

A Vision-Based Liquid Handling Robot for Automation of Chemical Experiments

Christoffer Stougaard Pedersen (cstp@itu.dk)
Martin Rønning Bech (mrob@itu.dk)
Mikkel Larsen (milar@itu.dk)

May 19, 2014

Abstract

The EVOBLISS project is a EU-wide cooperative endeavour to advance the development of artificial life research. One of the chief components in this feat is the development of an evolutionary robotic platform of which the first prototype, Splotbot, has been built. This paper describes the advancement against the second iteration of the prototype by solving specific shortcomings which have been voiced by the members of the EVOBLISS project, as well as advancing the current prototype in key aspects already present. This is done by building on the existing work done on the Splotbot platform to make a completely new prototype while thoroughly documenting the important design decisions and suggesting improvements where applicable. The result is a functioning prototype that improves the platform in key areas such as user interface and autonomous capabilities as well as proving the feasibility of key aspects such as running the platform completely standalone and scanning and stitching large areas. This paper seeks to provide key insights and solid recommendations for the next stages of development of the robotic platform.

Contents

1	Introduction	5
1.1	Scope of the project	5
1.2	End result	6
1.3	Past projects	6
1.4	Acknowledgements	7
1.5	Contents of the report	7
2	Constructing the physical robot	9
2.1	The hardware of Splotbot	9
2.2	From Splotbot to EvoBot	10
2.3	Making the frame	11
2.4	Designing and 3D printing the parts	11
2.5	Electronics	15
2.6	Experiments	16
2.7	Summary	16
3	Building the software to control the EvoBot	18
3.1	Description of the software running Splotbot	18
3.2	From Splotbot to EvoBot	19
3.3	Description of the software running EvoBot	19
3.4	Constructing the software core	20
3.5	Communication within the software	22
3.5.1	Communication from the core to the hardware	22

3.5.2	Communication within the core	24
3.5.3	Exposing the core to the user	24
3.6	Summary	25
4	Robot-camera calibration	26
4.1	Goals	26
4.2	Undistorting images	27
4.3	The relationship between images and the physical robot	27
4.4	Summary	30
5	Extracting experiment data using computer vision techniques	31
5.1	Goals	31
5.2	Droplet detection in Splotbot	32
5.3	Droplet detection in EvoBot	33
5.3.1	Considerations on the choice of filter	33
5.3.2	Considerations on the final choice of droplet	34
5.4	Testing the droplet detection	34
5.4.1	Degradation of performance	34
5.4.2	Quality of results	35
5.4.3	Choice of filters	38
5.5	Summary	39
6	Getting an overview of large surface areas	40
6.1	Goals	40
6.2	Scanning pipeline	41
6.3	Stitching images based on image features	42
6.4	Stitching images based on position	43
6.5	Stitching images based on both image features and position	47
6.6	Comparing the image stitching algorithms	47
6.6.1	Resulting images	48
6.6.2	Performance	49
6.7	Summary	52

7 Autonomous interaction with experiments	53
7.1 Goals	53
7.2 Executing instructions on EvoBot	55
7.3 Sending events with experiment data	56
7.4 Programming experiments	57
7.4.1 Rucola	57
7.5 Event reaction experiment	59
7.6 Summary	60
8 Logging experiment data	62
8.1 Goals	62
8.2 What is experiment data	63
8.3 Logging the data	63
8.4 Structuring the logged data	65
8.5 Use of limited hard disk space	66
8.6 Summary	66
9 Human interaction with the robotic platform	68
9.1 Goals	68
9.2 User interface in Splotbot	69
9.3 User interface in EvoBot	70
9.3.1 Constructing the graphical user interface	70
9.3.2 The bootstrap process	73
9.3.3 Choice of technologies	73
9.4 Summary	77
10 Conclusion	79
11 References	81

Chapter 1

Introduction

Hmm, ja vel, det er mere et sidemål for dem, så vidt jeg forstår.

This bachelor project resides within the scope of the EVOBLISS project that seeks to develop a robotic platform for supporting research on artificial, technological evolution with the goal of evolving microbial fuels cells in terms of robustness, longevity, or adaptability in order to improve wastewater cleanup. The main motivation for the EVOBLISS project is enhancing the understanding of living technologies and to gain an insight in the design of bio-hybrid systems.

The specific need for a robotic platform stems from a desire to achieve efficiency when running experiments. Other wishes that directly impacts this report in particular is an expressed wish for modularity and an expressed wish for advanced feedback and autonomous features from the system.

1.1 Scope of the project

This project is the continuation of past research, see section 1.3. For the remainder of this report we will refer to ‘the project’ as being the work carried out in this bachelor project.

This projects seeks to be an investigative design project with a proposed design in mind from the beginning. The two overall goals:

1. Assessment of the feasibility of our proposed design
2. Assessment of feasibility of specific additions to the existing functionality

To assess both of these goals the same method will be used, which is implementing the design and assessing the success of the implementation. It is worth noting that this approach has the consequence that none of the topics covered are investigated completely in-depth.

1.2 End result

The following will be delivered at the end of the project:

- A report containing:
 - An investigative review of and conclusion on whether or not the proposed design is feasible
 - A thorough review of the findings and choices made over the course of the project
 - A proposal for alternative design options where applicable
- A functional prototype demonstrating the implemented design

The common goal of the above is to objectively inform people building upon the work for the remainder of the EVOBLISS project about design choices made and hopefully provide guidance useful in future iterations of building the robotic platform.

1.3 Past projects

The work of this project is based on a previous robotic platform named Splotbot. It was originally developed as a master's thesis by Juan Manuel Parrilla Gutiérrez for his Master in Robotics at the University of Southern Denmark (Gutiérrez 2012). It was then improved in a project by Arwen Nicholson, also at the University of Southern Denmark, working on stability and running further experiments on the robot (Nicholson 2013).

Splotbot is based on a RepRap Prusa Mendel 3D printer, an open source 3D printer focusing on the possibility of self-replication. This printer has five stepper motors for moving along three axes as well for extruding plastic (Gutiérrez 2012, 15–31). For Splotbot, much of the hardware design was reused, but the frame was entirely new. It allows movement along two axes, X and Y. Six syringes had to be controlled, resulting in the need for twelve RC servo motors, which were added. It is controlled through an Arduino Mega 2560 (“Arduino Mega 2560” 2014) board, which is connected to a personal computer through USB. The user has to write Python code and use a Splotbot Python library to interact with the robot. The Splotbot library controls the robot by sending G-code (“G-Code” 2014) instructions to the Printrun (“Printrun Github Repository” 2014) application made to control 3D printers. Only a small subset of the available G-code instructions are used (Gutiérrez 2012, 33–48).

Printercore?

The functionality of Splotbot can be summarized in the following points. Splotbot is capable of (Gutiérrez 2012, 125–127):

- moving a carriage with 6 syringes along two axes

- controlling each syringe, moving it up and down as well as taking in and expelling liquids
- moving petri dishes around using a grip attached to the moving carriage
- obtaining live images from a stationary camera on which camera calibration is done manually.
- applying image processing techniques to detect droplets (single and multiple) and compute their centers and sizes. All the processing is done on the computer connected to Splotbot
- reacting on the result of the image processing in real time, modifying the experiment accordingly if necessary

With the improvements made by Arwen Nicholson, Splotbot is capable of running experiments for up to two hours without human intervention (Nicholson 2013, 26).

ikke sikker på man kan
bruge det ord på den
måde

1.4 Acknowledgements

Several people have aided us during the course of the project, the help from whom we are thankful:

- Lars Yndal has aided us in both finding out what hardware to buy in order to construct the EvoBot as well as in buying it
- Cathrine Siri Zebbelin Gyrn has helped us finding a suitable Plexiglas plate for the robot, and by allowing us to use workshop of the IxD Lab at the IT University of Copenhagen to work on the hardware construction
- Andrés Faína Rodríguez-Vila and Farzad Nejatimoharrami have helped by answering our hardware related technical questions as well as by ordering hardware parts

1.5 Contents of the report

The prototype created in this project, from here on named EvoBot, is meant as a combination of a demonstration of new features and a feasibility study of an implementation of the same features supported by the past projects on new hardware. Some of the existing features are therefore not as complete on EvoBot as they were on previous prototypes, as focus of the project is quite broad. EvoBot consist of a new hardware setup and a new rewritten software platform. Both are rebuilt from scratch, though based on what was made in past projects.

Each chapter of this report covers a topic of its own:

- **Chapter 2** explains how the physical robotic platform was constructed
- **Chapter 3** covers the software built in order to control the robot and support the wanted functionality
- **Chapter 4** focuses entirely on the camera of the robot and how it is calibrated to get rid of radial distortion and to relate the pixels of images grabbed with physical movement in the robot
- **Chapter 5** looks at how we use the camera to monitor experiments and gather data through tracking of colored droplets
- **Chapter 6** focuses on the camera as well, but investigates the problem of having to monitor experiments which cover a larger surface area than can be covered by a single image by grabbing multiple images and stitching them together
- **Chapter 7** looks at the problem of making EvoBot capable of interacting with running experiments without human interaction by giving the user a way to program entire experiments
- **Chapter 8** discusses the problem of what to do with the data generated by the experiments
- **Chapter 9** is about providing an interface through which the user can interact with EvoBot
- **Chapter 10** concludes on the results of the project

Chapter 2

Constructing the physical robot

This chapter focuses on building the physical part of EvoBot (the hardware). We start by briefly outlining the hardware of the Splotbot, as the EvoBot is very much based on this construction. We then move on to explain the modifications made from Splotbot to EvoBot, resulting in a description of the EvoBot setup along with an explanation of the reasoning behind the design.

2.1 The hardware of Splotbot

The basis of Splotbot is the metal frame. It is 25cm tall, 55cm long, and 35cm wide. In the center of the frame is a 3mm glass plate attached, on which experiments are run. An overview of the robot is given in figure 2.1 (a). The moving part of the robot is a top carriage which can move along two axes. This carriage is driven by belts. In order to move along the x axis, 3D printed pieces of hardware are mounted near the top corners of the frame. These hold an 8mm linear rail between them. On one side of the robot these pieces hold stepper motors with a pulley, while the other side have pulleys on ball bearings, allowing for driving a belt between them. This belt is then attached to another 3D printed piece running on ball bearings on the linear rails.

2.1a via latex

In order to move along the y axis, a stepper motor is attached to one of these pieces with the previously mentioned belt attached. A similar construction allows for driving a belt along the y axis. Two 8mm linear rails (one above the other) are held along the y axis on which the movable carriage is mounted on ball bearings. This carriage has six syringes mounted on it, each controlled by two RC servo motors.

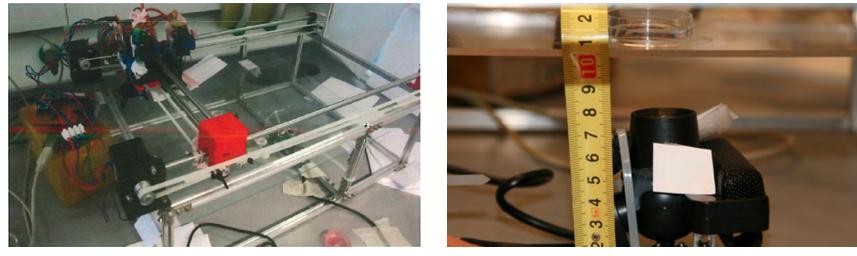


Figure 2.1: Images of Splotbot. Image (a) is from the thesis by Gutiérrez (2012). Image (b) is from the report by Nicholson (2013).

Splotbot has a camera to monitor experiments. This camera is fixed to the bottom of the robot as shown in figure 2.1 (b).

Finally, the Splotbot is controlled by an Arduino Mega 2560 (“Arduino Mega 2560” 2014). Instructions are sent from a personal computer connected via USB.

2.2 From Splotbot to EvoBot

There are several places where we found that the hardware of Splotbot was not sufficient for what we wished to achieve with EvoBot:

- The frame was a bit shaky when running, so we wanted the frame to be more robust
- We needed the camera to be able to move along two axes similar to the top carriage, so we wanted to replicate the top carriage design in the bottom part of the robot
- In order to make room for a bottom carriage, and also to preserve the possibility of exchanging the camera with larger cameras or scanners, we wanted more space below the glass plate
- The carriage is very shaky due to the rails on which it is attached being placed only with a vertical distance between them, so we wanted to alter this to be a horizontal distance only
- In order to make the EvoBot a standalone unit, we wanted to exchange the Arduino board with a more general purpose computer

The design with linear rails, belts, and stepper motors was proven to work on Splotbot, and given the limited scope of the project and us having hardly any knowledge about hardware, we decided not to spend time on trying to improve on it.

Finally, due to the limitations in available time we decided not to spend time on constructing syringe parts, though they are crucial to the functionality of the final iteration of EvoBot. We do, however, include controlling RC servo motors, so adding syringes similar to those on Splotbot should be possible to accomplish in a later iteration. We also did not work on building a gripper arm as the one on Splotbot, as this is currently being done in a separate project.

The following sections explain how we designed and built the hardware of EvoBot to suit our needs.

2.3 Making the frame

nu kunne vi ikke finde
rammetykkelsen for
Splotbot, så måske vi blot
bør skrive at den er større?

EvoBot has an aluminum frame similar to that of Splotbot. But the width of the frame is 3cm, making it a lot more sturdy. Furthermore, the dimensions of the frame have been increased to a height of 66.3cm, a length of 46.8cm, and a width of 76cm.

De to mål her bør nok være ens

Initially, EvoBot was constructed to be 66cm tall, 106cm long, and 76cm wide. This was mainly due to us having help (very much appreciated) with buying the hardware from a fellow student of ours, Lars Yndal, who in parallel with us were building a version of the EvoBot. His project needed a larger robot than ours, and he was quick to finding the hardware he needed to build it. As we at that point had no exact requirements or ideas as to what hardware we needed, we agreed that it would be a good idea to follow his lead and pool our purchases with his. This proved a successful strategy, as some parts had long delivery times and/or a limited selection of suppliers. However, as the work moved along, we found that the dimensions were too big for our needs, resulting in us only using about half of the length of the robot, as can be seen on the image of EvoBot in figure 2.2. We suggest that the frame is cut in half, removing the unused part. We omitted the cutting due to time constraints.

As on Splotbot, a transparent plate is attached to the frame on which the experiments are run. On EvoBot, this is a 10mm Plexiglas plate.

As the frame dimensions were altered, the 3D printed parts of the robot had to be redesigned to fit the change. This is covered in the next section.

2.4 Designing and 3D printing the parts

Det føles stadig underligt at skrive 5 her og have 6 billeder

EvoBot contains 5 unique 3D printed parts, all depicted in figure 2.3:

- An x axis holder with a stepper motor ((a), 4 pcs.)
- An x axis holder without a stepper motor but with a pulley instead ((b), 4 pcs.)

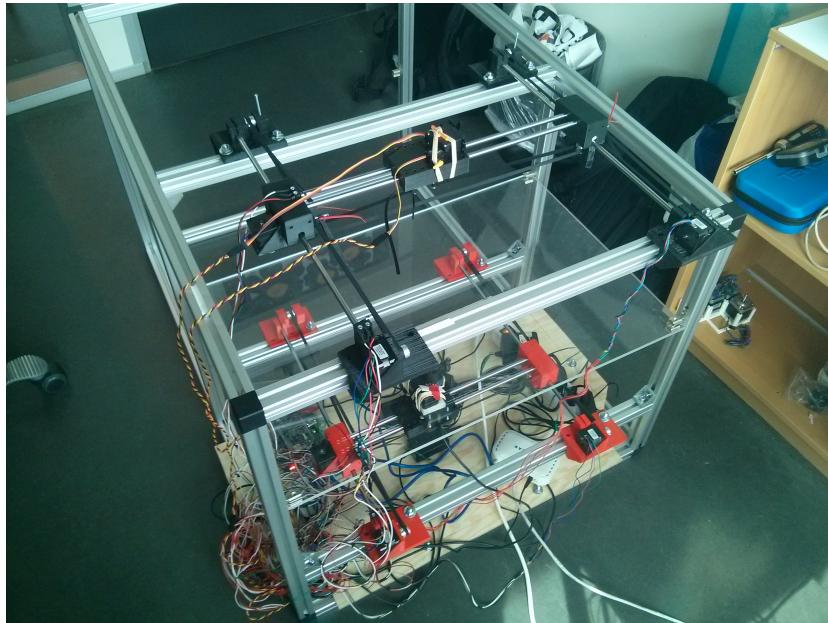


Figure 2.2: EvoBot.

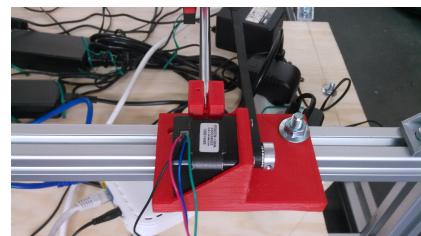
- A y axis holder with a stepper motor ((c), 2 pcs.)
- A y axis holder without a stepper motor but with a pulley instead ((d), 2 pcs.)
- A carriage ((e), 2 pcs.)

Each of the parts were designed and printed by us. As a disclaimer, none of us has previously made such a 3D design. Once a part was printed, we always ended up spending time sawing and drilling it to fit the robot. This was due to both imprecision in the print and due to the designs being less than perfect. As the printing of a part takes a long time (usually several hours), having many iterations of designing and printing the parts seemed quite time consuming. We therefore often deemed a print as being ‘close enough’ and then used tools to alter it slightly.

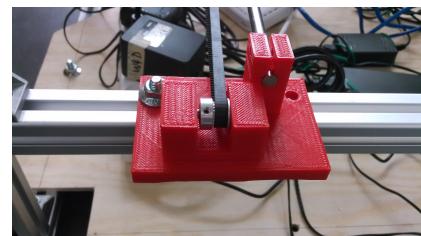
måske bare dropte
ordet slightly, er
lidt udefinerbart

The x axis holders are very much inspired by the ones on Splotbot, but with modifications to make them fit the larger frame. They have also been increased slightly in size, making them more thick and making sure they are fastened to two sides of the frame rather than one.

The y axis holders are also similar to those on Splotbot, but with their width increased so they run on two ball bearings rather than one, increasing the stability between the two holders when moving along the x axis. Also, the holes for the linear rails have been moved so the two rails are shifted only by a horizontal distance rather than a vertical one.



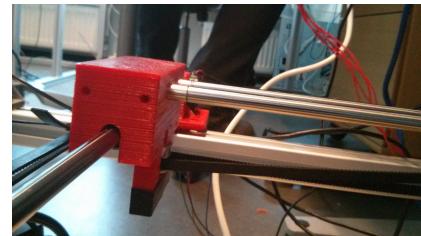
(a) X axis holder with motor



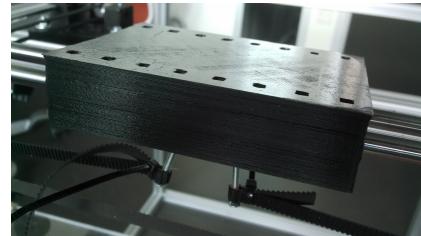
(b) X axis holder without motor



(c) Y axis holder with motor



(d) Y axis holder without motor



(e) Carriage



(f) Camera component

Figure 2.3: The 3D printed parts of EvoBot.



Figure 2.4: Issues with the design of the y axis holders.

We actually made two iterations of the design of the y axis holders, as the belts controlling the two axes were touching each other on the first design as shown in figure 2.4 (a). The second design shown in figure 2.4 (b) however introduced other issues, as the belts were lowered to an extend where the distance from belt to carriage is so long that it is difficult to attach the belt to the carriage. Currently, the first design is represented in the bottom of the robot and the second design in the top.

We have a quite practical issue with the y axis holders in the top, as we ran out of ball bearings of the right size. We did not have time to order new, so the holders are attached to the rails on ball bearings that are too small and then raised with small pieces of paper so the plastic does not hit the rails directly. The result is that the top x and y axes run very badly (when they run at all). This should easily be overcome with the correct ball bearings, but since the top carriage is not a necessity for what we wish to achieve in this project, we decided to leave the top as is.

The carriage of the EvoBot is designed entirely from scratch. There are several reasons for this. The first is that the carriage had to be modified for running on the horizontally shifted linear rails. Furthermore, where the carriage of Splotbot is made solely to house 6 syringes, we wanted the carriage of EvoBot to support various types of components. We consider a component as being an individual piece of hardware which can be separated from and added to the robot without interference with other parts of the setup. In order to allow such attachment and removal of components, we designed the carriage as a block with holes in it in which we insert wooden dowels. Any component to be added on the carriage then have to have similar holes, so it can easily be added on top with the dowels holding it in place.

The only such component we have currently made is the camera component shown in figure 2.3 (f). It consists of two plates separated by nuts and bolts, the bottom one having the holes allowing it to be attached to the carriage. The nuts and bolts setup allows for all four corners of the top plate being adjustable separately, so the camera frame plane can be aligned as close as possible with the Plexiglas plate. On the picture, the component is not fully attached to the

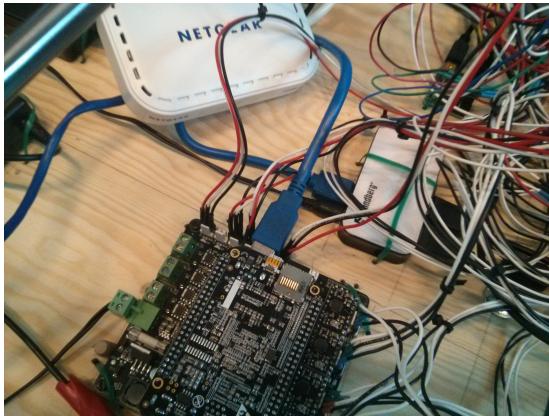


Figure 2.5: The electronics of EvoBot.

carriage as the holes of the component are a bit too small. But it is still very firmly attached.

2.5 Electronics

In order to make the robotic platform actually do something, we need some kind of electronics to control the stepper motors, limit switches, and RC servo motors. As mentioned, an Arduino board as used in Splotbot is not sufficient, as it is in the way of the robotic platform being completely standalone. This is because the logic contained in the Arduino board is the execution of simple instructions, whereas the needed image analysis and processing requires more powerful hardware. This is why we for this project decided to use a BeagleBone Black as the brain of the robot. Figure 2.5 shows a picture of the electronics of the EvoBot.

The BeagleBone Black is a microcomputer with 512Mb of RAM, a 1Ghz ARM processor, a single USB port, an ethernet port, and 2x46 GPIO ports. The board comes with an embedded Linux distribution called Ångström (“Beagle Bone Black” 2014). Because of it running Linux, the development and deployment of code for the board is as easy as connecting to the board and compiling and running the code directly on the board. The GPIO ports can be accessed through a device tree overlay, and the USB connections are registered as a regular device connected to the Linux system.

In addition to the BeagleBone Black in itself, we utilize two hardware components in order to communicate with the moving parts of the platform. The first is an expansion option called the BeBoPr++ cape (“BeBoPr Cape” 2014), which has connections for stepper motors and limit switches (along with other connections, which we do not use). For the stepper motors, separate hardware stepper

drivers are used. The cape also provides surge protection. The servo motors are controlled with a Polulu Servo Controller board, an USB device for controlling servo motors (“Polulu Servo Controller” 2014). In order to use multiple USB devices with the BeagleBone Black, we use an USB hub.

The final piece of hardware on EvoBot is a small wireless router, which is connected to the ethernet port of the BeagleBone Black. The BeagleBone Black is set up with a static IP address in the router, which means than access to the robot is as simple as connecting to the wireless network (wired connection is also possible) and connecting to the board using the known IP address.

In order to have the electronics mounted on EvoBot, we have attached a wooden plate to the bottom of the frame on which the electronics and power supplies are fastened.

2.6 Experiments

In order to assess the suitability of the hardware design, a simple experiment was designed. The goal of this experiment is to measure the approximate precision and consistency of a crucial part of the design, the motors and belts moving the carriage. To do this, the following steps where performed:

1. A cross is drawn on a piece of paper with millimeter precision lines
2. The EvoBot camera is placed in position (0,0)
3. The paper is placed on the Plexiglas plate above the camera so the cross is visible, and an image is grabbed
4. ~~The~~ EvoBot is given the instructions to fifty times move the camera the max position and then return to (0,0) again
5. Another image is grabbed
6. The two images are compared to see how far off the cross is in the second image compared to the first

Step six was done by overlaying the two images in image manipulation software and measuring the difference. As it turns out, however, there was no visible difference. The two images grabbed can be seen in figure 2.6.

2.7 Summary

EvoBot draws almost all of its inspiration in terms of physical hardware design from Splotbot, which in turn attributes its design to the RepRap Prusa Mendel 3D printer. EvoBot does add both entirely new features, such as the bottom carriage, and improves on the existing functionality, such as added stability to the frame and carriages. The modification of the frame meant that all the 3D

Dette har vi ikke nævnt i dette kapitel, kun i 'Introduction'

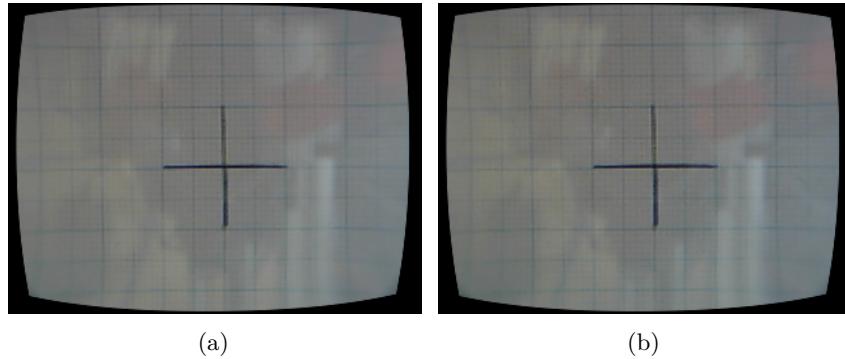


Figure 2.6: The resulting images of the movement precision experiment.

printed parts of Splotbot had to be redesigned to some extend. There are also parts of Splotbot which are not included in this iteration of EvoBot such as syringes and the gripper arm for moving petri dishes.

In terms of electronics, the Arduino used for controlling Splotbot has in EvoBot been replaced with a BeagleBone Black, a general purpose microcomputer, allowing the robotic platform to be completely standalone. The BeagleBone Black is accompanied both with special purpose hardware to drive motors, as well as commodity hardware such as a web camera and a simple wireless router which assures easy connectivity to the robot.

Chapter 3

Building the software to control the EvoBot

In this chapter we will first describe how the software in Splotbot is implemented. We will then compare the software of the Splotbot with that of the EvoBot. Then we will look at how the EvoBot core software application is constructed. Finally we will look at how the core software accesses the EvoBot hardware

3.1 Description of the software running Splotbot

Printron or Printcore?

The core of the software that is used to run Splotbot is the python based software Printron . Printron is made for RepRap 3D printers. It takes G-code instructions and translates them to motor movements etc. The Splotbot can be controlled by writing a set of G-code instructions to move the hardware because most of the common functionality, such as moving axes, is accessible by G-code.

Splotbot can be controlled by the use of a piece of python software built on top of Printron. This software generates G-code that is executed by Printron resulting in movement of Splotbot. The Splotbot software is structured much like a library and contains code for doing camera calibration, droplet tracking, robot movement and controlling the syringes. In combination these features can be used to design experiments to be run on the Splotbot.

Experiments for Splotbot are designed as Python scripts using the rest of the Splotbot code to provide the needed functionality. Experiments previously run on the Splotbot includes taking liquid from containers and injecting it into a petri dish, tracking a droplet and reacting to the droplet speed.

Jeg troede kun biblioteket var 200 linjers kode til at g

3.2 From Splotbot to EvoBot

The EvoBot software differs from the Splotbot software in many ways and is a complete rewrite of most of the functionality available in the Splotbot while also adding new features. The software is no longer Python based but is instead written in C++ and JavaScript. The main objective of the software is now to be a single application that is run on ~~the~~ EvoBot on startup. EvoBot also differs from Splotbot in that it has to be a platform for making many different types of experiments which might require other hardware to be added to the system. EvoBot therefore puts emphasis on an architecture that allows the functionality to be extended with other types of hardware.

The software still supports features such as droplet tracking, moving ~~the carriage~~, and moving servo motors. But it has been extended with features such as scanning and stitching a large area, a web interface ~~and~~ and a programming language. Focus has been put ~~on~~ making a standalone robot to which a user can connect and control without installing anything on his own computer.

Experiments are now run ~~by~~ using the EvoBot software rather than by writing a python script. Experiments can either be defined as low level instructions similar to G-code or via a domain specific programming language (DSL) made for ~~the~~ the EvoBot. ~~The~~ EvoBot also supports the possibility of making experiments that observes the status of the petri dish and reacts ~~based~~ on it, such as watching change in the speed of a moving droplet.

3.3 Description of the software running EvoBot

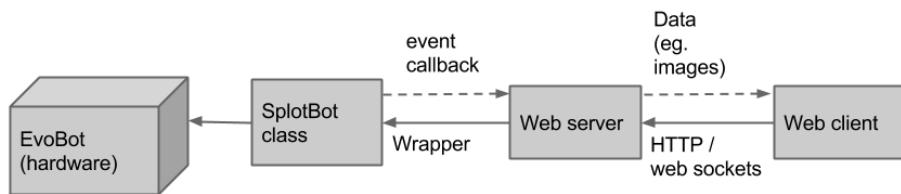


Figure 3.1: The software architecture. Note how the old naming of the ‘splotbot’ class is preserved for consistency

The software written for our prototype is structured in three main components: A core, the web server ~~and~~ and a client. The core handles the robot features, ~~the~~ the web server acts as a bridge between the core and the client. The client allows the user

control the robot and run experiments

to manipulate the robot real-time and also allows the user to send experiment code to be executed on the EvoBot. Below is a brief description of the software in its entirety. The source code is available in our Github repository (“Code Repository” 2014).

- The core of the software is written in C++ and is responsible for executing experiment code, communicating with the hardware, logging data, emitting events and in general it is the most extensive part of our code base with the main responsibility for handling the platform. The software consists of a module based system where modules are loaded at startup based on settings in a configuration file. This allows for modularity in our design and for new hardware to be added in the future.
- The EvoBot becomes programmable for a user through the use of a custom DSL language called Rucola. We designed this language to fit the needs for creating experiments. Rucola is available as a library, which the core application uses to compile and evaluate Rucola code. [ref](#)
- As part of our application we have a library of computer vision related tools that some of the components in the core application uses to provide features such as droplet detection and image scanning.
- The web server, written in NodeJS, wraps the core application providing access to it through a web socket and REST http interface. This means that EvoBot functions as a web service that can be accessed through any application language that supports web sockets and http.
- The client consist of a JavaScript based web client, that communicates with the server through web sockets and http requests. Like the core application, the client ~~similar~~ loads the configuration file and constructs the GUI based on this.

graphical user interface

3.4 Constructing the software core

Highlight i typewriter font hvis klassenavn

The software core spawns from the Splotbot class, which was given the name before the robot became EvoBot. The Splotbot class constructs all the components from the configuration file. All of the components are then instructed to register ~~all of~~ their actions in the action list. The action list is later used to call ~~of~~ the different component actions. Currently the EvoBot has five available components each with different functionality and hardware requirements:

- **Camera** handles video recording, image grabbing, and droplet tracking.
- **XYAxes** is used to control a set of two axes on the robot. The XYAxes has functionality such as moving to a specific position, and homing, resetting the position of the carriage to its initial value.
- **RCServoMotor** is used for a single servo motor and has functionality to move it. In a more complete setup this class would probably not be used in favor of using a class such as a ‘syringe’ component type

Byt de sidste to sætninger ru

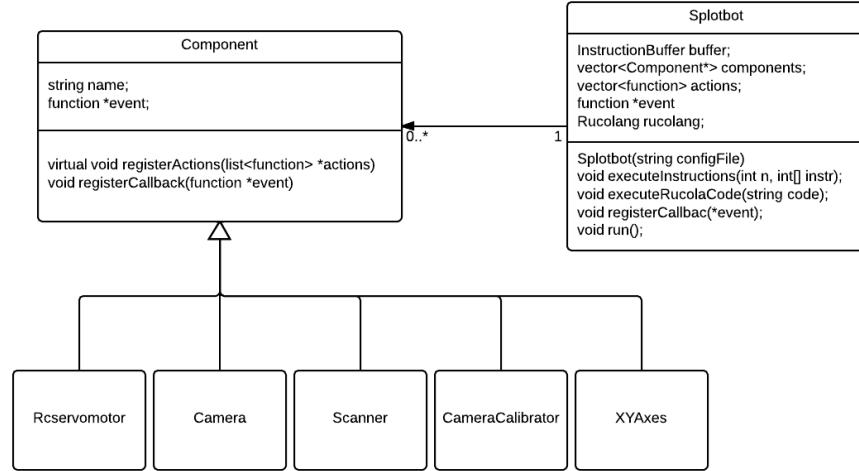


Figure 3.2: Simplified class diagram of the core EvoBot software.

- **CameraCalibrator** is used for calibrating the camera. This component has functionality to perform a camera calibration, and for saving and loading such calibrations.
- **Scanner** is used to scan large surface areas using image stitching techniques.

An important part of the design of the software core has been to ensure that it is kept modular with the intention of making it possible to extend it with more hardware options in the future. Modularity is achieved by making components for each feature of the robot encapsulating the functionality in a single place. A component is then defined in the configuration file to signal to the software that it should be available. Implementing a component can be done by using the following steps:

- The settings of the component must be defined e.g. a syringe component which consists of two servo motors connected to the physical Servo Controller. The definition must be reflected in the configuration file. The definition must at the very least have a type name (e.g. Syringe), a name (unique for each component instance), and how it is connected to the peripherals of the BeagleBone Black.
- The component must be implemented, inheriting from the **Component C++** class and implementing the virtual methods.
- The **componentinitializer.cpp** file must be updated to know about this new type of component including how to initialize it from the configuration file.

```

1  {
2      "type": "XYAxes",
3      "name": "BottomAxes",
4      "parameters": {
5          "x_port": "X",
6          "y_port": "Y",
7          "x_limit_switch_port": "J9",
8          "y_limit_switch_port": "J11",
9          "x_step_limit": 79,
10         "y_step_limit": 58
11     }
12 }
```

Figure 3.3: Example config

punktum

The configuration file is written in JSON, an example component can be seen in figure 3.3. As a part of the configuration every component needs to state its type, name and some parameters that the C++ code of the component will use. The parameters are often used to define on which ports some hardware can be accessed.

3.5 Communication within the software

Det første afsnit handler om hardware ...

In the following we attempt

The following attempts to outline how the described parts of the architecture interact with one another and, importantly, how they interact with the underlying hardware.

3.5.1 Communication from the core to the hardware

EvoBot consists of multiple hardware components which are all accessed in different ways. This section serves as a description for each of the hardware components accessed and explains how they are controlled.

- The camera of the application are accessed through OpenCV which uses video4linux as its underlying driver. The camera is accessed via the VideoCapture class in OpenCV where every frame can be grabbed with a single method call.
- The stepper motors are accessed through the BeBoPr++ 3D printer cape. This cape comes with software, named Mendel after the 3D printer model it is created for. Mendel is capable of executing G-code instructions, controlling the hardware.

- The servo motors are connected via the Pololu Servo Controller and can be directly communicated to through writing to the USB device. We based our implementation on the C program available on the Pololu website (“Pololu Maestro Servo Controller - Cross-Platform C” 2014).
- Limit switches are used to detect when the end of the axis has been reached. These are registered on the BeagleBone black as a device three overlay, meaning that communication can be done through the file system. Each of switch, of which there can be four total, has a folder with a ‘value’ file in it. By reading from this file, as you would any other ordinary file, it is possible to see whether the switch is pressed or not.

With the above three points, a few problems arose.

Regarding video, although video drivers are handled on the specific platform and are not an issue for the software, OpenCV comes with more dependencies. Namely codecs are defined with `ffmpeg`, which differs across platform, and the configuration used on the code does not necessarily match that which is installed. Members of the teams struggled with finding a common configuration, so the board were to be considered the lowest common denominator, making it each team members job to remember to change the configuration if he wanted to run the software locally.

Regarding stepper motors, a few modification had to be done. Firstly, the software is not build for our purposes, but rather specifically for controlling a 3D printer. This means that the 4 axes do not have the same properties, as one of them for instance is used as an extruder. We patched the software to remove boundary restrictions and to fix some calculations to make sure each axis moves at the same step size. The result is that we can successfully make our own homing functionality on each axis.

Another obstacle with the stepper motors is communicating instructions to them. The Mendel library that came with the BeBoPr++ cape, while open source, was not entirely transparent in its design. It is not very well documented and changing it often resulted in fatal errors. The implications of these difficulties was that we did not succeed in incorporating it into our own software directly. Instead we opted for having it run as-is (with our previous mentioned modification) and write to it from the outside. Specifically, each time we start the EvoBot software we:

nævnt før?

1. Write an empty string to the file, creating and truncating it as needed
2. Start the Mendel software
3. Open a continuous stream reading from the socket and pipe it into the Mendel executable
4. Pass the path to the socket to the EvoBot software

Lidt mangelfuld beskrivelse

This way, every time the EvoBot needs to communicate with stepper motors, all that is needed is to write G-code to this socket. While this approach has proven

jeg er lidt i tvivl om
der har afsnit er nødvendigt
og hvad det prøver at
fortæller mig

to work throughout the project, it does seem vulnerable to IO failures. A later project might look into improving on this method.

3.5.2 Communication within the core

The core is the one part of the architecture that functions entirely without the others. This means that it works under the assumption that no other components exists, and is only contacted from the outside in. It does, however, need to be in sync with the other components with regards to what is exposed for calling. As mentioned, this is done through the shared configuration file. This configuration file dictates which components exists, coupled with a knowledge about which actions these components can perform, each action will be given an index and uniquely refer to a method on a component instance. This knowledge of component actions is for now only stored in the configuration, but it should be trivial to read it from a configuration somewhere. For now we did not want to introduce complexity and maintenance by having a new often-changing configuration file while the system was developed, nor would we want to store this information in the current configuration as it would require the information to be duplicated on all components, or changing the structure of the file.

In addition to the exposed actions, the core, ~~of course~~, can also call itself. This is seen for instance with specialized functionality, such as computer vision, where a separate library is created and used where needed within the core. ~~Another example is of course just methods that are not exposed on components.~~

3.5.3 Exposing the core to the user

As mentioned, the user facing part of the EvoBot goes through a web server. This makes it rather trivial to make the actual user facing part, as it is a static web client. Exposing this client application is only a matter of having the web server already used for communication also host the client software.

The communication between the core and the web server is less trivial. The two parts are written in different languages, which means that a wrapper is required to have them communicate. The wrapper for doing this works mostly out of the box, and allows the web server to call methods and pass parameters to the core. We did however experience problems when the communication needs rose to be more than the most basic. Namely callbacks, which is an advertised functionality of the wrapper, proved to not work in our software. Speculation revolves around that the culprit is memory management in relation to the way we use threads in the core. Regardless of the underlying cause, we did not succeed in fixing the native callback functionality, and instead chose another way around the issue by doing the callback through over `http` using the curl library. This works, but it is likely to be slower than having a natively supported callback.

3.6 Summary

?

The Splotbot is run on a piece of Python software based on a library specifically used for controlling a 3D printer. The transition from Splotbot to EvoBot is very much a rewrite, resulting in a C++ library exposed through a web server, operated with a JavaScript based web client. The core of the EvoBot consists of different component working as an abstraction on top of the corresponding hardware components.

Hvordan kan vi lade være med at nævne config filen?

Chapter 4

Robot-camera calibration

EvoBot uses computer vision to obtain information about the experiment. The camera is therefore a central part of the current EvoBot setup. This was also the case for the Splotbot setup. We also use the same PlayStation Eye camera with a 10x zoom lense attached, meaning that the images grabbed suffer from the same radial distortion as described by Juan Gutierrez (Gutiérrez 2012, 59–65). Furthermore, one of the image stitching algorithms described in chapter 6 assumes prior knowledge about the relationship between the images grabbed and the physical robot. These are the topics of this chapter.

4.1 Goals

In terms of radial distortion we have the obvious goal that we want the robot to **be able to undistort images from radial distortion**. We also want to **be able to calibrate the camera in terms of its relationship to the movement of the carriage on which it is mounted**, as this is a necessary precondition for other functionality. In order to make the calibration as convenient as possible for the user, we set forth the goal that **camera calibration has to be done only once**. Of course, if the hardware setup changes, the camera will have to be recalibrated. This also means that if the setup is made by a technician, she can also calibrate the camera, and users will from there on not have to recalibrate it. Finally, we want the two different calibrations, for radial distortion and for the relationship with the physical movement, as **being conceptually a single calibration from the point of view of the user**. This limits the number of required input from the user, before the robotic platform can be put to use.

4.2 Undistorting images

In order to undistort the images grabbed we use an approach similar to the one in Splotbot (Gutiérrez 2012, 61–65). We grab 9 images of a chessboard pattern, of which the corners can be detected by use of OpenCV. We then use OpenCV again to estimate the intrinsic camera parameters, and again, we use OpenCV to use these parameters to undistort the grabbed images.

What we do differently than in Splotbot is that we use the fact that we can move the camera along two axes to automatically grab all the images needed to do the calibration. For this, we require the user to lay a 9x6 chessboard pattern over the camera. The entire chessboard pattern must be visible on the same image in order for OpenCV to be able to detect it. If the camera is at position (x, y) , then the camera is moved to the positions

$$\{(a, b) | a \in \{x - 1, x, x + 1\} \wedge b \in \{y - 1, y, y + 1\}\}$$

and an image is grabbed in each position. The chessboard pattern corners are detected in each image, and based on these corners the intrinsic camera parameters are estimated.

When turning on the camera, each time an image is grabbed, it is checked whether the camera has been calibrated. If it has, the image is undistorted. Figure 4.1 shows nine images used for a calibration and the resulting undistortion of an image.

We have experienced that the size of the chessboard pattern impacts the resulting undistortion. Figure 4.2 shows three calibrations done with different size chessboards. We have found that the larger an area of the image the chessboard pattern covers, the better is the resulting calibration. But only to the extend that the corners of the chessboard pattern must be visible in all of the images grabbed.

4.3 The relationship between images and the physical robot

Each time the camera is moved a step in either the x or y direction, the relationship between two images grabbed, one before and one after the movement, is a simple translation. If we assume that the camera is aligned with the axes of the robot and that the image plane is parallel to the Plexiglas plate. With this assumption, we can estimate the exact translation for a step in either direction respectively with the following steps:

1. Grab an image
2. Move a single step

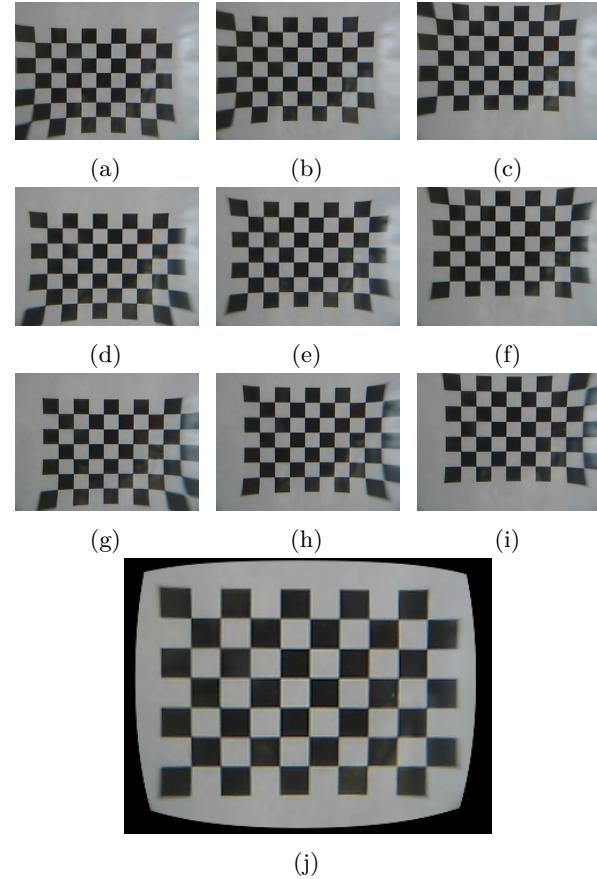


Figure 4.1: Calibration of the intrinsic camera parameters and undistortion of images. Images (a) - (i) show the images used for the calibration. Image (j) shows an undistorted image.

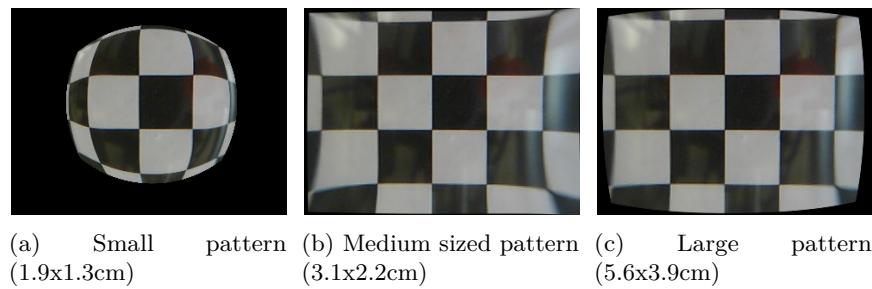


Figure 4.2: Calibration with three different sized chessboard patterns.

3. Grab a new image
4. Estimate the translation based on an object recognizable in both images

This has to be done for a step in each direction separately. In order to avoid errors having a large effect, we repeat this process multiple times in the implementation and average the translations computed.

The first three steps are trivial. For our calibration we use the same 9x6 chessboard pattern as a recognizable object. This has the advantage that it is easily detectable by OpenCV, and also that the calibration can be done in extension with the previous calibration. When the previous camera calibration finishes, the camera is already located below a detectable chessboard pattern, which can then be used for this calibration. This also helps fulfilling the goal of the user seeing the two actual calibrations as a single calibration.

Several consideration have been put into the final step. Our first attempt at estimating the translation was based on the capabilities of OpenCV. When the chessboard corners have been detected in both images, OpenCV provides a function for estimating an affine transformation between the two images based on the points. The result is a transformation matrix on the form (Paulsen and Moeslund 2012, 134–137):

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ 0 & 0 & 1 \end{bmatrix}$$

where a_3 is the translation in the x direction and b_3 is the translation in the y direction. From this matrix we then pulled the translation values. But when used in image stitching (chapter 6), these values were not very precise. The problem is that the affine transformation also encodes scaling, rotation, and shearing (Paulsen and Moeslund 2012, 134–136), and when we simply remove these, a part of the relationship between the position of the chessboard on the two images are lost.

We therefore tried a different, more simple approach. When detecting the chessboard corners in each of the images, we use the corners to compute the center of the pattern by averaging the x and y values of all the points. The result is that we have two points, (x_1, y_1) in the first image and (x_2, y_2) in the second image. The translation in each direction is then found by the simple calculation:

$$\begin{aligned} \Delta x &= x_2 - x_1 \\ \Delta y &= y_2 - y_1 \end{aligned}$$

This second method provided much better results. The calculation is depicted in figure 4.3. It is this implementation that is currently in use, and examples of application of the results of this calibration are given in chapter 6. But it

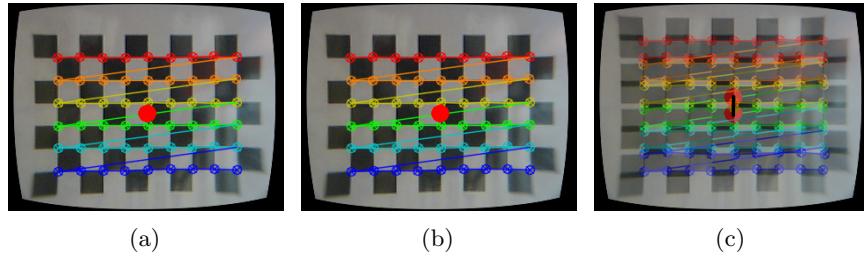


Figure 4.3: Calibration of the correspondence between physical steps and image pixels. Images (a) and (b) are the input images with the chessboard corners and center points detected. Image (c) shows the estimated translation vector illustrated by a black line.

is worth noting that the method is very sensitive to the physical setup. If the camera is not properly aligned, it might provide less than satisfying results. We have, however, not experimented with this.

Finally, we note that in order for the user not having to do these calibrations multiple times, we store the calibration results in a file, which can then be loaded at a later point if desired.

4.4 Summary

In order to remove the radial distortion of the camera used in the setup, we computed the intrinsic camera parameters by grabbing multiple images of a 9x6 chessboard pattern which were provided as input to an OpenCV function. The resulting matrix was then used to undistort the images grabbed from there on.

The relationship between moving the camera a single step and the corresponding transformation between two images grabbed before and after the move respectively was estimated. By grabbing an image of the chessboard pattern, moving the camera a single step, and grabbing a new image of the pattern, the vector from the center point of the pattern on the second image to the center point of the pattern on the first image provided a usable estimate of the translation.

The above calibrations were done entirely by EvoBot, requiring only from the user that she put a chessboard pattern on the Plexiglas plate where it was visible to the camera.

Chapter 5

Extracting experiment data using computer vision techniques

Having a platform for conducting experiments requires a bridge between the scientist designing the experiment and the results from the experiments. It is the most noble of tasks for a platform such as ~~The~~ EvoBot to not only extract this data but also to aid the scientist in understanding what is actually found in her experiments. One of the tools for doing is to visually aid in finding areas of interest from the recorded experiments. To do this, EvoBot will facilitate droplet tracking.

The following will attempt to clarify the exact goals of this task, followed by an account of how the task is solved in the Splotbot and EvoBot platforms respectively. Finally a series of experiments are made to illustrate the success of this implementation along with a discussion of the results.

5.1 Goals

The goals in this regard are rather simple. ~~The~~ EvoBot must **show a live feed from a camera**, and **upon clicking an area of interest the user will see the object clearly marked**. ‘Object’ here refers to a patch of color similar to and surrounding the clicked point in the image. In actual terms, this should amount to a droplet in a biological experiment.

In addition to visually marking the object, **relevant information must be extracted from the droplets properties**. Given a lacking knowledge of which attributes are actually interesting we will limit ourselves to showing the

speed of the droplet and nothing else. Other properties could be the size of the droplet, the circularity, etc.

An important consideration aside from the features listed~~is~~ is that performance can be impacted by such a feature. Droplet tracking will operate on a live video feed~~and~~ and will impact each frame. This will likely cause degraded performance. It is difficult to put an exact number on how much degradation can be accepted, but it is important to ~~consider~~
keep in mind

5.2 Droplet detection in Splotbot

The droplet detection and tracking done in the Splotbot project is thoroughly covered in the thesis by Gutierrez (Gutiérrez 2012, 134–140). In the interest of readability of this report, the following is an attempt at describing the parts relevant for the EvoBot project.

Droplet tracking is a twofold process consisting of

- Finding the droplet to track
- ~~Track~~ the path of the droplet

The tracking part is done using an advanced AI technique known as Self Organizing Maps. For the EvoBot project a different approach is taken, and it is therefore out of scope to describe this part. The part about finding the droplet is, however, relevant and is described in the following.

In Splotbot, finding the droplet is based on the user clicking the object of interest before the experiments starts. Like EvoBot, Splotbot uses OpenCV for most if not all vision related functionality. Gutierrez considers three different segmentation methods for detection the droplet:

Bullet list?

Motion based, subtracting frames from each other and seeing shifts in color. However, this is shown to not be robust. Because subtle changes in lightning has big effects.

Shape based, that is, to consider the shape of droplet as the unique property of each droplet. Hough transform, a voting based system for finding circles, is considered to base the shape on the circularity measurements of objects. Is discarded because circularity is not guaranteed in the droplets.

Color based, that is, consider the color value of a pixel as the defining factor for a droplet. To compare against other pixels in the image, the distance in HSV space is measured (three dimensional vectors) and similarity is held against a threshold to determine whether it should be considered an object of interest. Color based segmentation is chosen as it proves to be most robust.

Kig på mine rettelser fra s

Aside from the segmentation itself morphology is performed on the image to remove noise. Output is a binary image for each tracked object. The binary images are finally joined together with an XOR operation.

5.3 Droplet detection in EvoBot

As described in the previous section, a lot of work in terms of droplet detection has been done in the Splotbot project. It makes sense to reuse as much as possible of the knowledge gained in that project. Careful considerations have gone into the choices of technology and algorithms of course, but the Splotbot report is seen as a credible source of information. With this in mind, droplet detection in EvoBot consists of the following steps

1. User input determines pixel of interest
2. The color of the selected pixel determines blob of interest
3. To better extract the blob of interest, noise is removed using a filter
4. A binary image is extracted based on color segmentation, not motion nor shape of the droplet
5. Morphology is applied to the binary image, to remove for ‘holes’ in the blob
6. Droplets matching the size and color of the droplet are selected
7. Of these the largest droplet is chosen as the one to be tracked.

This is done repeatedly on each frame, and results in a continuous tracking of the droplet.

5.3.1 Considerations on the choice of filter

As mentioned above, we filter the original image, this is done for two reasons. The first is to increase the likelihood of the algorithm detecting the entire droplet. The second is that we seek to achieve as uniform a color of the entire droplet as possible. This is what Gutiérrez (2012) achieves by filling the droplet with the same color. The theory in the case of EvoBot is that the same effect can be achieved using the correct blurring mechanism.

The effect of this is that the image is blurred before tracking takes place resulting in a more even color across the droplet as well as some elimination of noise. We considered three blurring algorithms, with the following characteristics:

- **Gaussian blur:** Fast, but not edge preserving, degrading the results
- **Bilateral filtering:** Edge preserving, but too slow and only partially removes noise

- **Median filter:** Relatively fast, edge preserving, removes noise very well

The Median filter was chosen based on the above criteria. A comparison showcasing the differences is done in section 5.4.3.

5.3.2 Considerations on the final choice of droplet

Our implementation can only track one droplet at a time, whereas Gutiérrez (2012) demonstrates that the Splotbot has the ability to track several. It can be argued that Evobot can be extended to provide this functionality as well. In the underlying code it is a matter of extending a single method to take a list of colors and return a list of droplets, however, the design as a whole is more tricky, as it requires some thought into the user interface design to make this clear to the user. It is considered out of scope for this project to look further into this.

Finder han flere droplets m

Forskellig farve

5.4 Testing the droplet detection

The following is an attempt at accounting for the success of the droplet detection implementation. This will be done based on the criteria **degradation of performance** and **quality of results**. Both of these will be judged on subjective measurements, but are documented based on data extracted from EvoBot. Lastly, an experiment is done to determine the difference in results of the different filters discussed.

5.4.1 Degradation of performance

This experiment serves the purpose of illustrating how big an impact on performance the implemented droplet detection has. This impact is to be understood as in terms of actual effect on what is seen by the user. This means that we have to deal with a comparison of time in the real world. For this we have chosen the straightforward approach of using a stopwatch and have designed the following steps for the experiments:

1. An artificial droplet is placed in sight of the camera
2. Also in sight of the camera is a stop watch
3. In the span of 30 seconds, a video is recorded on the camera
4. Droplet detection is turned on
5. Again, 30 seconds of video is recorded
6. The amount of frames captured in each video is compared

Note that this of course has an element of uncertainty, as the 30 second mark is as measured by us carrying out the experiment, leaving our reaction time as a

Nope. Det er ud fra billeder f

source of error.] The experiment is run on the BeagleBone Black itself for the most relevant results. An image of the setup can be seen in 5.1. [figure](#)



Figure 5.1: Experiment setup measuring droplet performance.
[tracking](#)

The results are 68 frames in the 30 second video without tracking and 38 with tracking. This means that the resulting impact from the droplet detection is approximately **44 % fewer frames**. Given that the nature of the experiments to be run on the EvoBot are usually slow moving, we find these results acceptable.

[But if experiment that update quicker must](#)

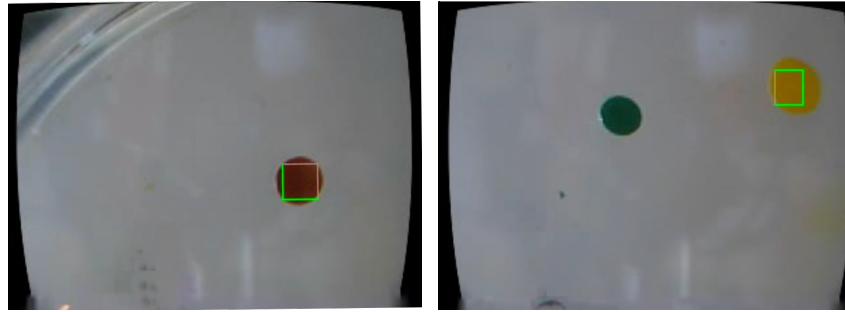
5.4.2 Quality of results

One of the important aspects of the droplet detection feature is of course the actual effects it can produce. As mentioned earlier, this will be done using subjective judgement. To best showcase the results we tried to mimic a real biological experiment to the best of our ability. This resulted in the following experiment:

1. A petri dish is filled with transparent cooking oil
2. In the cooking oil, 4 different colours of liquid ~~is~~ placed [are](#)

- the Plexiglas plate above**
3. The petri dish is placed on the camera
 4. The petri dish is slowly turned around its own center, moving droplets from the cameras point of view.
 5. While the droplets move, two of them are tracked individually as they move across the camera and in and out of the view

From this experiment we see a few interesting things. The first thing to mention is that the tracking works, as two colors can be tracked individually and as they move. This is somewhat poorly illustrated with images, but 5.2 gives an idea about the results. [figure](#)



(a) Tracking a red droplet. (b) Tracking a yellow droplet.

Figure 5.2: Successful droplet tracking.

The next thing to notice is that there are some effects of the implemented tracking that might strike some as odd. One of these effects is the behaviour when the droplets are leaving the screen. This results in the square marking the tracking staying in the camera view on the droplets last known position. This effect can be seen in figure 5.3a. The alternative solution to this would be to hide the box when no droplet is found, however, we do not see this as a negative effect, as it does give meaningful results (the droplets last known position). It would likely be a good idea to confirm this design decision with the users of the robotic platform.

There are also issues with the implemented tracking functionality. It is not robust in all cases, which can easily be illustrated with experiments. We are aware of this shortcoming, but it is important to keep in mind that the result of this project is a prototype showcasing the feasibility of the platform, and as such not all optimizations should be implemented. There are in general two problems, which are illustrated in figure 5.3b and 5.3c.

The first problem shows that the tracking can fail in certain ‘edge’ situations. This is illustrated with the droplet being actually at the edge, but really any change in light source can cause the problem. The cause of the issue is that we do not update the assigned color of the droplet over time. This means that the color extracted from the first clicked pixel is the color used for the droplet

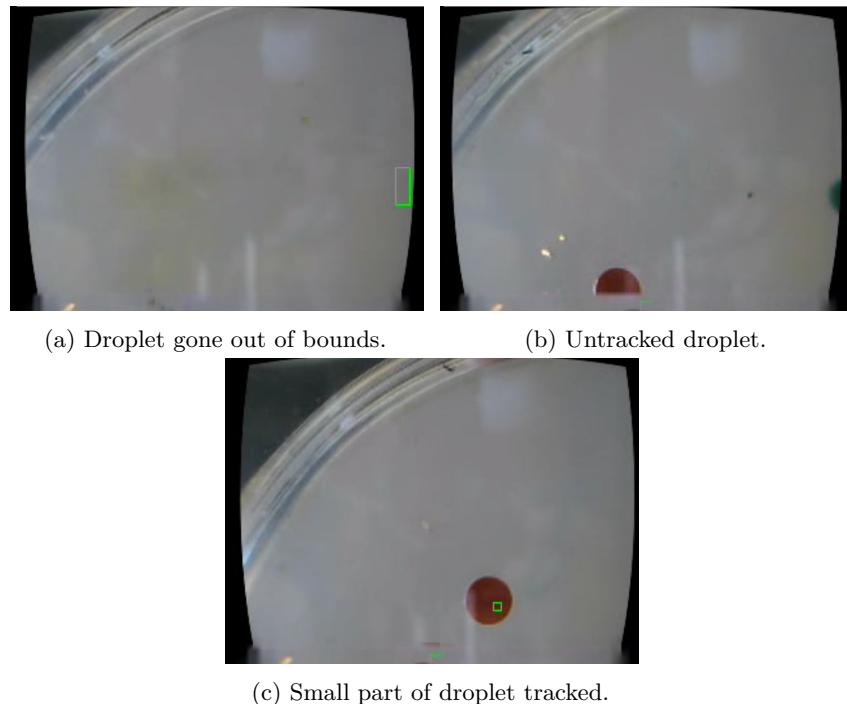


Figure 5.3: Edge cases and problems when trying to detect a droplet.

throughout the tracking. There are two ways of solving this. Gutiérrez (2012) solves the problem in software by constantly calculating the color of the droplet based on its most prominent color. Another way to solve it is by having a better test setup, where lightning does not have as big an effect.

The second problem shows that in some cases only a small part of the droplet gets tracked. This is again related to the color of the droplet and specifically how we make the color consistent across the droplet. As is further elaborated in 5.4.3 we use blurring for this purpose, whereas the approach taken in Gutiérrez (2012) is to actually fill the droplet with the same colour. There is again the solution of a better test setup, as the culprit is light differences.

5.4.3 Choice of filters

This experiments serves the purpose of comparing three blurring techniques to help decide which of them best serves the need of EvoBot. The considerations are performance along with the benefits of the specific technique. The desired benefits are:

- More even color across the droplet
- Elimination of noise

In order to test this, the following experiment has been run:

1. Use an image of two droplets in a petri dish
2. Add some noise on the image (using simple image manipulation software)
3. Apply each of the three filters to the image

Furthermore, the performance of each of the filters has been assessed. This is done with the actual droplet detection pipeline when the software is running on the BeagleBone Black. Before and after application of the filter, timestamps are retrieved, which are used to compute the running time. This is done with each of the three filters.

5.4a

The image used for the experiments and the results can be seen in figure 5.4. The image 5.4a was grabbed from a video of moving droplets. As mentioned, the salt and pepper noise is of course artificially added, and quite extensive. It is not of crucial importance that the chosen blurring technique will be able to remove all of this noise.

The Gaussian filter, image 5.4b, proved, as suspected, to be very fast at **16.4ms** on average. But it also gives rather poor results in terms of smoothing out the droplet colouring. It becomes very ‘grainy’ and not very even. It also does not remove the noise, but instead leaves a lot of spots in the droplet.

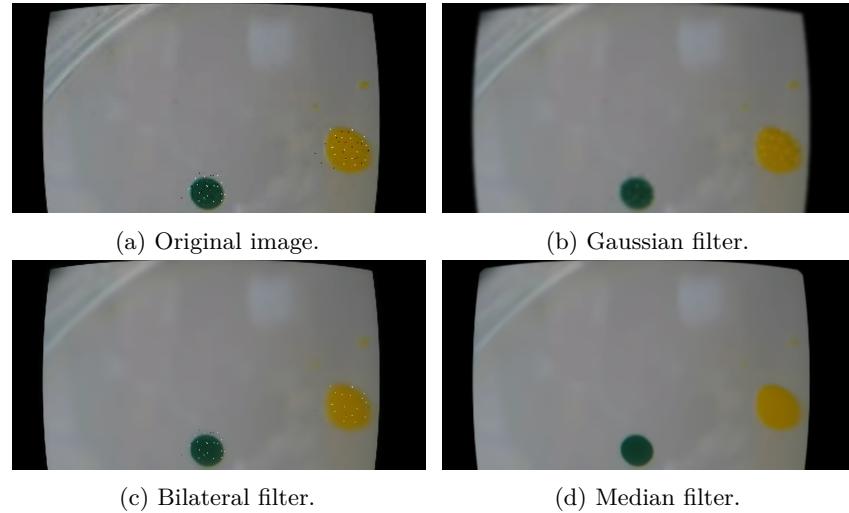


Figure 5.4: The resulting images of applying different filters to a noisy image.

The Bilateral filter, image 5.4c, was by far the slowest at **34358.4ms** on average. It does have nice results in terms of preserving the edges of the droplet. It appears to have only removed some of the noise in the image. The performance of the filter is **sadly** unacceptable. The performance cost is too much compared to the edge preservation achieved.

in

Lastly is the median filter, image 5.4d. Clocking at **331.6ms** on average and with very reasonable results, this is the filter that was chosen. The image shows that all noise is removed in one pass, and the droplets gets a very even and smooth color.

5.5 Summary

It is an essential feature of **the** EvoBot to provide visual aid in the form of droplet detection. This is a feature that is also present in the previous iteration, Splotbot. The relevant parts of the **Splotbot** report has been extracted and used as an offset of the design of droplet detection in EvoBot. This section has demonstrated the implementation with an emphasis on known shortcomings. Arguments have been posed for some of the design choices that have influenced the solution, and experiments have been showcased to give an idea about how well the solutions performs.

Måske lidt meget hvad vi har gjort i stedet for resultater, hvis jeg husker

Jeg ved ikke hvørordan du husker mig, men du har ret. Der er en forkert
vinkling i "This section has bla bla". Og ja, vi bør konkludere noget ala
at vi har ledt efter og fundet en god blurring løsning der giver de rigtige resultater i accepta-

Ladet. Udskift med
"however entirely"

Tror ikke Splotbot
har en rapport

Chapter 6

Getting an overview of large surface areas

In this section we look at an issue introduced, when the experiments to be performed on the EvoBot have a large surface area e.g. when doing experiments in a petri dish with a diameter of 14cm. This becomes a problem, when an area must be observed which is larger than what can be captured in a single image by the camera due to its limited field-of-view.

We first introduce our goals concerning the robotic platform regarding this. We then account for how we have tried to achieve these goals, along with the end results and a discussion of these.

6.1 Goals

The goals of this chapter stem from two properties of some of the experiments that the EvoBot must run:

1. Some experiments cover a large surface area
2. The experiments are usually slow moving

When we combine these with the fact that the camera of the current setup has a very limited field-of-view due to the attached zoom lens. This introduces difficulties to overcome, but also leaves room for them to be solved. The camera is movable, so it is possible to cover a larger surface area by moving the camera to different position, grabbing images, and combining the images to a single image. This is made possible due to the experiments being slow moving, leaving time for the camera to be moved and grab the images, without the experiments changing

much in the meantime. If the slow moving characteristic was not present the images grabbed would be inconsistent.

These considerations can be summed up in two goals for the EvoBot. We wish to combine the camera with the mobility of the bottom carriage to **automatically grab multiple images and stitch them together**, forming a single image of a large surface area. Furthermore, we wish to achieve this in **the shortest possible time**, in order to minimize the inconsistency between the images while also allowing as quick as possible feedback to the user.

It is worth noting that one stakeholder has shown interest in using other kinds of cameras/scanners, such as an OCT scanner, which takes three-dimensional images each covering a surface area of about 1cm^2 (“Wikipedia: Optical Coherence Tomography” 2014). Due to the scope of this project, we will only work with two-dimensional images, and we will use the camera also used in the Splotbot setup, as described in chapter 2.

6.2 Scanning pipeline

At this point in the development of the EvoBot, we already had separate components for the camera and for the set of x and y axes controlling the bottom carriage. We wished to reuse these as much as possible, avoiding doing the same work multiple times. One possibility was to add the logic concerning the scanning by image stitching to one of these components. But the responsibility does not conceptually reside in any of these components alone. Instead, we decided to introduce as **Scanner** component with references to the existing **Camera** and **XYAxes** components, encapsulating this new scanning logic in a separate component.

The scanning pipeline itself is quite simple. It consists of the following steps:

1. Input is the start and end positions of the camera, the step size between each image grabbed, the duration to sleep after each move of the camera before grabbing the next image, and the stitching algorithm to use.
2. The camera is first moved to the start position, where the first image is grabbed.
3. The camera is then moved to each position between the start and end positions defined by a rectangular grid with the given step size between each point, ending on the end position, grabbing an image at each point.
4. Each image is stored together with the location at which the image is grabbed.
5. The images are stitched together.

The sleep time allows for making sure the camera is moved correctly before grabbing the image. This is necessary due to the interactions with the stepper motors being asynchronous as described in chapter ??.

The following is an example of such a scanning:

1. The input provided is:
 - Start position: (10, 10)
 - End position: (20, 25)
 - Step size: 5
 - Sleep time before grabbing images: 1000ms
 - The algorithm is irrelevant in this case
2. The camera is moved to position (10, 10), where the first image is grabbed.
3. The camera is then moved to the positions (10, 15), (10, 20), (10, 25), (15, 10), (15, 15), (15, 20), (15, 25), (20, 10), (20, 15), (20, 20), (20, 25), where the rest of the images are grabbed.
4. Each image is stored with the corresponding location, so the first image is stored with (10, 10), the next with (10, 15), and so on.
5. The images get stitched together.

For this project we have considered three different algorithms for doing the actual stitching of the images:

1. Stitching based on image features
2. Stitching based on the position at which each of the images are grabbed
3. A combination of the first two, stitching based on both image features and position

We have implemented the first two algorithms on the EvoBot, and played around with implementing the third without finding time to complete the implementation. In the following sections we explain our implementations of each of these algorithms. For each of the algorithms described, we assume that the camera is calibrated and the images grabbed have been corrected for radial distortion as explained in chapter 4.

6.3 Stitching images based on image features

Stitching based on image features is in itself a pipeline consisting of many steps. A simple example of such a pipeline could be (Solem 2012, 91–100):

1. Input is a list of the images to stitch together
2. For each of the input images, find a number of features of interest. Such features are areas on each image that are easy to recognize
3. Compare the interest points of each image with the interest points of the other images, finding points that occur in multiple images

4. Use the points detected in multiple images to estimate how these images relate to each other in terms of position such as one image being slightly translated in the x direction compared to another image
5. Based on the obtained knowledge of the relationship between the images, transform all the images into a larger combined image which is the final result

In our case we decided to use an existing image stitching implementation found in OpenCV. This implements some version of the pipeline outlined above along with several other improvements such as image scaling, exposure correction, and blending of the images (“OpenCV Stitching Pipeline” 2014). Being part of OpenCV, we expect the implementation to be robust and tested.

There are also downsides to using a generic image stitching implementation. The first issue is concerning the performance of the algorithm. Every image is compared with every other image (though the use of algorithms such as RANSAC improves on this problem (Solem 2012, 92–98)), which in practice means that for every extra image to stitch together, the result is a large increase in the runtime of the algorithm.

The reason for this complexity is that the algorithm assumes no prior knowledge about the images given as input. But we know for each image the position at which the image was grabbed, and from the camera calibration (chapter ??) we know how these positions correspond to translations in the images. We use this knowledge in the stitching algorithm described in the next section.

6.4 Stitching images based on position

For the stitching algorithm based on image position we make a number of assumptions:

- When moving the camera in the x direction, this corresponds to a simple image translation in the x direction. The same applies to the y direction. This means that there is not rotation or scaling involved with the transformation. This imposes the requirement on the physical camera setup that the camera is aligned with the axes of the robotic platform.
- We know from the camera calibration the image translations corresponding to a step with the stepper motors in each direction.
- Each step with the stepper motors result in the same distance travelled.
- All images have the same size.

Based on these assumptions, we put forth the following stitching pipeline:

1. Input are the images to stitch, the positions at which they were grabbed, and the camera calibration linking camera movement with pixel translations

2. Search through all the positions, finding the min and max values of both the x and y coordinates
3. Get the width and height of the images
4. Based on this information, compute the size of the resulting image and create a black image of this size
5. For each image, compute the transformation matrix defining the transformation from the image to the resulting image
6. Use the transformation matrices to warp each image onto the resulting image

Several of these steps concern some kind of computation. The following provides an example of the stitching pipeline applied, also showing the calculations involved:

1. Input is four images grabbed at the positions (10, 10), (10, 15), (15, 10), and (15, 15), and the calibration results that a step in the x direction results in a translation of 20 pixels in the x direction and that a step in the y direction results in a translation of 20 pixels in the y direction:

$$\Delta x = 20$$

$$\Delta y = 20$$

2. The min and max values are computed:

$$\min X = 10$$

$$\min Y = 10$$

$$\max X = 15$$

$$\max Y = 15$$

3. The width and height of the images are retrieved:

$$width = 320$$

$$height = 240$$

4. The size of the resulting image is computed:

$$resultWidth = width + (maxX - minX) * \Delta x = 320 + (15 - 10) * 20 = 420$$

$$resultHeight = height + (maxY - minY) * \Delta y = 240 + (15 - 10) * 20 = 340$$

5. The transformation matrix for each image is computed. These matrices have the following form, where t_x and t_y are the translations in the x and y directions respectively (Paulsen and Moeslund 2012, 134–137):

$$\begin{bmatrix} 1 & 1 & t_x \\ 1 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

For the image at position (x, y) , we have that:

$$t_x = (x - \min_x) * \Delta x$$

$$t_y = (y - \min_y) * \Delta y$$

(This calculation actually depends on whether the translations are positive or negative, but for this example the above is sufficient). For this specific example, the result is the following values:

- (10, 10): $t_x = 0, t_y = 0$
- (10, 15): $t_x = 0, t_y = 100$
- (15, 10): $t_x = 100, t_y = 0$
- (15, 15): $t_x = 100, t_y = 100$

6. These transformation matrices are used to warp each image onto the resulting image.

In order to achieve the warping in the final step, we do the following:

1. We warp the image onto a temporary, black image of the same size as the resulting image.
2. We then threshold the image with a threshold of 1. A thresholding results in a binary image with all values below the given threshold being black and the remaining pixels being white (Paulsen and Moeslund 2012, 51–52).
3. We then use this binary image as a mask of the area to which we are to apply the image to the resulting image.

As we have experienced that the edge of each image did not look very nice when warped using this approach, we apply a morphological dilation on the mask before we use it, resulting in a bit of the image being warped being removed. The entire warping process is illustrated in figure 6.1. One important thing to note about this process is the blueish color seen on 6.1g. This color distortion appears once in a while, and is an unfortunate result which may lead a scientist to draw false conclusions. We have not to a sufficient degree investigated the cause of this issue, but we believe to be a problem with the threshold in the warping process. A later project should investigate this further.

The advantage of this stitching algorithm is that adding an image extra to stitch results in a constant number of added operations, making the runtime linear. The major disadvantage is the dependency upon the correctness of the assumptions. If the camera calibration is not precise, the stitching is equally imprecise. The error is accumulating, as the error is repeated for each step the camera is moved. This, however, is not visible in the image, as the error is constant between the images next to each other. Furthermore, if the camera is not aligned with the axes of the robotic platform, the algorithm does not produce a correct result.

In order to remove this sensitivity to calibration precision, we have considered a third algorithm described in the following section.

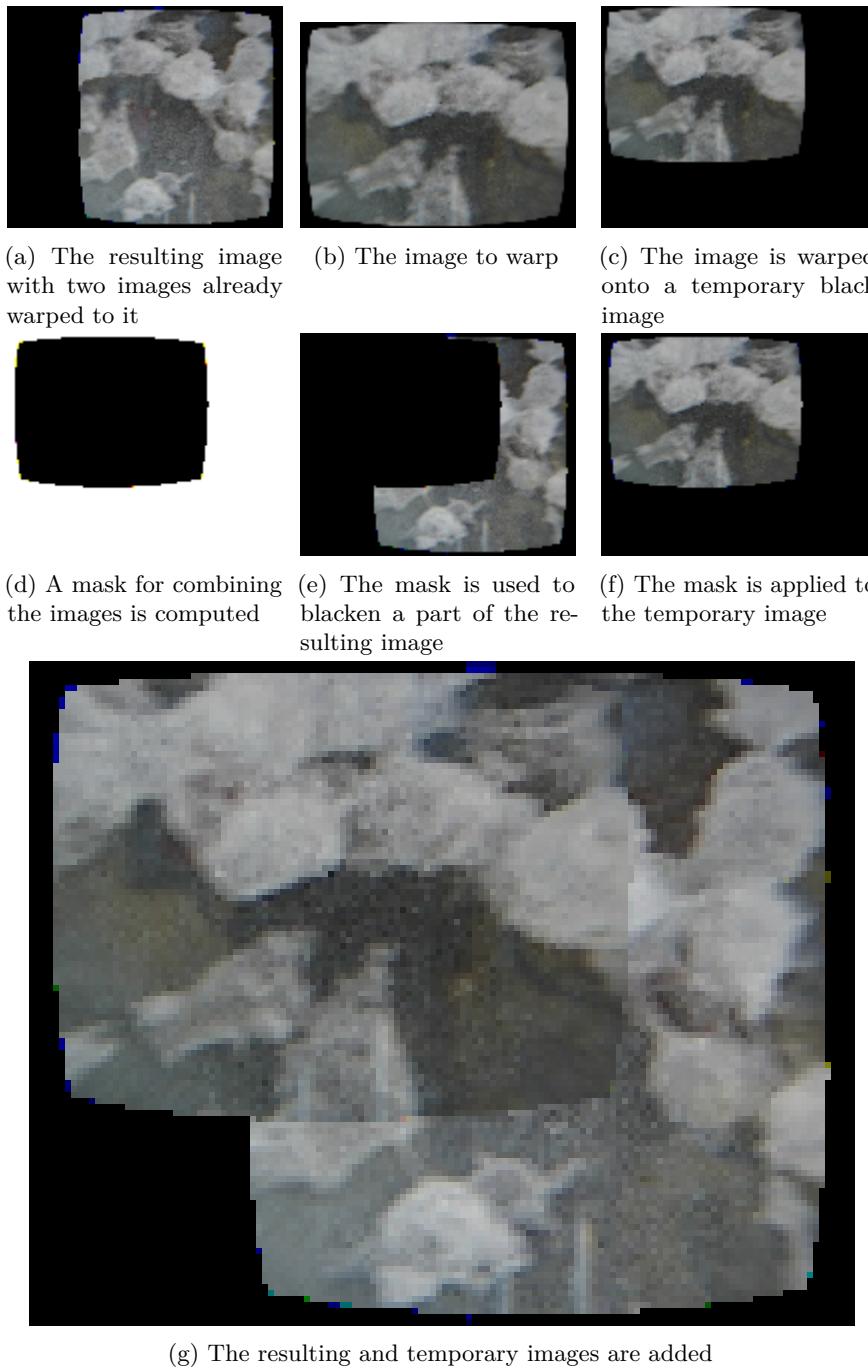


Figure 6.1: The process of warping an image onto the resulting image.

6.5 Stitching images based on both image features and position

We have considered adding an algorithm combining the advantages of the above two algorithms by stitching the images based on both image features and the position at which the images were grabbed, but we have not found the time to make a complete implementation.

This algorithm is best seen as an extension on the algorithm based solely on image features. But rather than the interest points of each image being compared with the interest points of every other image, we use the fact that we know which images are next to each other to only compare each image with its neighbours. In theory this reduces the complexity of the algorithm, due to fewer comparisons having to be made for each image to be stitched.

Our idea was to compute these regions of interest based on the same warping calculations done in the position based image stitching algorithm. These could provide the areas in which the images overlap. But we found that this approach had the inherent difficulties that when images are stitched together using the OpenCV image stitching implementation, the resulting image is sometimes scaled, sheared, or something else, resulting in the relationship between the resulting image and the positions at which the images are grabbed becoming unknown.

We played with the thought about implementing our own features based stitching algorithm which could take these things into account, but due to the limited scope of the project we did not find the time. We are certain that improvements to both the image stitching results and the runtime can be achieved, and it would therefore be interesting for another project to pick up this challenge.

In the following section, we compare the two implemented image stitching algorithms in terms of both results and performance.

6.6 Comparing the image stitching algorithms

We have compared the two implemented image stitching algorithms in terms of the quality of the resulting images and the performance. The comparison is aided by the running of a series of experiments providing simple benchmarks. For each experiment, we do the following:

- Load the images to stitch
- Get the current time
- Stitch the images together, storing the resulting image
- Get the current time again
- Compute the runtime of the algorithm based on the times saved

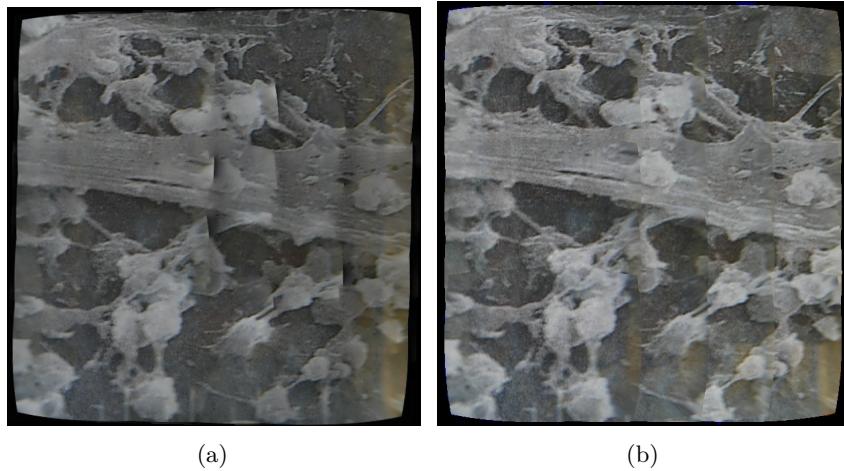


Figure 6.2: The resulting images of stitching together 20 images of the biofilm print using (a) the features based algorithm and (b) the position based algorithm.

Each image is run both on the BeagleBone Black on EvoBot and on a reference laptop computer, allowing us to assess the feasibility of the algorithm being used both in the current setup and in a setup with a more powerful computer. The reference computer has 16GB of ram and a dual core Intel i5-3320M CPU @ 2.60Ghz - 3.3Ghz.

6.6.1 Resulting images

For the experiments we have two different scenes of which we have grabbed a number of images, which we then stitch together. The first is a printed image of a close up view of biofilm (“Wikipedia: Biofilm” 2014). The second is a large petri dish with regular kitchen oil, droplets from food colour, and a white background. For the sake of brevity we have selected a few resulting images highlighting the differences between the algorithms.

The first images are the result of stitching 20 images of the biofilm print together with the two implemented algorithms respectively. These are shown in figure 6.2.

We consider both of these images good results, as both combine the input images in a realistic manner. The features based algorithm appears to include some of the black edges in the images in some locations, while other locations are blended extremely nicely, making the transition almost invisible. The position based algorithm stitches very evenly throughout, with the same slight lack of precision at each transition between images.

We next consider the case of stitching together only the first two of the previous

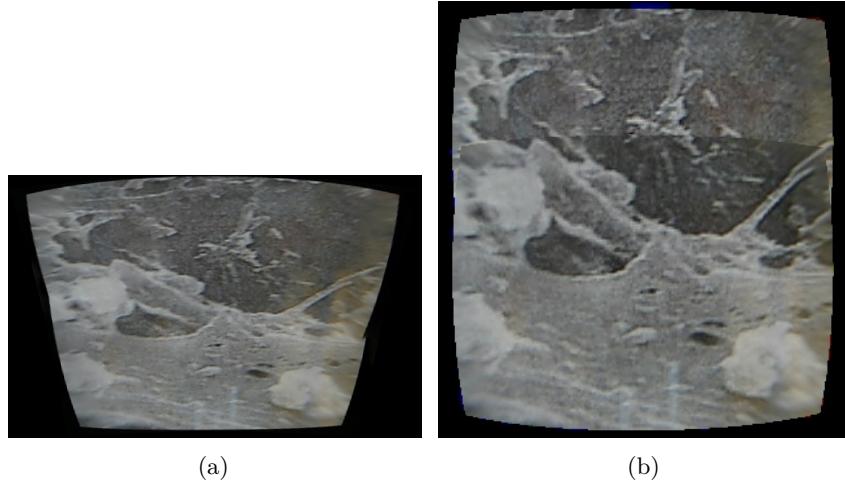


Figure 6.3: The resulting images of stitching together two images of the biofilm print using (a) the features based algorithm and (b) the position based algorithm.

20 images. The results are depicted in figure 6.3.

The position based algorithm has stitched these images together in exactly the same manner as with the 20 images. But the features based algorithm has altered the shape of the image, resulting in a less realistic depiction of the actual scene.

Finally, we consider the other scene of the petri dish with colored droplets. We only successfully stitched together the image with the position based algorithm, as the feature based algorithm is not capable of finding enough similar interest points on the images to produce a result. The resulting image from the position based algorithm is shown in figure 6.4.

Based on these results we conclude that even though the features based stitching algorithm is capable of creating the most invisible transitions between the images in some cases, the fact the it other cases creates unwanted results or even is not capable of producing a result make the position based stitching algorithm seem most suitable for the application.

6.6.2 Performance

For testing the performance of the two algorithms we have done a number of image stitchings of the print of biofilm. We have grabbed 45 images with a step size of five between them, which can be stitched together to a full image. We have then run each of the algorithms on subsets of these images of different lengths (2, 4, 9, 12, and so on), making sure the subset can be stitched together.

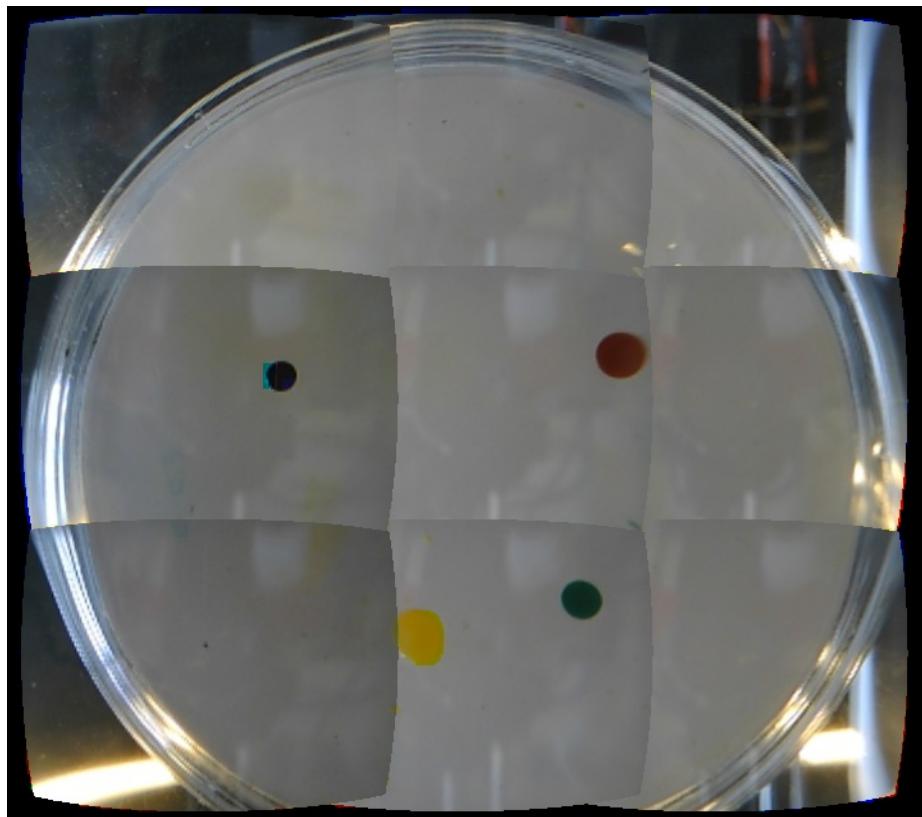


Figure 6.4: The resulting images of stitching together nine images of a petri dish with colored droplets using the position based algorithm.

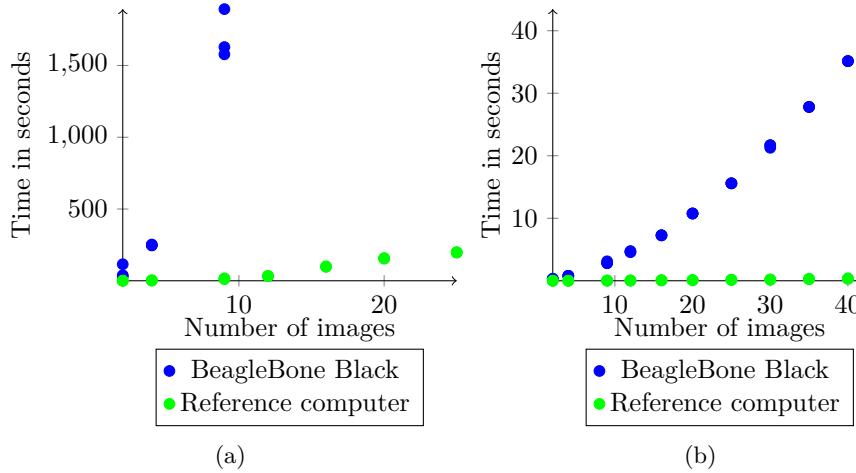


Figure 6.5: The run times of the stitching algorithms: (a) features based (b) position based

For each run, we note the run time. We repeat each stitching three times. The results are shown in figure 6.5.

If we start by looking at the features based stitching algorithm, it is clear that already at a low number of images (in this case nine) it becomes infeasible to use this algorithm on the BeagleBone Black. On the reference computer it is better, but the runtime still increases a lot for each image added. This is also the reason why we have stopped at stitching 25 images. It simply takes too long to stitch more images than that, and already at 25 image the stitching takes more than three minutes, making it difficult to achieve the goal of fast feedback. Another thing to note is that the algorithm includes a random element, meaning that stitching the same images can result in varying run times.

The position based algorithm has a much shorter runtime. Even on the BeagleBone Black, stitching the highest number of images tested in this experiment (45 images) is below 45 seconds. Also, from the graph the complexity appears to be linearithmic, though we have no proof of this. On the reference computer, even stitching 45 images has a run time of below 0.5 seconds. Also, the run times for this algorithm are much more stable than the case of the features based algorithm.

In terms of performance, the features based image stitching algorithm does not appear to be sufficient to be used on the EvoBot, if the goals set forth in the beginning of this chapter are to be reached. This appears to be the case even though a more powerful computer than the BeagleBone Black is used. But the position based algorithm do appear to be sufficient, also to be run on the BeagleBone Black, though stitching many more images than the 45 tested will certainly make the stitching somewhat slower. Replacing the BeagleBone Black

with are more powerful computer would have a great performance gains, allowing for much quicker user feedback.

6.7 Summary

Some experiments to be run on the EvoBot cover a larger area than what can be covered by a single image from the camera of the robotic platform. For solving this, a scanning pipeline was implemented, where multiple images are automatically grabbed and stitching together to a single image.

For the stitching, three algorithms were suggested which stitch the images based on image features, the positions at which the images were grabbed, or a combination of the two respectively. We have made working implementation of the first two.

Finally we compared the two implemented algortihms in terms of both the resulting images and the performance. This was based on experiments run directly on the BeagleBone Black and on a reference laptop computer. The position based algorithms was found most suitable for the application, as it provided stable results and the by far best performance. The performance could be greatly enhanced by using a more powerful computer than the BeagleBone Black.

Chapter 7

Autonomous interaction with experiments

Some experiments to be run on EvoBot are currently run manually by a person monitoring the experiment and upon some event modifying the experiment. EvoBot must be able to run such experiments autonomously. Autonomous is to be understood in the sense that the experiments will be started by a human but will be completely self sustained and be capable of actively reacting to changes happening in the observed experiment. This is the focus of this chapter.

We first introduce the our goals in this regard before looking at each part of the system, discussing how the goals are fulfilled. Finally we look at the result of the approach including a discussion of possible improvements and alternative solutions.

7.1 Goals

The idea behind an autonomous experiment is to define an experiment execution that loops and continuously adjusts over time. This is illustrated in figure 7.1. The loop begins with the initial actions that will be executed setting up the experiment. This initial setup could for instance include picking up liquid and putting it into a petri dish. After the initial actions the experiment is running, and components detect events such as droplets moving. Based on events in the system the robot then reacts on the on these events and their corresponding data, possibly doing new actions which changes the experiment. This loop creates a reactive feedback process where the outcome of the experiment is based on data gathered throughout the experiment.

Making a user capable of defining experiments in this system requires a few capabilities added to the robot. First of all, **the user must be able to send**

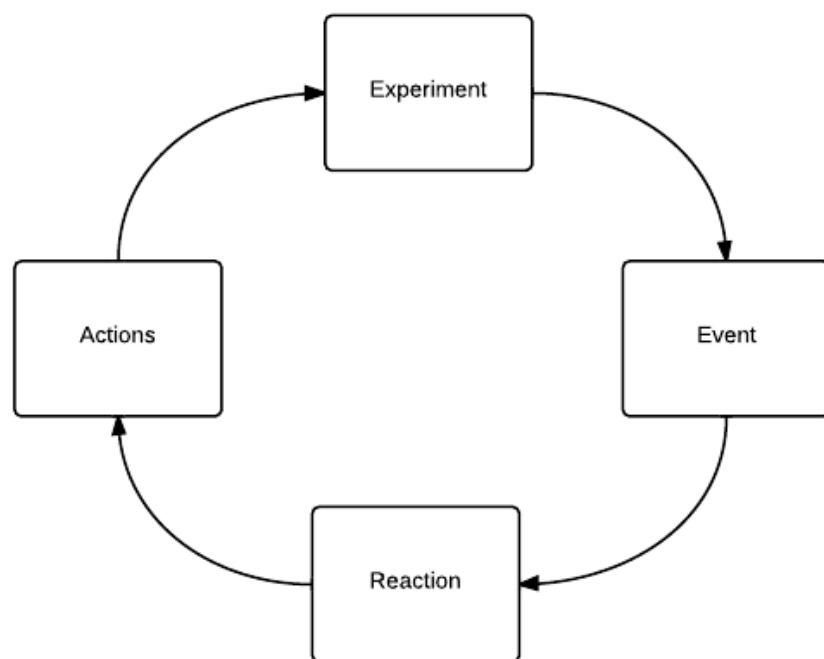


Figure 7.1: Reactive experiment loop.

Det virker liiidt som om
at vi forcerer fed-skrifts
formatet.

instructions to components. Secondly, the components **must be able to communicate events with data.** Lastly, **it must be possible for the events emitted by the components to influence further execution of the experiment based on definitions by the user.**

7.2 Executing instructions on EvoBot

When running experiments on EvoBot a core feature is the ability to send instructions to the individual components to make the robot perform different actions. We want a system that allows us to easily enqueue instructions and then get them executed sequentially making the robot carry out actions in order. We also want instructions to be capable of taking parameters, that is to allow us to instruct a component to ~~be based on some parameters~~ based on given parameters perform a certain action e.g. moving a motor to a certain position.

At the core of our instruction execution model lies two important data structures, the instruction buffer and the action list. The instruction buffer is a thread safe FIFO queue that contains integer instructions and integer arguments. The point is that new instructions gets added to the instruction buffer, which then gets emptied by a worker thread, executing every instruction. In this system we define instructions as integers being the index of the actions in the action list. The action list is a list of functions that takes a pointer to the instruction buffer. When an action gets invoked with the instruction buffer it starts by popping the wanted arguments from the instruction buffer, calling a method of a components ~~with the arguments~~ with the arguments. This cycle will continue as new instructions are put into the buffer and popped and executed. An illustration of this is given in figure 7.2. To ensure that the instructions are run in sequential order there is only a single worker thread. If a component wants something executed in parallel it has the responsibility of spawning a new thread.

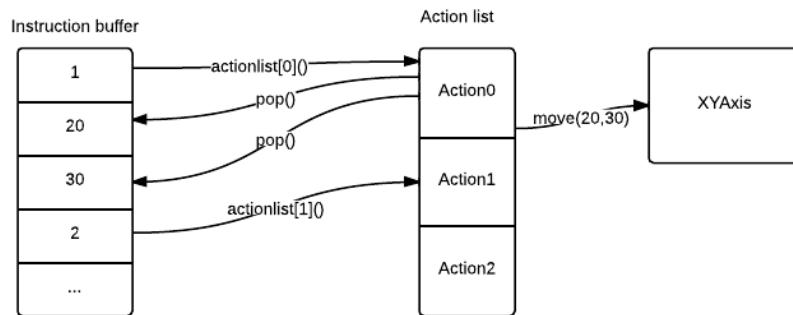


Figure 7.2: Instruction buffer and action execution cycle.

The actions in the action list are registered by each component. On the startup of EvoBot software, the configuration file is loaded and every component is initialized as described in chapter 3. When all components have been initialized, the software loops over them asking them to register their actions in the action list. The placement of a specific action in the action list is therefore determined by the order of the components in the configuration file, as well as how many actions each component has and also in which order a component has defined its actions. The code defining the logic for an action can be found in the specific component.

With the instruction buffer and the action list as described above we achieve the instruction execution as initially wanted. While this implementation fully supports the executing model, an observation can be made that we only support integers as arguments. A possibly useful extension could be to extend the engine to be capable of holding other data types such as strings, floats and doubles. Floats and doubles could fit into the current model as they could just be put into the buffer as one or more integers (depending on bit-lengths) and later be converted back into their desired type by the action call as they get popped. Strings of an undefined length would be harder and more inefficient to fit into the buffer, here we propose the possibility of storing a pointer to the string as an integer on the stack, making it possible for the action call to access the string at a later point by converting the integer to the pointer. Making more data types available would potentially make some component methods richer with more options. While we did not encounter such needs in our limited implementation it might be a desired feature at a later stage.

Ved godt det er mig der har skrevet det, men er lidt i tvivl om de her guidning afsnit til sidst er fede

~~The solution at this point allows for arbitrary user defined experiments, where a user can define a set of instructions that can make the robot do something. This does not however allow the user to get any feedback from the system, nor does it allow the user to define autonomous experiments.~~

7.3 Sending events with experiment data

At this point we can execute instructions and thereby make the robot capable of performing a user defined set of actions. Now we want to extend the model to enrich the communication possibilities of components. We want to make them capable of informing the user and the rest of the system of their state and possible discoveries in the experiment. To do this we want to make a system where components can announce that something have happened and include some data. The data attached to such announcements must be propagated to the user. An appropriate design for such a system seems to be an event based one. Vend sætningen om: An event based system seems appropriate for this situation.

To extend EvoBot to include an event system, we introduced the event function. This is a single function that every component calls when wanting to communicate an event. We store a this function in the `Splotbot` (main) class in a variable,

Yo crocodog, Jeg kan ikke se om fonten er rigtig her

making it possible to change the function at a later stage. Every component gets initialized with a pointer to the function, allowing them to call the function when needed and to ensure that updating the event function will propagate to the components. Our event system therefore allows us to bind an event function such that we can send data to the client etc.

Sending data required us to define what data we wanted to send for each event. We wanted the system to be universal so it could be used it to both propagate values such as droplet speed, as well as images from our camera to the client through the event system. The data is for flexibility purposes defined as a single string, making it the responsibility of the receiver to parser the string and handle the data.

her

The event system allows EvoBot to send data to the user, making **them** capable seeing images, numeric data etc. This however does not allow the user to make the robot react to the data, as the only way to extended the event binding is through modification of the C++ source code.

7.4 Programming experiments

Giving the users the option of defining a complete experiment required us to look back to figure 7.1. Here we could see that firstly the user must be able to control the loop. More specifically the user must be able to define the initial experiment instructions, **the** which events to react to, and the reactive instructions that must be fired off based on the event output. Only the initial instructions are already available by sending instructions codes (integers).

Omskriv til noget ala, "In an effort to make this design more user friendly"

But we wanted to make it more user friendly. The way to achieve this user friendliness is to make the language for the robot interaction closer to human language by introducing a programming language to run on the robot. Being software developers and therefore familiar with programming languages has the advantage of us being capable of designing and implementing such a language in reasonably short time, making it an attractive design decision. **from a development point of view.**

To do this we must first address the question of what language and why. An initial thought could be to use one of many available language interpreters for C++. An interpreter does however not fit our model exactly. An issue with using an interpreter is that suddenly we have to support an entire programming language with all of its features in our model and what capabilities it might have, so instead it seems natural to look at making a small and suitable DSL.

domain specific language, DSL

7.4.1 Rucola

Introducing Robotic Universal Control Language, or Rucola in short. The language was designed to fit our exact needs. It was built around two core

Det er vi vidst sluppet
okay fra. Godt arbejde :)

```

a = 20
b = a - 10

if(a > b){
...
} else {
...
}

print "Variables" (a,b)

```

Figure 7.3: Rucola language constructs.

features: Calling component actions and listening for events. The Rucola compiler exists as a sub folder in the main C++ code and functions largely as a black box from the point of the view of the rest of the software. It provides an interface for compiling a string of Rucola code into integer instructions as well as for invocation of events, resulting in further integer instructions. Within the Rucola compiler, the string is used to create an abstract syntax tree (AST), which is compiled into instruction code. We use Flex (“flex: The Fast Lexical Analyzer” 2014) as lexer and Bison as parser (“Bison - GNU Parser Generator” 2014) generators.

The basic language features of Rucola consists of integer arithmetics, variables, conditionals, and a print statement. The variable binding is global and is stored internally in Rucola, even between compilation and event calls. The variables can be reset by compiling with an empty string. The arithmetics, conditionals, and variable access happens on compilation time from Rucola to instruction codes. The print statement is for debugging purposes and is executed at compile time, so it is mostly useful for printing variables, therefore it simply takes a string to be printed and a tuple of expressions. An example of the different constructs can be seen in figure 7.3.

Calling component actions are somewhat trivial in our current model. What we do is that we first make every component register its actions in a map, where the key is the actions name and the value is a struct containing the instruction number of the action and how many arguments it takes. We denote this the action map. We then store the action map in another map where the keys are the component names. When a specific component is called with a specific action and arguments, we simply retrieve the instruction number of the action and the amount of arguments it takes via the map structure. We can check that it is both a valid component and action as well as if the amount of arguments is correct. Finally we translate the call into the action number followed by the arguments and include it in the list of instructions that we compile to. An example of a component action call is shown in figure 7.4.

and bindings

Component . action (1 ,2)

Figure 7.4: Rucola component action call.

(event: arg1 , arg2) -> { ... }

Figure 7.5: Rucola event binding.

Event callbacks are less trivial, but with a few adjustments to our model we can incorporate them into our system. Here we strike a boundary where we combine interpretation with compilation. On the time of initially compiling the code, we do not follow the event branches in the AST. We instead save them in a map from event name to the AST. As part of setting the event callback up in EvoBot, we include a call to the Rucola event compilation. This means that all events are received by Rucola before they are further emitted to the user. If a called event is bound in Rucola, we compile it into instructions and add them to the instruction buffer. If the event is not bound, no instructions are added.

For events to be really reactive we must also introduce event arguments. In a system where everything is integers, event arguments are of course also integers. Our approach to introducing this into the EvoBot event model is to extend the amount of arguments an event in EvoBot takes, from event name and data to also include a list of integers. The integers are bound to variables by the user at the time of the event call. This is done by taking the integer list and binding the integers to names in the internal variable map. It is up to the user to decide the names and the amount of variables she wants bound to variables. Figure 7.5 shows an example of the code of an event binding.

Our current implementation of the language faces a few minor issues that could be resolved. One of them is that the else branch in the if-else statement must contain some code to be valid Rucola code in the parser. Another nice to have feature would be to change event arguments to not be included in the global scope, to make them only available for the event itself unless assigned to a global variable.

7.5 Event reaction experiment

For the event based reactive system on the EvoBot we designed a small experiment to test the reaction time of EvoBot. The experiment is designed to measure the time that goes between something happening in the physical world until the EvoBot have processed and reacted on it. The experiment is as follows:

1. An event designed to watch the droplet speed is bound in the EvoBot

```

servoPos = 0

(Camera_dropletspeed: speed) -> {
    if (speed > 10) {
        Servo1.setPosition(servoPos)

        if (servoPos == 0) {
            servoPos = 90
        } else {
            servoPos = 0
        }
    } else { a = 0 }
}

```

Figure 7.6: Rucola test program for the event reaction experiment.

system using Rucola

2. When the speed of the droplet gets above a certain threshold it will move the servo motors
3. The camera is moved to simulate droplet speed above the threshold
4. The time from the actual movement of the camera to the movement of the servo motor is recorded. This is done with a physical stop watch. The precision of the timings are limited by this.

The experiment was repeated 21 times. The Rucola test program is shown in figure 7.6.

The experiment shows an average of about **6 seconds** in reaction time from a droplet movement detected to movement of the servo motors. The times recorded spans from around **5 to 8 seconds**, which is somewhat stable. For the kind of slow droplet experiments ~~the~~ EvoBot is expected to handle, we do believe these times would be acceptable for experiments, but it would likely cause issues, if experiments changing more quickly are run.

7.6 Summary

In order to make EvoBot capable of running autonomous experiments it was necessary to extend the EvoBot software with an event based system. The resulting software is capable of both executing instructions and emitting events with data. In order no make use of these capabilities, while also providing a way for the user to define experiments, a domain specific language named Rucola was created, ~~which~~ is executed directly on EvoBot. ~~This~~ allows the user to define

Rucola

, and allows

an entire reactive and autonomous experiments that can be run on the robotic platform.

Chapter 8

Logging experiment data

An important part of creating a platform for running experiments is to ensure that the user can understand the outcome of the experiment after it has been run. This chapter deals with the challenge and the implementation of a logging system for the EvoBot, making sure that the EvoBot will provide the user with the wanted data in the aftermath of an experiment.

We first introduce the goals of the logging system. We then look at exactly what experiment data is and how we log it. Then we look at how we structure the data for future analysis. Finally we discuss the issues faced with the limited hard disk space on the EvoBot.

8.1 Goals

Biological experiments usually involve designing, running, and analysing the result data of experiments. We set forth the goal of supporting every aspect of running an experiments. Running an experiment must produce data which can be analysed at a later point, making gathering data an important feature of EvoBot. We therefore want that **all data produced by an experiment is saved**. In order to make it easier to analyse the data afterwards, we want to make sure that **the data is structured in a way that makes it possible to analyse it**, and following on this that **the data is stored in formats that the user can load into software that can help with the analysis process**.

Making a tool for supporting the analysis process is **an other** issue which is not in scope of this project, but the logging of data is core to the EvoBot so we **wanted** to build it into **our solution**.

found it necessary the prototype

8.2 What is experiment data

An important part of our logging system is to determine exactly what data and information to log. We look specifically at logging as much data as possible. Not only experiment data but also data about the state of EvoBot in general. We have broken the different types of data into four categories, all important to get a total view of an experiment:

- Visible image data such as single images and video
- Data gathered through use of computer vision techniques carriages
- Physical behaviour on the robot, such as movement of the xy axis and servo motors.
- Software meta data, such as component initialization etc. This data is useful for debugging the robot at a later point.

An important part of our view of data in the EvoBot software is that we want to split data up on component type and component name. This means that the user can filter out a specific component or the components of a specific type, looking only at the data relevant to the user.

Much of the data gathered is image and video data, which must therefore also be logged. This makes it possible to recall exactly what occurred during the experiment.

8.3 Logging the data

Logging all the experiment data in EvoBot required us to build it into the core C++ program. We want the model to support the different target formats that we use now, and also support the possibilities of extending it in the future. Therefore it was natural to build the logging framework as a class structure, shown in figure 8.1. In our model we create an instance of the loggers needed for every component, allowing it to save the data that it uses to a supported format. A logger therefore is created with a specific component type and component name. The entry class is then used to create a new entry for saving and is parameterised with the specific type of data that needs to be saved. A logger takes a specific kind of entry and persists it.

In our current implementation we support three different logger types for logging video, images, and files respectively. Each of the loggers have different target formats and will treat the data differently.

The video logger will continuously take and write entries with OpenCV images to a video file from the time it is created until it is closed. The creation of a new instance of the `VideoLogger` class results in the creation of a new video file. The image logger will create a new image file from each entry given. Both the

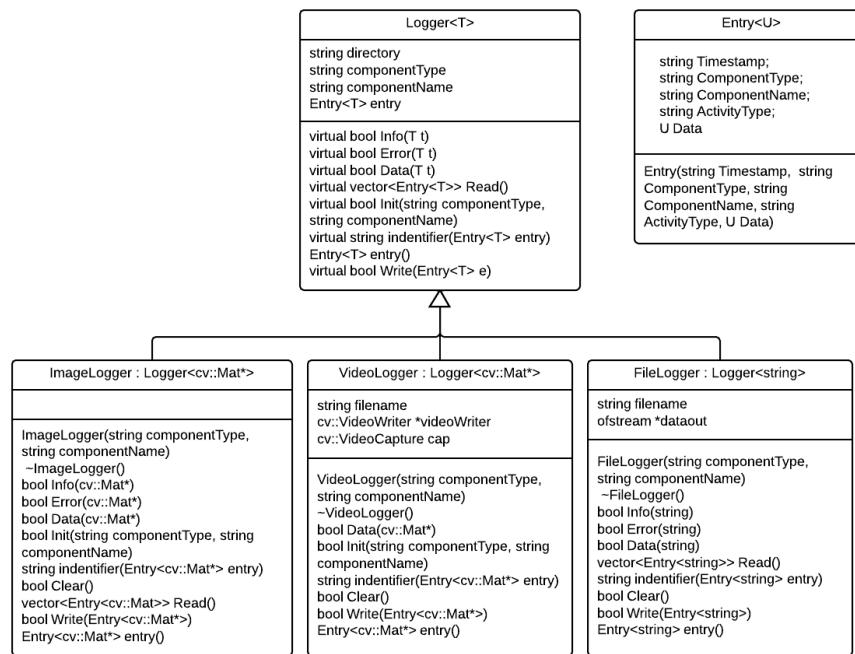


Figure 8.1: Class diagram of the class responsible for **logging** data.
classes logging

image logger and video logger will store component type, component name, and activity type in the file name. The file logger is slightly different as it will use a CSV file for writing every string entry it receives. The CSV file format was chosen because of its ease of implementation and support in programs such as Microsoft Excel (“Microsoft Excel” 2014).

In the current implementation, the logging is not as extensively used as we would like, and improvements can be made to include logging more widely in the program. If the time was available, we would like to ask actual users of the robotic platform for their logging needs, as we, because of our lack of domain knowledge, can only guess about both what data to log and how the data is to be used afterwards.

8.4 Structuring the logged data

An important part of presenting data to the user is how it is stored and structured. It is important to make the data as easy to analysis as possible, so it needs to be structured in a such as way that filtering the data to the users needs are possible. We save four main pieces of meta data useful for filtering:

- A time stamp. Every entry in the CSV file, every video file, and every image starts with a timestamp. This will be useful to distinguish data and in which order things have occurred.
- The component type which allows the user to filter information by a specific component type to get an overview of what the kind of component that have been logged.
- The component name, as this will allow the user to look at data produced by a specific component throughout the experiment.
- An activity type, allowing the user to filter on specific activity types defined by each component. An activity is e.g. ‘error’, ‘data’, and ‘info’.

The data is saved to a single folder in the file system in multiple files. All of the video and image files contains the meta data in the name so they can be filtered accordingly. The entries in the CSV file also contains all of the meta data as well as the data as a string.

punktum

The current model allows for easy access to all the files in a single folder, it could however be argued that the structure becomes a problem if multiple experiments are run without cleaning up the folder. A suggestion for improvements could be to save data for each individual experiment in a new folder, it would then be up to the user to say when an experiment ends and when it starts. We have not looked further into this because of limitations in time.

Vi kunne godt få nogle poi

8.5 Use of limited hard disk space

The Beagle Bone Black (BBB) on which the EvoBot system runs only has 16GB of hard disk space (SD card) available **currently**. This limitation will potentially be an issue over time, when the logging is expanded or when more data heavy components are introduced. While we have not addressed this issue in our current solution we will in this section document our solution ideas with potential for future implementation.

Introducing compression and archiving into our current solution will potentially make the solution more sustainable to future changes. It will not change the fact that at some point the SD will run out of space, but it will potentially accommodate issues with logging more data in an experiment. The solution could also be combined with a way to move the data to a different storage unit to save the time needed for transferring the data.

This leads us to the possible **introduction an other** storage media. This could e.g. be:

- Using an USB storage medium, such as a hard drive. Here there are potential for swapping hard drives when one is filled with data. The biggest downside is the physical maintenance this solution would require. A feature of this solution would be that getting all the data move **elsewhere** would be as easy as unplugging the hard drive.
- Using cloud storage. Here the idea would be that the EvoBot should be connected to the internet via a high speed connection. This would allow the EvoBot to gradually move the data from the BBB to a remote cloud storage. The problem here is the dependency on the speed of the internet connection, the upside is however the mobility and availability of the data world wide.

Vi nævner slet ikke her, hva
Vi har valgt at der kun gemic

A notable thing is that both solutions have an added cost to the entire solution. Picking a solution would be a decision to be made based on further investigation of the needs of the laboratories in question.

8.6 Summary

In order to make it possible for the EvoBot to save data created during an experiment we introduced a logging system. The system had to be capable of saving all the needed data from an experiment in a structured way and storing it in formats that allows the user to analyse it at a later stage.

First we established that there are four categories of data in the EvoBot image/video data, computer vision data, physical behaviour data and software meta data. To support all of the different data we introduced a logger class

structure that include three different kind of loggers for video, image and file logging. Data is then structured using files in the file system and meta data. The meta data includes a time stamp, component type, component name, and an activity type.

Finally we discussed the issues faced with limited hard disk space on EvoBot. We suggested the possibility of compressing the data and then moving it off the board, to either an external hard drive or a remote server.

[Igen, forkert format på summarien](#)

Chapter 9

Human interaction with the robotic platform

Few software projects are in themselves useful without a way for the software to interact with the world. In the case of EvoBot the world is mainly the scientist designated to use the robot. A world consisting of scientists can potentially be a large and diverse one. In the interest of being useful for as large a subset as possible it seems advisable to aim the design at the lowest common denominator and preferably make it easy to extend at a later time. At the same time, end users are likely to have advanced use cases in mind for EvoBot, making it relevant to not limit the possible usage of the platform with the user interface. This chapter section describes our experiences with trying to live up to this.

9.1 Goals

The following is a summary of the thoughts concerning the user interface of EvoBot. It is to be seen as an informal discussion with actual goals marked with **bold**.

The main value that we seeks to provide the users of EvoBot stems from the top level goal that it **must provide the user with the ability to control as well as receive feedback from experiments**. The goal is apparent directly from the expected uses of the robotic platform. The user is expected to program an experiment, which EvoBot can then run. The experiment data must then be available for the user to see, so the experiment data can be used in research. As an extension of this goal, we seek to allow the user to **see experiment data while the experiment is running**. One could imagine e.g. that the user wishes to run an experiment without knowing exactly what the outcome is. By allowing real-time result data monitoring, the experiment could keep

running, until the user decides to terminate it based on the result data. The following goals are concerned with how to achieve this interaction with the robotic platform.

When considering the goals it is certainly advantageous to consider the end-user of the system. The users of the EvoBot will be scientists occupied with advanced chemical and/or biological experiments. Acknowledging the highly specialized expertise these people possess coupled with the ever present fact that humans rarely are experts in many things at once, one important property of the EvoBot is that it **must run without special technical setup**. This means that the barrier to start using EvoBot must be inconspicuously small. This does not mean that there must be no setup process, but the steps, if any, should only have to be performed once and not have an impact on ongoing use. This enables a technician on-site to perform the installation for seamless ongoing usage.

The complement to the previous arguments is that we, the designers of EvoBot, certainly do not possess the expertise to figure out which experiments will actually be beneficial to run on the platform. It is therefore important to not design a user interface that will eventually get in the way of the user. In order to balance between functionality and ease of use, we set the goal for the graphical user interface to **not overshadow functionality that the EvoBot platform can perform**. Specifically the user interface should always provide the user with access to control the lowest level of functionality, e.g. direct access to driving a motor, making sure that any unexpected uses of the robot requiring the doing of so is equally possible as expected uses. At the same time, we strive to make the expected use as efficient as possible.

In order to make sure programming in Rucola, the domain specific language that can be executed on EvoBot, also does not require technical setup, we find it necessary to ensure that the graphical user interface **provides a way of writing and executing Rucola code directly on the robotic platform**.

The final goal for the EvoBot user interface is a non-functional one. Developing the robot is a constant reminder that the platform has certain limitations when it comes to computational power. Therefore it is relevant to aim for a solution that has a low performance overhead and favours the limited resource platform. This is a hard-to-quantify requirement and is therefore only stated informally.

9.2 User interface in Splotbot

Splotbot keeps its heritage as a 3D printer, as outlined by Gutiérrez (2012; Gutiérrez 2012, 43–48). All instructions to Splotbot are turned into G-Code at some point. This can e.g. be achieved through the [Pronterface](#) application used in Gutiérrez (2012). But Gutiérrez (2012) has also constructed a small Python library, increasing the level of abstraction of the interaction Splotbot related tasks such as “moveWater”. It only provides methods for sending instructions to

the Arduino board, whereas image grabbing, computer vision algorithms and so on have to be written by the user in Python. **Det fandt jeg jo så ud af ikke var helt rigtigt**

It can easily be argued that this solution is not very user friendly. It is highly likely that all users of the Splotbot (and EvoBot) will indeed not be proficient in G-Code nor in Python, but this also applies to Rucola (in the case of which it is even more likely). The lack of a simple way of doing the more complex tasks such as droplet detection in the Splotbot library makes it much more difficult to learn. It has the upside of the user being able to write any experiment that can be written in Python, providing every possibility of unexpected uses of the robot. But this is at the cost of ease of use. Furthermore, the user needs to make sure she has a compliant version of Python, the Printcore application used for sending G-Code instructions to the Arduino, and the needed Arduino drivers.

Denne paragraf skal kortes

skal vi skrive det er sovs?;)

9.3 User interface in EvoBot

The goal of not requiring special technical setup has a huge impact on the development of the platform, as it poses the requirement that all generally used platforms must be able to interact with EvoBot effortlessly. Little research is required to realize that this calls for some form of common runtime. In the modern world of heterogeneous platforms this common runtime realistically is a web browser, and preferably a reasonably up-to-date one. It is considered a prerequisite for EvoBot that such a platform is present on the PC of the user, and the graphical user interface can therefore safely be implemented in web technologies. As the power of the BeagleBone Black is limited, it is advantageous to move as much as possible of the computation to the computer of the client, which can also be achieved when running the application in a web browser.

With the above goals in mind, a design consisting of two components is proposed:

- Have EvoBot serve a web client to the web browser of the user
- Have a communications layer run on the BeagleBone Black, interacting with aforementioned graphical user interface and the underlying EvoBot software, mediating messages between the two. A graphical representation of this can be seen in figure 3.1.

9.3.1 Constructing the graphical user interface

The actual elements shown in the graphical user interface (GUI) are determined from the same configuration file used on robot startup as described in section 3. The GUI has the same kind of modularity as the rest of the software running ~~the~~ EvoBot, which means that every element in the configuration file has a standalone graphical component. An example could be a set of x and y axes which can be (1) homed and (2) set to a specific position. As shown in figure

9.1a, this functionality is graphically available for the user to access. This helps achieving two of the goals, namely that the user can control the low level parts of EvoBot (the single components) as well as the user receiving feedback from running experiments, as each control can react on messages sent from the EvoBot, such as status events, or, in the case of a camera where images grabbed are emitted as events, for giving direct visual feedback as seen in figure 9.1b.

We have personally experienced that it serves as very good feedback that when a component is added to the hardware and configuration file, the component is immediately available as a graphical control, providing not only the information that the component was registered correctly, but also allowing the user to control it, testing that everything works as expected. And since this is just one part of the graphical user interface, it puts no restriction on what can be done in the rest of the client, giving every possibility of extending with wanted functionality.



(a) The control of a component

(b) Camera feedback

The graphical controls do, however, not help with solving the goal of programming experiments. This is instead achieved by aiding the user in programming in the domain specific language, Rucola, described in chapter 7. A simple text editor with syntax highlighting is included in the web client with a button to run the code on EvoBot as shown in figure 9.2 (the syntax highlighting is actually C#, as the syntax is similar to Rucola. Providing proper keyword highlighting would be an improvement on this). This makes the flow of programming and running experiments as simple as possible without requiring the user to install anything on her computer.

We have considered one limitation with the text editor and programming language. Without knowing for sure (this ought to be tested on actual users), we expect that the learning of a programming language might be difficult for people who have never worked with programming before, which we believe is true for some of the users. It would be possible to add to the GUI a way of graphically writing

the code, providing drag-and-drop functionality and hopefully higher ease of use. We have also considered the possibilities of allowing a user to record their actions and save them as a program for later use. But the implementation of these is considered out of scope of this project.



Figure 9.2: The graphical user interface provides a text editor with syntax highlighting in which experiments can be programmed and run.

At a final note, the GUI contains a panel showing all the experiment data logged as described section 8, allowing her to see, download, and clear the data. Figure 9.3 shows a screenshot of this panel.

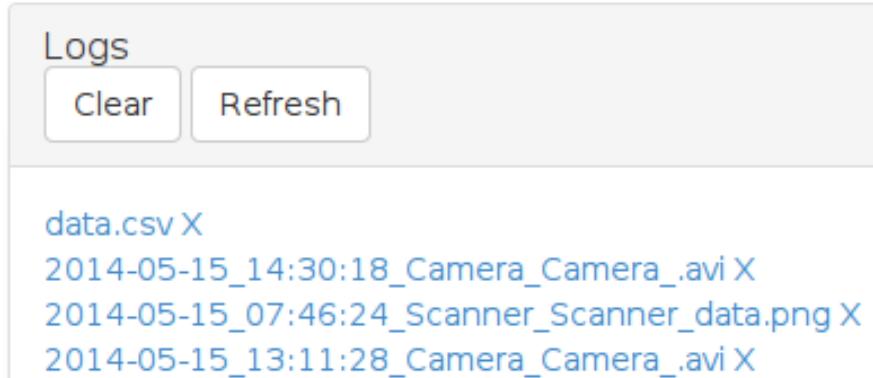


Figure 9.3: The panel where the user can see, download, and clear logged experiment data.

The design outlined above is illustrated in figure 9.4. It shows the calls involved in the process of starting up EvoBot, connecting to it, and interacting with it

through the web client.

9.3.2 The bootstrap process

Using EvoBot needs to be as simple as possible. This includes starting up the robotic platform as well as connecting to it. The first part is easily achieved as we have full control of what is run on the BeagleBone Black. By running our software as a service it starts up when the BeagleBone Black boots, allowing the user to simply power the board to start the program and make EvoBot ready for use. This will require a technician only if something goes wrong in the process.

Connecting to the robot have, however, introduced difficulties, as connecting to EvoBot requires two things:

- The user is on the same network as EvoBot
- The user knows the IP address of EvoBot

Both of these issues were initially solved in software. While developing the platform, we have used UNIX utilities to discover the IP address based on the MAC address of the BeagleBone Black, but the process differs already between Mac and Linux¹. It is unclear if it is even possible on a Windows or mobile platform. Instead a hardware solution was introduced in the form of a simple wireless router. This is connected to the BeagleBone and is configured to always assign the same IP to the machine. This way we know that ~~the~~ EvoBot can be accessed over the wireless network, “EvoBot”, and be found by pointing a web browser to 192.168.1.2:8000. This is intuitive for us, but not necessarily for end users of ~~the~~ EvoBot. To ease the process we have generated a QR code, which will open the correct URL, and is useful if running the client for instance on a tablet, as can be seen in figure 9.5. When using a regular PC, instructions for opening a browser and pointing it to the correct URL will still have to be provided.

9.3.3 Choice of technologies

Narrowing the technologies down to the development of a ‘web client’ is a rather imprecise definition. The world of web is a large one with an abundance of frameworks doing identical or similar things. The EvoBot client does not require very exotic features in this regard, but must at a minimum support calls against a REST and web socket server as well as provide support for constructing a reasonably looking GUI. No member of the team has extensive flair for UI- or aesthetic design, so any help in this regard is a plus.

¹the scripts used on both Mac and Linux is available are our ‘util’ repository (“Utilities Repository” 2014). are in

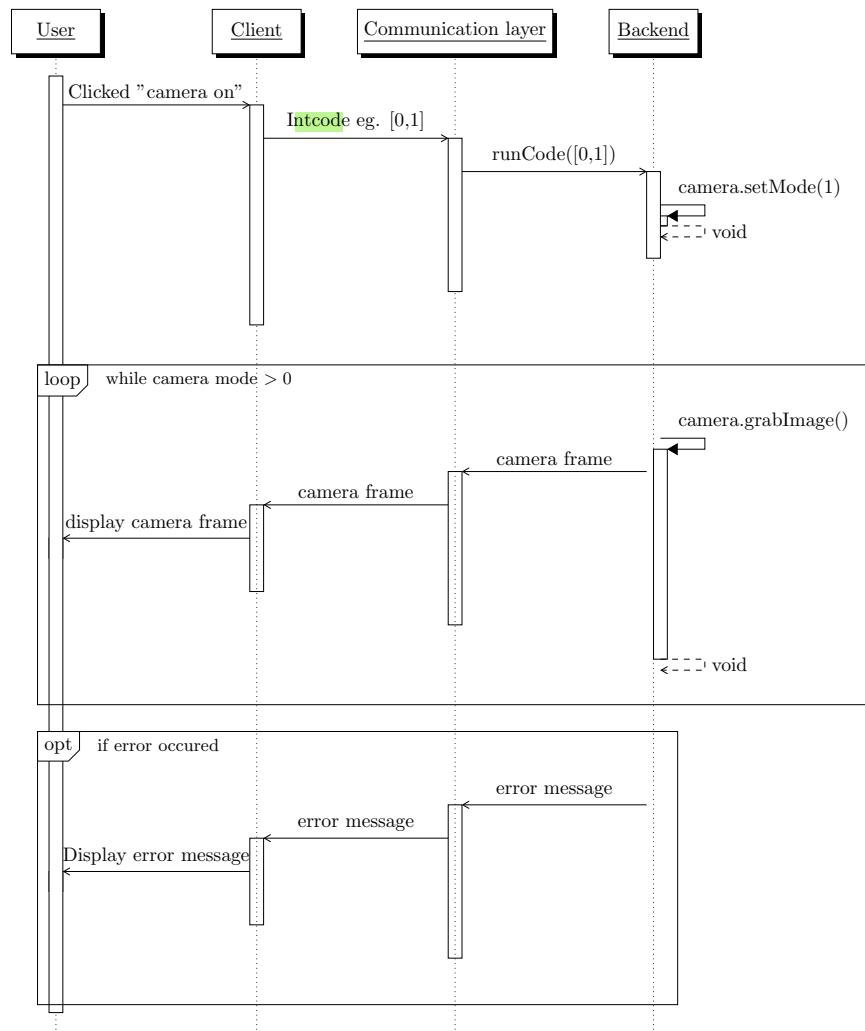


Figure 9.4: A sequence diagram showing the communications between the communication layer and the web client during startup and usage.

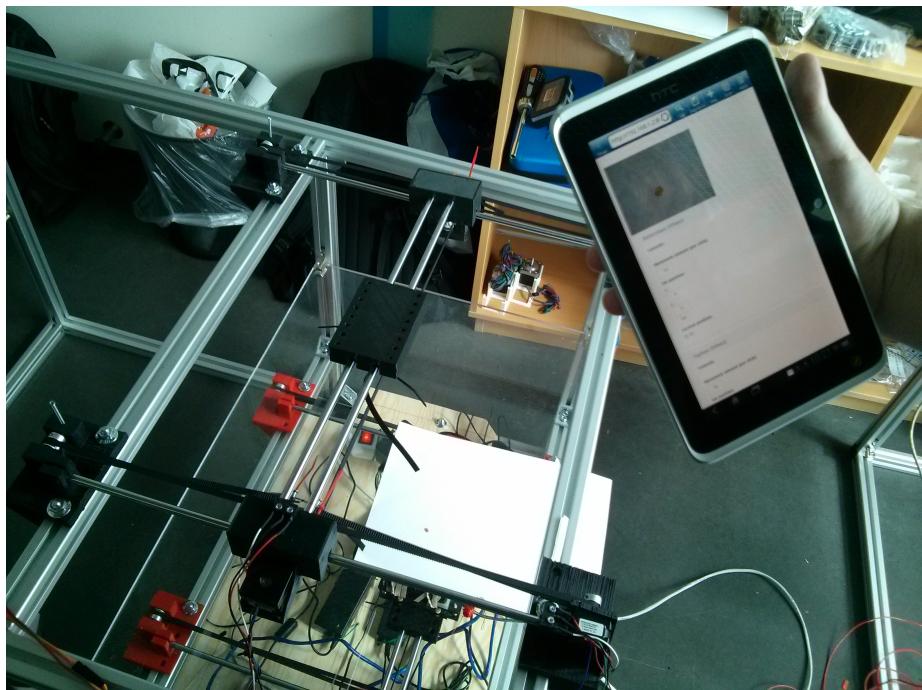


Figure 9.5: EvoBot controlled using a tablet.

If nothing is to be installed on the computer of the user, JavaScript seems the obvious choice, as this runs in every (reasonably) modern browser. Alternatives would be embeddable applications written in languages such as Java or Flash, but using such a technology would pose a further requirement on something to be installed, which can otherwise be avoided.

The framework AngularJS was chosen as a framework for developing the client as it provides all the features for making the development easier. The choice landed on AngularJS because of it being a stable framework made by Google and because we, the developers, have prior experience using it. Alternative solutions could be to either use a similar JavaScript library or a language that compiles to JavaScript with a framework around it. We decided to take the safe route and use technologies we have experience with.

The strengths of a chosen technology often becomes more clear when viewed in the company of alternatives. Our overall choice of technology, web and a REST and web socket server solves our requirements and comes with a few extra bonuses, including but not limited to:

- Keeping the pages static the GUI can in its entirety be moved to the client PC for rendering, sparing precious resources on the BeagleBone Black
- A vast number of libraries for GUI related development exists

We first considered writing the RESTful/web socket server in C++, continuing the existing path of development. C++ does not provide the best support for creating such a web service. So instead we decided to go for newer technologies and went looking for a language with interoperability with C++ and good support for creating a web service.

There is no shortage of languages that provides both support for RESTful interfaces and libraries for this task either. Many languages also has decent support for interfacing with C++, though this is often a task with varying success as the libraries for doing so are both complex and often unmaintained. NodeJS seemed as the stronger candidate because of it already being integrated with the BBB through the BoneScript libraries and because of the support for web sockets through the socket.io library, which makes it possible to send data continuously from the server to the client. NodeJS also has the nice benefit of being relatively lightweight and is built with non-blocking IO in mind, making it unlikely that this layer will become a bottleneck in the project. NodeJS was therefore chosen as the technology used for the communication layer.

The integration between the client layer and the communication layer is very tight in the sense that one will not work without the other. It makes sense therefore to make sure that full interdependence exists between the two layers. This is in practical terms achieved by having the communication layer be responsible for hosting the client layer. This has the added benefit of minimizing the steps needed for starting up EvoBot.

Det lyder som om, at fordi

Is corporate backing

When actually using the technology it was discovered that ease of use was not as high as one could have hoped. The most noteworthy of these issues is the interoperability with C++ as it was one of the major reason for choosing the language. There was both an element of actually not getting things to work, and a feeling that the combination is simply not ideal. ~~The thing that caused the most trouble was using callbacks between the two languages, which resulted in a work around using curl to send events from the C++ core to the NodeJS web service in lieu of getting the advertised functionality to work.~~

Kan det passe, at dette også

The idea that the combination is less than ideal springs from subjective observation where the solution seems ‘clunky’. There is much to be said about a language like JavaScript which lacks conveniences that we have come to love such as types and integers. Even more is to be said about combining it with languages that has a rich type system like C++ or a language like C where integers are quite omnipresent. For now it makes sense to mention how the development process has been firsthand. It is largely dominated by a lot of casting to odd predefined types, that often seems a bit far from the underlying data types in C and C++. This prompts thoughts about whether the languages are too far apart, and if a language closer to C would make the process easier. The second consideration is that it must be rather inefficient to keep crossing such a high barrier, which may be improved with a language with more similar semantics. The second thing that feels ‘clunky’ is the build process itself. Instead of regular Makefiles we are left with a specialized JSON file, which partly consists of text you would place in a Makefile, but now coupled to specific keys in a map structure which is sparsely (if at all) documented. This results in a lot of guessing about things that are usually trivial and much time spent getting it to work as expected. ~~Dette er sidste egentlige indhold~~

[Despite these challenges, against all odds, a few good men managed to](#)

9.4 Summary

EvoBot is not much good if no one can access and control it. With Splotbot this interaction is achieved through either low level G-Code instructions sent to the Arduino that controls it or through a small Python library capable of generating this G-Code. It does however require installation of software on the computer of the user.

For EvoBot we developed a web based user interface consisting of a front end client and a REST and web socket based web service which serves the front end code as well as communicates with the back end. This allows EvoBot to be used from any modern web browser.

The graphical user interface consists of graphical controls constructed from the configuration file also used to initialize the core software at startup. Furthermore it provides a text editor for writing and executing Rucola code as well as access and control over logged data.

The problem of establishing connection to the robot was handled by having a

fixed IP address on which to connect as well as a QR code for access from tablets. All this combined results in the goals of achieving control of the full capabilities of the robotic platform without technical setup are achieved.

Chapter 10

Conclusion

Based on the previous work on the robotic platform Splotbot, we have managed to build a new prototype named EvoBot, which has both overlapping and new functionality, while some functionality of the previous platform has not been implemented. The prototype is used for assessing a number of design decisions, providing guidance in future iterations of building the robotic platform.

All the hardware and software have been built from scratch, continuously focusing on modularity. As a result of this, EvoBot consists of a number of components in both hardware and software. Based on a configuration file, the relationship between these is established at startup.

A major change in the new prototype is that it has a moving camera. By first calibrating the camera, removing radial distortion and estimating the relationship between images and physical movement, we were able to make use of this for both droplet detection and scanning. Droplet detection was based on the corresponding feature in Splotbot, however with some modifications and limitations, and with extra considerations to the application of image filters. Scanning was something entirely new, for which a pipeline was developed where the moving camera is used to grab several images which are then stitched together to form a single larger image covering a large surface area.

In order to allow for programming of experiments on EvoBot, an instruction execution architecture was built. The concept of events was ~~then~~ added, allowing for the robot not only executing instructions but also emitting data. On top of this we implemented a domain specific language named Rucola, supporting integer arithmetics, variables, control flow mechanisms, execution of instructions on components, and binding to events. This meant that complex, autonomous experiments could be programmed and run directly on the robotic platform.

As experiments have the purpose of producing data for later analysis, data logging was added to the prototype. It is capable of logging videos, images, and textual data. The data is stored in a structured format, but we lack input

from the actual users about both what data they need and how it is to be used afterwards.

Finally, we have build a web based graphical user interface for controlling the robot. That it is web based means that using the robot requires no installation of software on the computer of the user. The user only needs an internet browser, as well as the ability to connect to the robotic platform. Connection can be either wired or wireless through a router mounted on it.

graphical

The user interface has separate controls for each of the component as defined in the configuration file read at startup. Furthermore, it provides a simple text editor with syntax highlighting for programming experiments, as well as access to all the logged data, giving the user access to all implemented functionality of EvoBot.

For this project we have build working prototype of a vision based liquid handling robot for automation of ch

e

Chapter 11

References

- “Arduino Mega 2560.” 2014. April. <http://arduino.cc/en/Main/arduinoBoardMega2560>.
- “Beagle Bone Black.” 2014. May. <http://beagleboard.org/Products/BeagleBone+Black>.
- “BeBoPr Cape.” 2014. May. http://elinux.org/Beagleboard:BeBoPr_Cape.
- “Bison - GNU Parser Generator.” 2014. May. <http://www.gnu.org/software/bison/>.
- “Code Repository.” 2014. May. <https://github.com/bachelor-2014/code>.
- “flex: The Fast Lexical Analyzer.” 2014. May. <http://flex.sourceforge.net/>.
- “G-Code.” 2014. April. <http://reprap.org/wiki/G-code>.
- Gutiérrez, Juan Manuel Parrilla. 2012. “Automatic Liquid Handling for Artificial Life Research.” Master thesis. University of Southern Denmark.
- “Microsoft Excel.” 2014. May. <http://office.microsoft.com/en-001/excel/>.
- Nicholson, Arwen. 2013. “Automatic Experiments in Artificial Life Using a Liquid Handling Robot Named Splotbot.” Project. University of Southern Denmark.
- “OpenCV Stitching Pipeline.” 2014. May. <http://docs.opencv.org/modules/stitching/doc/introduction.html>.
- Paulsen, Rasmus R., and Thomas B. Moeslund. 2012. *Introduction to Medical Image Analysis*. 2nd ed. Kgs. Lyngby, Denmark: DTU Informatics - Department of Informatics; Mathematical Modeling.
- “Pololu Maestro Servo Controller - Cross-Platform C.” 2014. May. <http://www.pololu.com/docs/0J40/5.h.1>.
- “Pololu Servo Controller.” 2014. May. <http://www.pololu.com/product/207>.

“Printrun Github Repository.” 2014. April. <https://github.com/kliment/Printrun>.

Solem, Jan Erik. 2012. *Programming Computer Vision with Python*. Pre-production draft.

“Utilities Repository.” 2014. May. <https://github.com/bachelor-2014/util>.

“Wikipedia: Biofilm.” 2014. May. <http://en.wikipedia.org/wiki/Biofilm>.

“Wikipedia: Optical Coherence Tomography.” 2014. May. http://en.wikipedia.org/wiki/Optical_coherence_tomography.