

# **Soar-SLAM**

**A cognitive architecture approach  
to the problem of  
Simultaneous Localization And Map building**

by

Chonarop Jamroendararasame

Submitted in partial fulfillment  
of the requirements for the degree of  
Master of Science  
in Computer Science

at

Ivan G. Seidenberg  
School of Computer Science and Information Systems  
Pace University  
May 2007

## **Thesis Signature (Approval) Page**

I here by certify that this thesis, submitted by Chonarop Jamroendararasame, satisfies the thesis requirements for the degree of Master of Science and has been approved.

---

Dr. D. Paul Benjamin  
Thesis Advisor  
School of Computer Science and Information Systems  
Pace University

---

Date

# **Abstract**

Creating techniques for autonomous mobile robots to observe and process geographical information for human is one of the most interesting research topics in artificial intelligence. Most of the techniques require lots of computation based on mathematic theories and equations.

In order to create such a robot, there are number of problems involve. One of the main concern is the problem of simultaneous localization and map building (SLAM) which asks if it is possible for a mobile robot to start in an unknown environment and then to incrementally build a map of the environment while simultaneously using this map to determine location of itself.

This thesis presents an approach to a new SLAM technique using rule-based system called Soar. Unlike the ordinary geographical data processing, Soar-SLAM processes information based on abstract thought, high-level deliberative reasoning and pattern recognition, like humans.

We will describe our implementation of Soar-SLAM and provide results of our experiments using the Pioneer robot in the Pace University Robotics Lab and a robot simulator.

# Table of Contents

Introduction.....	1
Objectives and Contribution.....	1
Simultaneous Localization And Map building (SLAM).....	2
Difficulties in SLAM.....	2
Pioneer Robot.....	2
Sonar Sensors.....	3
Driving System.....	3
Batteries.....	3
Communicate with the robot.....	4
Sonar Modeling.....	4
Hough Transform.....	6
Soar.....	8
Theory behind Soar.....	8
Rule-based System.....	9
Soar Basic Syntax.....	9
How does Soar work?.....	10
I-Support/O-support working memory element.....	13
ADAPT project.....	13
PlayerStage.....	14
Player.....	14
Stage.....	15
Java-Player.....	16
Advantages of PlayerStage.....	17
Soar-SLAM.....	17
Human with SLAM problem.....	17
Soar approach to SLAM problem.....	18
SoarRobot.....	19
Hough-Transform Java.....	19
Sonar Data Converter.....	20
Noise filter.....	21
Hough transforming.....	24
Extracting Wall.....	25
Synchronization between external world and Soar.....	26
Soar Rules.....	27
Working Memory Structure.....	27
robotagent.soar.....	28
slam.soar.....	29
Elaborate.soar.....	29
CopyWall.soar.....	30
CheckWallRelation.soar.....	32
CheckRobotPosition.soar.....	39

SubGoalElaborate.soar.....	43
Experiment and Result.....	45
Conclusion.....	49
Opportunities for Further Development.....	49
Research more detailed methods about how human solve the problem of SLAM.....	49
Combine information from multiple types of sensors.....	49
Implement Soar Learning.....	49

## List of Figures

Figure 1: (Top) Pioneer robot. (Left) Dimension. (Right) Positions and angles of sonars.....	3
Figure 2: Sonar sensor emitting a short burst of ultrasonic sound to measure the distance.....	4
Figure 3: Object cannot be detected if it is not in the visible angle.....	5
Figure 4: Superimposed of hand measured room model with sonar scan [4].....	5
Figure 5: a line represent by $m, b$ (left) or $r, \theta$ (right).....	6
Figure 6: (Left) Number of lines go through a point on $x, y$ image space. (Right) Plotting $(r, \theta)$ of all lines on $r, \theta$ parameter space.....	7
Figure 7: (Left) 3 points $(P, Q, R)$ line up on $x, y$ image space. (Right) 3 sinusoidal curves plotted from lines that go through $P, Q, R$ points.....	7
Figure 8: The line bisects all points $(P, Q, R)$ generated from $r_0, \theta_0$ using Hough transform.....	8
Figure 9: Soar-SLAM diagram, a part of ADAPT project.....	14
Figure 10: Stage screen shot with two Pioneer robots and serveral boxes.....	15
Figure 11: PlayerStage in a partition box to simulate the Pioneer 2 robot in the Pace Robot lab.....	16
Figure 12: a floor plan example.....	17
Figure 13: Soar-SLAM overall diagram.....	18
Figure 14: (a) positions and angles of transducers in the front of the robot, (b) robot placed in a box partition (in PlayerStage).....	20
Figure 15: sonar data plot in $x, y$ image space after converted.....	21
Figure 16, [3]: Sonar data returns observed from letting robot move along the path in an indoor environment. (dotted line represent real environment).....	21
Figure 17: How to create visible line.....	22
Figure 18, [3]: Sonar data plotted together with visible lines.....	23
Figure 19: Sonar data after run through the noise filter.....	23
Figure 20: Hough space voted with example sonar data (the higher vote number, the darker spot displayed).....	25
Figure 21: Zones on the Hough space that have voting numbers are higher than $k$ .....	25
Figure 22: (left) Lines correspond to $r, \theta$ extracted from Hough space (right) generated lines and Sonar pixels displayed together.....	26
Figure 23: (a) 2 walls line near each other, (b) 2 walls line next to each other.....	33
Figure 24: Show the position shifted between the robot and where it thinks it is. “ $x$ ” is the distance from the wall to the robot, “ $x'$ ” is the virtual distance calculated based on the information in Soar working memory.....	40
Figure 25: Robot placed in an empty rectangular room.....	45
Figure 26: Wall elements in Soar working memory after letting the robot run without Soar-SLAM rules in an empty room for 10 minutes.....	46
Figure 27: Screen shots captured from PlayerStage, the robot simulator.....	46
Figure 28: Wall elements created and update over time.....	47
Figure 29: Wall elements in Soar working memory after let the robot run with Soar-SLAM rules in a box partition for 10 minutes.....	48
Figure 30: Graph presents measured distance from the experiment.....	48

# Introduction

In order to create an autonomous mobile robot to do some tasks which involve high level of understanding and reasoning, we must first create a robot or computer that knows how to understand and reason about the world. Many scientists put work and effort in to this area of artificial intelligence, called cognitive architecture.

The idea of having robots do tasks for human is in our mind for decades, especially when those tasks appear to be risky, disgusting, or monotonous tasks. For instance, how great would it be if we can deploy a fully autonomous robot into an unknown environment such as cape or deep sea, and then it returns with useful geographical information.

To make this dream come true, many researchers spend time researching on many techniques and algorithms to help robots doing the tasks for human.

There are numbers of computer languages and architecture created for this research area. Soar is one of those architectures. It is equipped with capabilities to reason and learn things. Soar's reasoning and learning capabilities are based on research into human cognition. We are using Soar to try to create a mobile robot that reasons and learns like people do.

A mobile robot would not be able to do a task if it can't perceive environmental information. Acquiring and understanding geographical information for a mobile robot is one of the most important and interesting part, and one of the reasons that we talk about Simultaneous Localization and Mapping (SLAM) problem. SLAM has never been implemented in a cognitive architecture before. This is the goal of my thesis.

To solve the SLAM problem, we can not avoid the task of observing environmental information. There are numerous kinds of sensors for the purpose. Of course, the higher performance, the more expensive they would be. Sonar sensor is one of the economic type of sensor which is widely used by many researchers. It comes with very unique capabilities in various circumstances.

To enable the computer to understand raw sensory information, many techniques have been invented. Hough-transform is one of the techniques used to find various kinds of objects from information like a digital image. This thesis uses the Hough-transform to process the raw data, and Soar to do the localization and mapping.

## ***Objectives and Contribution***

This thesis is an effort to develop techniques, using Soar cognitive architecture, to solve the problem of Simultaneous Localization and Mapping for the Pioneer mobile robot [1] based on environmental information from sonar sensors.

The contribution of this work is to provide cognitive algorithms for solving the SLAM problem as a part of the ADAPT project which is implementing unified cognitive architecture for a mobile robot.

## ***Simultaneous Localization And Map building (SLAM)***

The term “Simultaneous Localization And Map building (SLAM)” is the problem involving two main processes, localization and map building. *Localization* is a technique that allows a robot to locate the location of itself in the environment. *Map building or Mapping* is a technique used to construct map of the environment based on available information.

In conclusion, SLAM is mainly talking about whether it is possible for an autonomous vehicle to start in an unknown environment and then to incrementally build a map of the environment while simultaneously using this map to compute absolute vehicle location.

### ***Difficulties in SLAM***

The main difficulty of the SLAM problem is uncertainty. The robot does not know where it is, or what it is looking at. It can only make noisy, probabilistic observations of its surroundings and every time it moves, it does so imprecisely - adding uncertainty into an already uncertain pose.

One of the basic robot navigation system is called Dead reckoning (DR), the process of estimating current robot position based upon a previously determined position, or fix, and advancing that position based upon known speed, elapsed time, and course [2]. But in the real world, there are number of problems to make the DR work perfectly. For instance, the robot might slip during the course. Wheels might not run exactly at the same rate. These are added to the uncertainty.

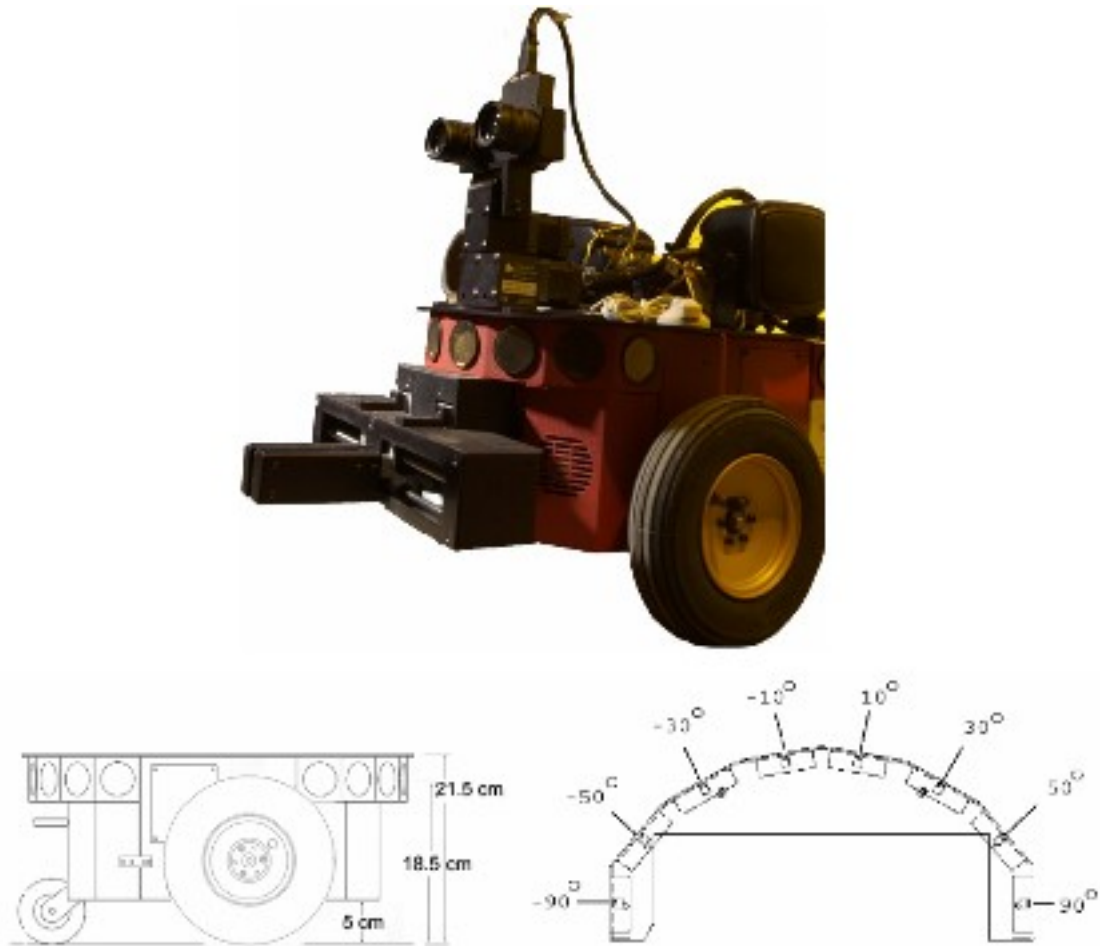
Dead reckoning is like walking around with your eyes closed. If you stand in a location and look around, then close your eyes and try to walk somewhere without looking, you are using dead reckoning. The farther you walk, the less accurate your position will be.

Another uncertainty adder is Sonar sensor. Scanning the environment using sonar give us very limited and less accurate information compare to other kinds of sensors such as laser scanner. Further more, information given by a ring of sonar sensors is sparse and its angular precision is limited by the large sonar beam-width, typically in the range of 20 to 30 degrees [3].

### ***Pioneer Robot***

Pioneer robot is an intelligent mobile robot with solid rubber tires, a two-wheel differential, reversible drive system and a rear caster for balance. It also contains basic components for sensing and navigation in a real-world environment such as multiple sonar sensors, stereo camera, position-speed encoders. These equipments are all managed by an on board micro-controller and mobile-robot server software. Beside all basic components, Pioneer robot also comes with built-in on board computer with both wired and wireless Ethernet cards [1].





*Figure 1: (Top) Pioneer robot. (Left) Dimension. (Right) Positions and angles of sonars*

### *Sonar Sensors*

Out Pioneer robot is equipped with two set of multiplexed sonar arrays, front and rear. Each array contains eight transducers, positioned as shown in the figure 1, that provide object detection and range information. The sonar acquisition rate is 25 Hz (40 milliseconds per sonar per array). Sensitivity ranges from six inches to nearly 16 feet [1].

### *Driving System*

Driving system is based on high-speed, high-torque, reversible-DC motors, each equipped with a high-resolution optical quadrature shaft encoder for precise position and speed sensing and advanced dead-reckoning [1].

### *Batteries*

Pioneer robot may contain up to three, hot-swappable, seven ampere-hour, 12 volts direct-current (VDC) sealed lead/acid batteries (total of 252 watt-hours), accessible though a hinged and latched back door [1].

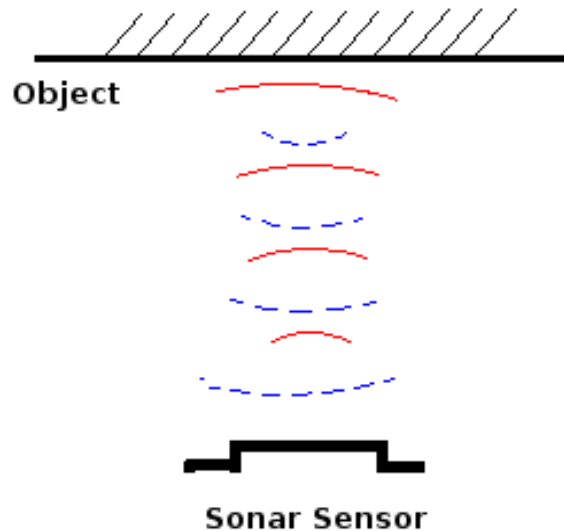
### *Communicate with the robot*

With a server program on the robot on board computer, communication with the robot can be done by sending/receiving binary streams through the Ethernet using basic socket programming.

### **Sonar Modeling**

The term **S**ound **N**avigation and **R**anging, SONAR, usually means to various kinds of sound based ranging sensor. But in robotic researching, the word SONAR is particularly referred to ultrasonic ranging sensor. The sonar sensor works based on two simple facts, sound travels at approximately 1065 feet per second at sea level and the sound at higher frequency is more directional.

Distance between two points can be measured by emitting a short burst of high frequency sound and then measure the time when the echo arrive which is also known as time-of-flight (TOF). Distance can be calculated by cut the TOF into a half and then multiply it with the sound speed.

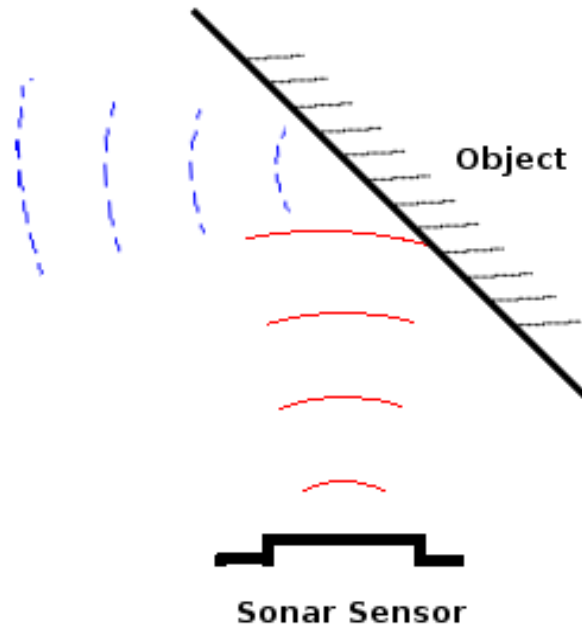


*Figure 2: Sonar sensor emitting a short burst of ultrasonic sound to measure the distance*

Using sonar provides us many advantages such as:

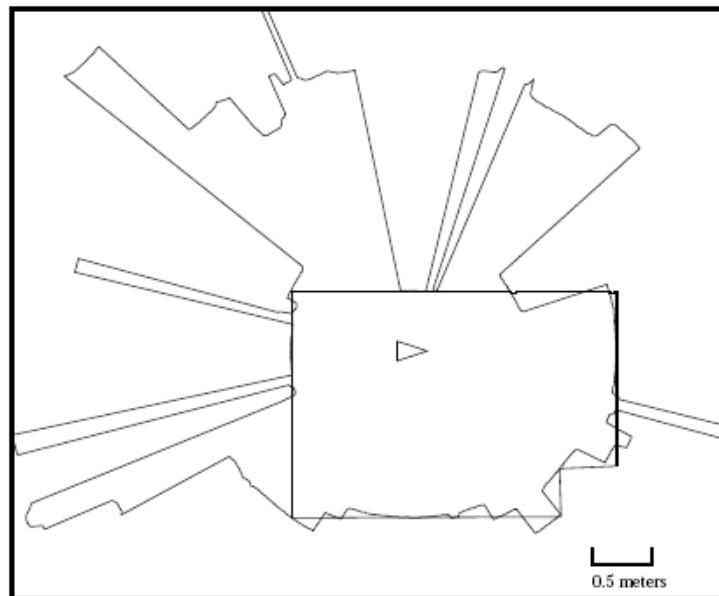
- detect and measure the distance to objects in noisy environment such as dusty, bright.
- detect small objects in longer distances.
- less affected by target materials, surfaces, and colors.
- lower cost to maintenance.

On the down side, sonar sensor is angel limited because it rely on the bounced back sound. Objects which angeled to the sensor outside the visible angel will no be detected or detected but not accurately.



*Figure 3: Object cannot be detected if it is not in the visible angle*

Because of this characteristic of sonar sensor, we cannot purely rely on the information given by sonar. For instance, if we put the sonar sensor in an empty room and then measure all the distances by turning the sensor around. The room model drawn directly from the measured distance would be like figure 4 below.



*Figure 4: Superimposed of hand measured room model with sonar scan [4]*

## Hough Transform

The Hough transform is a feature extraction technique used in digital image processing. It was invented by Richard Duda and Peter Hart in 1972, who called it a "generalized Hough transform" after the related 1962 patent of Paul Hough. The classical transform identifies lines in the image, but it has been extended to identifying positions of arbitrary shapes such as straight lines, circles, ellipses, etc [2].

Usually, when we work on a digital image, we refer to points on an image space where X and Y axes are the horizontal and vertical axes respectively. Each point on this image space is a pair of values (x,y).

Hough transform is mainly about transforming pixels between x,y image space and another parameter space, called *Hough space*, where the two axes are not X and Y anymore. Transforming process can be done by converting, using a relational equation, pixels of an image on image space to pixels on *Hough space*. The transformation can reveal some interesting characteristics of the image on *Hough space* which easier for computing purpose.

To keep the idea simple, this thesis will mention only Hough linear transform which is the simplest case of Hough transforming used to detect line features in a digital image.

A straight line can be described as  $y = mx + b$  and is plotted for each pair of values (x,y). However, the characteristics of that straight line is not x or y, but its slope  $m$  and intercept  $b$ . Based on that fact, the straight line  $y = mx + b$  can be represented as a point (b, m) in  $b,m$  parameter space, where the two axes are  $b$  and  $m$ .

Representing a line by using (b,m) in the  $b,m$  parameter space could make application complicated since both parameters are unbounded; As lines get more and more vertical, the magnitudes of m and b grow towards infinity.

In the Hough linear transform, we represent a line using  $r$  and  $\theta$  ( $\theta$ ). The parameter  $r$  represents the smallest distance between the line and the origin, while  $\theta$  is the angle of the locus vector from the origin to this closest point. A straight line from the x,y image space can also be transform into a point on the Hough space by the following equation.

$$r = x.\cos\theta + y.\sin\theta \quad \text{equation - (1)}$$

Advantage of representing a line the Hough space over  $b,m$  parameter space is that both  $r$  and  $\theta$  are bounded;  $\theta$  can go as high as 360, while  $r$  can be as large as the size of image in the image space. So computation on these values is a lot easier compare to those in the other parameter space.

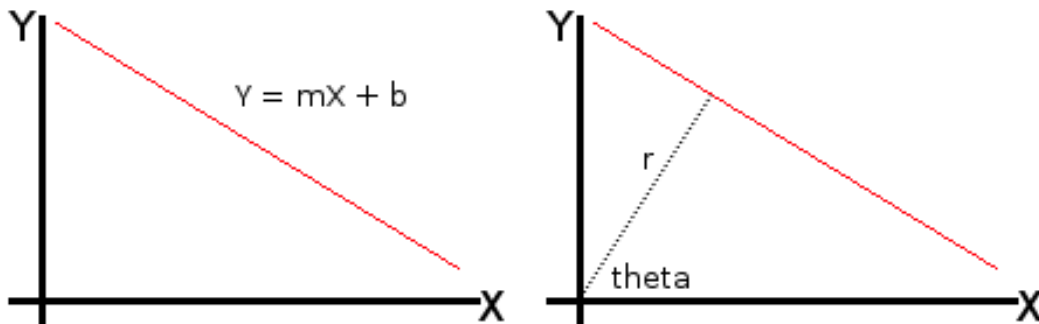


Figure 5: a line represent by  $m,b$  (left) or  $r,\theta$  (right)

Considered a point  $P$  on  $x,y$  image space, it is well known that an infinite number of lines  $(r, \theta)$  can go through the point, all at different angles. If we plot  $(r, \theta)$  of all those lines on Hough space, we will get a sinusoidal curve. This curve is actually a result of Hough linear transformation of a point from image space to Hough space.

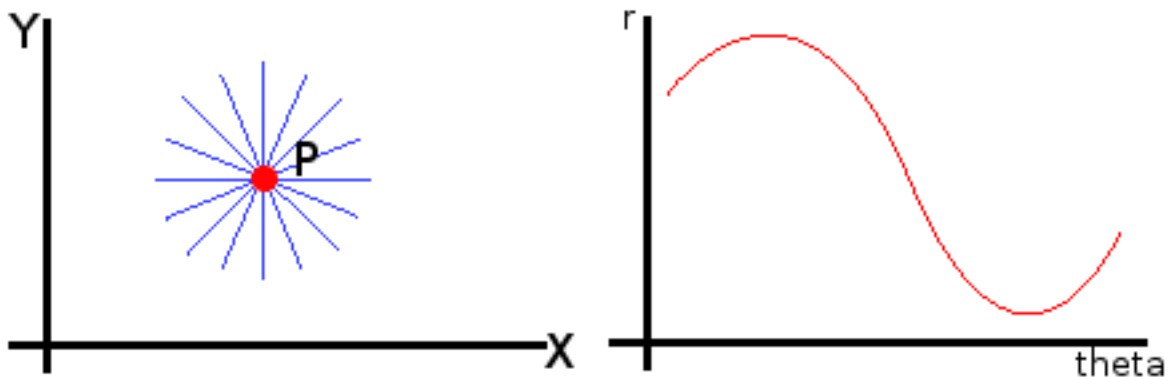


Figure 6: (Left) Number of lines go through a point on  $x,y$  image space. (Right) Plotting  $(r, \theta)$  of all lines on  $r, \theta$  parameter space

Supposed a digital image has 3 points ( $P, Q$ , and  $R$ ) lineup together in  $x,y$  image space. Using the Hough linear transform, we will have 3 curves on the Hough space as shown in the figure 7

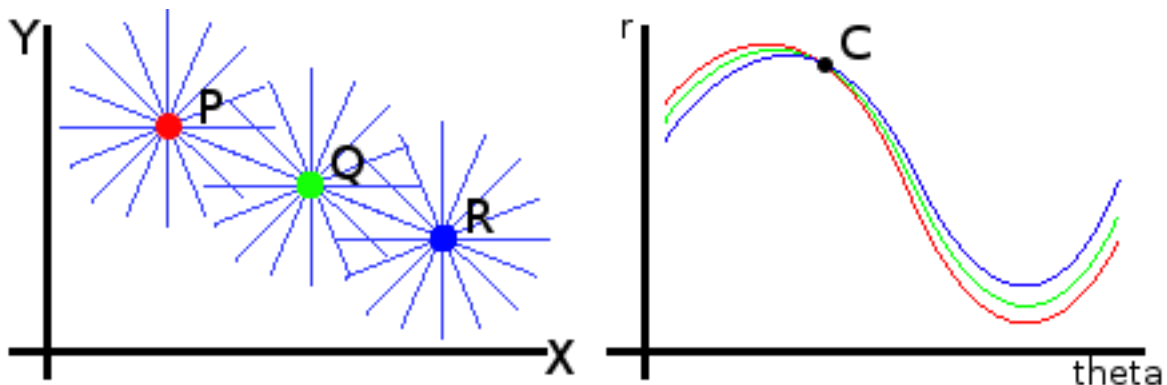


Figure 7: (Left) 3 points ( $P, Q, R$ ) line up on  $x,y$  image space. (Right) 3 sinusoidal curves plotted from lines that go through  $P, Q, R$  points.

The interesting characteristic of the image showed on the Hough space after transformation is at the point  $C(r_0, \theta_0)$  where all 3 sinusoidal curves intersect. These values  $r_0$  and  $\theta_0$  reflect the line on image space which bisects all 3 points being tested.

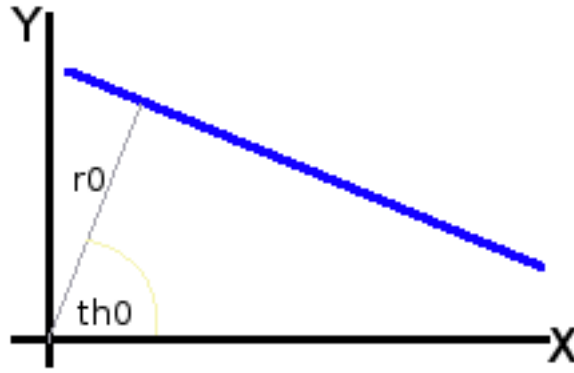


Figure 8: The line bisects all points  $(P, Q, R)$  generated from  $r0, theta0$  using Hough transform

By applying Hough linear transform to the data acquired by sonar sensors of the Pioneer robot, we can generate lines representing walls along the path that the robot has been through.

## **Soar**

Soar is an unified architecture for constructing general intelligent systems, and intended to support all the capabilities required of a general intelligent agent such as working on extremely difficult, open-ended problems, representing/using appropriate forms of knowledge, employing problem solving methods, interacting with outside world, and learning about all aspects of tasks.

Soar provides the fixed computational structures in which knowledge can be encoded and used to produce action in pursuit of goals. It differs from other programming languages in that it has an embedded specific theory of the appropriate primitives underlying reasoning, learning, planning, and other capabilities which assumed to be necessary for intelligent behavior.

### *Theory behind Soar*

Soar has been created in an attempt to define a unified theory of cognition. So, there is no theory behind, but Soar is a theory, a candidate unified theory of cognition.

Soar theory says that the following are characteristics of cognitive behavior [5].

- Goal-oriented.
- Take place in a rich, complex, detailed environment.
- Require a large amount of knowledge.
- Require the use of symbols and abstractions.
- Flexible, and be a function of the environment.
- Require learning from the environment and experience.

## *Rule-based System*

Rule-based system is a fairly simplistic “expert systems”, consists of rules, rule interpreter, and working memory.

Rule-based system works on a simple technique. It starts with rules, which contain all of the appropriate knowledge encoded into If-Then rules, and a working memory, which may or may not initially contain any data, assertions or initially known information.

Then, the system examines all the rule conditions (*left-hand-side*) and determines a subset, the conflict set, of the rules whose conditions are satisfied based on information in the working memory. Of this conflict set, one of those rules is triggered (fired). Which one is chosen is based on a conflict resolution strategy. When the rule is fired, any actions specified in its clause (*right-hand-side*) are carried out.

These actions can modify the working memory, the rule-base itself, or do just about anything else the system programmer decides to include. This loop of firing rules and performing actions continues until one of two conditions are met: there are no more rules whose conditions are satisfied or a rule is fired whose action specifies the program should terminate.

For example, a very simple rule-based system, written in plain English, to determine whether or not to bring out an umbrella. In working memory, information about today's weather forecast can be put in by external world interpreter. The rules in this system may look like

R1: IF *the forecast says it will be raining today*  
THEN *bring out an umbrella*

R2: IF *the forecast says it won't be raining today*  
THEN *do not bring out an umbrella*

## *Soar Basic Syntax*

In Soar, every rule starts with the symbol “*sp*”, which stands for “*Soar production*.” The remainder of the rule body is enclosed in curly braces: “{“ and “}”. The body consists of the rule name, followed by one or more conditions (the left-hand-side part), then the symbol “-->”, and then one or more actions (the right-hand-side part) [6]. Conditions and actions are enclosed in parenthesis: “(“ and “)”. Below is a template for rules:

```
sp {rule*name
    (condition)
    (condition)
    ...
-->
    (action)
    (action)
    ...
}
```

Following are the example rules written in Soar.

```
sp {R1
  (State <s> ^forecast-info RAIN)
-->
  (<s> ^output BRING-AN-UMBRELLA)
}
sp {R2
  (State <s> -^forecast-info RAIN)
-->
  (<s> ^output DO-NOT-BRING-AN-UMBRELLA)
}
```

*How does Soar work?*

Basic operations in Soar, without going into the details, are pretty similar to those of rule-based system:

- Examine the left-hand-side of the rules.
- Match the conditions to the information in working memory.
- Fire the action on the right-hand-side of the matched rules.

Unlike ordinary rule-based system, Soar is equipped with more complicated operations and conflict resolution strategy.

- *Operators*
- *Execution Cycle*
- *Impasses and Learning.*

## **Operators**

Operators are the decisions that Soar makes based on its rule firings. They answer the question, “What should I do next?” Operators may trigger an action internally or send a message to external software [7]. For example, in Soar-SLAM, operators do both tell the robot to perform an action such as move/turn/stop, and update/delete map information in Soar working memory.

## **Execution Cycle**

Soar execution cycle can be categorized into three phases, based on its handling of operators:

- Proposal phase
- Decision phase
- Application phase



### *Proposal Phase*

In this phase, Soar matches and fires all possible rules. It is possible that when one rule is fired, it will cause another rule to be fired. Soar will continue to match and fire rules until no more rule can be fired.

Rules are matched and fired (conceptually) simultaneously in Soar, even though only one rule can be fired at a time in actual execution. To maintain this concept, Soar keeps the working memory consistent. When a fired rule changes something working memory which cause the left-hand-side conditions of the earlier fired rule to be false, Soar retracts the action(s) of the false rule. So at the end of the phase, it seems like all rules have been fired at the same time.

This would be the major cause of infinite loop in Soar; if there is one (or more) rule that its action cause its condition to be false, then Soar will keep fire and retracts this rule forever.

For the proposal phase, here are two types of rules fired.

- Elaborations rules – rules that make changes in working memory, but don't propose operators.
- Proposal rules – rules that propose operators.

Proposal rules propose operators by putting an element, which has “^operator” as its attribute plus acceptable preference “+”, into the root of working memory (problem state).

```
sp {Propose*grab-umbrella
    (State <s> ^forecast-info RAIN)
-->
    (<s> ^operator <o> +)
    (<o> ^name grab-umbrella)
}
```

### *Decision Phase*

Because Soar matches and fires all possible rules in the proposal phase, it is possible that more than one proposal get fired . But only one of the operator proposals will be selected and applied in each cycle. This is why Soar developers created Decision phase.

In this phase, Soar makes a selection using conflict resolution strategy based on proposals' preferences which are specified at the time of proposing the operator or later on by other rules. Preferences can be either signs (>, <, =) or number or no preference.

### *Application Phase*

When Soar knows which operator proposal is the one to be applied, it adds an “^operator” attribute to the problem state. At this moment, another type of rule, called application rule, will be matched and fired. Below is an example of the application rule.

```
sp {Apply*grab-umbrella
    (State <s> ^operator <o>)
```

```

        (<o> ^name grab-umbrella)
    -->
        (write |Please take an umbrella|)
        (halt)
    }

```

Application rule is processed similar to the other rules in the way that it will be matched and fired if the left-hand-side conditions are true and will be retracted if the conditions are false. But the difference is the actions of application rules are persistent even if the rule has been retracted.

Same as the proposal phase, Soar will match and fire all possible rules until no more rule can be fired. Then it begins the proposal phase of the next cycle.

### **Impasses and Learning.**

As mentioned in decision phase, preferences can be given at the creation time or later by other rules. It is possible that the given preferences are incomplete or inconsistent. (The preferences can be incomplete in that no acceptable operators are suggested, or that there are insufficient preferences to distinguish among acceptable operators. The preferences can be inconsistent if, for example, operator A is preferred to operator B in one rule, but operator B is preferred to operator A in another rule.)

Soar calls this situation as an impasse. There are several impasse types, but all of them mean that Soar does not know what to do next. The consequence action of trying to resolve the impasse problem, Soar *subgoals*.

By subgoal, Soar create a substate, a new state and problem space with some information about the impasse, and begins its execution cycles on the substate. The main idea is that all rules, phases in this substate will lead to resolving the impasse in the superstate. Soar may create a hierarchy of superstates and substates in working memory if it hits impasses during execution in any substate.

In every state, there is a “^superstate” attribute which contain the name of the superstate. This allows programmer to keep track of what state they are working on. In the top most state value of “^superstate” attribute is “nil”,

One advantage of this subgoal using state/substate is that it provides a way of partitioning knowledge and limiting the operators to be considered in searching for a goal's desired state. We can compare this advantage to Object Oriented Programming. One state is one class, which has one main goal. During the execution of the class, we sometimes need to solve problems that belong to another class. So we subgoal to another class and return back with useful information. Then, we can continue working on the first class.

Soar's subgoal mechanism also makes it possible for Soar to learn. When an operator impasse is resolved, it means that Soar has, through problem solving, gained access to knowledge that was not readily available before. Therefore, when an impasse is resolved, Soar has an opportunity to learn, by summarizing and generalizing the processing in the substate.

Soar's leaning mechanism is called chunking; it attempts to create a new rule called a chunk. The conditions of the chunk are the elements of the state that (through some chain of rule firings) allowed the impasse to be resolved; the action of the rule is the working memory element or preference that resolved

the impasse (the result of the impasse). The conditions and actions are variable so that this new rule may match in a similar situation in the future and prevent an impasse from arising again [7].

In other words, Soar can learn by memorizing and summarizing left-hand-side the conditions and right-hand-side actions which occurred during impasses and subgoalings. As a result, it create a new rule which ready to be matched and fired. In the future, if the same situations occurred, instead of subgoalings, Soar can matches and fires this rule.

#### *I-Support/O-support working memory element.*

Soar's working memory can be separated into two part, i-support and o-support working memory. It resembles the humans' short-term and long-term memory respectively.

I-support working memory is the memory space that Soar allocates for temporary working memory elements which will be wiped out when the creator rules are retracted. This came from the idea that humans do not memorize every pieces of information. We just retrieve them, process them, make use from the results from them, and finally forget them. For example, if someone tells us a phone number, we then process them by phone the number, and we finally forget the number.

O-support working memory is another memory space allocated for long term working memory elements. This kind of the working memory elements will not be wiped out with the retraction of the creator rules. Take a look at the same example above, if we often call that phone number, after a few times we would memorize it into the long-term memory part of our brain. Next time, we can recall the number out right away without need to read it from somewhere else.

### **ADAPT project**

ADAPT (Adaptive Dynamics and Active Perception for Thought) is a project talking about a cognitive architecture, robot, and the real world. Its goal is to develop an architecture for a robot to provide it with the abilities of understanding, reasoning, and solving tasks at, or close to, human level. There are two main parts, a Soar cognitive architecture, and RS (Robot Schema) language [8], to be combined in ADAPT.

RS language is a formal language for representing networks of concurrent, real time schemas. A schema can be described as a plan or model of an aspect in the real world that contains necessary information such as facts, actions, connections to other components or schemas.

For instance, the schema for a wall would contain information that describe walls, such as size, location, color, texture. It would also connect them to some facts, such as walls are solid and can't get through, and to some actions that can be performed with walls, such as crashing with another object (schema). The wall schema would be connected to other schemas for floor, physic, and other related aspects.

The main advantage of Robotic Schema language is that each schema has an associated Port Automaton (a finite-state automaton equipped with a set of synchronous communication ports [8]) that define the semantics of the schema. ADAPT uses this advantage of RS language to create, for example, semantics for the use of natural language, and combined hierarchies of schemas together.

Despite the benefit from RS, by using only RS cannot create an architecture that posses cognitive

abilities as we want. Because RS cannot support dynamics, cognitive plausibility, and learning. This is where Soar come into part. Together with Soar [9], RS/Soar is now capable of both constructing schema network, reasoning, and learning.

ADAPT robot is trying to represent its understanding to the real work by creating a virtual copy of the world in its head. This virtual world is connected to and managed by RS/Soar which retrieves information of the real world through various kind of sensory, such as vision system [10] and sonars.

While the vision system is under construction, this thesis is an approach to provide ADAPT the ability of understanding and solving SLAM problem thorough information from sonar sensory.

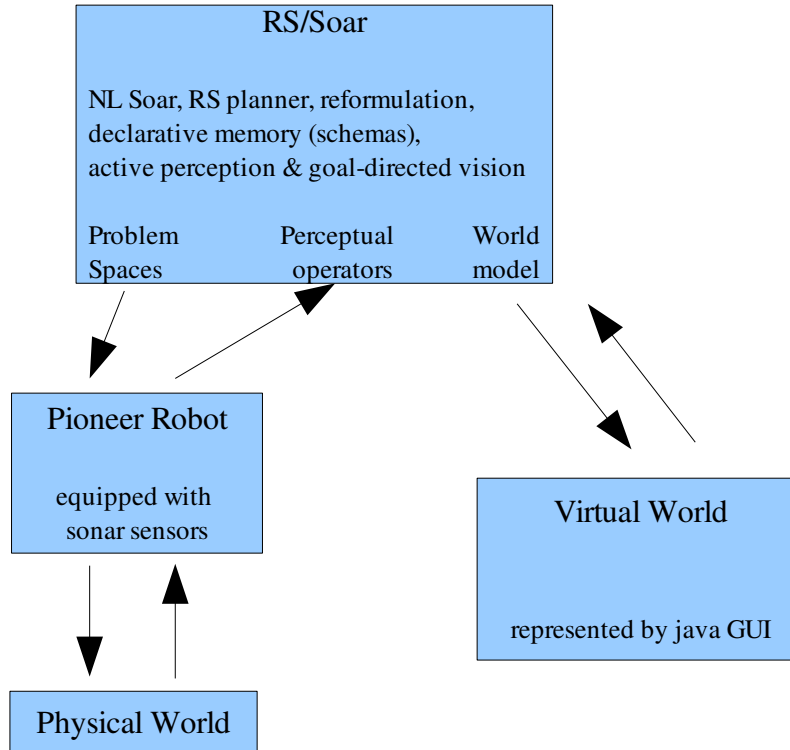


Figure 9: Soar-SLAM diagram, a part of ADAPT project

## PlayerStage

PlayerStage actually came from two parts of The Player Project [11], established since 2001 with the main goal of offering the free software tools for robot and sensor applications. The project consists of three main parts, Player, Stage, and Gazebo.

### Player

Player can be described as a cross-platform robot device interface and server. Although it was originally developed in order to support the Pioneer 2 robot [1], now Player provides support and control over various kinds of robots and sensors. It also provides a simple interface which accessible via TCP socket from client applications, written in any computer languages.

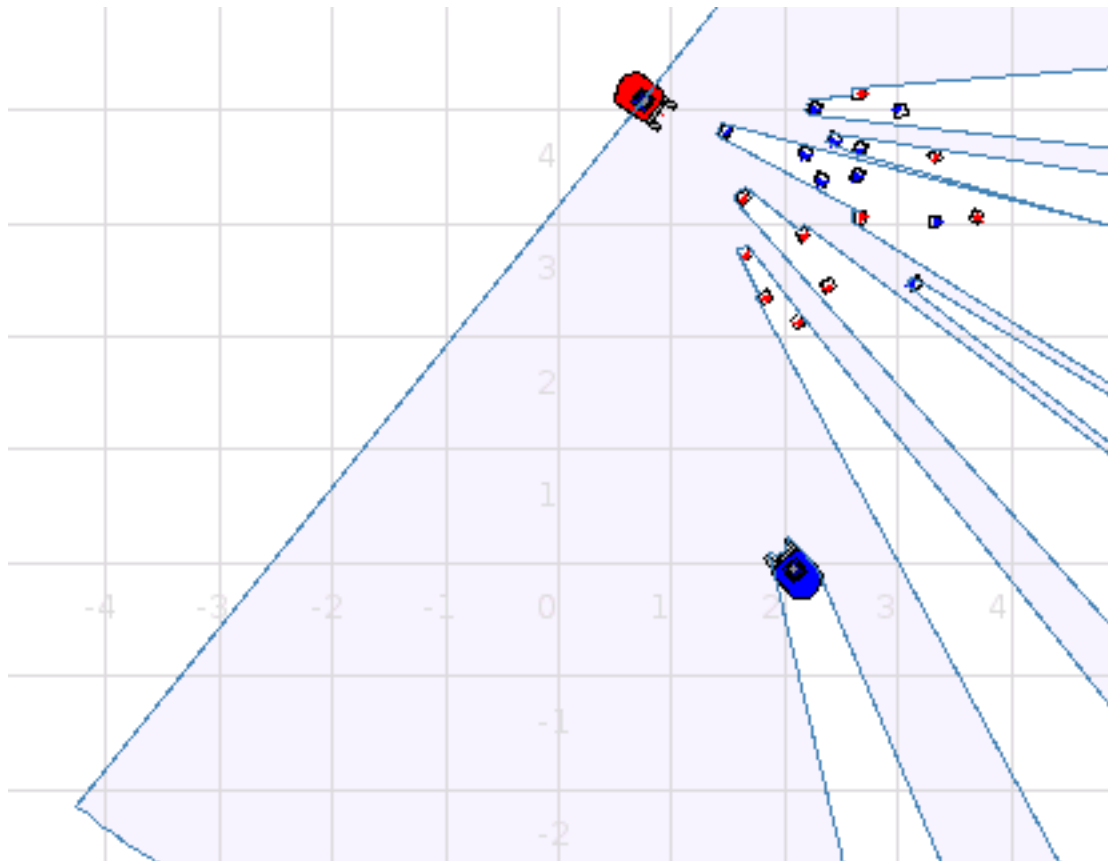
Player needs a configuration file to tell that which physical devices of the robot can be mapped to

which Player interfaces. Then, Player can instantiate the necessary drivers and gain control over the devices. Following is a simple example configuration file for Pioneer robot, equipped with a SICK LMS200 [12].

```
driver
(
  name "p2os"
  provides ["odometry::position2d:0"]
  port "/dev/ttyS0"
)
driver
(
  name "sicklms200"
  provides ["laser:0"]
  port "/dev/ttyS1")
```

### Stage

Stage is a two-dimensional robot simulator, capable of simulating a population of mobile robots, sensors and objects in a two-dimensional bitmapped environment. It also provides fairly simple, computationally cheap models for a number of devices [13].



*Figure 10: Stage screen shot with two Pioneer robots and several boxes*

Stage can be used, at least, in two different ways. As a C library, Stage can be taken care of as a library to provide robot simulation within other applications. It gives developers the ability to create their custom simulation models.

With Player, Stage can also be treated as a plugin module, known as *libstageplugin*. PlayerStage, *Player* with *libstageplugin*, provides virtual robots, equipped with virtual devices.

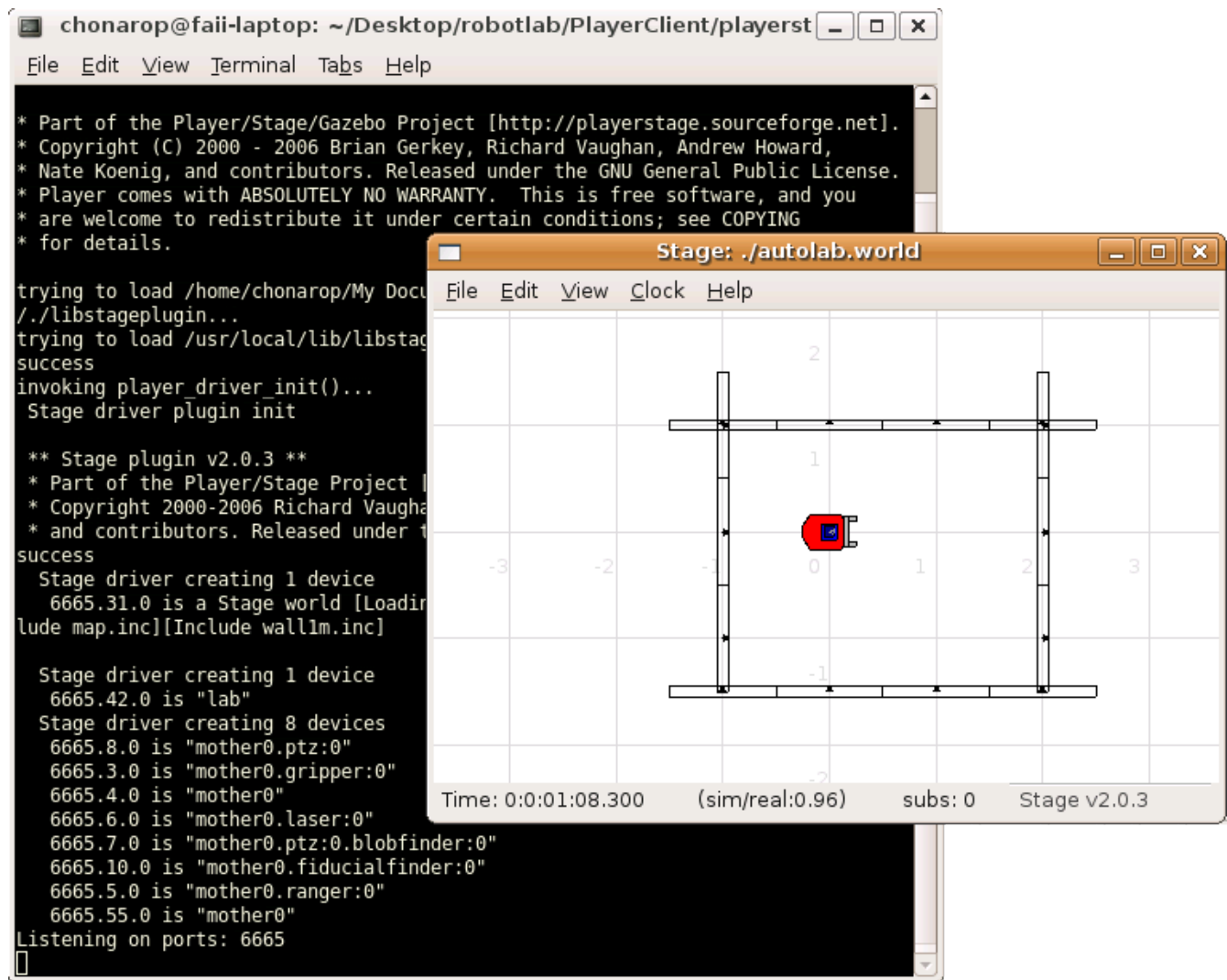


Figure 11: PlayerStage in a partition box to simulate the Pioneer 2 robot in the Pace Robot lab

### Java-Player

To use the PlayerStage, we must communicate via the TCP socket, which can be done through almost every computer programming languages. Java is one of the main computer languages that we use in many parts of ADAPT project. Fortunately, Java Player [14], another project related to the Player Project, offers “JavaClient” as a Java library that provides a number of necessary classes and functions to talk with PlayerStage through the socket.

### *Advantages of PlayerStage*

In research using a simulated robot is a very nice idea which also save a lot of time and cost. For instance, thousands of times the robot is needed to be moved and placed in different situations. Despite wasting time dragging the real robot from place to place, we can just drag the virtual robot in to the desired position in no time.

Further more, instead of finding and/or building a sophisticated real testing environment, we can create a virtual testing environment, by just editing the configuration file, with less cost and time.

Because PlayerStage is first created based on the Pioneer 2 robot, which is the same model that we are using, I believe that we will require less time and code modification when we migrate the software to the real Pioneer robot.

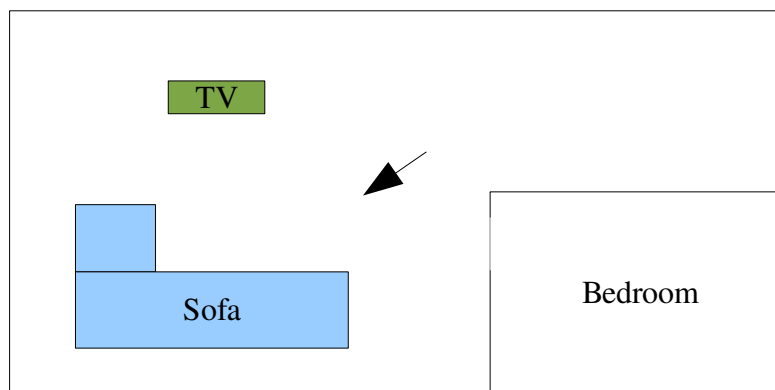
PlayerStage can also be configured to simulate error in robot's dead reckon system along the way robot running, just like the real Pioneer robot.

However, there are some concerns during the development. It turns out that either JavaClient or PlayerStage does not work well with too heavy stream of commands generated from complex schemas in Soar, while the real Pioneer robot seems to be fine.

## **Soar-SLAM**

### *Human with SLAM problem*

Imagine we are in a house, how can we know where we are? Based on information from our sensor, the eyes, we can tell that we are in a bedroom because we see the room and the bed, or in the living room because we see a sofa set and a TV. We know where we are by reasoning information of the environment with our understanding to the world.



*Figure 12: a floor plan example*

The figure 12 is an example floor plan of an apartment. The arrow represent a person stand facing the sofa. In this situation, we know that we are in the middle of the apartment, a sofa set is not so far in the front, and the bedroom is on our left. We don't know, and probably don't want to, that we are facing south/west (if north is up), 4 feet 7 inches from the sofa, and actually the bedroom is 98 degree to the left.

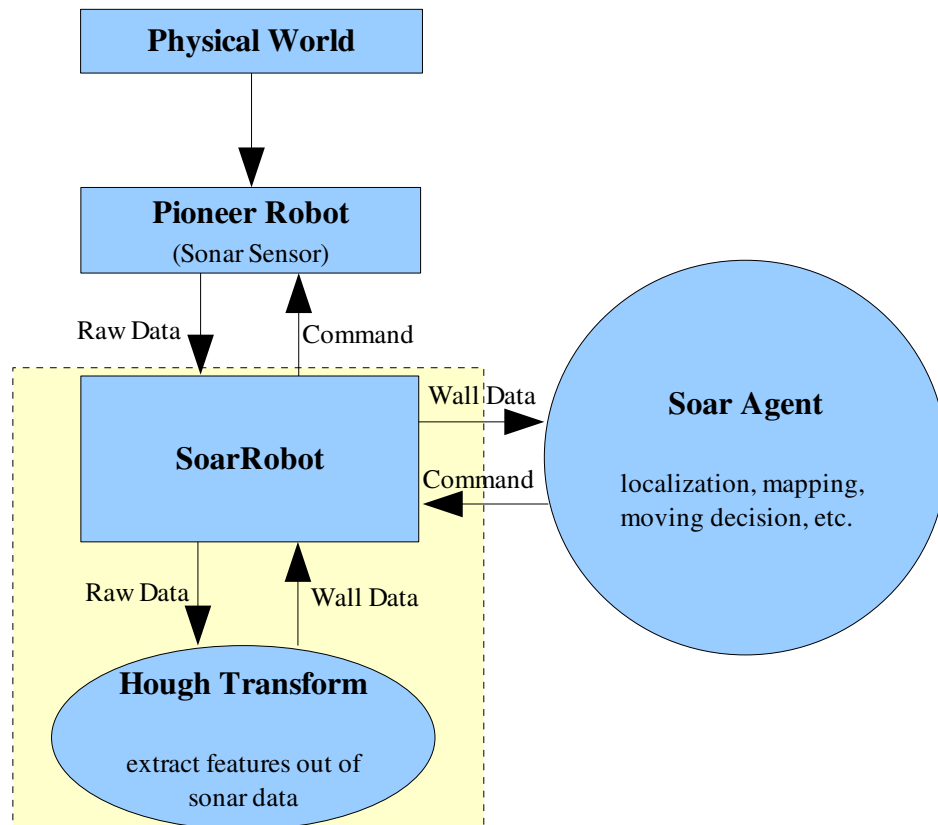
This information is too much in detail and not necessary for us to survive.

So the answer to the question of “how can human solve SLAM problem?” would be we roughly process overall information from our sensors. We process more on the detail only when its need. For example, most of us wouldn't know the exact size of our bedroom until we want to buy an a/c. We walk around the room without crashing on any objects by adjusting ourself based on estimated distance from our eyes.

### *Soar approach to SLAM problem*

Soar-SLAM is a new approach to resolve the SLAM problem using cognitive architecture such as Soar. As a cognitive theory, we are trying to solve the problem by imitate the way that human solving the SLAM problem.

As mentioned in the introduction part, SLAM problem involves two main processes, *localization* and *mapping*, at the same time. To be able to resolve the problem in human way, we create some hypothesis about how human resolve both problems.



*Figure 13: Soar-SLAM overall diagram*

Soar-SLAM can be separated into 3 major parts: the robot in physical world, the Java part, and the Soar part.

The Pioneer robot that we have is built to run in indoor environment with sonar sensors and a stereo camera. Although it is equipped with the stereo camera, the vision model for the robot is still in



under construction phase at the time this thesis is developing. So, I create this thesis based on sonar sensing information.

The Java part has two major roles. First of all, Java acts like central port for all other parts. (There will be more parts involved in the main ADAPT project.) Its main duty is to manipulate data from/to all links in the project such as put necessary data into Soar's working memory, read commands generated from Soar, feed raw data to the Hough transform, feed commands to the robot, read information from the robot.

Another role of Java part is do the complicated mathematical calculation. Because Soar has been developed in a specific way to help in cognitive researching purpose. It cannot easy do complex mathematical calculations such as matrix calculation or data conversion. To solve this problem, Soar provides a command for developers to call external Java (or C) functions to do the calculation for Soar.

Soar part plays the brain role. It is responsible for all tasks that involve complicated reasoning, learning, making decisions. For example, slow the robot down when it is heading toward the wall, back off when it get stuck, or move in to the zone that still need some more exploration.

### ***SoarRobot***

*SoarRobot*, coded in Java, is the central part of the project. Its main duty is to control overall processes, and to forward all data from/to three parts, the Pioneer robot (which could be substitute by PlayerStage simulator), Hough-Transform Java, and Soar.

The way *SoarRobot* works can be separated into two parts. The first part is initializing all the process:

- Create and initialize connection to the robot over TCP socket.
- Create Soar a kernel and agent.
- Load all rules in to Soar.
- Generate GUI for user.

When user click at the “Run” button, *SoarRobot* starts the next part which is looping through the following steps until the user trig a stop event:

- Read all information from the robot.
- Pass sonar information to the Hough-transform Java.
- Put all information, generated from the Hough-transform Java together with the one from robot, into input-link of Soar working memory.
- Let soar run for one cycle.
- Read command(s) from Soar's output-link.
- Process and forward command(s) to the robot.

### ***Hough-Transform Java***

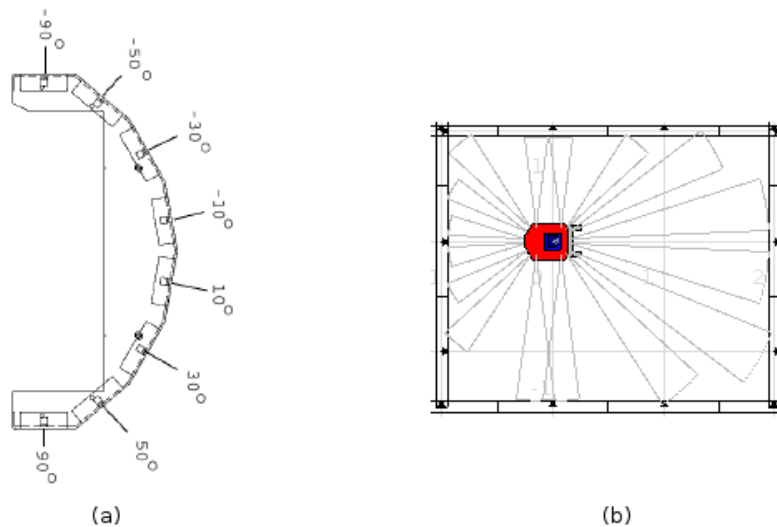
Hough-Transform Java functions as a wall extractor from the stream of sonar sensing information.

This part is implemented in Java because finding walls information from heavy stream of numbers requires lots of mathematical computation, and Soar is not good in this matter.

Basically four main steps are done in this part, data converting, noise filtering, Hough transforming, and wall extracting.

### *Sonar Data Converter*

The Hough transform receives input data as an array of x,y coordinates. But raw sonar data from the Pioneer robot is generated in arrays of numbers. Each number represents the distance from robot to the object in each directions. So before the data from robot can be processed in Hough Transform, it needs to be transformed from raw data.



*Figure 14: (a) positions and angles of transducers in the front of the robot, (b) robot placed in a box partition (in PlayerStage).*

For the situation as in the figure 14, returned sonar values from the Pioneer robot can be seen as an array of integers, for example, in this case (1215, 1510, 2249, 2464, 2464, 2741, 2426, 1965, 1965, 1863, 1395, 1239, 1239, 1395, 1510). Each integer correspond to the distance detected by each transducer. The converter will pair each sonar value with the angle of its transducer, then convert those pairs in to x,y coordinates on the image space. Figure 15 shows the coordinates that correspond to the example situations.

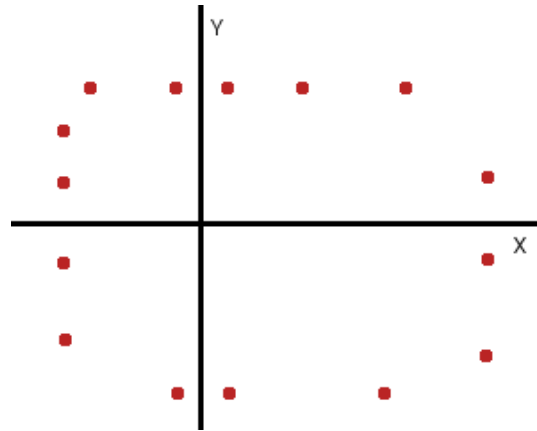


Figure 15: sonar data plot in x,y image space after converted

### Noise filter

Illustrate in the figure 16, dots represent sonar returns from letting robot run for a while. Compare to the physical environment (represented by the dotted lines), we can obviously see a number of noise returns, which are erroneous and should be ignored. There are various possible causes of the error, such as moving people, sonar cross-talk, and echoes. Sonar signals are analogous to the ripples on a pond. Ripples can interfere with each other and bounce back and forth off many objects, generating more ripples. A sonar echo can bounce off several objects and come back to the robot from a completely different angle, and delayed a significant time. This makes it look to the robot as though there's an object in that direction that is very far away.

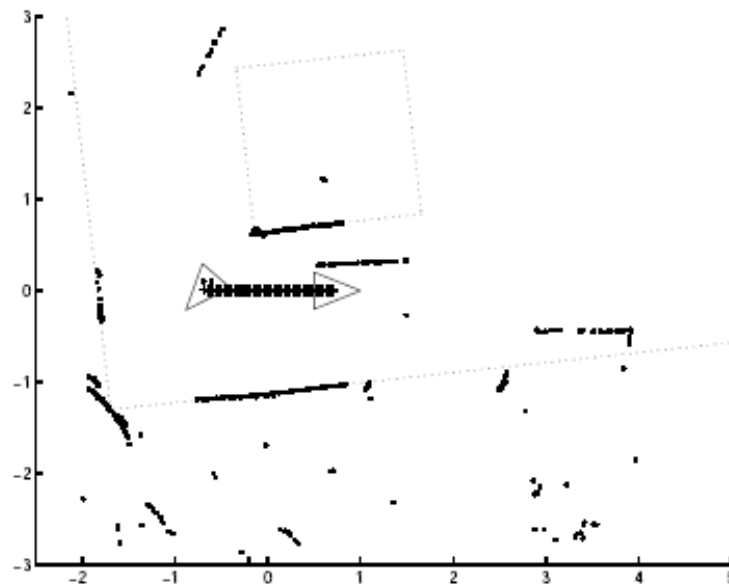


Figure 16, [3]: Sonar data returns observed from letting robot move along the path in an indoor environment.  
(dotted line represent real environment)

To filter out such noise data, we can take advantage of the characteristic of sonar sensing. Each sonar data has its own visible line, the line represented possible location of the detected object. Visible line can be calculated from the visibility angle of sonar ( $\beta$ ) and the distance from robot to the object.

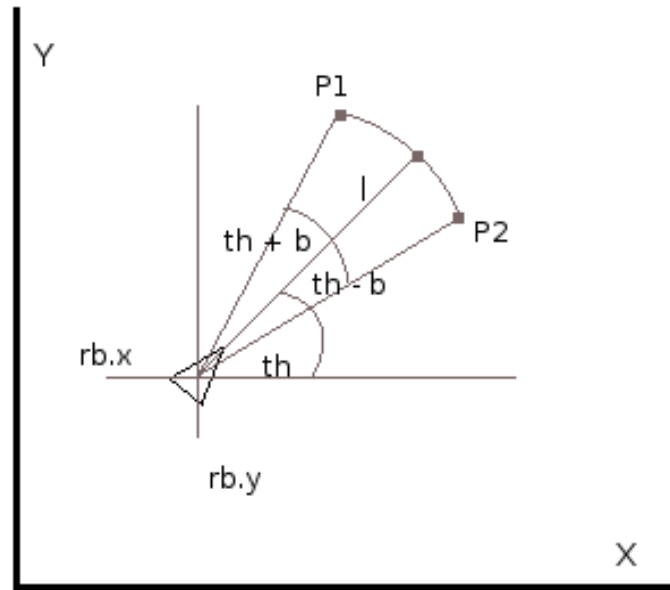


Figure 17: How to create visible line

An example sudo code to calculate visible line can be found below:

$\beta$  = visibility angle of sonar  
 $x,y$  = the target coordinate  
 $l$  = length from the sensor (robot) to the target coordinate  
 $theta$  = the response angle  
 $rb(x,y)$  = position of the robot when retrieving the target coordinate

Point  $p1$  = curve starting point

$$p1.x = l * \cos (theta - \beta/2)$$

$$p1.y = l * \sin (theta - \beta/2)$$

Point  $p2$  = curve ending point

$$p2.x = l * \cos (theta + \beta/2)$$

$$p2.y = l * \sin (theta + \beta/2)$$

for  $i = p1.x$  to  $p2.x$

$$xx = i$$

$$yy = \sqrt{l^2 - y^2}$$

add new Point of  $(xx + rb.x, yy + rb.y)$  to Point array

return point array as a line.

If we apply *visible lines* to every returns, as shown in figure 18, we can also see that the returns from real features form solid lines or sections, but false returns do not.

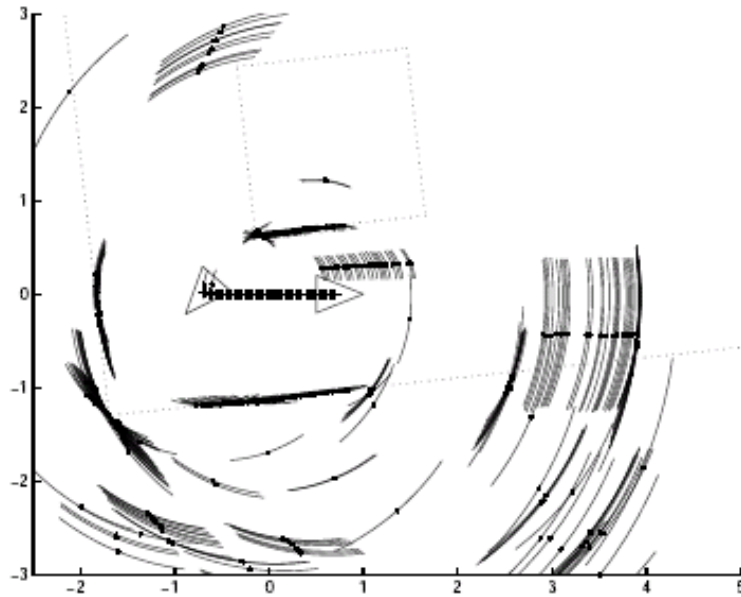


Figure 18, [3]: Sonar data plotted together with visible lines.

The noise filter works by plotting all these sonar dots and lines on a *Vote Space*, the space that each coordinates (x,y) contains number of vote. Then select only dots fall into the region which has vote number more than some specific  $k$  value. The result is illustrate below (figure 19).

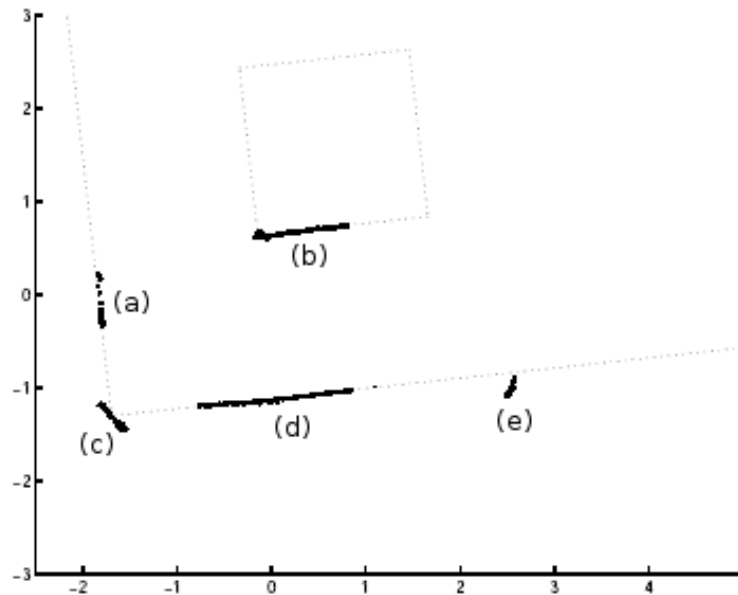


Figure 19: Sonar data after run through the noise filter

Notice that this filter cannot filter out 100% of noise. (As shown in figure 19, The false return in zone (e) still pass through the filter.) But it is relatively good enough for us. We can act with the error by either adjusting the value  $k$  in the filter or simply ignoring and let Soar handling it.

### *Hough transforming*

After pass the raw robot data through *sonar data converter* and *noise filter*, now we have sonar data in the form of pixels on x,y image space with noise reduced. Next step is to transform all those data into Hough space.

Every single pixel in sonar data will be transformed into a correspond sinusoidal curve by using the equation (1).

$$r = x \cdot \cos(\theta) + y \cdot \sin(\theta)$$

where  $x, y$  is the coordinate of the pixel  
 $\theta$  varies from 0 to 359.

An example sudo code of the transforming algorithm is as below:

```

for  $i = 0$  to number of pixels in the images
    for  $j = 0$  to 360
         $x = \text{retrieve } x \text{ from pixel}[i]$ 
         $y = \text{retrieve } y \text{ from pixel}[i]$ 
         $\theta = j$ 
         $r = x \cdot \cos(\theta) + y \cdot \sin(\theta)$ 
        *** vote  $r, \theta$  on hough space ***

```

As mentioned in the introduction, we are interesting in the points, in Hough space, where lots of sinusoidal curves intersect each other. In the real implementation, Hough space is not only just a plain  $r, \theta$  space, but it also acts like a *vote space*, which every single pixel contain a vote number together with its coordinate.

To vote pixels into Hough space, simply find the coordinate corresponds to the pixel, and then increase its value by one. Shown in the figure 20 is all sinusoidal curves transformed from the example pixels in the figure 19.

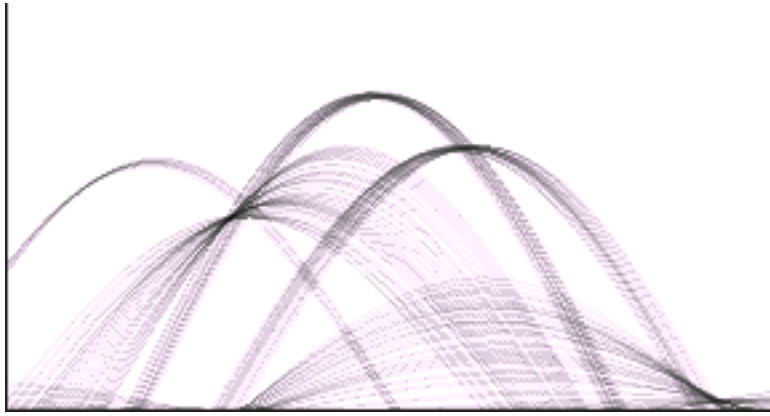


Figure 20: Hough space voted with example sonar data (the higher vote number, the darker spot displayed)

### Extracting Wall

The last step in this part is to generate wall elements for further use in Soar part. Each Wall element carries the following values:  $r$ ,  $\theta$ , start point, end point, robot position  $(x, y)$ , time of creation.

- $r$ ,  $\theta$  represent alignment of the wall referenced from the global map.
- start and end point specify the exact location the the wall occupying.
- robot position tells the position of the Pioneer robot when the wall is created.
- time of creation for reference which wall come first.

A static value  $k$  is defined to be the critical point. This  $k$  should be tested and adjusted to match the sampling data from testing environment in the way that it can reflect most of the wall information, and hide out most of the false information.

In the voted Hough space, the wall extractor will scan through and locate zone(s) which the voting numbers are higher than critical point  $k$ . Using the previous example of Hough space, there are five zones that voting numbers are higher than  $k$  as shown in figure 21.

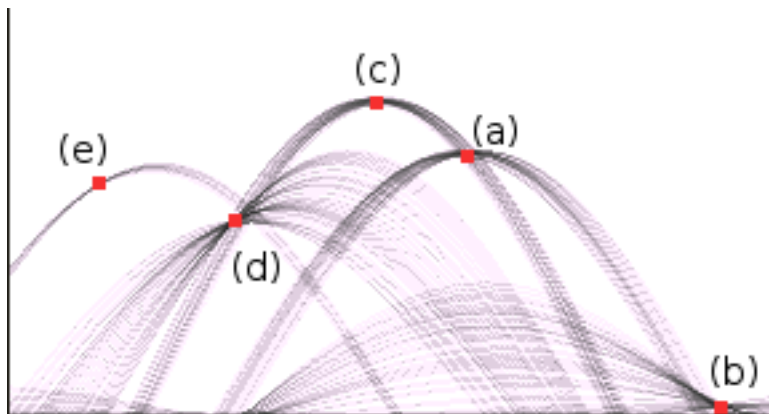


Figure 21: Zones on the Hough space that have voting numbers are higher than  $k$

Find a single dot in the middle of each zone, we will have five pairs of  $(r, \theta)$  values. Then if we create lines on regular image space correspond to these  $(r, \theta)$ , it will look like figure 22 below.

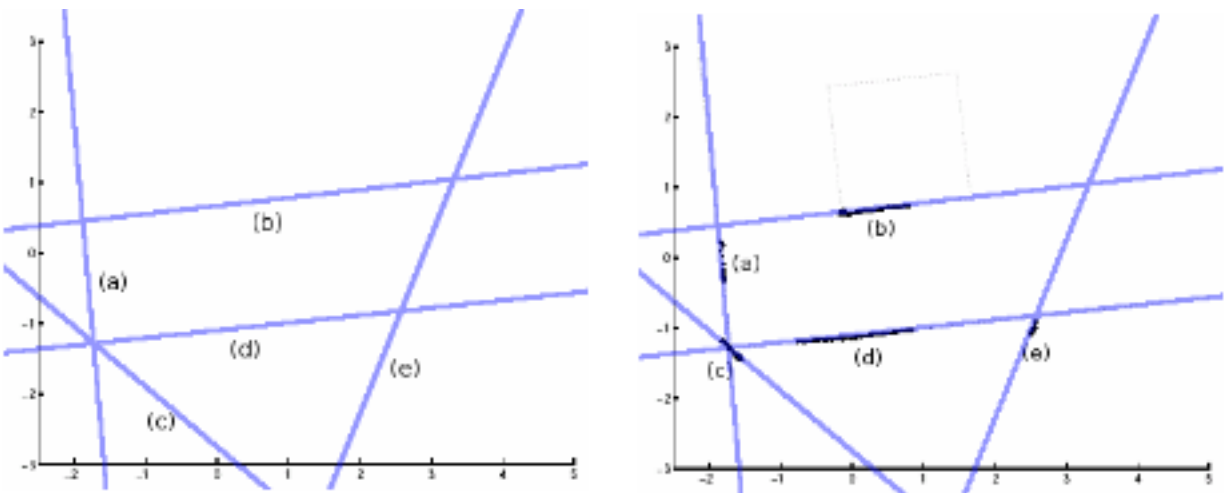


Figure 22: (left) Lines correspond to  $r, \theta$  extracted from Hough space (right) generated lines and Sonar pixels displayed together.

The very last thing to be done is find the beginning and ending points for all lines in figure 22 (right), pack them together with all other information as Wall elements, and feed them into the input-link of Soar working memory.

### **Synchronization between external world and Soar**

To be able to connect Soar with the outside world, Soar developers designed a working memory element called “io” (input/output) as a port connected to outside world. We can look at the “io” as a port in personal computer. A USB port, for example, is one of the way computer used to communicate many external devices, such as keyboard, mouse, printer, etc.

“io” actually carries another two working memory element, “input-link” and “output-link” under itself. Obviously, *input-link* is for receiving all input, and *output-link* is for sending all output. To make the world easier, in Soar 8, Soar developers also provide a Soar interface called “*Soar Markup Language:SML*” for us to easily communicate with Soar.

SML has been designed to support multiple languages, C++, Java, and TCL. Not only provides methods to read/write the Soar I/O, it also allows us to take more advantages from Soar. For instance, by supporting multiple clients connection to a single Soar kernel and embedded Soar kernel under other applications, we can create multiple agents environment and let multiple Soar kernels do the work simultaneously.

Further more, SML also supports universal access to Soar command line which mean we gain more control. With the command line we do various things, such as turn the learning mode on/off during the middle of execution, or reading specific working memory elements.



Another interesting feature, is user define function supported. Because Soar is not good in mathematic calculation, for instance, many rules will be needed just to find a distance from a point to a line. So we can let Java do the math for Soar by creating a user define function in Java, and call it from Soar.

## **Soar Rules**

### *Working Memory Structure*

Information in Soar, beside rule sets, is represented in the form of elements in working memory. Every single element has attribute and value, represented in the form of “^attribute value”. If any element has child element, the child element will be listed indent to itself. For example.

```
^parent P1
  ^child1 C1
    ^child_level2 C3
  ^child2 C2
```

There are two main structures needed to be mentioned in this section. The first one is structures of robot data where all information about current robot data can be found.

```
^io
  ^input-link
    ^currentrobotdata
      ^Th [float] #angle relative to the coordinate system
      ^Vel [float] #current velocity
      ^X [integer] #current X coordinate
      ^Y [integer] #current Y coordinate
      ^sonar
        ^number [0..15] #show which sonar provides the information
        ^reading [integer] #value returned by the sonar
```

*Remark 1:* There are 16 ^sonar elements for the Pioneer 2 robot, and 7 ^sonar elements for the Pioneer 1 robot.

*Remark 2:* Some more information, which is not in use in this thesis but it could be useful in the future development, is also provided here, eg. ^MoveDoneDist, ^RotVel, ^LeftVel, ^RightVel, ^IsLeftMotorStalled, ^IsRightMotorStalled, etc. which contain information about the rotational velocity, the velocity of the left and right wheels, whether the motors are stalled, etc.

Another important WME structure is the wallData structure. All information about wall, representing the virtual world that Soar knows can be found here.

```
^wallData
  ^Wall
```

```

^Start          #point where the line start
  ^X [integer]
  ^Y [integer]
^End            #point where the line end
  ^X [integer]
  ^Y [integer]
^Ref            #location of the robot when the wall was generate
  ^X [integer]
  ^Y [integer]
  ^Th [0..359]
^Length [float] #length of the wall
^R [integer]    #distance from the origin (0,0)
^Th [0..359]    #angel relate to Y axis
^Time [integer] #time of generation

** There is only one ^wallData in the state, all ^Wall element are linked to the
^wallData element

```

#### *robotagent.soar*

“*robotagent.soar*” is the very first soar file, hard coded in SoarRobot, to be load into Soar working memory. It provides us the flexibility to add/edit/delete links to other Soar files without editing and recompiling the Java source code.

Necessary initializing Soar commands can be found here [6]:

- **max-chunks** <number> , **max-elaborations** <number> - limit the maximum number of chunks and elaborations rules that may be create during a decision cycle. This limits are included in Soar to prevent getting stuck in an infinite loop during the processes .
- **pushd** <directory> – push the specified directory into the directory stack, and change into it.
- **popd** – pop current directory off the stack, and change to the next directory on top of the stack.
- **learn** <parameter> – set parameters for Soar learning mechanism; eg *-off* to turn off the learning, *-on* to turn it on.
- **source** <filename> - load and evaluate the contents of a file.

In Soar-SLAM *robotagent.soar* basically loads Soar files from two rule sets: Moving rule sets and SLAM rule sets.

### *slam.soar*

“*slam.soar*” is the first and main rule set of Soar-SLAM rules. Actually, It acts just like *robotagent.soar*, but has been nailed to control over only SLAM part. By having the *slam.soar*, developer can turn on/off SLAM feature without touching other schemas by adding or removing the line “*source slam.soar*” in *robotagent.soar*.

### *Elaborate.soar*

The rules in *Elaborate.soar* work together in elaborating necessary information, further than the one provided in input-link. The first kind of elaboration is creating constants such as acceptable distance and angle between two walls.

#### **English explanation:**

```
if there's a state and
    that state has an attribute “^superstate”
then attach all these information under that state
```

#### **Soar Rule:**

```
sp {slam*elaborate*constant
    (state <s> ^superstate <ss>)
-->
(<s> ^locali-const <c>)
(<c> ^r-range 400
    ^start-end-range 300
    ^th-range 20
    ^overlap-percent 0.40)
}
```

**Remark:** There is only one condition in the left-hand-side of the rule. This rule will be fired once when a state is created, and will be retracted only with the state. Because every state in Soar must have the “^superstate” attribute. As a result, all the information elements are stucked to the state all the time, even though it is an i-support rule.

Another kind of elaboration in this part is existed because of the lack of mathematics capability of Soar. No mathematical calculation is allowed on the *left-hand-side* of the rule due to efficiency matter. Anyway, mathematical calculations on the *right-hand-side* part are allowed. In regular programming language, for instance, we can simply say “*if* ( $i > i+3$ ) *then do something*”, but it is omitted in Soar.

So when I want to match two walls that are near each other. I cannot say

```
If (location of wall-1 - location of wall-2) < k
then do something
```

Instead I have to create two rules:

*elaborate rule:*

*If there are wall-1 and wall-2  
then find the distance between wall-1 and wall2*

*Matching rule:*

*If there are distance between two wall that less than k  
then do something.*

The second type of elaboration rules in “*Elaborate.soar*” is explained and shown below.

**English explanation:**

*If there are two walls and a comparison table  
then calculate distance and angle between the walls  
and attach the result into the table*

**Soar rule:**

```
sp {slam*create*wall*theta*comparison*table
  (state <s> ^superstate nil
    ^wallData( ^Wall <w1>
               ^Wall {<w2> <> <w1>})
    ^theta-comp-table <t>)
  (<w1> ^Th <th1> ^R <r1>)
  (<w2> ^Th {<th2> >= <th1>} ^R <r2>)}
-->
(<t> ^couple <c>)
(<c> ^Wall <w1>
    ^Wall <w2>
    ^th-diff (int (exec anglediff <th1> |,| <th2>))
    ^r-diff (abs (- <r1> <r2>)))}
```

### *CopyWall.soar*

Wall elements generated by Java Hough transform are passed through Soar via “*input-link*”. But all data under *input-link* is considered to be **read-only** element type. Rules in *CopyWall.soar* are together to copy all Wall data from *input-link* to the appropriate place, *^wallData*.

To make Wall elements stay permanently in Soar working memory, using rules that proposing and applying operators are required. Below are two rules that act as Wall elements copier in Soar-SLAM.

**Proposal rule:**

**English explanation:**

*If there's a wall element in ^io.input-link.wallData which has*

no “^Copied true” element

**Then** propose an “CopyWall” operator to copy all information

**Soar Rule:**

```
sp {slam*propose*copy*wall*from*input-link
  (state <s> ^superstate nil
    ^io.input-link.wallData <source>
    ^wallData <dest>)
  (<source> ^Wall <w>)
  (<w> ^Start (^X <sx> ^Y <sy>)
    ^End (^X <ex> ^Y <ey>)
    ^Ref (^X <rx> ^Y <ry> ^Th <rth>)
    ^R <r>
    ^Th <th>
    ^Length <l>
    ^Time <t>
    -^Copied true)
-->
```

```
(<s> ^operator <o> + =)
(<o> ^name CopyWall
  ^Wall <w>
  ^Start <start>
  ^End <end>
  ^Ref <ref>
  ^R <r>
  ^Th <th>
  ^Length <l>
  ^Time <t>)
(<start> ^X <sx> ^Y <sy>)
(<end> ^X <ex> ^Y <ey>)
(<ref> ^X <rx> ^Y <ry> ^Th <rth>)}
```

**Application Rule:**

**English explanation:**

*If there is an “CopyWall” operator*

**Then** copy the wall along with its information into the right place, and mark “^Copied” flag to the source element.

**Soar Rule:**

```

sp {slam*apply*copy*wall*from*input-link
  (state <s> ^superstate <ss>
    ^operator <o>
    ^io.input-link.wallData.Wall <w>
    ^wallData <dest>)
  (<o> ^name CopyWall
    ^Wall <w>
    ^Start (^X <sx> ^Y <sy>)
    ^End (^X <ex> ^Y <ey>)
    ^Ref (^X <rx> ^Y <ry> ^Th <rth>)
    ^R <r>
    ^Th <th>
    ^Length <l>
    ^Time <t>)
  -->
  (<dest> ^Wall <ww>)
  (<ww> ^Start <start> ^End <end> ^Ref <ref>
    ^R <r> ^Th <th> ^Length <l> ^Time <t>)
  (<start> ^X <sx> ^Y <sy>)
  (<end> ^X <ex> ^Y <ey>)
  (<ref> ^X <rx> ^Y <ry> ^Th <rth>)
  (<w> ^Copied true))}

```

**Remark:** Even though the “input-link” is for read only purpose, we can still create such a ^copied element for our convenient. This element won't be able to see by external agent. And trying to modify elements under input-link would crash the Soar kernel.

The condition “-^Copied true” in the proposal rule serve indirect purpose here. Normally, we don't need this type of condition to prevent the rule matching on the same Wall more than one time. By Soar nature, even without the condition, it will match this rule with each Wall only once at the time of creation.

After the moment that the operator from this rule has been selected and applied, it will mark an element “^Copied true” to the Wall element. Then, right after that moment, the proposal rule is retracted because the conditions is no longer true. As a result, applied “CopyWall” operator is also retracted. Then, Soar can continue match and apply other operators.

### *CheckWallRelation.soar*

The rule set in *CheckWallRelation.soar* is part of the mapping process in Soar-SLAM. Soar-SLAM actually does not try to draw the map directly from the input information, instead, it corrects the

created map from Java Hough Transform by detecting and correcting false information in the map.

To illustrate the idea of how does the process work, let figure 23 below are examples of wall elements detected after letting the robot run in a box partition for a while.

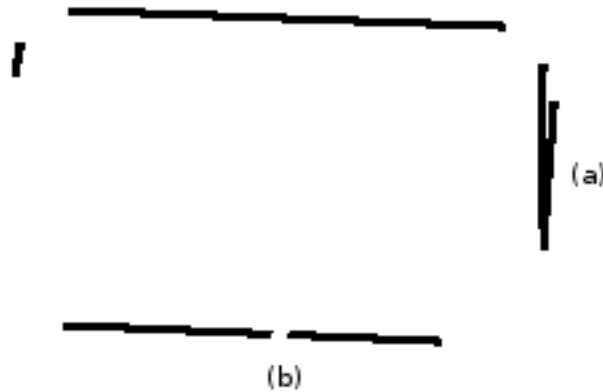


Figure 23: (a) 2 walls line near each other, (b) 2 walls line next to each other

In figure 23 (a), if we are correcting the map by ourself, we would know that these two walls are actually generated from the same physical wall. Then, We would correct the point (a) by probably draw a line somewhere in the middle and get rid of the two walls.

In figure 23 (b), we can also see that these two walls would be the same physical wall, Then we can correct the error by simply merge both near ends together.

Another interesting point is that if we take a good look at both couples (a) and (b), the two walls in both couples hold very close  $r$  and  $\theta$ . ( $r$  is perpendicular distance from the origin to the line,  $\theta$  is the angle between the line and Y axis.) This is how we can define the term “near” between to line in Soar.

Below is the rule that starts investigation on the two lines. (Note that some information is calculated by rules in *Elaborate.soar*)

**English explanation:**

*If  $r$  and  $\theta$  of any two walls are closer than some specific values in elaborated constants*

*then propose a “CheckWallRelation” operator.*

**Soar Rule:**

```
sp {slam*check*wall*relation
  (state <s> ^superstate nil
    ^wallData <wd>
    ^theta-comp-table <t>
    ^locali-const (^th-range <thr>
      ^r-range <rr>))
```

```

(<wd> ^Wall <w1> {<w2> <> <w1>})
(<t> ^couple (^Wall <w1>
              ^Wall <w2>
              ^th-diff {<th> < <thr>}
              ^r-diff {<r> < <rr>}}))
(<w1> ^Length {> 0} ^Th <th1>)
(<w2> ^Length {> 0} ^Th {<th2> >= <th1>})
-->
(<s> ^operator <o> + =)
(<o> ^name CheckWallRelation
    ^Wall <w1> <w2>)}

```

I intentionally did not create the application rule for the above proposal rule, I am using the Soar SubGoal feature to resemble the way human approach the problem. First, we see if two lines are close or related to each other. Then we investigate how they are related, and what the answer or resolution of this relation would be.

Following is the Soar version of investigation. First the proposal rule matches and causes operator no-change impasse. This impasse then lead Soar into a sub-state, “*CheckWallRelation State*”, where the sub-goal is to find the actual relation between the two walls specified by the operator.

Processes in the *CheckWalRelation* sub-state can be explain below:

- Find all necessary information such as:
  - How far are the edges?
  - What is the different between the lengths?
  - Are those two overlap each other like figure 23 (a) or line next to each other like figure 23 (b)?
- Reasoning relation between the two walls as the following:

Hypothesis 1:

*If the two walls were created at the **same time**, and they overlap each other more than  $x$  percent of the length, then both of them were generated from the same physical wall. So propose an operator to correct this error.*

Hypothesis 2:

*If one wall were created after another, and they overlap each other more than  $x$  percent of the length. Then both of them were generated from the same physical wall. We also assume that at the moment when the second wall were generated, the robot already lost its track. So we propose an operator to update the position of both wall and robot to the right place.*



The first hypothesis acts only as a map correction, a part of map building while the second hypothesis acts as both map building and localizing because it both correct the map and locate the position of the robot. Proposal and application rules below are the implementation of both hypothesis.

#### **The Soar rule for Hypothesis 1:**

##### **- Proposal rule**

##### **English explanation:**

*If the two walls overlap each other more than some value  
and both walls were create at the same time <t>  
then propose an "UpdateWallInfo" opeartor with new wall  
specification embedded.*

##### **Soar Rule:**

```
sp {slam*CheckWallRelation*2*Propose*wall*info*3
  (state <s> ^name CheckWallRelation
    ^overlap-factor <olf>
    ^overlap {<ol> > <olf>}
    ^target-wall <w1> {<w2> <> <w1>})
  (<w1> ^Length <l1> ^Time <t> ^R <r1> ^Th <th1> ^Length <l>
    ^Start (^X <start1X> ^Y <start1Y>)
    ^End (^X <end1X> ^Y <end1Y>)
    ^Ref (^X <refX> ^Y <refY> ^Th <refTh>))
  # notice that I use <t> below, instead of <t2>
  # means that ^Time of <w2> must equal to ^Time of <w1> above
  (<w2> ^Length <l2> ^Time <t> ^R <r2> ^Th {<th2> >= <th1>}
    ^Start (^X <start2X> ^Y <start2Y>)
    ^End (^X <end2X> ^Y <end2Y>))
  -->
  (<s> ^operator <oo> + =)
  (<oo> ^name UpdateWallInfo
    ^Wall1 <w1> ^Wall2 <w2>
    ^R (/ (+ <r1> <r2>) 2)
    ^Th (/ (+ <th1> <th2>) 2)
    ^Length <l> ^Time <t> ^Ref <ref> ^Start <st> ^End <ed>)
    # new start point
    (<st> ^X (int (/ (+ <start1X> <start2X>) 2))
      ^Y (int (/ (+ <start1Y> <start2Y>) 2)))
```

```

# new end point
(<ed>   ^X (int (/ (+ <end1X> <end2X>) 2))
        ^Y (int (/ (+ <end1Y> <end2Y>) 2)))
(<ref>  ^X <refX>
        ^Y <refY>
        ^Th <refTh>))}

```

**- Application rule:**

**English explanation:**

*If there is an "UpdateWallInfo" opearator  
then create a new wall based on information in the operator  
and also delete the two walls.*

**Soar Rule:**

```

sp {slam*CheckWallRelation*Condition1*Apply*update*wall*info
  (state <s>   ^name CheckWallRelation
              ^operator <o>))
  # locate the ^wallData node
  (state <sss> ^superstate nil ^wallData <wd>))
  (<o> ^name UpdateWallInfo
      ^Wall1 <w1> ^Wall2 <w2> ^R <new-r> ^Th <new-th> ^Time <t>
      ^Start (^X <new-sx> ^Y <new-sy>)
      ^End (^X <new-ex> ^Y <new-ey>)
      ^Ref (^X <refX> ^Y <refY> ^Th <refTh>))
  (<wd> ^Wall <w1> <w2>))
  (<w1> ^R <r> ^Th <th> ^Start <ws> ^End <we>))
  (<ws> ^X <sx> ^Y <sy>))
  (<we> ^X <ex> ^Y <ey>))
-->
  (<wd> ^Wall <ww>))
  (<ww> ^R <new-r>
        ^Th <new-th>
        #new length, using user defined function "getlength"
        ^Length (int (exec getlength <new-sx>|,|<new-sy>|,|<new-ex>|,|
<new-ey>))
        ^Time <t> ^Ref <ref> ^Start <start> ^End <end>))
  (<start> ^X <new-sx> ^Y <new-sy>))
  (<end> ^X <new-ex> ^Y <new-ey>))

```

```

(<ref> ^X <refX> ^Y <refY> ^Th <refTh>)
# delete both Walls <w1> <w2>
(<wd> ^Wall <w1> - <w2> -)}

```

#### **The Soar rule for Hypothesis 2:**

##### **- Proposal rule**

##### **English explanation:**

*If the two wall overlap each other more than some value  
but they hold different time of creatation  
then propose an "ChangeRobotPosition" operator with information  
about how far are those two*

##### **Soar Rule:**

```

sp {slam*CheckWallRelation*2*Propose*ChangeRobotPosition
  (state <s> ^name CheckWallRelation
    ^overlap-factor <olf>
    ^overlap {<ol> > <olf>}
    ^target-wall <w1> {<w2> <> <w1>})
  (<w1> ^Length <l1> ^Time <t1> ^R <r1> ^Th <th1>)
  (<w2> ^Length <l2> ^Time {<t2> > <t1>} ^R <r2> ^Th <th2>)
-->
  (<s> ^operator <o> +)
  (<o> ^name ChangeRobotPosition
    ^th-diff (int (exec anglediff2 <th1>|,|<th2>))
    ^Wall-1 <w1>
    ^Wall-2 <w2>
    ^r-diff (- <r1> <r2>))}

```

##### **- Application rule:**

##### **English explanation:**

*If there is an "ChangeRobotPosition" operator  
then put the setOdometry command to the Soar output-link, the  
new location can be calculated based on the distance and angle  
between the two wall. Also, update the wall that were created  
late into the right place based on the same informaion.*

##### **Soar Rule:**

```

sp {slam*CheckWallRelation*2*Apply*ChangeRobotPosition
  (state <s> ^name CheckWallRelation
    ^operator <o>

```

```

      ^io (^output-link <ol>
            ^input-link.currentrobotdata (^X <x>
                                           ^Y <y>
                                           ^Th <th>)))

(<o> ^name ChangeRobotPosition
    ^th-diff <dth>
    ^r-diff <dr>
    ^Wall-2 <w2>
    ^Wall-1 <w1>)
(<w1> ^Time <t1>)
(<w2> ^Start <start> ^End <end> ^Time <t2> ^R <r2> ^Th <th2>)
(<start> ^X <old-sx> ^Y <old-sy>)
(<end> ^X <old-ex> ^Y <old-ey>)
-->

# First, put command to set odometry in output link
(<ol> ^setOdometry <so>)
(<so> ^param1 (+ <x> (* <dr> (cos (* (+ <th> <dth>) 0.0174532925))))
      ^param2 (+ <y> (* <dr> (sin (* (+ <th> <dth>) 0.0174532925))))
      ^param3 (+ <th> <dth>))

# Then, move the wall to the right place
(<start> ^X (+ <old-sx> (int (* <dr> (cos (* (+ <th> <dth>) 0.0174532925))))) <old-sx> -
      ^Y (+ <old-sy> (int (* <dr> (sin (* (+ <th> <dth>) 0.0174532925))))) <old-sy> -)
(<end> ^X (+ <old-ex> (int (* <dr> (cos (* (+ <th> <dth>) 0.0174532925))))) <old-ex> -
      ^Y (+ <old-ey> (int (* <dr> (sin (* (+ <th> <dth>) 0.0174532925))))) <old-ey> -)
(<w2> ^Time <t1> <t2> -
      ^R (+ <r2> <dr>)<r2> -
      ^Th (+ <th2> <dth>) <th2> -)})

```

**Remark:** the soar code such as “^Wall <w1> -” means delete <w1> working memory element. For convenient purpose, we can add new element and delete the old element in the same line by using command like “^Time <t1> <t2> -”; meaning add <t1> and delete <t2>

If there is no rule matched, it means that Soar cannot determine the relation of the two walls or actually they are not related to each other. Soar, then, create another impasse called *state-no-change* impasse. Below is another key rule that detect this situation and remove the proposal for walls investigation to let Soar move on the other operators.

**English explanation:**

*If there is impasse state-no-change from the "CheckWallRelation" state.*

*Then remove the "CheckWallRelation" operator in the top most state.*

**Soar Rule:**

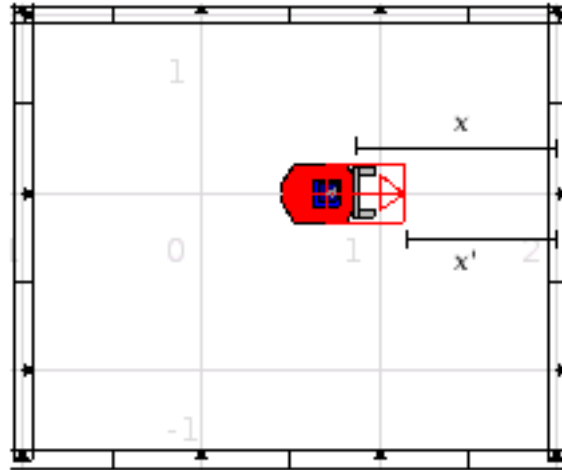
```
sp {slam*CheckWallRelation*NoMatch
  (state <s> ^superstate <ss>
    ^impasse no-change
    ^attribute state)
  (<ss> ^superstate <sss>
    ^name CheckWallRelation)
  (<sss> ^operator <o>)
  (<o> ^name CheckWallRelation
    ^Wall <w1> {<w2> <> <w1>})
-->
  (<sss> ^operator <o> -)}
```

Either one of the rules in *hypothesis-1*, *hypothesis-2*, or *relation-not-found*, is applied. It will cause some changes to the elements or operators in top state, and that changes will cause the "CheckWallRelation" operator to be retracted. Once the operator is retracted, the sub-state along with all working memory elements in the sub-state are wiped out, then Soar can continue matching another operator.

*CheckRobotPosition.soar*

Imagine when we are in an empty room, we can visualize the room structure in our head right after the moment we saw it. Then, if we close our eyes and start walking, the more steps we walk, the less confidence we have. One of the things we can do to avoid ourself walk right in to the wall is reaching out our hands. Once our hands touch the wall, the confidence comes back. We are sure that we standing far from the wall just about the length of the hand.

"*ChechRobotPosition.soar*" contains rules to work on the localization problem by imitating the idea explained above. Soar rules try to verify that when there is a virtual wall in the front, left, or right of the robot at some  $x$  distance, the sonar returns from the same direction should show the same or close distance. If Soar finds the distance from sonar is differed from the  $x$  distance, it will assumes that the robot is not in the spot where it should be. As a result, Soar propose an operator to update the robot position to the right place.



*Figure 24: Show the position shifted between the robot and where it thinks it is. “ $x$ ” is the distance from the wall to the robot, “ $x'$ ” is the virtual distance calculated based on the information in Soar working memory.*

Figure 24, captured from the PlayerStage robot simulator, illustrates the difference between the position of the real robot and the place where it thinks it is. The solid red box represents the real robot which is  $x$  unit far from the wall. The empty red box shows the position in the robot's head and the virtual distance to the wall is represented by  $x'$ .

Follow are the Soar rules version of the mentioned idea. They can be categorized into two parts, the elaboration part and the proposal/application part. The elaboration part mainly create the necessary information to be matched by proposal rule. There are three important elaboration rules. The first one tell whether walls is in the front, left, right, or back, and the second one calculate the virtual distance of the wall from the robot. The last one tells the different between the distance from sonar sensor and the virtual distance.

**English explanation:**

*If there is a virtual wall in the ^wallData and  
there is robot odometry information  
then call Java “checkwallrelation” and  
attach the result to the wall*

**Soar Rule:**

```
sp {slam*elaborate*wall*relation*with*robot
  (state <s> ^superstate <ss>
    ^wallData.Wall <w>
    ^io.input-link.currentrobotdata (^X <rx> ^Y <ry> ^Th <rth>))
  (<w> ^Start (^X <sx> ^Y <sy>))
```

```

      ^End (^X <ex> ^Y <ey>)
      ^R <rw> ^Th <thw>)

-->
(<w> ^relation-to-robot (exec checkwallrelation <sx>|,|<sy>|,|
<ex>|,|<ey>|,|<rw>|,|<thw>|,|<rx>|,|<ry>|,|<rth>)))}

```

**Remark:** By this rule, information about relation between the robot and walls, which are *front, left, right, back, or none*, is calculated and attach to every wall.

**English explanation:**

*If there is a virtual wall which is in the front/left/right/back of the robot*

*then let Java calculate the distance between the robot and the virtual wall and attach the result to the wall*

**Soar Rule:**

```

sp {slam*elaborate*direction*distance*from*wall
  (state <s> ^wallData.Wall <w>
    ^io.input-link.currentrobotdata(^X <xro> ^Y <yro> ^Th <thro>))
  (<w> ^relation-to-robot {<dir> <> none}
    ^Start (^X <xs> ^Y <ys>)
    ^End (^X <xe> ^Y <ye>)
    ^R <rw> ^Th <thw>)

-->
(<w> ^<dir> (int (exec checkwalldistance <xs>|,|<ys>|,|<xe>|,|
<ye>|,|<rw>|,|<thw>|,|<xro>|,|<yro>|,|<thro>|,|<dir>))))}

```

**English explanation:**

*If there is a wall located in front/left/right of the robot at <dist1> far but the sonar that sensing in the same direction return <dist2> which is no equal to <dist1>*

*Then calculate how much is the different and attach the result to the state*

**Soar Rule:**

```

sp {slam*direction*checking*elab
  (state <s> ^superstate <ss>
    ^wallData.Wall <w>
    ^locali-const <c>
    ^sonar (^number <n> ^direction <dir>))
  (<w> ^relation-to-robot <dir>

```

```

    ^<dir> {<dist1> > 0})
  (<c>    ^direction <dir>
    ^<dir> {<dist2> <> <dist1>}
    -^<dir> {<d> < <dist2>})
-->
  (<s> ^diff-detected <dd>)
  (<dd> ^direction <dir>
    ^distance (- <dist1> <dist2>)))}

```

With information from all above rules, now Soar can determine whether the robot suffer from the position shifting or not.

**Proposal Rule:**

**English explanaiion:**

*If there is the different distances between virtual distance and sonar sensing distant, and the different is in reasonable range ( more than 100 and less than 1000)*

*Then propose an "ChangeRobotPosition2" operator to update the robot position.*

**Soar Rule:**

```

sp {slam*direction*checking*propose*update*robot*data
  (state <s> ^diff-detected (^direction <dir>
    ^distance {<d> > 100 < 1000}))
-->
  (<s> ^operator <o> + =)
  (<o> ^name ChangeRobotPosition2
    ^r-diff <d>
    ^th-diff (int (exec dir2angle <dir>)))}

```

**Application Rule:**

**English explanation:**

*If there is an "ChangeRobotPosition2" operator*

*Then put the setOdometry command with calculated parameters into Soar's output-link.*

**Soar Rule:**

```

sp {slam*direction*checking*apply*update*robot*data
  (state <s> ^operator <o>
    ^io ( ^output-link <ol>
      ^input-link.currentrobotdata (    ^X <x>

```



```

^Y <y>
^Th <th>)))

(<o>    ^name ChangeRobotPosition2
      ^r-diff <dr>
      ^th-diff <dth>)
-->
(<ol> ^setOdometry <so>)
(<so> ^param1 (+ <x> (* <dr> (cos (* (+ <th> <dth>) 0.0174532925)))))
      ^param2 (+ <y> (* <dr> (sin (* (+ <th> <dth>) 0.0174532925)))))
      ^param3 <th>)})

```

### *SubGoalElaborate.soar*

Rules in “*SubGoalElaborate.soar*” work as the information forwarder. They mainly copies or links some necessary information from the top state down to the specific sub-state, *CheckWallRelation* sub-state in this case.

There are two rule in this file, the first one forwards ^io to every sub-state, and the other forward the selected operator and its detail from the top state to the *CheckWallRelation* sub-state.

#### **English explanation:**

*If there is a state <s> linked to another superstate <ss>  
and the superstate has ^io  
then link the same ^io to the state <S> itself*

#### **Soar Rule:**

```

sp {slam*subGoal*input-link*coppier
  (state <s> ^superstate <ss>)
  (<ss> ^io <io>)
-->
  (<s> ^io <io>)}

```

#### **English explanation:**

*IF there is an selected operator <o> named “CheckWallRelation”  
in the superstate <ss>, which is superstate of the state <s>  
then link the <o> to the state <s> under new attribute called  
^operator-detail*

#### **Soar Rule:**

```

sp {slam*subGoal*operator*detail*coppier

```

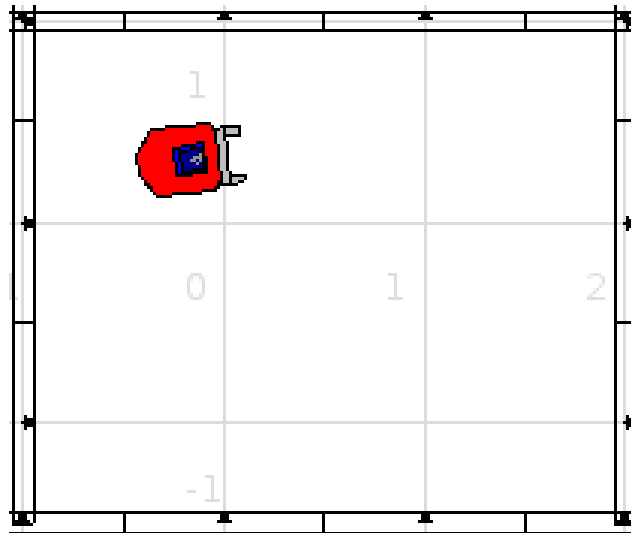
```
(state <s> ^superstate <ss>)  
(<ss> ^operator <o>)  
(<o> ^name CheckWallRelation)  
-->  
(<s> ^operator-detail <o>)}
```

## Experiment and Result

The experiment were done by letting the robot run in an empty rectangular room for about 10 minutes. In the first round, all Soar-SLAM rules were turn off, and the robot were configured to move in very simple manner.

- The robot tends to move forward.
- Whenever the wall blocked its way in the front, it turns right.
- Once it turns into the angle that the wall cannot be seen in the front, it moves forward again.

With this manner, the robot just moved around along the walls. Figure 25 is the PlayerStage captured screen, representing the simulated Pioneer robot in a two-dimensional room.



*Figure 25: Robot placed in an empty rectangular room.*

Along the running course, the robot suffered from error in dead reckon system. In other word, the robot thought it was in the location that different from the actual location. But all sonar sensors still returned the actual distance from the actual location to the actual walls. As a consequence, the virtual walls were generated in the Soar's working memory based on false information as shown in figure 26.

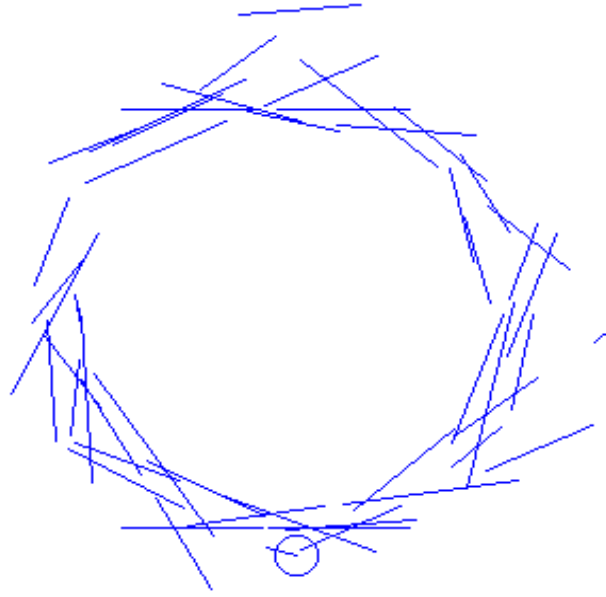


Figure 26: Wall elements in Soar working memory after letting the robot run **without** Soar-SLAM rules in an empty room for 10 minutes.

In the second round, the robot was tested on the same situation and environment with the Soar-SLAM rules turned on.

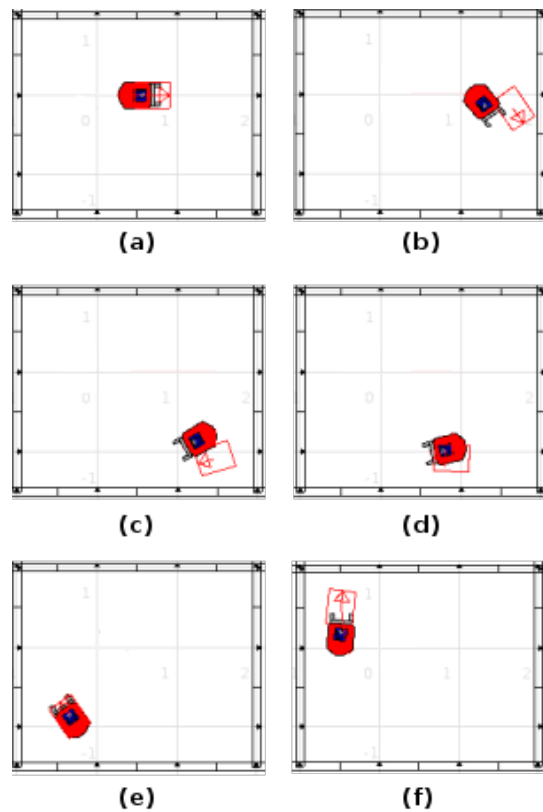
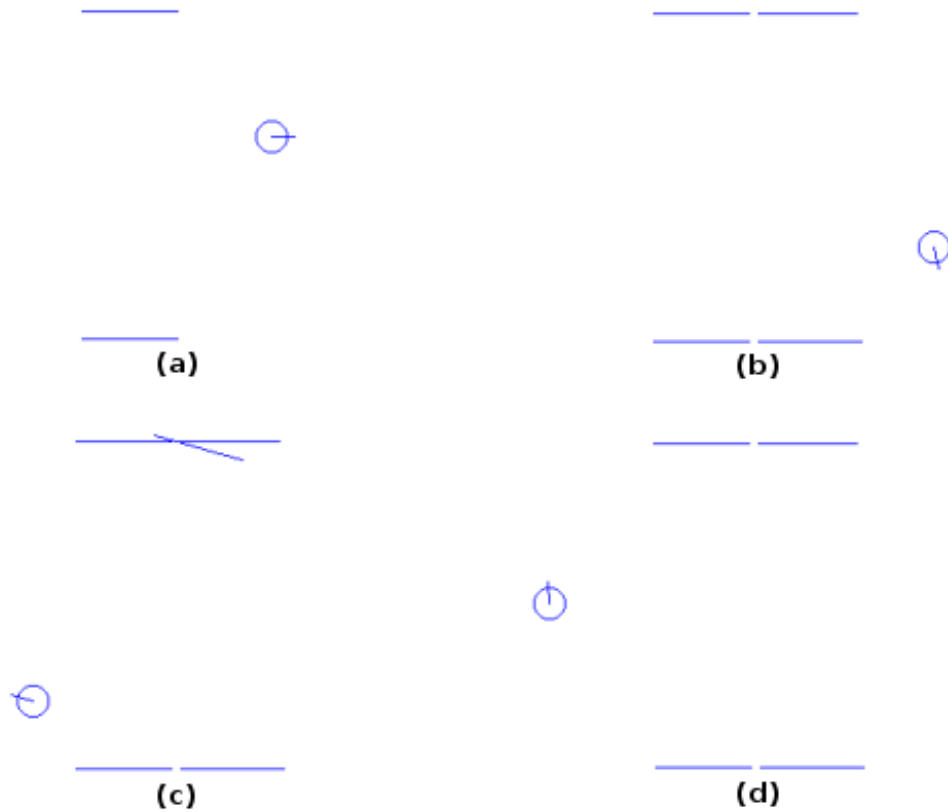


Figure 27: Screen shots captured from PlayerStage, the robot simulator.

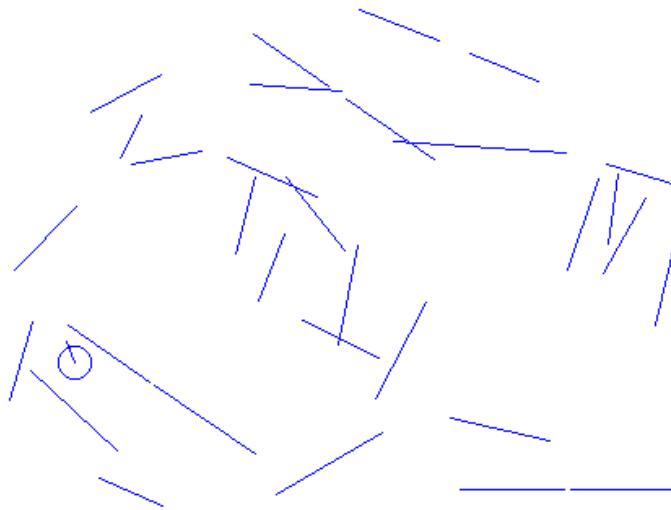
Figure 27 (a) to (f) illustrate the locations of the real robot (the solid red boxes) and the location where it think it was (the clear red boxes) along the course. The error distance occurred all the time. At the moment in the figure 27 (e), Soar-SLAM detected some error in Soar working memory, and then adjusted the location x,y in robot's memory based on that information.



*Figure 28: Wall elements created and update over time.*

Wall elements in Soar's working memory, generated from the same situation, are illustrated in figure 28 (a) to (d). An error wall element were generated sometime during the experiment. Then Soar-SLAM detected and fixed that error as shown in figure 28 (c) and (d).

Figure 29 illustrates the wall elements generated in Soar working memory by help of Soar-SLAM. We can see that number of walls in the second test (figure 29) is relatively less than that of the first test (figure 26). Because many false wall elements were deleted and/or updated during the course.



*Figure 29: Wall elements in Soar working memory after let the robot run with Soar-SLAM rules in a box partition for 10 minutes*

Figure 30 below is the graph comparing the result of both experiment. The Y axis represents the measured distance from actual position of the robot to the position where it thought it was. The X axis represent the time of measuring during the course.



*Figure 30: Graph presents measured distance from the experiment*

We can see from the comparison graph that robot returned a better result with the Soar-SLAM rules.

## Conclusion

Using a cognitive architecture such as Soar to solve SLAM problem gave us a better map result. Even though the generated map cannot be used for global planning (because it does not understand the relationships between rooms) it is effective for local motion planning., using just a few basic operators.

Although we can slowly move around in the completely dark room, we can do much better in the day time or with the lights turned on. Relying purely on sonar sensing is a very tough problem just like human with the eyes blinded.

Possibly implementing some more techniques or concepts about how humans solve the SLAM problem in the completely dark environment would help us pursue a much better result on this topic.

## Opportunities for Further Development

### ***Research more detailed methods about how human solve the problem of SLAM.***

We can obviously see that the map created by Soar-SLAM still need a lot improvement to improve the accuracy of robot guidance. Researching and creating more assumption on the topic of how human resolve SLAM problem would benefit this project a lot. For example, blind people use characteristic of objects beside foot paths as their landmark.

In the near future, when the stereo camera model for our robot is ready, researching and implementing the abstract thought of how human solve the SLAM problem by information from the eyes would be the real improvement on the topic of SLAM under cognitive concept.

### ***Combine information from multiple types of sensors***

Humans rely on five types of sensing, sight, hearing, taste, smell, and touch. We combine all sensing information together to get the best solution for every situations. It would be a big forward step if a robot can learn how to combine different type of information together to solve the problem.

### ***Implement Soar Learning***

Soar learning capability is one of the reasons that we base our research on it. To be able to use this feature correctly, we need more details and assumptions of the basic SLAM solution concepts. Once we have much enough rules implemented based various basic ideas, we will be able to take more advantage from Soar by letting it learn and create rules for more complicate ideas by itself.

## Bibliography

- 1: ActiveMedia Robotics, LLC., Pioneer 2 / People Bot Operation Manual, 2002
- 2: Wikipedia, Hough Transform, 2007
- 3: Juan D. Tardos, Jose Neira, Paul M. Newman, John J. Leonard, Robust Mapping and Localization in Indoor Environments using Sonar Data., 2002
- 4: John J. Leonard, Directed Sonar Sensing for Mobile Robot Navigation, 1992
- 5: Jill Fain Lehman, John Laird, Paul Rosenbloom, Gentle Introduction to Soar, 2006
- 6: John E. Laird, Soar 8 Tutorial, 2006
- 7: Robert I. Follek, SoarBot: A rule-based system for playing poker, 2003
- 8: Lyons, D.M. and Arbib, M.A., A Formal Model of Computation for Sensory-based Robotics, 1989
- 9: Larid, J.E., Newell, A. and Rosenbloom, P.S., , 1987
- 10: D. Paul Benjamin, Tom Achtemichuk, Damian Lyons, A Fast Predictive Vision System for a Mobile Robot, 2006
- 11: playerstage.sourceforge.net, The Player Project: Free Software tools for robot and sensor application,
- 12: Brian Gerkey and contributors, The Player Robot Device Interface, 1999-2007
- 13: Richard Vaughan and contributors, The Stage Robot Simulator, 1998-2006
- 14: Java-Player, Java Client for Player/Stage, 2002