Generelt er der rigtig mange stavefejl, grammatiske fejl, uformelle sammentrækninger og formuleringer, der ikke giver mening.

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 17, 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 proves the feasibility of key aspects such as running the platform completely standalone and scanning and stitching large areas in linear time. This paper should serve as provide key insights and solid recommendations for the next stages of the EVOBLISS project.

development of the robotic platform

Er det ikke kun en lille bid a EVOBLISS vi berører?

Contents

Chapter 1

Introduction

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

Vi har vel ingen intention om at kigge på hele EVOBLISS, der består af mange dele. Vi er kun en bid af EVOBLIS

This project is the continuation of past research, see section 1.3. Even with this amount of past work to build on, the EVOBLISS project in its entirety is all too large for inclusion in a single bachelor thesis. For the remainder of this report we will refer to "the project" as being the work carried out in this bachelor thesis.

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

are

Assessment of 1. The feasibility of our proposed design

Assessment of 2. Feasibility of specific additions to the existing functionality

, which is ...

To assess both of these goals the same method will be used: Implementing the design and assess the success of the implementation. It is worth noting that this approach has the consequence that none of the topics covered are investigated ~~in great detail. Instead, many topics are covered~~ slightly.

.. but not indepth

slightly virker weird

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.

Bør vi nævne vores git repos her?

1.3 Past projects

The work of 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, which was handed in in June 2012 (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, which was handed in in June 2013 (Nicholson 2013).

Relevance?

is

Splotbot was based on a RepRap Prusa Mendel 3D printer, and 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 was reused, but the frame was entirely new. It allowed movement along two axes, X and Y, and six syringes had to be controlled, resulting in the need for twelve RC servo motors, which were added. It was controlled through an Arduino Mega 2560 ("Arduino Mega 2560" 2014) board, which was connected to a personal computer through USB. The user of the robot had to send G-code ("G-Code" 2014) instructions to the robot, which were sent by use of the Printrun ("Printrun Github Repository" 2014) application made to control 3D printers. Only a small subset of the G-code instructions were used (Gutiérrez 2012, 33–48).

The hardware *design* was

Generelt synes jeg det her afsnit skal i nutid.

Faktisk lavede han et python 'library' som generede og sendt så man kunne programm i python.

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

- moving a carriage with 6 syringes along two axes

- controlling each syringe, moving it up and down and pick up and drop 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 (the camera calibration is done manually //TODO make sure this is correct)
- applying image processing techniques to detect droplets (single and multiple) and compute their centers and sizes. All the processing was done on the computer connected to Splotbot
- reacting on the result of the image processing in real time, modifying the experiment accordingly if necessary

Det forstod jeg det s

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

1.4 Contents of the report

Siden der bliver lavet flere EvoBot, bør vi så skriv

The prototype created for 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 the EvoBot as they were on previous prototypes, as focus of the project is quite broad. The EvoBot consist of a new hardware setup and a new rewritten software platform both rebuild from scratch but building upon and to some part extending what was previously made in past projects.

Somewhat eller in some areas

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

- **Chapter 2** explains how the physical robotic platform was constructed
- **Chapter ??** covers the software build 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 focuses on 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 the EvoBot capable of interacting with running experiments without human interaction through giving the user a way to program entire experiments

investigates

by

- **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 the EvoBot
- **Chapter 10** concludes on the results of the project

1.5 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, as well as allowing us to use workshop of the IxD Lab at the ITU 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

Chapter 2

Constructing the physical robot

This chapter focuses on building the physical part of the 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 ??cm metal frame. It is ??cm tall, ??cm long, and ??cm wide. In the center of the frame is a ??glass?? plate attached, on which experiments are run. An overview of the robot is given in figure 2.1 (a).

'the next part', hvor er den forrige og næste? det virker som en weird måde at skrive her

The next 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. ~~Furthermore,~~ 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. This setup is shown in figure 2.1 (b).

In order to move along the y axis, a stepper motor is attached to one of these pieces with the previously mentioned belted attached. A similar constructing allows for driving a belt between 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

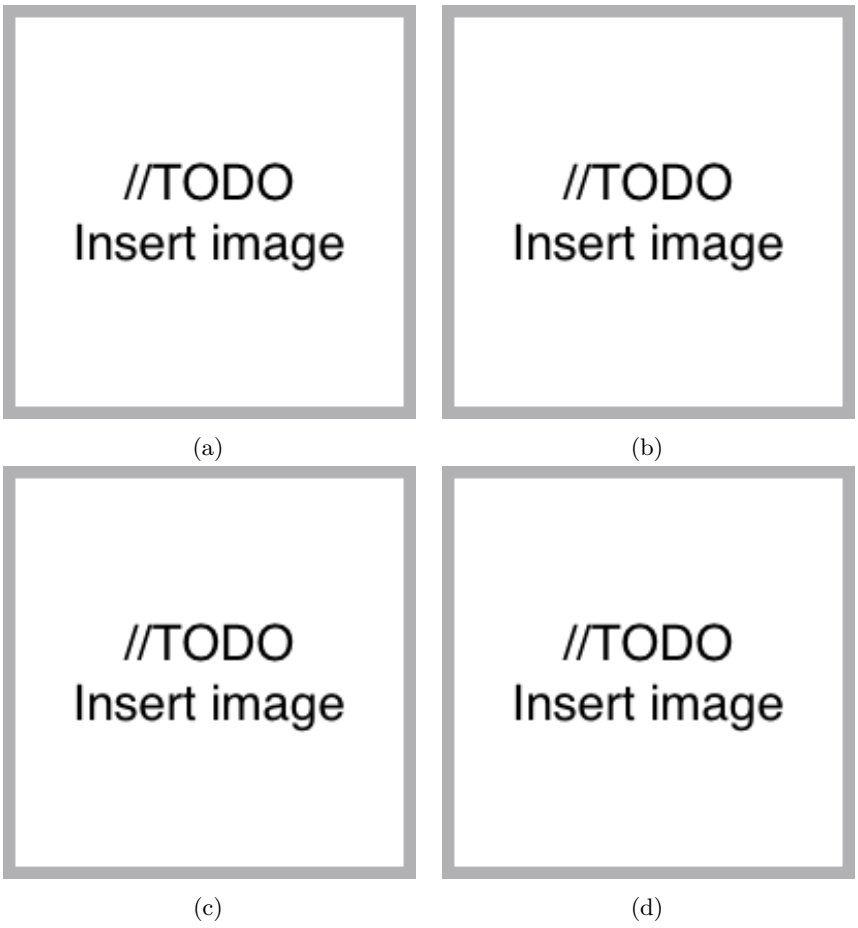


Figure 2.1: Images of Splotbot.

controlled by two RC servo motors. The y axis and carriage are shown in figure 2.1 (c).

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

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 Splotbot:

Er også et erklæret mål på EVOBLIS

- 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 access 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 ~~of~~ 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 stand alone unit, we wanted to exchange the Arduino board with a more general purpose computer

are

Omskriv. Er lang og kluntet

The carriage running on linear rails driven by belts and stepper motors seemed to work well enough on Splotbot for us to not want to spend time trying to improve them due to the limited scope of the project and due to us having hardly any knowledge about hardware.

I stedet for 'nice to have' sætningen her, burde vi så ikke sige at de skal laves i en senere iteration?

Jo

Finally, due to the limitations in available time we decided not to spend time on constructing syringe parts, though they be nice to have on the EvoBot as well. We do, however, include the control of RC servo motors in the scope of the project, so adding syringes similar to those on Splotbot should be possible at a later point in time.

Crucial features such as syringes are omitted in this project, and sho RC Servo Motors are in place on d

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

2.3 Making the frame

The 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 it being ??66cm?? tall, ??50cm?? long, and ??76cm?? wide.

Though this is true, it is also highly incriminating. I think it can be rephrased. There are advantages to using the same parts across the projects

Initially, the 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 as finding out what hardware he needed to build it. As we at that point had very little idea of what we needed, we agreed that it would be a good idea to follow his lead and buy the materials needed for building his take on the robot. However, as the work moved along, we found that the dimensions we 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 the EvoBot in

Figure 2.2: The EvoBot.

figure 2.2. We suggest that the frame is cut in half, removing the unused part ~~of the frame as it is only in the way~~, but we did not get to doing this. This was omitted due to time constraints

As on Splotbot, a transparent plate is attached to the frame on which the experiments are run. On the EvoBot, this is a ??mm Plexiglass plate. 'Plexiglas' er vidst et mæ

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

2.4 Designing and 3D printing the parts

Udover vores components. Skal det nævnes her?

The 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.)
- 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.)

neither of the group/project members...

Each of the parts were designed and printed by us. As a disclaimer, neither of us ~~as~~ 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

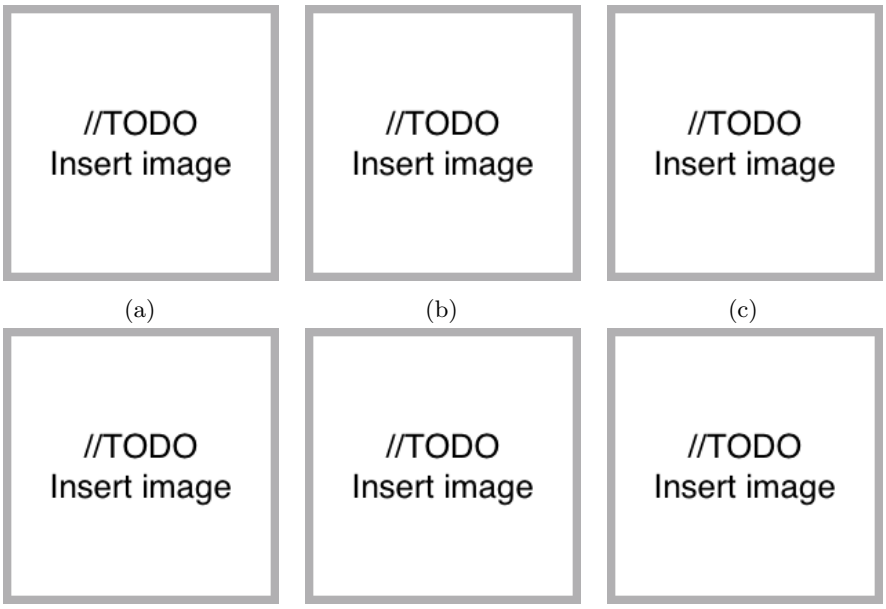


Figure 2.3: The 3D printed parts of the EvoBot.

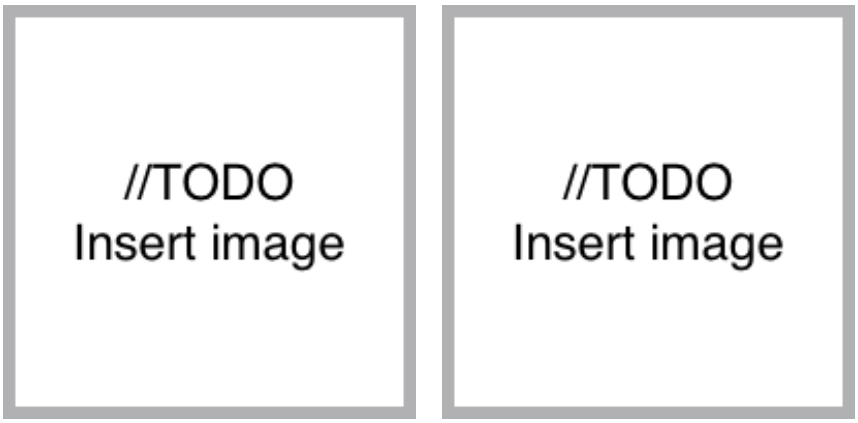
both imprecision in the print and due to the designs being less than perfect. As the printing of a part takes a long time (usually several hours), having many iterations of designing and printing the parts seemed quite time consuming. We therefore often deemed a print as being 'close enough' and then used tools to alter it slightly.

Lad os nævne at flere sideløbend

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.

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.



(a) (b)

Figure 2.4: The issues with the design of the y axis holders.

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/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.

x and y

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 the 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 by 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 Plexiglass plate. On the picture, the component is not fully attached to the carriage as the holes of the component are a bit too small. But it is still very firmly attached.

Figure 2.5: The electronics of the EvoBot.

~~The final part of the hardware are the electronics, which are described in the next section.~~

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 stand alone. 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. At the same time we need to keep the price of the robot low. 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.

I feel that we haven't sufficiently s

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 ~~being a~~ Linux ~~computer~~, 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 three 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 Polulo Servo Controller board, an USB device for controlling servo motors ("Polulo 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 the EvoBot ~~as~~ 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 attached to the EvoBot, we have attached a wooden plate to the bottom of the frame on which the electronics and power supplies are fastened.

Var det i det her afsnit vi gerne ville have motor experimentet? og hvor ville vi har camera update speed

The BeagleBone Black is very limited in computational power. The suitability of the board will be discussed in relation to the topics covered in the remaining chapters. Lidt ud af ingenting at vi sviner den stakkels beagle?

2.6 Summary

We first described the hardware of Splotbot, the robot on which the EvoBot is based. We then gave an overview of the differences in hardware needs on the two robots respectively, such as the EvoBot needing a bottom carriage for the camera whereas the camera is fixed on Splotbot. Based on this we introduced our design of the parts of the EvoBot which we have 3D printed, along with a discussion of the parts of the design that did not work as well as hoped. Finally, we explained the electronics used in EvoBot, being controlled by a BeagleBone Black microcomputer with a BeBoPr++ cape extension, a separate servo controller connected through USB, and an USB hub for attached several USB devices.

Jeg havde en idé om at summary

Bør vi nævne den trådløse router igen?

Chapter 3

Building the software to control the EvoBot

Vi skal blive enige om, hvordan vi bruger robotte

In this chapter we will first describe how the software in the Splotbot is implemented. We will then compare the Splotbot's software with the EvoBot's software. Then we will look at how the EvoBot's core software application is constructed ~~and~~ finally we will look at how the core software accesses the EvoBot~~'s~~ hardware

Jeg ser det ikke som robott

3.1 Description of the software running Splotbot

the Python based software Printrun

The core of the software that is used to run Splotbot is the Printrun python based software. Printrun is made for RepRap 3D printers. It takes gcode instructions and translates them to motor movements etc~~. and is built for 3D printing items by controlling the movement of the hardware~~. The Splotbot can be controlled by writing a set of gcode instructions to move it's hardware because most of the common functionality is available in gcode like moving axis.

jeg er ikke vild med ejefald i

burde nok lave refs til Splotbot raport

Written on top of the Printrun code is Splotbots custom python software. This software generates gcode that is executed by Printrun to get Splotbot to perform actions. 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.

Jeg er ikke vild med ejefald i

Well, I hvert fald ikke formen. Jeg s

Experiments for Splotbot are design 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 inject them into a

Them liquids man

petri dish, tracking a droplet and reacting to the droplet speed injecting more liquid when the droplet movement have halted.

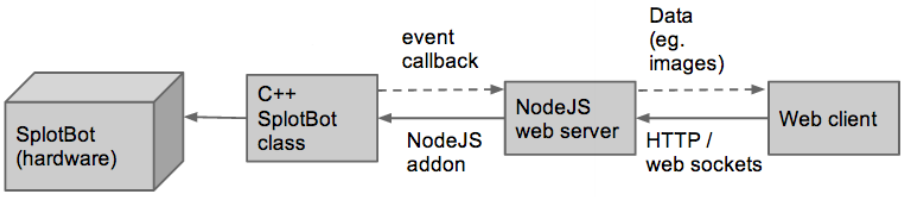
3.2 From Splotbot to EvoBot

The EvoBot software differers from the Splotbot software in many ways and is a complete rewrite of most of the functionality available in the Splotbot but also including new features. The software is no longer Python based and 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 differers 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 therefor emphasis 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 ~~has~~ extended with features such as scanning, a web interface and a programming language. Focus has been put on making a standalone robot with ~~a~~ software that can be connected to and used without the need of writing or changing the C++ code.

Experiments are now run by using the EvoBot software rather than extending it with a new python script. Experiment can either be defined as low level instructions similar to Gcode or via a domain specific programming language (DSL) made for the EvoBot ~~and run on the EvoBot's software.~~ 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 ~~the droplet's changes in speed.~~

3.3 Description of the software running the EvoBot



The software written for our prototype is structured in three main components, the core, the NodeJS server code and the client. The core handles the robot features, the NodeJS acts as a bridge between the core and the client. The client allows the user to manipulate the robot real-time and also allows the user to send

be executed on

experiment code to the EvoBot. Below is a breif description of the software in its entirety. The source code ~~code for our implementation~~ 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 can be loaded and runtime 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.
- 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 NodeJS part of the application wraps the core application providing access to it through a web socket and REST http interface. This makes EvoBot into a web service that can be access through any application language that supports either web sockets or http.
- The client consist of a JavaScript based web client, that communicates with the server through web sockets and http requests. The client similar to the server loads the configuration file and constructs the GUI based on the available components.

are loaded at startup

This ...

Teknologier ud
også fra diagrammet

The web server, written in
NodeJS, wraps ...

This means that the EvoBot

Both, not either, rig

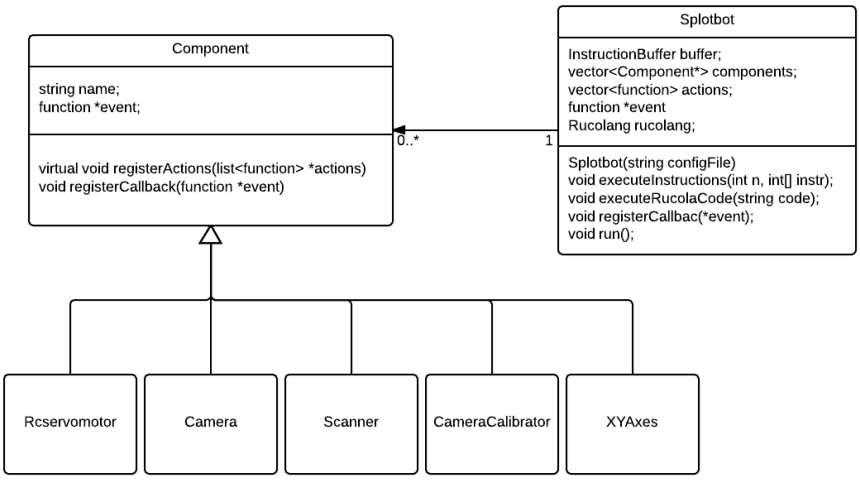
Like the core application, the client loads the system configuration

and constructs the GUI based on this.

New section about the communication between the software layers

Sorry at jeg går meta og retter en masse comments. Det skal forstås som at jeg er enig i at kommentarteksen bør gøres gældende (med mine rettelse

3.4 Constructing the software core



Alle figurer skal have et nummer og en caption

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 5 available components each with different functionality and hardware requirements:

five

video recording, image grabbing, and droplet tracking.

- **Camera** handles the general video recording of the experiment. But it also handles the droplet tracking when it has been instructed to do so.
- **XYAxes** are used to control a set of two axis on the robot. The XYAxes has functionally such as move to a specific position and homing it.

for homing, resetting the posit

Måske forklare homing i e

RCServoMotor is

- **Rcservomotor** are 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".
- **CameraCalibrator** are used for calibrating the camera. This component have functionality to perform a camera calibration, but also to just load the most recent from a file.

and for saving and loading su

- **Scanner** are used to scan large surface areas using image stitching.

techniques

An important part of the design of the software core have 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 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.

The configuration file is written in JSON and an example component can be found below. 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.

```
{
    "type": "XYAxes",
    "name": "BottomAxes",
    "parameters": {
        "x_port": "X",
        "y_port": "Y",
        "x_limit_switch_port": "J9",
        "y_limit_switch_port": "J11",
        "x_step_limit": 79,
        "y_step_limit": 58
    }
}
```

3.5 Controlling the hardware from the software

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 it is done.

- 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 a software that can use gcode to access printer

hardware including stepper motors. We have however made modifications to the software to allow for multiple axis. 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 make our own homing functionality on each axis and that each axis behave similarly. To use the 3D printer application we start it and make it read from a file using the Unix tail on a file and we then write to this file to transmit gcode to the application.

- 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)

We have modified the softw

punktum

Problems

3.6 Summary

In this chapter we first looked at the Splotbot's software implementation, we then compared this implementation to the EvoBot's software implementation. We then described the complete software that is running the EvoBot. Then we looked at how the software core is constructed and finally we looked at how the software core controls the different kind of hardware available in the EvoBot.

Flere punktummer, og lad o

Chapter 4

Robot-camera calibration

Jeg tror vi mangler en sætning der lige tager læseren i hånden

also the case

The camera is a central part of the current EvoBot setup. This was the same 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

Vi skal lige huske at sikre os at alle goals afsnittende kører samme format

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 as 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 see the two different calibrations, for radial distortion and for the relationship with the physical movement, as conceptually being a single calibration from the point of view of the user, and we therefore want that **both calibrations are a single calibration from the point of view of the user**. Hopefully, this limits the number of required input from the user, before the robotic platform can be put to use.

Redundancies

I don't do hopes. I do aims though

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 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 then 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:

Long

1. Grab an image
2. Move a single step

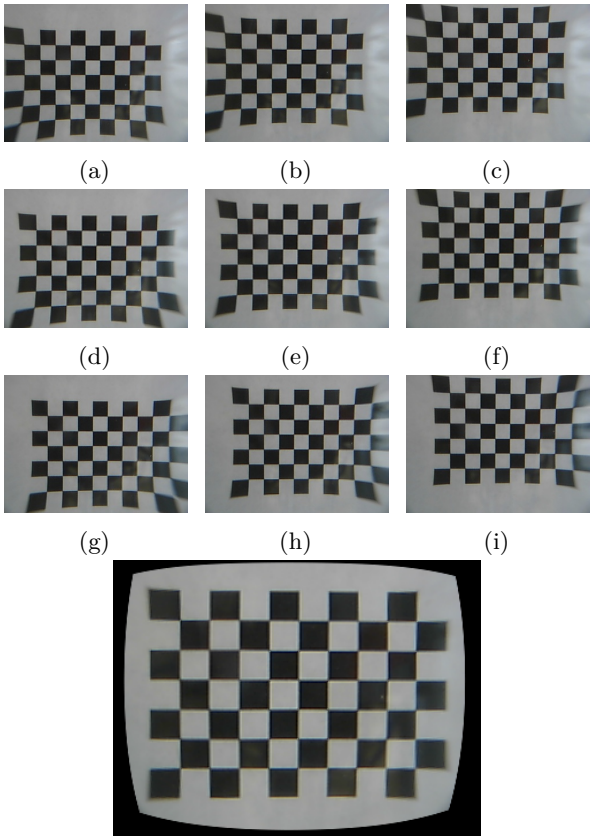
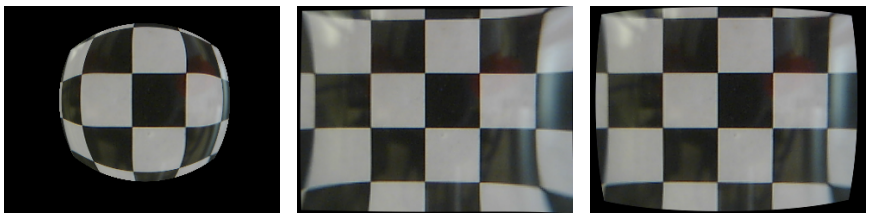


Figure 4.1: Calibration of the intrinsic camera parameters and undistortion of images. Images (a) - (i) show the images used for the calibration. Image (j) shows an undistorted image.



(a) Small pattern (1.9x1.3cm) (b) Medium sized pattern (3.1x2.2cm) (c) Large pattern (5.6x3.9cm)

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

Nitpicking perhaps, men det vir

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 ~~conceptually~~.

I don't think the user sees concepts

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):

Jeg har savnet Paulsen og Moesl

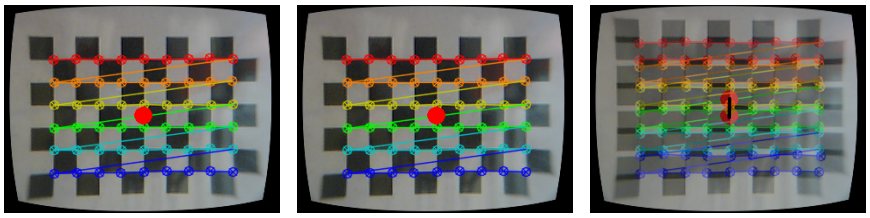
$$\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_2 \\ 0 & 0 & 1 \end{bmatrix}$$

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:

$$\Delta x = x_2 - x_1$$
$$\Delta y = y_2 - y_1$$

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



(a) (b) (c)

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 the 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 designer 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 platform respectively. Finally a series of experiments are made to illustrate the success of this implementation along with a discussion of these 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 considerations aside from the features listed, is that performance can be impacted by such a feature. Droplet tracking will operate on a live video feed, 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.

5.2 Droplet detection in Splotbot

The droplet detection and tracking done in the Splotbot project is thoroughly covered in the thesis (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 Organized 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 coming section.

'Self Organized' eller 'Self O

'following' eller 'next'. Det andet lyder som om den ikke er færdig (i hvert fald for mig)

5.3 Finding the droplet

Actually finding the droplet is based on user input for starting the experiment (first frame), that is user clicks objects of interest. This was shown to be more robust than automatically doing it infer which frame to start tracking.

Finding the droplet is based o

virker clunky med den sætning

Like EvoBot, Splotbot uses OpenCV for most if not all vision related functionality. This functionality specifically is segmentation. Three different forms of segmentation is considered in the report

are

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

subtracting

seing

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.

considering

considering

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.

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.

5.4 Droplet detection in EvoBot

lidt clunky med 'and we have'

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 at much as possible of the knowledge gained in that project, and we have. 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. Color of selected pixel determines blob of interest.
3. To better extract the blob of interest, noise is removed using a filter.
4. Binary image is extracted based on color segmentation, not motion nor shape.
5. Morphology is used on the binary image, to account for "holes" in the blob.
6. Droplets matching the size and color of the droplet ~~is~~ are selected.
7. Of these the largest droplet is chosen as the one to be tracked.

The

A

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

cpp file?

Architectually, most of this is very specific to computer vision, and resides in a separate class `computer_vision/dropletdetector.cpp`. The User interacting part takes place in the camera component, which has a specific action for extracting the relevant information.

5.4.1 Considerations on the choice of filter

Punktum

The first is ...

The second is ...

As mentioned above, we filter the original image, this is done for two reasons, one is to increase the likelihood of the algorithm detecting the entire droplet. The other 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 a correct blurring mechanism.

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

konsistens

- **Gaussian blur**: Fast, but not edge preserving, degrading the results
- **Bilateral filtering**: Very good results, very preserving, but too slow.
- **Median filter**: A compromise between the above two. Relatively fast and edge preserving.

Dette er ikke et fuldent bille

. A ...

Median filter was chosen based on the above criteria, a comparison showcasing the differences is done in 5.5.3.

section

5.4.2 Considerations on the final choice of droplet

Our implementation

whereas

The EvoBot solution can only track one droplet at a time, Gutiérrez (2012) demonstrates that the splotbot has the ability to track several. It can easily be argued that ~~it~~ the 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 though into the user interface design to make this clear to the user. It is considered out of scope for ~~the EvoBot~~ project to look further into this.

this

5.5 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 two criteria **degradation of performance** and **quality of results**. Both of these ~~will~~ are judged on subjective measurements, but will be properly documented based on data extracted from the EvoBot. Lastly, an experiment is done to determine the performance of one specific part of the implementation, namely the **choice of morphology**

the

mener vi virkelig performanc

punktum

5.5.1 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

Jeg skal lige have en forklaring af de 30

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 source of error. The delay of the camera **??**TODO is also a factor of uncertainty. We do, however, feel that this experiment despite the uncertainty shows to a useful extent the approximate performance degradation.

Hvar har kamera delay med

The experiment is run on the BeagleBone black itself for the most correct results. An image of the setup can be seen in 5.1.

Figure 5.1: Experiment setup measuring droplet performance

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 **0.56 fewer frames**. Given that the nature of the experiments

Det er ikke 56% færre ...

to be run on the EvoBot are usually slow moving, we find these results to be acceptable.

5.5.2 Quality of results

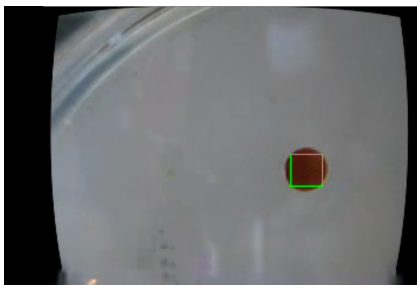
One of the important aspects of the droplet detection feature is of course the actual effects it can produce. As mentioned earlier, this will be done using subjective judgement. To best showcase the results we tried to mimic a real biological experiment to the best of our ability. This resulted in the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

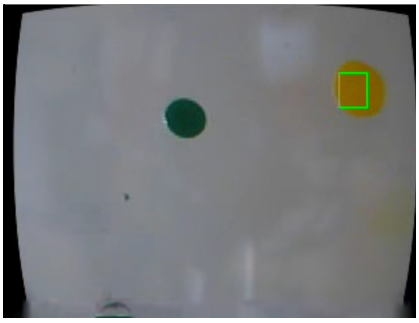
1. A petri dish is filled with transparent cooking oil
2. In the cooking oil, 4 different colours of liquid is placed
3. The petri dish is placed on the camera
4. The petri dish is slowly turned around its own center, moving droplets from the cameras point of view.
5. While the droplets move, two of 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. Operating from a good news first principle, the first thing to mentions is that the tracking works, as two colors can be tracked individually and as they move. This is somewhat poorly illustrated with images, but **??** gives an idea about the results.

billeder i forskellig højde, oh min OCD

Jeg vil foreslå, at vi samler alle billederne i én figur. Vi bruger meget plads på meget lidt





Alle billeder skal have et num

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

Moving from good to neutral of news, 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 5.3. 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).

Måske burde det nævnes, at man kunne tænke mere over dette

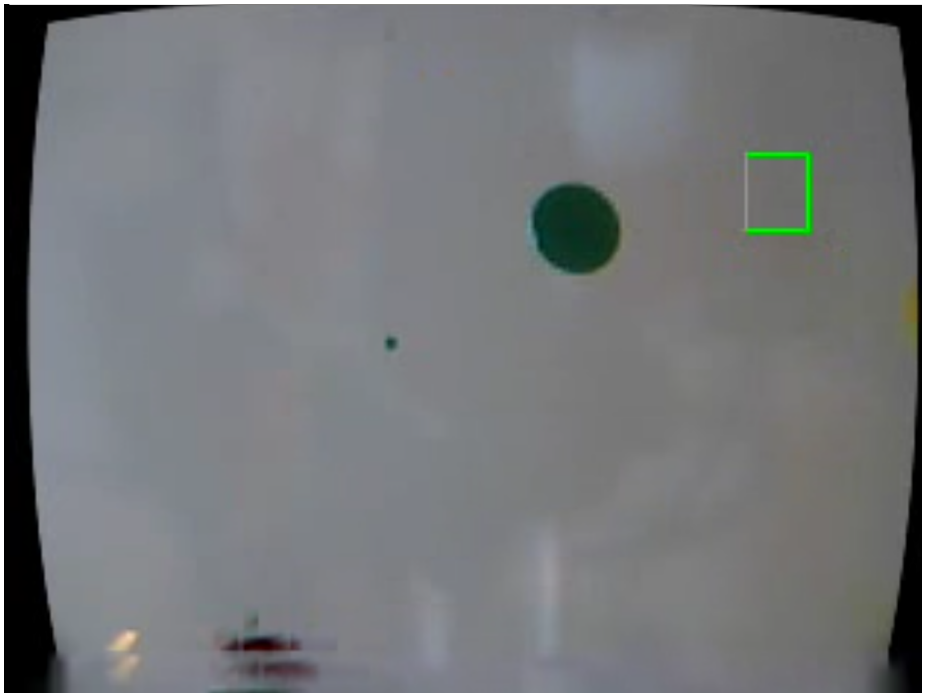


Figure 5.3: Droplet gone out of bounds

Lastly we have to mention the bad news. The implemented tracking functionality is not robust in all cases, and this 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 5.4 and 5.5 respectively.

Figure 5.4: Problem one, untracked droplet

single quotes

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 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 5.5.3 we use blurring for this purpose, whereas the approach taken in Gutiérrez (2012) is to actually fill the droplet with the same colour. Given that the fill method exists for this purpose whereas the blurring has this as more of a side effect, the result is almost certainly better with fill. There is again ~~also~~ the solution of a

Huh?

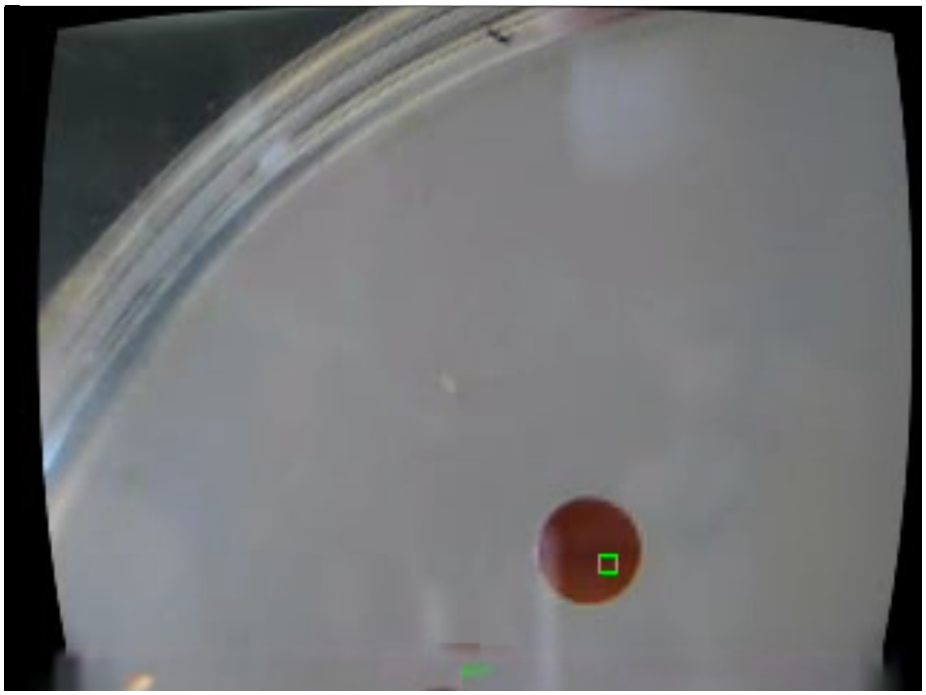


Figure 5.5: Problem two, tracks only part of droplet

better test setup, as the culprit is light differences.

5.5.3 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

The benefits are further outlined in ???

The experiment carried out is as follows:

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 algorithms to the image, with a timestamp before and after.

Har vi ikke kørt dem på BBB?

These experiments were run on a reference computer with 16GB of ram and a dual core Intel i7-3520M CPU @ 2.90Ghz, mangler der ikke noget?

The image used for the experiments can be seen in figure 5.6. It 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 results are illustrated in the following:

Gaussian proved, as suspected, to be very fast at **20ms**. As can be seen however,

Igen, lad os vise de tre billeder ved siden af hinanden

Figure 5.6: Original image

it also gives rather poor results in terms of smoothing out the droplet colouring. It becomes very "grainy" and not very even. It also doesn't remove the noise, but instead leaves a lot of spots in the droplet.

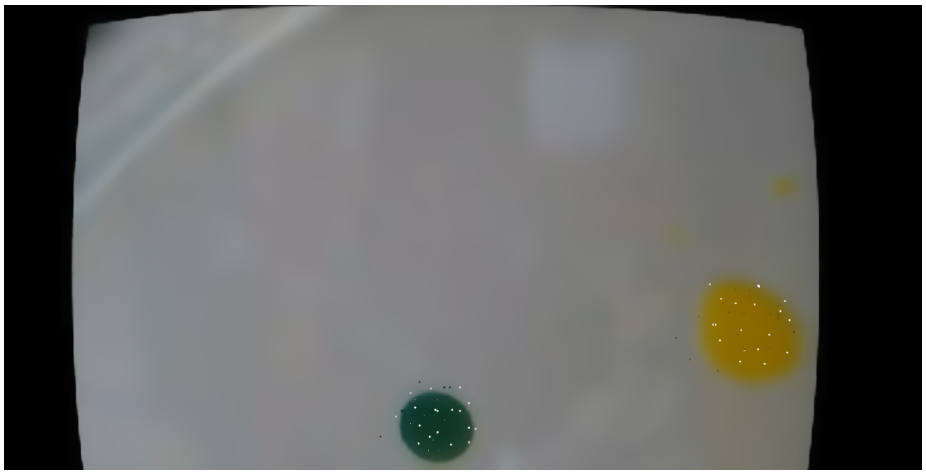


Figure 5.7: Result of bilateral filter

Bilateral filter was by far the slowest at **2632ms**. It does have a nice preserving results in terms of the actual look of the droplet. This image shows it having removed all the black noise from the image, but the white noise can easily be inverted resulting in a noise-free image. The performance of the filter is sadly unacceptable, and its preserving features are perhaps quite a bit better than what is needed.

Mangler caption

Lastly is the median filter. Clocking in at **65ms** 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 colour.

5.6 Summary

Jeg er ikke vild med formated

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.

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 lense ~~as explained in chapter ??~~, this introduces difficulties to overcome, but also leaving room for them to be solved. The single camera is movable, so it is possible to cover a larger surface area than can be captured in a single image by moving the camera to different position, grabbing images, and combining the images to a single image. ~~And~~ this is made possible due to the experiments being slow moving, as this leaves

Bør være to sætninger

time for the camera to be moved and for grabbing the images, without the experiments changing much in the meantime, which would cause the images grabbed being inconsistent.
to be
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 of ~~the robotic platform~~ 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 1 cm^3 ("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 **??**.

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/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.
x and y

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 **??**. Kan det passe, et denne diskussion, der før lå i kapitl

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. Vi kan bruge set notation he
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 an 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. Kan vi smide en køretid på her?

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:

$$minX = 10$$

$$minY = 10$$

$$maxX = 15$$

$$maxY = 15$$

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:

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

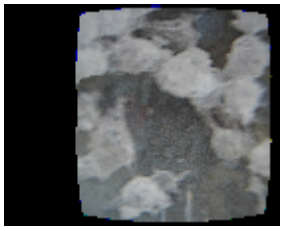
6. These transformation matrices are used to warp each image onto the resulting image.

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

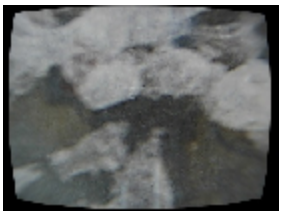
1. We warp the image onto a temporary, black image of the same size as the resulting image.
2. We then threshold the image with a threshold of 1. A thresholding results in a binary image with all values below the given threshold being black and the remaining pixels being white (Paulsen and Moeslund 2012, 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.

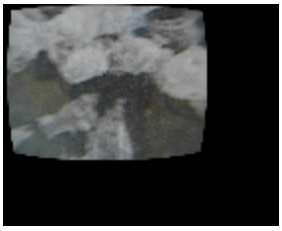
The advantage of this stitching algorithm is that adding an image extra to stitch results in a constant number of added operations, making the runtime linear. The major disadvantage is the dependency upon the correctness of the assumptions. If the camera calibration is not precise, the stitching is equally 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



(a) The resulting image with two images already warped to it

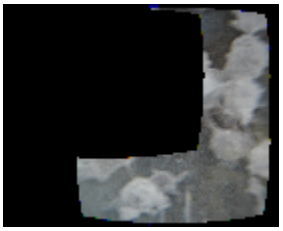


(b) The image to warp

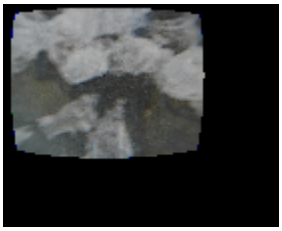


(c) The image is warped onto a temporary black image

(d) A mask for combining the images is computed

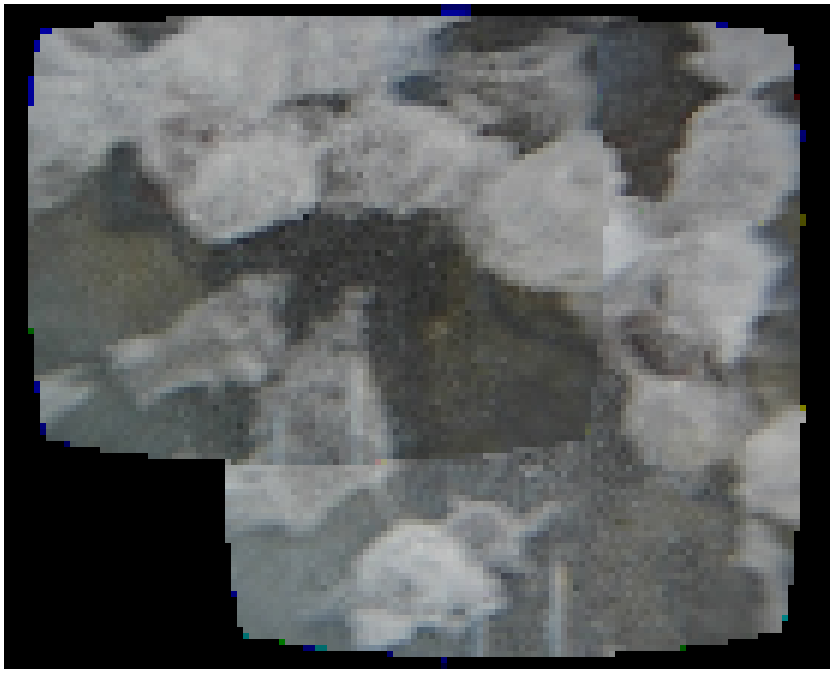


(e) The mask is used to blacken a part of the resulting image



(f) The mask is applied to the temporary image

Nævn de sporadiske farver



(g) The resulting and temporary images are added

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

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.

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.

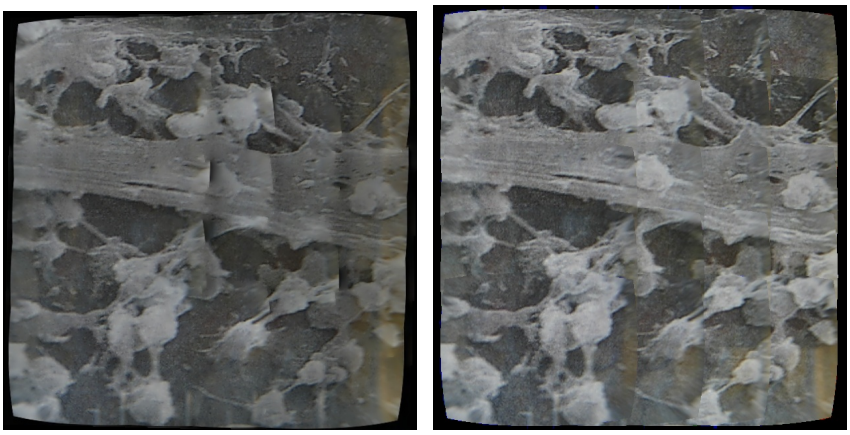
Our idea was to to compute these regions of interest based on the same warping calculations done in the position based image stitching algorithm. These could to the accuracy of the position based stitching algorithm provide the areas in which the images overlap. But we found that this approach had the 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:



(a) (b)

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.

- 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

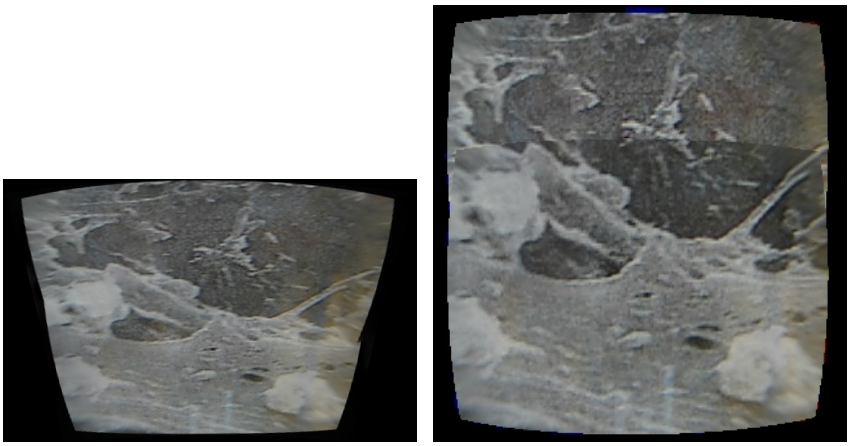
Each image is run both on the BeagleBone Black on EvoBot and on a reference laptop computer, allowing us to assess the feasibility of the algorithm being used both in the current setup and in a setup with a more powerful computer. The reference computer has 16GB of ram and a dual core Intel i5-3320M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



(a) (b)

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.

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 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 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.

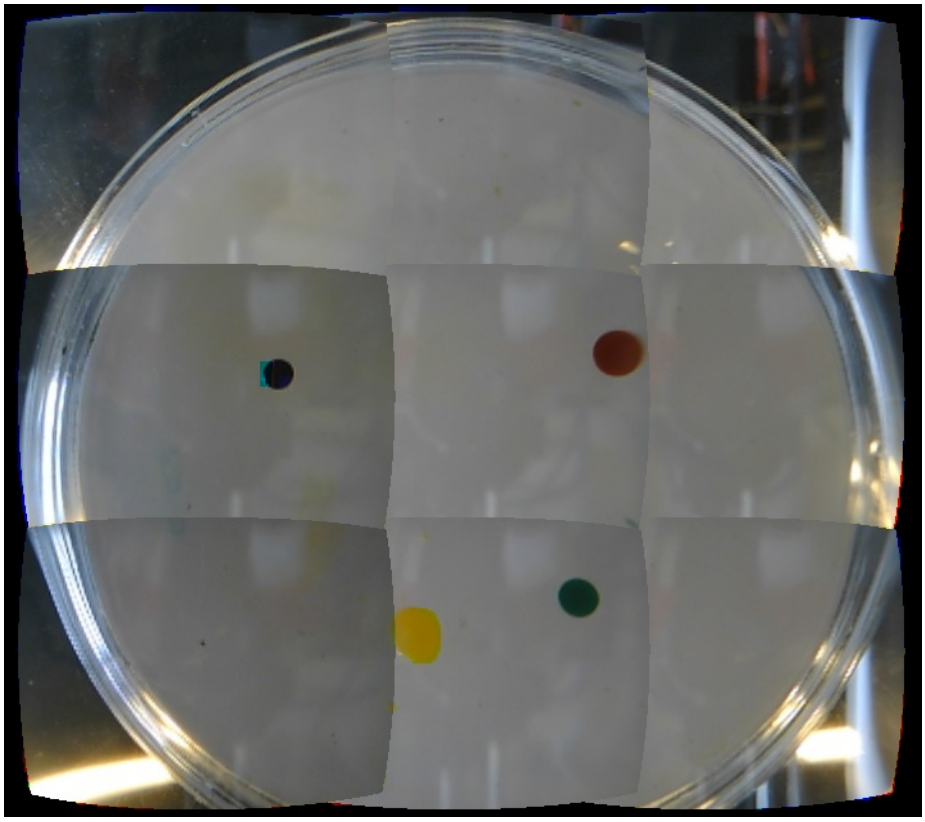


Figure 6.4: The resulting images of stitching together nine images of a petri dish with colored droplets using the position based algorithm.

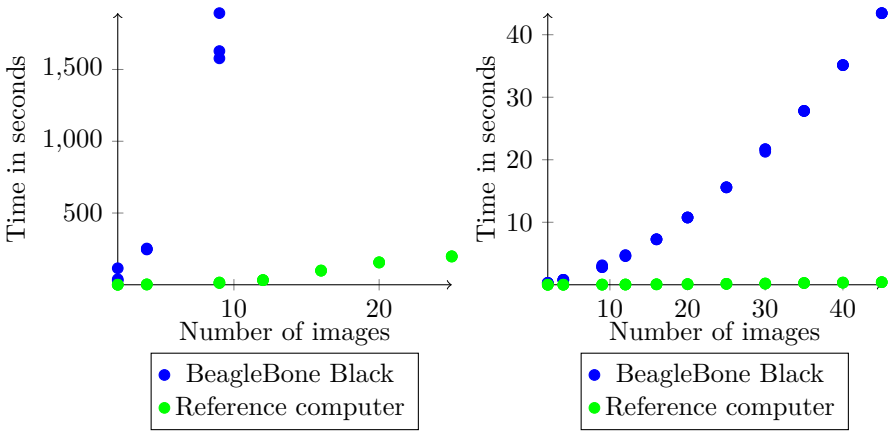


Figure 6.5: The run times of the stitching algortihms: (a) features based (b) position based

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. 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, is 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 to 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 Beagle-Bone 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 prove 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.

Jeg er ikke sikker på om de

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 does 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

The EvoBot should not only be able to run experiments, it should also be capable of running autonomous experiments. Autonomous in the sens 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 system. This chapter looks at the implementation of the system that allows a user to define an experiment and how this system allows for having autonomous experiments.

We first introduce the goals ~~of the experiment running system~~. We then look at each part of the system discussing how it has tried to achieve the goals and 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 a experiment execution that loops and continuously adjusts overtime. This is illustrated in figure 7.1. The loop begins with the initial actions that will be executed setting up the experiment for example picking up liquid and putting in into a petri dish. After the initial actions the experiment will be running, events with data such as droplet movement will be detected by components. Based on events in the system the robot will then react on the data possibly doing new actions which will change the experiment. This loop will create a~~n autonomous~~ reactive feedback process that will do actions to change the outcome of the experiment based on data gatherer throughout the experiment.

Making a user capable of defining ~~autonomous~~ experiments in this system requires

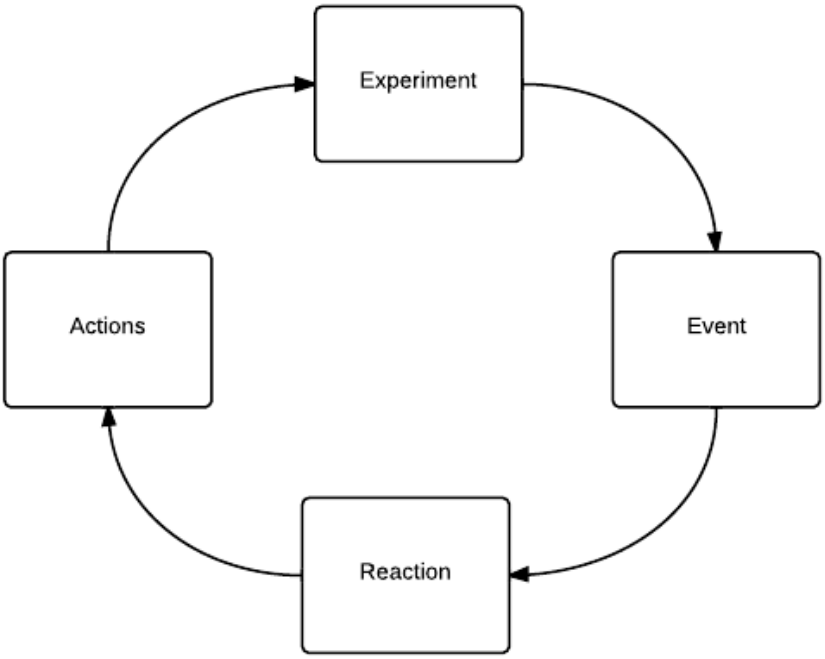


Figure 7.1: Reactive experiment loop.

a few capabilities added to the robot~~, making these capabilities available will be the goals of this chapter~~

1. Being able to send instructions to components to perform actions
2. Components should be able to communicate events with data in their execution and in the observed experiment.
3. Being able to program the experiment and get it executed on EvoBot. In which it should be possible to send instructions to components and react to data communicated by the components.

Disse punkter er en smule

In ~~this chapter we will in~~ section 7.2 look at how instructions are executed on EvoBot, we ~~will~~ then in section 7.3 look at the event system in EvoBot and in section 7.4 we look at exactly how the user is to program the experiments. In section 7.5 we run experiments on the system and finally in section 7.6 we summarise the chapter.

Erstat kommaer med punktummer

Denne blok skal nok fjernes for at holde formattet ens i rapporten

7.2 Executing instructions on the EvoBot

When running experiments on EvoBot a core features 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 based on some parameters perform a certain action, for example we might want instruct a motor to move to a certain position.

e.g. a motor moving to a certain position.

At the core of our instruction execution model lies two important data structures, the first being the instruction buffer and the second being the action list. The instruction buffer is a thread safe FIFO queue that contains integer instruction and integer arguments. The point is that new instructions gets added to the instruction buffer, which then gets emptied by a main 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 components method with the arguments. This cycle will continue as new instructions are put into the buffer and taken out and executed, it can be seen as an illustration in figure 7.2. To ensure that the instructions are run in sequential order there is only a single worker thread and if a component wants something executed in parallel it will have the responsibility of spawning a new thread.

The actions in the action list are directly registered by the individual component. On the startup of the EvoBot software the configuration file is loaded and every component is initialized. After every component have been initialized, the

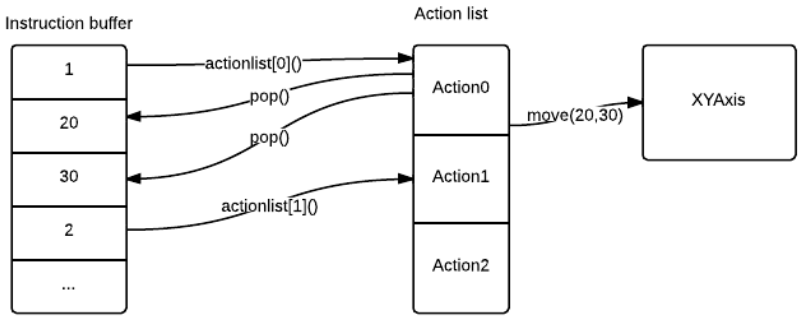


Figure 7.2: Instruction buffer and action execution cycle.

software loops over them asking them to registered their actions in the action list. The placement of a specific action in the action list is therefor determined by the order of the components in the configuration file, as well as how many actions each component have and also in which order a component have defined it's actions. Every action is simply a function that takes a pointer to the instruction buffer, when the action is invoked it pops the amount of arguments it needs from the buffer, the action definition is defined in the components implementation. The code for an action can be found in the specific component often defined as a simple lambda expression that pops from the instruction buffer and wraps a normal method.

duplikeret info

normal method? Jeg er ikke sikker på, at den sidste sætning er interessant på målgruppen

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 ~~we want~~, an observation that can be made is 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. ~~Differently~~ 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 a limited implementation it could be and ~~issue~~ at a later stage.

desired feature

The solution in this state 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

~~So now~~ 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 possibility 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 including some data. We want to be able to listen to this output and propagate it to the user. In other words we want an event based system.

I'm not sure event system is equival

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 ~~this pointer~~ a pointer to this function in the Splotbot (main) class in a variable, making it possible to change the function at a later stage. Every component gets initialized with a pointer to the function in the Splotbot (main) class, allowing them to call the function when needed and to ensure that updating the event function will propagate to the components. Our event system therefor 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 we 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 defined as a single string, making it the responsibility of the receiver to parser the string and handle the data.

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 and the only way to extended the event binding is through changing the C++ source code.

7.4 Programming experiments

Given the users the option of defining a complete experiment

~~Making the user capable a completely~~ defining an experiment requires us to look back to figure 7.1. Here we can see that firstly the user must be able to control the loop, more specifically the initial experiments instructions, the event reaction and the reactive instructions that must be fired off based on the event output. The initial ~~experiments~~ instructions are actually already available by directly coding in instructions codes. We however want to make it more human friendly, so the natural option was to introduce a programming language to the EvoBot, making the user capable of programming a complete experiment and making it run on the robot.

?? user

Hvad gør dette til 'natural option'? Hvilke andre muligheder har vi?

But ~~However~~ we must first address the question of what language and why. An initial thought could be to ~~be using~~ 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 DSL ~~instead~~.

7.4.1 Rucola

Jeg tager en search replace på therefor

Introducing Robotic Universal Control Language, or in short Rucola. The language is designed to fit our exact needs and are therefor build around two core features, calling component actions and listening for events. Rucola exists as a sub folder in the ~~main~~ C++ code, where it functions as a black box. The interaction between Rucola is compiling a string with code into instruction and being able to invoke events for more instructions. We use Flex ("flex: The Fast Lexical Analyzer" 2014) as lexer and Bison as parser ("Bison - GNU Parser Generator" 2014) generators. Rucola takes a string creates an AST and compiles that into instruction code.

Beskrivelsen af interaktionen er fo

Rucola compiler

komma komma

it integer

The basic language features of Rucola consists of arithmetics, variables, conditionals and a print statement. The variables binding is global and is stored internally in Rucola even between compilation and event calls, this can however 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, therefor it simply takes a string to be printed and a tuple of ~~variables~~/expressions. Example of the different constructs can be seen in below:

komma

punktum

An

```
a = 20
b = a - 10

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

print "Variables" (a,b)
```

Calling component actions are somewhat trivial in our current model. What we do is that we first make every component register ~~there~~ its actions in a map, where the key is the actions name and the value is a struct containing the actions instruction number and how many arguments it takes. We then store the action map in a map mapping from the component name to the action map. When a specific component is called with a specific action and arguments, we simply retrieve the action's instruction number and the amount of arguments it takes via the map structure, we can therefor both check that its a valid component

map in a map mapping ...

ejefald forebeholdes normalt dyr og mennesker

the instruction number of the action

punktum

and action as well as checking if the amount of arguments are 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 can be found below:

```
Component.action(1,2)
```

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 don't 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. If upon called an event is bound in Rucola we compile it into instructions and put it on the stack. If an event isn't bound we simply return an empty list of instructions.

For events to be really reactive we must also introduce event arguments. In a system where everything is integers, event arguments are also ~~simply~~ integers. Our approach to introducing this into the EvoBot event model is to extended the amount of arguments an event in EvoBot takes, from event name and data to also include a list of integers, these will only be used in Rucola. The integers will be bound to variables by the user at the events call time, this is done by taking the integer list and binding them to the names in the Rucola event's internal variable map. It is then up to the user to decide the names and the amount of variables he want to include into his scope, it is all about the amount of variables available and the order of them. An example of an event binding can be found in the code below:

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

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. An other 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 it self unless assigned to another 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 the EvoBot. The experiment is designed to measure the time that goes between something ~~have happen~~ in the physical world until the EvoBot have processed and reacted ~~to~~ it.

En liste ville give mere overblik (og konsistents med de andre kapitler)

The experiment is designed as follows. We bound an event in the EvoBot system using Rucola, the event is designed to watch the droplet speed and when it gets above a certain limit it will move the servo motors. The experiment is then to move the cameras xy axis to get the movement speed of the droplet above the limit and make the stepper motors move. We then time the time from the actual movement of the xy axes to the movement of the stepper motor and record this. This should give us an idea of the reaction time EvoBot can have on an actual event. Below is first the test program used and then the resulting table.

punktum

Der er flere ting, der ikke er

Kan vi ikke skrive al det her i enter

```
servoPos = 0
Servo1.setPosition(0)
Servo2.setPosition(90)

BottomAxes.home()
BottomAxes.setPosition(10, 10)

(Camera_dropletspeed: speed) -> {
    if (speed > 10) {

        print "Speed " (speed)

        servoPos = (servoPos % 90) + 10
        Servo1.setPosition(servoPos)
        Servo2.setPosition(90 - servoPos)
    } else { b = 3 }
}

Camera.mode(2)
```

Alting skal være figurer med captions. Eller

Trial	Time (Seconds)
1	05:54
2	05:40
3	05:37
4	07:04
5	06:44
6	06:41
7	05:70
8	04:94
9	07:30
10	06:02

Trial	Time (Seconds)
11	06:04
12	05:61
13	06:79
14	05:92
15	05:88
16	06:45
17	07:91
18	05:76
19	05:85
20	04:74
21	05:41
Average	**06:02**

For uformelt

So an average of almost 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, somewhat stable in time. The real question here is whether these results are acceptable for actual experiments. For the kind of slow droplet experiments the Splotbot was capable of handling, we do believe these times would be acceptable for experiments.

7.6 Summary~~/Conclusion~~

To make the EvoBot capable of running autonomous experiments, we extended the EvoBot software with the capabilities to sending instructions to components and we made it possible for components to communicate events with data. We then made it possible to program experiments.

Jeg synes are dette afsn

To allow for execution of instructions we introduced an instruction buffer. This buffer holds instructions and parameter values. An action list is created at startup to contain all of the possible component actions available as functions, an instruction number is the index to the action list. On execution the buffer takes the instruction and uses it to call an action from the list of actions. An action will then take the parameters from the buffer and do a method call to a component. Combined the instruction buffer and action list allows a user to defined a set of instructions and values to execute a none responsive experiment.

We then introduced events to the system, allowing a component to communicate when something have happen. All components will be initialised with the same pointer to an event function. They can then use the function to send an event with event data.

Finally we completed the system with a programming language that allows the user to define autonomous experiments. The language includes basic features such as arithmetics, variables, conditionals and a print statement. More importantly the language also contains the possibility of calling components actions, which will result in instructions and values being put on the instruction buffer. To make the language capable of making autonomous experiments, it also contains a possibility for binding and listening for events and event data and use it to call more component actions.

Combined these features allows the user to define an entire reactive and autonomous experiment in the programming language and get it run on the EvoBot using the instruction buffer and event system.

Chapter 8

Logging experiment data

An important part of creating a platform for running experiments is to ensure that the user can understand the experiment's outcome after the experiment have 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.

In this chapter 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 and finally we discuss the issues faced with the limited space on the EvoBot.

Goals

8.1 ~~The goal~~

designing, running, and analysing the result data of experiments

Biological experiments usually involves making experiments and analysing the experiment and the results. EvoBot's goal is to support every aspect of running automated experiments and therefor gathering data to understand the experiment and result is an important feature of the EvoBot. For the initial logging system of the EvoBot we looked at the basic goals EvoBot have to atleast fulfill to support the analysis of experiments:

EvoBot does not have a g

its

Besluttede vi ikke at være konsist

Vi skal lige finde ud af det fælles goal format. Punkter, ingenting eller bold tekst?

- All the data needed from an experiment is saved.
- ~~That~~ the data is structured in a way that makes it possible to analyse it.
- ~~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 and will not be discussed in this chapter, but the logging of data is core to the EvoBot so we wanted to build it into our solution.

which is not in scope of this

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
- Computer vision related data, droplet speed etc.
- 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.

Data gathered through computer vision techniques

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, so each component creates it's own entries which we can later filter from the overall data set to focus on what exactly a component is doing. While also being able to filter on a specific component type. ??

Image and video data is especially essential to our logging system as even with computer vision our software will not be able to obtain every detail of what is happening in the image. We therefor want to ensure that at any time camera data is available it will be saved. That is not to say that textual based data isn't just as valuable, ~~being able to create statistic from number is a powerful tool~~. ??

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. Therefor it was natural to build the logging framework as a class structure as shown in figure **??**. In our system 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 therefor 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 will take a specific kind of entry and persists it.

punktum

logger types:

In our current implementation we support three different kind of formats video, images and files. 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, punktum

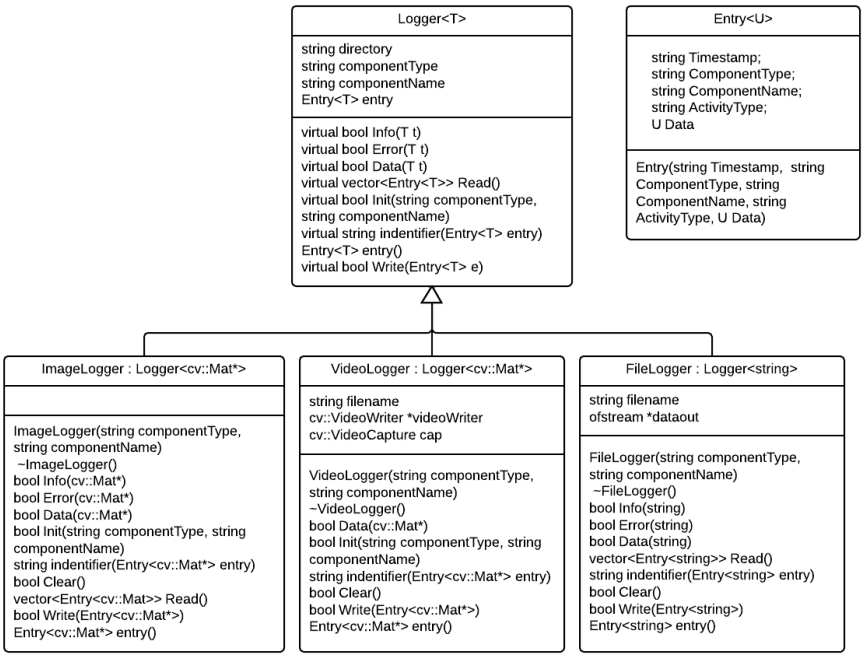


Figure 8.1: Class diagram for logging structure

making a new video logger will yield a new video file. An image logger will create a new image file from each entry given. Both the 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 it's ease of implementation and support in programs such as Microsoft Excel ("Microsoft Excel" 2014).

Currently in EvoBot the logging is integrated into multiple components and we log data such as videos, scanned images and some data. The logging is however not as extensively used as we would like and improvements can be made to include logging more widely in the program. We would also like to have more options for the user to decide how data should be logged, requiring different kind of loggers, possibly allowing the user to pick options such as databases etc.

But this is trivially added at a later point, we didn't have time

Vi siger her, at vi vil have er

Would we, or would we just like to have some user input about what they actually want to have?

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 time stamp. 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 have logged.

been

as

- The component name, 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.

These are for isntance, error, data, info

Disk. Disc er kun i discman :D

is

The data is saved to the ~~disc~~ in multiple files. The structure of these are a single folder in which every file gets stored. All of the video and image files contains the meta data in the name so they can be filtered accordantly. The entries in the CSV file also contains all of the meta data as well as the data as a string.

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.

Vi bør nævne, at vi har valgt fra at kigge på det

8.5 Use of limited hard disk space

only has

The Beagle Bone Black (BBB) on which the EvoBot system runs, ~~there is only~~ 16GB of hard disc (SD Card) space available currently. This limited hard disc space 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. I won't 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.

Location.

This is not true. I can think of many schemes where one would blindly write to the network

This leads us to introducing an other storage ~~place~~. An important part of any storage scheme would be to first store the data locally on the BBB and then move it to some place else, ensuring that data is not lost in the transfer. We see two options available.

- 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.

Should we mention something abou

A notable thing is also 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

To make it?

In order to make

~~Making~~ 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 the 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.

Chapter 9

Jeg er forvirret. Dette kapitel er slet ikke opdateret til at følge den nye struktur som vi diskuterede og som også står på Trello. Det er e

Human interaction with the robotic platform

Few software project 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 for the lowest common denominator and preferably make it easy to extend at a later time. At the same task, end-users are likely to have advanced use-cases in mind for the EvoBot, making it relevant to not limit the usage of the platform with the user interface. This 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 ~~formal~~ goals marked with **bold**.

The main value that EvoBot seeks to provide its users stems from the top level goal that it **must provide the user with the ability to control as well as receive feedback from it**. 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 the 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**, as this allows for the scientist to monitor an experiment while it is running. One could imagine e.g. that the user wished to run an experiment without knowing exactly what ~~to expect~~. By allowing

outcome/result

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 further 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 the 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.

Sproget i det her afsnit virker meget anderledes end resten af rapporten. Sådan lidt flyvsk og langt.

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 easy of use, we set the goal of 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.

An 'experiment' can be many things, and it is not possible to foresee all the different experiments the user might need to run. We therefore find it necessary that the **user can program experiments making use of all functionality of EvoBot**. For this we pose a requirement on the user that she has a certain level of EvoBot proficiency. Such proficiency could for instance be the ability to write in a programming language understood by the EvoBot, as to give ourselves a fair chance of not trying to cope with providing both very high usability while keeping the complexity of the functionality. Sticking to what we are familiar with, programming languages, is a way of limiting the scope of the project.

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 (or how I learn to stop worrying and love python/gcode)

The splotbot keeps its heritage as a 3D printer, as outlined in (Gutiérrez 2012, 43–48) all interactions to the splotbot goes directly through G-Code. This means that in order for a user to use the Splotbot she has to be proficient with G-Code or use a piece of software designed for this purpose. In the thesis, Pronterface is shown as an example of this. In the Gutiérrez (2012) thesis is also an example of a Python script written to perform an experiment. It pulls the level of interaction up from G-Code and into more Splotbot related tasks such as "moveWater".

Igen har vi overset at Splotbot har et python library der pakker G-Code væk. Så man skal bare kunne kode Python.

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. There is a clear upside in the fact that G-Code is a generally used format, and as such creating a client on top of the Splotbot will be a matter of implementing G-Code and not necessarily tightly coupled with the splotbot. This means that one could build on a platform such as Pronterface to create a suitable client. The python script in Gutiérrez (2012) thesis does just that, but it in turn imposes the requirement to know python on the user. It also does not provide much in terms of continously interacting with the experiments, as the experiments are designed statically, and the user has no way to interact while the experiments run.

9.3 User interface in EvoBot

As stated in the 9.1 section, the EvoBot **must run without special technical setup**. This goal ~~as~~ 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 the 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 as a prerequisite for EvoBot that such a platform is present on the client PC, and the GUI can therefore safely be implemented in web technologies. Furthermore there is an informal requirement regarding usage consumption. It is therefore advantageous to move as much as possible of the computation to the computer of the client, which, incidentally is also done by 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 user that runs on her computer
- Have a communications layer run on BeagleBone, 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 **??**.

9.3.1 Construction of the graphical user interface

The actual elements shown in the GUI is determined from the config file as described in ??. 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 component in the GUI. An example could be a set of X/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, give their current position, or, in the case of a camera, can give direct visual feedback as seen in figure 9.1b.

(a) The control of a component

(b) Camera feedbacl

The graphical controls do, however, not help with solving the goal of programming experiments. It is instead achieved by introducing a domain specific language for controlling the robot as described in chapter 7. But the graphical user interface can aid the user in programming in this language. Therefore, 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 and furthermore without requiring the user to install anything on her computer.

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

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

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

9.3.2 The bootstrap process

Using the 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. By running our software as a service it starts up when the BeagleBone boots, allowing the user to simply power the board to start the program and make the EvoBot ready for use. This will require a technician only if something goes wrong in the process.

The second part where the user must connect to the robot have, however, introduced difficulties, as connecting to EvoBot requires two things:

- The user is on the same network as the EvoBot
- The user knows the IP of the 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, but the process differs already between Mac and Linux[1]. 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 from a tablet for instance. When using a regular PC instructions for opening a browser and pointing it to the correct URL will still have to be provided.

[1] the scripts used on both Mac and Linux is available in our util repository ("Utilities Repository" 2014)

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.

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 frontend client as it provides all the features (and more) that was seen as requirement. Furthermore there exists tools around it which can help with generating the files and structure, compensating for the teams lack of knowledge in the UI field.

The communication layer is written in NodeJS as it is built with great support for web clients while boasting support for low-level interaction with C and C++. It has a third not insignificant advantage that the BeagleBone runs a variant of the language as its native scripting language. There are no plans to rewrite any parts of the robot-interacting code in JavaScript, but its reassuring to know that the platform is natively, and likely actively, supported.

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 the EvoBot.

9.3.4 Summary of the design

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

9.4 Discussion

Det er da første gang vi har en diskussion, passer formatet til resten af rapporten?

Though thought was put into the design decisions, each of the decisions have shown to have certain drawbacks, which will be discussed here.

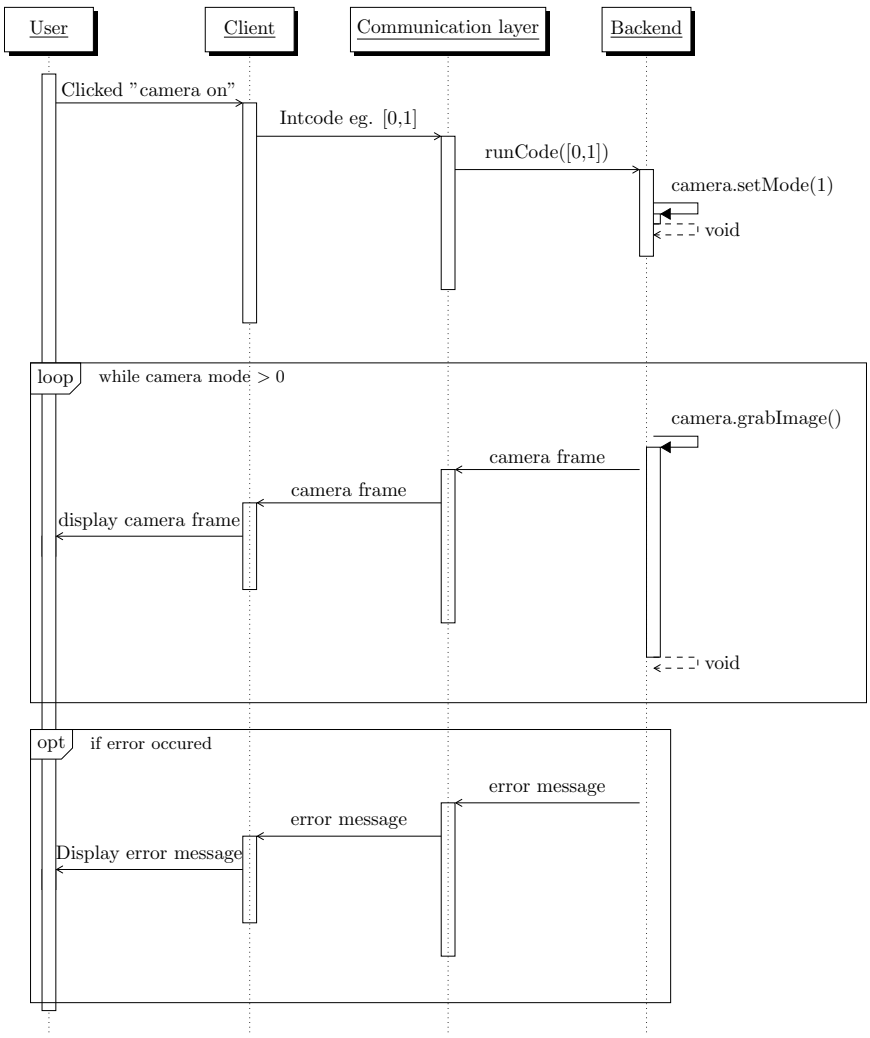


Figure 9.4: A sequence diagram showing the communications between the communication layer and the web client during startup and usage.

9.4.1 Construction of the graphical user interface

The design decision about construction graphical controls from the components defined in the configuration file is actually one with which we have found to problems in our own use of the EvoBot. We have 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.

?

One limitation we have considered, though, is concerning 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 user. But the implementation of these is considered out of scope of this project.

Vi nævner to ting i flæng her. Måsk

9.4.2 Choice of technologies

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:

komma

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

Splotbot løste ikke noget.
Det gjorde Juan

Splotbot solved the cross-platform difficulties in a different manner, namely by using a cross platform GUI library (such as GTK, QT, Java-Swing, etc). But they all require software to be installed locally, possibly alienating some users by making it more difficult to get started using the EvoBot.

As for the specific choice of AngularJS, it was done mostly based on:

- prior experience within the team
- the fact that it is a fairly stable framework with some years behind it
- it has large corporate backing by Google and by a large online community

Plenty alternatives exists for angular, mainly divided into two categories:

- Similar JavaScript frameworks
- Languages that compile to JavaScript with a framework around it

In the first category, there are plenty to choose from, but they are all either younger/less mature and/or has less support/learning resources. The second category is exciting because JavaScript is not the favourite language of any of us. The thought of changing it for something better is very appealing, and there seems to be one for every style of programming:

- Lisp: Clojurescript, SchemeToJs
- Functional: Elm(Haskell-subset), js_of_ocaml (OCaml)
- Imperative: Dart, llvm-to-js, Go2js

All of these would make for an exciting project on their own, and could provide some much missed features to the world of web-programming. Their are also not all esoteric, as some of them has large backing and communities around them. However, we deemed this approach too much of a risk to pick up and use for this project, as we lack experience with them.

Making the application support a RESTful/web socket interface made us think about the possibilities with C++, while C++ is great at system programming it does not propose the best support for creating a web service. So instead we choose to go for newer technologies and went looking for a language with interoperability with C++ and good support for creating a web service.

Kortere og mindre følelsesladent. Men bliver nok bedre af at man skifter format til at mixeimplementation og diskussion.

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 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.

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 just is not ideal. The thing that caused the most trouble was using callbacks between the two languages, which resulted in using a work around using curl in lieu of getting the advertised functionality to work.

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 for from the underlying datatypes 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 likely could have been kept in a more familiar way.

9.4.3 Alternative solutions and improvements

We believe that the current solution lives very well up to the goals defined, and unless further issues with the design are found, we recommend keeping this way of providing a graphical user interface. Changes to other parts of the EvoBot are likely to bring greater value than changes to the user interface.

9.5 Summary

Hele dette kapitel skal nok opdateres i format. Overvej at følge strukturen på Trello kortet.

The EvoBot is not much good if no one can access and control it. The Splotbot solves this with low level interactions with the motor, as well as an early example of an alternative interface in the form of a python script. EvoBot is aiming for very high availability and is designed with a web-based user interface consisting of a frontend client and a REST based webserver which serves the frontend code as well as communicates with the backend. This allows the EvoBot to be used from any modern web browser.

Specific problems such as initial connection with the EvoBot have been handled, and the goals of achieving control of the full capabilities of the robotic platform without technical setup is achieved.

Chapter 10

Conclusion

//TODO

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.

"Polulo 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.