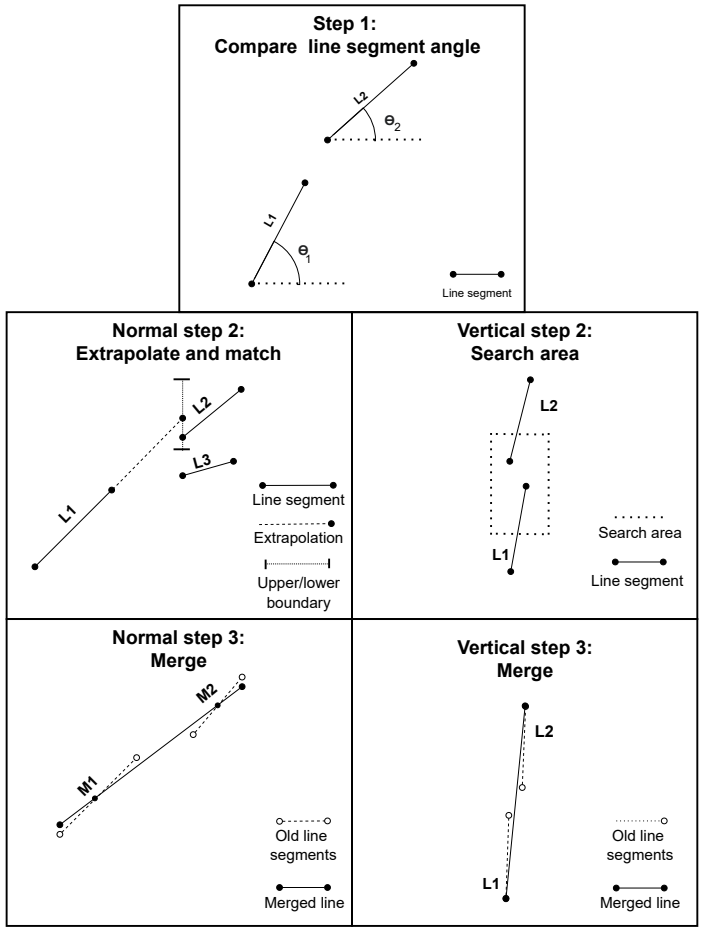


Figure 4.2: The matching and merging process described in section 4.1

Line segment merging

If the line segment matching algorithm determines that two lines are matching the next step is to merge these lines. The merging algorithm is presented in Algorithm 5. For vertical lines this is simply done by calculating which of the permutations within the thresholds that results in the longest merged line. This is possible as finding a vertical match is equivalent to finding the ”interior” points of two lines stacked vertically. The ”exterior” points can then be set as the end point of a merged line with limited loss in accuracy (as the angle difference and x-displacement of the lines are small). The end points corresponding to each permutations is show in the third row of Table 4.2 and the resulting merged line can be seen in **Vertical step 3** in Figure 4.2.

Algorithm 5 Merging algorithm

$match \leftarrow False$
$x_{dist} \leftarrow abs(L2_{x_2} - L1_{x_1})$
$y_{dist} \leftarrow abs(L2_{y_2} - L1_{y_1})$
if $vertical == True$ **then**
    $theta \leftarrow angle(p_{max})$
    $merged\_line \leftarrow (p_{max}, theta)$
end if
if $vertical == False$ **then**
    $M1 \leftarrow middle\_point(L1)$
    $M2 \leftarrow middle\_point(L2)$
    $slope \leftarrow (M2_y - M1_y)/(M2_x - M1_x)$
    $point_s \leftarrow (L1_{x1}, M1_x - (slope * diff(M1_x, L1_{x1})))$
    $point_e \leftarrow (L2_{x2}, M2_x + (slope * diff(M2_x, L2_{x2})))$
    $theta \leftarrow get\_angle(slope)$
    $merged\_line \leftarrow (point_s, point_e, theta)$
end if
Return $merged\_line$

One approach to merge lines that are not vertical is described and tested in Hussien and Sridhar [1993]. Starting with two line segments that are determined to match the process starts by calculating the middle point of each line. These two points are then considered as the central part of the merged line, and the resulting slope is calculated. The central line is then extrapolated out to the intersection of the normal from the segments endpoints. A slight adjustment to the last step was done to simplify the process, instead extrapolating endpoints to the x-coordinate of the start point of the leftmost line and end point of the rightmost. An example of the merging can be seen in **Normal step 3** of Figure 4.2.

4.2 Line segment detection test

The setup in figure Figure 4.3 was used to test the performance of the image processing algorithm that extracted, merged and calculated the real-world distance of the line segments. It consists of a camera base, a maze replica of known proportions, the openMV CAM M7 and a computer to interface the microcontroller. The test was done by placing the maze replica on the ground, attaching the M7 above in the camera base pointing downwards and then running the algorithm with the replica oriented differently. The test was done after setting tuning the camera parameters to account for the lighting conditions in the room. The test was performed to establish how accurate the line extraction were, how well the matching and merging algorithm works.

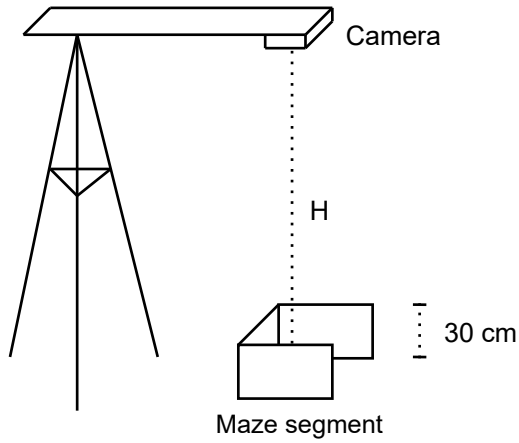
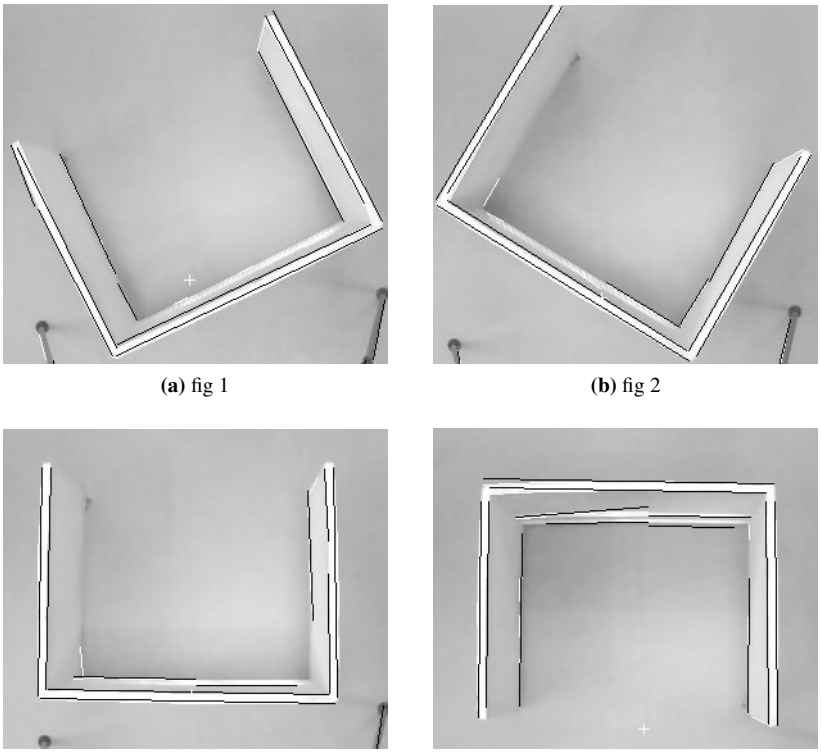


Figure 4.3: Setup for the segment detection and gsd accuracy test

4.3 Results

Figure 4.4 shows the detected edges of a maze-like structure, where black lines indicate the extracted line segments. White lines are line segments that were present before the line merging algorithm was used.



(a) fig 1 **(b)** fig 2

(c) fig 3 **(d)** fig 4

Figure 4.4: Line segments extracted in the line segment test

5

Global coordinates

5.1 Theory

Global coordinates

To convert pixel lengths and coordinates from an image in to real-life units several transformations have to be done. At the most fundamental a pixel corresponds to a specific length a given distance away. After this length is found it might be necessary to account for objects that are in front or behind this image-plane, described in 5.1. It is then possible to convert the pixel length to real-life length of a line segment, and from there derive the global coordinates using a known or estimated 3D-position of the camera.

Length extraction and mapping

Extracted line segments from the LSD are given in pixel coordinates. As the intention is to distribute the information extracted to a network of robots, the coordinates first have to be scaled to a real world measurement, then related to the other robots through a common coordinate system. The scale is found by calculating the Ground Sample Distance (GSD) (as covered in Bjoernsen [2017], pg 4-6). It relates scale between pixel values and real life distance, given a known distance from a sensor. It is based on the pinhole model, which is a simplified model of how the sensor array in a camera captures the incoming light from the surroundings. The formula used to calculate the GSD is presented in Equation 5.1, where **p** is pixel pitch, **H** is the distance between the ground plane and **f** is the focal width. The pixel pitch is calculated as the physical sensor width divided by the width of pixels in

the sensor array. This variation of the GSD assumes that the angle between the lens and the ground plane is zero.

$$GSD = \frac{p * H}{f} \tag{5.1}$$

The relation between GSD, heigth from sensor, pixel pitch and focal width can be seen on the left in Figure 5.1. The right side of the figure presents the problem of determining the GSD when the object protrudes out of the ground plane. If not accounted for the calculated real-world distance from GSD will have an error **e** proportional to the protrusion. This can be accounted for by subtracting the height of the object H_w from the total height **H**, which in practice moves the image plane to the top of the object. It does however require that H_w is known or can be estimated.

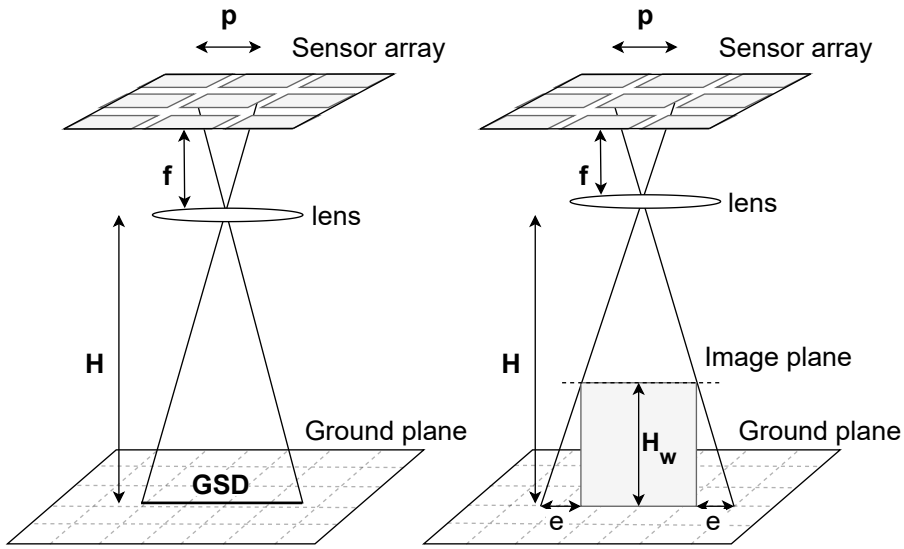


Figure 5.1: GSD relation and error propagation due to object above image plane.

Coordinate transformation

After finding the GSD the next step is to map the scaled coordinates to the global coordinate system. This can be done by using a homogeneous transformation matrix of the lie group SE(3) (Briot and Khalil [2015]), consisting of a translation and a rotation in 3-D. These can be seen in Figure 5.2, where T_1 represents the translation consisting of the

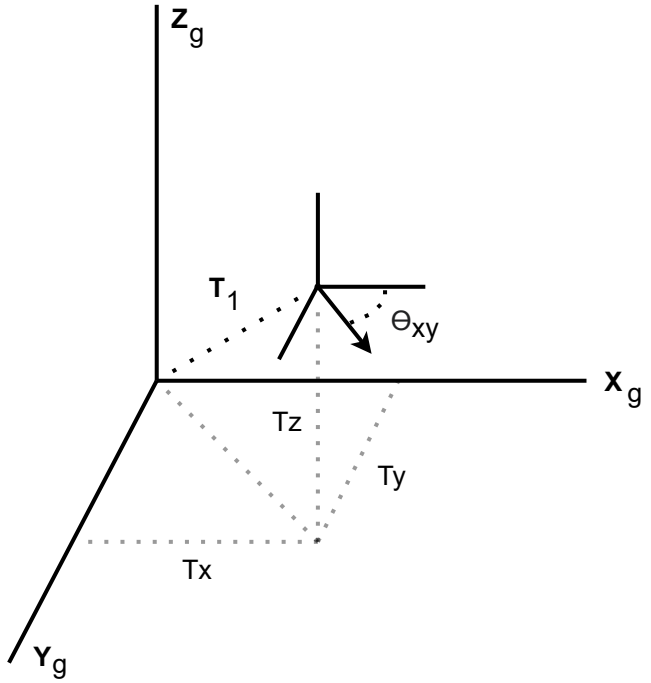


Figure 5.2: Relative translation and rotation of UAV to global coordinate system

components T_x, T_y and T_z. The rotation is shown as θ_{xy}. The transformation can be represented on matrix form by Equation 5.2:

$$SE(3) = \begin{bmatrix} R & T_1 \\ 0 & 1 \end{bmatrix} \tag{5.2}$$

where the translation matrix is given by $T_1 = \begin{bmatrix} T_x & T_y & T_z \end{bmatrix}^T$. The rotation matrix simplifies to Equation 5.3 with the assumption that the rotation around the y and x-axis is zero, which is an assumption already established when calculating GSD.

$$R = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

Mapping a coordinate from the UAV's local reference system to the global coordinate system can by done by matrix multiplication as shown in Equation 5.4

$$\begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & T_x \\ sin(\theta) & cos(\theta) & 0 & T_y \\ 0 & 0 & 1 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{loc} \\ y_{loc} \\ z_{loc} \\ 1 \end{bmatrix} = \begin{bmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \\ 1 \end{bmatrix} \tag{5.4}$$

5.2 Global coordinates test

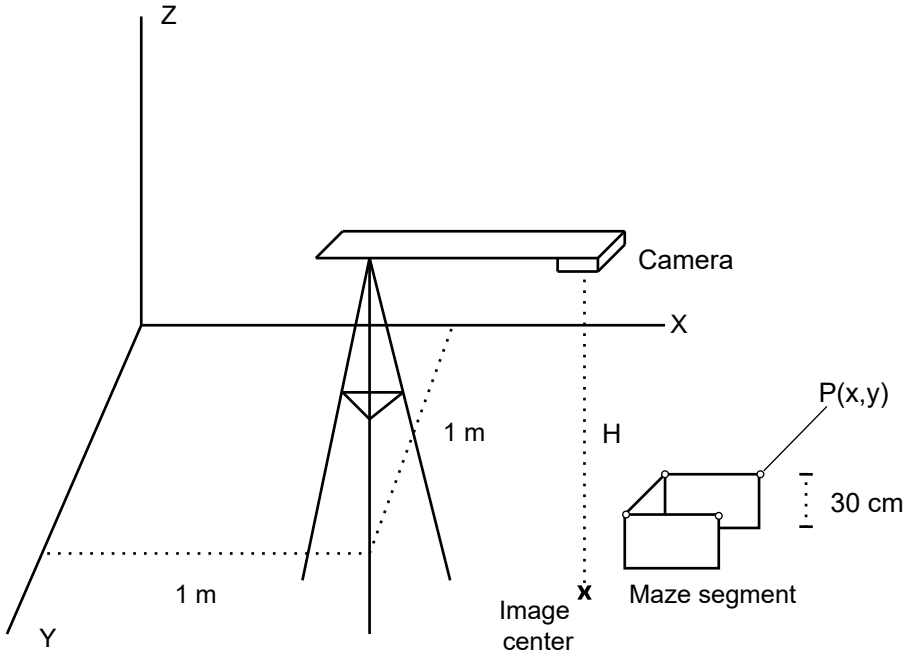


Figure 5.3: Setup for the global line segments test

The testing of the global coordinates mapping was split in two. The first test was done with line segment pixel coordinates resulting from section 4.3. The second part of the test was done to ensure that the coordinate transformation was done correctly. The setup for the global line segments test is illustrated in figure Figure 5.3 and the physical setup shown in Figure 5.4. The global coordinate system is shown in the background, with the camera base centered at the position (100, 100, 0) relative to this frame. From there the camera sensor is extended on an arm which points in the same direction as the relative angle in the xy-plane. The extension means that the global coordinates of the image sensor center will be on the form $(100+cos(\theta), 100+sin(\theta), 150)$. The maze is placed under the sensor and the global x and y coordinates of each corner is measured, as well as the angle of the back

wall to the x-axis. The points P1, P2, P3, P4 are measured in the global coordinate system and logged. The camera then captures an image, processes the line segments and runs them through the global coordinate transform. The measured and calculated coordinates are then compared.

5.3 Results

This section presents the results from the tests described in section 5.2. For the first test the results can be seen in Table 5.1 and Table 5.2. They show the comparison between calculated and real-life distance for each side of the maze segment at H = 150 cm. Table 5.3 and Table 5.4 presents the same comparison sampled at H = 120 cm.

Central	Left	Right
58.91	46.09	46.61
57.00	44.95	45.63
58.17	46.6	46.6
58.59	45.46	44.89
58.13	46.61	46.13
58.21	45.94	45.97

Table 5.1: Calculated and actual real life distance (cm) using GSD at H = 150 cm

Description	Central	Left	Right
Actual dist (cm)	53.0	42.5	42.5
Avg Diff (cm)	5.2	3.4	3.5
Avg Diff (%)	10 %	8 %	8 %

Table 5.2: Average error from true real-life distance at H = 150 cm

Central	Left	Right
57.08	43.45	45.32
57.35	42.82	44.51
56.12	43.42	45.33
54.34	44.03	46.57
55.03	43.85	45.16
55.98	43.51	45.38

Table 5.3: Calculated and actual real life distance (cm) using GSD at 120 cm

For the second test the GSD was scaled down by 8.7 % to try to account for the average error determined found in Table 5.2. An offset also had to be inserted to make sure that the camera center corresponded to the image center. All tests were done with the camera base

Description	Central	Left	Right
Actual dist (cm)	53.0	42.5	42.5
Avg Diff (cm)	3.0	1.0	2.9
Avg Diff (%)	6 %	2 %	7 %

Table 5.4: Average error from true real-life distance at H = 150 cm

stationed at the global coordinate (100,100) and the offset from to the camera being 45.5 cm. The angle was set off the y-axis and was -45, degrees, -90 degrees and -225 degrees for Table 5.5, Table 5.6 and Table 5.7 respectively. Each table presents the measured and calculated point corresponding to the points in Figure 5.4, as well as the error between them. *Real X* and *Real Y* is the measured coordinates of each point, *Clc X* and *Calc Y* is the coordinates returned from the system, *Err X* and *Err Y* is the error in percent between the real and calculated distances. Due to constraints in the measurement method (measurement tape) only one significant figure is carried throughout the results.

Point	Real X	Real Y	Calc X	Calc Y	Err X (%)	Err Y (%)
P1	32.0	129.0	29.5	127.2	2.5	1.8
P2	64.0	103.0	63.9	105.4	0.1	-2.4
P3	95.0	148.0	93.9	148.4	1.1	-0.4
P4	62.0	174.0	58.7	170.7	3.3	3.3

Table 5.5: Test of global coordinate transform at -45 degrees

Point	Real X	Real Y	Calc X	Calc Y	Err X (%)	Err Y (%)
P1	15.0	100.0	14.1	100.9	0.9	-0.9
P2	40.0	65.0	40.0	67.5	0.0	-2.5
P3	82.0	100.0	81.4	99.5	0.6	0.5
P4	55.0	130.0	55.7	131.1	-0.7	-1.1

Table 5.6: Test of global coordinate transform at -90 degrees

Point	Real X	Real Y	Calc X	Calc Y	Err X (%)	Err Y (%)
P1	107.0	69.0	107.2	70.0	-0.2	-1.0
P2	122.0	30.0	122.4	31.3	-0.4	-1.3
P3	170.0	50.0	171.9	49.0	-1.8	1.0
P4	155.0	89.0	158.5	87.5	-3.5	1.5

Table 5.7: Test of global coordinate transform at -225 degrees

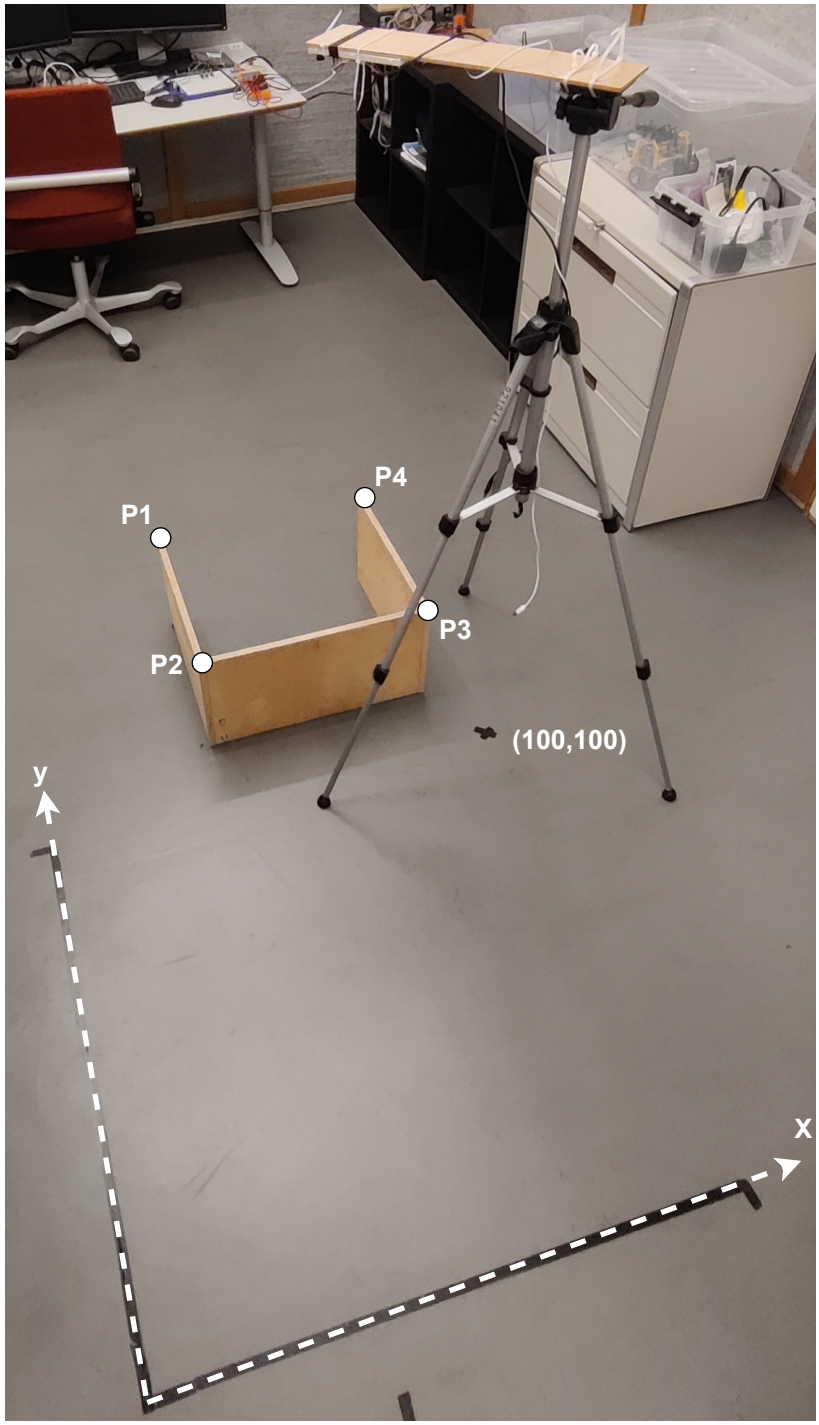


Figure 5.4: Physical setup for the global line segments test

6

Serial Peripheral Interface

6.1 Theory

The Serial Peripheral Interface (SPI) is a communication protocol that enables full duplex communication over a wired interface (Dhaker [2018]). By using a Master/Slave hierarchy one central unit (Master) can communicate with several other units (Slaves) over the same bus. Physically the SPI-connection between the units consist of 4 wires plus a common ground. The wires are labeled MISO (Master-In-Slave-Out), MOSI (Master-Out-Slave-In), CS (Chip Select) and SCLK (Clock). Figure 6.1 show the general setup for a single master-slave pair. For each slave unit one separate CS-wire has to be connected, as the master needs a way to signal that the data on the bus is for a specific slave. There are four distinct SPI-modes, which differs by what logic level that signals the start of a transaction and on which clock edge the data should be sampled. The different modes are shown in table Table 6.1

SPI-mode	CS active	Trigger edge
SPI0	Low	Falling
SPI1	Low	Rising
SPI2	High	Falling
SPI3	High	Rising

Table 6.1: SPI-mode configurations

A typical SPI-transaction based on SPI-mode 0 is shown in figure Figure 6.2. For illustrative purposes the example only shows the data being sent in half-duplex.It is however possible for master and slave to send data simultaneously on the same clock signal. The

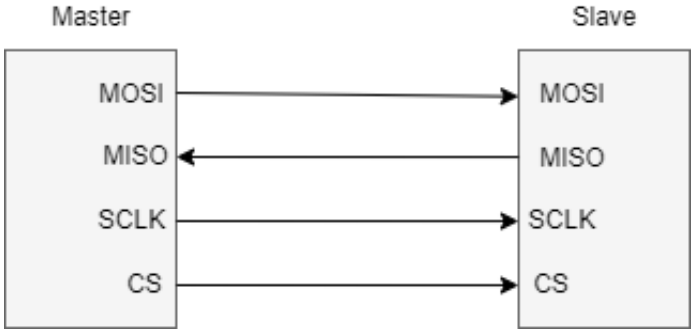


Figure 6.1: General setup for SPI master-slave pair

master unit starts a transaction with a slave by setting the CS signal low. It then feeds a clock signal on SCLK while also passing the data it wants to send on the MOSI wire. The clock signal makes it possible to synchronize the transaction through a common transfer window. The slave reads the value of the MOSI line on every rising edge in the SCLK signal (dashed lines). The first data burst is finished when the SCLK goes high. The second time the SCLK signal is fed from the master the slave sends data on the MISO line which is then sampled by the master. When all data is sent and received the master sets the CS signal high and the transaction is finished.

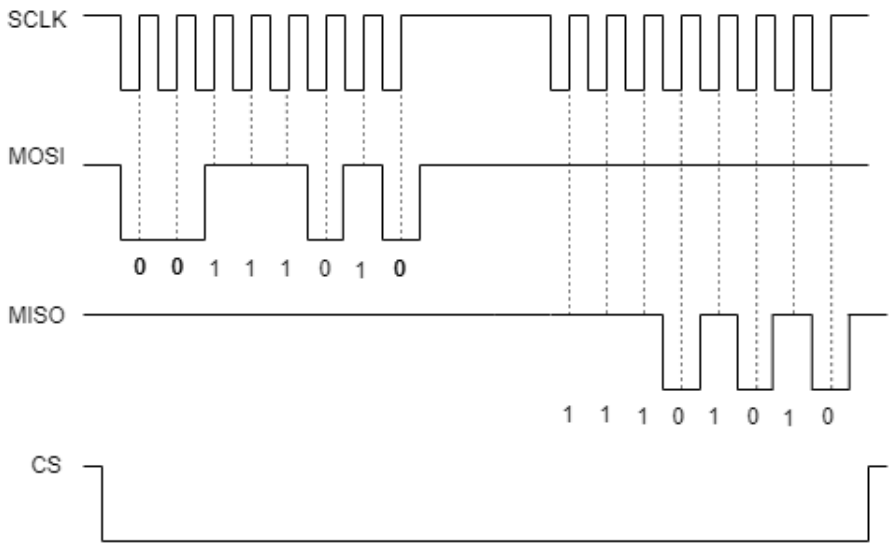


Figure 6.2: A typical SPI-transaction

Wiring diagram

The Serial Peripheral Interface requires five wires to be connected between the master and the slave. These wires are mapped to Ground, Master-In-Slave-Out (MISO), Master-Out-Slave-in (MOSI), clock wire (SCK) and Chip-Select (CS). The wiring diagram is shown in Figure 6.3 and the pin-configuration is shown in Table 6.2. In this project the nRF52 had the role as master while the M7 had the role as slave. In addition to the wiring diagram both units were connected to a computer through two USB 2.0 to microUSB cables. This was done in order to observe the terminal output for each unit.

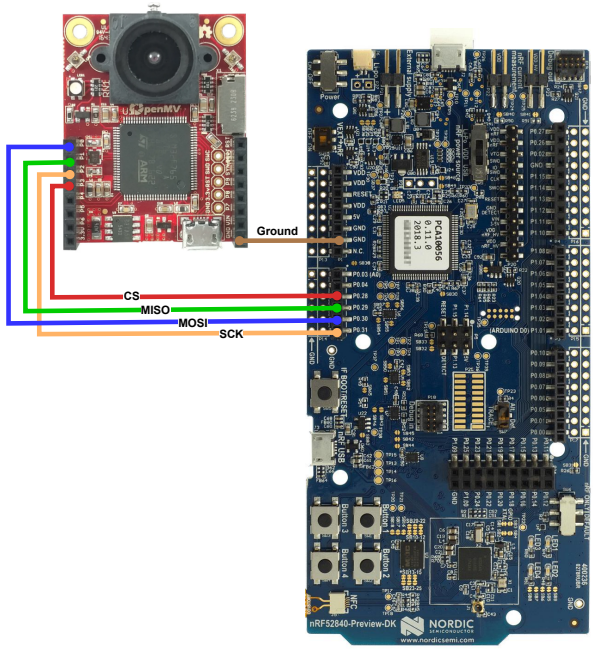


Figure 6.3: Wiring diagram for SPI-communication. Component images from Nordic [a] and OpenMV [a]

Function	M7 pin	nRF52 pin
Ground	GND	GND
MOSI	P0	P0.30
MISO	P1	P0.29
SCK	P2	P0.31
CS	P3	P0.28

Table 6.2: Pin mapping for SPI-communication

6.2 Serial Peripheral Interface test

The setup for the SPI communication test is described in section 6.1. The test was done by sending and receiving every type of command and data that would be present in the normal operation of the system. This included the CAP, REC and LIN command, as well as position data and line coordinates. When the different commands were received the respective units also had to successfully parse the information and do the corresponding action to the commands, updating its state variables in the process. This test was done to ensure that the flow of the information passes correctly and that no unit is stuck in a state which would hinder further operation. The acceptance criteria for the SPI-communication test is summarized in Table 6.3.

Specification	Acceptance criteria
1.1	nRF52 sends and receives a known message. Ensure that the information received is the same as the information sent
1.2	M7 sends and receives a known message. Ensure that the information received is the same as the information sent
1.3	Send LIN and REC command to nRF52, it flags that these messages are received. Send CAP command to M7, it flags that the command is received.
1.4	When nRF52 receives LIN, parse following data and extract correct amount of lines to be transferred. When M7 receives CAP, extract the correct position sent and start edge extraction.
1.5	The delay between each message at maximum transfer capacity should be short enough to not create undue delay.
1.6	Let master and slave send commands when the other is powered down. Make sure that both retries sending the same command until the other part has received the message.
1.7	Observe that the communication starts and that data is passed correctly when powering up both units.
1.8	Start the system and let it run continuously. The system behaviour should always be in an defined state.
1.9	Ensure that the states transitions according to the state machine presented in Figure 7.2.

Table 6.3: Acceptance criteria used for the SPI-communication test

6.3 Results

The SPI-communication test was validated using the acceptance criteria presented in Table 6.3. Criteria 1.1-1.4, 1.6 and 1.9 were fulfilled without any errors experienced during testing. Criteria 1.5 was validated to a lower threshold of 100 ms between each transaction at maximum transfer speed. Below this the behaviour of the system degraded and successful transfer of data could not be guaranteed. 1.7 was fulfilled as long as the M7 was powered up before the nRF52. In the case where the nRF52 was powered up first the M7 transitioned to the wrong state and communication broke down. As long as the correct start-up sequence was followed 1.8 was fulfilled.

7

Embedded system

7.1 Theory

Context diagram and structured analysis

Figure 7.1 shows the context diagram and structured analysis of the system. The blue rectangles in the context diagram represents the physical hardware being interfaced by the software. The software modules are represented by circles in the structured analysis. Central in the system the four lines named MISO, MOSI, CS and SCK represent the physical wires connecting the two units over the SPI-interface.

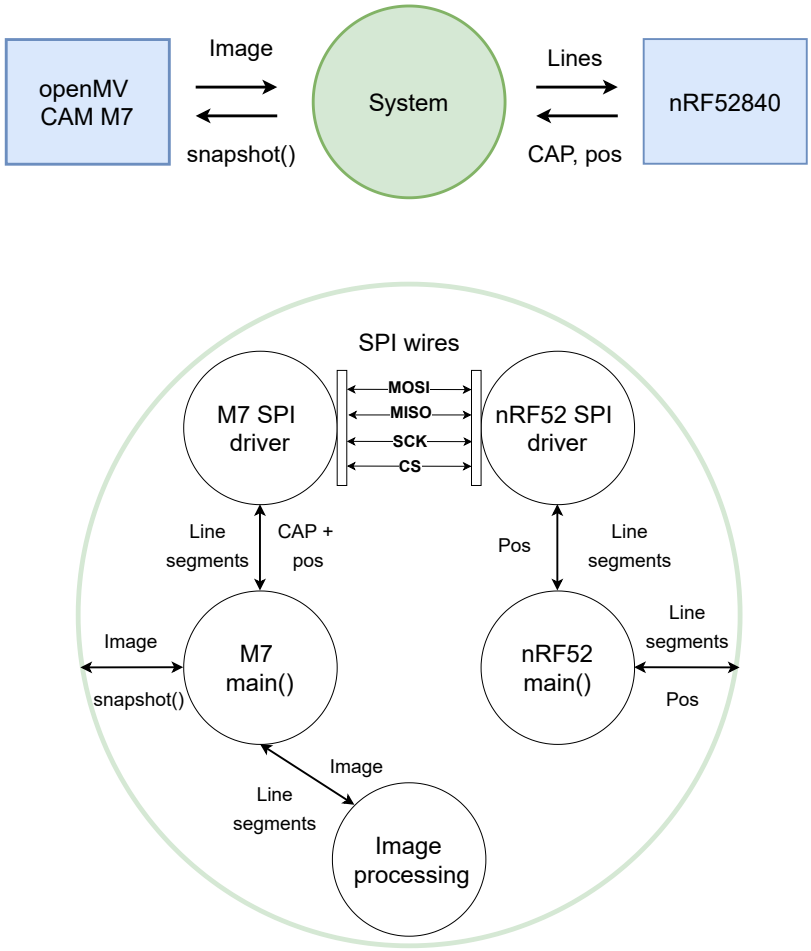


Figure 7.1: Context diagram and structured analysis of the system

Software Modules

The software design can be split up into the following modules:

- openMV CAM M7 SPI driver:
  Handles the SPI communication on the M7 side. Waits for CS to go low (ready to send), sends and receives data by interfacing with the SPI-library of the M7.
- M7 main:
  Keeps track of current operating mode, interfaces with camera to take and receive images, packs data to be sent and decodes data received. Passes and receives data from SPI-driver. Calls image processing module to get line segments.

- nRF52 SPI driver:
  Handles the SPI communication on the nRF52 side. Sets CS low/high, sends and receives data using the hardware interface, sets the SPI-mode and the clock frequency.
- nRF52 main:
  Keeps track of current operating mode, gets position estimate, packs data to be sent and decodes data received. Passes and receives data from SPI-driver.
- Image processing:
  Receives an image, finds and returns line segments.

State machine

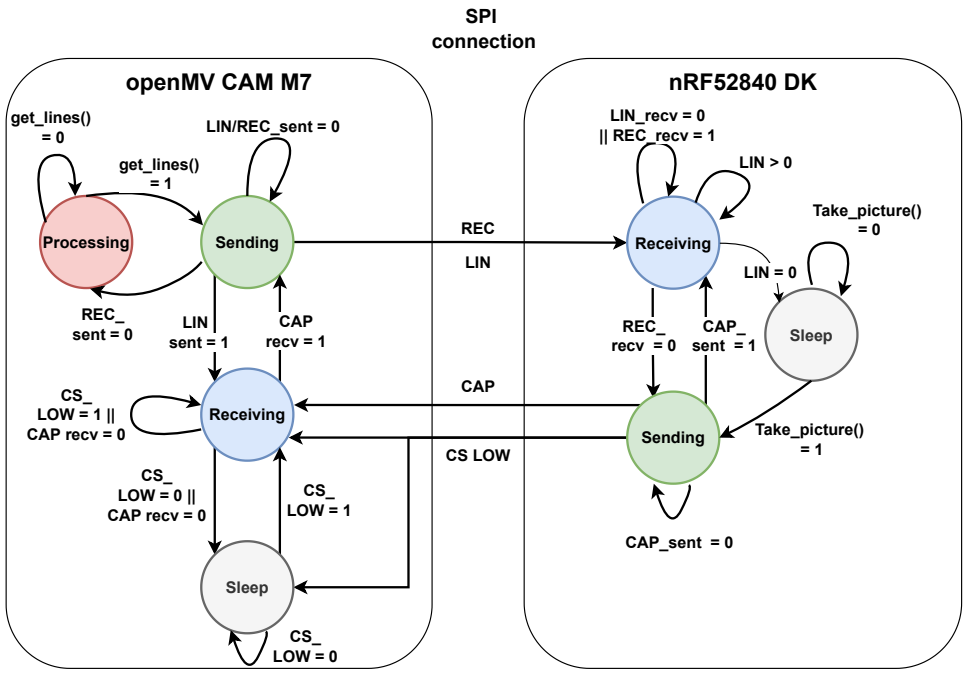


Figure 7.2: State machine

The planned states and state transitions are presented in the State Machine Diagram in Figure 7.2. The planned flow of the transactions is as follows: Both units start in the *Sleep* state. nRF52 transitions to *Sending* when in position to take image (Done manually as the optimal triggering point for image capture is outside the scope of this thesis). It sets CS low and sends a **CAP** command. nRF52 jumps between *Sending* and *Receiving*, repeating the **CAP** command until a **REC** command is received. M7 transitions to *Receiving* when

CS goes low, and stays until **CAP +** position is received. It then transitions to *Sending*, sends a **REC** command until nRF52 receives and sets CS High. M7 transitions to *Processing* where it captures an image and extracts lines. When done M7 transitions back to *Sending*, sends a **LIN** command with amount of lines found, then sending one packaged line segment at a time. nRF52 receives the **LIN** command, extracts amount of lines and decrements for each line received. When all lines are received nRF52 sets CS high and transitions to *Sleep*. With CS high M7 also transitions back to *Sleep*

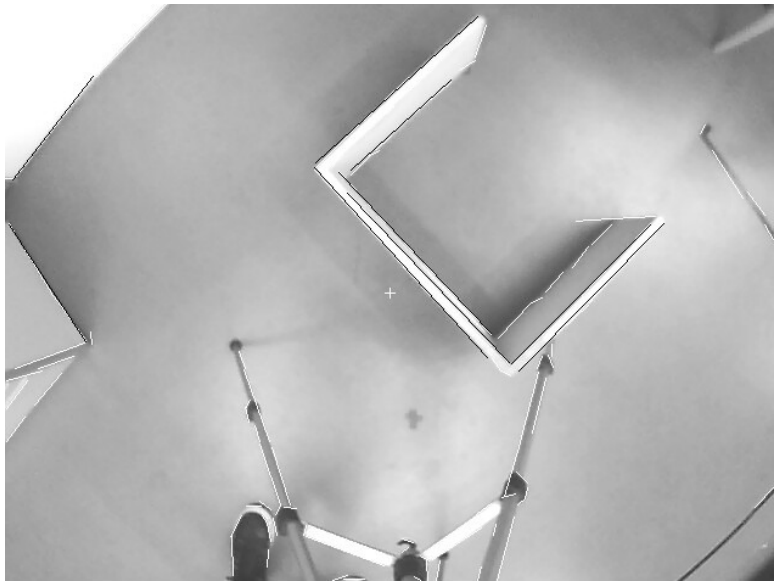
7.2 Embedded system test

The setup for the complete system test is the same as in section 5.2, with two additions. The first is that the position estimate and capture moment is controlled and sent by the nRF52840 to the M7, which then does the full processing and returns the real-world line segments over SPI. These can then finally by read in the nRF52 terminal. The second is that this test was done ”dynamically”, where the camera was rotated with a pre-determined angle after an image was captured. A new image was then captured at the new pose (Activated by the nRF52 sending a new CAP command) and the global coordinates again calculated. This makes it possible to see how well the extracted line segments match between images during continuous operation. This test was done to ensure that the different modules were integrated properly, and that the information flow from capture commands and position to returning line segments can be followed.

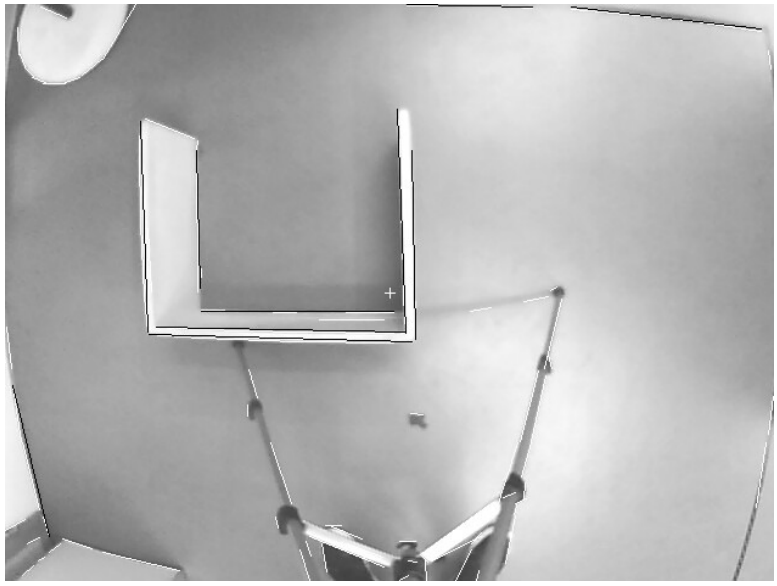
7.3 Results

This section presents the results from the test described in section 7.2. The test was done at a starting angle of 225 degrees left of the y-axis with the camera base centered at (100,100). The rotation between images was manually done and was specified to be 45 degrees to the right. In addition to the GSD being scaled as in section 5.3 the end results are truncated to $\pm$ 1 cm. Figure 7.3 presents the two images captured during the tests with the line segments transferred to the nRF52 presented in black. The lines were filtered to only include long segments in an effort to reduce the number of false positives. Figure 7.4 presents the log output for the nRF52 when receiving the line segments over SPI. The left side corresponds to the image before rotation and the second after. The white boxes marks the line segments corresponding to maze edges. The unmarked lines correspond to edges from other objects or the other edge side of the wall segments. Table 7.1 and Table 7.2 presents the start and end points of the marked lines, which corresponds to the corners of the wall segment. In

addition the error between calculated coordinates and the measured distances as well as the error between images are added.



(a) Embedded system test image before rotation



(b) Embedded system test image after 45 degree rotation

Figure 7.3: Captured images for the mbedded system test

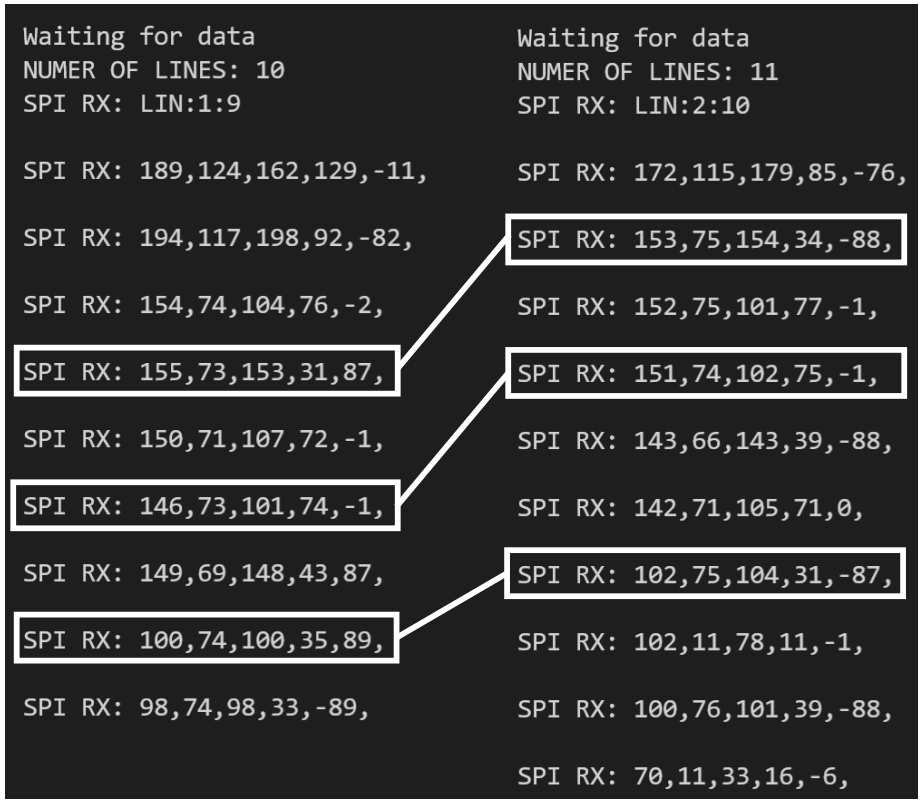


Figure 7.4: nRF52 log of received lines corresponding to the images in Figure 7.3. White boxes correspond to a wall edge.

Point	Real X	Im1-X	Im2-X	Err Im1 (%)	Err Im2 (%)	Err Im1/Im2 (%)
P1	153	153	154	0	0	-3
P2	153	155	153	0	-2	-2
P3	100	100	102	-1	0	-1
P4	100	100	104	-4	0	4

Table 7.1: Real and calculated x-coordinates of the edge points of the maze segment and corresponding error.

Point	Real X	Im1-Y	Im2-Y	Err Im1 (%)	Err Im2 (%)	Err Im1/Im2 (%)
P1	31	31	34	-3	-1	-1
P2	73	73	75	-2	0	2
P3	73	74	75	-2	-2	-2
P4	31	35	31	0	-4	-4

Table 7.2: Real and calculated y-coordinates of the edge points of the maze segment and corresponding error.

8

Discussion

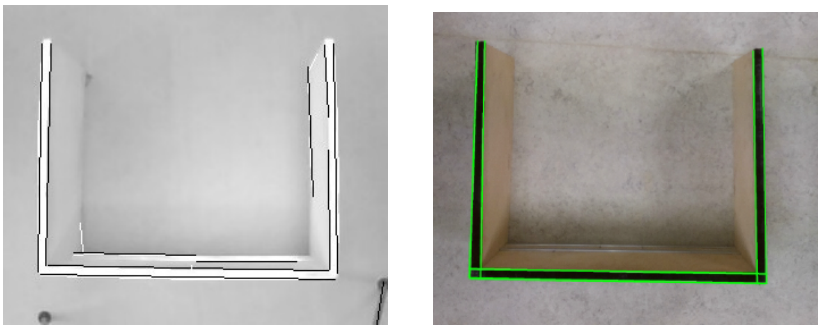
8.1 Segment detection

From the results in section 4.3 it is clear that the line extractor can extract line segments that fits well with the actual maze section being mapped. There are however several limitations that also became apparent. Perhaps the most obvious is the fact that edges at the intersection between floor and wall are detected in every test as seen in Figure 4.4. Although these are actual edges they are nonetheless not desirable to disseminate through the network because of two reasons; They are below the virtual image plane and therefore calculated with an error in GSD, and they are also offset due to the camera perspective. In real-life coordinates the edges should correspond in the xy-plane to the top of the respective wall. A way to alleviate this problem could be to mark the maze segment so that the contrast is greater than the rest of the image and weak edges could be filtered out. It does however require every maze segment to be marked in the same manner. Another error can be seen in Figure 4.4 (a), where there are two line-segments that should be connected on the left leg. The likely culprit was the angle difference, which was 5 degrees and therefore higher than the threshold used in this test. This shows how much affect a single threshold can have on the final result. Another error that is present is that some of the line segments does not fully encompass the edge it is supposed to represent. In Figure 4.4 (d). this is visible in the top right corner. This leads to an error in the calculated end point.

Another aspect that was observed during testing was that the number of segments detected could heavily impact the run time to extract the line segments. This is most likely stemming from the match-and-merge algorithm, where the amount of possible matches grows exponentially with the amount of lines. Some effort was done to alleviate this. Be-

fore matching the lines were sorted using quicksort (average O(nlog(n)) complexity). This was done to try to cluster lines that were spatially close, so that potential matches would be found faster and that you only had to look to the right in the working list of segments. This was tied in with dynamic handling by deleting matched segments from the working list, which reduced the size as the matching progressed. Although this seemed to improve the run-time somewhat more optimization might be required if the timing constraints of the image processing needs to be reduced.

As the test for the line segment extraction was based on Bjoernsen [2017] it is natural to compare the corresponding results. Figure 8.1 shows Fig 3 from Figure 4.4 on the left and Figure 35 of Bjoernsen [2017] on the right. It is clear that the line segments corresponding to the edge of the maze represents the overall shape in both tests. The reason for the more jagged edges on the left is the difference in resolution, as Bjørnsen's tests were done with a camera that could capture up to 3280x2464 resolution, while the tests done in this project thesis were capped at 640x480. The aforementioned unwanted lines along the base is not present on the right. This might be due to the distinct marking on the top of the maze, which makes it possible to tune the edge detector to only look for harsh edges. If the marked edges is a possibility for the full maze this could probably be accounted for in the current software by tuning the camera parameters to further enhance the edges between maze-edge and wall (i.e. changing contrast or brightness). If the marked edges are not wanted a possible solution could be to only accept line segments that are as large as the smallest expected wall, which would filter out many of the errenous lines. Another solution could be to take several images with different positions and then match the lines. As long as the position passes over the edge in question the base edge will only be observed for half the images due to a shift in perspective and can therefore be filtered out.



(a) **(b)**

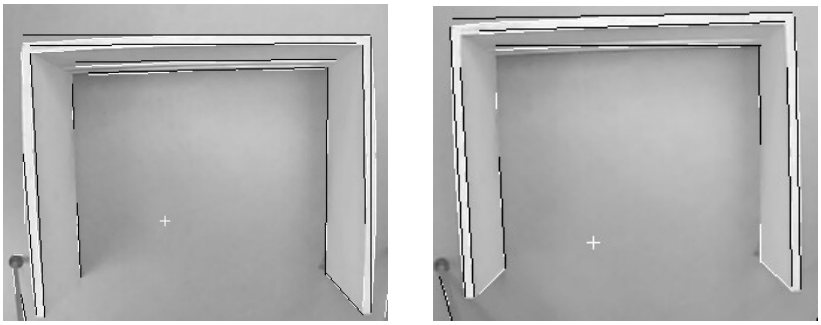
Figure 8.1: Left: Fig 3 from Figure 4.4, Right: Figure 35 of Bjoernsen [2017]

8.2 Global coordinates

For the accuracy of the real-life distance calculated with the GSD the results shows that there is a relatively constant error present at a given height. For the test at H = 150 this error averaged out to 8.7 %, while for H = 120 this came out to 5%. Again comparing to the results of Bjoernsen [2017] the error found in this thesis is worse overall. In the worst case (at H = 150) the error is 3-4 times greater. However, the measured error in this thesis was very consistent between test at a given height. This means that it is possible to scale the GSD given the heigth to counteract this error. It would however require that a relationship between these were established by experimental data. As to the reason for the error several sources could have contributed, in particular lens distortion and orientation inaccuracies of the camera module. The first case happens due to the curvature of the camera lens, and is usually worse for lenses with a wide field of view. It is possible to correct for this, but in the implementation of this system a trade-off between accuracy and resolution had to be chosen. The M7 implements a lens correction function, but it would require the images to be cropped, losing valuable information. This is due to the fact that full VGA-resolution image cannot be copied in the frame buffer of the M7 without running out of memory. Orientation errors could stem from the camera not being completely perpendicular to the ground during testing. As both the GSD and the coordinate transformation requires that this is the case any deviation would result in an error of the final results. To ensure that this error is minimized the test setup would have to be optimized to be more rigid, or a IMU could be integrated to measure the orientation. The latter would however require that the mapping incorporated the angle in the calculations.

Figure 8.2 shows two tests where the lens distortion is especially present. The distortion can be observed to be worse at lower heights, most likely due to the fact that the maze segment will cover a larger part of the image and therefore be further away from the image center. Using the width of the wall (1.5 cm) as a base-measure the error at the left side of the central wall seems to be about 1.5 cm at H = 120 and 0.75 cm at H = 150. In this context an interesting aspect is that the line merging process has an ”averaging” attribute. Even though the walls seems to be curved the merged line strikes through in a way that in some places counteract the distortion. This is especially prevalent on the right side of (a), where the straight line connects the right edge points even though the wall between them is distorted.

Comparing the global coordinates of the corners of the maze segment with the ones calculated from the image it is clear that they correspond well overall. The largest error can be found in Table 5.7 with 3.5 % for P4’s x-coordinate, and the smallest in Table 5.5 with 0.1 % for P2’s x-coordinate. These tests were performed after the GSD was scaled



(a) Line merge at H = 120 **(b)** Line merge at H = 150

Figure 8.2: Line segment inaccuracy due to lens distortion

to 150 cm, showing that the proposed solution has merit. The errors could have stemmed from several sources. The most likely culprit besides the already discussed lens distortion and camera angle is inaccuracies in the measurements. This could occur when placing the base at the coordinate (100, 100), setting the angle from the x-axis or measuring the edge points of the maze segments. The question of how much weight that should be put on the accuracy of the mapping needs to be seen in context with the accuracy of a position estimate from an UAV. If using distance sensors as in Bjerke [2022] (Section 8.1) the error from the estimate could be assumed to be around 1 cm, which means that the errors are comparable in size. To improve on the accuracy of the system several steps can be taken. One is to merge the results of the line extraction from several images. If a way to ”label” the line segments is implemented it could also be possible to implement full bundle adjustment (Triggs et al. [2000]), both optimizing the pose estimate and the global coordinates of the line segments.

8.3 SPI-communication

Although the SPI-communication test demonstrated that communication between the nRF52840 and the M7 is viable, several limitations were discovered while implementing the solution. One is that having a soft limitation on the speed between transactions could pose a problem. It is likely that the overhead required to improve this this is non-trivial, as both the timing and redundancy aspects would have to be handled to ensure reliable communication. Overall the robustness of the solution can be called into question, especially if one or both of the units have to process other events in combination with communication. The inherent lack of error checking and redundancy in the SPI-protocol means that one can-

not guarantee successful communication without implementing such features in software. The features needed for the standalone application in this project thesis were implemented (REC and LIN commands), but it is not given that this is enough if integrated in a larger system with stricter timing considerations. The way the nRF52 is polling the M7 while waiting for lines also leaves something to be desired. The original reason for choosing SPI over other available protocols with error checking (such as UART) was that the transferred data was intended to be full image files. As such transfer speed was the most important metric, with SPI being the clear candidate. With the final version of this system shifting away from image transferring it is possible that SPI, although working, is a less preferable choice to other available options.

8.4 Embedded system

One of the first things that are apparent when looking at the results of the embedded system test is that line segment coordinates match well between images. As seen in Figure 7.4 it is possible to match the lines extracted from the maze, if accounting for the fact that some lines might be oriented 180 degrees. As seen in Figure 7.3 the line extractor manages to find an edge for each section of the maze. It can however also be observed that some other edges are detected, e.g. in the left corner of (a), as observed in earlier tests. It is possible that one could remove these erroneous edges either as discussed in section 8.1 or by limiting the edge search to a user-defined rectangle within the image. Filtering out edges that are less than a certain length seems have merit. This can can be seen e.g. around the camera base and along the image border, where filtered lines are represented in white.

Looking at Table 7.1 and Table 7.2 both the error between edges in two distinct images as well as the overall error compared to the measured distance is small. The worst result is a 4% error in the P4 coordinate, corresponding the corner furthest away from the image center. This could in parts be due to image distortion. This test was done to ensure that all modules implemented worked together and gave meaningful results. The results show that this has been achieved, although under pre-defined pose conditions not subject to any noise. While some optimization and improvements should be made for robustness, the implemented embedded system should be able to operate as a ”black-box” if fed position data and when to capture images.

9

Conclusion

In the problem description of this thesis is was stated: *The final product is intended to be a functioning embedded system for the mapping of a maze, using components fit as a payload for an UAV.*. The test results of the embedded systems test shows that this overarching goal has been achieved, with some limitations. Without any other intervention from the user than powering up the system the nRF52840 and openMV CAM M7 manages to continuously communicate with each other, extract line segments from the environment under changing circumstances and prepare these to be distributed to the rest of the network. The hardware chosen is up to the task of image processing, but requires development overhead because of the limited computational power and RAM. The total weight of the system (assuming the nrf52840 DK is replaced with the corresponding chip) is 16-18 g, which fits with the goal of minimizing payload weight. The matching and merging algorithm works for the intended purpose, although not being compatible with real-time processing in most cases. The error in mapping for the embedded system was $< 5\%$ overall during testing, and could be further improved upon by various optimization strategies.The SPI-protocol works as a framework for communication between the units. However, it might be preferable to choose a more reliable protocol for future projects.

10

Further work

10.1 Integrate position estimator and Bluetooth

During testing the position estimate was either dummy data (for SPI-communication test) or measured and input manually. As a system for position estimation was implemented in Bjerke [2022] a natural next step would be to merge the systems so that the position could be fed directly. Furthermore the system created in this project thesis cannot communicate with the rest of the network of cooperating robots. This is necessary if one wants to distribute the extracted line segments. Several of the works done in the overarching project of cooperating robots have included communication with a server over BLE and MQTT. To enable communication could therefore be based on merging such a solution in the SW implementation on the nRF52 side.

10.2 Optimization

Although the way of extracting line segments and merging them works, the process can be slow, and the processing needed scales poorly when many line segments are present. If processing speed is deemed important there are several areas of the software that potentially could be refactored to improve run-time. The matching and merging algorithms have some of the largest overheads as they iterate over every line multiple times. If whole line segments are deemed unnecessary (a maze ”wall” can be represented by several segments) the merging algorithm can most likely be removed. This should probably be done after the system can access the central server, as testing should be done on how the lines behave

when integrated into the network. If line merging is wanted a first step in refactoring is to represent the lines on polar form while processing. Edge cases with regards to vertical lines in Cartesian coordinates ended up requiring much more SW overhead than expected. Another way to further improve the system would be to implement some sort of full bundle adjustment as discussed in section 8.2. As the M7 only tasks are to capture and process images it can be expected that there will be spare computational resources between capture commands. Although bundle adjustment is computationally heavy it might be able to run continuously in the background, pausing when images are to be processed.

10.3 Drone

The final intended product that this project thesis supplements is a drone with a payload that can extract line segment of a maze while hovering above. Although several of the main components are still in development it is still a possibility to test some of the capabilities of the systems in a dynamic setting. This would also require that communication with the server is implemented. Both the M7 and the nRF52840 runs on 3.3 V and could be connected to a small battery. The system could be attached as is to a commercially available drone (or with a 3D-printed enclosure). This would enable testing of dynamic aspects such as tracking over time or the effect of instability in the camera platform.

Bibliography

Kristian Bjoernsen. Mapping a maze with a camera using a raspberry pi. 2017.

Jean-Michel Morel, Gregory Randall, Jeremie Jakubowicz, and Rafael Grompone Von Gioi. Lsd: A line segment detector - researchgate.net. *https://www.researchgate.net*, Mar 2012. URL https://www.researchgate.net/publication/279348491_LSD_A_line_segment_detector.

Semiconductors Nordic. Nrf52840-dk - development kit, nrf52840 bluetooth soc, ant/ant+, arduino compatible, segger j-link. a. URL https://au.element14.com/nordic-semiconductor/nrf52840-dk/dev-kit-bluetooth-low-energy-soc/dp/284232102.

OpenMV. Openmv for optical flow¶. a. URL https://cdn.shopify.com/s/files/1/0803/9211/files/cam-v3-pinout.png?6147773140464094715.

Jonas Øygard Bjerke. Position measurement for quadcopter. 2022.

Marius Blom. nrf52 with openthread. 2020.

Maria Gilje. Implementing bluetooth le on a matlab controlled nrf52840 robot. 2022.

Marcus Steffensen Vormdal. Spi framework for nrf52840 and openmv cam m7. 2022.

OpenMV. Openmv cam m7. OpenMV, b. URL https://openmv.io/products/openmv-cam-m7.

Semiconductors Nordic. nrf52840 soc product brief version 3.0. Nordic Semiconductor, b. URL https://nsscprodmedia.blob.core.windows.

net/prod/software-and-other-downloads/product-briefs/nrf52840-soc-v3.0.pdf.

John Miano. *Compressed image file formats: JPEG, PNG, GIF, XBM, BMP*. Addison-Wesley, 2000.

Deep dive into bit depth. Teledyne, Sep 2021. URL https://www.photometrics.com/learn/camera-basics/bit-depth#:~:text=Scientific%20cameras%20typically%20use%20bit%20depths%20of%208,a%20typical%20cell%20image%20at%20different%20bit%20depths.

Bassam Hussien and Banavar Sridhar. Robust line extraction and matching algorithm. In David P. Casasent, editor, *Intelligent Robots and Computer Vision XII: Algorithms and Techniques*, volume 2055, pages 369 – 380. International Society for Optics and Photonics, SPIE, 1993. doi: 10.1117/12.150154. URL https://doi.org/10.1117/12.150154.

Sébastien Briot and Wisama Khalil. *Homogeneous Transformation Matrix*, pages 19–32. Springer International Publishing, Cham, 2015. ISBN 978-3-319-19788-3. doi: 10.1007/978-3-319-19788-3_2. URL https://doi.org/10.1007/978-3-319-19788-3_2.

Piyu Dhaker. Introduction to spi interface - epfl. Analog Devices, Sep 2018. URL https://wiki.epfl.ch/mecatro-me-424/documents/introduction-to-spi-interface.pdf.

Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment — a modern synthesis. In Bill Triggs, Andrew Zisserman, and Richard Szeliski, editors, *Vision Algorithms: Theory and Practice*, pages 298–372, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44480-0.

GeeksforGeeks. Python program for quicksort. Sep 2022. URL https://www.geeksforgeeks.org/python-program-for-quicksort/.

Torbjørn Øvrebekk. ncs spi-master-slave example. 2021. URL https://github.com/too1/ncs-spi-master-slave-example.

Appendix

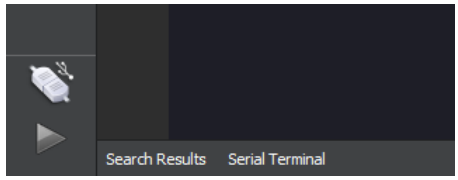
A Environment setup

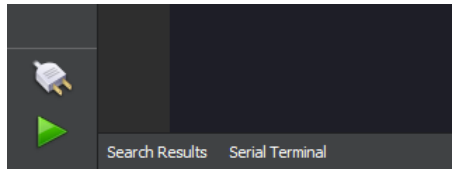
This section describes how to set up the environments needed to interface with both units. The operating system used was Windows 10. The procedures were first presented in Vormdal [2022].

openMV IDE and software installation

The openMV cameras comes with a proprietary IDE that is used to create programs, debug and flash the hardware. The IDE download can be found at `https://openmv.io/pages/download`. For this project version 2.9.7 was used. After downloading the next step is to run the openmv-ide-windows-(version).exe and follow the installation prompts. When finished openMV IDE will be installed and can be started.

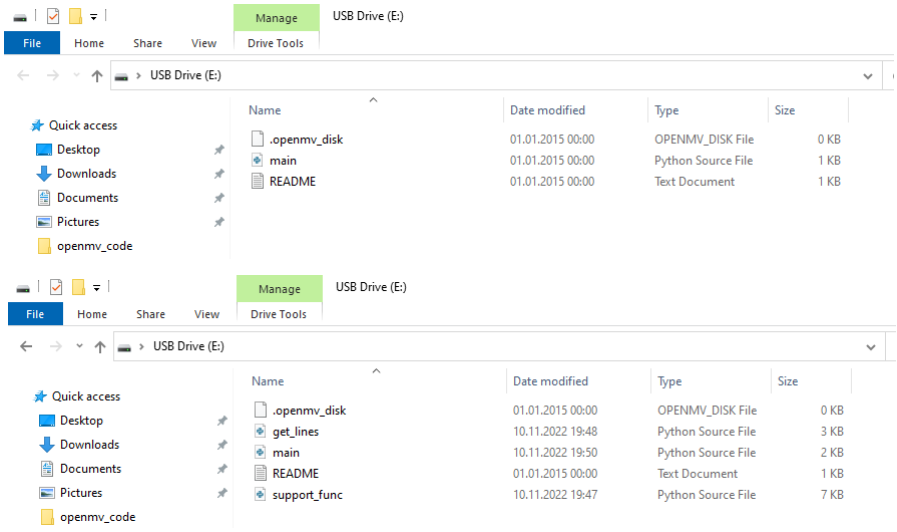
The next step is to connect the M7 to the computer using a USB 2.0 to microUSB-cable. The M7 should then show up as a separate disk (e.g. E:). One should then be able to connect the M7 to the IDE by pressing the connect button in the lower left corner:



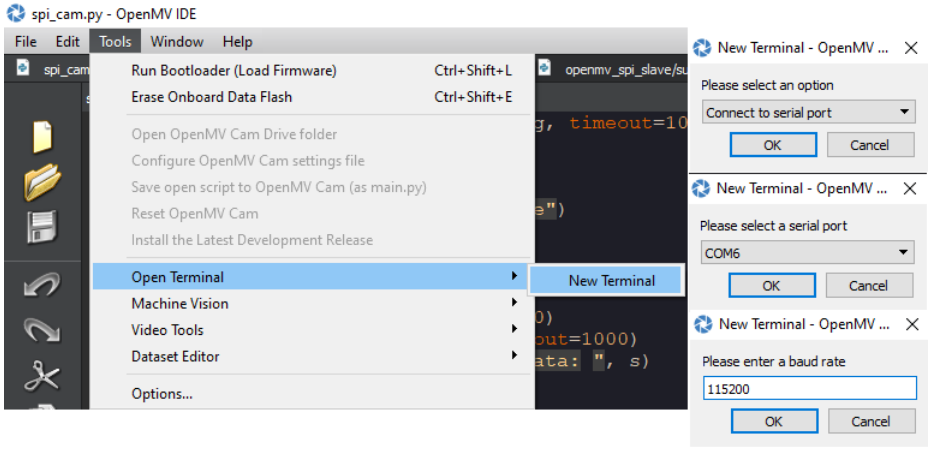


There are two ways to flash and run programs on to the M7. By pressing the green ”play” button while connected the current program file selected in the IDE will be flashed and run. The second is to copy-and-paste the files in to the directory that showed up when connecting the USB. In this project he second option is needed, as the code is split over several files. The implemented code can be found in the folder: spi_system/openmv_spi_slave.

Copy and paste main.py, support_func.py and get_lines.py from this folder into the M7 directory. Owerwrite the main.py. The before and after of the directory should look like this respectively:



Restarting the M7 will automatically start the loaded software, and it will run continuously until interrupted. To see the terminal output and the frame buffer without interrupting the process do the following procedure:



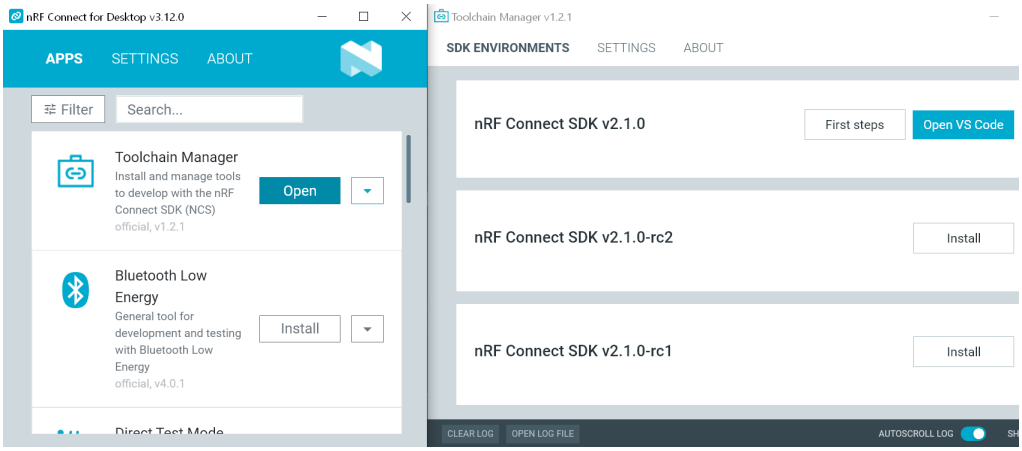
Note that the port will be determined by where the USB is connected. If this is uncertain try every port until output in the terminal can be observed. The setup of the M7 is now complete.

nRF52 DK

To interface with the nRF52840 DK Nordic Semiconductor's development software as well as Visual Studio Code was used.

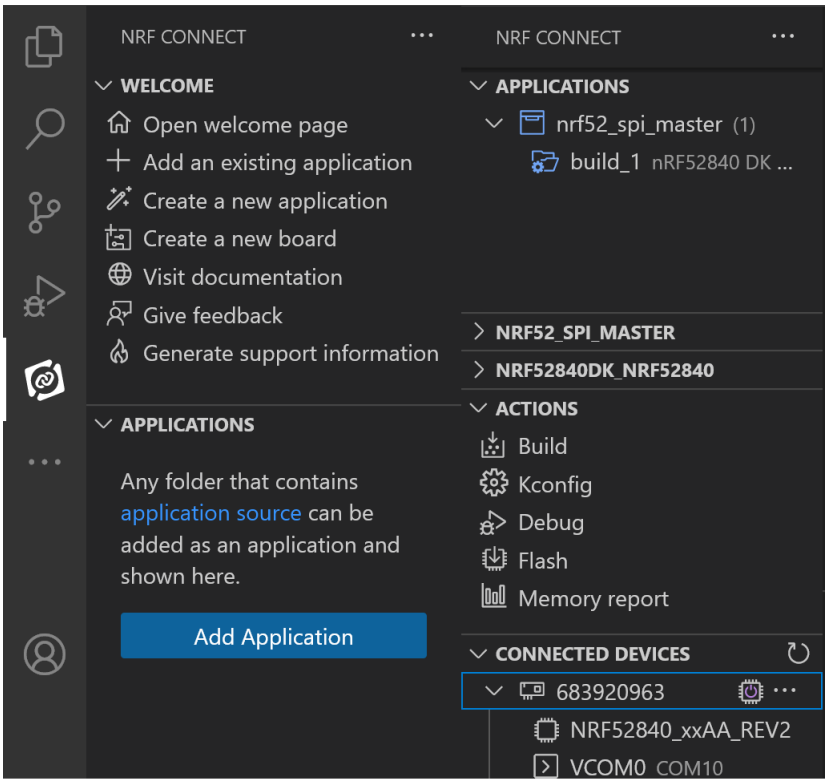
Begin by downloading and installing Visual Studio Code (VS Code). The download can be found here: `https://code.visualstudio.com/Download`. Run the installer, follow the installer instructions and then start the program. Under the extensions window on the left search for *nRF Connect for VS Code* and install the extension that shows up. Close VS Code.

The next step is to download and install the Nordic's developer tools. The download link can be found here: `https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-desktop` and here: `https://www.nordicsemi.com/Products/Development-tools/nRF-Command-Line-Tools` It is necessary to download both *nRF Connect for Desktop* and *nRF Command Line Tools*. After the download is complete run both installers and follow the prompts. Open nRF Connect and Install the *Toolchain Manager*. Open the Toolchain Manager and install *nRf Connect SDK v2.1.0*. You will then get the option to Open VS Code. This will prompt several missing dependencies, and also a prompt to install these directly. After installing, the nRF Connect for Desktop and Toolchain manager should look like this: Open VS Code in the Toolchain



Manager again and VS Code should open. When VS Code is opened the next step is to create an application out of the source code provided. This can be done by going to the *nRF Connect* section on the left side, expanding the menu option "Applications" and then pressing *Add application* This will prompt a file explorer window. After navigating to the folder *spi_system/nrf52_spi_master*, press *Select folder*. VS Code will then recognize

the folder as an application. From there expand the menu options *Actions* and *Connected devices*. Connect the nRF52 to the computer with a USB-2.0 to microUSB cable, refresh the *Connected devices* by hovering over the expansion button. The left side of VS code should then look like this:



The final step is to press the *Flash* button under the *Action*-menu. To access the terminal hover over *VCOM0* under the *Connected devices* and press the connect symbol that shows.

B Software implementation

M7 python implementation

The implemented code for the openMV CAM M7 consists of three files. *support_func.py* implements the matching and merging algorithms presented in section 4.1 and section 4.1, as well as the coordinate transformation presented in section 5.1. *get_lines.py* calibrates the camera, segments the image to feed in to the LSD described in section 4.1 and returns

the lines after merging them. *main.py* handles the SPI communication and state transitions described in section 7.1. This section presents the overall structure and implementation of each file, as well as some of the subtleties needed to make the implementation complete.

main.py

main.py handles the states and SPI-communication for the M7. The file consists of initialization of the states and data variables, configuration for the line merger, a main loop that waits or processes line segments depending on state and an external interrupt with a corresponding handler. The external interrupt is needed as the SPI-library of the M7 does not implement a check for the logic level of CS, which is used in SPI-communication. Every SPI-transaction is therefore done within the interrupt-handler, which comes with restrictions to memory access. It is necessary to initialize every state variable and data buffer outside the handler and include these explicitly within the function. Changing or accessing variables can only happen one element at the time, with array operations casting MemoryErrors. The handler separates between two cases through the bool **data_ready**, with True corresponding to line segments being ready to send and False waiting for CAP commands and position estimate. For all three files the function calls **gc.collect()** and **del** (followed by variable names) can be found. These are present to ensure that the M7 does not experience heap fragmentation. **del** marks the variable for garbage collection, and **gc.collect()** garbage collects. This is necessary because of the iterative nature of the image processing combined with the small heap-size. If a variable is cast and collides with an already existing variable in the heap the program terminates and needs outside intervention to restart.

get_lines.py

get_lines.py implements the overarching image processing. This is wrapped in two functions. **calibrate_sensor()** calibrates the camera to user specification. This might be needed if moving to areas with different lighting conditions. The function **get_lines(params, pos)** is the wrapper function that captures an image and calls the functions in *support_func.py* in the right order. The LSD can only run on pictures up to 160x120 before running out of memory. The first step after capturing an image is therefore to split it into segments that are small enough to be processed by the Line Segment Detector. It runs the LSD on each segment in turn, storing line data from each call in an collector array. When the process is done the collector array is passed to the line merger. The line merger returns a list of merged lines, which is then passed to the coordinate transform function.The file is also responsible for visualising the image and merged lines in the IDE.

support_func.py

support_func.py implements all functions related to line processing. This includes the following functions and classes:

Algorithm implementations

Several of the functions in *support_func.py* implements the algorithms presented in **??**.

merge_lines(params, lines) implements the overarching merging algorithm presented in Algorithm 1.

is_match(params, l1, l2) implements the matching algorithm presented in Algorithm 2.

vertical_match(l1,l2) implements the vertical matching algorithm presented in Algorithm 3.

horizontal_match(params, l1,l2) implements the horizontal matching algorithm presented in Algorithm 4.

merge(vertical, lines, end points) implements the merging algorithm presented in Algorithm 5.

Both *horizontal_match()* and *merge()* can encounter division by zero errors when calculating the slope of vertical lines. This is handled by catching the exception and letting the vertical equivalent handle the edge case.

class MuLines

Mutable Line class, as it was found necessary to extend the functionality of the line object when processing. Can be initialized with a touple on the form (x1, y1, x2, y2) or by passing a openMV line object. Formats the latter so that every line goes from left to right, and sets the angle to zero if the line segment is close to vertical pointing downwards. Although this is a approximation that is not completely accurate it is only used to conform with the threshold used in the matching algorithm.The angle propagated after processing only depends on the real transformed coordinates and is therefore not affected.

calculate_distance(l, pos, image_size)

Implements the transformation from pixel coordinates to global coordinates as described in section 5.1. Starts by calculating the GSD and the image centre based on the resolution of the image and the position estimate. An offset is added to the image centre as there is a discrepancy between the physical centre of the lens and the image center. The GSD also had to be scaled slightly to match real-life distance. The calculated distance was consistently 10% higher than actual measurements during testing. After converting to real-life units the coordinates are mapped to the global frame using Equation 5.4. Finally the corresponding angle to the line is calculated and the global line segment is returned.

quicksort(l)

Based on GeeksforGeeks [2022] with some modifications to sort on the x-coordinate.

nRF52840 C implementation

The nRF52840 C implementation is based on the example code in Øvrebekk [2021]. It uses the SPI-drivers implemented in Zephyr, which is integrated in Nordic Semiconductors' toolchain. The full build comes with a large overhead of machine generated code. The only files presented in this section are the ones that were edited during development or might be needed for future work.

main.c

Contains all the high-level code that is flashed to the nRF52 and runs continuously. It consists of functions to initialize the SPI-library, a SPI-communication handler called **spi_msg()** and the **main()** loop.

SPI initialization

Several steps have to be completed to set up the framework for SPI-communication. The structs ***spi_dev**, **spi_done_sig**, **spim_cs** as well as the function **spi_init(void)** maps the needed physical GPIO pins and interrupts to a software representation. The struct **spi_cfg** defines the SPI-mode, the clock speed, the master-role and the transfer size.

main(void)

Loops continuously when the nRF52840 is connected to power. Transitions between sending and receiving depending on the last message received over the SPI-connection. If ready

to receive new line segments **get_pos()** is called, which returns user set CAP commands. This is then passed to the function **spi_msg()** which activates a SPI-transaction and updates state flags. If waiting to receive line data the function opens a SPI-transfer window by calling **spi_msg()** with a ”dont-care” data package at regular intervals.

spi_msg(char cap_command[20], int* num_lines, bool *waiting, int *idx)

Implements the nRF52 SPI driver described in section 7.1. It initializes the buffers that are used to store the information to be sent and received, calls **spi_tranceive_async()** to start a SPI transaction and processes the data that is received. Separates between the three receive cases **REC**, **LIN** and line objects. **REC** signals that the M7 is processing and the Waiting flag is therefore set. If **LIN** is received the number of expected lines are extracted and the next data packets up until this number will be processed as line objects. These are packaged as line structs and returned. This is the final product of the system, ready to be further processed or transferred to the main server of the project cooperating robots.

nrf52840dk_nrf52840.overlay

No changes of this file were necessary during the project, but it is included as it might be necessary for further work. The .overlay format describes attributes of a DeviceTree object, which again holds the configuration for a specific hardware structure. The code maps the physical GPIO pins to a digital representation, which is passed to the SPI-framework. To change which pins are active changes can be made to **SPIM_SCK**, **SPIM_MOSI** and **SPIM_MISO** in the structure *spi1_default*. The CS pin can be selected by changing **cs-gpios** of the object *my_spi_master*.