OBSTACLE DETECTION AND PATHFINDING FOR MOBILE ROBOTS

A THESIS SUBMITTED TO THE GRADUTE SCHOOL OF APPLIED SCIENCES OF NEAR EAST UNIVERSITY

By
MURAT ARSLAN

In Partial Fulfillment of the Reguirements for The Degree of Master of Science in Computer Engineering

NICOSIA, 2016

ONAY SAYFASI

ACKNOWLEDGEMENTS

Studying at the Department of Computer Engineering, working with a highly devoted teaching community, remain one of the most memorable experiences of my life. This acknowledgement is an attempt to earnestly thank my teachers who have directly helped me during preparation of my thesis.

I would like to take special privilege to thank my supervisor Prof.Dr. Rahib Abiyev who allocated me a thesis in the area of my interest. It was because of his invaluable suggestions, motivation, cooperation and timely help in overcoming problems that the work is successful.

ABSTRACT

In this thesis, obstacle detection via image of objects and then pathfinding problems of NAO humanoid robot is considered. NAO's camera is used to capture the images of world map. The captured image is processed and classified into two classes; area with obstacles and area without obstacles. For classification of images, Support Vector Machine (SVM) is used. After classification the map of world is obtained as area with obstacles and area without obstacles. This map is input for path finding algorithm. In the thesis A* path finding algorithm is used to find path from the start point to the goal.

The aim of this work is to implement a support vector machine based solution to robot guidance problem, visual path planning and obstacle avoidance. The used algorithms allow to detect obstacles and find an optimal path. The thesis describe basic steps of navigation of mobile robots.

Keywords: A*; image processing; motion planning; NAO; path finding; support vector machine

ÖZET

Bu tezde, objelerin resimlerinden, engel tanima ve yon bulma yöntemleri, insansi NAO robot kullanilarak uygulanmıştır. Objelerin fotoğraflarını çekmek için NAO robotun kamerası kullanılmıştır. Çekilen bu fotoğraf resim işleme teknikleri kullanılarak, engel olan ve engel olmayan olarak iki farklı şekilde sınıflandırılmıştır. Bu sınıflandırılma için Support Vector Machine (SVM) kullanılmıştır. Bu sınıflandırılmadan sonraki bilgiler yön bulma algoritmasının girdisidir. Bu tezde başlangıç ile bitiş noktasındaki yolu bulabilmek için A* yön bulma algortiması kullanılmıştır.

Bu çalışmanın amacı, görsel yol planlama ve engelden kaçmak için robota SVM tabanlı bir çözüm uygulamaktır. Kullanılan algoritmalar engelleri algılamak ve en uygun yolu bulmak için kullanılmıştır. Bu tez mobil robotların navigasyonunun temel adımlarını açıklamaktadır.

***Anahtar Kelimeler*:** A*; hareket planlama; NAO; resim işleme; support vector machine; yön bulma

TABLE OF CONTENTS

ACKNOWLEDGMENTS………………………………………………………... i

ABSTRACT…………………………………………………………………….. ii

ÖZET…………………………………………………………………………. iii

TABLE OF CONTENTS ………………………………….…………………... iv

LIST OF FIGURES…………………………………………………………… vi

LIST OF TABLES..…………………………………………………………… vii

LIST OF ABBREVIATIONS………………………………………….......... viii

CHAPTER 1: INTRODUCTION……………………………………………… 1

CHAPTER 2: LITERATURE REVIEW

2.1 Obstacle Detection.....…………….…………………………………………... 3

2.1.1 Naive bayes classifier ……………………………………………… 3

2.1.2 Artificial neural network………………………………………….... 4

2.1.3 Support vector machine ……………………………………………. 6

2.1.4 Decision tree learning …………………………………………….... 7

2.2 Pathfinding ……………………………………………………………….... 8

2.2.1 Artificial potential field………………………………………….. 9

2.2.2 Vector field histogram …………………………………………….. 11

2.2.3 Dijkstra’s algorithm………………………………………………… 12

2.2.4 A* algorithm………………………………………………...... 13

2.2.5 D* algorithm………………………………………………………….. 14

2.2.6 Rapidly exploring random tree…………………………….....….. 15

2.3 Related Works…………………………………………………………... 15

2.4 Summary………………………………………………………………… 17

CHAPTER 3: OBJECT DETECTION BASED ON IMAGE PROCESSING

3.1 Object Detection ……………………………………………………….. 18

3.2 Support Vector Machines………………………………………………. 18

3.2.1 Linear SVM…..……………………………………………………….. 21

3.2.2 Non-linear SVM……………………………………………………... 23

3.2.3 Multiclass SVM…………………………………………………..... 23

CHAPTER 4: PATHFINDING FOR MOBILE ROBOTS

4.1 A* Search Algorithm……………………………………………………. 26

4.1.1 Description………………………………………………………….. 27

4.1.2 Properties………………………………………………………….... 28

CHAPTER 5 : DESIGN OF SYSTEM

5.1 Hardware……………………………………………………………….... 31

5.2 Software…………………………………………………………………. 34

5.2.1 The NAOqi process……………………………………………... 35

5.3 System Design………………………………………………………….... 41

CHAPTER 6 : CONCLUSION…………………………………………...….. 56

REFERENCES……………………………..………………………………………. 58

APPENDIX: Source Code.………………………………………………………….. 64

LIST OF TABLES

Table 3.1: Error Rate of SVM and MLP For Text Verification.…………………….. 25

Table 4.1: Results ………………....………………………………………………. 30

Table 5.1: NAO Robot Parameters 34
…………………………………………………..

LIST OF FIGURES

Figure 3.1: Support Vector Machines…………………………………………….. 20

Figure 3.2: Multi-Class Support Vector Machines……………………………….. 25

Figure 4.1: A* Algorithm…………………………………………………… 28

Figure 4.2: A*, RRT, APF……………………………………………………... 30

Figure 5.1: NAO Robot ……………………………………………………. 32

Figure 5.2: NAO Robot Specifications…………………………………………….. 33

Figure 5.3: NAO Robot Cross-Platform System…………………………………... 35

Figure 5.4: NAOqi Libraries and Modules………………………………………… 36

Figure 5.5: NAOqi Libraries Modules and Methods………………………………. 37

Figure 5.6: NAOqi Memory System……………………………………………….. 38

Figure 5.7: Steps of the system…………………………………………………..... 42

Figure 5.8: Graphical User Interface……………………………………………... 45

Figure 5.9: Static Area with Obstacles………………………………………….... 46

Figure 5.10: Static Area After Perspective Correction…………………………… 47

Figure 5.11: Processed Area………………………………………………… 48

Figure 5.12: Dataset………………………………………………………… 49

Figure 5.13: Area in Binary Matrix Format………………………………………. 50

Figure 5.14: Area with Path……………………………………………………. 51

Figure 5.15: Perspective Correction Steps……………………………………….... 52

Figure 5.16: Preprocess Steps…………………………………………………….. 53

Figure 5.17: Processed Area to Matrix Format…………………………………… 54

Figure 5.18: Processed Area to Matrix Format…………………………………… 55

LIST OF ABBREVIATIONS

ANN:	Artificial Neural Network
APF:	Artificial Potential Field
HOG:	Histogram of Oriented Graphs
HSV:	Hue Saturation Value
LPA:	Label Propagation Algorithm
MLP:	Multi Layer Perceptron
NAO:	NAO is an autonomous, programmable humanoid robot
OSPF:	Open Shortest Path First
RGB:	Red, Green, Blue
RRT:	Rapidly Exploring Random Tree
SVM:	Support Vector Machine
VFH:	Vector Field Histogram

CHAPTER 1
INTRODUCTION

Nowadays robotics are actively use in our daily life. They started to apply in domestic, industry, game playing, army etc. Robots started to help human in many fields. They are helping to improve product quality and also capacity in industry.

One of important problem in robotics is the designing intelligent robots performing all the human actions in certain fields. For designing such intelligent robots using new soft computing techniques and science. The designing of intelligent robots, needs to design set of modules such as object detection, image processing, path planning, obstacle avoidance, motion control etc. problems.

Obstacle detection and obstacle avoidance are important problems in robot navigation. One way of detecting obstacles is the use of image recognition. Using image recognition the world map can be classified in area with obstacles and without obstacles. This information in future can be used for path-finding.

Robots are moving in unpredictable, cluttered, unknown complex and dynamic environments. In this environment, the avoidance of mobile robots from obstacles becomes an important problem. Many obstacle avoidance algorithms are proposed. "Bug" algorithms follow the edges of obstacles without considering the goal. They are time consuming. Artificial potential field (APF) is most commonly used method that utilizes attractive and repulsive fields for goals and obstacles, respectively. But the APF has several disadvantages: (a) when there are many obstacles in the environment the field may contain a local minima; (b) the robot unable to pass through small openings such as through doors; (c) the robot may exhibit oscillations inits motions. Vector Field Histogram (VFH) uses a two-dimensional Cartesian histogram grid as a world model and the concept of potential fields. The VFH algorithm selects a shorter path than bug algorithms but it takes more time to manipulate. Other goal oriented algorithms are dynamic window, “agoraphilic” and Rapidly-exploring Random Trees (RRT) algorithm is a faster algorithm and can be applied for pathfinding in dynamic environments. But frequently the path determined by RRTs may be very long. The RRT-smooth algorithm was proposed to shorten the RRT length. In this thesis A* search algorithm was used. A* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution for the one that incurs the smallest cost (least distance traveled, shortest time, etc.), and among these paths it first considers the ones that appear to lead the most quickly to the solution (Abiyev and others, 2015).

The aim of the thesis is the design of efficient algorithms for detection and avoidance of obstacles and also pathfinding, using NAO robot. The design of algorithms are based on image processing and decision making technique.

In robot guidance problem, some extra sensors, like ultrasound sensors, infrared distance sensors used to detect obstacles. These sensors do not give information about obstacle's shape and color. In this project, only camera was used to detect obstacles.

The design of a mobile robot that can navigate and localize obstacle in an unknown environment is based on visual ques such as a camera, path-finding is based on a navigation algorithm that final path for the robot to the goal.

This thesis is split into 5 chapters, conclusions, references and appendix

- Chapter 1 is introduction to pathfinding and image processing.
- Chapter 2 represents literature review on object localization and path-finding algorithms.
- Chapter 3 goes over the problem of detecting real world objects in images or series of images such as videos.
- Chapter 4 goes over the current techniques in the field of path finding.
- Chapter 5 covers the design of our mobile robot, given detailed information about hardware and software also about the design of the system.

Conclusions includes important results obtained from the thesis. In Appendix, source code is given with detailed comments.

CHAPTER 2
LITERATURE REVIEW

2.1 Obstacle Detection

Robot navigation includes obstacle detection, pathfinding and obstacle avoidance algorithms. Obstacle detection is an important step which includes set of algorithms.

Object detection is a computer technology linked to computer vision and image processing that deals with detecting objects such as humans, cars, building, obstacles etc. in digital images and videos.

In this thesis classification technique for obstacle detection is applied. Various machine learning algorithms are used for object detection some of which are outlined below.

2.1.1 Naive bayes classifier

Naive Bayes has been studied extensively since the 1950's. It was introduced under a different name into the text retrieval community in the early 1960's (Russel and others, 2003). It is a popular (baseline) method for text categorization, the problem of judging documents as belonging to one category or the other (such as spam or legitimate, sports or politics, etc.) with word frequencies as the features. With appropriate pre-processing, it is competitive in this domain with more advanced methods including support vector machines. (Rennie and others, 2003) It also finds application in automatic medical diagnosis (Rish, 2001).

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features.

Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in learning problem. Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

In the statistics and computer science literature, Naive Bayes models are kenned under a variety of denominations, including simple Bayes and independence Bayes (Hand and Yu, 2001). All these denominations reference the utilization of Bayes' theorem in the classifier's decision rule, but Naive Bayes is not (obligatorily) a Bayesian method (Rennie and others, 2003; Hand and Yu, 2001).

2.1.2 Artificial neural network

Warren McCulloch and Walter Pitts (1943) designed a computational model for neural networks predicated on mathematics and algorithms called threshold logic. This model paved the way for neural network research to split into two distinct approaches. One approach fixated on biological processes in the brain and the other fixated on the application of neural networks to artificial intelligence (McCulloch and others, 1943).

Artificial neural networks (ANNs) are a family of models inspired by biological neural networks (the central nervous systems of animals, in particular the brain) which are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. Artificial neural networks are typically specified using three things,

- Architecture specifies what variables are involved in the network and their topological relationships for example the variables involved in a neural network might be the weights of the connections between the neurons, along with activities of the neurons.

- Activity Rule Most neural network models have short time-scale dynamics: local rules define how the activities of the neurons change in response to each other. Typically the activity rule depends on the weights (the parameters) in the network.

- Learning Rule The learning rule specifies the way in which the neural network's weights change with time. This learning is usually viewed as taking place on a longer time scale than the time scale of the dynamics under the activity rule. Usually the learning rule will depend on the activities of the neurons. It may also depend on the values of the target values supplied by a teacher and on the current value of the weights.

For example, a neural network for handwriting recognition is defined by a set of input neurons which may be activated by the pixels of an input image. After being weighted and transformed by a function (determined by the network's designer), the activations of these neurons are then passed on to other neurons. This process is repeated until finally, the output neuron that determines which character was read is activated.

A key advance that came later was the backpropagation algorithm which efficaciously solved the exclusive-or problem, and more commonly the problem of fastly training multi-layer neural networks (Werbos, 1974).

In the mid-1980s, parallel distributed processing became accepted under the name connectionism. The textbook by David E. Rumelhart and James McClelland (1986) provided a full exposition of the use of connectionism in computers to simulate neural processes.

Neural networks, as utilized in artificial intelligence, have traditionally been viewed as simplified models of neural processing in the brain, even though the affiliation between this model and the biological architecture of the brain is debated; it's not clear to what degree artificial neural networks mirror brain function (Russel, 2012).

Support vector machines and other methods, much simpler such as linear classifiers slowly overtook neural networks in machine learning popularity.

2.1.3 Support vector machine

The original SVM algorithm was created by Vladimir N. Vapnik and Alexey Ya. Chervonenkis in 1963. In 1992, Bernhard E. Boser, Isabelle M. Guyon and Vladimir N. Vapnik suggested a way to create nonlinear classifiers by applying the kernel trick to maximum-margin hyperplanes (Boser and others, 1992). The current standard incarnation (soft margin) was proposed by Corinna Cortes and Vapnik in 1993 and published in 1995.

Support Vector Machine is a model under the Supervised Learning model of Neural Networks. The algorithm emphasises on analyze data and recognition patterns that are used for classification and regression analyses. SVM categorizes the training set into one of the two categories, SVM's Algorithm builds a model to assign new coming sets into one category or the other, making it a non-probabilistic binary linear classifier.

The dataset in the SVM model is represented by points in space. When new dataset come they classify by the side where there in. With the help of the "Kernel Trick" SVM can also perform nonlinear classification. SVM implicitly maps the inputs into p-dimensional feature spaces.

When data are not labeled, supervised learning is impossible, and an unsupervised learning is required, which attempts to find natural clustering of the data to groups, and then map new data to these formed groups. The clustering algorithm which provides an enhancement to the support vector machines is named support vector clustering and is usually used when only some data is labeled or data is not labeled as a preprocessing for a classification pass (Ben-Hur and others, 2001).

SVMs are useful in text and hypertext categorization as their application can significantly reduce the need for labeled training instances in both the standard inductive and transductive settings.

Categorization of images can also be performed by using SVMs. Experimental results show that SVMs have higher search accuracy than traditional query refinement schemes after just three to four rounds of relevance feedback. This is also true of image segmentation systems, including those using a changed version SVM that uses the privileged approach as suggested by Vapnik (Barghout, 2015).

The SVM algorithm has been commonly applied in the biological and other sciences. They have been used to classify proteins with up to ninety percent of the compounds categorized correctly. Permutation tests based on SVM weights have been suggested as a mechanism for interpretation of SVM models (Cuingnet and others, 2011). Support vector machine weights have also been used to interpret SVM models in the past.(Statnikov and others, 2006) Posthoc interpretation of support vector machine models in order to classify features used by the model to make predictions is a almost new area of research with special significance in the biological sciences.

2.1.4 Decision tree learning

Decision tree learning uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value. It is one of the predictive modeling approaches used in statistics, data mining and machine learning. Tree models where the target variable can take a finite set of values are called classification trees. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called regression trees.

Decision trees can also be seen as generative models of induction rules from empirical data. An optimal decision tree is then defined as a tree that accounts for most of the data, while minimizing the number of levels (or "questions").(Michalski and others, 2013) Several algorithms to generate such optimal trees have been devised, such as ID3/4/5, CLS, ASSISTANT, CART (Utgoff, 1989).

2.2 Pathfinding

Path planning has been one of the important problems in robotics. Path planning is finding a continuous collision-free path, from a start point, to a goal point or region, and obstacles in the space.

In a static and known environment, the robot knows the entire information of the environment before it starts moving. Because of this the optimal path could be computed offline before to the movement of the robot begins.

The path planning methods for a static, known environment are relatively mature. Representative path planning methods for known static environment include the Visibility Graph method (Lozano-Perez and Wesley, 1979), Voronoi diagrams method (Aurenhammer, 1991), the Cell Decomposition method (Sleumer and Tschichold-Gurman, 1999), the Potential Field method (Ge and Cui,2002) and Vector Field Histogram (Borenstein and Koren, 1991).

Visibility Graph is using in computational geometry and robot path planning, it is a graph of intervisible locations, typically for a set of points and obstacles in the Euclidean plane. Every node in the graph means a point location, and every edge represents a visible connection between them. If the line segment connecting two locations does not cross with any obstacle, an edge is drawn between them in the graph. When the set of locations lies in a line, this means as an ordered series. Visibility graphs have been extended to the realm of time series analysis.

Voronoi Diagram is a partitioning of a plane into regions predicated on distance to points in a concrete subset of the plane.That set of points is designated beforehand, and for each seed there is a corresponding region consisting of all points more proximate to that seed than to any other.

Cell Decomposition is that a path between the initial configuration and the goal configuration can be resolute by subdividing the free space of the robot's configuration into more minuscule regions called cells. After this decomposition, a connectivity graph, is constructed according to the adjacency relationships between the cells, where the nodes represent the cells in the free space, and the links between the nodes show that the corresponding cells are adjacent to each other. From this connectivity graph, a perpetual path, or channel, can be tenacious by simply following adjacent free cells from the initial point to the goal point.

Potential Field method's approach is to treat the robot's configuration as a point (customarily electron) in a potential field that amalgamates magnetization to the goal, and repulsion from obstacles. The resulting trajectory is output as the path. This approach has advantages in that the trajectory is engendered with little computation. However, they can become trapped in local minima of the potential field, and fail to find a path.

Also, the genetic algorithm, the simulated annealing algorithm, and other optimization methods have been used to obtain the optimal path for mobile robots. Davidor (1991) developed a custom genetic algorithm with a modified crossover operator to optimize robot path. Nearchou (1998) used the number of vertices produced in visibility graphs to build fixed length chromosomes in which the presence of a vertex within the path is indicated by setting of a bit at the appropriate locus. The method applied a reordering operator for performance enhancement, and the algorithm was capable of determining a near-optimal solution. Fan and others (2004) developed a fixed-length decimal encoding mechanism to replace the variable-length encoding mechanism and other fixed-length binary encoding mechanisms used in the genetic approach for robot path planning.

A sensor-based path planning method was proposed to help underwater robotic vehicles perform real-time path planning in a static and unknown environment (Ying and others, 2000).

2.2.1 Artificial potential field

The application of artificial potential fields for avoidance the obstacles was first created by Khatib. This design uses repulsive potential fields around the obstacles to push the robot away and an attractive potential field around goal to attract the robot. Therefore, the robot experiences a generalized force equal to the negative of the total potential gradient. This force runs the robot downhill towards its goal configuration until it arrives a minimum and it stops. The artificial potential field approach can be applied to both global and local methods (Janabi-Sharifi and Vinke, 1993; Park and others, 2001).

The potential force has two components: attractive force and repulsive force. The goal position produces an attractive force which makes the mobile robot move towards it. Obstacles generate a repulsive force, which is inversely proportional to the distance from the robot to obstacles and is pointing away from obstacles. the robot moves from high to low potential field along the negative of the total potential field. Consequently, the robot moving to the goal position can be considered from a high-value state to a low-value state.

The Artificial potential fields can be achieved by direct equation similar to electrostatic potential fields or can be drive by set of linguistic rules (Fakoor and others, 2015).

The artificial potential field methods provide simple and effective motion planners for practical purposes. However, there is a major problem with the artificial potential field approach. It is the formation of local minima that can trap the robot before reaching its goal. The avoidance of local minima has been an active research topic in potential field path planning. As one of the powerful techniques for escaping local minima, simulated annealing has been applied to local and global path planning.

The avoidance of local minimum has been an effective research topic in the APF based path finding. However, the previous solutions are limited to simple formations of obstacles or available for known environments. But Lee and Park designed a virtual obstacle concept is proposed as an idea to escape a local minimum. The imaginary obstacle is located around local minimum point to force the robot from the point. This technique is useful for the local pathfinding in unknown areas. The sensor based discrete modeling method is also planned for the simple modeling of a mobile robot with range sensors. This modeling is easy and good because it is designed for a real-time path planning (Lee and Park, 2003).

2.2.2 Vector field histogram

In robotics, Vector Field Histogram (VFH) is a real time motion planning algorithm proposed by Borenstein and Koren (1991). The VFH utilizes a statistical representation of the robot's environment through the so-called histogram grid, and therefore places great emphasis on dealing with uncertainty from sensor and modeling errors. Unlike other obstacle avoidance algorithms, VFH takes into account the dynamics and shape of the robot, and returns steering commands specific to the platform. While considered a local path planner, i.e., not designed for global path optimality, the VFH has been shown to produce near optimal paths.

The original VFH algorithm was based on previous work on Virtual Force Field, a local path-planning algorithm. VFH was updated and renamed VFH+ (Ulrich and Borenstein, 1991). The approach was updated again and was renamed VFH*(Ulrich and Borenstein, 2000). VFH is currently one of the most popular local planners used in mobile robotics, competing with the later developed dynamic window approach. Many robotic development tools and simulation environments contain built-in support for the VFH.

At the center of the VFH algorithm is the use of statistical representation of obstacles, through histogram grids (see also occupancy grid). Such representation is well suited for inaccurate sensor data, and accommodates fusion of multiple sensor readings.

The VFH algorithm contains three major components:

- Cartesian histogram grid: a two-dimensional Cartesian histogram grid is constructed with the robot's range sensors, such as a sonar or a laser rangefinder. The grid is continuously updated in real time.
- Candidate valley: consecutive sectors with a polar obstacle density below threshold, known as candidate valleys, is selected based on the proximity to the target direction.

- Once the center of the selected candidate direction is determined, orientation of the robot is steered to match. The speed of the robot is reduced when approaching obstacles head-on.

The VFH+ algorithm improvements include:

- Threshold hysteresis: a hysteresis increases the smoothness of the planned trajectory.
- Robot body size: robots of different sizes are taken into account, eliminating the need to manually adjust parameters via low-pass filters.
- Obstacle look-ahead: sectors that are blocked by obstacles are masked in VFH+, so that the steer angle is not directed into an obstacle.
- Cost function: a cost function was added to better characterize the performance of the algorithm, and also gives the possibility of switching between behaviors by changing the cost function or its parameters.

In VFH*, the algorithm verifies the steering command produced by using the A* search algorithm to minimize the cost and heuristic functions. While simple in practice, it has been shown in experimental results that this look-ahead verification can successfully deal with problematic situations that the original VFH and VFH+ cannot handle (the resulting trajectory is fast and smooth, with no significant slowdown in presence of obstacles).

2.2.3 Dijkstra's algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later (Dijkstra, 1959).

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other (Melhorn and Sandres, 2008). It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph show cities and edge path costs show driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest way between one city and all other cities. As a result, the shortest path algorithm is generally used in network routing protocols, most notably IS-IS and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's.

2.2.4 A* algorithm

AI researcher Nils Nilsson was trying to improve the pathfinding done by a robot in 1968, the robot that could navigate in a room with obstacles. This path-finding algorithm which is called A1, was faster than the best method, Dijkstra's algorithm, for finding shortest way in graphs. Bertram Raphael did some significant improvements on this algorithm, naming the revision A2. Then Peter E. Hart designed an argument that established A2, with only small changes, to be the best possible algorithm for finding the shortest paths. Rapheal, Hart and Nilsson developed a proof that the revised A2 algorithm was perfect for finding shortest ways under certain well-defined conditions.

This algorithm is generally used in pathfinding and graph travelsal, the process of plotting and efficiently traversable way between multiple points, named nodes. Noted for its performance and accuracy, it enjoys widespread use. On the other hand, in practical travel-routing designs, it is generally less performed by algorithms which can pre-process the graph to attain better performance (Delling and others, 2009), even though other works has found A* to be superior to other ways (Zeng and Church, 2009).

Hart and others (1968) first explained the algorithm. It is an extension of Edger Dijkstra's 1959 algorithm. A* shows better performance by using heuristics to guide its search.

2.2.5 D* algorithm

D* (pronounced “D star”) is any one of the following three related additional search algorithms:

- The original D*, Stentz (1995), is an informed incremental search algorithm.
- Focused D* is an informed incremental heuristic search algorithm by Stentz (1995) that combines ideas of A* (Hart and others, 1968) and the original D*. Focused D* resulted from a further development of the original D*.
- D* Lite is an incremental heuristic search algorithm by Koenig and others (2004) that builds on LPA*, an incremental heuristic search algorithm that combines ideas of A* and Dynamic SWSF-FP (Ramalingam and Reps, 1996).

All three algorithms solve the same assumption-based path finding problems, including planning with the freespace assumption, where a robot has to navigate to given coordinates in unknown terrain. It makes expectations about the unknown part of the terrain (for example: that it doesn’t contain obstacles) and finds a shortest way from its actual coordinates to the goal coordinates under these assumptions (Koening and others, 2003). The robot then follows the way. When it observes new map information (such as previously unknown obstacles), it adds the information to its map and, if necessary, plans a new shortest way from its current coordinates to the given goal coordinates. It repeats the process until it reaches the goal coordinates or determines that the goal coordinates cannot be reached. When traversing unknown terrain, new obstacles may be discovered frequently, so this planning needs to be fast. Incremental (heuristic) search algorithms speed up searches for sequences of similar search problems by using experience with the previous problems to speed up the search for the current one. Assuming the goal coordinates do not change, all three search algorithms are more capable than repeated A* searches. D* and its variants have been actively used for mobile robot and autonomous vehicle navigation. Current systems are typically based D* Lite rather than the original D* or Focused D*. In fact, even Stentz’s lab uses D* Lite rather than D* in some implementations (Wooden, 2006). Such navigation designs include a prototype system tested on the Mars rovers Opportunity and Spirit and the navigation system of the winning entry in the DARPA Urban Challenge, both developed at Carnegie Mellon University.

The original D* was submitted by Anthony Stentz in 1994. The name D* comes from the term “Dynamic A*”, because the algorithm behaves like A* except that the arc costs can change as the algorithm works.

2.2.6 Rapidly exploring random tree

A rapidly exploring random tree (RRT) is an algorithm designed to efficiently search nonconvex, high-dimensional spaces by randomly building a space-filling tree. The tree is constructed incrementally from samples drawn randomly from the search space and is inherently biased to grow towards large unsearched areas of the problem. RRTs were submitted by LaValle (1998). They easily handle problems with obstacles and differential constraints (nonholonomic and kinodynamic) and have been widely used in autonomous robotic path planning (LaValle and Kuffner, 2001).

RRTs can be viewed as a technique to generate open loop trajectories for nonlinear systems with state constraints. An RRT can also be considered as a Monte-Carlo method to bias search into the largest Voronoi regions of a graph in a configuration space. Some variations can even be considered stochastic fractals.

2.3 Related Works

Obstacle detection and obstacle avoidance are important problems for robot navigation. Many techniques and different algorithms have been used in this topic. Obstacle detection systems for mobile robots have included bump sensors, ultrasonic sensors, laser range finders and stereo vision. Ubbens and Schuurman (2009) propose a single camera feeding a support vector machine (SVM) classifier to autonomously navigate a ground-based mobile robot around obstacles. A single camera is mounted to the robot. SVM is trained to classify obstacles on different surfaces. Anything that is not recognizable on floor surface is classified as an obstacle. Images are preprocessed using a Fast Fourier Transform (FFT). The results were satisfactory. But small obstacles and obstacles that have similar intensity on the floor caused misclassifications. This problem can be solved by using another preprocessing techniques.

Similar techniques were used for aerial vehicles. Cooper and others (2016) developed a controller for a helicopter to detect obstacles and avoid them. They used live image sequence captured by the camera mounted in front of a helicopter as an input of the system. This system detects obstacles and suggests the best turning angles for the helicopter to avoid obstacles. Image was divided 64x64 pixels and manually labeled with 0 and 1 which represents obstacles and free space. They have developed a vision-based algorithm to detect obstacles from single image which is captured from the camera by using support vector machine (SVM) based on spectral signature. They were the first group applying spectral signature to solve obstacle detection. They achieved 3 frame-per-second detection rate and it is little bit slow. Test results show that they had %75 success rate for corridor and open area. More improvements needed to apply to the system.

Another techniques were applied for obstacle detection and avoidance. Ulrich and Nourbakhsh (2000) designed a wheel chair which has vision-based obstacle detection system by using single passive color camera. It works in real-time, and provides a binary obstacle image. This system is based on the appearance of individual pixels. Any pixel that differs in appearance from the ground as classified as an obstacle. But there are some assumptions like obstacle should differ from the ground which is relatively flat and no overhanging obstacles. In the first step 320 × 260 color input image is filtered with a 5 × 5 Gaussian filter to reduce noise. In the second step, RGB values are converted into the HSI (hue, saturation, and intensity) color space. In the third, hue and intensity values of the reference area are histogramed into two one-dimensional histograms, one for hue and one for intensity. In the last step, all pixels of the filtered input image are compared to the hue and intensity histograms and classified as an obstacle or ground.

As a result, there are some studies about this topic and various techniques were applied in similar ways. Some range-based obstacle sensors were used like, ultrasonic sensors, laser rangefinders and radars. Ultrasonic sensors are cheap but has poor resolution and effects from reflections. Lasers and radars have better resolution but they are complex and expensive. These sensors may help monocular vision systems for getting better results. In monocular vision based obstacle detection systems, some used poor preprocessing techniques to decreasing process times for real-time systems. But it causes misclassifications. Some tried to apply new techniques but new things need improvements as expected. Some did not use regular classification algorithms. These topics are open to discuss and one of them may works better than the others in some different environments. But none of them use path-finding algorithms to find the shortest and suitable way. In real time applications time and energy saving are important.

2.4 Summary

Path planning has been one of the important problems in robotics. Path planning is finding a continuous collision-free path, from a start point, to a goal point or region, and obstacles in the space. In previous works, a path is calculated by searching a graph or a grid of free spaces. In recent years, the randomized approaches such as Rapidly exploring random tree algorithm and Extended Rapidly exploring random tree algorithm appears to be successful in many practical applications which require high-dimensional motion planning.

All these above mentioned works are search-based. Another set of algorithms are called potential field these family of algorithms are more suitable for for real-time path planning domains.

In the field of pattern recognition, a variety of classification methods have been used. Among them, this work choose to use support vector machine (SVM) as a classifier. SVM is one of the powerful classifiers and has been successfully applied to many object recognition tasks such as 3D object recognition, face recognition, and pattern matching-based tracking. This paper describes our support vector machine-based path planner (called SVPP).

CHAPTER 3
OBJECT DETECTION BASED ON IMAGE PROCESSING

3.1 Object Detection

Object recognition is an important task in image processing and computer vision. It is the process of classifying a specific object in a digital image or video, which mostly means finding instances of real-world objects such as faces, cars, and buildings in images or series of images such as videos. This method is widely used in applications such as image retrieval, surveillance, security, and automated vehicle parking systems.

Humans recognize large range of objects in images with small effort, despite the fact that the image of the objects may vary somewhat in different viewpoints in many different sizes and scales or even when they are rotated. Objects can even be recognized when they are partially obstructed from view. This task is still a challenge for computer vision systems. Many approaches to the task have been implemented over multiple decades.

Object detection algorithms typically use extracted features and learning algorithms to recognize instances of an object category. Common techniques include edges, gradients, Histogram of Oriented Gradients (HOG), Haar wavelets, classification techniques and linear binary patterns.

Object detection is useful in applications such as video stabilization, automated vehicle parking systems, and cell counting in bio-imaging.

3.2 Support Vector Machines

In the context of machine learning, a support vector machine (SVM) is a supervised learning model and learning algorithms that is used to analyze data to create a model that can be used for classification and regression analysis.

It works by taking two sets of training data each of them marked belonging to one category of two categories, an SVM algorithm will build a model that will assign new examples into either one of the categories, making it a what is called non-probabilistic binary linear classifier.

An SVM model is a description of the samples in the training data as points in space, they are mapped so that samples of the separate categories are divided by a gap that is as large as possible. New samples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. This is linear classification.

With the help of a technique called a kernel trick a SVM can efficiently perform a nonlinear classification, implicitly mapping their inputs into high-dimensional feature spaces.

In addition to using SVMs for supervised learning (which means all sample data is labeled as belonging to one or the other category.) SVMs can be used to do unsupervised learning (using unlabeled data), which attempts to find natural clustering of the data into categories, and then new samples are mapped to these categories. This clustering algorithm which provides an improvement to the support vector machines is called support vector clustering.

Formally in order to do classification or regression, a SVM algorithm must constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space. The hyperplane with the largest distance to the nearest sample data point of any class (also known as functional margin) is selected to to achieve good separation. The larger the margin the lower the generalization error of the classifier.

It is often the case that the sets to classify are not linearly separable in that space. It was proposed that the original finite-dimensional space be mapped into a much higher-dimensional space, in order to make the separation easier in high dimensional space.

In order to make the computational load reasonable mapping used in SVM schemes are designed so that they ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function $\kappa(\chi,\gamma)$ selected to suit the particular problem. The hyperplanes in the higher-dimensional space are defined as the set of points whose dot product with a vector in that space is constant. The vectors defining the hyperplanes can be chosen to be linear combinations with parameters αi images of feature vectors χi that occur in the database. With this choice of a hyperplane, the points χ in the feature space that are mapped into the hyperplane are defined by the relation: $\sum \alpha ik(\chi i, \chi) = constant$. Note that if $\kappa(\chi,\gamma)$ becomes small as γ grows further away from χ to the corresponding data base point χi. In this way, the sum of kernels above can be used to measure the relative nearness of each test point to the data points originating in one or the other of the sets to be discriminated. Note the fact that the set of points χ mapped into any hyperplane can be quite convoluted as a result, allowing much more complex discrimination between sets which are not convex at all in the original space.

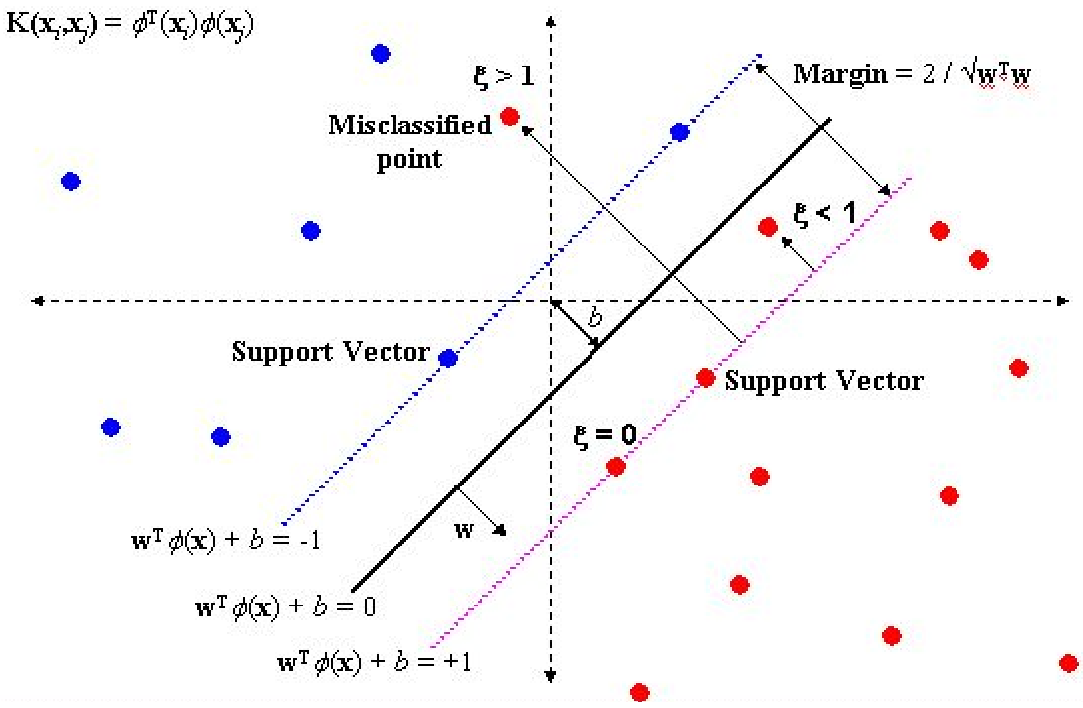


Figure 3.1: Support vector machines

3.2.1 Linear SVM

Given a training dataset of n samples in the form of $(\vec{x}\ 1,\ \gamma 1), \ldots, (\vec{x}\ \mathrm{n},\ \gamma n)$ where the γi are either 1 or -1 , each indicating the category to which the point $\vec{x}$ belongs to. Each $\vec{x}$ is a ρ-dimensional real vector.We want to find the "maximum-margin hyperplane" that divides the group of points $\vec{xi}$ for which $\vec{yi} = 1$ from the group of points for which $\vec{yi} = -1$, which is defined so that the distance between the hyperplane and the nearest point from either group is maximized.

A hyperplane can be written as the set of points $\vec{xi}$ satisfying the following $\vec{w} \,.\, \vec{x} + \mathrm{b} = 0$, where $\vec{w}$ is the normal vector to the hyperplane (can be non normalized). The parameter $\frac{b}{||\vec{w}||}$ determines the offset of the hyperplane from the origin along the normal vector $\vec{w}$.

Hard Margin: If the samples in the training dataset are linearly separable, one can select two parallel hyperplanes that separates the two classes of data, so that the distance between them is as large as possible. The region within these two hyperplanes is called the "margin", and the maximum-margin hyperplane is the hyperplane that lies halfway between them. These hyperplanes can be described using the equations

$$\vec{w} \,.\, \vec{x} + \mathrm{b} = 1 \tag{3.1}$$

and

$$\vec{w} \,.\, \vec{x} + \mathrm{b} = -1 \tag{3.2}$$

Geometrically, the distance between these two hyperplanes is $\frac{2}{\|\vec{w}\|}$, so in order to maximize the distance between the planes one wants to minimize $\vec{w}$. Also in order to prevent data points from falling into the margin, following constraints can be added: for each i either

$$\vec{w} \,.\, \vec{xi} + \mathrm{b} \geq 1 \text{ , if } \gamma i = 1 \tag{3.3}$$

or

$$\vec{w} \,.\, \vec{xi} + \mathrm{b} \leq -1 \text{ , if } \gamma i = -1 \tag{3.4}$$

These constraints ensure that every data point must be on the correct side of the margin. Which can be rewritten as:

$$\gamma i(\vec{w} \,.\, \vec{xi} + \mathrm{b}) \geq 1, \text{ for all } 1 \leq i \leq n \tag{3.5}$$

Above can be combined into a optimization problem:
"Minimize $\|\vec{w}\|$ subject to $\gamma i(\vec{w} \,.\, \vec{x} + \mathrm{b}) \geq 1$, for i = 1, … , n"
The $\vec{w}$ and b that solve this problem determine our classifier, $\vec{x} \rightarrow sgn(\vec{w} \,.\, \vec{x} + \mathrm{b})$.
An easy-to-see but important consequence of this geometric description is that max-margin hyperplane is completely determined by those $\vec{x}i$ which lie nearest to it. These $\vec{x}i$ are called support vectors.

Soft Margin: In order to use SVMs in cases where the data is not linearly separable, the hinge loss function can be used,

$$max(0, 1 - \gamma i(\vec{w} \,.\, \vec{x} + \mathrm{b})) \tag{3.6}$$

This function is zero if the constraint in the above is satisfied, which means that, if $\vec{x}i$ lies on the correct side of the margin. For data on the wrong side of the margin, the function's value is proportional to the distance from the margin. Then the following should be minimized

$$\left[\frac{1}{n}\sum_{i=1}^{n} max(0, 1 - \gamma i(\vec{w} \,.\, \vec{x} + b))\right] + \lambda \| \vec{w} \|^{2} \qquad (3.7)$$

where the parameter λ determines the trade off between increasing the margin-size and ensuring that the $\vec{x} i$ lie on the correct side of the margin. Thus, for sufficiently small values of λ, the soft-margin SVM will behave identically to the hard-margin SVM assuming that the input data are linearly classifiable, but should still learn a viable classification rule if not.

3.2.2 Non-linear SVM

Vapnik who constructed a linear classifier in 1963, proposed the original maximum-margin hyperplane algorithm. In 1992, Vladimir N. Vapnik, Bernhard E. Boser and Isabelle M. Guyon suggested a technique to design nonlinear classifiers by applying the kernel trick to maximum-margin hyperplanes. The result was formally similar, except that each dot product is replaced by a nonlinear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space. The transformation may be nonlinear and the transformed space high dimensional; although the classifier is a hyperplane in the transformed feature space, it may be nonlinear in the original input space.

Some common kernels include:

- Polynomial (homogeneous): $k(\vec{xi}, \vec{xj}) = (\vec{xi}, \vec{xj})^{d}$
- Polynomial (inhomogeneous): $k(\vec{xi}, \vec{xj}) = (\vec{xi}, \vec{xj} + 1)^{d}$
- Gaussian radial basis function: $k(\vec{xi}, \vec{xj}) = \exp(-\gamma \|\vec{xi} - \vec{xj}\|^{2})$, for $\gamma > 0$. Sometimes parametrized using $\gamma = 1/2\sigma^2$
- Hyperbolic tangent: $k(\vec{xi}, \vec{xj}) = tanh(k\vec{xi}, \vec{xj} + c)$, for some (not every) $k > 0$ and $c < 0$

The kernel is related to the transform $\varphi(\vec{xi})$ using the equation $k(\vec{xi}, \vec{xj}) = \varphi(\vec{xi}) . \varphi(\vec{xj})$. The value w is also in the transformed space, with $\vec{w} = \sum i\ \alpha i \gamma i \varphi(\vec{xi})$. Dot products with w for classification can again be computed by the kernel trick, i.e. $\vec{w}\ .\ \varphi(\vec{x}). = \sum i\ \alpha i \gamma i k(\vec{xi}, \vec{x})$.

3.2.3 Multiclass SVM

Multiclass SVM tries to assign samples into categories by using support vector machines, where the categories are chosen from a finite set of several categories.

The most common approach for doing so is to reduce the single multiclass problem into multiple binary classification problems. Common methods for such a technique include:

- Using a binary classifiers which distinguishes between one of the categories and the rest also called one-versus-all or between every pair of categories also called one-versus-one. Classification of new samples for the one-versus-all technique is done by using a winner-takes-all strategy, which means the classifier with the highest output function assigns the category. For the one-vs-one technique, classification is done by a max-wins strategy, which means that every classifier assigns the instance to one of the two classes, then the vote for the assigned class is increased by one vote, and finally the class with the most votes determines the instance classification.
- Error-correcting output codes
- Directed acyclic graph SVM (DAGSVM)
- Crammer and Singer proposed a multiclass SVM method which casts the multi class classification problem into a single optimization problem.

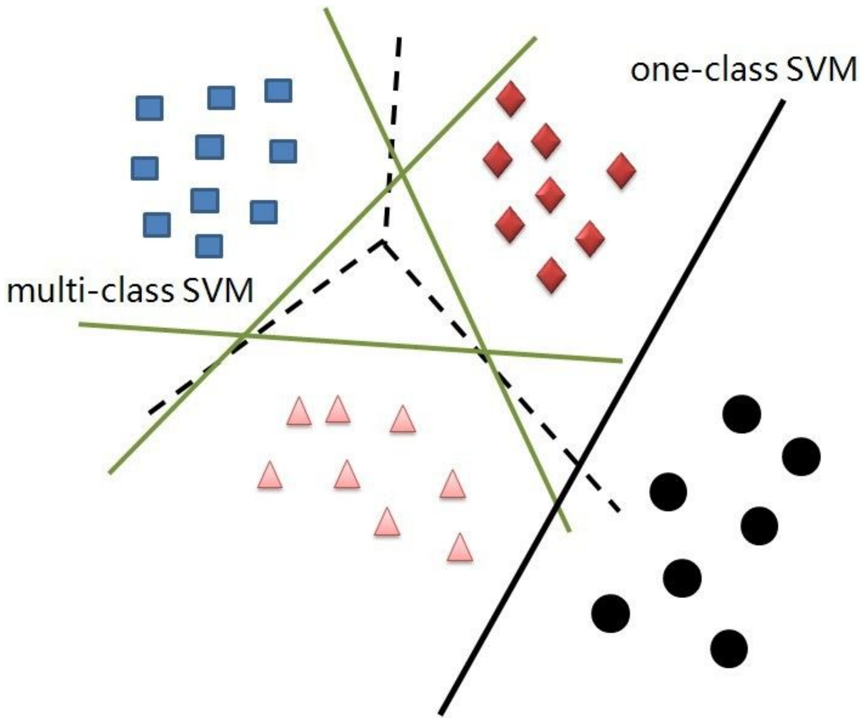


Figure 3.2: Multi-class support vector machines

Chen and Odobez (2002) compared support vector machines and neural networks for text texture verification. They used 2400 candidate text regions. The performance of verification listed in Table 3.1. is measured error rate of sample vectors. SVM shows better performance than multiple layer perceptron (MLP).

Table 3.1: Error rate of SVM and MLP for text verification.

Training Tools	DIS	DERI	CV	DCT
MLP	7.70%	6.00%	7.61%	5.77%
SVM	2.56%	3.99%	1.07%	2.92%

CHAPTER 4
PATHFINDING FOR MOBILE ROBOTS

Pathfinding is a fundamental problem for mobile robots. Path finding usually describes the process of finding the shortest path between two points using a computer application. No mobile robot could work on almost any task without moving to another point in the their environment. There are various ways for determining the shortest path between points in space, but the majority of them uses graph searching methods. The field is based heavily on Dijkstra's algorithm for finding the shortest path on a weighted graph. A path finding method searches a graph by starting at one point and exploring adjacent nodes until the destination node is reached, generally with the intent of finding the shortest route.

4.1 A* Search Algorithm

In computer science, A* is a computer algorithm which is commonly used for pathfinding, it is the process of calculating an efficiently path between two nodes. It is commonly used because of its performance and accuracy. However, in practice when used for travel-routing systems, it is outperformed by algorithms which can preprocess the graph to gain better runtime performance, although other works has found A* to be superior to other approaches.

Nils Nilsson who is an AI researcher was trying to improve the pathfinding used in the robot called Shakey. This robot that can navigate a room filled with obstacles. A1 which is faster version of the best known method, Dijkstra's algorithm, used for finding the shortest paths in graphs. After that Bertram Raphael improved this algorithm and gave a name A2. Then Peter E. Hart introduced A2, with only small changes, to be best algorithm for finding shortest paths. Haart, Nilsson and Raphael then jointly developed a proof that the revised A2 algorithm was optimal for finding shortest paths under certain conditions.

4.1.1 Description

A* solves problems by searching all possible paths to the goal for the one that incurs the smallest cost depending on the cost function, and among these paths it will first consider the ones that leads to least costly solution. It is formulated with weighted graphs, starting from a specific point, it builds a tree of paths starting from that point, expanding the paths one step at a time, until one of the paths reaches at the goal point.

A* needs to determine which of the partial paths to expand in order to reach the goal node at each iteration. It does this using an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes

$$f(n)=g(n)+h(n) \tag{4.1}$$

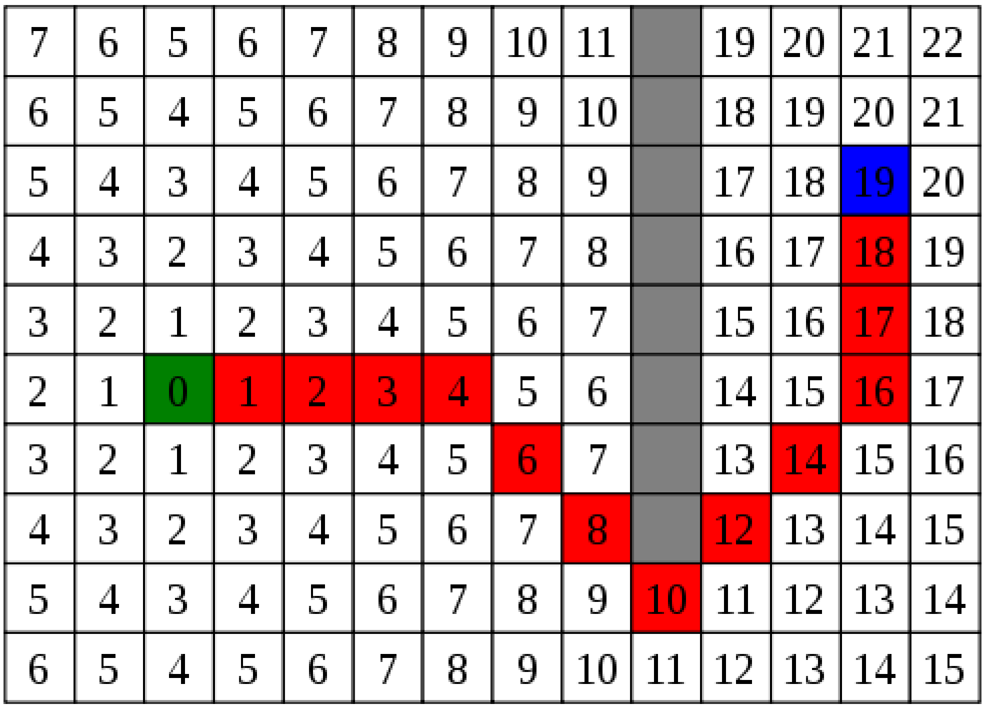
where *n* is the last node on the path, *g(n)* is the cost of the path from the start node to *n*, and *h(n)* is a heuristic function that estimates the cost of the cheapest path from n to goal. There are multiple heuristic functions to choose from depending on the problem. For the algorithm to find the actual shortest path, the heuristic function must be admissible, which means that function can never overestimates the actual cost to get to the nearest goal node.

As an example, when searching for the shortest route on a map, *h(x)* might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points.

A typical implementation of A* uses a priority queue to perform the repeated selection of minimum cost nodes to expand. This priority queue is called the open set. At each iteration of the algorithm, the node with the lowest *f(x)* value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

If the heuristic h satisfies the additional condition *h(x) <= d(x, y) + h(y)* for every edge *(x, y)* of the graph (where d denotes the length of that edge), then h is called monotone, or consistent. In such a case, A* can be implemented more efficiently—roughly speaking, no node needs to be processed more than once (see closed set below) and A* is equivalent to running Dijkstra's algorithm with the reduced cost *d(x, y) = d(x, y) + h(y) - h(x)*.

Additionally, if the heuristic is monotonic (or consistent, see below), a closed set of nodes already traversed may be used to make the search more efficient.



7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Figure 4.1: A* algorithm

4.1.2 Properties

Like breadth-first search, A* is complete and will always find a solution if one exists. If the heuristic function h is admissible, meaning that it never overestimates the actual minimal cost of reaching the goal, then A* is itself admissible (or optimal) if we do not use a closed set. If a closed set is used, then h must also be monotonic (or consistent) for A* to be optimal. This means that for any pair of adjacent nodes *x* and *y*, where *d(x,y)* denotes the length of the edge between them, we must have:

$$h(x) <= d(x,y) + h(y) \tag{4.2}$$

This ensures that for any path X from the initial node to *x*:

$$L(X) + h(x) <= L(X) + d(x,y) + h(y) = L(Y) + h(y) \tag{4.3}$$

where L is a function that denotes the length of a path, and Y is the path X extended to include y. In other words, it is impossible to decrease (total distance so far + estimated remaining distance) by extending a path to include a neighboring node. (This is analogous to the restriction to nonnegative edge weights in Dijkstra's algorithm.) Monotonicity implies admissibility when the heuristic estimate at any goal node itself is zero, since (letting P = (f,v1,v2,...,vn,g) be a shortest path from any node f to the nearest goal g):

$$h(f) <= d(f,v\_1) + h(v\_1) =< d(f,v\_1) + d(v\_1,v\_2) + h(v\_2) <= .... <= L(P) + h(g) = L(P) \tag{4.4}$$

A* is also optimally efficient for any heuristic h, which means that no optimal algorithm employing the same heuristic will expand fewer nodes than A*, except when there are multiple partial solutions where h exactly predicts the cost of the optimal path. Even in this case, for each graph there exists some order of breaking ties in the priority queue such that A* examines the fewest possible nodes.

One of the most common solutions is to implement the A* search algorithm. This algorithm has been described in 1968 and has been used in many different ways. A* is optimally efficient for a certain heuristic. In practice however, pathfinding using A* algorithm might have problems with memory and time for certain applications shortest path is not always desired, because finding the shortest optimal path can take sometime to calculate other methods such as RRT tries to work around this by calculating a path fast which is not guaranteed to be optimal.

Figure 4.2 depicts the graphical simulation result of robot navigation. Table 4.1 demonstrate the simulation results using A*, APF and RRT obstacle avoidance algorithms. Here the results are obtained for 1000 runs and environment was fixed. RRT algorithm runs faster than the others but length is long. APF algorithm finds the shortest path but running time is not acceptable. As shown the time and distance results of the A* obstacle avoidance algorithm is better for finding the path and running time. (Abiyev and others, 2015)

Table 4.1: Results

Methods	Time	Length
A*	22.534779	792.2
APF	102.47793	732.0
RRT	8.2619800	849.9

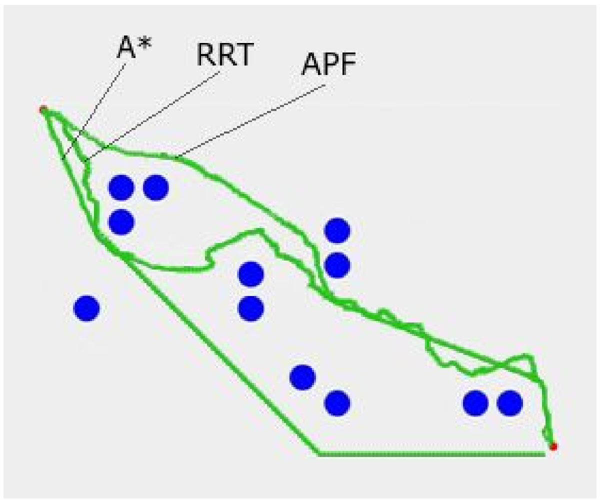


Figure 4.2: A*, RRT, APF

CHAPTER 5
DESIGN OF THE SYSTEM

This chapter covers the design of the system. It includes detailed information about hardware and software. The basic software and hardware parts of NAO robot are explained. The stages of the designed system are described. The realisation of each stage is represented.

5.1 Hardware

There are a lot of humanoid robots like Darwin, MiniHUBO, Bioloid etc.. can be used in this project. But NAO robot's educational version is cheaper and its performance is better than the others. It has also useful API's for Python and C++ programming languages. And a lot of examples can be found on internet. Because of these reasons NAO was used in this project.

NAO is an autonomous, programmable humanoid robot developed by Aldebaran Robotics (Figure 5.1). Several versions of the robot have been released since 2008. The NAO Academics Edition which was used for the research, was developed for universities and laboratories for research and education purposes. It was released to institutions in 2008, and was made publicly available by 2011. NAO robots have been used for research and education purposes in numerous academic institutions worldwide.

The various versions of the NAO robotics platform feature either 14, 21 or 25 degrees of freedom (DoF). All NAO Academics versions feature an inertial measurement unit with accelerometer, gyrometer and four ultrasonic sensors that provide NAO with stability and positioning within space. The legged versions included eight force-sensing resistors and two bumpers.

The NAO robot is controlled by a specialized Linux-based operating system, dubbed NAOqi. The OS powers the robot's multimedia system, which includes four microphones (for voice recognition and sound localization), two speakers (for multilingual text-to-speech synthesis) and two HD cameras (for computer vision, including facial and shape recognition). The robot also comes with a software suite that includes a graphical programming tool dubbed Choregraphe, a simulation software package and a software developer's kit.

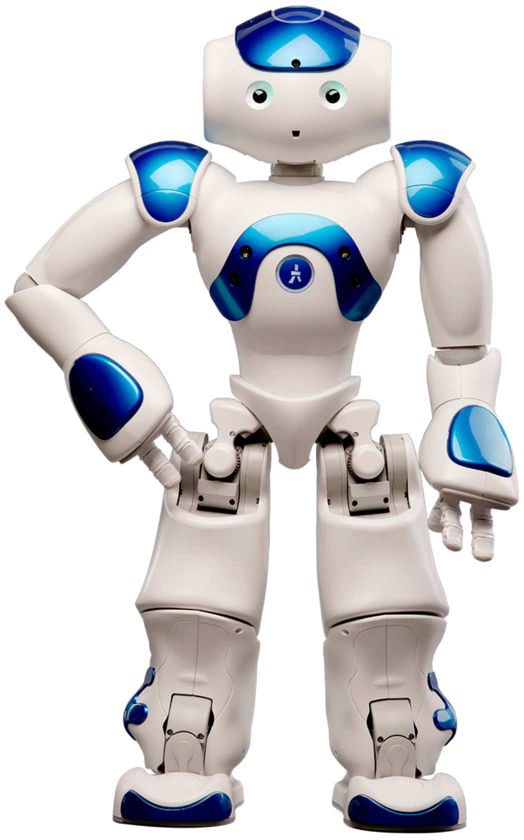


Figure 5.1: NAO robot

The structure of NAO robot is given in Figure 5.2. The robot has the following parts, tacticle sensors, cameras, sonars, joints etc. NAO has two identical cameras which are located in the forehead. Those cameras work up to 1280x960 resolution at 30 fps and can be used to identify objects in the visual field such as obstacles and goals.

The NAO robotics platform feature either 14, 21 or 25 degrees of freedom. It has internal measurement unit with accelerometer, gyrometer and four ultrasonic sensors which are for stability and positioning. It also has eight force-sensing resistors and two bumpers.

NAO robot has 25 motors. All movement is controlled by these motors. Three different kind of motors are used. Type 1 motors are used in the legs, type 2 motors in the hands and type three motors used in the head and arms. All motors are controlled by using PID. Torque and Velocity are recorded for all the movements. There are many joints on NAO robot. Some of them moved individually, or be mirrored.

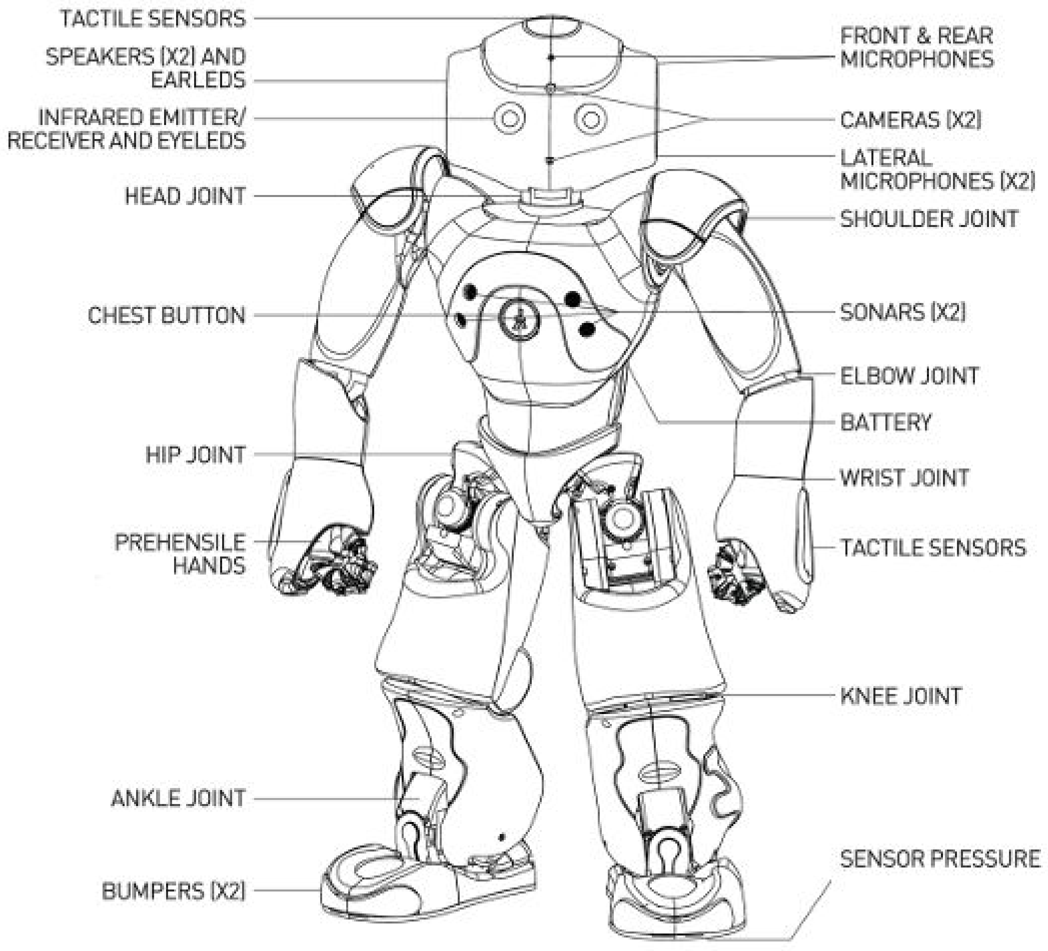


Figure 5.2: NAO robot specification

Table 5.1: NAO robot parameters

Height	58 centimetres (23 in)
Weight	4.3 kilograms (9.5 lb)
Power supply	lithium battery providing 48.6 Wh
Autonomy	90 minutes (active use)
Degrees of freedom	25
CPU	Intel Atom @ 1.6 GHz
Built-in	OSNAOqi 2.0 (Linux-based)
Programming languages	C++, Python, Java, MATLAB, Urbi, C, .Net
Sensors	Two HD cameras, four microphones, sonar rangefinder two infrared emitters and receivers, inertial board nine tactile sensors, eight pressure sensors
Connectivity	Ethernet, Wi-Fi
Compatible OS	Windows, Mac OS, Linux

5.2 Software

Interacting with the robot hardware is handled by the NAOqi framework. NAOqi is the main software which runs on the robot and controls it. The NAOqi Framework is the programming framework used to program NAO. It answers to common robotics needs including: parallelism, resources, synchronization, events. This framework allows homogeneous communication between different modules, homogeneous programming and homogeneous information sharing.

NAOqi framework:

- is cross-platform. It can be developed in Windows, Linux or Mac operating systems.
- is cross-language. It has a API for both C++ and Python.
- also provides introspection, which means the NAOqi framework knows which functions are available in the different modules and where.

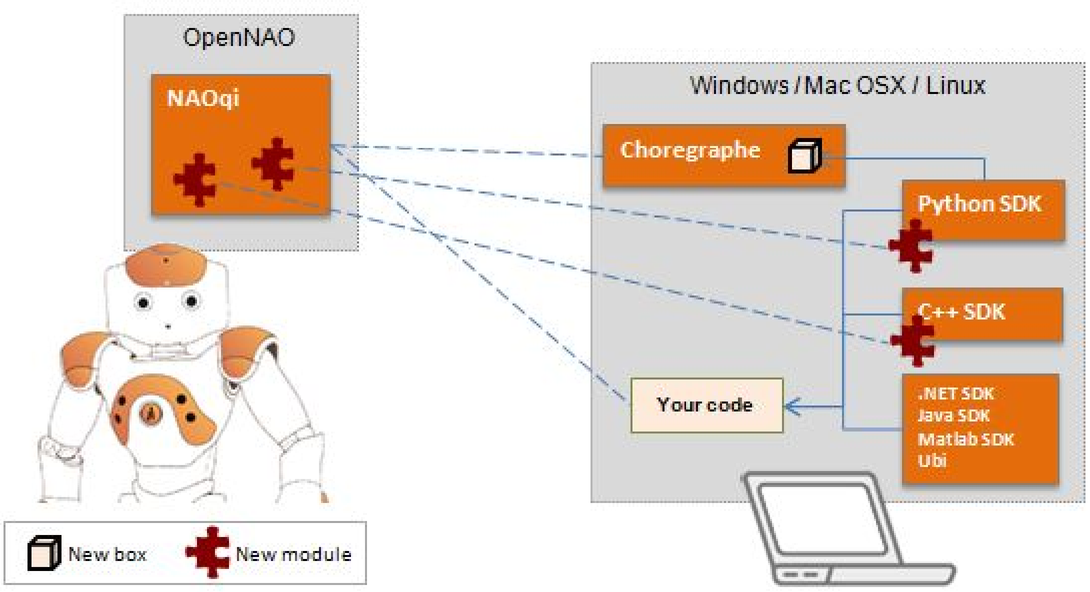


Figure 5.3: NAO Robot Cross-Platform System

It is possible to develop in C++ and Python. In two cases, programming techniques are exactly the same, all existing API can be indifferently called from any supported languages. In this thesis Python language is used to implement the algorithms.

5.2.1 The NAOqi process

The NAOqi framework which works on the NAO robot is a broker. It loads a preferences file called autoload.ini that chooses which libraries it should load at the beginning. Every library contains one or more modules that use the broker to advertise their methods.

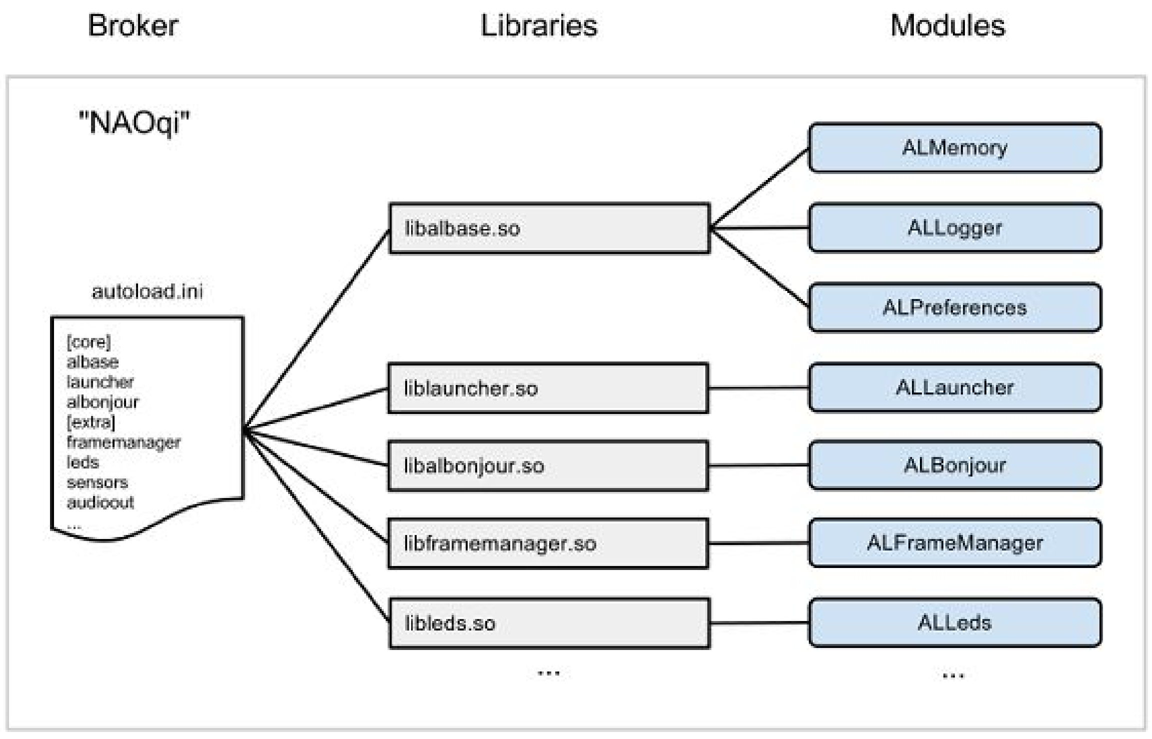


Figure 5.4 : NAOqi libraries and modules

The broker afford lookup services so that every module in the tree or across the network can find any method that has been advertised.

Loading modules forms a tree of methods connected to modules, and modules connected to a broker.

Broker: A broker is an object which has two main roles:

- It provides directory services: Letting you to find modules and methods.
- It provides network access: Allowing the methods of attached modules to be called from outside the process.

Brokers works on backwards transparently, let you to develop that will be the same for calls to “local modules” (in the same process) or “remote modules”.

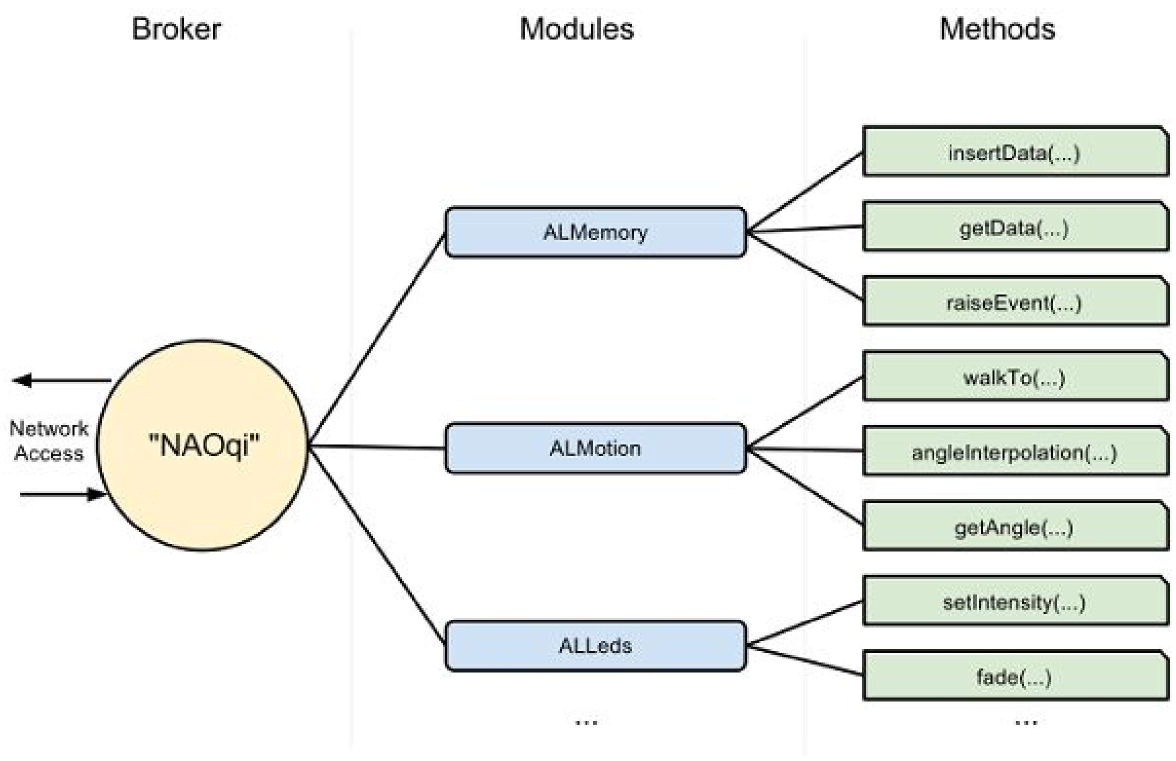


Figure 5.5: NAOqi libraries, modules and methods

Proxy: A proxy is an object that will act as the module it represents.

For instance, if you create a proxy to the ALMotion module, you will get an object including all the ALMotion methods.

When creating proxy to a module, you have two options:

- Easily use the name of the module. In this case, the running code and the connected module must be in the same broker. This is a local call.
- Use the name of the module, and the IP and port of a broker. In this situation, the module must be in the corresponding broker.

Modules: Every Module is a class within a library. When the library is called from the autoload.ini, it will automatically instantiate the module class.

In the constructor of a class that derives from ALModule, you can “bind” methods. This advertises their names and method signatures to the broker so that they become available to others.

A module can be either remote or local.

- If the module is remote, the module is compiled as an executable file, and can be execute outside the NAO robot. Remote modules are simply to use and can be debugged quickly from the outside, but the remote modules are less efficient for speed and memory usage.
- If the module is local, the module is compiled as a library, and can only be worked on the robot. But, they are more efficient than a remote module.

Memory: ALMemory is the NAO robot’s memory. Every modules can read or write data, subscribe on events so as to be called when events are raised.

ALMemory is an array of ALValue’s. Accessing the variable is thread safe. Read and write critical sections can be used to avoid bad performance when memory is read.

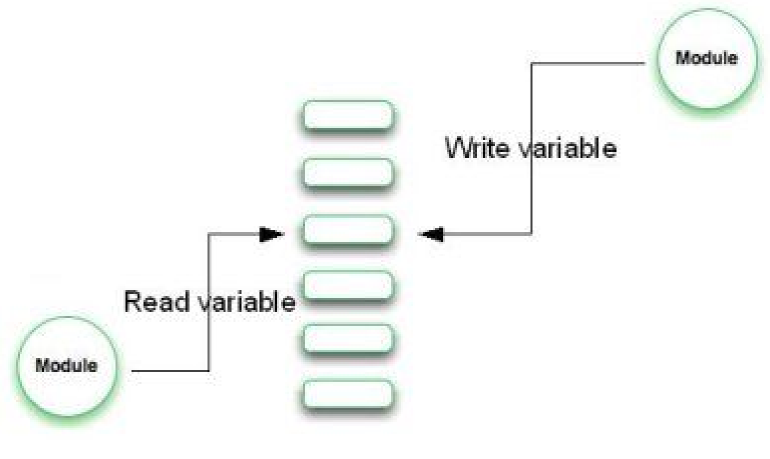


Figure 5.6: NAOqi memory system

ALMemory contains three types of data and provides three different APIs.

- Mainly data from sensors and joints
- Event
- Micro-event

Python: Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.

Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems. Using third-party tools, such as Py2exe or Pyinstaller, Python code can be packaged into standalone executable programs for some of the most popular operating systems, so Python-based software can be distributed to, and used on, those environments with no need to install a Python interpreter.

CPython, the reference implementation of Python, is free and open-source software and has a community-based development model, as do nearly all of its variant implementations. CPython is managed by the non-profit Python Software Foundation.

OpenCV: For processing camera images from the NAO robot OpenCV computer vision library is used. OpenCV (Open Source Computer Vision) is a library mainly aimed at real-time computer vision, originally developed by Intel's research center in Russia, later supported by Willow Garage and now maintained by Itseez. The library is cross-platform and free for use under the open-source BSD license.

OpenCV's application areas include:

- 2D and 3D feature toolkits
- Egomotion estimation
- Facial recognition system
- Gesture recognition
- Human–computer interaction (HCI)
- Mobile robotics
- Motion understanding
- Object identification
- Segmentation and recognition
- Stereopsis stereo vision: depth perception from 2 cameras
- Structure from motion (SFM)
- Motion tracking
- Augmented reality

To support the above areas, OpenCV includes a statistical machine learning library that contains:

- Boosting
- Decision tree learning
- Gradient boosting trees
- Expectation-maximization algorithm
- k-nearest neighbor algorithm
- Naive Bayes classifier
- Artificial neural networks
- Random forest
- Support vector machine (SVM)

Scikit-learn: Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

The scikit-learn project started as scikits.learn, a Google Summer of Code project by David Cournapeau. Its name stems from the notion that it is a “SciKit” (SciPy Toolkit), a separately-developed and distributed third-party extension to SciPy. The original codebase was later rewritten by other developers. Of the various scikits, scikit-learn as well as scikit-image were described as “well-maintained and popular” in November 2012.

As of 2015, scikit-learn is under active development and is sponsored by INRIA, Telecom ParisTech and occasionally Google (through the Google Summer of Code).

Scikit-learn is largely written in Python, with some core algorithms written in Cython to achieve performance. Support vector machines are implemented by a Cython wrapper around LIBSVM; logistic regression and linear support vector machines by a similar wrapper around LIBLINEAR.

NumPy (pronounced “Numb Pie” or sometimes “Numb pee”) is an open source extension to the Python programming language, adding support for large, multidimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

5.3 System Design

The design of system includes the following steps, shown in Figure 5.7

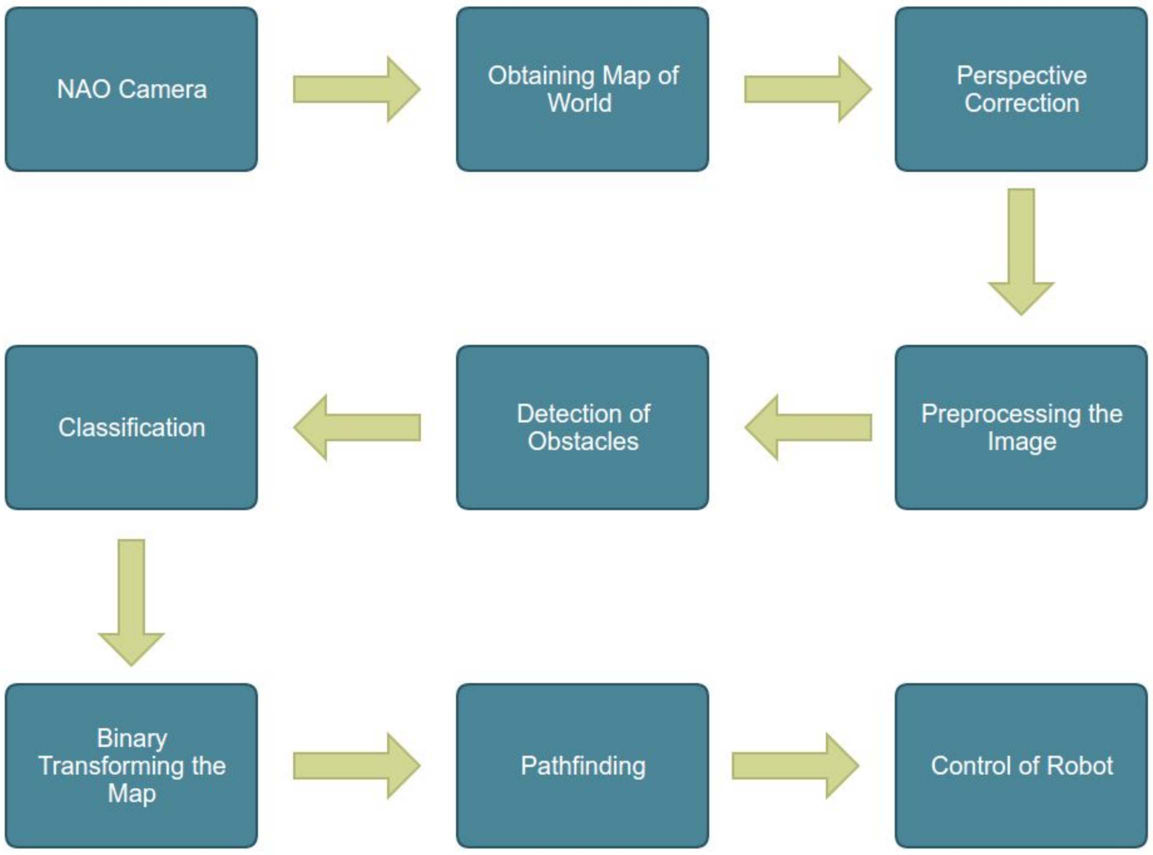


Figure 5.7: Steps of the system

Steps of the system

1) Image Capturing
    a) Image is captured by the camera which is located on NAO robot's forehead.

2) Obtaining Map of World
    a) Image is read by using opencv and converted from rgb to hsv value to find the fixed white area.
    b) Lower and upper white hsv values defined.
    c) Threshold the HSV image to get only white colors by using opencv's cv2.inRange() function.
    d) cv2.morphologyEx() function's closing technique is used for the remove small holes.
    e) Images is blurred to fix erosion by using cv2.GaussianBlur function.

f) To getting contours cv2.threshold() and cv2.findContours() function is applied.
g) To get coordinates of a rectangle around the contour "tuple(cnt[cnt[:,:,0].argmin()][0])" technique is used for four coordinates.
h) Lines and dots are drawn to show the area by using cv2.line() and cv2.circle() functions.
i) Four coordinates are put in the array to be used for perspective correction.

3) Perspective Correction
    a) The width of the new image is computed which will be the maximum distance between bottom-right and bottom-left x-coordinates or the top-right and top-left x-coordinates by using this technique.
    ```
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))
    ```
    b) The height of the new image is computed, which will be the maximum distance between the top-right and bottom-right y-coordinates or the top-left and bottom-left y-coordinates by using this technique.
    ```
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))
    ```
    c) The dimension of the new image is get, the set of destination points to obtain a "birds eye view" of the image constructed, top-left, top-right, bottom-right, and bottom-left points are specified by using this technique.
    ```
    dst = np.array([
            [0, 0],
            [maxWidth - 1, 0],
            [maxWidth - 1, maxHeight - 1],
            [0, maxHeight - 1]], dtype = "float32")
    ```
    d) Perspective transform matrix computed and applied by using cv2.getPerspectiveTransform() and cv2.warpPerspective() functions.

   e) Perspective corrected image is saved.

4) Preprocessing the Image
   a) Canny Edge Detection technique is used for finding the edges.
   b) Corner Harris Detection technique is used for detecting the corners.
   c) Contour detection is applied by using opencv's cv2.findContours() function and rectangles are drawn around obstacles for safety region.

5) Detection of Obstacles
   a) Preprocessed image is split into 20x20 pixels 768 pieces.
   b) All images are separated "negative" and "positive" in two classes.
   c) Selected images are put into folders for training SVM and dataset is created.

6) Classification
   a) Pictures are selected one by one to get histograms values for training SVM.
   b) Depend on histograms values all images are identified 1 or 0.
   c) SVM is trained.

7) Binary Transforming the Map
   a) Depend on this values binary matrix are created.

8) Pathfinding
   a) Using binary matrix, A* algorithm is applied to find shortest path.
   b) From this coordinates waypoints are found.

9) Control of Robot
   a) Imaginary path is processed and turned it into real path values.
   b) Movement commands which are NAOqi framework's commands for moving the NAO robot, are sent from computer to NAO robot by using robots ip and port.

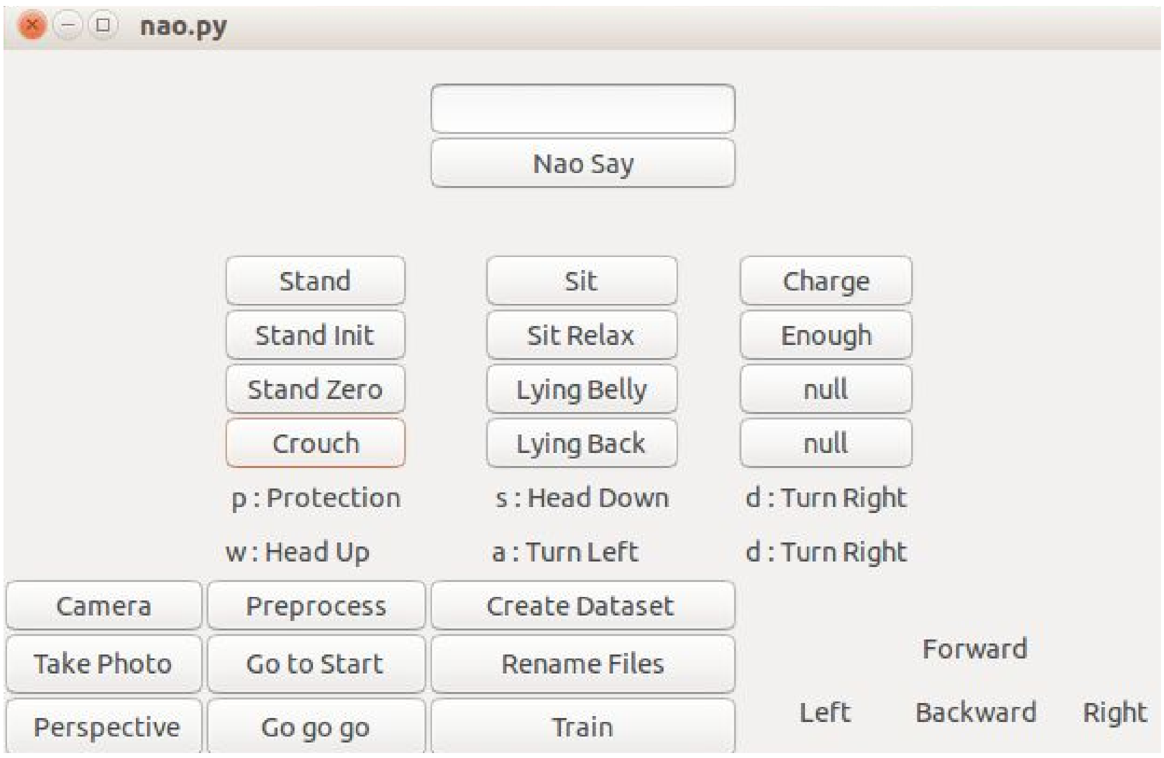


Figure 5.8: Graphical user interface

A simple graphical user interface (Figure 5.8) was written to control NAO robot. It includes some movement functions and process steps. You can move NAO robot through this gui application.

NAO stands a place where can see the fixed area. The obstacles which are different colors and shapes located on the fixed white area. An image is captured using the on board camera on the NAO Robot's head. Firstly software connects to predefined NAO's ip and port. NAO has 2 cameras and they provide a up to 1280x960 resolution at 30 frames per second. But in this project 640x480 resolution was used. Because if the resolution increases preprocess step also increases. After that photo will be taken in array format and then convert it into real image by using opencv. Next problem is to find fixed area's corners for performing perspective corrections. If the image's width is not enough for to see all fixed white area, NAO can go back until the area fit the image.

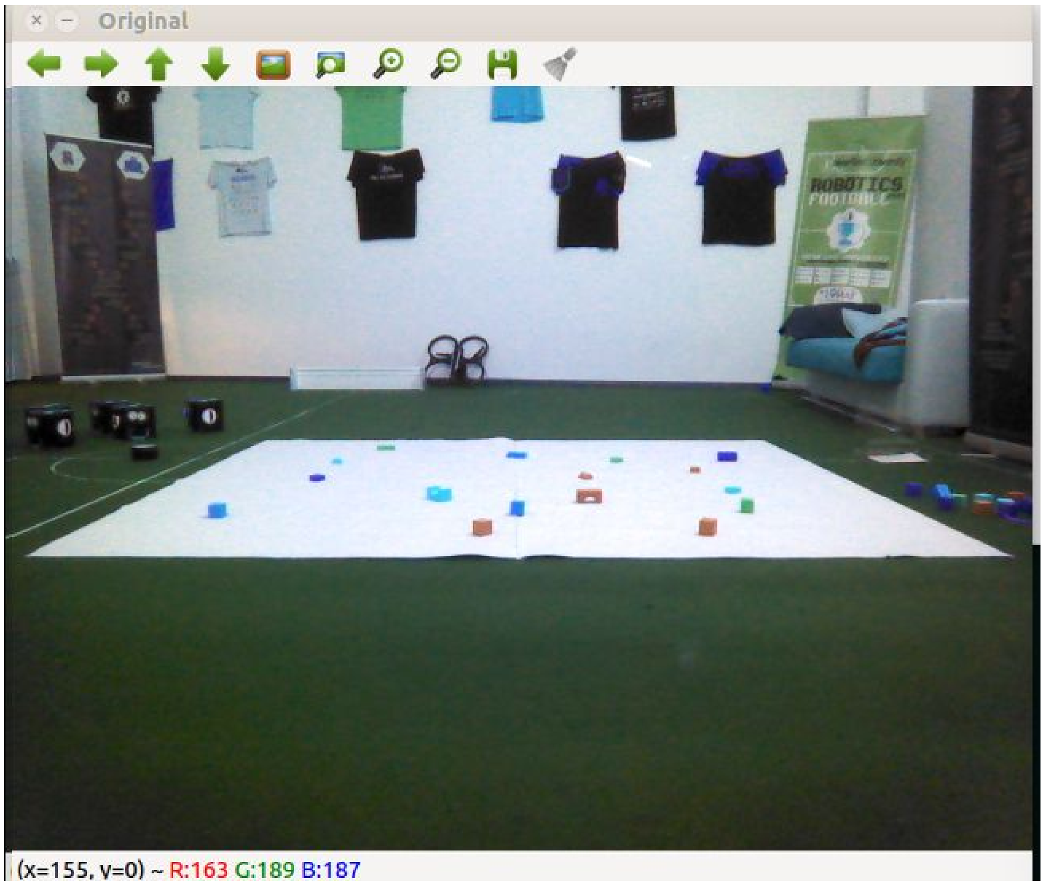


Figure 5.9: Static area with obstacles

First we need to read this image for converting from RGB (Red, Green, Blue) to HSV (Hue Saturation Value). Then we define white colors lower and upper boundaries because of the fixed area's color. If the fixed area's color is different; this color's boundaries must be used. In other case, If the area's color is not homogeneous, texture recognition can be applied. After finding the white fixed area with erosion, some filtering applied to the image such as closing. Closing fills small holes inside the foreground objects, or small black points on the object. Then we find the contour of fixed white area and get the coordinates of a rectangle's corners. We compute the width of the new image, which will be the maximum distance between bottom-right and bottom-left x-coordinates or the top-right and top-left x-coordinates. And then we compute the height of the new image, which will be the maximum distance between the top-right and bottom-right y-coordinates or the top-left and bottom-left y-coordinates. Now that we have the dimensions of the new image, construct the set of destination points to obtain a "birds eye view" of the image, again specify points in the top-left, top-right, bottom-right, and bottom-left order. Finally save the image.

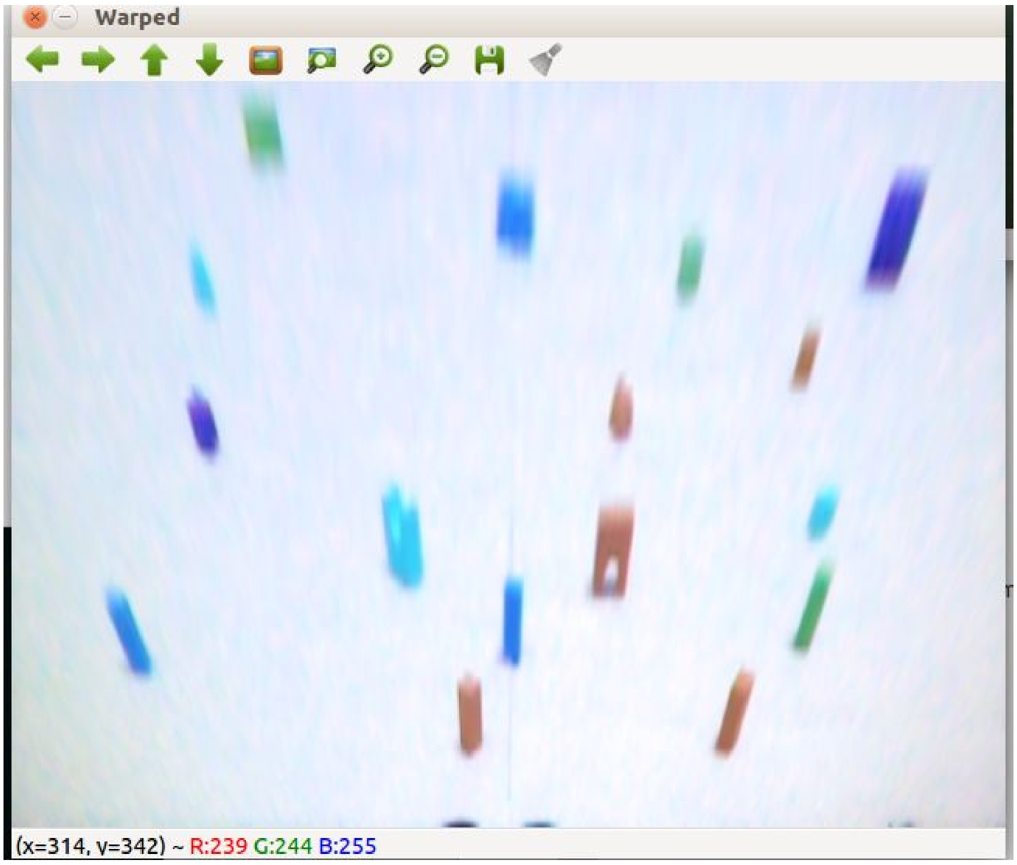


Figure 5.10: Static area after perspective correction

Now our fixed white area (Figure 5.10) is ready to apply processing steps. First we apply canny edge detection to detect a wide range of edges in images. In this step contrast is a factor that affect the performance of segmentation. You need to provide a natural contrast for the segmented image to get a more accurate segmentation process. If contrast between obstacles and floor is not much, segmentation process will be hard. After we apply corner detection and contours to rectangles around and inside obstacles for to describe safe area. Because A* search algorithm gives us to shortest path and this path is too close to obstacles but NAO robot needs place to avoid obstacles.

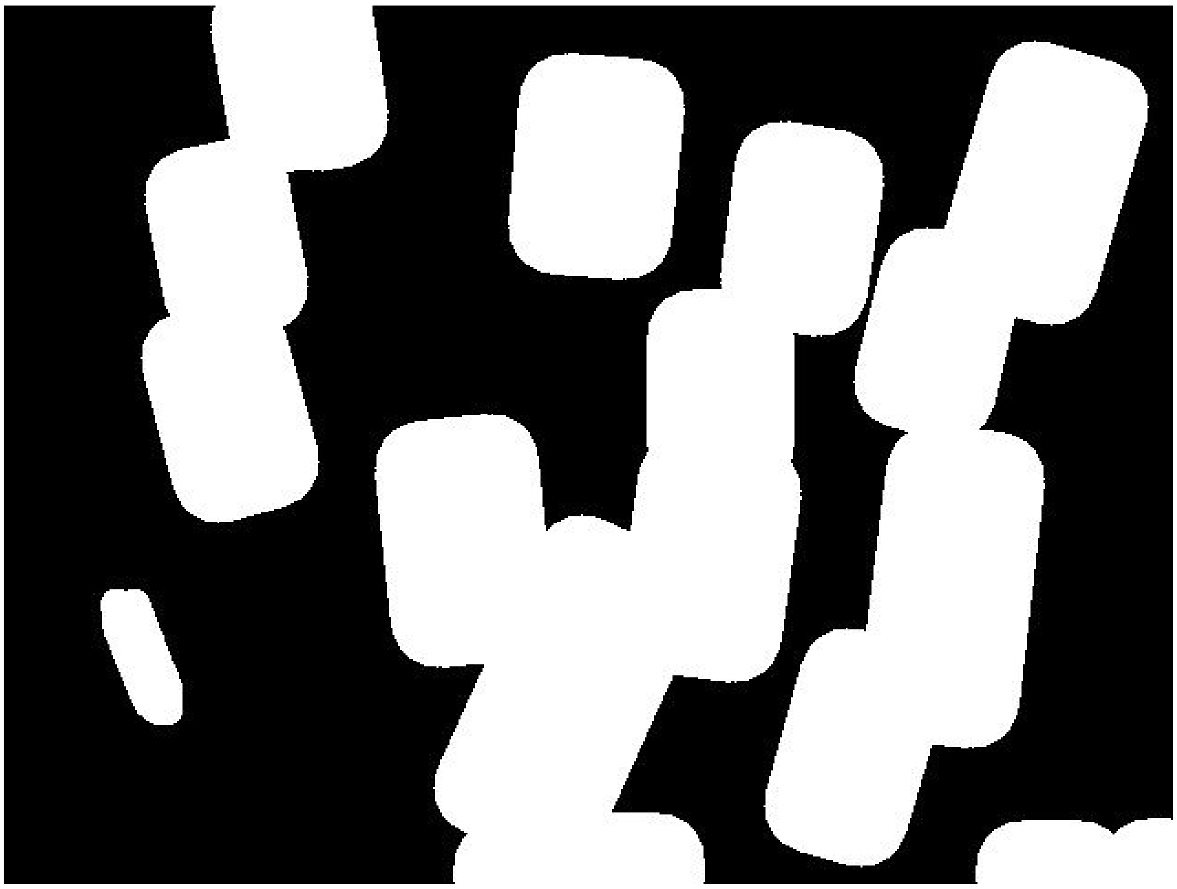


Figure 5.11: Processed area

In this step, we crop our fixed area which is 640x480 pixels, into 20x20 pixels 768 pieces (Figure 5.12) for converting image into binary matrix. After that we allocate this little images, positive and negative folders. Mostly white pictures were put in negative folder and mostly black pictures were put in positive folder. Positive represents the area without obstacles. Negative represents the area with obstacles. Mostly black images are area and mostly white images are obstacles. Finally a dataset created to train the SVM.

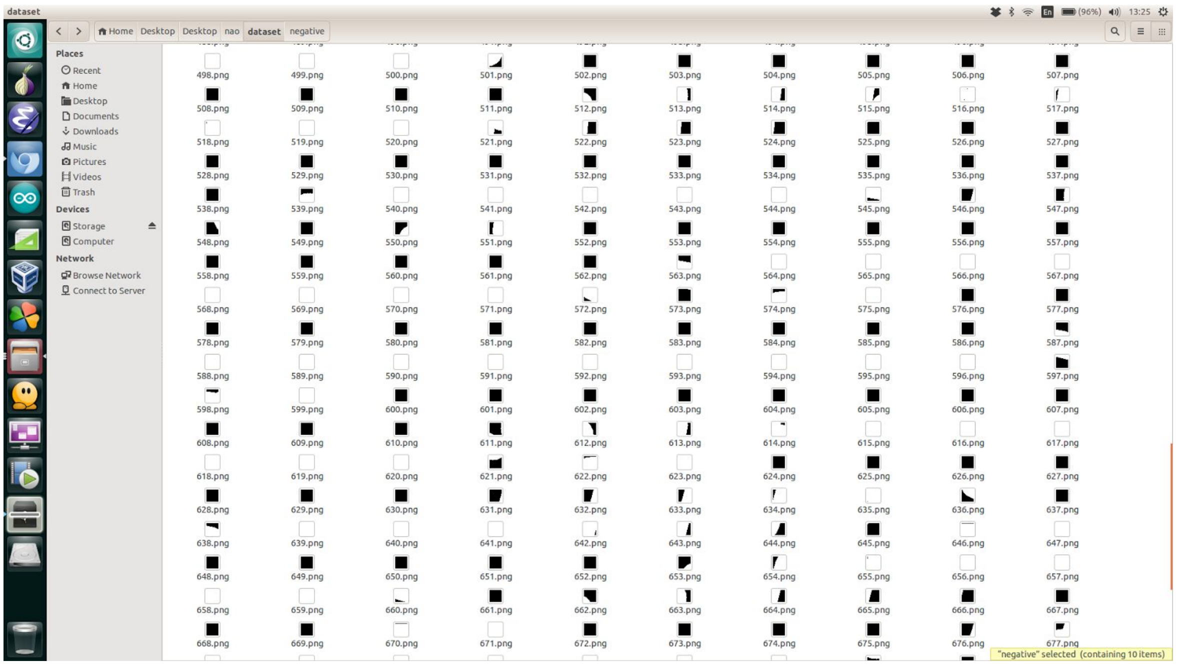
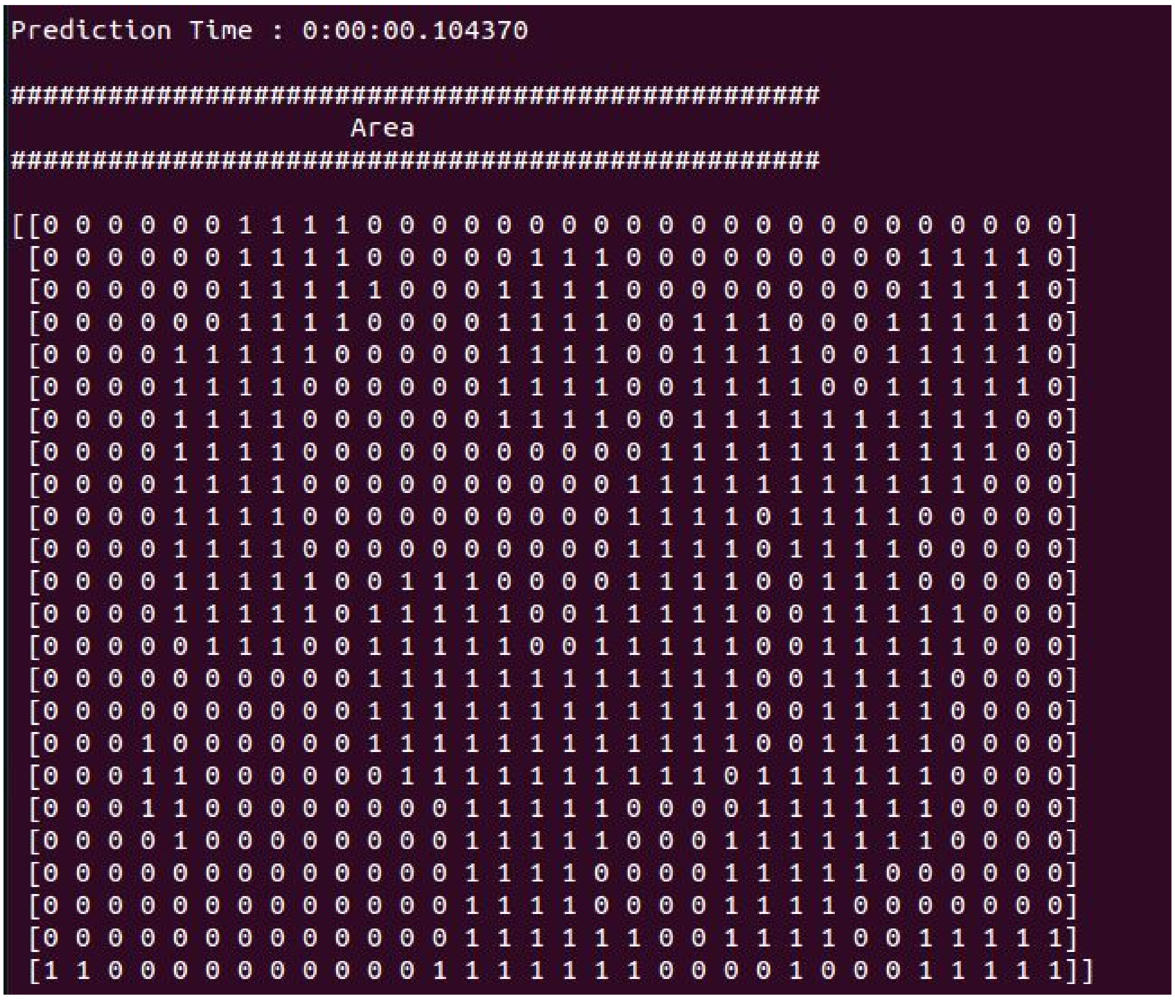

dataset
Home
Desktop
Desktop
nao
dataset
negative
Places
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Devices
Storage
Computer
Network
Browse Network
Connect to Server
498.png 499.png 500.png 501.png 502.png 503.png 504.png 505.png 506.png 507.png
508.png 509.png 510.png 511.png 512.png 513.png 514.png 515.png 516.png 517.png
518.png 519.png 520.png 521.png 522.png 523.png 524.png 525.png 526.png 527.png
528.png 529.png 530.png 531.png 532.png 533.png 534.png 535.png 536.png 537.png
538.png 539.png 540.png 541.png 542.png 543.png 544.png 545.png 546.png 547.png
548.png 549.png 550.png 551.png 552.png 553.png 554.png 555.png 556.png 557.png
558.png 559.png 560.png 561.png 562.png 563.png 564.png 565.png 566.png 567.png
568.png 569.png 570.png 571.png 572.png 573.png 574.png 575.png 576.png 577.png
578.png 579.png 580.png 581.png 582.png 583.png 584.png 585.png 586.png 587.png
588.png 589.png 590.png 591.png 592.png 593.png 594.png 595.png 596.png 597.png
598.png 599.png 600.png 601.png 602.png 603.png 604.png 605.png 606.png 607.png
608.png 609.png 610.png 611.png 612.png 613.png 614.png 615.png 616.png 617.png
618.png 619.png 620.png 621.png 622.png 623.png 624.png 625.png 626.png 627.png
628.png 629.png 630.png 631.png 632.png 633.png 634.png 635.png 636.png 637.png
638.png 639.png 640.png 641.png 642.png 643.png 644.png 645.png 646.png 647.png
648.png 649.png 650.png 651.png 652.png 653.png 654.png 655.png 656.png 657.png
658.png 659.png 660.png 661.png 662.png 663.png 664.png 665.png 666.png 667.png
668.png 669.png 670.png 671.png 672.png 673.png 674.png 675.png 676.png 677.png
"negative" selected (containing 10 items)
13:25

Figure 5.12: Dataset

An SVM is trained to detect obstacles in the image. Image histograms are used for training the SVM. System knows which image is obstacle or not. After that we created a binary matrix (Figure 5.13) which represents the fixed area. In this matrix, 1 is area with obstacle and 0 is area without obstacle.



```
Prediction Time : 0:00:00.104370

##################################################
                     Area
##################################################

[[0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0]
 [0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0]
 [0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0]
 [0 0 0 0 0 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1]
 [1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 1 1 1 1 1]]
```

Figure 5.13: Area in binary matrix format

Pathfinding is applied to find a path to guide the robot to its destination (Figure 5.14). But the problem is NAO doesn't know where the path begins. Because of this reason a marker should show the starting point. For solving this problem i put NAO's red ball on start point. Because NAOqi framework has a special function for finding red ball and return the coordinates of red ball. By using these coordinates NAO can reach the starting point. After that NAO can follow the path and reach the destination.

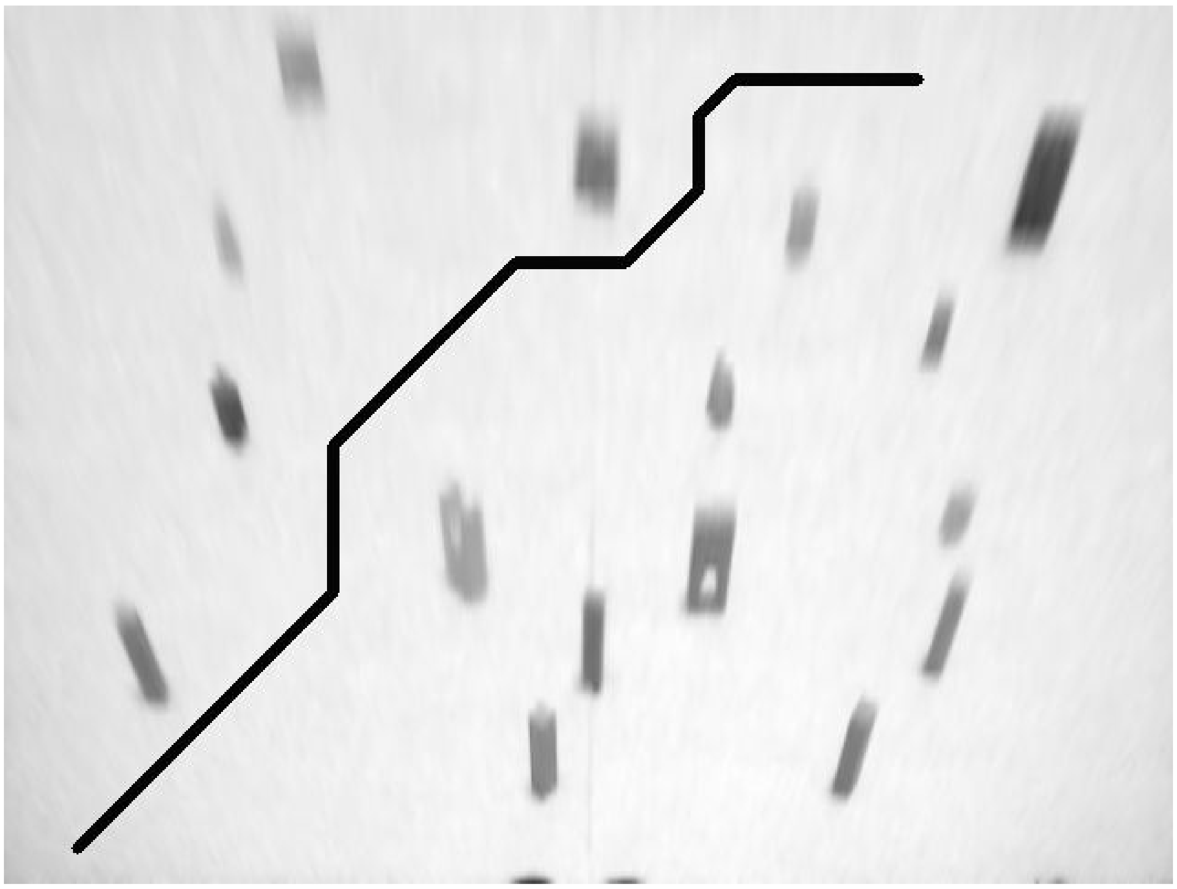


Figure 5.14: Area with path

Image capturing is handled by the on board camera and the software. Captured image is then transferred to the host computer for processing. Perspective correction is done using the following method. Image is read, by default OpenCV uses the RGB color model which is an additive color model in which red, green and blue light are added together in various ways to reproduce a broad array of colors. The name of the model comes from the initials of the three additive primary colors, red, green and blue, in order to make the image processing easier RGB model is then converted to HSV model which is the two most common cylindrical-coordinate representations of points in an RGB color model. This representations rearrange the geometry of RGB in an attempt to be more intuitive and perceptually relevant than the cartesian (cube) representation. Then the image is thresholded using the colors of the obstacles and a closing rectangle of the surface is calculated, the image is blurred to get rid of imperfections and the contours of the surface is calculated. Perspective correction is applied to account for the location and height of the robot. The resulting image is saved for further processing (Figure 5.15).

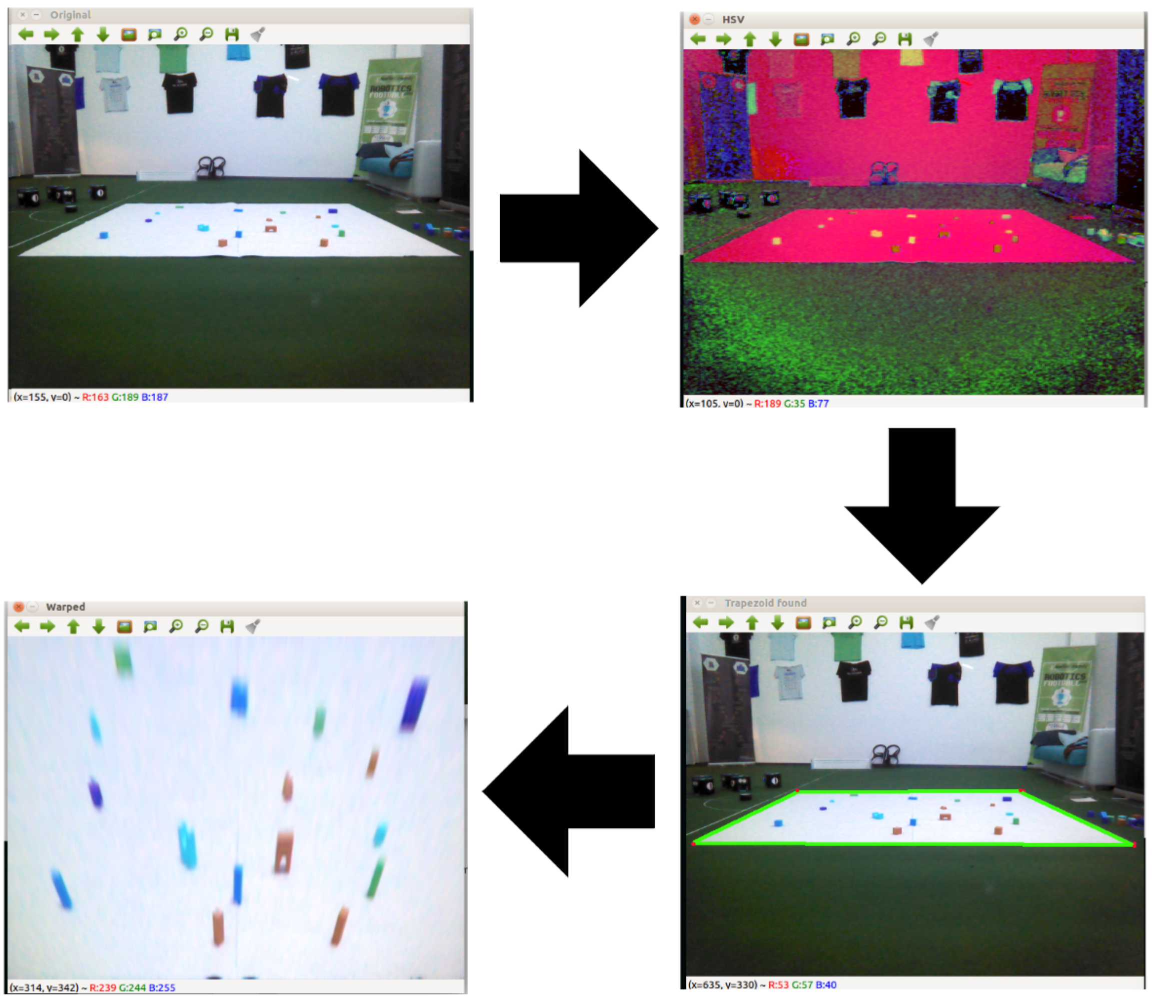

Original
(x=155, y=0) ~ R:163 G:189 B:187
HSV
(x=105, y=0) ~ R:189 G:35 B:77
Trapezoid found
(x=635, y=330) ~ R:53 G:57 B:40
Warped
(x=314, y=342) ~ R:239 G:244 B:255

Figure 5.15: Perspective correction steps

Surface area extracted from the above process is further processed to actually find the obstacle on the surface. It is done by applying canny edge detection. Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986. Corner harris detection is applied which is an approach used within computer vision systems to extract certain kinds of features and infer the contents of an image in this case corners of the obstacles are detected (Figure 5.16).

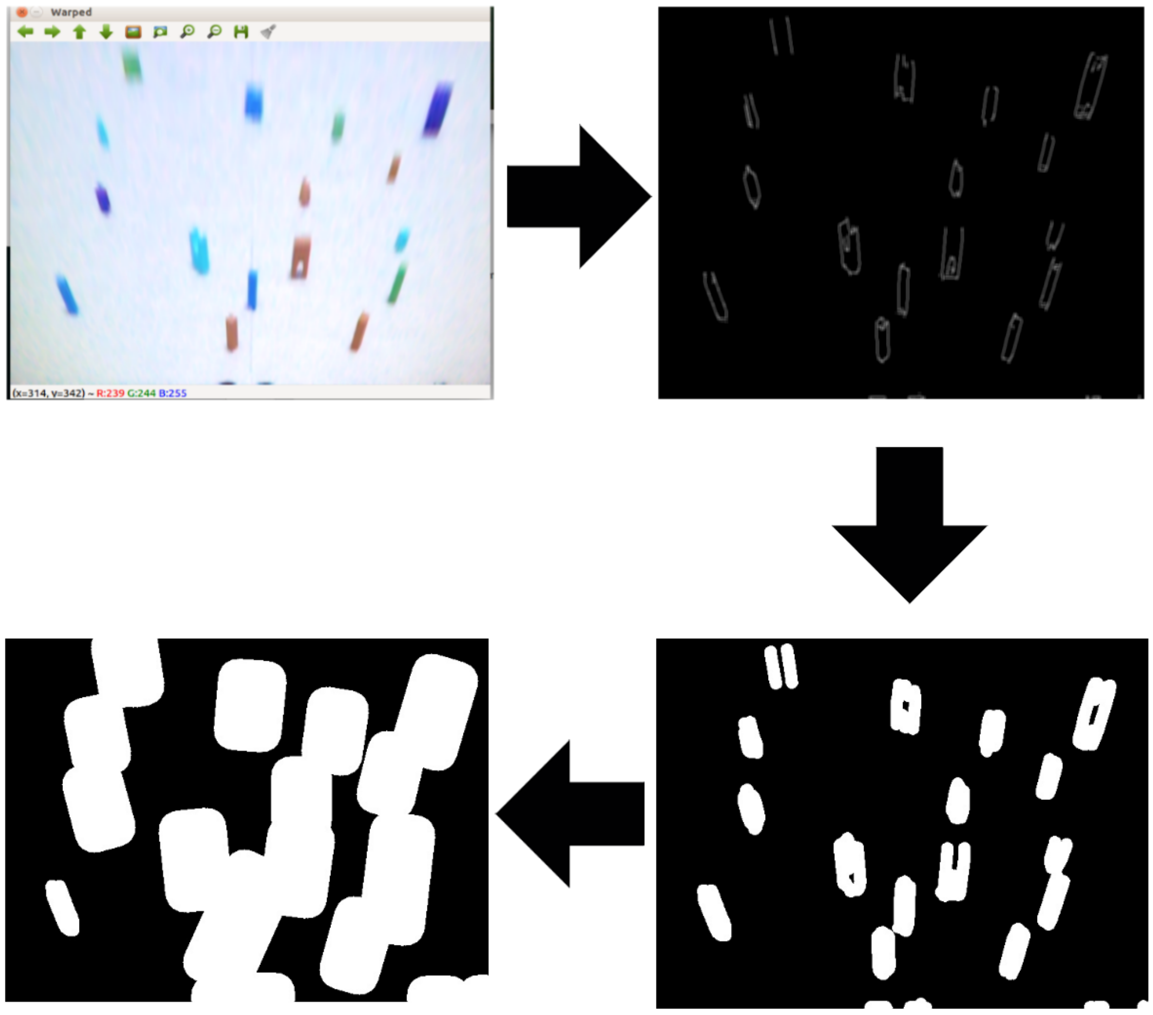
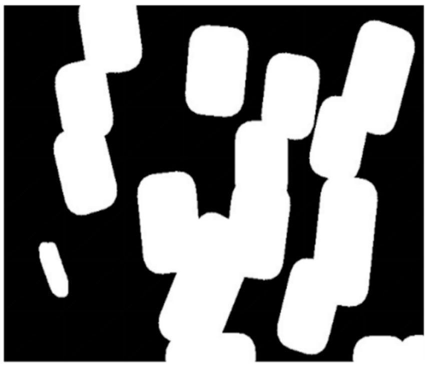
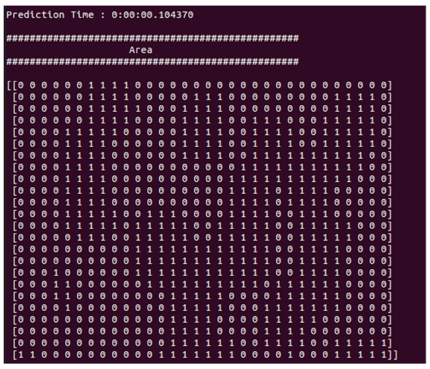


Figure 5.16: Preprocessing steps

A dataset is created to be used by the SVM, by taking the image and splitting the image into smaller images that are 20x20 pixels. Positive set of images contains the obstacles on the surface and the negative set of images that contains the images that the robot can move.

SVM is trained that detects smaller images with obstacles and without obstacles. Histograms for the images are used for features for the SVM. Given a set of training examples, each marked for belonging to one of two categories, SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier.

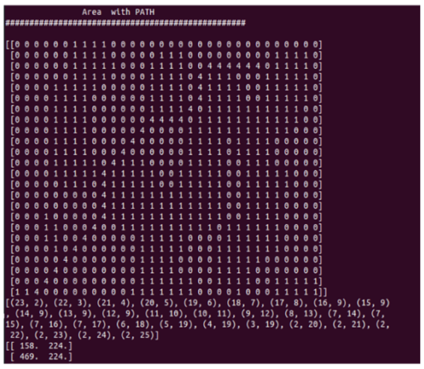




```
Prediction Time : 0:00:00.104370

##################################################
                    Area
##################################################

[[0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0]
 [0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0]
 [0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0]
 [0 0 0 0 0 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1]
 [1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 1 1 1 1 1]]
```



```
               Area  with PATH
##################################################

[[0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0]
 [0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 0 4 4 4 4 4 4 0 1 1 1 1 0]
 [0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 4 1 1 1 0 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 4 1 1 1 1 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 4 1 1 1 1 0 0 1 1 1 1 1 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 4 0 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 4 4 4 4 0 1 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 4 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 0 4 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 0 0 0 4 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 1 0 4 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 1 4 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0]
 [0 0 0 0 0 1 1 1 0 4 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 4 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 4 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 4 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 4 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 4 0 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 1 0 4 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 4 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0]
 [0 0 0 0 4 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0]
 [0 0 0 4 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1]
 [1 1 4 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 1 1 1 1 1]]
[(23, 2), (22, 3), (21, 4), (20, 5), (19, 6), (18, 7), (17, 8), (16, 9), (15, 9)
, (14, 9), (13, 9), (12, 9), (11, 10), (10, 11), (9, 12), (8, 13), (7, 14), (7,
15), (7, 16), (7, 17), (6, 18), (5, 19), (4, 19), (3, 19), (2, 20), (2, 21), (2,
 22), (2, 23), (2, 24), (2, 25)]
[[ 158.  224.]
 [ 469.  224.]
```

Figure 5.17: Converting image to matrix

Once the above steps are done. Mobile robot has the ability to detect new obstacles on the surface. At this point path-finding can be applied to the surface using the previously trained SVM. Designed system works by taking picture of the surface detecting obstacles using the trained SVM which returns a binary matrix of the surface then applying A* path-finding algorithm on the surface. Then the robot moves to the start position which is denoted by a red ball. Finally robot follows the path calculated (Figure 5.18).

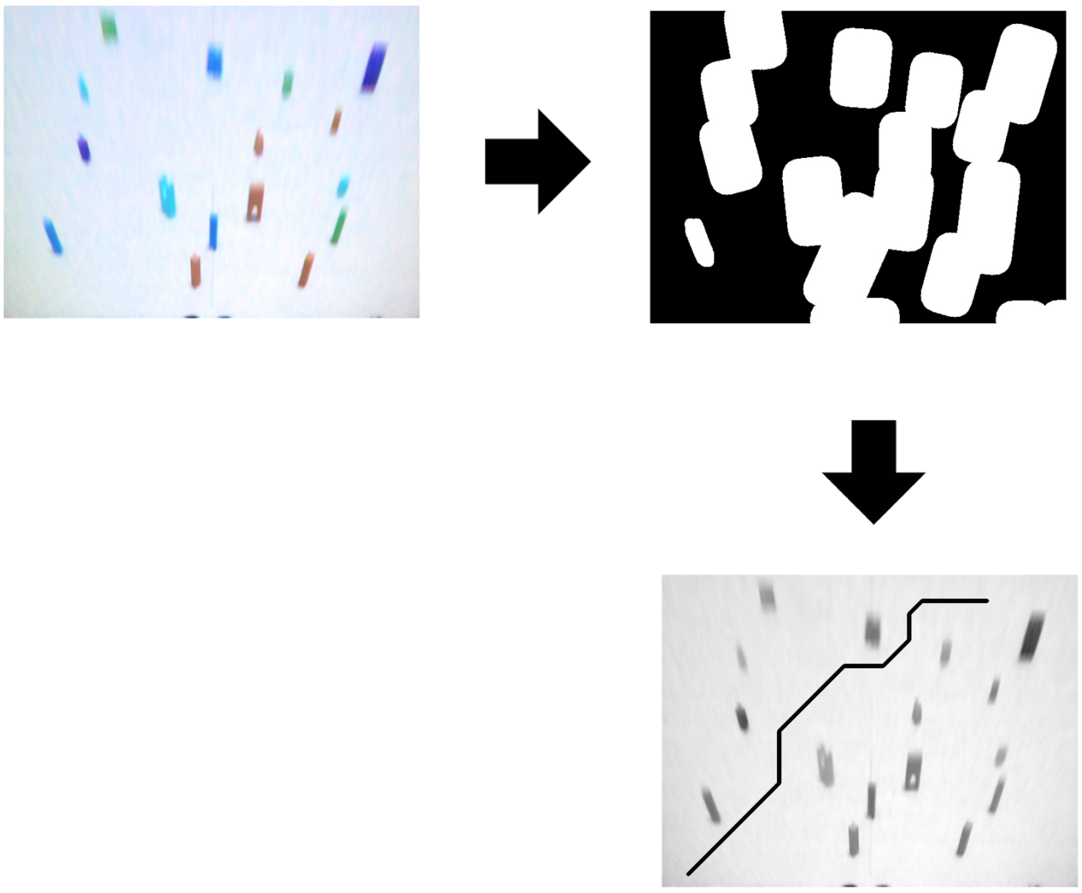


Figure 5.18: Main steps of pathfinding

CHAPTER 6
CONCLUSION

In this thesis, robot navigation system based on Support Vector Machine and path finding algorithm is designed and used in real time application by using NAO robot. Code is available for testing and some videos proof its good behavior.

Analysis of mobile robots have shown that one of important blocks in robot navigation are object detection and path finding. Review of object detection algorithms have been done. SVM classification algorithm and image processing techniques are used for detection of objects. Perspective correction techniques are also used for image processing.Image capturing and processing are done using NAO robot

The structure of designed robot navigation systems is presented. The functions of their main steps are described.

For the classification of the images MLP and SVM algorithms were compared and SVM is chosen for classification purpose. A support vector machine which is a supervised learning algorithm is used to analyze data and create model of real world. Results of SVM classification is used in path finding.

One of the difficulty was finding fixed area. The fixed area in first image which was captured from NAO's camera was perspective because of the angle. To solve this problem perspective correction was applied. The other problem was light system. If the light changes, fixed white areas white color range's changes too. To solve this problem, special software which finds range of colors, was used.

After classification of world map A* algorithm is applied for path finding problem. The theoretical background of A* algorithm has been given. Some path planning algorithms were compared and A* algorithms is chosen for path selection.

The image processing techniques, SVM algorithm and A* path-finding algorithm are used in navigation of NAO robot.

Designed system relies on only vision data to navigate. This allows the mobile robot to navigate in environments where other traditional sensors (sonars, magnetometers etc.) won't work or does not work reliably. But they can be used for improvements for suitable environments. New techniques, like Deep Learning, can be tried to improve this system. Designed systems applications include industrial automation, mobile robots in hazardous environments.

Currently designed system works on static obstacles, for future work the designed system will be modified to work with dynamic moving obstacles.

REFERENCES

Abiyev, A. H., Akkaya, N., Aytac, E., Günsel, I. & Çagman, A. (2015, January). *Brain Based Control of Wheelchair. In Proceedings of the International Conference on Artificial Intelligence* (ICAI) (p. 542) Las Vegas: USA.

Abiyev, R. H., Akkaya, N., Aytac, E., Arici, M. & Bayezit, M. *NEUIslanders Team Description Paper 2012*. Robocup SSL, MexicoCity, Mexico.

Abiyev, R., Ibrahim, D. & Erin, B. (2010). *Navigation of mobile robots in the presence of obstacles. Advances in engineering software*, 41(10), 1179-1186.

Abiyev, R., Ibrahim, D. & Erin, B. (2010). EDURobot: An Educational Computer Simulation Program for Navigation of Mobile Robots in the Presence of Obstacles. *International Journal of Engineering Education*. Vol.26. No.1, pp.18-29, 2010.

Abiyev, R. H., Akkaya, N., Aytac, E. & Ibrahim, D. (2014). Behavior Tree Based Control For Efficient Navigation of Holonomic Robots. *International Journal of Robotics a and Automation*, *29*(1), 32-38.

Abiyev, R. H., Günsel, I., Akkaya, N., Aytac, E., Çağman, A. & Abizada, S. (2016). Robot Soccer Control Using Behaviour Trees and Fuzzy Logic. *In Proceedings of the International Conference on Procedia Computer Science (*pp. 477-484). Jakarta: Indonesia.

Abiyev, R. H., Akkaya, N., Aytac, E., Günsel, I. & Çağman, A. (2015). Improved Path-Finding Algorithm for Robot Soccers. *Journal of Automation and Control Engineering Vol*, *3*(5), 77-83.

Abiyev, R. H., Akkaya, N. & Aytac, E. (2013, June). Control of soccer robots using behaviour trees. *In Proceeding of the Control Conference on ASCC, 9th Asian (pp. 1-6).* Istanbul: Turkey.

Abiyev, R. H., Akkaya, N. & Aytac, E. (2012, May). Navigation of mobile robot in dynamic environments. In Proceedings of the International Conference on Computer Science and Automation Engineering (pp. 480-484). Zhangjiajie: China.

Abiyev, R. H., Bektas, S., Akkaya, N. & Aytac, E. (2013). *Behavior* Tree Based Control of Holonomic Robots. *In International Journal of Robotics and Automation, 24*(8), 120-124.

Abiyev, R. H., Akkaya, N., Aytac, E., Günsel, I. & Çağman, A. (2015) Improved Path-Finding Algorithm for Robot Soccers. *Journal of Automation and Control Engineering, 3*(5), 38-42.

ARSLAN, M., ARICI, M., CAGMAN, A., HUSEYIN, S., EMREM, F., YILMAZ, B. & AYTAC, E. (2011). *NEUIslanders 2016 Team Description Paper*.

Ben-Hur, A., Horn, D., Siegelmann, H. T. & Vapnik, V. (2001). Support vector clustering. *Journal of machine learning research*, *2*(12), 125-137.

Borenstein, J. & Koren, Y. (1991). The vector field histogram-fast obstacle avoidance for mobile robots. *Journal of Robotics and Automation*, *7*(3), 278-288.

Boser, B. E., Guyon, I. M. & Vapnik, V. N. (1992, July). A training algorithm for optimal margin classifiers. *In Proceedings of the fifth annual workshop on Computational learning theory* (pp. 144-152). Pittsburgh: USA.

Cortes, C. & Vapnik, V. (1995). Support-vector networks. Machine learning. *In the Proceedings of the International Conference in Artificial Neural Networks*, (pp. 273-297). Bochum: Germany.

Cuingnet, R., Rosso, C., Chupin, M., Lehéricy, S., Dormont, D., Benali, H. & Colliot, O. (2011). Spatial regularization of SVM for the detection of diffusion alterations associated with stroke outcome. *Journal of Medical Image Analysis*, *15(*5), 729-737.

Davidor, Y. (1991). Genetic Algorithms and Robotics: A heuristic strategy for optimization. *Journal of World Scientific, 16(5), 654-660.*

Delling, D., Sanders, P., Schultes, D. & Wagner, D. (2009). Engineering route planning algorithms. *In Proceedings of the International of Algorithmics of large and complex networks,* (pp. 117-139). Berlin: Germany.

Dijkstra, E. W. (1959). Numerische Mathematik. Amsterdam: Holland.

Erin, B., Abiyev, R. & Ibrahim, D. (2010). Teaching robot navigation in the presence of obstacles using a computer simulation program. *International Journal of Procedia-Social and Behavioral Sciences, 2*(2)*,* 565-571.

Fakoor, M., Kosari, A. & Jafarzadeh, M. (2015). Revision on fuzzy artificial potential field for humanoid robot path planning in unknown environment. *International Journal of Advanced Mechatronic Systems, 6*(4), 174-183.

Fan, J., Wu, G., Ma, F. & Liu, J. (2004, August). Reinforcement Learning and ART2 Neural Network Based Collision Avoidance System of Mobile Robot. *In International Journal on Neural Networks, 5*(6*),* 150-155.

Ge, S. S., & Cui, Y. J. (2002). Dynamic motion planning for mobile robots using potential field method. *International Journal of Autonomous Robots,* 13(3), 207-222.

Hand, D. J., & Yu, K. (2001). Idiot's Bayes—not so stupid after all. *International Journal of statistical review, 69*(3), 385-398.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *International Journal of Systems Science and Cybernetics, 4*(2), 100-107.

Hart, P. E., Nilsson, N. J. & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *International Journal of Systems Science and Cybernetics*, 4(2), 100-107.

Janabi-Sharifi, F. & Vinke, D. (1993, August). Integration of the artificial potential field approach with simulated annealing for robot path planning. *International Journal of Intelligent Control Systems*, (pp. 536-541).

Koenig, S. & Likhachev, M. (2005). Fast replanning for navigation in unknown terrain. *International Journal of Navigation on Robotics*, 21(3), 354-363.

Koenig, S., Likhachev, M. & Furcy, D. (2004). Lifelong planning A*. *International Journal of Artificial Intelligence, 155*(1), 93-14

LaValle, S. M. & Kuffner, J. J. (2001). Randomized kinodynamic planning. *The International Journal of Robotics Research, 20*(5), 378-400.

Lee, M. C. & Park, M. G. (2003, July). Artificial potential field based path planning for mobile robots using a virtual obstacle concept. *In Proceedings of the International Conference on the Advanced Intelligent Mechatronics*. (pp. 735-740). Kobe: Japan

Lozano-Pérez, T. & Wesley, M. A. (1979). An algorithm for planning collision-free paths among polyhedral obstacles. *International Journal of Communications of the ACM, 22*(10), 560-570.

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *International Journal of the bulletin of mathematical biophysics*, *5*(4), 115-   133.

Mehlhorn, K. & Sanders, P. (2008). Algorithms and data structures: The basic toolbox. *International Journal of Science & Business Media, 10(*15), 216-223.

Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (2013). Machine learning: An artificial intelligence approach. *International Journal of Science & Business Media, 8*(12), 356-363.

Park, M. G., Jeon, J. H. & Lee, M. C. (2001). Obstacle avoidance for mobile robots using artificial potential field approach with simulated annealing. *International Journal of Industrial Electronics, 9*(13), 1530-1535.

Rahib, H., GUNSEL, A. P. D. I., AKKAYA, N., ARSLAN, M., ARICI, M., CAGMAN, A. & AYTAC, E. *NEUIslanders 2015 Team Description Paper*.

Rahib, H., GUNSEL, A. P. D. I., AKKAYA, N., ARSLAN, M., ARICI, M., CAGMAN, A. & AYTAC, E. *NEUIslanders 2014 Team Description Paper*.

Ramalingam, G. & Reps, T. (1996). An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, *21(*2), 267-305.

Rish, I. (2001, August). An empirical study of the naive Bayes classifier. *International Journal of empirical methods in artificial intelligence, 22*(15), 41-46.

Stentz, A. (1995, August). The focussed D* algorithm for real-time replanning. *International Journal in artificial intelligence, 5*(11) 1652-1659).

Ulrich, I. & Borenstein, J. (2000, April). VFH*: Local obstacle avoidance with look-ahead verification. *In Proceedings of the International Conference on Robotics and Automation* (pp. 2505-2511). San Francisco: USA

Ulrich, I. & Nourbakhsh, I. (2000, July). Appearance-based obstacle detection with monocular color vision. *In Proceedings of the International Conference Innovative Applications of Artificial Intelligence.* (pp. 866-871). Austin: USA

Ying, N. A. N., Eicher, L., Xunzhang, W., Seet, G. G., & Lau, M. W. (2000). Real-time 3D path planning for sensor-based underwater robotics vehicles in unknown environment. *In Proceedings of the International Conference in OCEANS MTS,* (pp. 2051-2058). California: USA.

Zeng, W. & Church, R. L. (2009). Finding shortest paths on real road networks: the case for A*. *International journal of geographical information science*, *23*(4), 531-543.

Russell, I. (2012). *Neural Networks Module.*

APPENDIX
SOURCE CODE

ball.py

```
import motion
import math
import time

from NAOqi import ALBroker
from NAOqi import ALProxy
import config as c

class NAO():

  def __init__(self):
    self.myBroker = ALBroker("myBroker","0.0.0.0",0,c.IP,c.PORT)
    self.motion   = ALProxy("ALMotion")
    self.tracker  = ALProxy("ALRedBallTracker")
    self.vision   = ALProxy("ALVideoDevice")
    self.tts      = ALProxy("ALTextToSpeech")
    self.currentCamera = 0
    self.setTopCamera()
    self.tracker.setWholeBodyOn(False)
    self.tracker.startTracker()
    self.ballPosition = []
    self.targetPosition = []


  def __del__(self):
    self.tracker.stopTracker()
    pass


# If NAO has ball returns True
def hasBall(self):
    self.checkForBall()
    time.sleep(0.5)
    if self.checkForBall():
        return True
    else:
        return False


# NAO scans around for the redball
def searchBall(self):
    self.motion.stiffnessInterpolation("Body", 1.0, 0.1)
    self.motion.walkInit()
    for turnAngle in [0,math.pi/1.8,math.pi/1.8,math.pi/1.8]:
        if turnAngle > 0:
            self.motion.walkTo(0,0,turnAngle)
        if self.hasBall():
            self.turnToBall()
            return
        for headPitchAngle in [((math.pi*29)/180),((math.pi*12)/180)]:
            self.motion.angleInterpolation("HeadPitch",
            headPitchAngle,0.3,True)
            for headYawAngle in [-((math.pi*40)/180),-((math.pi*20)/180),
            0,((math.pi*20)/180),((math.pi*40)/180)]:
                self.motion.angleInterpolation("HeadYaw",
                headYawAngle,0.3,True)
                time.sleep(0.3)
        if self.hasBall():
            self.turnToBall()
            return


# NAO walks close to ball
def walkToBall(self):
    ballPosition = self.tracker.getPosition()
    headYawTreshold = ((math.pi*10)/180)
    x = ballPosition[0]/2 + 0.05
    self.motion.stiffnessInterpolation("Body", 1.0, 0.1)
    self.motion.walkInit()
    self.turnToBall()
    self.motion.post.walkTo(x,0,0)
    while True:
    dist = self.getDistance()
    print dist
    self.setTopCamera()
    if dist < 0.7:
    self.setBottomCamera()
    if dist == None:
    self.motion.stopWalk()
        print "Stop!"
        break
    if dist < 0.1:
    self.motion.stopWalk()
        print "Stop!"
        break
      headYawAngle = self.motion.getAngles("HeadYaw", False)
      if headYawAngle[0] >= headYawTreshold or headYawAngle[0] <=
      -headYawTreshold:
```

```
    while dist > 0.1111111:
        self.motion.stopWalk()
                self.turnToBall()
                self.walkToBall()
        break


# NAO turns to ball
def turnToBall(self):
    if not self.hasBall():
        return False
    self.ballPosition = self.tracker.getPosition()
    b = self.ballPosition[1]/self.ballPosition[0]
    z = math.atan(b)
    self.motion.stiffnessInterpolation("Body", 1.0, 0.1)
    self.motion.walkInit()
    self.motion.walkTo(0,0,z)


# checks ball
def checkForBall(self):
    newdata = self.tracker.isNewData()
    if newdata == True:
        self.tracker.getPosition()
        return newdata
    if newdata == False:
        self.tracker.getPosition()
        return newdata


# has to be called after walkToBall()
```

```
def walkToPosition(self):
    x = (self.targetPosition[0]/2)
    self.motion.walkTo(x,0,0)


# Determine safe position
def safePosition(self):
    if self.hasBall():
        self.targetPosition = self.tracker.getPosition()
    else:
        return False


# gets the distance from ball
def getDistance(self):
    if self.hasBall():
        ballPosition = self.tracker.getPosition()
        return math.sqrt(math.pow(ballPosition[0],2) +
        math.pow(ballPosition[1],2))


# setting up top camera
def setTopCamera(self):
    self.vision.setParam(18,0)
    self.currentCamera = 0


# setting up bottom camera
def setBottomCamera(self):
    self.vision.setParam(18,1)
    self.currentCamera = 1
    # protection off to move free
    def protectionOff(self):
        self.motion.setExternalCollisionProtectionEnabled( "All", False )
        print "Protection Off"



    # protection on
    def protectionOn(self):
        self.motion.setExternalCollisionProtectionEnabled( "All", True )
        print "Protection On"

NAO().protectionOff()
NAO().searchBall()
if NAO().hasBall() == True:
    NAO().walkToBall()
```

config.py

```
# NAO
IP = "192.168.10.13"
PORT = 9559
CAMERAID = 18
RESOLUTION =  2   # Image Size
COLORSPACE = 11   # Select RGB

# GUI
GLADE_FILE_PATH = "ui/NAO.glade"
DATASET_DIRECTORY = "/home/murat/Desktop/Desktop/NAO/dataset"
NEGATIVE = "/home/murat/Desktop/Desktop/NAO/dataset/negative"
```

```
POSITIVE = "/home/murat/Desktop/Desktop/NAO/dataset/positive"
```

hsv.py

```
import cv2
import numpy as np

cap = cv2.imread("image.png")

def nothing(x):
    pass
# Creating a window for later use
cv2.namedWindow('result')

# Starting with 100's to prevent error while masking
h,s,v = 100,100,100

# Creating track bar
cv2.createTrackbar('h', 'result',0,179,nothing)
cv2.createTrackbar('s', 'result',0,255,nothing)
cv2.createTrackbar('v', 'result',0,255,nothing)

while(1):

    frame = cv2.imread("image.png")

    #converting to HSV
    hsv = cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)

    # get info from track bar and appy to result
    h = cv2.getTrackbarPos('h','result')
    s = cv2.getTrackbarPos('s','result')
    v = cv2.getTrackbarPos('v','result')

    # Normal masking algorithm
    lower_blue = np.array([h,s,v])
    upper_blue = np.array([180,255,255])

    mask = cv2.inRange(hsv,lower_blue, upper_blue)

    result = cv2.bitwise_and(frame,frame,mask = mask)

    cv2.imshow('result',result)

    k = cv2.waitKey(5) & 0xFF
    if k == 27:
        break

cap.release()

cv2.destroyAllWindows()
```

movement.py

```
import sys
import os
import time
from gi.repository import Gtk, GObject
from NAOqi import ALProxy


import config as c
import photo  as p
import svm

tts        = ALProxy("ALTextToSpeech", c.IP, c.PORT)
motion     = ALProxy("ALMotion", c.IP, c.PORT)
posture    = ALProxy("ALRobotPosture", c.IP, c.PORT)
photo      = ALProxy("ALPhotoCapture", c.IP, c.PORT)
#navigation = ALProxy("ALNavigation", c.IP, c.PORT)

headposition = 0
protection   = 0

class NAOController:

        """Represents NAO Controller GUI

        params: glade_file_path - path:string
        """
    def __init__(self, glade_file_path=c.GLADE_FILE_PATH):
            self.glade_file_path = glade_file_path

            # Gtk Builder Init
            self.builder = Gtk.Builder()
            self.builder.add_from_file(self.glade_file_path)
            self.builder.connect_signals(self)

            # Add UI Components
            self.window = self.builder.get_object("NAOControllerWindow")
            self.speechbox = self.builder.get_object("speechbox")

            # Show UI
            self.window.show_all()

        ### NAO Posture Functions
    def NAOStandInit(self, widget):
        posture.goToPosture("StandInit", 1.0)
            print "Stand Init"

    def NAORelax(self, widget):
        posture.goToPosture("SitRelax", 1.0)
            print "Sit Relax"

    def NAOZero(self, widget):
        posture.goToPosture("StandZero", 1.0)
            print "Stand Zero"

    def NAOBelly(self, widget):
        posture.goToPosture("LyingBelly", 1.0)
            print "Lying Belly"

    def NAOBack(self, widget):
        posture.goToPosture("LyingBack", 1.0)
            print "Lying Back"

    def NAOStand(self, widget):
        posture.goToPosture("Stand", 1.0)
            print "Stand"

    def NAOCrouch(self, widget):
        posture.goToPosture("Crouch", 1.0)
            print "Crouch"
```

```
def NAOSit(self, widget):
   posture.goToPosture("Sit", 1.0)
      print "Sit"

### NAO Motion Functions
def NAOEnough(self, widget):
   motion.moveInit()
      motion.post.wakeUp()
   tts.say("Enough")
      print "WakeUp"

def NAOCharge(self, widget):
   motion.moveInit()
      motion.post.rest()
   tts.say("Charge me")
   print "Rest"

   ### Text to Speech
   def NAOSay(self, widget):
      tts.say(self.speechbox.get_text())
      print "Say: %s" % self.speechbox.get_text()

   ### Destroy GUI
   def destroy(self, widget):
      print "destroyed"
      Gtk.main_quit()

### SVM Preprocessing Functions
def openCamera(self, widget):
      p.openCamera()
```

```
def perspective(self, widget):
        p.perspective()

def preprocess(self, widget):
        p.process()

def ball(self, widget):
        p.ball()

    def takePhoto(self, widget):
        p.takePhoto()

    def cropImage(self, widget):
        p.cropImage()

    def nameChanger(self, widget):
        p.nameChanger()

    def drawGrid(self, widget):
        p.drawGrid()

    def giveMePath(self, widget):
        svm.path()

    def goToFinish(self, widget):
        svm.moveToPoint()

    ### Key Pressed Event to control NAO remotely
    def keyPressed(self, widget, event):
        global headposition
        global protection
```

```
key_code = event.get_keycode()[1]

if key_code == 33:  # P Protection On / Off
        if protection == 0:
                motion.setExternalCollisionProtectionEnabled
                ( "All", False )
                protection = 1
                print "Protection Off"
        else:
                motion.setExternalCollisionProtectionEnabled
                ( "All", True )
                protection = 0
                print "Protection On"

if key_code == 111:  # UP NAO Forward
        print "Moving Forward..."
        motion.moveInit()
        motion.walkTo(0.1, 0, 0)

if key_code == 116: # Down NAO Backward
        print "Moving Backward..."
        motion.moveInit()
        motion.moveTo(-0.1, 0, 0)

if key_code == 113: # Left NAO Left
        print "Moving Left..."
        motion.moveInit()
        motion.moveTo(0, 0.1, 0)

if key_code == 114: # Right NAO Right
```

```
        print "Moving Right..."
        motion.moveInit()
        motion.moveTo(0, -0.1, 0)

if key_code == 38: # a Turn Left
        print "Turning Left..."
        motion.moveInit()
        motion.moveTo(0, 0, 1)

if key_code == 40: # d Turn Right
        print "Turning Right..."
        motion.moveInit()
        motion.moveTo(0, 0, -1)

if key_code == 25: # w Head Down
        headposition = headposition + 0.1
        motion.angleInterpolation("HeadPitch",
        headposition, 0.5, True)
        if headposition > 0.5:
                headposition = 0.5
        print "Head Position: %s" % headposition

if key_code == 39: # s Head UP
        headposition = headposition - 0.1
        motion.angleInterpolation("HeadPitch",
        headposition, 0.5, True)
        if headposition < -0.5:
                headposition = -0.5
        print "Head Position: %s" % headposition

#print "keyPressed: %s" % key_code
```

```
    def keyReleased(self, widget, event):
        key_code = event.get_keycode()[1]
        #print "keyReleased: %s" % key_code
```

NAO.py

```
from gi.repository import Gtk
import movement


if __name__ == "__main__":
    controller = movement.NAOController()
  Gtk.main()
```

perspective.py

```
import cv2
import numpy as np

# Load image
im = cv2.imread('image.png')
img = cv2.imread('image.png')

# Kernel configuration
kernel = np.ones((35,35),np.uint8)



# Convert BGR to HSV
hsv = cv2.cvtColor(im, cv2.COLOR_BGR2HSV)



# define range of white color in HSV
lower_blue = np.array([0,0,168])
```

```
upper_blue = np.array([179,255,255])


# Threshold the HSV image to get only green colors
mask = cv2.inRange(hsv, lower_blue, upper_blue)


# Closing
closing = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)


# Blur
blur = cv2.GaussianBlur(closing,(5,5),10)


# Getting contours
ret, thresh = cv2.threshold(blur, 127, 255,0)
contours,hierarchy = cv2.findContours(thresh,2,1)
cnt = contours[0]


# Get coordinates of a rectangle around the contour
leftmost = tuple(cnt[cnt[:,:,0].argmin()][0])
topmost = tuple(cnt[cnt[:,:,1].argmin()][0])
bottommost = tuple(cnt[cnt[:,:,1].argmax()][0])
rightmost = tuple(cnt[cnt[:,:,0].argmax()][0])

print leftmost
print topmost
print bottommost
print rightmost
```

```
# Drawing lines and circles
hull = cv2.convexHull(cnt,returnPoints = False)
defects = cv2.convexityDefects(cnt,hull)

for i in range(defects.shape[0]):
    s,e,f,d = defects[i,0]
    start = tuple(cnt[s][0])
    end = tuple(cnt[e][0])
    far = tuple(cnt[f][0])
    cv2.line(im,start,end,[0,255,0],2)
    cv2.circle(im,far,5,[0,0,255],-1)


# Corner coordinates
#coordinates = [(topmost[0],topmost[1]), (bottommost[0], topmost[1])
,(rightmost[0], rightmost[1]), (leftmost[0],leftmost[1])]
coordinates = [(155,348),(471,352),(569,402),(58,394)]


# Put coordinates in array
pts = np.array(coordinates, dtype = "float32")
print pts
(tl, tr, br, bl) = pts


# compute the width of the new image, which will be the
# maximum distance between bottom-right and bottom-left
# x-coordiates or the top-right and top-left x-coordinates
widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
```

```
widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
maxWidth = max(int(widthA), int(widthB))


# compute the height of the new image, which will be the
# maximum distance between the top-right and bottom-right
# y-coordinates or the top-left and bottom-left y-coordinates
heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
maxHeight = max(int(heightA), int(heightB))


# now that we have the dimensions of the new image, construct
# the set of destination points to obtain a "birds eye view",
# (i.e. top-down view) of the image, again specifying points
# in the top-left, top-right, bottom-right, and bottom-left
# order
dst = np.array([
   [0, 0],
   [maxWidth - 1, 0],
   [maxWidth - 1, maxHeight - 1],
   [0, maxHeight - 1]], dtype = "float32")


# compute the perspective transform matrix and then apply it
M = cv2.getPerspectiveTransform(pts, dst)
warped = cv2.warpPerspective(img, M, (maxWidth, maxHeight))

resize = cv2.resize(warped, (640,480))

# show the original and warped images
```

```
cv2.imshow("HSV", hsv)
cv2.imshow("Original", img)
cv2.imshow("Trapezoid found", im)
cv2.imshow("Warped", resize)
#cv2.imwrite("areas/edge.jpg",resize)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

photo.py

```
# Image Library
import Image
import cv
import cv2
import os
import glob
from NAOqi import ALProxy
import numpy as np
import config as c
from sys import executable
from subprocess import Popen

image     = "areas/area.jpg"
directory = c.DATASET_DIRECTORY
negative = c.NEGATIVE
positive = c.POSITIVE

def openCamera():
   # Live Camera
   Popen([executable, 'video.py'])

def perspective():
   # Perspective Correction
   Popen([executable, ’perspective.py’])

def process():
   # Preprocessing
   Popen([executable, ’preprocess.py’])

def ball():
   # Preprocessing
   Popen([executable, ’ball.py’])

def takePhoto():

  # Setting up proxy
  cameraProxy = ALProxy("ALVideoDevice", c.IP, c.PORT)

  # Camera ID
  cameraProxy.kCameraSelectID = c.CAMERAID

  # Camera Parameters
  cameraProxy.setParam(cameraProxy.kCameraSelectID,c.CAMERAID)

  # Subscribe Camera Proxy
  videoClient = cameraProxy.subscribe("python_client",
  c.RESOLUTION, c.COLORSPACE, 5)

  # image[6] contains ASCII
  NAOImage = cameraProxy.getImageRemote(videoClient)

  # Unsubscribe Camera Proxy
  cameraProxy.unsubscribe(videoClient)
  # Image Size and Pixel Array
  imageWidth  = NAOImage[0]
  imageHeight = NAOImage[1]
  imageArray  = NAOImage[6]

  # Create an Image
  im = Image.frombytes("RGB", (imageWidth, imageHeight), imageArray)
  img = np.array(im)
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  #ret,thresh = cv2.threshold(graY144   ,0,255,
  cv2.THRESH_BINARY+cv2.THRESH_OTSU)

  # Save image.
  cv2.imwrite('image.png',img)

  # Show image
  #cv2.imshow('Image',thresh)
  print "Photo Taken..."

def cropImage():
   img = cv2.imread("tmp/contour.jpg")
   i = 0
   if os.path.isdir(directory) == False:
      os.mkdir(directory)  # Create a folder
      os.chdir(directory)  # Change directory
   else:
      os.chdir(directory)  # Change directory

   os.mkdir("negative")
   os.mkdir("positive")
```

```
    for v in range(0, 640, 20):
        for c in range (0, 480, 20):
            crop_img = img[c:20+c, v:20+v] # Crop from x, y, w, h
            #cv2.imshow("cropped", crop_img)
            cv2.imwrite(str(i) + '.png', crop_img)
            i += 1
    print "Image Cropped and dataset created..."


def nameChanger():
    global positive
    global negative
    os.chdir(negative)
    for i, f in enumerate(glob.glob('*.png')):
        print "%s -> %s.png" % (f, i)
    os.rename(f, "%s.png" % i)

    os.chdir(positive)
    for i, f in enumerate(glob.glob('*.png')):
    print "%s -> %s.png" % (f, i)
    os.rename(f, "%s.png" % i)



def drawGrid():
    img = cv2.imread(image)
    x1 = 0
    x2 = 700
    for k in range(0, 700, 20):
        y1 = k
        y2 = k
        cv2.line(img,(x1,y1),(x2,y2),(255,0,0),1)

    y1 = 0
    y2 =700
    for k in range(0, 700, 20):
        x1 = k
        x2 = k
        cv2.line(img,(x1,y1),(x2,y2),(255,0,0),1)

    cv2.imwrite('image.png', img)


if __name__ == '__main__':
  showImage()
```

preprocess.py

```
import cv2
import numpy as np
from matplotlib import pyplot as plt


# EDGE DECTECTION

image = cv2.imread('areas/edge.jpg',0)
edges = cv2.Canny(image,50,70)
blur = cv2.GaussianBlur(edges,(5,5),100)
cv2.imwrite('tmp/edge.jpg',blur)
```

```
#CORNER

filename = 'tmp/edge.jpg'
img = cv2.imread(filename)
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

gray = np.float32(gray)
dst = cv2.cornerHarris(gray,2,3,0.04)
#dst = cv2.dilate(dst,None)

cornerimg = cv2.convertScaleAbs(dst)

cornershow = cornerimg.copy()

# iterate over pixels to get corner positions
w, h = gray.shape
for y in range(0, h):
  for x in range (0, w):
    #harris = cv2.cv.Get2D( cv2.cv.fromarray(cornerimg), y, x)
    #if harris[0] > 10e-06:
    if cornerimg[x,y] > 164:
     # print("corner at ", x, y)
      cv2.circle( cornershow,  # dest
            (y,x),       # pos
            4,           # radius
            (255,0,0)    # color
            )
cv2.imwrite('tmp/corner.jpg',cornershow)
```

```
# CONTOUR

im = cv2.imread('tmp/corner.jpg')
imgray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
ret,thresh = cv2.threshold(imgray,127,255,0)
contours, hierarchy = cv2.findContours(thresh,cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

len(contours)
cnt = contours[0]
print(len(cnt))

for h,cnt in enumerate(contours):
    # Draw rectangles around and inside obstacles
    rect = cv2.minAreaRect(cnt)
    box = cv2.cv.BoxPoints(rect)
    box = np.int0(box)
    #print box
    if box[0][0] > 100:
        cv2.drawContours(im,[box]*2,0,(255,255,255),-50)
        cv2.drawContours(im,[box]*2,0,(255,255,255),50)



#SHOW IMAGES

plt.subplot(221),plt.imshow(image,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(222),plt.imshow(edges,cmap = 'gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
plt.subplot(223),plt.imshow(cornershow,cmap = 'gray')
plt.title('Corner Image'), plt.xticks([]), plt.yticks([])
plt.subplot(224),plt.imshow(im,cmap = 'gray')
plt.title('Contour Image'), plt.xticks([]), plt.yticks([])

cv2.imwrite('tmp/contour.jpg',im)

resized_image = cv2.resize(im, (640, 480))
cv2.imwrite('areas/area2.jpg',resized_image)

plt.show()
```

svm.py

```
import os
import sys
import fnmatch
import getopt
import cv2
import numpy as np
import numpy
import datetime
from sklearn import svm
from heapq import *
from gasp import *
from matplotlib import pyplot as plt
from math import degrees, atan2
from NAOqi import ALProxy


number_of_bins = 64
positive = 'dataset/positive'
negative = 'dataset/negative'
```

```
ds = ’dataset’
data = []
angle = 0


# Get all ’png’ images from ’negative’ and ’positive’ folder
def getImages():
    imageFiles = []
    for i in range(2):
        if i == 0:
            path = positive
        elif i == 1:
            path = negative
        for j in sorted(os.listdir(path)):
            if fnmatch.fnmatch(j, ’*.png’):
                imageFiles.append(j)
        i += 1
    return imageFiles


# Returns histogram result
def getHistogram(imageFiles):
    image = cv2.imread(imageFiles)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    histogram = cv2.calcHist([gray],[0],None,[number_of_bins],
    [0,number_of_bins])
    transp = histogram.transpose()
    return transp.astype(np.float64)


# Gets all pictures’ histograms
```

```
def getHistograms():
    images = getImages()
    histogramMap = {}
    for i in images:
        im = positive +’/’+ i
        histogramMap[im] = getHistogram(im)
    for i in images:
        im = negative +’/’+ i
        histogramMap[im] = getHistogram(im)
    return histogramMap.values()


# Sets the values positive 1 negative 0 for svm values
def getValues():
    values = []
    for i in range(2):
        if i == 0:
            path = positive
            j = 0
        elif i == 1:
            path = negative
            j = 1
        for i in sorted(os.listdir(path)):
            values.append((j,))
    return values


# Splits the matrix in desired format
def split(mtx,num):
    matrix = np.array(mtx)
    matrix_splitted = np.array(np.split(matrix, num))
    return np.flipud(matrix_splitted)



# SVM learn and classify
def train():
    now = datetime.datetime.now()
    array = []
    trainData = map(lambda x: x[0], getHistograms())
    value = getValues()

    classify = svm.SVC(kernel='linear')
    classify.fit(trainData, value)



    for j in range(768):
        predict = getHistogram(ds +'/'+ str(j)+'.png')
        result = classify.predict(predict)

        if result == [1]:
            i = 0
        else:
            i = 1
        array.append(i)
    array = split(array,32).T
    #array = np.flipud(array)
    array = np.fliplr(array)

    print " "
    print "Prediction Time : " + str(datetime.datetime.now() - now)
    print " "
    print "####################################################"
    print "                    Area                    "
    print "##################################################"
    print " "
    print array

    #array[0,0] = 3
    return array




# A* Algorithm
def heuristic(a, b):
    return (b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2

def path():

    array = train()
    start = (24,1)
    goal  = (2,25)

    neighbors = [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]

    closeSet = set()
    cameFrom = {}
    gscore = {start:0}
    fscore = {start:heuristic(start, goal)}
    oheap = []

    heappush(oheap, (fscore[start], start))

    while oheap:
```

```
current = heappop(oheap)[1]

if current == goal:
   global data
   while current in cameFrom:
      data.append(current)
      current = cameFrom[current]



   ## Draws path
img = cv2.imread('areas/edge.jpg',0)

   for i in range(0,len(data)) :
        x = int(data[i:][0][0])  #takes first element of first list
        y = int(data[i:][0][1]) #takes second element of first list
 j = i + 1
 if j>len(data)-1:
  j = len(data)-1     # should be -1 coz no more list
 z = int(data[j:][0][0]) # takes first element of second list
 t = int(data[j:][0][1]) # takes second element of second list
        array[x,y] = 4   # puts 4 in array
 cv2.circle(img,(y*20,x*20), 2, (255,0,0), -1)  # circle path
 cv2.line(img,(y*20,x*20),(t*20,z*20),(5,5,2),5)  # line path

      #print x
      #print y
   print " "
   print "####################################################"
   print "                Area  with PATH                  "
   print "####################################################"
```

```
    print " "
    print array
print data[::-1]

cv2.imwrite('areas/area-path.png',img)

 # SHOW PICTURES
org = cv2.imread('areas/edge.jpg',0)
path = cv2.imread('areas/area-path.png',0)
plt.subplot(221),plt.imshow(org,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(222),plt.imshow(path,cmap = 'gray')
plt.title('Path Image'), plt.xticks([]), plt.yticks([])
plt.show()

    return data

closeSet.add(current)
for i, j in neighbors:
    neighbor = current[0] + i, current[1] + j
    tentative_g_score = gscore[current] + heuristic(current, neighbor)
    if 0 <= neighbor[0] < array.shape[0]:
        if 0 <= neighbor[1] < array.shape[1]:
            if array[neighbor[0]][neighbor[1]] == 1:
                continue
        else:
            # array bound y walls
            continue
    else:
        # array bound x walls
        continue
```

```
            if neighbor in closeSet and tentative_g_score >=
            gscore.get(neighbor, 0):
                continue

            if  tentative_g_score < gscore.get(neighbor, 0) or neighbor not in
            [i[1]for i in oheap]:
                cameFrom[neighbor] = current
                gscore[neighbor] = tentative_g_score
                fscore[neighbor] = tentative_g_score + heuristic(neighbor, goal)
                heappush(oheap, (fscore[neighbor], neighbor))
                path = heappush(oheap, (fscore[neighbor], neighbor))

    return False

# End of A* algorithm



# Bearing Algorithm
def gb(x, y, center_x, center_y):
    global angle
    angle = degrees(atan2(y - center_y, x - center_x))
    bearing1 = (angle + 360) % 360
    bearing2 = (90 - angle) % 360
    #print "gb: x=%2d y=%2d angle=%6.1f bearing1=%5.1f bearing2=%5.1f" %
    (x, y, angle, bearing1, bearing2)
    #print angle
    return angle

def moveToPoint():
    checkpoint = []
```

```
point_list = []
way_point = []
angle_point = []
coordinates = []
cm = 0.07
for i in range(len(data)-1):
   gb(data[i][0],data[i][1],data[i+1][0],data[i+1][1])
   first_point = (data[i][0],data[i][1])
   second_point = (data[i+1][0],data[i+1][1])
   lst = (angle, first_point, second_point)
   point_list.append(lst)

way_point.append(point_list[0][1])

for i in range(len(point_list)-1):
   if point_list[i][0] != point_list[i+1][0]:
      way_point.append(point_list[i+1][1])

way_point.append(point_list[len(point_list)-1][1])
way_point = way_point[::-1]

print point_list
print way_point

for i in range(len(way_point)-1):
   gb(way_point[i][0],way_point[i][1],way_point[i+1][0],
   way_point[i+1][1])
   first_point = (way_point[i][0],way_point[i][1])
   second_point = (way_point[i+1][0],way_point[i+1][1])
   lst = (angle,second_point)
   if i == 0:
        angle_point.append(data[-1])
    angle_point.append(lst)
    if i == 0:
        coordinates.append(tuple(numpy.subtract(angle_point[0],
        angle_point[1][1])))
    if i > 0:
        coordinates.append(tuple(numpy.subtract(angle_point[i][1],
        angle_point[i+1][1])))
print coordinates

motion = ALProxy("ALMotion", "192.168.10.13", 9559)
tts = ALProxy("ALTextToSpeech", "192.168.10.13", 9559)
postureProxy = ALProxy("ALRobotPosture", "192.168.10.13", 9559)
motion.moveInit()
motion.setStiffnesses("Body", 1.0)
motion.setMoveArmsEnabled(True, True)
motion.setMotionConfig([["ENABLE_FOOT_CONTACT_PROTECTION", False]])
    motion.setExternalCollisionProtectionEnabled( "All", False )
    print "Protection Off"
    tts.say("Hummm, it seams easy!")

for i in range(len(coordinates)):
    print "Walking"
    print coordinates[i][0]*cm, coordinates[i][1]*cm
    if coordinates[i][0] > 0 and coordinates[i][1] < 0:
        motion.moveTo(0, 0, -0.785398,
             [ ["MaxStepFrequency", 0.5],  # low frequency
               ["TorsoWy", 0.1] ])         # torso bend 0.1
        motion.waitUntilMoveIsFinished()
        motion.moveTo(coordinates[i][0] * cm, 0, 0,
             [ ["MaxStepFrequency", 0.5],  # low frequency
        ["TorsoWy", 0.1] ])        # torso bend 0.1 rad in front)
    motion.waitUntilMoveIsFinished()
    motion.moveTo(0, 0, 0.785398,
        [ ["MaxStepFrequency", 0.5],  # low frequency
          ["TorsoWy", 0.1] ])        # torso bend 0.1
    motion.waitUntilMoveIsFinished()

elif coordinates[i][0] > 0 and coordinates[i][1] == 0:
    motion.moveTo(coordinates[i][0] * cm, 0, 0,
        [ ["MaxStepFrequency", 0.5],  # low frequency
          ["TorsoWy", 0.1] ])        # torso bend 0.1 rad in front))
    motion.waitUntilMoveIsFinished()

elif coordinates[i][0] > 0 and coordinates[i][1] > 0:
    motion.moveTo(0, 0, 0.785398,
        [ ["MaxStepFrequency", 0.5],  # low frequency
          ["TorsoWy", 0.1] ])        # torso bend 0.1
    motion.waitUntilMoveIsFinished()
    motion.moveTo(coordinates[i][0] * cm, 0, 0,
        [ ["MaxStepFrequency", 0.5],  # low frequency
          ["TorsoWy", 0.1] ])        # torso bend 0.1 rad in front))
    motion.waitUntilMoveIsFinished()
    motion.moveTo(0, 0, -0.785398,
        [ ["MaxStepFrequency", 0.5],  # low frequency
          ["TorsoWy", 0.1] ])        # torso bend 0.1
    motion.waitUntilMoveIsFinished()

elif coordinates[i][0] == 0 and coordinates[i][1] < 0:
    motion.moveTo(0, 0, -1.5708,
        [ ["MaxStepFrequency", 0.5],  # low frequency
          ["TorsoWy", 0.1] ])        # torso bend 0.1
        motion.waitUntilMoveIsFinished()
        motion.moveTo(abs(coordinates[i][1]) * cm, 0, 0,
            [ ["MaxStepFrequency", 0.5],  # low frequency
              ["TorsoWy", 0.1] ])        # torso bend 0.1 rad in front))
        motion.waitUntilMoveIsFinished()
        motion.moveTo(0, 0, 1.5708,
            [ ["MaxStepFrequency", 0.5],  # low frequency
              ["TorsoWy", 0.1] ])        # torso bend 0.1
        motion.waitUntilMoveIsFinished()

    elif coordinates[i][0] < 0 and coordinates[i][1] == 0:
        motion.moveTo(0, 0, 1.5708,
            [ ["MaxStepFrequency", 0.5],  # low frequency
              ["TorsoWy", 0.1] ])        # torso bend 0.1
        motion.waitUntilMoveIsFinished()
        motion.moveTo(abs(coordinates[i][0]) * cm, 0, 0,
            [ ["MaxStepFrequency", 0.5],  # low frequency
              ["TorsoWy", 0.1] ])        # torso bend 0.1 rad in front))
        motion.waitUntilMoveIsFinished()
        motion.moveTo(0, 0, -1.5708,
            [ ["MaxStepFrequency", 0.5],  # low frequency
              ["TorsoWy", 0.1] ])        # torso bend 0.1
        motion.waitUntilMoveIsFinished()
motion.moveTo(0.6,0,0,
[ ["MaxStepFrequency", 0.5],  # low frequency
  ["TorsoWy", 0.1] ])        # torso bend 0.1 )
postureProxy.goToPosture("Sit", 1.0)
tts.say("Finish")
print "Finish"
```

video.py

```
import sys

from PyQt4.QtGui import QWidget, QImage, QApplication, QPainter
from NAOqi import ALProxy

# To get the constants relative to the video.
import vision_definitions


class ImageWidget(QWidget):
    """
    Tiny widget to display camera images from NAOqi.
    """
    def __init__(self, IP, PORT, CameraID, parent=None):
        """
        Initialization.
        """
        QWidget.__init__(self, parent)
        self._image = QImage()
        self.setWindowTitle('NAO')

        self._imgWidth = 640
        self._imgHeight = 480
        self._cameraID = CameraID
        self.resize(self._imgWidth, self._imgHeight)

        # Proxy to ALVideoDevice.
        self._videoProxy = None
    # Our video module name.
    self._imgClient = ""

    # This will contain this alImage we get from NAO.
    self._alImage = None

    self._registerImageClient(IP, PORT)

    # Trigget ’timerEvent’ every 100 ms.
    self.startTimer(100)


def _registerImageClient(self, IP, PORT):
    """
    Register our video module to the robot.
    """
    self._videoProxy = ALProxy("ALVideoDevice", IP, PORT)
    resolution = vision_definitions.kQVGA  # 320 * 240
    colorSpace = vision_definitions.kRGBColorSpace
    self._imgClient = self._videoProxy.subscribe("_client",
    resolution, colorSpace, 5)

    # Select camera.
    self._videoProxy.setParam(vision_definitions.kCameraSelectID,
                              self._cameraID)


def _unregisterImageClient(self):
    """
    Unregister our NAOqi video module.
```

```
        """
        if self._imgClient != "":
            self._videoProxy.unsubscribe(self._imgClient)


    def paintEvent(self, event):
        """
        Draw the QImage on screen.
        """
        painter = QPainter(self)
        painter.drawImage(painter.viewport(), self._image)


    def _updateImage(self):
        """
        Retrieve a new image from NAO.
        """
        self._alImage = self._videoProxy.getImageRemote(self._imgClient)
        self._image = QImage(self._alImage[6],           # Pixel array.
                             self._alImage[0],           # Width.
                             self._alImage[1],           # Height.
                             QImage.Format_RGB888)


    def timerEvent(self, event):
        """
        Called periodically. Retrieve a NAO image, and update the widget.
        """
        self._updateImage()
        self.update()
```

```
  def __del__(self):
    """
    When the widget is deleted, we unregister our NAOqi video module.
    """
    self._unregisterImageClient()



if __name__ == '__main__':
  IP = "192.168.10.13"  # Replace here with your NAOQi's IP address.
  PORT = 9559
  CameraID = 0

  # Read IP address from first argument if any.
  if len(sys.argv) > 1:
    IP = sys.argv[1]

  # Read CameraID from second argument if any.
  if len(sys.argv) > 2:
    CameraID = int(sys.argv[2])


  app = QApplication(sys.argv)
  myWidget = ImageWidget(IP, PORT, CameraID)
  myWidget.show()
  sys.exit(app.exec_())
```