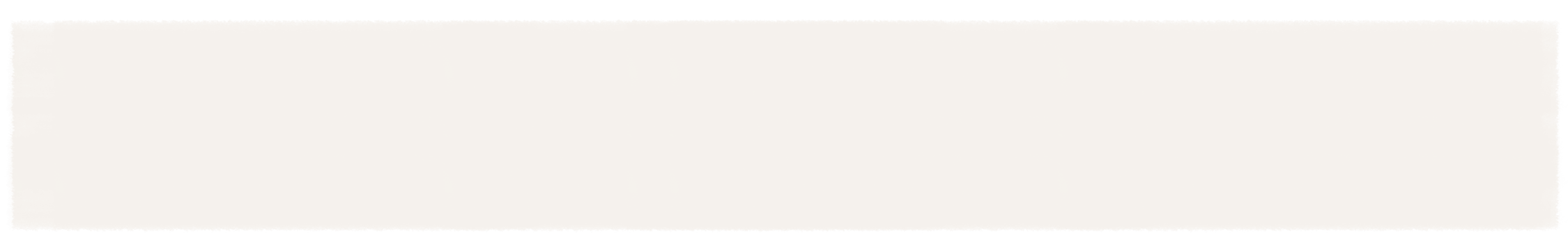
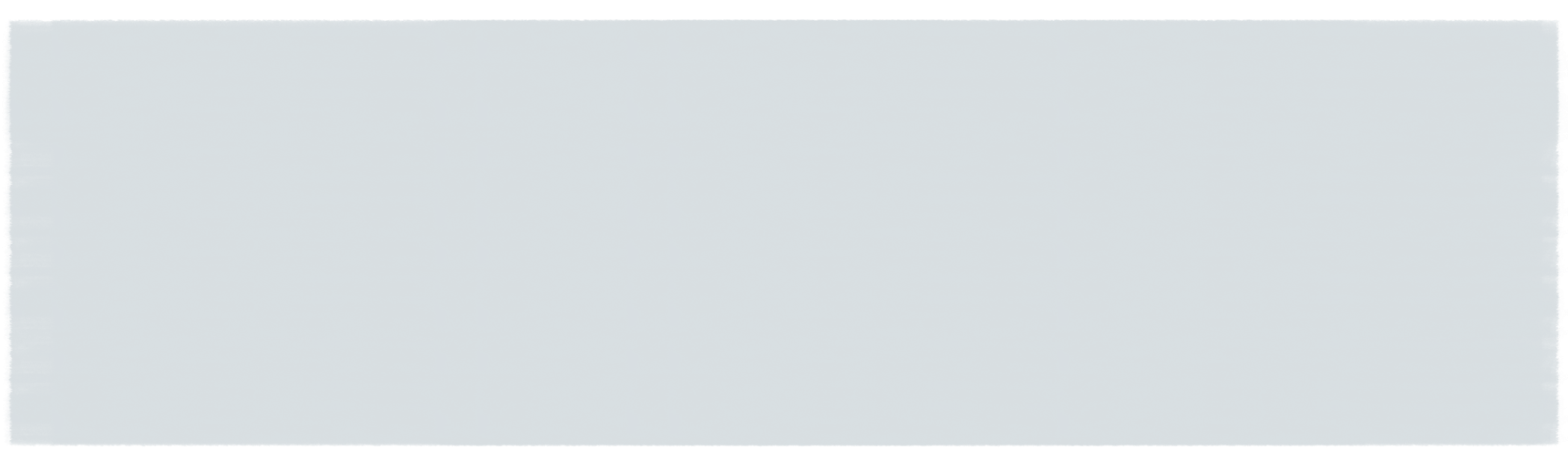
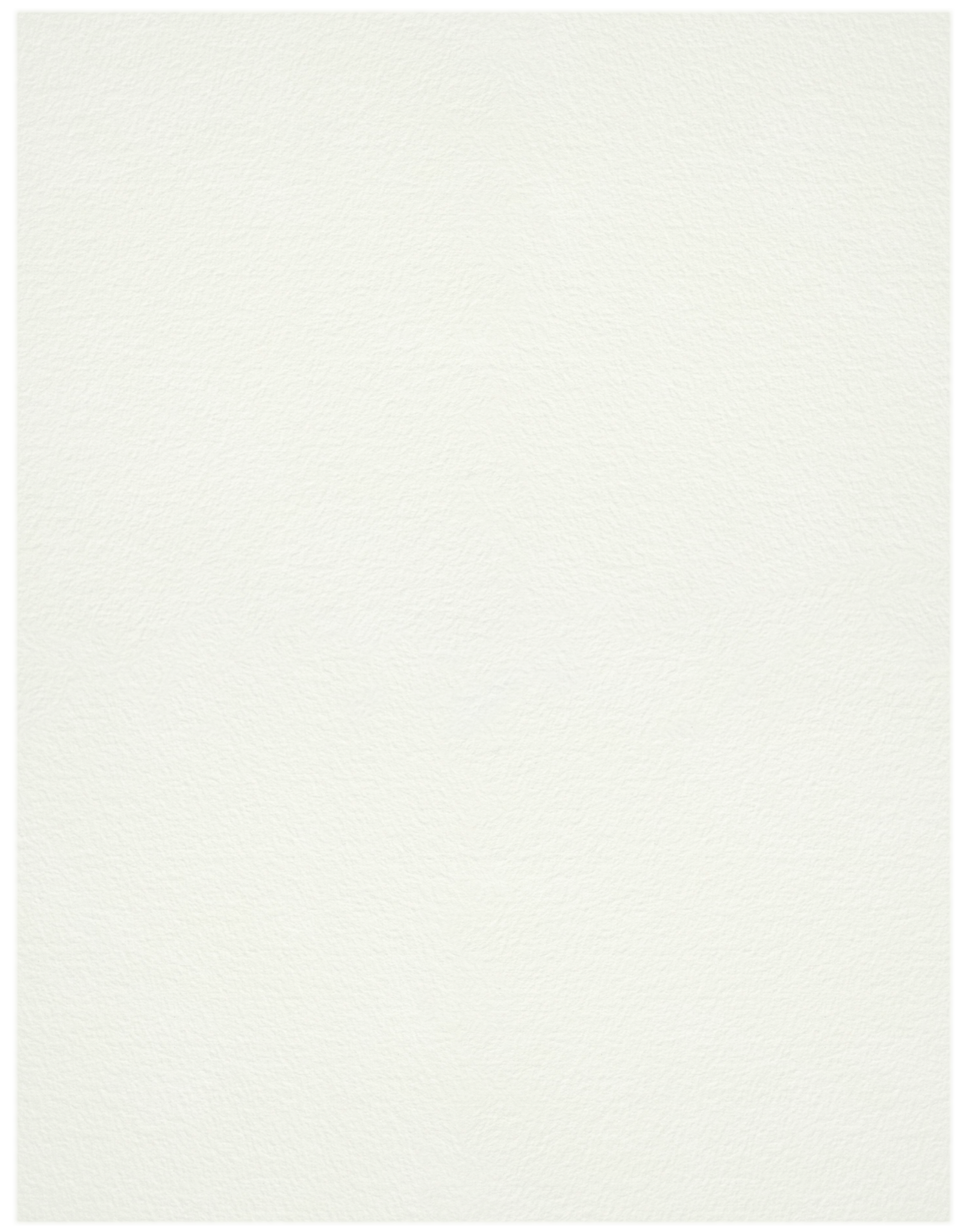
**Khepera project**



**In the context of maze solving**

**Xavier Bouthillier**

Table of Contents

Introduction 5

Tapir Modules 6

Introduction 6

Orientation histogram detector 6

Introduction 6

Detection of the object and his orientation 6

Improvements 6

How to use 7

Orientation color detector (Color detector and Triangle tracker) 8

Introduction 8

Detection of the object and his orientation 8

Improvements 8

How to use 8

Edit 9

Colored grid detector 10

Introduction 10

--------- (Algorithm) 10

Improvements 10

How to use 10

Maze detector 12

Introduction 12

--------- (Algorithm) 12

Improvements 12

How to use 12

Multiple object shared memory 14

Introduction 14

--------- (Algorithm) 14

How to use 15

CLsquare modules and Maze-Khepera Plant 15

Introduction 15

Multiple object shared memory tapir detector (class) 15

Introduction 15

--------- (Algorithm) 16

How to use 16

Maze-Khepera data (class) 16

Introduction 16

--------- (Algorithm) 17

Improvements 17

How to use 17

MazeReward module 18

Introduction 18

How to use 18

Directed Khepera (class) 18

Introduction 18

Communication 18

Navigation 18

Improvements 19

How to use 19

MazeKhepera Plant module 20

Introduction 20

Gather information 20

Pilot the Khepera 20

Improvements 20

How to use 21

How to make an experiment 21

Maze 21

Khepera 21

CLsquare 22

## Abstract

The main goal of this document is to provide a description of the modules

# Introduction

## What is Khepera

The Khepera III is small differential wheeled robot (diameter of 5.5cm). It has eleven infrared sensors, nine around it and two on the bottom. Additionally, it has five ultrasonic sensors around it. It can communicate with Bluetooth and is equipped with a battery that provides him approximately four hours of autonomy. There is a commercialized device called KoreBot II that can provide the Khepera III limited computing capacity.

As the research in the lab is directed toward expensive machine learning algorithms, the limited computing capacity of the KoreBot II was not interesting. It would be more efficient to use the computing power of a dedicated computer and communicate with the Khepera III through Bluetooth.

The actual researches of Manuel Blum do not explicitly concern problems of mobile robotics. Thus, it has been decided that the first environment in which the Khepera III will evolve would not take use of the sensors, but of an external detector that keeps track of the robot position and orientation. The main idea is simply to solve a physical problem based on the knowledge from a virtual training on this problem.

The CLS^2 system already includes a maze plant where it is possible to solve the shortest-path problem from a given position to a given target. As I will concentrate on building the modules needed to detect and pilot the Khepera III, we will use this plant so I do not have to worry about the problem itself (solving a maze).

In a first time, I will build a module to detect the position and the orientation of the Khepera III with the help of a camera hanging from the ceiling above the maze. In a second time, I will create a navigation system for the robot with the help of the position detector. In a last step, I will build a module to get a virtual discretized description of a physical maze.

In the first chapter of this document, I will explain the modules built for the detectors inside of the Tapir system. In the second chapter, I will explain every classes and modules I had to build for the CLS^2. Then in the third chapter, I will explain how to set-up and run an experiment with the Khepera III and a maze using the tools explained in previous chapters.

# Tapir Modules

## Introduction

The Tapir system is used to detect objects within images. I will use it to detect the Khepera III position and orientation and to build the discretized description of the maze as well. For the maze description, you can see it as the detection of every block (and their type) of the maze seen as a grid.

I first built a detector based on the histogram detector. This detector proved to be too unstable and difficult to manage when ambient lightning change. I then developed a simple color detector that proved to be more stable and easily manageable in this environment.

Then, I developed a simple colored grid detector to discretize an image and get the mean color of each block. Based on this detector, I built the maze detector that is designed to detect more specifically given colors according to the maze definition (wall color, empty space color, etc). Finally, I built a module to output multiple objects over shared memory, as I needed this for the maze output.

## Orientation histogram detector

### Introduction

I first built a detector to detect the robot position and orientation. To do so, I used a histogram detector to detect three dots and calculate the angle of the height of the triangle. In order to work properly, two points must be clearly closer to form a triangle that is near to be isosceles. (drawing)

### Detection of the object and his orientation

The detector first detects blobs with the help of the histogram detector. When this is done, it iterate through the blobs and selects the three biggest ones. It then selects the two closer points and calculates the line from D to C to get the angle theta, which is the orientation of the object

The position of the object is the mean position of the three points.

### Improvements

The detector is weak to noise. If one of the dots is not correctly detected and get smaller than some noisy blobs, the noisy blob is detected as one of the dots and the calculated position is not good.

We could keep the three points in memory and take the closest points at the next detection. It could still be unstable if the movement is fast and noise is closer than the next position. The best option may be to weight the distances with the size of the blobs.

The detector was to unstable because of small changes in ambient light. Also, we needed to remove the Khepera from under the camera every time we restarted the detector, because the histogram should not contain colors of the Khepera. So I decided to build a new detector that would detect specific colors. This way, the detector would be more reliable or could me more easily reconfigured during ambient light changes. Also, we could also leave the Khepera under the camera all the time.

The Orientation histogram detector is not included in Tapir because the Orientation color detector works better on every aspects, so the one is useless.

### How to use

1. Find a color that is well detected with the detector in the desired environment.
2. Put 3 dots of the found color on the object you want to detect (square of colored paper seems to fare better than colored dots with pencils on white paper)
3. Get the object out of the vision
4. Start the detector and wait a few second until it stops adding frames to the histogram
5. Add the object to the environment.

#### Parameters config file

* examples\_init – number of frames added to the histogram at the beginning
* examples\_renew – number of frames added to the histogram when pressing ‘+’
* box – draw box around all blobs in the image
* area – (min,max) define minimum and maximum size to detect objects (objects smaller than min or bigger than max are not detected)
* filename – filename where to save histogram (optional)
* autoload – load the histogram saved in ‘filename’ at the beginning

#### Keys:

* + - add ‘examples\_renew’ frames to the histogram
* ‘ ‘ (space) – add frames to the histogram until ‘ ‘ (space) his pressed again.
* ‘s’ – save the histogram in ‘filename’
* ‘l’ – load the ‘filename’ histogram

## Orientation color detector (Color detector and Triangle tracker)

### Introduction

As the first detector was not stable enough, I built a second detector that detects dots within a given color range. Three dots are selected and it then calculates the angle of the height of the resulting triangle. In order to work properly, two points must be clearly closer to form a triangle that is near to be isosceles. (drawing)

### Detection of the object and his orientation

The detector enhance the colors given the \_saturation and \_brightness values. It then detects blobs using the opencv function, cvInRangeS(). Once the blobs are detected, it selects the three biggest blobs. The two closer blobs are used to calculate the line from middle point D to C. The angle theta is the orientation of the object.

The position of the object is the mean position of the three points

### Improvements

There is less noise than in the previous detector, but it is still weak to noise. If one of the dots is not correctly detected and get smaller than some noisy blobs, the noisy blob is detected as one of the dots and the calculated position is not good.

We could keep the three points in memory and take the closest points at the next detection. It could still be unstable if the movement is fast and noise is closer than the next position. The best option may be to weight the distances with the size of the blobs.

The drawback compared to Histogram detector is that we need to find the best color range at the beginning of every set of experiment, because the ambient light may be different from a day to another. When this is done, it is a lot more stable than histogram detector but it is still vulnerable to changing of ambient light. In overall, the changes in the color range at the beginning of a new set of experiment are small or inexistent.

The big advantage compared to Histogram detector is that we do not need the get the object out of the vision every time we start again the detector.

### How to use

1. Before choosing a color, you may test which one is easier to detect with this detector, given the color in the environment.
2. Put 3 dots of the found color on the object you want to detect (square of colored paper seems to fare better than colored dots with pencils on white paper)
3. Start the detector. The object can be already in the vision or not, it doesn’t matter for this detector.
4. Use the keys to change the color range in order to find the best color range for a stable detection
5. Print the color range to the screen, remember it and write it down in the configuration file so you do not have to modify it at each start.

#### Parameters for configuration file

* box : draw box for detected blobs (default true)
* area : min max minimal and maximal blob sizes
* color : R G B (mean color)
* wide : R G B (ranges)
* saturation : level of saturation (default 0)
* brightness : level of brightness (default 0)

#### Keys:

* 4 => R+ 1
* r => R- 1
* 5 => G+ 1
* t => G- 1
* 6 => B+ 1
* y => B- 1
* 7 => Rrange+ 1
* u => Rrange- 1
* 8 => Grange+ 1
* i => Grange- 1
* 9 => Brange+ 1
* o => Brange- 1

* b => change brightness mode
* upkey => Brightness+ 1
* downkey => Brightness- 1
* s => change saturation mode
* upkey => Saturation+ 1
* downkey => Saturation- 1
* p => Print color/range or brightness or saturation based on current mode
* e => Get out of brightness/saturation mode (to color/range mode)

### Edit

During the refactoring of the modules to make them compatible with the new Tapir interface, Thomas Lampe have separated the detection and blob selection process. He placed the selection process in a TriangleTracker, which makes more sense since it is a tracking process. The detector was renamed ColorDetector. The major improvements to stabilise more the detection of the triangle will now lie in the TriangleTracker rather than in the ColorDetector.

## Colored grid detector

### Introduction

I needed to get a discretized description of the environment. To do so, I built a detector that discretizes the image and calculates the mean color of every block of the grid. The detector uses a given list of colors and finds the closest colors of this list to the mean color of every block. This can be used to assign types to blocks as in a maze (walls, open space, target, etc).

### --------- (Algorithm)

1. The detector enhance the colors given the \_saturation and \_brightness values.
2. It calculates the mean color of every block.
3. It draw the mean color as a thick border inside blocks of the raw image
4. It calculates the closest color from the mean block color
5. It draws the block color in the edited image (under the raw image)
6. It add blocks to the detected objects :

* x,y (upper left corner)
* angle=0, size=color index

1. Draw the grid on the raw an edited image
2. If in get\_block\_color mode, it draws the mean color of a given block in the edited image.
3. Ok(?)

### Improvements

This is not a major problem, I would even say less serious than minor, but the grid is incorrectly drawn when we change the min max values of the subframe. It does not happen on start, only on live changes. It is sometimes annoying, but it does not impact the performance of the detector, neither the usability of it.

It could be useful to change the blocks information mapping. The color index is saved in the size attribute, but we could save the actual size of the block, the length of the side, in this attribute. Still, for rectangular blocks there would miss an attribute to save both side lengths. Even though, the color index could be saved in the angle attribute, which I think will never be useful in this context.

### How to use

1. Set the xmin, xmax, ymin and ymax values using the arrows with to narrow down the subframe around the maze. Be sure that the terminal as the focus. Print on screen the value, write it down and save them in the configuration file.
2. Use the get\_block\_color mode to get the color values (RGB) of specific blocks, so you can give more accurate colors to detect in the configuration file.
3. You can adjust the brightness and saturation values to get better results

#### Parameters config file

* grid\_x – int, Number of blocks in x
* grid\_y – int, Number of blocks in y (default = grid\_x)
* x\_min – int, Lower limit of the detection frame on the image
* x\_max – int, Upper limit of the detection frame on the image
* y\_min – int, Left limit of the detection frame on the image
* y\_max – int, Right limit of the detection frame on the image
* print\_block\_color – book, Print the mean color of the block on the raw image
* get\_block\_color - bool, Enable the get\_block\_color on start
* get\_color\_x – int, Initial x position of get\_block\_color block
* get\_color\_y – int, Initial y position of get\_block\_color block
* saturation – int, Modify saturation of the image
* brightness – int, Modify brightness of the image
* colors – int,int,int|int,int,int|..., given colors to detect in RGB form

#### Keys:

* ‘l’ : change minimum (left,up) bounds mode
  + left : y\_min -= 2
  + right : y\_min += 2
  + up : y\_min -= 2
  + down : y\_min += 2
  + p : print x\_min,x\_max,y\_min,y\_max
* ‘u’ : change maximum (right,down) bounds mode
  + left : y\_max -= 2
  + right : y\_max += 2
  + up : x\_max -= 2
  + down : x\_max += 2
  + p : print x\_min,x\_max,y\_min,y\_max
* ‘b’ : change brightness mode
  + left, down : -5 to brightness
  + right, up : +5 to brightness
  + p : print brightness value
* ‘s’ : change saturation mode
  + left, down : -2 to saturation
  + right, up : +2 to saturation
  + p : print saturation value
* ‘c’ : get\_block\_color mode
  + left : move get\_block\_color block to the left
  + right : move get\_block\_color block to the right
  + down : move get\_block\_color block down
  + up : move get\_block\_color block up
  + p : print get\_block\_color block position and comparisons (distance) with mean color of the block and the given colors
* ‘e’ : exit current mode

## Maze detector

### Introduction

The colored grid detector worked well to detect mean colors of the blocks, but it was too unstable in some circumstances, around the target for example. The mean color sometimes alternates quickly between two or three mean colors. To fix this, I built a subdetector that has two method of detection. First it detect the mean color and the closest mean color as for the Colored grid detector, but if the closest mean color is not close enough (distance bigger than max\_distance\_to\_color) then the detected color is the closest one between the wall color, by default black, and the ground color, by default white.

### --------- (Algorithm)

1. The detector enhance the colors given the \_saturation and \_brightness values.
2. It calculates the mean color of every block
3. It draw the mean color as a thick border inside blocks of the raw image
4. It calculates the closest color from the mean block color
5. Set block color
   1. If the closest color is closest than a given threshold, block color is set to the closest color value
   2. If the closest color is more far than the given threshold, block color is set to closest color between wall color and ground color
6. It draws the block color in the edited image (under the raw image)
7. Add blocks to the detected objects
   1. x,y (upper left corner)
   2. orientation=0, size=color index (ground=0,wall=1)
8. Draw the grid on the raw an edited image
9. If in get\_block\_color mode, it draws the mean color of the (get\_block\_x,get\_block\_y) block in the edited image.

### Improvements

The same improvements of the colored grid detector apply to this one.

### How to use

1. Find colors that are easy to detect. Typically black and white for the walls and the free space, and a basic color like red for the target. An easy way to build it is with red and black tape on a white board.
2. You can build the maze before, but it is easier too build it while the detector is on, so you can keep track of how well it is detected.
3. Once the camera is placed above the maze or the empty board, set the xmin,xmax,ymin,ymax values using the arrows to narrow down the subframe around the maze. Be sure that the terminal as the focus. Print on screen the value, write it down and save them in the configuration file.
4. Use get\_block\_color mode to get the color values (RGB) of the wall, the ground and the target. Wall and ground colors are black and white by default.
5. You can adjust the brightness and saturation values to get better results

#### Parameters config file

* grid\_x – int, Number of blocks in x
* grid\_y – int, Number of blocks in y (default = grid\_x)
* x\_min – int, Lower limit of the detection frame on the image
* x\_max – int, Upper limit of the detection frame on the image
* y\_min – int, Left limit of the detection frame on the image
* y\_max – int, Right limit of the detection frame on the image
* print\_block\_color – book, Print the mean color of the block on the raw image
* get\_block\_color - bool, Enable the get\_block\_color on start
* get\_color\_x – int, Initial x position of get\_block\_color block
* get\_color\_y – int, Initial y position of get\_block\_color block
* saturation – int, Modify saturation of the image
* brightness – int, Modify brightness of the image
* colors – int,int,int|int,int,int|..., given colors to detect in RGB form
* ground - int,int,int, color of the ground (default = 155,155,155)(white)
* walls – int,int,int, color of the walls (default = 0,0,0)(black)
* max\_distance\_to\_color - int, maximal distance to color before it is counted as ground of walls

#### Keys:

* ‘l’ : change minimum (left,up) bounds mode
  + left : y\_min -= 2
  + right : y\_min += 2
  + up : y\_min -= 2
  + down : y\_min += 2
  + p : print x\_min,x\_max,y\_min,y\_max
* ‘u’ : change maximum (right,down) bounds mode
  + left : y\_max -= 2
  + right : y\_max += 2
  + up : x\_max -= 2
  + down : x\_max += 2
  + p : print x\_min,x\_max,y\_min,y\_max
* ‘b’ : change brightness mode
  + left, down : -5 to brightness
  + right, up : +5 to brightness
  + p : print brightness value
* ‘s’ : change saturation mode
  + left, down : -2 to saturation
  + right, up : +2 to saturation
  + p : print saturation value
* ‘c’ : get\_block\_color mode
  + left : move get\_block\_color block to the left
  + right : move get\_block\_color block to the right
  + down : move get\_block\_color block down
  + up : move get\_block\_color block up
  + p : print get\_block\_color block position and comparisons (distance) with mean color of the block and the given colors
* ‘e’ : exit current mode

## Multiple object shared memory

### Introduction

The Tapir Sharedmemory output module only shared one object, by the previously build detectors, colored grid detector and maze detector needed to share multiple objects at the same time.

The objects will be saved in a shared memory segment at a given index (key). This address (key) can be used in another program to fetch the information.

### --------- (Algorithm)

1. Allocate a shared memory segment with shmget during initialization
2. Assign an index of the shared memory segment for the list of mutexes
3. Assign an index of the shared memory segment for the list of objects
4. During send, for each object\_i in objects:
   1. It locks the mutex\_i of the object\_i
   2. Saves the object\_i
   3. Release the mutex\_i

#### Memory structure

On the machine I used, sizeof(mutex) was 40 bytes and sizeof(double) 8 bytes. I could have used only one mutex for all objects, but it would have slowed down the process. In overall, I never needed to share more than 100 objects. For every objects, there is 5 double, for the slower version it would take:

100 \* 5 \* 8 +40 =~ 4000 Byte = 4 KB

For the faster version, we add a mutex for every object, thus it takes:

100 \* (5\*8 +40) = 8000Byte = 8KB

It takes 2 times more space, but 8KB is still really small, so it is not a problem. Faster is better.

Mutex 1

…

Mutex n

item 1 x

item 1 y

item 1 size

item 1 angle

item 1 runtime

…

item n x

item n y

…

### How to use

Define the number of object that needs to be send and write it in the configuration file. Define a unique key\_id and key\_path combination that will be used in the tapir detector in CLsquare.

#### Parameters config file

* key\_path – pathname for the key of shared memory segment (read about ftok())
* key\_id – key for the shared memory segment (read about ftok())
* nb\_objects – the number of objects to be send

# CLsquare modules and Maze-Khepera Plant

## Introduction

* What is the goal?
  + Find and drive the Khepera through a maze to the target
* How is it achieved?
  + Get a Discretized description of the maze and pass it through the MazePlant so we do not have many modifications to do
  + Move the Khepera according to the maze positions in the MazePlant with the help of a Detector for the Khepera position and orientation.

## Multiple object shared memory tapir detector (class)

### Introduction

The Tapir detector sends multiple objects for the maze detection. I needed to built a TapirDetector class to receive those objects from shared memory.

The saved objects will be read at the memory address given by the pathname and the keyid.

### --------- (Algorithm)

1. «Connect» to a shared memory segment with shmget
2. Assign an index of the shared memory segment for the list of mutexes
3. Assign an index of the shared memory segment for the list of objects
4. During update, for each object\_i in objects:
   1. It locks the mutex\_i of the object\_i
   2. Reads the object\_i
   3. Release the mutex\_i

#### Memory structure

Mutex 1

…

Mutex n

item 1 x

item 1 y

item 1 size

item 1 angle

item 1 runtime

…

item n x

item n y

…

### How to use

Set the number of objects, key\_path and key\_id as set in the configuration file in tapir.

#### Parameters config file

* key\_path – pathname for the key of shared memory segment (read about ftok())
* key\_id – key for the shared memory segment (read about ftok())
* nb\_objects – the number of objects to be send

## Maze-Khepera data (class)

### Introduction

From the detector, we have a discrete description of the maze, but it is still unreadable for the maze plant. For make use if it, I wrapped the Mazedata class that defines the maze structure used by the maze plant. The wrapper takes in parameter the Tapirdetector, reads the blocks from the detector and writes a maze definition file that MazaData could read.

### --------- (Algorithm)

1. It first get the maze description from TapirDetector
2. It calculates the maze boundaries (Xmin,Xmax, Ymin,Ymax, X\_extend and y\_extend):
   * Min and max are calculated regarding minimum and maximum positions of the blocks (Do not forget that the detector can detect a subframe of the image, thus min and max values are important)
   * Since we have upper-left position of the blocks (ref), xmin and ymin correspond to exact xmin and ymin values but xmax and ymax are the upper-left position of the max block. Thus, real xmax and ymax are xmax+grid\_x\_width,ymax+grid\_y\_width \*\*\*\*\* need to check
   * X\_extend,y\_extend = grid\_x,grid\_y (number of different x positions and different y positions)
3. It builds a grid to save the maze info
4. It loops through the blocks and save the type info in the grid
5. Once the grid is build, it saves it in a file as a maze definition.
6. MazeData is constructed using the saved file maze

### Improvements

The maze is saved in a grid before writing it in a file. We could have saved the blocks directly from the list to the file without making a grid, but this way we can save it serially from the grid to the file. As it is right now, the objects are already in-order when they are send from the MazeDetector, so we could save it directly in a file without saving it in a grid. But by doing this, an in-order list of objects would be a prerequisite. This way, we could, for whatever reason, send not in order objects and it would work as well.

We could have transferred the data directly to MazeData without saving it in a maze definition file. We could have inherited from MazeData to do so. But we would need to modify some part of MazeData as they are not private but protected. Also, MazeReward and MazeGraphic rely on the maze definition file to build their maze definition. They could have use the new MazeKheperaData, but it is better if we can avoid modifying more modules.

Whenever someone wants to modify the mapping of the block information (x=x,y=y,size=color index), the lines to modify are in the loop of the save\_maze() function. See the improvements subsection of colored grid detector for more information on this.

### How to use

The maze detector must be on. You need to pass the corresponding configuration file for the detector to the class and give a file name to save the maze.

(make a graph for this)

X = (blocks[i].x-x\_min)/ (x\_max-x\_min)/(x\_extend-1)

Y = (blocks[i].y-y\_min)/ (y\_max-y\_min)/(y\_extend-1)

Use can then use the functions get\_RX() and get\_RY() to get the (X\_i,Y\_i) values of the image from the (X\_g,Y\_g) coordinates of the grid. Also, you can use get\_centered\_RX() and get\_centered\_RY() to get the (X\_i,Y\_i) in the center of the block in the image from the (X\_g,Y\_g) coordinates of the grid.

## MazeReward module

### Introduction

If we use Plant as reward module, the mainloop of CLsquare try to destroy the plant 2 times at the end of the «experiment». It tries to destroy the plant, and latter the reward module, but the reward module is the same as the plant, so the program crash because the plant is already destroyed.

So I decided to build a dummy reward module. It returns reward 1 whatever are the state and the action. It loads the maze from maze.def to know where is (are) the goal(s) so it can evaluate the function is\_terminal().

### How to use

Be sure that the maze definition file is named maze.def. The MazeKheper plant should do this by default. There is nothing to do other than this.

## Directed Khepera (class)

### Introduction

This class aims to pilot the Khepera to a given (x,y) position. To do so, we get the position and orientation of the Khepera from a camera hanging from the ceiling. It needs to calculate the needed orientation to drive to a (x,y) position and sends a command to the Khepera via a serialport to make it turn or drives forward.

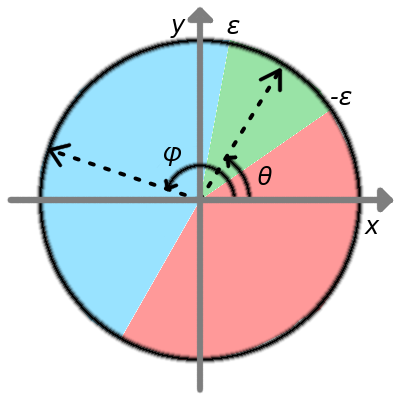
### Communication

It first opens a port for serial port communication. The serialport communicates via Bluetooth. In Ubuntu, it is usually “/dev/rfcomm0”.

To write commands to the buffer, it is imperative to add a carriage ‘\r’ at the end. If not, the Khepera will not recognize the command.

The buffer is read char by char. The robot always answers to commands. The buffer is emptied after every command that is sent.

### Navigation

1. It first get the position of the robot including the orientation.
2. If the detection is not successful and position is unknown, it sends a command to immobilize the robot
3. It then calculates the desired angle with desired\_angle = atan2(y-new\_y,x-new\_x)
4. It add PI to the calculated angle and the robot orientation to get range (0,2PI) rather than (–PI,PI)
5. Angle error = desired\_angle – robot\_angle
6. If the angle error is absolutely smaller than epsylon ( abs(angle\_error) < epsilon) it sends a forward command
7. If the absolute angle\_error is between epsilon and PI is sends a turn right command
8. Else send a turn left command
9. Do this while (x-new\_x)^2+(y-new\_y)^2 > epsilon^2

The turn right and turn left commands are graded on the level of angle error. To simplify calculations, negative angles are inverted by adding 2\*PI before calculating turning speed. The function of the speed is the following:

Speed = angle\*(\_max\_speed-\_min\_speed)/(PI-angle\_threshold) + (PI\*\_min\_speed-angle\_threshold\*\_max\_speed)/(PI-angle\_threshold)

### Improvements

There is no command for the robot for a definite angle rotation. It would be a great enhancement for the rapidity and accuracy of the robot movements if we could find a way to make it precise turns of a given angle with a single command. There is a command to make a precise turn of the wheels, but it doesn’t interpolate directly to a complete robot rotation of a precise angle. We would need to calculate how much the Khepera moves for a precise rotation of the wheels.

The update of the tapir detector and the time between read/write commands on the buffer of the serial port are really slow. The tapir detector seems to update values at a rate between 1 to 2 times per second, which is incredibly slow. It would be really helpful to speed up both processes to get higher accuracy in the navigation.

### How to use

The tapir detector must be on. Set parameters of the tapir detector in khepera\_tapir.cfg. To learn how to set up the Bluetooth connection, take a look at the section khepera the chapter How to make an experiment.

Constructor parameters

* PORT – port of the serialport. Usually “/dev/rfcomm0” in Ubuntu.
* Max\_speed, min\_speed (optional) - Be carefull if you set those parameters. The update time of the detector is slow and serial port communication too, so if the max\_speed is too high, there might be overruns.

Init parameters

* Manipulate - If true, the robot enters an infinite loop where you can send commands. Write “quit” to end, it then returns false and CLsquare ends.
* Delta\_t – The sleep time between each loop of the navigation algorithm. Delta\_t seems to need to be equal or higher than 1.0 second because of the time that take Tapir.update() and write/read serialport commands.

## MazeKhepera Plant module

### Introduction

We want to find the optimal paths through a maze and pilot the Khepera from his position to the target. To do so, the plant needs the maze information through the Tapir detector. The action chosen by the controller will be send to the DirectedKhepera that will navigate it to the target point with the help of the OrientationDetector (ColorDetector and TriangleTracker).

### Gather information

It first loads the maze with the help of MazeKheperaData. I don’t know why, but if the detector has run once, then the information will still be accessible via shared memory even If the detector is down.

If the parameter with\_robot is set to true, it will construct and initialize the robot. Otherwise, it will run as a classic MazePlant without Khepera.

If the parameter test\_robot\_block is set to true, the program enters an infinite loop and the user is asked to enter x,y position corresponding to the grid positions. This can be used to debug maze block positions. This option might be useless now that the plant is working properly.

### Pilot the Khepera

The action is chosen by the controller (as in MazePlant) and is represented by the next position (x\_g,y\_g) in the grid. The position is translated by MazeKheperaData to an (x\_i,y\_i) position in the image which correspond to the center of the grid block. This position (x\_i, y\_i) is send to DirectedKhepera that navigates the Khepera to this position.

### Improvements

The most annoying thing with this plant is when we need to abort the experiment while the robot is moving. We always need to keep a second clsquare running with the option manipulate\_robot=true to be able to send a stop command. It would be nice if the plant could catch an abort and send automatically a stop command to the Khepera.

### How to use

1. Set the maze detector. Be sure that the detection is correct.
2. Start frameview if it is not already running
3. Start CLSquare
4. If with\_robot=true, stop the maze detector when asked for and start the Khepera detector.

#### Parameters config file

* with\_robot – Use the Khepera robot during the experiment (default false)
* manipulate\_robot – Manipulate the Khepera with manual commands (D,l0,l0)
* test\_robot\_blocks – Send manually the Khepera to grid positions (x\_g, y\_g)

The manipulate\_robot and test\_robot\_blocks parameters prevent experiments. Clsquare stay locked in the init function and quit when the infinite loop of manipulate\_robot or test\_robot\_blocks is broken.

# How to make an experiment

## Maze

1. Find colors that are easy to detect. Typically black and white for the walls and the free space, and a basic color like red for the target. An easy way to build it is with red and black tape on a white board.
2. You can build the maze before, but it is easier too build it while the detector is on, so you can keep track of how well it is detected.
3. Once the camera is placed above the maze, set the xmin,xmax,ymin,ymax values using the arrows with the MazeDetector to narrow down the subframe around the maze. Be sure that the terminal as the focus. Print on screen the value, write it down and save them in the configuration file.
4. Use get\_block\_color functions to get the color values (RGB) of the wall, the ground and the target. Wall and ground colors are black and white by default.
5. You can adjust the brightness and saturation values to get better results
6. You are now done with the maze

## Khepera

1. Find a color that is easy to detect given the colors of the maze. Different textures or materials may have different effectiveness. Drawing color on a white paper seems really ineffective.
2. If you use the HistogrammOrientationDetector, you will need to get the Khepera out of the camera view every time you start the detector. If you use the OrientationColorDetector (ColorDetector and TriangleTracker), you can leave the robot in the view of the detector when you start it, it is not a problem.
3. Put the Khepera on.
4. In Ubuntu, open the Bluetooth administration panel. If Khepera do not appear in the detected devices click on search. If it still does not appear, the battery may not be full enough. Connect the Khepera on the charge and try again. It should appear if it is on and plugged on the charge.
5. Click on pair (key) if not paired. The access key is 0000
6. Click on setup to setup the serial port.
7. If it works, write down the port. (/dev/rfcomm0 most of the time)
8. If it does not work. Try the following : http://www.k-team.com/forum/index.php?topic=553.0
9. To test the connection and to be able to stop the Khepera if the CLsquare experiment fails, run ./CLsquare manipulate\_robot.cls
10. If the connection fails, try to use minicom as it is described here: http://www.k-team.com/forum/index.php?topic=553.0
11. Once the connection is correctly established, try moving the robot with commands D,l5000,l-5000. you can change values 5000 and -5000 for any values you want between -20000 and 20000.
12. The Khepera is now ready.

## CLsquare

1. Set starting positions in Maze.init,
2. Set khepera\_tapir.cfg and tapir.cfg (maze)
3. Set train.cls file, you can leave the plant part empty, you don’t want to train with the robot
4. Set with\_robot.cls file. Be sure that with\_robot=true is in the Plant module.
5. Start the maze detector
6. Start frameview in clsquare ./bin/frameview &
7. Start training with ./CLsquare train.cls
8. When training is done, you can start experimenting with Khepera. Before starting the experiment, be sure to have a CLsquare running with manipulate.cls so you can stop the robot if any problem arises. For example if CLsquare crash or if the robot fall out of the maze. You should always leave it running when you are experimenting with the Khepera. (D,l0,l0 to stop the robot)
9. When training is done, start the experiment with Khepera with ./CLsquare with\_robot.cls
10. When asked for, stop MazeDetector and start the detector for the khepera
11. The Khepera should now be on his way to the target