The “Piano Movers Problem”

Vision and Robotics Workshop

By

Tomer Livneh

Instructor: Dr. Nir Sochen
Lab Manager: Rami Ben-Ari

November 2006
תוכן

שאלוني וitori עםיהו זכרוכי-
שאלוני, זכרו-כלי, זכרו-כלי, זכרו-כלי, זכרו-כלי.
Preface

“Given an open subset U in n-dimensional space and two compact subsets C₀ and C₁ of U, where C₁ is derived from C₀ by a continuous motion, is it possible to move C₀ to C₁ while remaining entirely inside U?”²

This is what the “piano movers problem” is all about according to Wolfram Mathworld website (See [2]).

As it turns out the problem is very reach and many solutions were and still are developed. This paper will summarize a project done in the scope of a workshop in vision and robotics at Tel-Aviv University. The project deals mainly with path planning and controlling a robot. I will present both theoretical and practical issues I’ve encountered during this project and try to keep it clear and simple.

The first and second chapters are a description of the problem and how we’re going to deal with it. Chapter 3 deals with the path finding task and without doubt, is the most important part of this paper. Chapters 4 and 5 cover the control of the robot. The appendices in the end include elaboration of some issues which didn’t fit in the body of the text.
# Table of contents

1. **Problem definition and model assumptions** .................................................. 4  
2. **Solution strategy** .................................................................................. 5  
3. **Path planning** ...................................................................................... 6  
   3.1 Probabilistic Road Maps (PRM) ................................................................. 7  
   3.2 Analysis of PRM (Revised from [1]) .......................................................... 8  
   3.3 Graph search – A* algorithm ................................................................. 12  
   3.4 Optimization of the path ....................................................................... 14  
4. **Controlling the robot** ........................................................................... 17  
   4.1 Getting the robot state ...................................................................... 18  
5. **Cleaning the image** ............................................................................ 20  
6. **Test results** ......................................................................................... 21  
   Appendix A: The Pinhole model ............................................................... 24  
   Appendix B: The distance function ........................................................... 26  
   Appendix C: changing the robots direction ............................................. 27  
   Appendix D - Software & Hardware ....................................................... 28  
References .................................................................................................... 31
1. Problem definition and model assumptions

As presented in the preface the “Piano mover’s problem” is a mathematical problem in which we are to determine a certain property of some given space. This project is more of an “engineering” project in that the problem is to plan and control the motion of a real robot.

Basically the problem is very simple:
A robot stated at a scene with obstacles should be driven to an arbitrary given target point. Of course, the target point must be free of obstacles. The means to be used are a web cam situated above the scene, a caterpillar like robot with IR interface, enabling it to receive orders in real time and a computer to receive the camera output, perform calculations and produce output in the form of instructions to the robot.
The desired solution is a computer application that gets the target point as input and automatically plan and control the robot motion until it gets there.

We will assume that the image camera is stated high enough so the picture produced is not distorted in a way that changes the angles between lines. i.e. if \( l_1, l_2 \) are two lines in the scene and \( T(l_1), T(l_2) \) are there image as “seen” by the camera then \( \Box (l_1, l_2) = \Box (T(l_1), T(l_2)) \). For that matter we will also assume that the camera projection plane is parallel to the floor. For more details refer to appendix A and to [3].

The camera produce a gray scale image and we will assume the background is much brighter then the objects at the scene (robot and obstacles) so by using a simple threshold we can distinguish the objects at scene from the background.

Another assumption is that the robot has two symmetry axes parallel to the floor. This assumption will be used when calculating the robots location.

Last, though the scene is two dimensional our robot state varies not only in its position (translational degrees of freedom) but also in its direction (rotational degree of freedom) and we have a problem with three degrees of freedom. This rotational degree of freedom is cyclic which makes the problem a lot more complex. As we will see in the path planning chapter, we are able to make a simple, yet useful simplification in order to treat the problem as if it had only two (translational) degrees of freedom.
2. Solution strategy

When coming to solve the problem there are two major issues to be considered:

The first is planning a good (i.e. short) path from the robot origin location to the target. The second is controlling the robot on its way to the target according to the path planned.

Both of these issues require us to have reliable information regarding our scene but before we address this issue it would be wise to find out exactly what kind of information we need to produce. The next chapter will deal with the path planning algorithm.
3. Path planning

At the very beginning of the work, the path planning problem was identified as the main issue of this project. The desired path should be short and the application should calculate it in a reasonable time.

There are two major types of path planning algorithms. In the first, one is trying to capture the space geometry by identifying and characterizing the objects in the scene. The second type consists of sampling based planners, in which we sample different states of the robot and then uses some method, generally known as the “collision detector”, to determine whether these states are free of obstacles. Consider figure 3.1 as an illustration.

Figure 3.1

Example of a 2-d scene with obstacles. The objects numbered 1 to 7 are the obstacles. P1 and P2 are potential states of the robot.

The planners of the first category will try to represent the fact that the scene in figure 3.1 consists of 7 obstacles, each holds certain properties such as: boundaries, volume and maybe even convexity. Later on, this information will be used to plan the path for the robot. Sampling based planners, on the other hand, do not try to achieve this kind of insight about the scene. Instead all we need is a method by which we can determine whether a given state of the robot is free (P1) or occupied (P2).

Achieving a full representation of the scene is a problem of it’s on in computational geometry and might take a lot of processing time, depending on the information produced. Sampling based algorithms are usually characterized in being fast and simple. The sampling scheme can vary according to need. More details about sampling schemes can be found in [1].
After considering several options the algorithm chosen for the path planning task was an algorithm known as “PRM” followed by an algorithm called “A*” (A-star). Both of them will be presented and the PRM algorithm will be analyzed in order to show it’s “correctness”.

3.1 Probabilistic Road Maps (PRM)

PRM is a well studied algorithm for path planning and has many variations. The algorithm used in this work is the basic PRM. Though it is possible to analyze PRM with respect to our private problem, I would like to introduce a more general analysis, of PRM operating in $\mathbb{R}^d$, using the common phrases in this field.

For that matter I would like to introduce of the concept of “Configuration space” is in order.

The Configuration space is defined as the group of all potentially possible states of the robot. Each of these states is referred as a configuration. Given a certain configuration space we will be interested in the group of all free configurations, i.e. all the states of the robot where it doesn’t collides with an obstacle. We will normally mark this subset of the configuration space by $Q_{\text{free}}$ where $Q$ stands for the entire space. A free configuration is a point $q \in Q_{\text{free}}$.

Prior to the formal presentation I’ll try to give a general description of the way PRM operates.

The basic PRM constructs a graph $G = (V, E)$. The nodes in $V$ are a set of robot configurations chosen by some method over $Q_{\text{free}}$. An edge in $E$ between two nodes $n_1$ and $n_2$ represent the fact that it is possible to get directly from $n_1$ to $n_2$. The function used to check this property and plan the local path between two nodes is generally referred as the “local planer” and its implementation should reflect the robot maneuvering potential. We will mark the local planner with the Greek letter $\Delta$.

The name roadmap arises from the simple analogy between nodes and edges of $G$ to junctions and roads of an everyday roadmap.

According to the basic PRM we first sample a predefined number of points from $Q_{\text{free}}$. Then, for each node $n$ we find its $k$ closest neighbors according to a “distance function” $\text{dist}$ (See Appendix B for more details). The distance function should reflect the probability that the local planner will succeed in finding a path between two nodes. Finally, we use the local planner to check whether it is possible to connect $n$ to each of its $k$ closest neighbors. If the answer is yes we add an appropriate edge to our graph.

When coming to answer a query we first add the origin and target configurations to our graph, and then use one of the many algorithms out there in order to a path between the two.
This procedure of first building a roadmap and only then get to answer the query makes the basic PRM a multi-query algorithm.

It is now time to present and analyze our algorithm formally.

**Algorithm 1 – Roadmap Construction Algorithm (taken from [1])**

Input:
- \( n \): number of nodes to put in the roadmap
- \( k \): number of closest neighbors to examine for each configuration

Output:
- A roadmap \( G = (V, E) \)

\[
\begin{align*}
V & \leftarrow \emptyset \\
E & \leftarrow \emptyset \\
\text{while} \ |V| < n \text{ do} & \\
\quad q & \leftarrow \text{a random configuration in } Q \\
\quad \text{until } q \text{ is collision – free} & \\
\quad V & \leftarrow V \cup \{q\} \\
\end{align*}
\]

\[
\begin{align*}
\text{end while} & \\
\text{for all } q \in V & \\
\quad N_q & \leftarrow \text{the } k \text{ closest neighbors of } q \text{ chosen from } V \text{ according to } \text{dist} & \\
\quad \text{for all } q' \in N_q & \\
\quad \quad \text{if } (q, q') \notin E \text{ and } \Delta(q, q') \neq \text{NIL} & \\
\quad \quad \quad E & \leftarrow E \cup \{(q, q')\} & \\
\quad \quad \text{end if} & \\
\quad \text{end for} & \\
\text{end for} & \\
\end{align*}
\]

**3.2 Analysis of PRM (Revised from [1])**

The goal of this section is to establish the “correctness” of PRM. It will be shown that if there is a free path between two configurations \( a \) and \( b \), then the probability of PRM failing to find a free path between them depends on:

1) The length of the “known” path.
2) The distance of the path from the obstacles.
3) The number of configurations in the roadmap.

The analysis will result in an upper bound on the probability of failing. It also gives us a clue on the type of spaces in which PRM is most likely to fail.
We will begin the analysis with some definitions.

Let $Q_{free}$ be an open subset of $[0,1]^d$ and let $\text{dist}$ be the Euclidean distance metric on $\mathbb{R}^d$.

The local planner of PRM connects points $a, b \in Q_{free}$ only if $\overline{ab} \subseteq Q_{free}$.

**Definition:** A path $\gamma$ in $Q_{free}$ from $a$ to $b$ is a continuous map $\gamma : [0,1] \to Q_{free}$ such that $\gamma(0) = a$ and $\gamma(1) = b$.

**Definition:** The clearance of a path $\gamma$ is the farthest distance away from the path at which a given point can be guaranteed to be in $Q_{free}$. We will denote it by $\text{clr}(\gamma)$.

Formally: $\text{clr}(\gamma) = \sup \{ d \in \mathbb{R} \mid \forall a \in [0,1]^d . (\text{dist}(a, \gamma) < d \to a \in Q_{free}) \}$.

**Definition:** two configurations $a, b \in Q_{free}$ which can be connected by a path $\gamma$ in $Q_{free}$ are said to be connectable.

**Definition:** we will say that PRM is **probabilistically complete** if for any two connectable configurations $a, b$:

$$\lim_{n \to \infty} \Pr[(a, b) \text{ failure}] = 0,$$

where $\Pr[(a, b) \text{ failure}]$ is the probability that PRM fails to answer the query $(a, b)$ after a roadmap with $n$ samples has been constructed.

Before we start the proof of correctness, it will be useful to make the following assumption and observation.

Let $\mu$ denote the volume of a region of space, normalized so $\mu([0,1]^d) = 1$. For any measurable subset $A \subset \mathbb{R}^d$, $\mu(A)$ is its volume.

We will assume that PRM samples points from $Q_{free}$ in a uniform distribution. If this is the case then if $A \subseteq Q_{free}$ is measurable and $x$ is a random point chosen from $Q_{free}$ by the Sampling configurations procedure of PRM, then:

$$\Pr(x \in A) = \frac{\mu(A)}{\mu(Q_{free})}.$$

Let us now prove the correctness of the algorithm.

**Theorem 1 (PRM is probabilistically complete):**

Let $a, b \in Q_{free}$ be connectable. Then the probability that PRM fails to answer correctly the query $(a, b)$ after generating $n$ samples, satisfies the inequality:
Pr[(a, b) Failure] \leq \left[ \frac{2L}{\rho} \right] e^{-\sigma \rho^d n}, \text{ where } L \text{ is the length of the ("known") path } \gamma \text{ connecting } (a, b), \rho = \text{cl } (\gamma), B_1 \text{ is the unit ball in } \square^d \text{ and } \sigma = \frac{\mu(B_1)}{2^d \mu(Q_{\text{free}})}.

**Proof:**

Let \( \rho = \text{cl } (\gamma) \). \( \rho > 0 \) since \( \gamma \subset Q_{\text{free}} \) and \( Q_{\text{free}} \) is open. Denote \( m = \left\lfloor \frac{2L}{\rho} \right\rfloor \) and observe that there are \( m \) points on the path \( a = x_1, \ldots, x_m = b \) such that \( \text{dist}(x_i, x_{i+1}) < \frac{\rho}{2} \). Let \( y_i \in B_{\rho/2}(x_i) \) and \( y_{i+1} \in B_{\rho/2}(x_{i+1}) \) where \( B_r(x) \) is a ball of radius \( r \) around the point \( x \). Then the line \( \overline{y_iy_{i+1}} \) must lie in \( Q_{\text{free}} \) since both endpoints are in the ball \( B_{\rho}(x_i) \) (See Figure 3.2). Let \( I_1, \ldots, I_m \) be a set of indicators such that \( I_i \) witnesses the event that there is a point \( y \in N \) satisfying \( y \in B_{\rho/2}(x_i) \). It follows that \( \text{PRM} \) will succeed in answering the query \( (a, b) \) if \( I_i = 1 \) for all \( 1 \leq i \leq m \). Using the union bound we get:

\[
\Pr[(a, b) \text{ failure}] \leq \Pr \left[ \bigcup_{i=1}^{m} I_i = 0 \right] \leq \sum_{i=1}^{m} \Pr[I_i = 0].
\]

The independence of the events \( I_i = 0 \) is a consequence of the samples being independent. The probability of a single randomly generated point of \( Q_{\text{free}} \) falling in \( B_{\rho/2}(x_i) \) is \( \frac{\mu(B_{\rho/2}(x_i))}{\mu(Q_{\text{free}})} \). Hence the probability that none of the \( n \) uniform, independent samples falls in \( B_{\rho/2}(x_i) \) is \( \Pr[I_i = 0] = \left( 1 - \frac{\mu(B_{\rho/2}(x_i))}{\mu(Q_{\text{free}})} \right)^n \).

The sampling is uniform and independent so

\[
\Pr[(a, b) \text{ failure}] \leq \sum_{i=1}^{m} \Pr[I_i = 0] = m \cdot \Pr[I_i = 0] = \left[ \frac{2L}{\rho} \right] \left( 1 - \frac{\mu(B_{\rho/2}(x_i))}{\mu(Q_{\text{free}})} \right)^n.
\]

noting that \( \frac{\mu(B_{\rho/2}(x_i))}{\mu(Q_{\text{free}})} = \frac{(\rho/2)^d \mu(B_1(x_i))}{\mu(Q_{\text{free}})} = \sigma \rho^d \) and making use of the relation \( (1 - \beta)^n \leq e^{-\beta n} \) for \( 0 \leq \beta \leq 1 \) we obtain the required result:

\[
\Pr[(a, b) \text{ Failure}] \leq \left[ \frac{2L}{\rho} \right] (1 - \sigma \rho^d)^n \leq \left[ \frac{2L}{\rho} \right] e^{-\sigma \rho^d n}. \quad \blacksquare
\]
Note that the last inequality is not necessary to prove PRM completeness, but it may come handy when one wants to estimate the amount of work to be done in order to answer a certain query, i.e. how many samples to take in order to ensure success with certain probability.

The analysis reveals us the fact that PRM is most likely to miss path’s that are very close to obstacles (small \( \rho \)), and might not work if going thru narrow passages is required. We also find that its success probability grows exponentially with larger \( n \) (more samples) and larger \( \sigma \) (lower dimensions), and grows linearly with smaller \( L \)’s, i.e. shorter paths).

**Figure 3.2**

![Figure 3.2](image)

**Implementation**

It was stated in the beginning that we can simplify our problem by eliminating the rotational degree of freedom. We do this by treating our robot as if it was a round disk, centered at some fixed point of the robot, with such radius that every point of the robot is covered by the disk.

This way, a configuration is merely a point on the scene, and it’s free if and only if there are no obstacles closer to it then the radius of the robot.

In order to check if some point belongs to \( Q_{\text{free}} \) or not, we need to check all the points that their distance from that point is less or equal to the robots radius. Since perfect control isn’t possible, we need to create a buffer zone. This means that the robot has an effective radius which takes its movement inaccuracies into consideration.

The robot is treated as a round disk, so instead of repeatedly checking the points around some location, we can simply “blow up” the obstacles by the robots effective radius and treat the robot as if it was a dimensionless point in the new “smaller” free configurations space. This “blow up” is demonstrated in figure 3.3.
The PRM algorithm is implemented by the function `prm2d (prm2d.m)`. The sampling scheme is random and done using the function `getsamples (getsamples.m)`. Each pixel is a configuration and considered free if and only if its value is zero. In order to speed up execution the function tries to sample all the points at once instead of running in a loop.

Choosing the k closest neighbors is done by the function `dummknn (dummknn.m)`. This name is no accident and implies that the function can be improved in terms of runtime complexity.

The distance function is the Euclidean distance and it is implemented inline as part of the `dummknn` function.

The local planner is implemented in the function `isfreepath (isfreepath.m)` and it tries to connect between the origin and target configurations using a straight line.

### 3.3 Graph search – A* algorithm

As mentioned in the previous section there are numerous algorithms designed to search a path on a weighted graph. Some of them, like Dijkstra’s algorithm are guarantied to find the shortest path (The path between the origin and target nodes that have the minimum edges weight). Dijkstra's algorithm works by visiting nodes in the graph starting with the origin. It then repeatedly examines the closest not-yet-examined node, adding its nodes to the set of nodes to be examined. It expands outwards from the origin until it reaches the target. Other graph searching algorithms like the BFS (Best-First-Search) works in a somewhat similar way, except that they have some estimate (called a *heuristic*) of how far from the target any node is. Instead
of selecting the node closest to the origin, it selects the node closest to the target (according to the heuristic). BFS is not guaranteed to find a shortest path. However, it runs much quicker than Dijkstra's algorithm because it uses the heuristic function to guide its way towards the target very quickly.

The A* Algorithm combines between the Dijkstra’s algorithm and BFS. It's like Dijkstra's algorithm in that it can be used to find a shortest path. It's like BFS in that it can use a heuristic to guide itself. In the simple case, it is as fast as BFS:

The secret to its success is that it combines the pieces of information that Dijkstra's algorithm uses (favoring nodes that are close to the origin) and information that BFS uses (favoring nodes that are close to the target). In the standard terminology used when talking about A*, \( g(n) \) represents the cost of the path from the origin to any node \( n \), and \( h(n) \) represents the heuristic estimated cost from node \( n \) to the goal. A* balances the two as it moves from the origin to the target. It maintains two lists: open and closed, where the closed list holds nodes that have already been expanded and the open list contains the neighbors of those in the closed list. Each time through the main loop, it expands the node \( n \) that has the lowest \( f(n) = g(n) + h(n) \).

The analysis of A* won’t be given here and can be found in [4]. Still, one property that rises from the analysis is worth mentioning: if the heuristic \( h \) has the monotone property, i.e. \( h(m) \leq h(n) + \text{cost}(m,n) \) where \( \text{cost}(m,n) \) is the real cost of getting from node \( m \) to node \( n \), then A* is guaranteed to find a shortest path (If one exists). Obviously, if we set \( h(n) \equiv \text{const} \) and there are no negative weight edges on the graph, then we will find the shortest path (In this case, A* performs exactly like Dijkstra’s algorithm).

**Algorithm 2 – A**

Conventions:

\[ h(n) \] – A heuristic estimate of the cost from \( n \) to the target.

\[ g(n) \] – The lowest cost known from the origin to \( n \). Updated during the run.

\[ f(n) = h(n) + g(n) \] – An overall estimate to the cost of getting from the origin to the target thru \( n \). Updated during the run.

**input:**

A weighted graph \( G = (V, E) \)

\( n_i \) – The origin node.

\( n_f \) – The target node.

**output:**

A path from \( n_i \) to \( n_f \) (If one exists)
Piano Movers Problem

Path planning

Close $\leftarrow \emptyset$, Open $\leftarrow n_i$

While Open $\neq \emptyset$

\[ n \leftarrow \overline{n} \in \text{Open with minimal } f(\overline{n}) \]

Close $\leftarrow$ Close $\cup \{n\}$, Open $\leftarrow$ Open $\setminus \{n\}$ $\left\{ \text{Selection} \right\}$

if $n = n_f$

\[ \text{return solution}(n) \] $\left\{ \text{Termination} \right\}$

end if

$N_n \leftarrow \{\text{neighbors of } n\}$

for each $n' \in N_n$

$\text{parent}(n') \leftarrow n$

$g'(n') \leftarrow g(n) + \text{cost}(n,n')$

$f'(n') \leftarrow g'(n') + h(n')$

if $(n' = \overline{n} \in \text{Open} \ \& \ \ f'(n') \geq f(\overline{n}))$

\[ \text{continue} \] $\left\{ \text{Expansion} \right\}$

else if $(n' = \overline{n} \in \text{Close})$

\[ \text{continue} \]

else

Open $\leftarrow$ Open $\setminus \{\text{copy of } n\}$

Open $\leftarrow$ Open $\cup \{n\}$

end if

end for

end while

return failure

3.4 Optimization of the path

Now that we found a free path we can address another issue and that is optimizing it. Naturally, any optimization procedure will force us to obtain more information about the scene or else A* would have find the optimized path. There are numerous ways for optimizing our path, depending on the characteristics we want our path to have. Among them, we can try and make it shorter, smoother (smaller angles if any) or maybe some combination of them. The robot in use moves in straight lines and turns on the spot so making the path smoother will not give us much benefit. Therefore our effort is to make the path shorter. The algorithm used for this task is a greedy, straight forward algorithm. It gets a list of nodes representing the known path and repeatedly tries to skip unnecessary nodes of the path, making use of the local planner. It marks the origin node as the current and then uses the local planner to repeatedly check if it can connect the current node to the nodes at the end of the path. If so, then the process repeats with the node that was connected to the origin taking the origin place.
(becomes the current node). This procedure continues until the current node is the target. The correctness of the algorithm is subject to the correctness of the original path and the local planner and won’t be proven here (It’s easy).

**Algorithm 3 – Optimization of the path**

Conventions:
\[ \Delta \] – The local planner.

Input:
- prevpath – A list of nodes defining free path.

Output:
- newpath - A list of nodes defining a free path as good or better than prevpath.

\[
\begin{align*}
\text{fromidx} & \leftarrow 1 \\
\text{toidx} & \leftarrow \text{size of prevpath} \\
\text{newpathidx} & \leftarrow \text{from}
\end{align*}
\]

\[
\begin{align*}
\text{while} \left( \text{fromidx} \neq \text{toidx} \right) & \quad \text{Termination} \\
\quad \text{while} \left( \text{not } \Delta \left( \text{prevpath} \left[ \text{fromidx} \right], \text{prevoath} \left[ \text{toidx} \right] \right) \right) & \\
\quad \text{end while} \\
\quad \text{add toidx to newpathidx at the end} & \quad \text{Optimization} \\
\quad \text{fromidx} & \leftarrow \text{toidx} \\
\quad \text{toidx} & \leftarrow \text{size of prevpath} \\
\quad \text{end while} \\
\text{newpath} & \leftarrow \text{prevpath} \left[ \text{newpathidx} \right]
\end{align*}
\]

Though not tested it is reasonable to suggest the following improvement:

Algorithm 3 makes use of free configurations already in our roadmap. It doesn’t make use of the fact that each of the local path’s (represented by the edges of the graph) is also made of free configurations (At least in this project). We can utilize this fact by adding points along the local paths as nodes in the graph and only then make use of algorithm 3. This procedure can be repeated a fixed number of times or until no improvement was made.

**Algorithm 4 – “Improved” optimization of the path**

Conventions:
- segment – A function that adds nodes to a path (divides the path into smaller segments).

Input:
- prevpath – A list of nodes defining free path.
- n – Number of maximum iterations to take.
Output:
newpath - A list of nodes defining a free path as good or better than Prevpath.

\[
\text{segpath} \leftarrow \text{segment (prevpath)} \\
\text{newpath} \leftarrow \text{optimized segpath according to algorithm 3} \\
\text{while \((\text{newpath is better then prevpath} \& \& i < n)\)} \\
\quad \text{prevpath} \leftarrow \text{newpath} \\
\quad \text{segpath} \leftarrow \text{segment (prevpath)} \\
\quad \text{newpath} \leftarrow \text{optimized segpath according to algorithm 3} \\
\quad \text{\(i = i + 1\)} \\
\text{end while} \\
\text{return newpath}
\]

**Implementation**

The A* algorithm is implemented in the function Astar (Astar.m).

It uses the function euccost (euccost.m) to calculate the Euclidean distance between two points. This distance is both the heuristic \(h\) and the cost function \(g\), which makes \(h\) monotone and a shortest path is guaranteed.

The optimization (algorithm 3) is implemented in the function optpath (optpath.m).
4. **Controlling the robot**

The control scheme used is very simple and reflects the robot maneuvering abilities: move straight, turn right \left. The path is piecewise linear and the controller takes the robot in a “straight” line from point to point along the path. The simplest way of presenting it is by using the following flow chart.

Figure 4.1

Flow chart of the controller.

There are two squares in this diagram that requires elaboration: Checking if the robot is on target and checking the robot direction. The next section will deal with getting the robot state and direction from a binary image of the scene. The procedure used to get this image is described in chapter 5.
4.1 Getting the robots state

Assuming we have a grayscale image of the scene represented by an m-by-n matrix $I$, we can define its center of mass coordinates by:

$$x_{cm} = \frac{\sum_{j=1}^{n} \left( \frac{m}{i} \sum_{i=1}^{m} I_{ij} \right)}{M}, \quad y_{cm} = \frac{\sum_{i=1}^{m} \left( \frac{n}{j} \sum_{j=1}^{n} I_{ij} \right)}{M},$$

where $M = \sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij}$ is the total mass of the image. If the image is binary and its non-zero elements are symmetrically distributed about some line (axis of symmetry) then the center of mass will lie on that line. If it has two symmetry axes then the center of mass will lie on their intersection. Thus, if we have a binary image of the scene and we eliminate all obstacles from it so only the pixels “belongs” to the robot are ones, we can utilize the symmetry assumption and find the robots location by calculating the center of mass of such image. As mentioned we would like a binary image representing only the robot. This can be done by keeping in memory a binary image $BG$ of the scene without the robot, and subtracting (Group subtraction) this image from a binary image $SC$ of the scene including the robot. The procedure is demonstrated in figure 4.2. In this figure we also see the camera instability effect at the edges. Those are dealt by an opening procedure (To learn about opening and dilating an image, refer to any basic text book about image processing).

Calculating the robots direction is done by repeatedly calculating its location while it moves forward, and finding the vector connecting these points.

Finding the robots location.

The left picture is an image of the scene including the robot. To its right we have the picture of the scene without the robot. The second picture from the right is their subtraction and presents edge effects caused by camera instability. The picture on the right is the result of an opening procedure performed on the third and calculating its center of mass will give us the robots location.
Implementation

The controller is implemented in two main functions. The first is followpath (followpath.m) which gets the waypoints along the planned path and repeatedly calls the function go2point (go2point.m) which takes the robot to the next point in a “straight” line.

When commanding the robot to turn right (for example), there are two possible calls. The first is turn right until ordered otherwise and the second is turn right x units. The direction of the robot is calculated only during the periods it moves forward. For that reason, we must find a way to estimate how much did it turn during a call to a turn right procedure. As it turns out there is a linear relation between the number of units the robot was ordered to turn and the direction change in angles. This relation is revealed by test and can be reversed to find how much units to turn in order to correct the direction. The results are presented in appendix C.

The function go2point is the heart of the controller. It absorbs all the inaccuracies of the robot and camera. It has a direction thresh value and it corrects the robots direction only if it’s heading more then this value away from the target. Similarly, it has a position thresh value and the movement stops as the robots is closer to the target point then this value.

The function go2point uses the functions getlocation (getlocation.m) and calcdirection (calcdirection.m) in order to find the robots current location and direction respectively.

The function getlocation uses the function comass (comass.m) which calculates the center of mass of a grayscale image.
5. Cleaning the image

As suggested above, having reliable information of the scene is a crucial issue in handling the task at hand. We already established that a binary image where 0’s stands for background is enough for our purposes. The camera produces a color image represented in rgb format. Using the camera driver we can eliminate saturation and thus receive a gray scale image. This grayscale image is then converted to a binary image using a simple threshold. The resulting image needs to be cleaned and this is done, again, by an opening procedure. An example of this procedure is given in figure 5.1

Figure 5.1

![Image](image1.png)

The image on the upper left is the original image as received from the camera. To its right, a B&W image after a threshold procedure. The image on the bottom is the result of opening the image on the right with a disk shaped structuring element.

Implementation

Getting a clean image is done using the function getcleanscene (getcleanscene.m). This function uses getscene (getscene.m) in order to get a binary image and then opens it as a cleaning procedure.

The getscene function uses the vfm package (See appendix) in order to capture the camera output and converts it to a binary image. It also cuts the image margins according to the parameters defined in the m file sceneborders.m.
6. Test results

In order to test the system (hardware and software) a “main” function was built. The function name is movepiano (movepiano.m). This function gets the necessary user input, and then runs the functions that were mentioned earlier.

Three settings were tested. The first is relatively simple and uses 250 samples. The other two are a bit more difficult and requires the robot to go thru narrow passages. The two differ from each other only by the number of points added to the roadmap.

They will all be presented side by side for comparing purposes.

Figure 6.1 presents the background returned by the function getcleanscene of settings a, b and c. Note that settings b and c are the same (This means that $Q_{\text{free}}$ is the same in b and c).

![Figure 6.1](image)

The background of three settings.

Figure 6.2 presents the dilated image of the background and the roadmap created in each of the cases. The dilated images were created using a disk shaped structuring element with radius 30. The roadmap of settings a, b and c were created using 250, 100 and 300 samples respectively. The difficulties in capturing narrow passages are demonstrated on the roadmap generated in setting b and solved in setting c by using three times the samples used in b.
In figure 6.3 we can see the final results of the planned paths (original and optimized), as well as the real path that robot went.

All the examples present improved paths generated by the optimization procedure. We can also see that the robot followed the path with relative accuracy so the controller, though simple, worked well. The origin and target locations of settings b and c are essentially the same and the difference in the path found by the application is due to the larger number of samples in setting c.
Figure 6.3

The path found by the A* algorithm is presented by the green line. The red line is the optimized path (according to algorithm 3) – This is the path the robot should have followed. In blue we have the actual path the robot made as captured by the getlocation function.
Appendix A: The Pinhole model

The pinhole model is an approximation for how real world images are captured by the camera. It refers to the camera lens as a single point thru which the camera projects the image onto a plane (which will here by referred as the projection plane) placed a distance f (focus) “behind” the camera lens.

The relation between real world dimensions and the dimensions seen by the camera using the pinhole model is developed and explained in [3]. I will bring only its results.

Using capital letters to mark real world coordinates and lower case letters for the projection coordinates we have the relations: $x = \frac{f}{Z} X$, $\ y = \frac{f}{Z} Y$.

Let the camera distance from the floor be H. We assume that the camera is very high compare to all objects of the scene. This implies that for any given object we can write $Z = H + \Delta Z$ where $|\Delta Z|$ is very small compare to |H|.

In order to show that the angle of the vector pointing from $(x_1, y_1)$ to $(x_2, y_2)$ on the projection plane is a good approximation to the angle of the vector pointing from $(X_1,$
Y₁, Z₁ to (X₂, Y₂, Z₂) (providing |Z₁|,|Z₂| << |H|) we facilitate the relation

\[
\frac{1}{x + \delta} = \frac{1}{x} \cdot \frac{1}{1 + \frac{\delta}{x}} \approx \frac{1}{x}
\]

for |δ| ≪ |x| and obtain:

\[
\tan(\alpha) = \frac{y₂ - y₁}{x₂ - x₁} = \frac{f}{Z₂} \cdot \frac{Y₂}{Z₁} - \frac{f}{Z₁} \cdot \frac{Y₁}{Z₁} = \frac{Y₂}{H + \Delta Z₂} - \frac{Y₁}{H + \Delta Z₁} \approx \frac{Y₂}{H} - \frac{Y₁}{H} = \frac{Y₂ - Y₁}{H}
\]

\[
\frac{x₂ - x₁}{f} = \frac{X₂}{H + \Delta Z₂} - \frac{X₁}{H + \Delta Z₁} \approx \frac{X₂}{H} - \frac{X₁}{H} = \frac{X₂ - X₁}{H}
\]

which implies \( \alpha \approx A \) and a negligible distortion of the angles.
Appendix B: The distance function

The distance function in this implementation is simply the Euclidean metric. It is simple to calculate and imply the actual complexity of getting from one free point to another. It should be noted that the real cost of getting from one configuration \( q_1 \) to another \( q_n \) in a path made of a polygon line \( L \) defined by the break points \( q_1, q_2, \ldots, q_n \), is effected by the length of the path \( L \) (\( \text{length}(L) = \sum_i \text{dist}(q_i, q_{i+1}), \quad i = 1, 2, \ldots, n - 1 \)) but not only. It was mentioned in the body of the text that a suitable dilation of the binary image of the scene will simplify the problem of the path planning by reducing the degrees of freedom from 3 to 2. While this operation has great benefits in terms of implementation simplicity and computation time, it costs us a loss of information (we omitted the rotational degree of freedom).

The turns during the movement also takes time. The Euclidean metric reflects this cost but not accurately.

Let us mark the vector between \( q_i \) and \( q_{i+1} \) by \( u_i \) (\( u_i = q_i - q_{i+1} \)). marking the absolute value of the linear velocity of the robot \( v \) and the absolute value of the rotational velocity of the robot \( \omega \) and using \( \cos \theta = \frac{u_i \cdot u_j}{|u_i| \cdot |u_j|} \equiv \theta_{ij} \)

we obtain the time to follow \( L \):

\[
T(L) = \sum_{i=1}^{n-1} \left| u_i \right| \cdot \frac{1}{v} + \omega \sum_{i=1}^{n-1} \theta_{i,i+1} = \frac{\text{length}(L)}{v} + \omega \sum_{i=1}^{n-1} \theta_{i,i+1}.
\]

Looking at the last expression it is clear that the simple Euclidean distance is simply not reach enough. For example, the Euclidian length of path b in figure B1 is equal to that of path c but it is clear that path b is better in terms of movement time.

However, Things aren’t so bad. Though not exact it still relate to rotations due to the simple fact that the shortest path between two points is a straight line which yields no turns along the way (path a).

Figure B1

Three possible paths between two points. Path “a” is the shortest while path b and c have the same length and differ only by the number of breakpoints.
Appendix C: Changing the robots direction

We have two options when commanding the robot to turn (a turn right for example): one is to issue a command to turn right until another command is issued. The other is to send a command to turn a predefined number of units. The second option enables us to direct the robot to turn as much as we need without monitoring the movement. The relation between the number of units and the angle change in radians is given in graphs C1 and C2. The points on the line represent experimental data and the line is the result of a linear fit done by excel. The data is presented next to the graphs and the line equation above the trend line. The \( R^2 \) parameter shows us that the fit is very good and the relation is indeed linear.

### Graph C1
Angle change per unit turn - Left

\[
y = 0.0376x + 0.0498 \\
R^2 = 0.9999
\]

<table>
<thead>
<tr>
<th>units [#]</th>
<th>angle [Rad]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.4221</td>
</tr>
<tr>
<td>30</td>
<td>1.2013</td>
</tr>
<tr>
<td>50</td>
<td>1.9106</td>
</tr>
<tr>
<td>90</td>
<td>3.4231</td>
</tr>
<tr>
<td>120</td>
<td>4.573</td>
</tr>
</tbody>
</table>

### Graph C2
Angle change per unit turn - Right

\[
y = 0.0386x + 0.0279 \\
R^2 = 0.9998
\]

<table>
<thead>
<tr>
<th>units [#]</th>
<th>angle [Rad]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.3844</td>
</tr>
<tr>
<td>20</td>
<td>0.8273</td>
</tr>
<tr>
<td>40</td>
<td>1.5708</td>
</tr>
<tr>
<td>90</td>
<td>3.526</td>
</tr>
<tr>
<td>120</td>
<td>4.6429</td>
</tr>
</tbody>
</table>
Appendix D - Software & Hardware

Application Environment

The application was developed and tested in Matlab 6.5 under operation system Windows XP.

The Working Directory

Running the application requires putting all the relevant files in a working directory. This directory should be added to Matlab's search path.

To do it from Matlab Interface press:

File→Set Path→Add Folder. Then Browse to your working directory and Press Ok.

It can also be done directly from Matlab's command line by typing:

path('〈your working directory full path〉',path);

The Camera

The camera used in the project was the Lego MindStorms Command Vision camera. In this project Matlab acquires images from the camera using the vfm package (Vision for Matlab).

The vfm utility should can be downloaded from:

http://www2.cmp.uea.ac.uk/~fuzz/vfm/default.html.

After unzipping the package, the files vfm.m and vfm.dll from the folder ‘compiled’ should be copied to the working directory.

In order to get an image from the camera, type for instance:

Im = vfm(‘grab’,1);

Im is \(m \times n \times 3\) array containing the representation of a colorful image in RGB matrix format.

For more information about the vfm package functions type ‘help vfm’.

Installing Software into the Robot RCX

The robot used in this workshop is Lego MindStorms Robotics Inventions system 2.0 model. The commands are sent to the robot by an infrared Command Tower transmitter.

In order to enable Matlab to control the robot motion the following steps should be performed:

In the MVP lab computer there should be two folders: ‘C:\LEJOS’ and ‘C:\j2sdk1.4.2’. Now do as following:
1. Open a cmd window (Click start→run then type cmd and click OK)
2. Change the current directory to:
   `cd c:\LEJOS\bin`
3. Set the LEJOS_HOME environment variable to the directory you installed Lejos into:
   `set LEJOS_HOME = c:\LEJOS`
4. Add the following paths to the PATH environment model:
   `set PATH=%PATH%;c:\LEJOS\bin;c:\j2sdk1.4.2\bin`
5. Add the Lejos classes to the CLASSPATH environment variable:
   `set CLASSPATH = %CLASSPATH%;.;%LEJOS_HOME%\lib\classes.jar;\%LEJOS_HOME%\lib\pcrcxcomm.jar`
6. Set the RCXTTY environment variable to the ‘Tower’ device:
   `set RCXTTY = USB`

Now turn on the robot and note that the command tower is aimed at the robot infrared window (the black rectangle). Then do

7. Install the Lejos operating system into the robot rcx by typing:
   `lejosfirmdl --tty=usb –f`
   The tower should blink green, when you start lejosfirmdll.
8. Now download to the robot rcx its ‘driving’ class by typing:
   `lejosrun --tty=usb –f rover.bin`

After doing the above steps the installation process is over. This procedure should be done only once.

More reference can be found at

**Controlling the Robot Via Matlab**

In order to enable the robot to receive commands from The Matlab interface (only after the above installation process has been done) turn on the robot and press on Run (the green button). Now the robot is waiting for commands.

Matlab sends commands to the robot via special class called ‘josx.vision.RCX’.
The class libraries reside in the MVP lab computer at
‘F:\old_disk\MATLAB6p5\java\rcx’.
So in order to send commands to the robot one must create an instance of the class and then use the class methods.

Here is an example of a Matlab code:

```matlab
Robot = josx.vision.RCX;  % creating a class instance
Robot.forward(30);  % moving forward
pause(2);
```

29
Robot.backward(30); % moving forward
pause(2);
Robot.spinRight(30); % turning right
pause(2);
Robot.spinLeft(30); % turning left

Please note that because of implementation defects the commands might not be consistent with the robot real movement. It means for instance that as response to the forward command the robot moves backwards. Just check it and write your code accordingly.

The Matlab Application

Next is a list of the M-files used by the application and their functionality.

- movepiano.m – The “main” function, guiding the user thru trying the robot.
- prm2d.m – crates a roadmap.
- getsamples.m – returns free points from the scene.
- dummkmnn – finds the k closest neighbors of a node.
- isfreepath.m – check if the straight line between nodes is free (The local planner).
- addnodes – adds nodes to an existing roadmap. Very similar to prm2d.
- euccost.m – returns the distance (in pixels) between two nodes.
- Astar.m – Search for a shortest path on the graph.
- optpath.m – optimizing a correct path.
- followpath.m – the highest level control function.
- go2point - lower level control. takes the robot in a “straight” line to the target point.
- getlocation.m – calculates the robots location based on the current image and a background image.
- calcdirection.m – calculates the direction of a vector pointing from one point to the other.
- comass.m – calculates the center of mass of a grayscale image.
- getcleanscene.m – gets the image from the camera and cleans it.
- getscene.m – gets the image from the camera.
- cutpic.m – cuts the picture according to the borders defined in sceneborders.m.
- graphplot.m – a script (not a function) that presents the graph built.

The application also uses 4 files that hold parameters of the scene and robot:

- sceneborders.m – the borders of the scene.
- setrobotradius.m – The robots “radius”.
- setbufferwide.m – The security zone.
- setturnparameters.m – The constants found according to appendix c, and the time it takes the robot to turn left and right.

All the files must be located at directories belongs to the matlabs search path.
The application is operated by Typing “movepiano” at matlabs command line.
References


