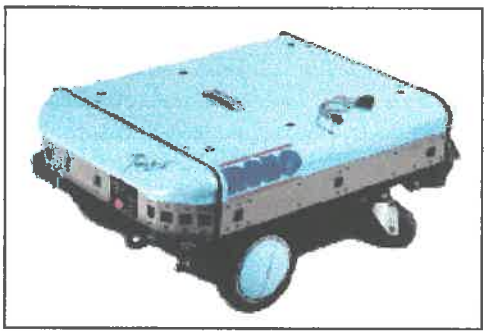
Universidade de Aveiro

Departamento de Electrónica e Telecomunicações
Secção Autónoma de Engenharia Mecânica

Programa de Navegação e Comunicações para um Robot Móvel



Realizado no âmbito de um projecto de 5º ano, do curso de Eng. Electrónica e Telecomunicações, por:

Emanuel Amaral de Oliveira **Nº 10580**
Paulo Miguel de Jesus Dias **Nº 12218**

Orientação conjunta de:

Luís Almeida (Dep. Electrónica e Telecomunicações)
Vítor Santos (Dep. Mecânica)

9 de Julho de 1998

Índice

Prefácio

O trabalho apresentado neste relatório é resultante de um projecto cujos objectivos se destacam pela sua originalidade, pois para além da construção de ferramentas básicas ao controlo do robot, desenvolveram-se mecanismos para atravessamento de passagens estreitas de forma autónoma sem nenhum conhecimento prévio do espaço onde o robot evolui.

Assim, a primeira parte deste projecto, consistiu na construção de um conjunto de ferramentas básicas de comunicação com o robot, bem como módulos que permitam auxiliar a navegação.

Numa segunda parte, levou-se o robot a reconhecer e atravessar, autonomamente, passagens estreitas.

Nos módulos agora apresentados, principalmente os módulos de emergências e ultra-sons, é possível a introdução de alguns melhoramentos, nomeadamente na escolha dos sensores para a deteção de uma situação de emergência, bem como o aumento da versatilidade do módulo de gestão dos sensores de ultra-sons, permitindo, por exemplo, que sejam definidas estratégias de disparo dos sensores. É ainda possível colocar o tratamento das mensagens nos módulos respectivos por forma a só responder a uma dada mensagem quando o módulo correspondente está presente.

No que diz respeito ao atravessamento de portas, é necessário introduzir alguns melhoramentos nos algoritmos por forma a torná-los mais robustos, sendo ainda precisos mais testes e em situações mais variadas, já que todo este trabalho foi desenvolvido com base numa só configuração da passagem.

Os autores

I - Programa de Navegação e Comunicações para um Robot Móvel

1. Descrição Geral

Todo o trabalho desenvolvido no âmbito deste projecto foi realizado com a plataforma rectangular, Robuter III, da Robosoft. Este robot tem a bordo um 68040 da Motorola onde corre o sistema operativo de tempo real Albatros, desenvolvido pela Robosoft.

1.1 Objectivos

Pretendeu-se com esta parte do projecto, o desenvolvimento de uma aplicação, a ser executada no robot, que permita a interpretação de comandos remotos, envolvendo deste modo a comunicação entre uma aplicação remota e o robot.

Para além da interpretação de comandos, a aplicação também realizará algumas acções de apoio à navegação do robot, principalmente a detecção de situações de emergência. (ver 2.3).

1.2 Modularidade

A modularidade da aplicação foi um aspecto tido sempre em conta ao longo do desenvolvimento. Sendo a aplicação modular, é possível acrescentar ou remover módulos que se acharem convenientes, sem que para isso seja necessário alterar os módulos já existentes. Assim, pode o utilizador, para além dos módulos existentes nesta aplicação (ver 2), adicionar ou remover os módulos que desejar, e/ou substituir um módulo existente por um outro, bastando para isso incluí-lo na *Makefile*.

É também bom de referir que a modularidade não permite uma total liberdade em relação ao desenvolvimento dos módulos a adicionar. Existem regras de comunicação entre os vários módulos (ver 1.3) que devem ser respeitadas, pelo que no desenvolvimento de um módulo a adicionar, deve ser tido em conta os módulos com os quais irá comunicar, para que sejam usados os canais e protocolos de comunicação correctos.

1.3 Comunicação entre módulos

Existem estruturas de dados que podem ser partilhadas por vários módulos, como por exemplo a estrutura de dados associada aos dados de ultra-sons, que para além de ser actualizada pelo módulo responsável pela gestão dos sensores de ultra-sons pode ser lida por outros módulos que tenham interesse nos valores resultantes das leitura dos ultra-sons.

Seria bastante interessante utilizar mecanismos que limitassem o acesso a uma dada estrutura de dados a apenas alguns módulos, e que esse acesso fosse de alguma forma condicionado, permitindo que, por exemplo, só um módulo actualize uma dada estrutura de dados e que todos os outros só possam ler os valores nela contidos.

Para implementar tal estratégia de comunicação entre módulos, poder-se-ia, por exemplo, usar mensagens entre os vários módulos ou usar a linguagem de programação orientada por objectos no desenvolvimento dos módulos, pois assim é possível o encapsulamento da informação (dados e funções de acesso aos dados), permitindo o estabelecimento de hierarquias limitando o acesso aos dados.

Apesar da elegância das soluções apresentadas atrás, optou-se por uma comunicação entre processos baseada em variáveis globais. Esta solução embora menos elegante é sem dúvida fácil de implementar, pois ler e escrever em variáveis globais é feito de modo análogo a ler ou escrever em variáveis locais; e pode ser também mais eficiente, pois contorna todos os aspectos protocolares associados a um mecanismo de, por exemplo, troca de mensagens, que implica quase sempre algum atraso nas acções a desempenhar por cada um dos módulos.

Na figura 1, aparecem as principais variáveis globais que foram utilizadas associadas ao módulo do qual elas dependem e ligadas aos vários módulos onde elas são utilizadas, dando uma ideia global da estrutura de dados que foi utilizada para a comunicação entre os processos.

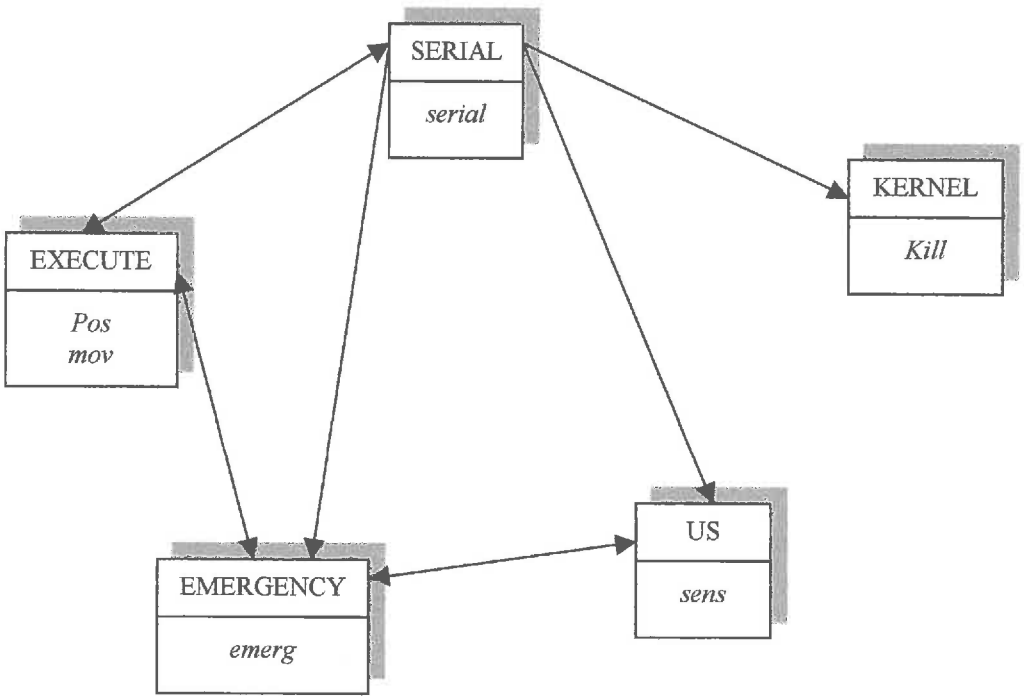


Figura 1: Estrutura de dados usada para a comunicação entre processos

2. Módulos desenvolvidos

2.1 Comunicação série – SERIAL.C

Este módulo é encarregue da gestão de todas as mensagens que são recebidas pela porta série no robot. Para além de testar se existe mensagens, este módulo trata-as, seja através do envio da respectiva resposta ou da actualização dos parâmetros adequados.

A filosofia actual tem o inconveniente de não ter em conta a presença de um módulo para responder as respectivas mensagens.

2.1.1 Formato das mensagens utilizadas

O formato utilizado para as mensagens que utilizamos é baseado sobre pesquisas anteriores desenvolvidas na área de comunicação com plataforma móveis.

De seguida são apresentadas as mensagens tratadas, sendo possível acrescentar novas mensagens associadas a outras funções, tal como a manobra duma unidade *pan-tilt*, permitindo assim a expansão das mensagens existentes.

Neste formato, as mensagens são constituídas pela categoria, representado pelo primeiro caracter, a seguir vem o tipo de mensagem e por fim, os parâmetros, se necessários.

A categoria e o tipo estão juntos e os parâmetros separados entre eles por um separador (em princípio o espaço) a mensagem é terminada por um caracter de fim de linha (\r), sendo o comprimento da mensagem variável.

A sintaxe geral é a seguinte:

```
<mensagem recebida> ::= <categoria> <tipo> [<parâmetros>] <terminador>
<mensagem enviada>  ::= <tipo> <informação> <terminador>
<parâmetros>        ::= <parâmetro> <separador> <parâmetro> ...
<informação>        ::= <informação> <separador> <informação> ...
<terminador>        ::= <fim de linha (/r)>
<separador>         ::= <espaço> ou <virgula>
```

2.1.1.1 Categorias

As três categorias que utilizadas são:

B : pedidos de informação ao robot.
A : envio de comandos.
U : mensagens associadas aos ultra-sons.

Mensagens de pedido de informação

São mensagens que começam com o caracter B e que pedem informações ao robot, pelo que esperam uma resposta.

#Define	Char	Descrição e exemplo	Exemplo de resposta
IN_POST	'O'	Pedido da postura do robot **BO**	S 200 120 34 1 162341
IN_US	'U'	Pedido dos dados dos ultra-sons **BU**	U 750 1232 ... 162341
IN_POSTUS	'V'	Pedido da postura e dados dos ultra- - sons **BV**	V 200 120 34 1 750 ... 162341
IN_CLOCK	'C'	Pedido do relógio interno do robot **BC**	C 162341
IN_USTOUT	'T'	Pede o valor do *Time out* na leitura dos ultra-sons **BT**	T15

Mensagens unidireccionais

São mensagens que começam com a letra A e que enviam um comando para o programa efectuar. São respondidas com um *ack* ou um *error*.

#Define	Char	Descrição e exemplo	Exemplo de comandos
IN_VELOCITY	'M'	Impõe um movimento em velocidade (cm/s) **AM <esquerda> <direita>**	AM 5 4
IN_SERV	'S'	Activa/desactiva o processo de baixo nível de controlo de velocidade **AS<estado> N:inactivo F: activo**	ASN
IN_ALBATROS	'Y'	Comando directo ao Albatros **AY<comando>**	AYODOM ON
IN_ODOM	'C'	Impõe a postura do robot **AC<postura>**	AC0 0 0
IN_POSITION	'W'	Impõe um movimento em posição. **AW <x y theta>**	AW 300 250 10
IN_KILL	'X'	Termina a aplicação	AX

Mensagens associados aos ultra-sons

São mensagens que começam com a letra U e que permitem pedir informações (exigindo neste caso uma resposta) ou enviar configurações (respondidas só pelo *acknowledge*).

#Define	Char	Descrição e exemplo	Exemplo de comando
IN_USTABLE	'T'	Configura a tabela de sensores activos **UT<tabela>**	UT101…1001
IN_NDELAY	'D'	Configura o tempo de disparo entre nodos **UD<tempo>**	UD5
IN_GNDELAY	'N'	Pede o tempo de disparo entre nodos **UN**	UN
IN_STAT	'S'	Pede informações estatísticas relativas aos ultra-sons: **USN**: tempo médio para ler um sensor (em ticks) **USS**: tempo médio para ler os sensores todos (em ticks)	S45
IN_SUSTOUT	't'	Impõe o time_out **Ut<time_out>**	Ut30
IN_TOGSENS	'u'	Modo automático-desliga sensores inúteis **Uu< 0: inactivo 1: activo>**	Uu1

2.1.1.2 Respostas

A seguir, só indicamos as respostas específicas, uma mensagem que só configura o sistema e não espera resposta específica deve sempre ser respondida com um *ack* ou *error*.

É ainda de notar que todas as resposta terminam com um terminador que é o fim de linha.

#Define	Char	Descrição e exemplo	Resposta a
OUT_POST	'S'	Postura do robot **S <x> <y> <θ> <s> <t>**	BO
OUT_US	'U'	Dados dos ultra-sons **U <u1>…<u24> <s> <t>**	BU
OUT_POSTUS	'V'	Postura e dados dos ultra-sons **S <x> <y> <θ> <u1>…<u24> <s> <t>**	BV
OUT_CLOCK	'C'	Relógio interno do robot **C <relógio>**	BC
OUT_GDEL	'N'	Tempo de disparo entre nodos **N <tempo>**	UN

2.1.2 Variáveis utilizadas

O processo ***serial()*** é responsável por uma variável (**serial**) que é global a este módulo, não a partilhando com os outros, afim de evitar que esses possam ter acesso a linha série. Para garantir isso, essa variável foi designada como *static*.

Na variável **serial** estão definidas todas as variáveis que controlam a linha série:

```
typedef struct {
  char buff_in[MAX_LENGTH];      /*buffer de entrada*/
  char buff_out[MAX_LENGTH];     /*buffer de saida*/
  int desc;                      /*Descritor da linha série*/
  int send;                      /*Indica a existência de uma resposta
                                 para enviar*/
} SERIAL
```

2.1.3 Como acrescentar uma mensagem

Mais uma vez por forma a tornar o nosso programa o mais versátil possível, o módulo de comunicação série foi desenvolvido de maneira a ser fácil acrescentar novas mensagens. Para tal, basta acrescentar no módulo **serial.h** os *defines* relativos à nova mensagem.

Depois, a mensagem tem de ser tratada no módulo **serial.c,** mais precisamente na função ***decode()***. No caso de existir uma resposta à mensagem (para além de um *ack* ou um *error*) é necessário actualizar a variável **serial.send** com o valor *TRUE* por forma sinalizar ao módulo que este deve enviar a resposta para a linha série.

2.1.4 Processo

O processo baseia-se principalmente em duas funções. A primeira chamada ***serial_spy()*** é responsável pela leitura da linha série. É de notar que o processo de leitura é não bloqueante: se não houver nada a ler, a função retorna. Esta função também envia as respostas para a linha série se a variável **send** estiver activa.

O algoritmo desta função é o seguinte:

Leitura de um carácter da linha série, se não houver nada, retorna;
Se o carácter não for um terminador, acrescenta-o ao **buff_in** e retorna;
Caso contrário, acrescenta o terminador ao **buff_in**, actualiza o **emerg.time_stamp** (usado pelo processo de emergência para detectar perdas de comunicações) e invoca a função ***decode()***;
Por fim testa se existe uma resposta e envia-a se necessário.

A função ***decode()***, que é invocada quando uma mensagem foi recebida, é encarregue de descodificar a mensagem e depois actuar em função desta. Esta actuação tanto pode ser construir a resposta como actualizar as variáveis globais adequadas para realizar a operação pedida (actualização das velocidades, alteração da tabela de sensores activos, ...).

2.2 Módulo de leitura dos sensores de ultra-sons - US.C

Este módulo é responsável pela gestão dos sensores de ultra-sons. Com este módulo é possível estabelecer um conjunto arbitrário de sensores activos, sendo também possível a definição do tempo de atraso entre o disparo de nodos (*node firing delay*).

Para um correcto uso deste módulo, é necessário que antes que o processo ***read_sensors()*** seja lançado, se invoque a função ***init_read_sensors()***. Esta função realiza três funções que são essenciais para o correcto funcionamento deste módulo.

Em primeiro lugar atente-se á **nova configuração dos sensores de ultra-sons** existente no robot, apresentada na figura 2. Aqui poder-se-á ver a troca das posições dos sensores 2 e 24 em relação à configuração original. Isto permite que estes sensores fiquem mais afastados dos sensores existentes em cada um dos nodos aos quais pertencem, atenuando assim alguns efeitos de *cross-talk*.

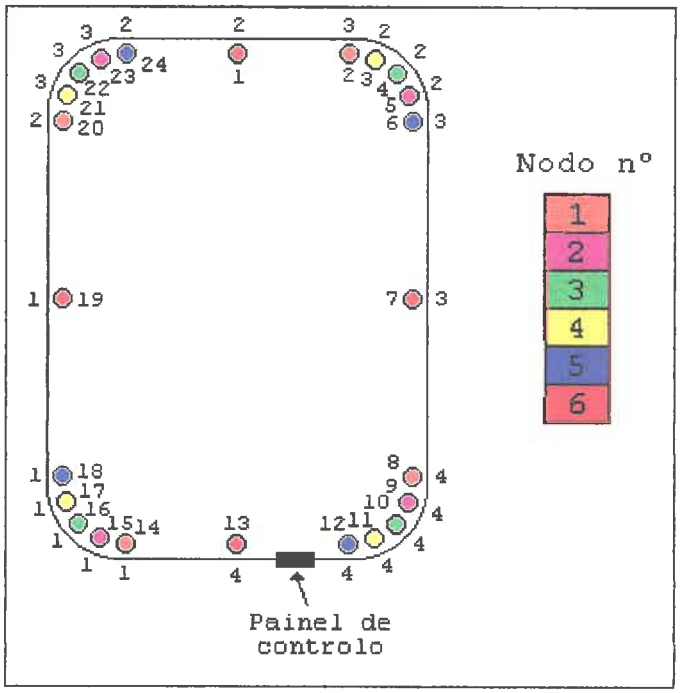


Figura 2: Configuração dos sensores de ultra-sons

Afim de garantir o funcionamento correcto da função *get_us_conf()* com esta nova configuração de sensores, a função *init_us* actualiza a tabela de configuração de sensores incluída na DDB (*Device Data Base*).

O valores produzidos por este módulo são distâncias que são dadas em milímetros. Após alguns testes verificou-se que os valores medidos pelos sensores de ultra-sons eram superiores aos reais, sendo o erro praticamente constante. Assim, afim de melhorar os resultados obtidos foi diminuído o valor da **velocidade do som**, usado pela função READ para o cálculo da distância, para o valor 300 m/s, conseguindo-se, deste modo, valores bastantes precisos para distâncias entre 15 e 100 cm; note-se que este intervalo de distâncias é bastante importante na avaliação de uma situação de emergência

Com o propósito de diminuir o tempo de leitura dos sensores, o ***time-out*** foi reduzido de 100 ms para 33ms, fazendo com que a máxima distância que se poderá medir seja aproximadamente 5m.

Um dado que merece ser referido, dado a sua importância para o desenvolvimento de aplicações que envolvam este módulo, é o do tempo de leitura dos 24 sensores de ultra-sons que é de aproximadamente 29 *ticks* (290ms). Nestas condições, fizeram-se alguns testes que mostraram que não havia *cross-talk*.

Foram implementadas duas novas mensagens relacionadas com este módulo (ver 2.1) relacionadas com os dados estatísticos referentes às leituras dos ultra-sons. O valor do *scan_time* (ver 2.2.1) só é calculado se os todos os nodos tiverem sensores activos.

2.2.1 Variáveis globais

Este módulo disponibiliza uma variável global cuja nome é **sens** sendo a sua estrutura definida do seguinte modo:

```
typedef {
  int change_flag          /* Indica que houve mudança nos sensores activos */
  int active_sensors[24]; /* Configuração dos sensores activos */
  int node_delay;          /* Node firing delay */
  long values[24];         /* Valores das leituras */
  struct {                 /* Usado para fins estatísticos */
    long node_mean_time;   /* Tempo médio de leitura de um nódo */
    long scan_time;        /* Tempo médio de leitura dos 6 nodos */
  } stat;
} US_DATA;
```

2.2.2 Como usar o módulo US.C

Depois de incluir o ficheiro US.C na lista de ficheiros a serem compilados, no(s) módulo(s) que partilham a estrutura de dados produzida por este, devem:

- incluir a linha: ***extern US_DATA sens;***

- para alterar a configuração de sensores activos, colocar o valor ACTIVE ou NOT_ACTIVE na posição do *array* ***sens.active_sensors[]*** correspondente ao sensor pretendido e de seguida colocar a ***sens.change_flag*** com o valor TRUE para que o módulo de leitura dos ultra-sons actualize a sua tabela interna de sensores activos;

- para ler valores, referenciar o *array* ***sens.values[]*** que contém o valor da última leitura feita para cada um dos sensores. Note-se que caso o sensor esteja desactivado, o seu valor é zero;

- para modificar o *node firing delay*, escrever o novo valor na variável ***sens.node_delay;***

- verificar se o ficheiro ***generic.c*** se encontra na lista de ficheiros a serem compilados.

Por fim, no ficheiro ***kernel.c***, incluir o processo ***read_sensors()*** na tabela de processos a activar e invocar a função ***init_us()*** antes de lançar o processo (o não uso da função ***init_us()*** pode levar a resultados inesperados).

2.2.3 Processo e funções associados ao módulo US.C

Para além do processo ***read_sensors()*** responsável pela gestão dos sensores de ultra-sons, estão disponíveis com este módulo as seguintes funções:

int get_sensor_number(int Node_number, int Number_in_node);
Retorna o número do sensor (1..24) do sensor cujo nodo é *Node_number* e cuja posição dentro do nodo é *Number_in_node*.

int get_sensor_node(int number);
Retorna o número do nodo do sensor número *number* (1..24).

2.3 Módulo de emergências - EMERG.C

Este módulo, tem as seguintes funções:

- detectar a aproximação crítica a objectos;
- detecção de colisão;
- detecção de perda de comunicações.

Qualquer uma destas situações de emergência pode ser activada ou desactivada usando a combinação adequada das *flags* US, BUMPER e COM, como no seguinte exemplo:

emerg.on = BUMPER | COM detecta colisões e perda de comunicações

Afim de garantir as funções atrás referidas, este módulo necessita de quatro tipos de dados:
i) distâncias medidas pelos sensores de ultra-sons;
ii) velocidades de ambas as rodas do robot;
iii) *time-stamp* da última mensagem recebida;
iv) estado dos *bumpers* (sensores de contacto).

Para detectar a aproximação crítica a objectos, e entenda-se por aproximação crítica, uma abordagem a um objecto que provoque uma colisão, são analisados dois tipos de dados: dados dos ultra-sons e velocidades das rodas.

Com a velocidade de cada uma das rodas, são calculados o raio e a direcção da curvatura da trajectória do robot, da seguinte forma:

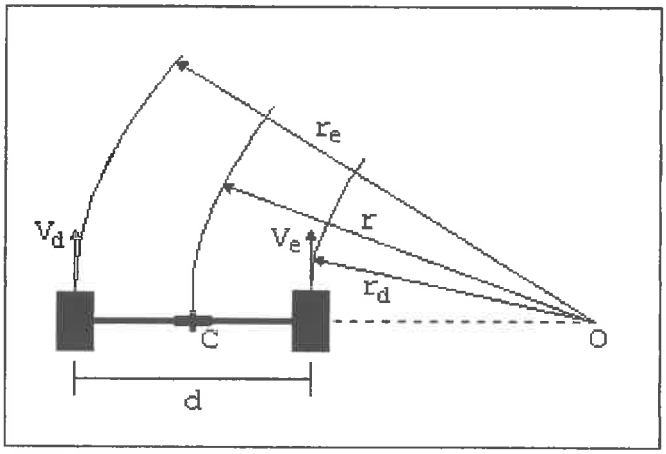


Figura 4: Determinação do raio da curvatura

tendo em conta a figura 4, onde **d** representa a distância entre as rodas motrizes, temos:

$$\omega = \frac{V_e}{r+\frac{d}{2}} = \frac{V_d}{r-\frac{d}{2}}$$, velocidade angular do robot em relação ao ponto C

que pode ser escrito da seguinte forma:

$$\begin{cases} V_e = \omega r + \frac{\omega d}{2} \\ V_d = \omega r - \frac{\omega d}{2} \end{cases}$$

Logo

$$V = \frac{V_e + V_d}{2}$$, velocidade linear do ponto C

$$\omega = \frac{V_e - V_d}{d}$$, velocidade angular do ponto C

Atendendo que $r = \frac{V}{\omega}$, vem:

$$r = \frac{V_e - V_d}{V_e + V_d}\frac{d}{2}$$

Depois de determinado o raio da curvatura, é necessário determinar qual o sensor central* correspondente a essa trajectória. Para tal, vamos associar a cada sensor, uma determinada trajectória. Assim, tendo em conta a figura 5, podemos escrever que a circunferência de raio **r** e que contém o ponto **(0,0)** tem a seguinte equação:

* Para uma dada direcção do movimento do robot é escolhido um conjunto de sensores, definidos no módulo de emergências por *watch dog sensors*. Esse conjunto de sensores é escolhido, salvo em casos particulares, de modo que a sua disposição seja centrada no sensor determinado a partir do raio da curvatura - a esse sensor chama-se *sensor central.*

$$(x-r)^2 + y^2 = r^2$$

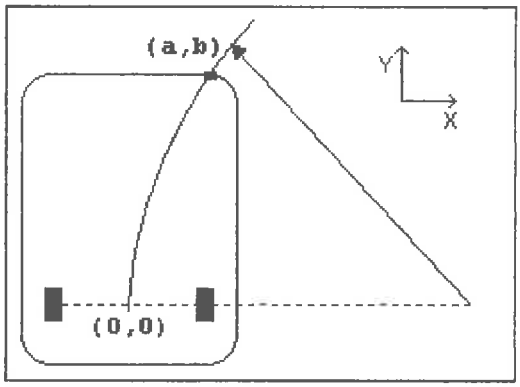


Figura 5: Determinação do sensor central

Para que o ponto **(a,b)** pertença à circunferência, tem que satisfazer a equação anterior, pelo que:

$$(a-r)^2 + b^2 = r^2$$

Dado isto, para cada par de sensores adjacentes, foi calculado uma fronteira que é utilizada como limiar de decisão na escolha de um sensor de acordo com o raio da curvatura, como ilustra a figura 6.

Conclui-se assim, que para um sensor cujas coordenadas são **(a,b)** existe uma trajectória de raio

$$r = \frac{a^2 + b^2}{2a}$$

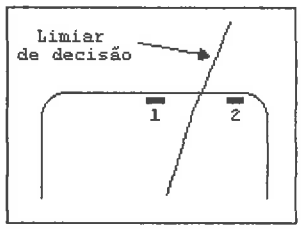


Figura 6: Escolha do limiar de decisão

Para os valores de ultra-sons era necessário definir à partida quais os sensores importantes na detecção de uma situação de emergência. Uma primeira conclusão foi que só seriam necessários os sensores que estivessem relacionados com o movimento do robot, isto é, num movimento para a frente, por exemplo, só faz sentido analisar os sensores da frente afim de detectar a aproximação a um objecto.

É claro que podemos ter, por exemplo, o robot a movimentar-se para a frente e um objecto em aproximação crítica por trás. Esta é uma situação que para além de ser pouco provável, especialmente num ambiente estático, implicaria uma resposta do robot muito para além do simples parar. Este tipo de situações sai um pouco do âmbito deste trabalho, pelo que não foram objecto de estudo. Mas é importante realçar, que todo o *software* desenvolvido permite ser expandido, pelo que muito facilmente se pode incluir um módulo que trate este tipo de situações.

Tendo uma noção de quais os sensores a analisar, era necessário determinar o número óptimo de sensores. Foi então necessário chegar a um número que minimizasse a possibilidade de colisão e o número de paragens indevidas (falsas emergências).

Após vários testes, entre os quais, testes de abordagem a objectos segundo direcções quase paralelas a estes, testes com curvas bastante acentuadas e testes de rotações, verificou-se que 5 sensores eram um bom compromisso para satisfazer os requisitos referidos atrás.

Os cinco sensores são escolhidos de modo a estarem centrados no sensor central, determinado segundo o método atrás descrito. Esta regra não é aplicada quando o robot descreve curvas apertadas ou rotações, pois nestes casos é necessário que os sensores não só abranjam a direcção do movimento, mas também outras direcções de modo a proteger a plataforma, mas mantendo sempre o número de cinco sensores a serem analisados.

Para uma análise dos sensores analisados em cada situação recomenda-se que se façam testes com o robot no modo *auto-toggle of useless sensors*, e se verifique quais os sensores activados (analisados).

No modo *auto-toggle of useless sensors* só são activados os (cinco) sensores necessários ao módulo de emergências. Caso o utilizador envie uma nova tabela de configuração de sensores, essa tabela é ignorada, pelo que a imposição por parte do utilizador de uma tabela de configuração de sensores activos só pode ser feita com o *auto-toggle* no estado *disable*. Caso o robot esteja parado, todos os sensores estão desligados, pois a ausência de movimento implica a não existência de aproximações críticas a objectos - emergência.

É de notar também que usando este modo, qualquer mudança de movimento leva o robot a parar pois este necessita de esperar por novos dados de ultra-sons.

Devido à sua inércia, o robot necessita de distâncias diferentes de paragem em função da velocidade. Optou-se assim para uma solução em que a distância de segurança aumenta proporcionalmente à velocidade. Como resultado disto, o robot consegue aproximar-se mais de um objecto quando se movimenta a uma velocidade reduzida.

Para além da detecção de aproximação crítica a objectos é também analisada a ocorrência de colisão. Esta análise é feita mediante o teste ao estado dos *bumpers*. Caso estes estejam activados é imediatamente sinalizada a *flag* de emergência.

A situação de colisão ocorre apenas quando a configuração do espaço envolvente é tal que os sensores de ultra-sons falham. Essas falhas deve-se a limitações do próprio sistema de ultra-sons, que, por exemplo, quando o ângulo de incidência é bastante aberto, leva a que não exista reflexão do feixe emitido, tornando o objecto *invisível* aos ultra-sons.

É em situações análogas às referidas atrás que os *bumpers* tomam vital importância, impedindo que o robot imponha um movimento em velocidade mesmo em situação de colisão.

Por último existe o teste de perda de comunicações. Este teste é feito com base no *time stamp* da última mensagem recebida e do tempo actual do relógio do robot. Caso exista uma diferença superior a 3 minutos é activada a *flag* de emergência.

Resta referir que, no âmbito deste trabalho, a acção tomada em caso de emergência e a paragem conseguida através de um *SERV OF* feito logo que seja detectada uma emergência por este módulo.

2.3.1 Variáveis globais

O módulo de emergências disponibiliza uma variável de nome **emerg** cuja estrutura de dados é definida do seguinte modo:

```
typedef struct {
  int flag;          /* Flag de emergência */
  int rtm;           /* Pedido para movimentar (Request To Move) */
  int us_change;     /* Pedido de mudança da tabela de sensores activos */
  long time_stamp;   /* Time stamp da última mensagem recebida */
  int auto_toggle;   /* Auto-toggle of useless sensors */
} EMERGENCY;
```

2.3.2 Como usar o módulo EMERG.C

Para o correcto uso do módulo de emergências deve-se:

- incluir o ficheiro US.C na lista de ficheiros a serem compilados

- nos módulos onde se pretende fazer uso do módulo de emergências, deve-se incluir a linha: ***extern EMERGENCY emerg;***

- para alterar a configuração de sensores activos colocar o valor *ACTIVE* ou *NOT_ACTIVE* na posição do *array* ***sens.active_sensors[]*** correspondente ao sensor pretendido e de seguida colocar a ***emerg.us_change*** com o valor *TRUE* para que o módulo de emergências actualize permita ou não a actualização da tabela de sensores activos mediante o estado do *auto-toggle of useless sensors*;

- para fazer o *enable* do *auto-toggle of useless sensors*, fazendo com que só os sensores importantes para o módulo de emergências estejam activos, colocar o valor *TRUE* na variável ***emerg.auto_toggle***;

- para validar um pedido de movimento, colocar a variável ***emerg.rtm*** com o valor *TRUE*;

- sempre que seja recebida uma mensagem, actualizar o ***emerg.time_stamp*** com o valor do relógio na altura de recepção;

- verificar a existência das seguintes variáveis globais:
  MOVES mov (ver 2.4);
  US_DATA sens; (2.2);

- verificar se o ficheiro ***generic.c*** se encontra na lista de ficheiros a serem compilados.

- no ficheiro ***kernel.c***, incluir o processo ***emergency()*** na tabela de processos a activar e invocar a função ***init_emergency()*** antes de lançar o processo (o não uso da função *init_emergency()* pode levar a resultados inesperados).

2.3.3 Processo e funções associados ao módulo EMERG.C

O processo responsável pelas emergências é o ***emergency()***. São também disponibilizadas neste módulo as seguintes funções:

int sub24(int, int);
Subtrai dois números de sensores.

int add24(int, int);
Adiciona dois números de sensores;

int get_bumper_state(void);
Retorna o estado do *bumper*.
Valores de retorno: `NOT_PRESSED`
`PRESSED`

int get_main_sensor(void);
Retorna o sensor central;

2.4 Módulo executor de movimentos – EXECUTOR.C

Este módulo é encarregue de supervisionar todos os movimentos do robot. É pois crucial que só um processo possa actuar sobre os motores do robot por forma a evitar situações conflituosas em que dois processos tentariam dar ordens opostas à plataforma levando o robot a oscilar.

O processo está encarregue principalmente de quatro acções:

- Actualizar as variáveis de posição.
- Testar uma mudança do estado do serv e realizar essa mudança
- Realizar os movimentos necessários no caso de não haver emergência.

2.4.1 Variáveis globais

O executor de movimentos é responsável por duas das variáveis globais que são utilizadas para a comunicação entre processos e definidas no *headerfile* associado.

A variável **pos** contém todas as informações sobre a posição do robot relativamente a sua posição no instante em que o programa foi lançado já que estas posturas são obtidas através do comando do Albatros *ODOM* que é inicializado na função ***init_execute()***. Os campos desta variável são:

```
typedef struct {
  long int x;     /*Posição do robot num eixo ortogonal as rodas*/
  long int y;     /*Posição do robot num eixo paralelo as rodas*/
  long int theta; /*Orientação do robot em relação ao eixo dos x*/
} POSTURE;
```

A variável **mov** existem quatro campos relativos a movimentação do robot, nomeadamente o estado do servo, as velocidades e a necessidade de alterar as velocidades das rodas.

```
typedef struct {
  int status;  /*estado do serv       0-serv of
                                      1-serv on*/
  int left;    /*Velocidade do motor esquerdo*/
  int right;   /*Velocidade do motor direito*/
  int flag;    /*Flag que indica uma mudança nas velocidades*/
} MOVES;
```

2.4.2 Como usar o módulo EXECUTOR.C

Para usar este módulo, para além de incluir o ficheiro executor.c nos ficheiros a compilar, no caso de utilizar as variáveis globais do módulo, é necessário acrescentar nos módulos de interesse as linhas:

extern POSTURE pos;
extern MOVE nov;

Para ler os valores dos odómetros, basta ler os valores x, y e theta contidos na estrutura pos.

Se se pretender realizar um movimento, é necessário colocar nas variáveis **mov.left** e **mov.right** as velocidades desejadas. De seguida, o programador tem dois meios para sinalizar ao executor que as velocidades foram alteradas:

- colocar a variável **mov.flag** a *TRUE*. Neste caso, o movimento será iniciada sem previamente testar condições de emergências, logo um movimento de alta velocidade poderá iniciar-se antes de ser detectada uma emergência provocando em certos casos, situações indesejáveis e **perigosas**, pelo que esta opção é de evitar;
- colocar a variável **emerg.rtm** a *TRUE*. Neste caso, o módulo de emergência será consultado antes de efectuar qualquer movimento pelo que o movimento só será efectuado se for possível. **Esta opção é vivamente aconselhada** ao programador no desenvolvimento de novos módulos.

As operações de actualização do freio (*SERV OF/ON*) são feitas através da variável status: 0, o freio é desactivado, 1, ele é activado.

2.4.3 Processo

Para além do processo, está definida a função *init_execute* no modulo executor.c. Esta função é invocada no *kernel* antes de arrancar o processo por forma a inicializar as variáveis que lhe são necessárias.

2.5 Kernel

Como o programa está baseado em processos de forma a torná-lo tão modular quanto possível, o *kernel* tornou se um módulo bastante simples, pois apenas invoca as funções que inicializam cada um dos processos antes de os lançar e testa uma variável que só será activada para sair do programa e fechar os processos.

2.6 Funções genéricas

Como apareceu a necessidade de definir funções que seriam utilizadas por mais de um módulo, foram definidas num ficheiro generic.c funções de interesse geral que podem ser executadas em qualquer módulo. Estas funções são:

long get_time();
Retorna o tempo do sistema em *ticks*.

delay_ticks(int);
Provoca uma espera do número de *ticks* desejados.

delay(int);
Provoca uma espera do número de milisegundos desejados.

II - Programa para atravessamento de passagens estreitas

1. Descrição Geral

Depois de realizado o programa para navegação e controlo remoto do robot, foi desenvolvido um trabalho que permite ao robot passar portas ou outras passagens estreitas.

Na sua configuração original, os sensores do robot não lhe permitem vencer uma passagem estreita, já que não conseguem medir distâncias abaixo dos 20 cm. Nessas condições, o robot não poderia passar, com segurança, portas com menos de 110 cm.

O objectivo final proposto foi o de conseguir levar o robot a detectar e passar de forma autónoma, portas ou passagens com mais de 90 cm de largura.

Para vencer a limitação dos sensores cuja medição mínima está próxima dos 20 cm, foi decidido colocar dois sensores de ultra-sons em cima da unidade *Pan & Tilt* do robot. Esses sensores não precisam de medir distâncias inferiores aos 20 cm, já que a largura do robot está próxima dos 30 cm. Assim, dão uma informação valiosa para a passagem de zonas mais apertadas.

Desta forma, a *Pan & Tilt* passou a ser usada como um "radar" permitindo tirar medidas em todas as direcções.

O trabalho foi então dividido em três partes bem distintas:

Leitura dos dados dos dois sensores adicionais;
Detecção e aproximação da porta;
Atravessamento da porta ou passagem.

2. Leitura dos dados dos dois sensores adicionais

Todo o trabalho relativo a leitura dos dados dos dois sensores extras está localizado no módulo **ptu.c**

Como quase todos os outros módulos, este começa por definir duas funções, ***init_ptu()*** invocada pelo ***kernel*** para inicializar as variáveis do processo e abrir a porta série através da qual é feita a comunicação com a *Pan & Tilt* e a função ***close_ptu()*** responsável pelo fecho dessa porta série.

2.1 Sincronização da *Pan & Tilt* com os sensores

A sincronização entre a posição da *Pan & Tilt* e os disparos dos sensores é feita utilizando duas funções, por um lado a função ***ptu()*** e por outro, a função ***read_pt()***.

A função ***ptu()*** é responsável pelo movimento da *Pan & Tilt*. Neste caso, como os dois sensores estão diametralmente opostos, basta um movimento de 180 graus para conseguir medidas em todas as direcções. Nesta função, é portanto testada a posição da *Pan & Tilit*, e no caso desta estar

colocada num dos extremos do movimento, é enviado, com a função ***send2ptu()***, um comando para que ela se movimente até ao outro extremo.

Esta função também está continuamente a enviar o comando PP (pedido da posição da *Pan & Tilt*) e a ler a resposta da linha série, a partir da qual actualiza a posição (**position**) e e o ângulo (**angle**).

Com base no ângulo a função ***read_pt()***, invocada pelo processo ***ptu()***, dispara os ultra-sons se o ângulo for múltiplo de 5, com uma tolerância de 1 grau. Esta tolerância foi escolhida de modo a que com o periodo do processo ***ptu()*** igual a 30 ms, não se perca nenhuma medida ao longo dos 360°.

A velocidade de *Pan & Tilt* foi fixada no valor 1500 para que nos 30 ms o deslocamento seja inferior a 3°. Isto foi calculado tendo em conta que uma posição da *Pan & Tilt*, corresponde a 0,051°

2.2 Dados obtidos

Os primeiros dados que obtivemos apresentavam bastante ruído, como se pode verificar nas figuras 1 e 2 do anexo B. Verificou-se a existência de valores baixos e pontuais sem presença de qualquer obstáculo nessa direcção.

Este problema foi resolvido (cf. figuras 3 a 11 do anexo B), após muitas experiências, ligando a caixa metálica de suporte do sensores, à massa do robot.

Conclui-se assim, que o ruído tinha origem eléctrica provocado pelos pulsos de 400V existentes nos sensores na altura do disparo.

Apesar de consideravelmente melhores, os valores de ultra-sons ainda apresentavam algum ruído, como se pode verificar nas figuras 6 e 11. Isso levou a desenvolver uma função ***filter()*** que realiza uma filtragem dos dados: caso exista um valor bastante baixo no meio de dois altos, este é igualado a média dos dois outros.

Uma analise dos sensores de ultra-sons, revelou que o erro cometido apresenta uma variação aproximadamente linear com a distância. Se for ajustada a velocidade do som para que o valor medido seja exacto para uma dada distância, os valores menores vem afectados de um erro positivo e os maiores de um erro negativo.

Como a precisão era um ponto crucial nesta parte do trabalho, decidimos portanto fixar uma velocidade do som para que a distância medida a 1m seja exacta e depois foi calculado a recta média e usamos esse resultado para corrigir os dados obtidos.

Com uma velocidade do som de 335m/s, obtivemos o gráfico seguinte:

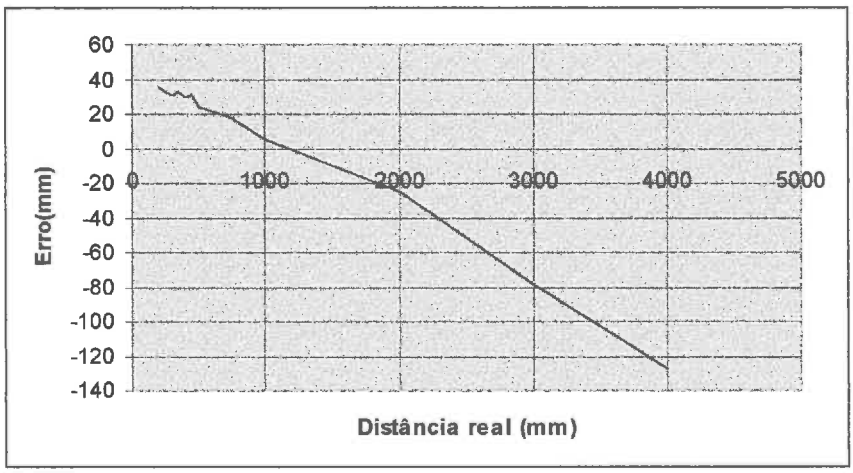


Figura 1

A equação da recta média (valores medidos em função dos valores reais) que obtivemos foi :

$Y = 0.96*X + 43$ onde Y é o valor medido, e X é o valor real.

É com base nesta recta que os valores lido pelos ultra-sons da *Pan & Tilt* são corrigidos na função ***read_pt()***.

Este módulo disponibiliza assim três variáveis globais onde está presente a informação dos sensores da *Pan & Tilt* :

pt_values[MAX_SENS]: obtidos directamente dos sensores.
fvalues[MAX_SENS] : obtidos depois da filtragem pela função ***filter()***.
dvalues[MAX_SENS] : derivada dos valores medidos (diferenças dos valores adjacentes, dois a dois).

3. Módulo de deteção de passagem e alinhamento – GATE.C

3.1 Descrição

A função deste módulo e a detecção de passagem estreita a uma distância até 2m aproximadamente, permitindo que o robot, ao longo desses 2 m, se movimente de modo a que a abordagem da passagem seja o mais alinhada possível.

Este processo implica a existência de duas etapas: detecção de passagem estreita e movimentação para a passagem.

3.2 Algoritmo

Em relação à primeira acção, o algoritmo desenvolvido consiste em detectar as direcções onde os dados dos sensores de ultra-sons apresentam grandes transições positivas, indicando assim a possibilidade de existência de uma passagem. As transições são detectadas analizando a derivada dos valores dos sensores ultra-sons; nas figuras 2 e 3 são apresentados os valores dessa derivada para o caso apresentado no anexo B na figura 8.

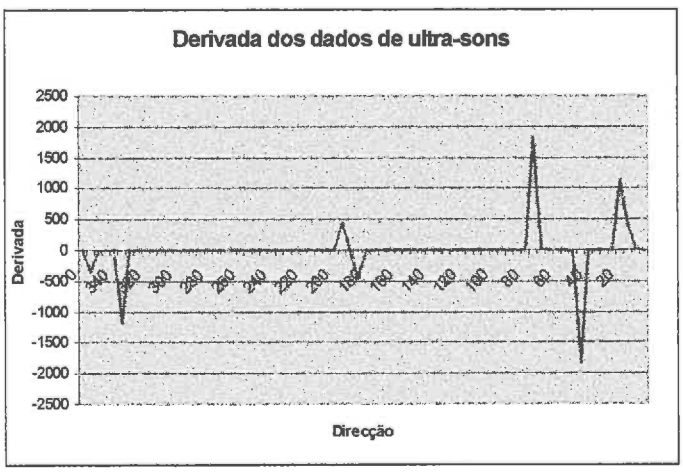


Figura 2

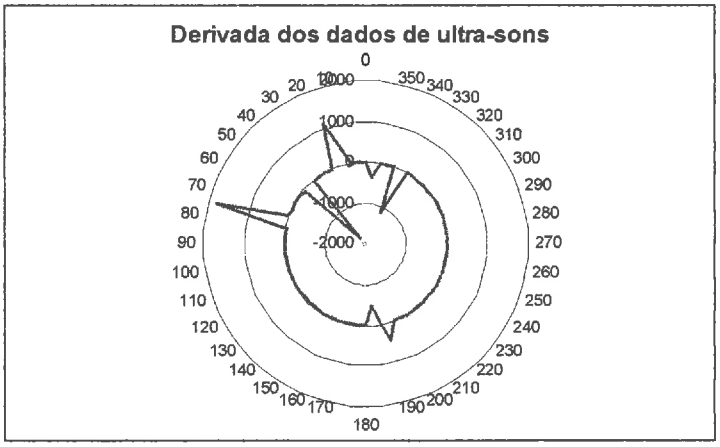


Figura 3

A verificação se se trata ou não de uma passagem é feita analizando o número de valores elevados que estão contidos entre a transição positiva e a transição negativa que se seguirá. Caso este número seja menor que 8, é considerado que se está na presença de uma passagem e nesse caso são identificadas as direcções onde se situam os cantos.

Em relação à movimentação para a passagem, que só é feita caso os cantos da mesma estejam identificados, começa com a orientação do robot segundo o canto que lhe é mais distante, como mostra a figura 4.

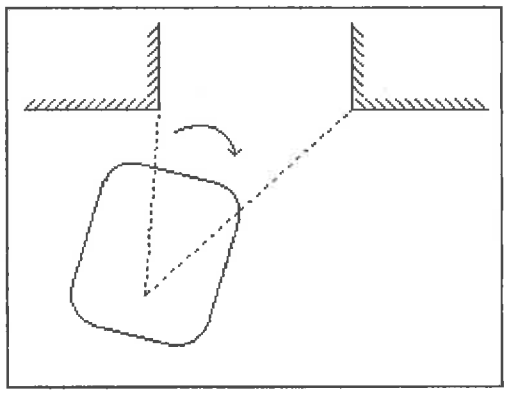


Figura 4

Depois de orientado, o robot dirige-se para esse canto (ver figura 5), até que as distâncias aos cantos sejam aproximadamente iguais, indicando que o robot está centrado na passagem. Nessa situação o robot orienta-se para a passagem, afim de se alinhar, como mostra a figura 6

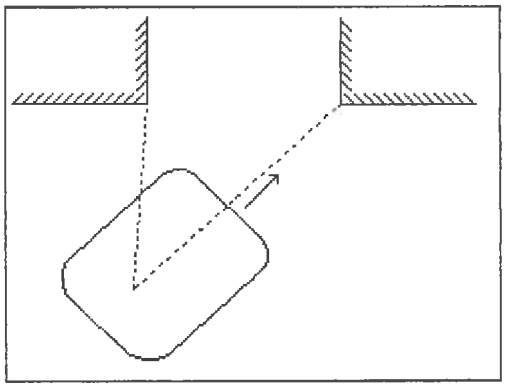


Figura 5

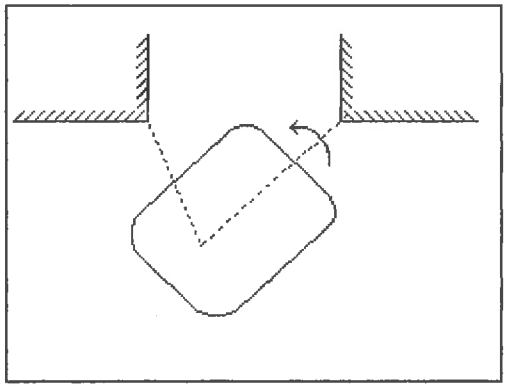


Figura 6

Se depois de alinhado, o robot se encontrar suficientemente afastado, move-se em direção à passagem até ficar mesmo à entrada (figura 7).

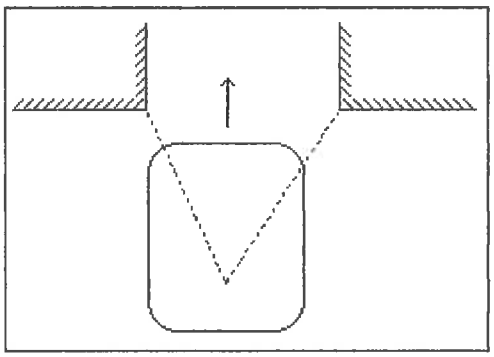


Figura 7

4. Atravessamento da passagem

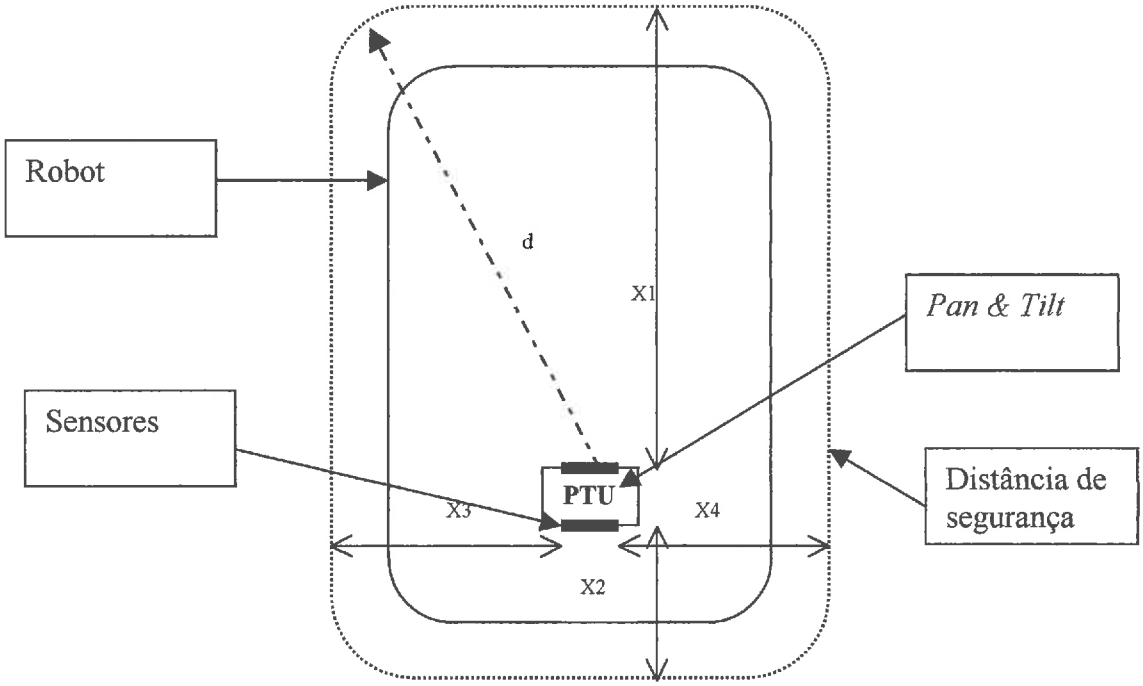
Quando o robot detecta uma porta, a sequência que ele segue é a seguinte: o módulo ***gate()*** detecta a porta e desloca-se até ela. Quando ele considera que já está suficientemente perto, coloca a variável **door** a ON, indicando à função ***doors()*** que a porta já está perto e que agora pode-se entrar no "modo" de atravessamento de portas.

A estratégia usada para o robot passar a porta é a seguinte: para cada ângulo medido, é calculado a distância mínima que o robot pode medir nessa direcção sem bater. Se alguma medida está abaixo desse valor, o robot sabe que está excessivamente perto da porta e portanto vira para o outro lado.

4.1 Cálculo das distâncias de segurança

As distâncias de segurança são calculadas na função ***init_doors()*** invocada pelo kernel antes de lançar o processo.

Estas são calculadas para cada ângulo em que são tiradas medições (de 5 em 5 graus), como é mostrado a seguir.



Os valores X1, X2, X3 e X4, são calculados a partir dos *defines* definidos no ficheiro ***ptu.h***, essas constantes são(entre parenteses estão os valores em milímetros usados com esta configuração):

```
ROBOT_LENGTH : comprimento do robot (1025)
ROBOT_WIDTH : largura do robot (700)
PAN_X_POSITION : distância do centro da Pan & Tilt até à frente do
                 robot (700)
PAN_Y_POSITION : distância do centro da Pan & Tilt até à esquerda do
                 robot (350)
BOX_WIDTH : Largura da caixa onde estão os sensores (corresponde a
            distância existente entre os dois sensores) (100)
SAFETY_DIST : Distância de segurança entre o robot e os obstáculos
              (35)
```

Nessas condições, temos:

```
X1 = PAN_X_POSITION - BOX_WIDTH/2 + SAFETY_DIST
X2 = ROBOT_LENGTH - PAN_X_POSITION - BOX_WIDTH/2 + SAFETY _DIST
X1 = PAN_Y_POSITION - BOX_WIDTH/2 + SAFETY_DIST
X1 = ROBOT_WIDTH - PAN_Y_POSITION - BOX_WIDTH/2 + SAFETY_DIST
```

A distância de segurança **d** da figura é então calculada da seguinte forma:

```
d = X1 / cos (ângulo)
```

Se um dia se opta por uma colocação diferente da *Pan & Tilt,* o módulo continua a funcionar bem a partir do momento onde estes cinco parâmetros estão ajustados.

4.2 Deslocamento dentro da passagem

O comportamento do robot dentro da passagem está definido na função ***doors()***.

Como se optou por considerar a parte mais afastada da *Pan & Tilt* como a parte da frente do robot, só se consideram os valores medidos entre 90 e –90 graus, pois se a parte da frente do robot passa, a parte traseira, menos afastada da *Pan & Tilt*, também passa.

Nesta função, antes de tudo, são desactivadas as emergências associadas aos ultra-sons e as comunicações (só o sensor de contacto - *bumper* - continua activo).

A seguir, o robot usa todos os valores medidos enter 5 e 90 graus (lado esquerdo do robot) e entre 270 e 355 graus (lado direito do robot) e opta entre um dos comportamentos seguintes:

Todas as medidas estão acima da distância de segurança, o robot segue em frente;

Detecção de uma medida inferior a distância de segurança só do lado direito, o robot vira para a esquerda;

Detecção de uma medida inferior a distância de segurança só do lado esquerdo, o robot vira para a direita;

Detecção de medidas inferiores a distância de segurança em ambos os lados, o robot pára e espera cinco segundos (tempo definido no *define* TIME_BEFORE_MOVE) por forma a ter medidas actualizadas em todas as direcções, se ao fim desse tempo, as medidas continuam iguais, avisa que não consegue passar a porta e desliga o modo de passagem de portas.

Detecção de medidas superiores a distância MAX_DIST de um lado ou outro, o robot para desliga o modo de passagem de portas e sinaliza que a porta foi passada.

A velocidade de movimentação do robot, neste modo, é bastante reduzida (da ordem do cm/s), pois os dados da *Pan & Tilt* são obtidos a uma taxa relativamente baixa (um varimento de 360° em aproximadamente 3s).

5. Mensagens adicionadas no módulo serial.c

Ao longo este trabalho, foi necessário acrescentar algumas mensagens para poder utilizar os vários módulos desenvolvidos, estas mensagens foram:

Mensagens de pedido de informação

#Define	Char	Descrição e exemplo	Exemplo de resposta
IN_PTU_US	'P'	Pedido os dados da *Pan & Tilt* **BP**	P 2491 2491 ... 750 162341
IN_PTU_FUS	'F'	Pedido dos dados filtrados da *Pan & Tilt* **BF**	U 2491 2491 ... 750 162341
IN_PTU_DUS	'D'	Pedido das derivadas do valores medidos na *Pan & Tilt* **BV**	V 0 0 200 –300 ...750 162341

Mensagens unidireccionais

#Define	Char	Descrição e exemplo
IN_PTU_ON	'p'	Ligar *Pan & Tilt* **Ap**
IN_PTU_OFF	'g'	Desligar *Pan & Tilt* **Ag**
IN_GATE	'G'	Ligar módulo de busca de porta **AG**
IN_END_GATE	'T'	Desligar o módulo de busca de porta **AT**
IN_DOOR	'd	Ligar modo de passagem de portas **Ad**
IN_DOOR_OFF	'f'	Desligar modo de passagem de portas **Af**

Programa de demonstração

A. Descrição Geral

A.1 Introdução

Após o desenvolvimento do programa de navegação e comunicações para o robot, foi nos proposto desenvolver um pequeno programa de demonstração afim de realçar algumas das potencialidades do robot e do *software* desenvolvido até então, no âmbito deste projecto.

Com esta demonstração mostrou-se principalmente como inserir novos módulos fazendo uso dos módulos já existentes.

A.2 Objectivos

O programa de demonstração tem como função fazer seguir o robot por um dado percurso repetidamente e caso seja detectado um obstáculo a sua frente deverá parar até que o obstáculo seja removido.

A movimentação do robot é feita com base na odometria, o que acarreta alguns problemas pois os erros cometidos pelos odómetros são cumulativos.

Se o programa funcionasse em malha aberta, isto é, se periodicamente não fosse verificada a posição real em relação a posição desejada do robot, é bem provável que este se desviasse rapidamente da trajectória pretendida. Assim, foi desenvolvida uma forma de calibrar periodicamente o robot, garantindo que em todos os ciclos sejam corrigidos os erros de odometria.

A.3 O módulo dem.c

O módulo **dem.c** contém todas as funções responsáveis por: imposição da trajectória pré-definida; deteção de obstáculos; calibração.

A.3.1 Percurso realizado

As principais razões que levaram a escolha do percurso foram, a necessidade de ter curvas bastante abertas (por forma a minimizar os erros de odometria) e de ter uma zona de auto-calibração. Tentou-se realizar uma larga gama de movimentos para tornar a demonstração mais atractiva, embora limitados pela disposição espacial do próprio laboratório onde a demostrção se realizou.

De seguida, na figura 1, é apresentado o percurso que foi programado nesta demonstração e que tenta cumprir as especificações referidas acima.

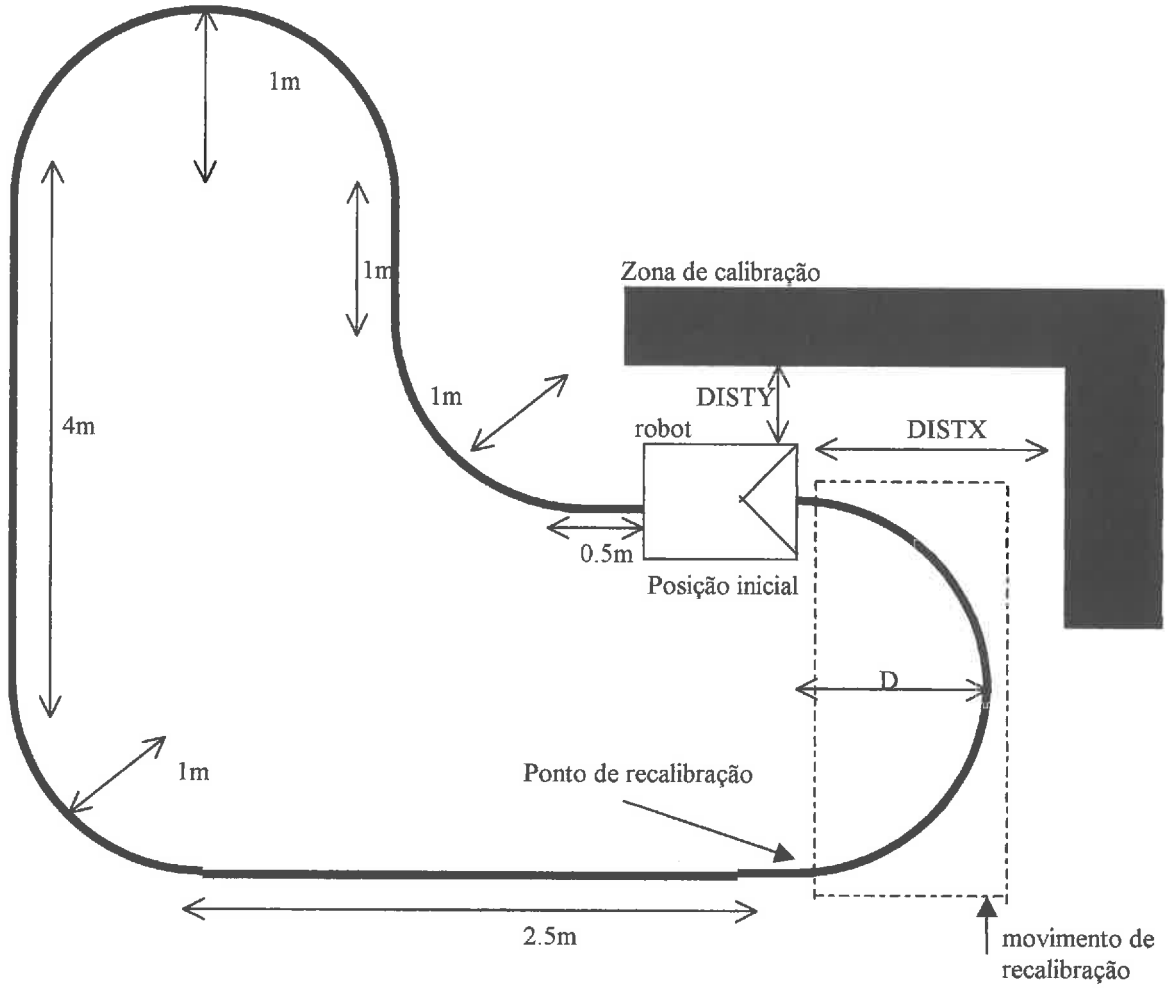


Figura 1

A.3.2 Odometria

O módulo que foi desenvolvido está dividido em duas partes bem distintas, o ***main()*** encarregue de toda a parte de odometria invocando só no início do movimento a função de auto-calibração para calibrar o robot antes de um novo ciclo.

A parte de odometria foi pensada como uma máquina de estados, em que cada estado corresponde a um movimento do robot. Quando o robot atinge um ponto em que o movimento muda, a máquina de estados evolui para o estado seguinte.

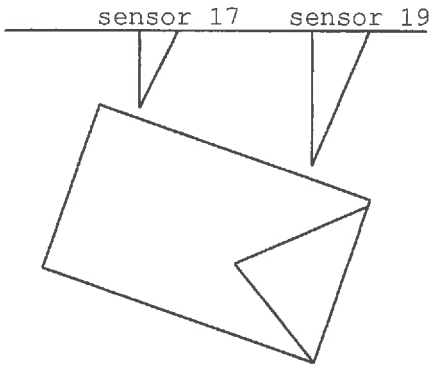
A.3.3 Auto-calibração

A auto-calibração é feita na função ***orientação()*** que o módulo invoca antes de iniciar um novo ciclo. A auto-calibração utiliza pontes de referência para confirmar a posição real do robot. Nesta demonstração, foram utilizados caixotes colocadas na zona de calibração com mostra a figura 1.

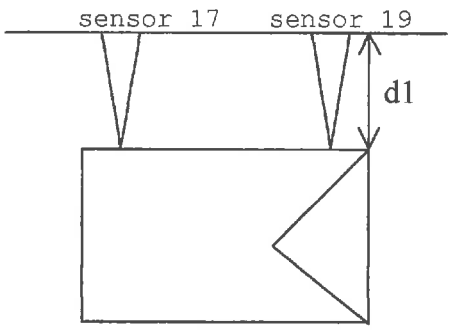
O robot ao chegar a zona de calibração vai comparar as distâncias medidas até aos objectos e os valores da odometria conseguindo assim corrigir os erros que foi acumulando ao longo do movimento.

O processo de auto-calibração divide-se em três fases:

1 - Numa primeira fase, o robot realiza um movimento sobre ele próprio até que os valores lidos pelos sensores 17 e19 sejam idênticos o que lhe garante que ele está orientado segundo os obstáculos do seu lado esquerdo.

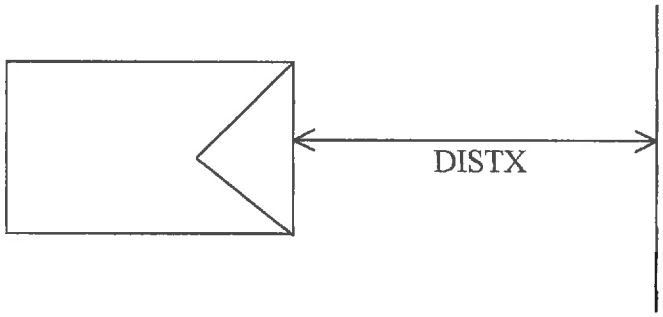


Robot não orientado



Robot orientado

2 - Na segunda fase da calibração, o robot deslocar-se-á até a uma distância DISTX (definida no módulo **dem.h**) do obstáculo a sua frente o que lhe permite corrigir a sua odometria no eixo dos x.



3 - Por fim, o robot realiza um semí-círculo de raio D, onde D è dado por:

$$D = \frac{2 * RAIO + DISTY - ERRO}{2}$$

onde RAIO, é o raio do círculo se não houver erro no y (aqui é 1m)
DISTY, é a distância normal ao obstáculo sem erro (aqui 1m)
ERRO é a distância medida pelos sensores até ao obstáculo.

Ao fim deste semi-círculo, o robot chega ao ponto de auto-calibração que será o mesmo para cada calibração, seja qual for o erro com que ele chegue a zona de auto-calibração. É nesse ponto que os valores de odometria estão inicializados, dando-se início a um novo ciclo.

Esta calibração em três fases permite corrigir a postura do robot (posição X, posição Y e orientação).

Por forma a permitir adaptar o programa a outras situações, é de notar que as distâncias aos objectos e o raio do círculo na zona de calibração podem ser ajustados permitindo portanto modificar a configuração da zona de calibração.

A distância DISTX, têm de ser cuidadosamente escolhida por forma a garantir que o robot possa realizar a circunferência de raio D sem parar por causa do objecto a sua frente.

A.3.4 Paragem em frente de obstáculos

A parte de paragem em frente dos obstáculos já é feita pelo módulo de emergência pelo que o módulo de demonstração limita-se a colocar nas variáveis mov.left e mov.right (que pertencem ao executor de movimentos) as velocidades pretendidas e activar a flag emerg.rtm do módulo de emergências. No caso de não haver obstáculo, o movimento será executado pelo executor, caso contário, o módulo de emergência não o permitirá.

A.4 Conclusões

Conseguiu-se com este programa uma demonstração bastante simples e fiável pois a auto-calibração evita a acumulação dos erros de odometria de uma passagem para a outra.

Este exercício, também mostra a simplicidade com que se pode acrescentar novas tarefas ao programa base aproveitando todo o trabalho já feito.

Por fim, esta demonstração mostra que é possível fazer uso da odometria para a movimentação do robot, bastando desenvolver técnicas de navegação que se apoiam sobre o conhecimento de alguns pontos de calibração nos quais o robot pode usar as medidas dos ultra-sons para determinar a sua exacta posição.

ANEXO B – *Dados obtidos com os sensores da Pan & Tilt*

Dados obtidos sem ligação da caixa metálica a massa do robot

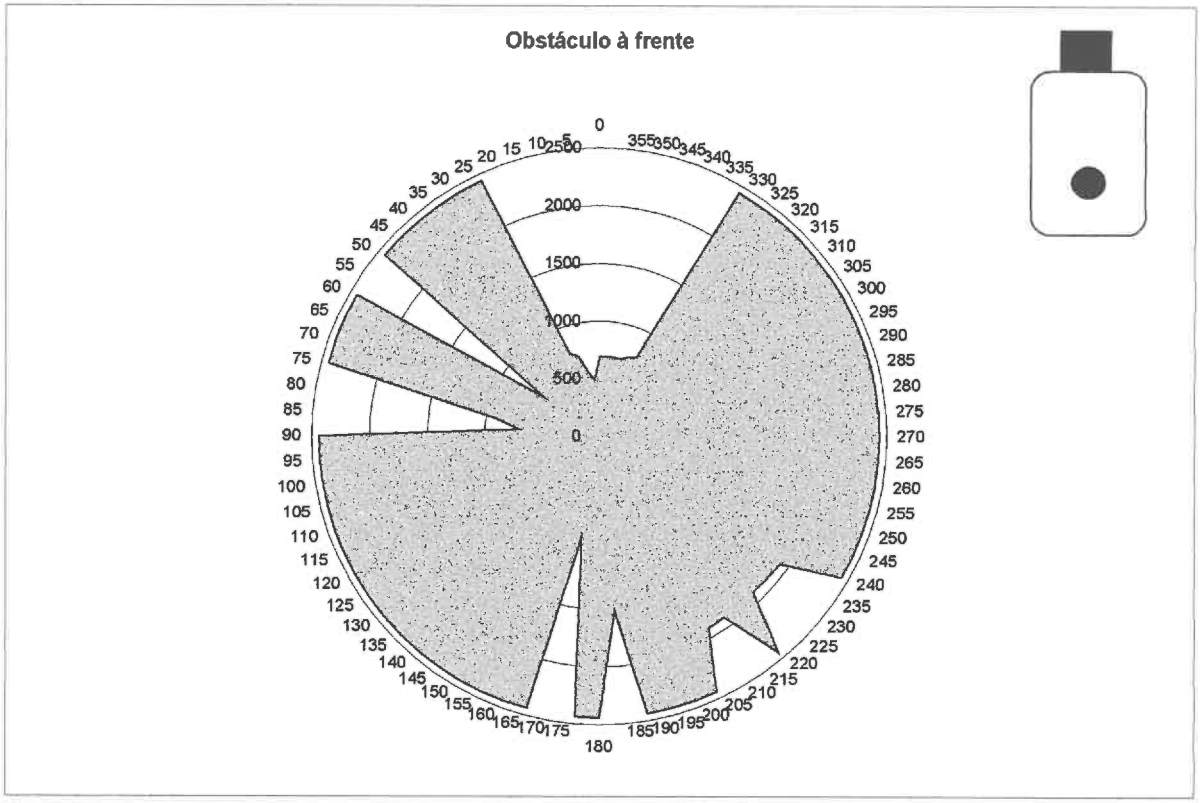


Figura 1

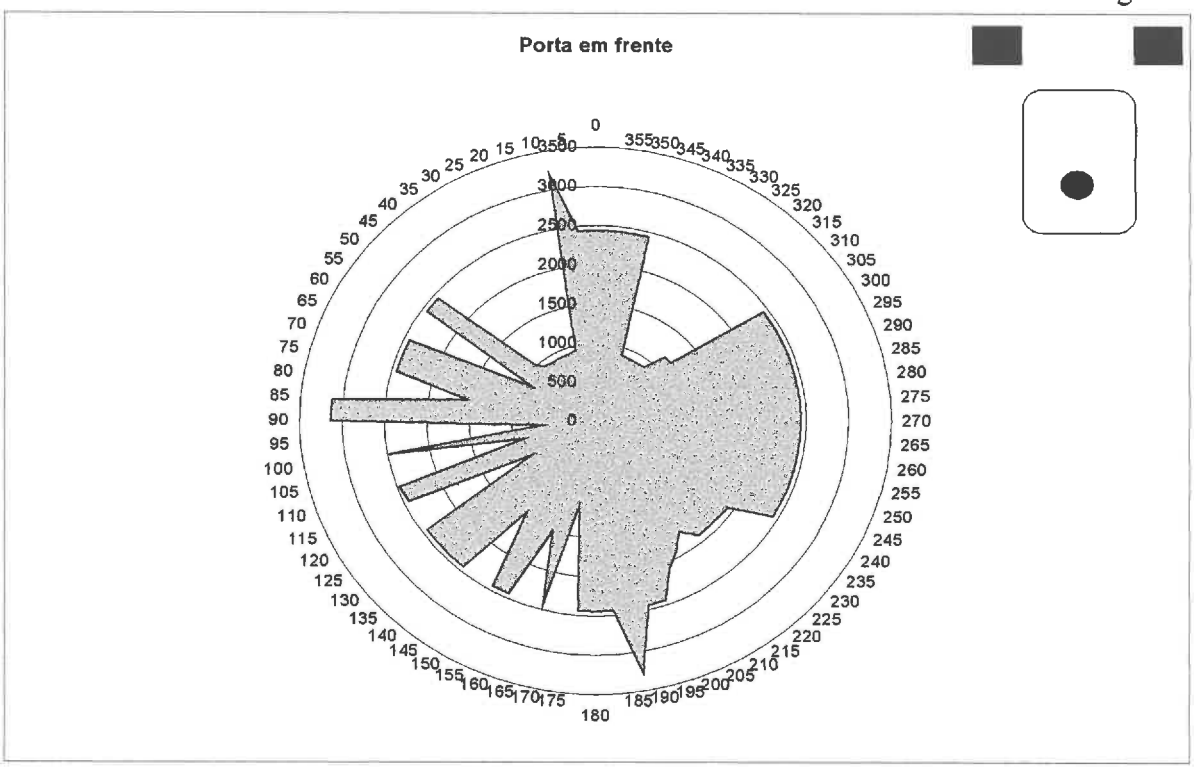


Figura 2

Dados obtidos obtido com ligação da caixa metálica a massa do robot

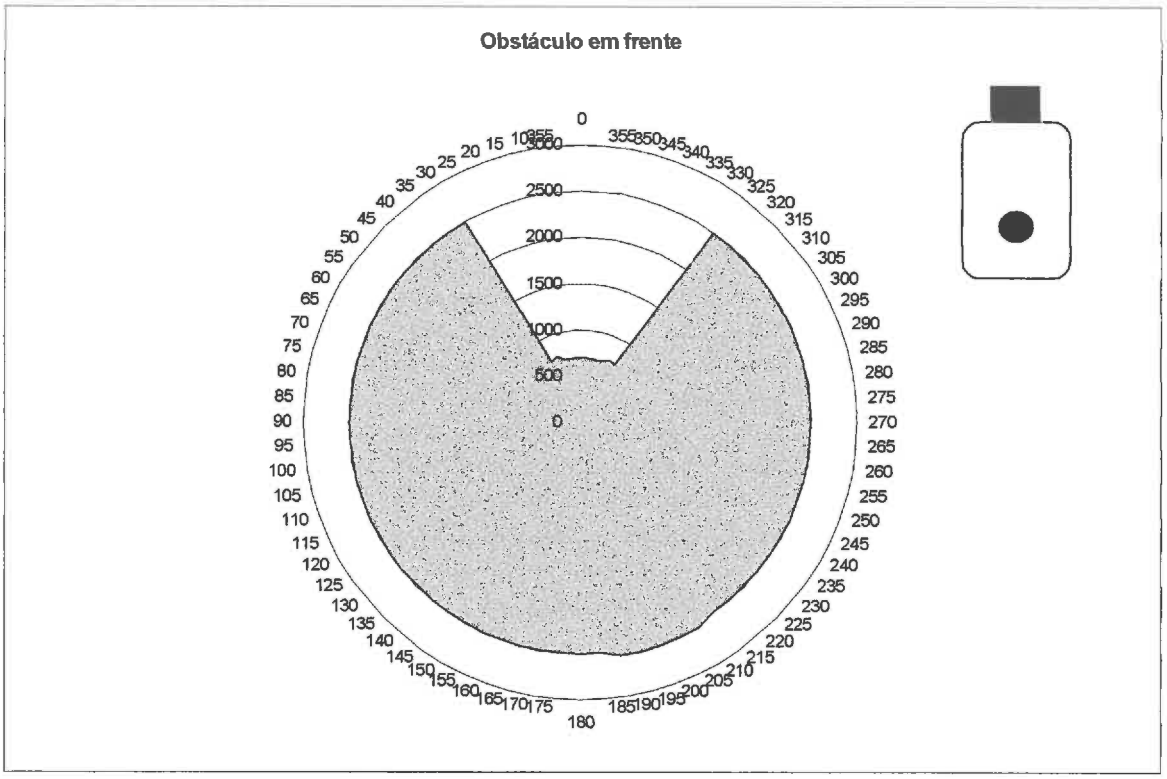


Figura 3

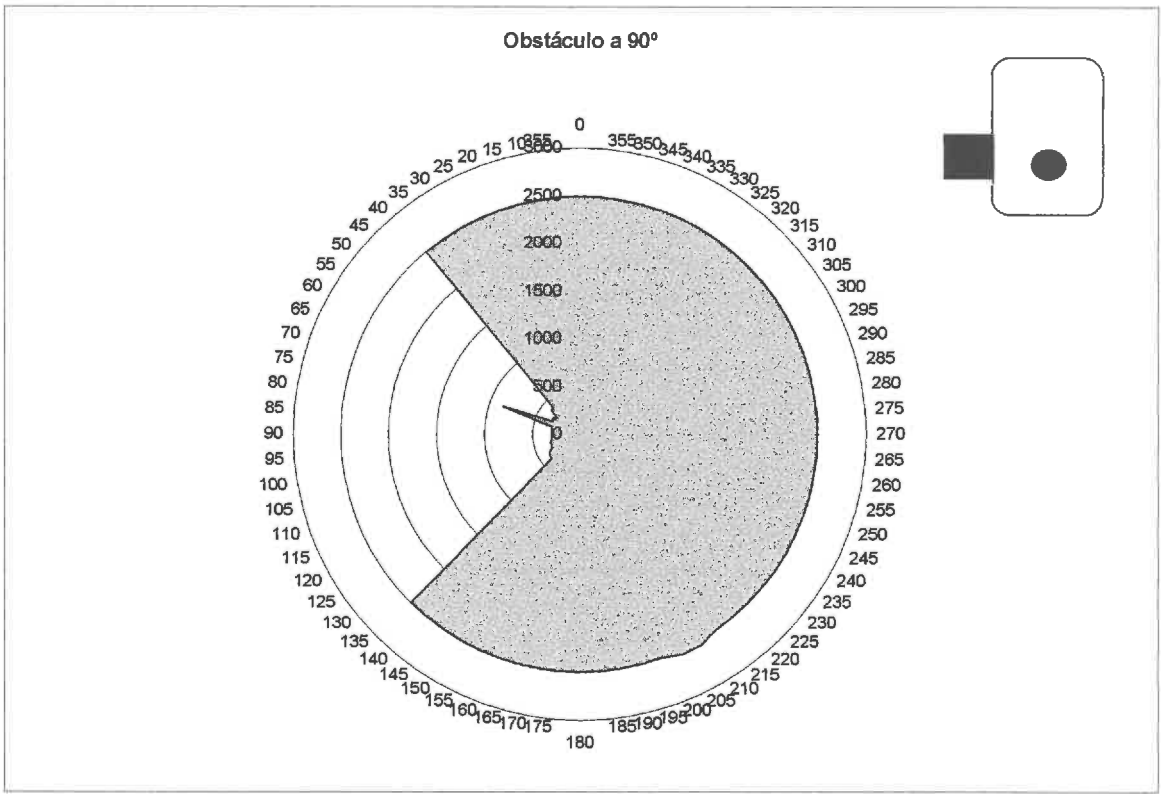


Figura 4

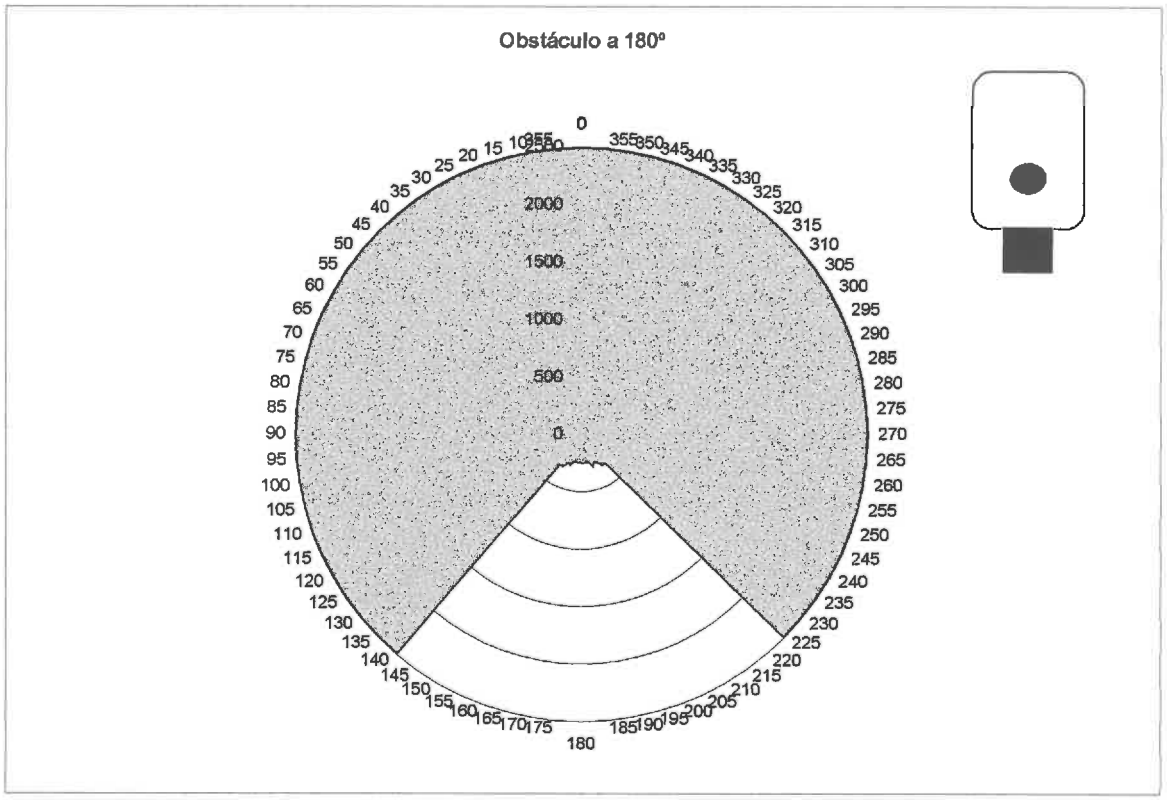


Figura 5

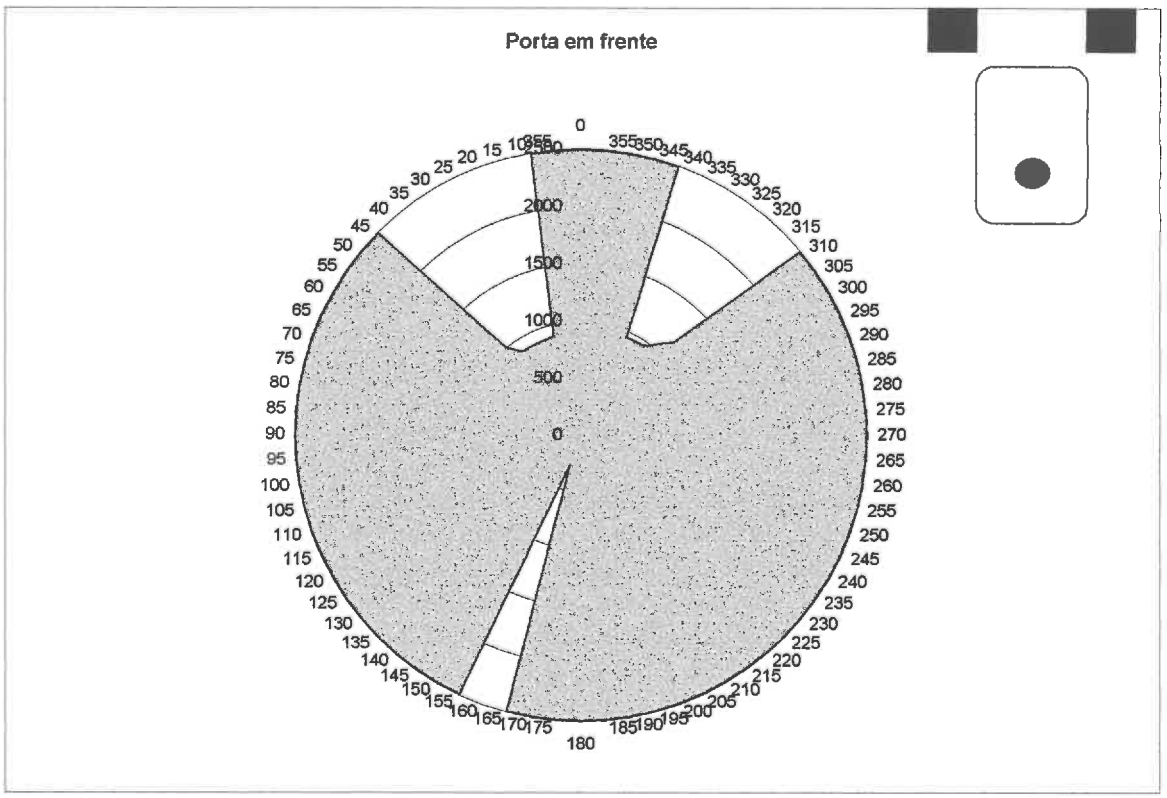


Figura 6

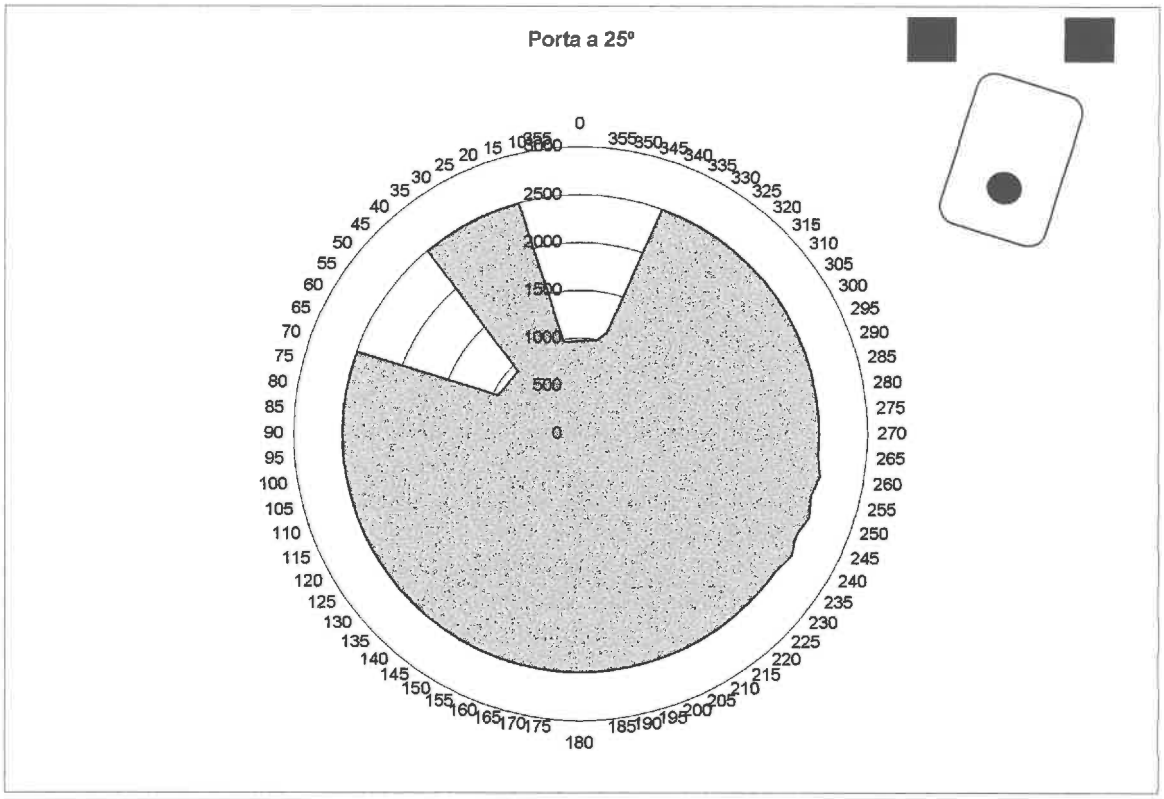


Figura 7

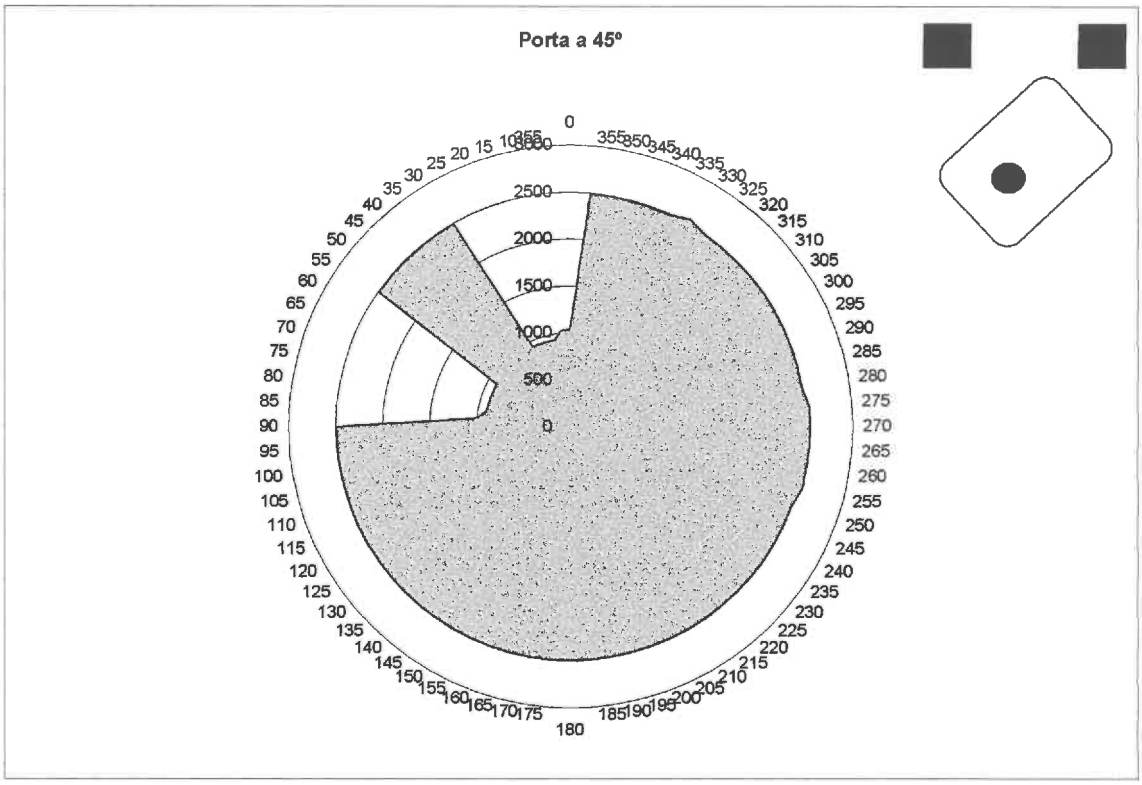


Figura 8

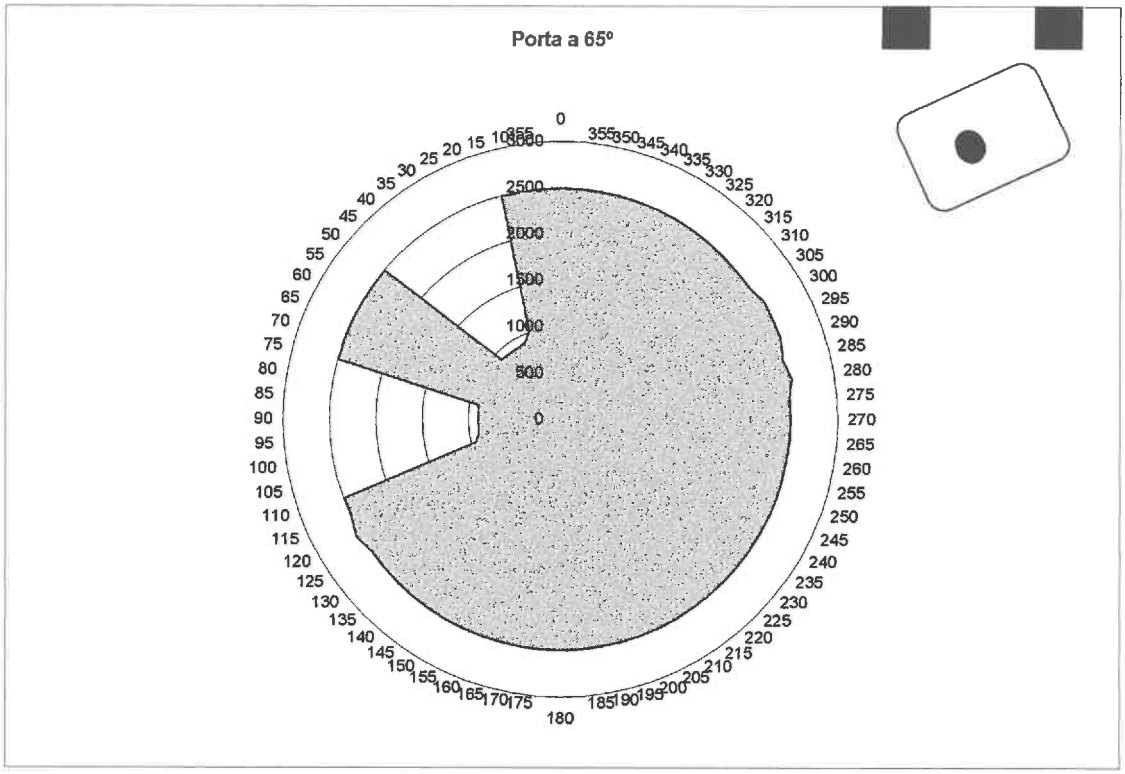


Figura 9

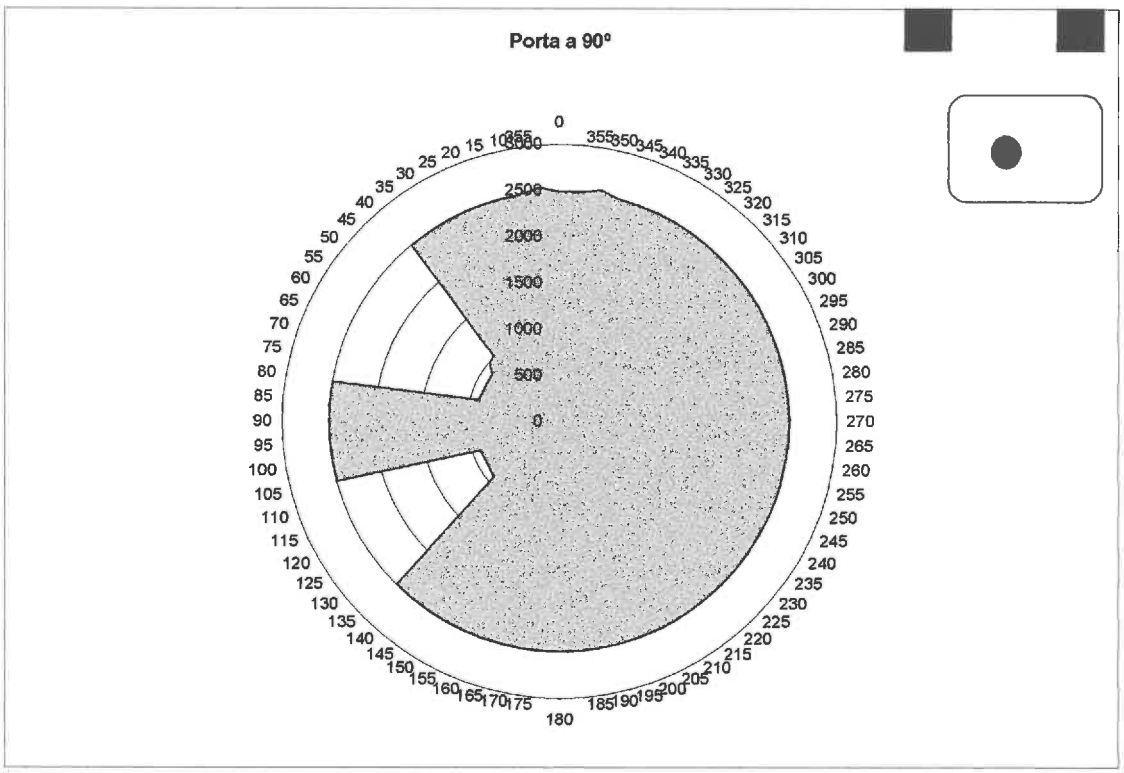


Figura 10

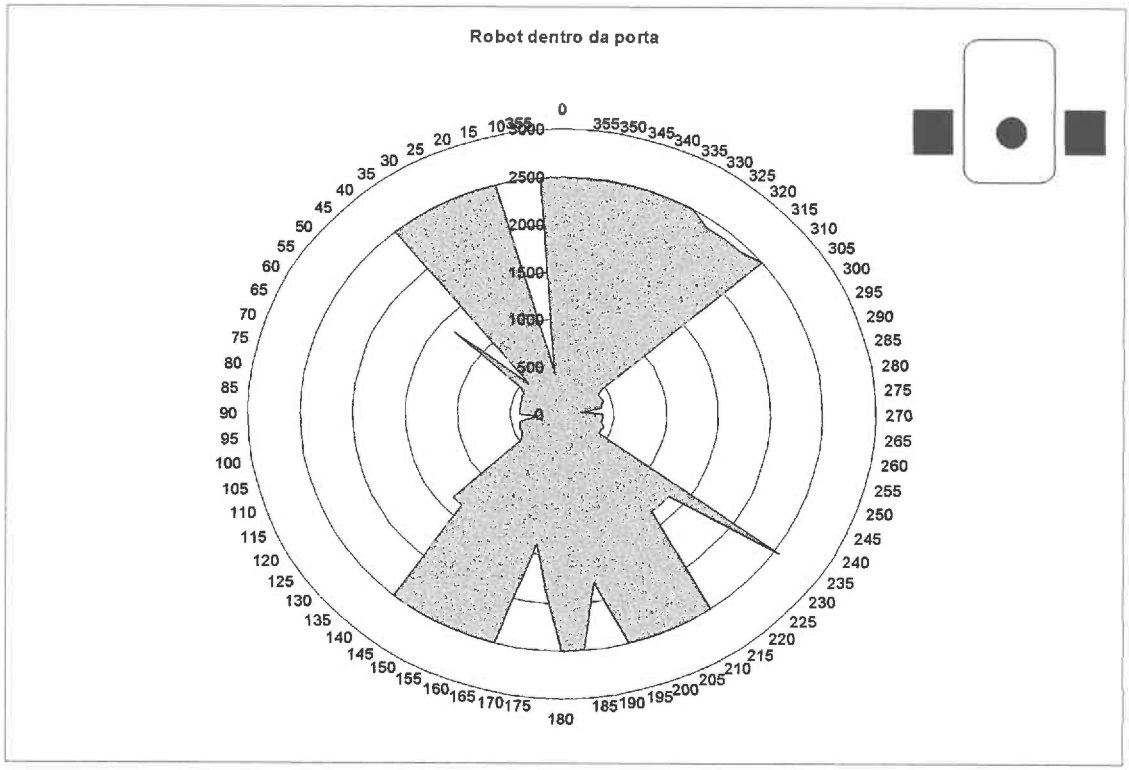

Robot dentro da porta
0
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175 180 185 190 195 200 205 210 215 220 225 230 235 240 245 250 255 260 265 270 275 280 285 290 295 300 305 310 315 320 325 330 335 340 345 350 355
3000
2500
2000
1500
1000
500
0

Figura 11

ANEXO I.1 - *headerfiles*

serial.h

```
#define ERROR       -1
#define OK           1

#define MAX_LENGTH 200       /* Maximum message length */

#define SEPARATOR1   32
#define SEPARATOR2   44
#define TERMINATOR   13

#define MSGC_DATAQUERY  'B'
#define IN_POST         'O'
#define IN_US           'U'
#define IN_POSTUS       'V'
#define IN_CLOCK        'C'
#define IN_USTOUT       'T'

#define MSGC_UNIDIRCOM  'A'
#define IN_VELOCITY     'M'
#define IN_POSITION     'W'
#define IN_ODOM         'C'
#define IN_SERV         'S'
#define IN_ALBATROS     'Y'
#define IN_KILL         'X'

#define MSGC_USACTION   'U'
#define IN_USTABLE      'T'
#define IN_NDELAY       'D'
#define IN_GNDELAY      'N'
#define IN_STAT         'S'
#define IN_SUSTOUT      't'
#define IN_TOGSENS      'u'

typedef struct {
  char buff_in[MAX_LENGTH];
  char buff_out[MAX_LENGTH];
  int desc, send;
} SERIAL;


/* Performs all actions related with received message */
void decode(void);

int init_serial_spy(void);

void close_serial_spy(void);

/* Process that "spys" the serial line */
void serial_spy(void);
```

us.h

```
#define MAXSENS          24
#define DEFAULT_N_DELAY   3
#define DEFAULT_TIME_OUT 33
#define NOT_ACTIVE        0
#define ACTIVE            1


typedef struct {
  int change_flag;
  int active_sensors[24];
  int node_delay;
```

```
/* Add two sensor numbers */
int add24(int, int);


/* Returns bumper state */
int get_bumper_state(void);

/* stop in case of emergency */
void emergency_stop(void);


/* Returns main sensor given the actual orientation of motion */
int get_main_sensor(void);


/* Make all initializations in data structs used by emergecy process */
void init_emergency(void);


/* Emergency process */
void emergency(void);
```

executor.h

```
typedef struct {
  long int x;
  long int y;
  long int theta;
} POSTURE;


typedef struct {
  int status;
  int left;
  int right;
  int flag;
} MOVES;

void init_execute(void);

/* Execute all actions related with motion */
void execute(void);

/* Execute serv of before leaving program*/
void close_execute(void);
```

generic.h

```
#define FALSE 0
#define TRUE  -1

/* Returns system time */
long get_time(void);


/* Delay (ticks) */
void delay_ticks(long);


/* Delay (miliseconds) */
void delay(int);
```

```
  int time_out;
  long values[24];
  long node_times[6];
  struct {
        long node_mean_time;
    long scan_time;
  } stat;
} US_DATA;


/* Active Sensors Table - AST */

typedef struct {
  int node[6];    /* Contains the number of active sensores in a node */
  char sens[6][7];
} ACTIVE_TABLE;


/* Gets sensor number (1..24) given Node_number and Number_in_node */
int get_sensor_number(int Node_number, int Number_in_node);


/* Performs all measures for all active sensors in the given Node_number */
void read_node(int Node_number);

void init_read_sensors(void);

/* Performs all reading for all active sensors on AST */
void read_sensors(void);



int get_sensor_node(int);
```

emerg.h

```
/* #define US_LIMIT      1100     Flight time in microseconds */

/* #define US_LIMIT      220    Distance in mm */

#define COMM_LIMIT    18000   /* 3 minute(s) */

#define NOT_PRESSED   1       /* Bumper state */
#define PRESSED       0

#define DBW  600   /* Distance between wheels */

#define LIMIT12   2476
#define LIMIT23   1200
#define LIMIT34   1011
#define LIMIT45   864
#define LIMIT56   710

#define NONE 0
#define BUMPER 1
#define US 2
#define COM 4

typedef struct {
  int flag;      /* Indicates an emergency execption */
  int rtm;       /* Request To Move */
  int us_change; /* Request to change AST */
  long time_stamp;
  int auto_toggle;
  int on;
} EMERGENCY;

/* Subtract two sensor numbers */
int sub24(int, int);
```

ANEXO I.2 - Código

serial.c

```
#include <libcstd.h>
#include <libport.h>
#include <libcalbl.h>
#include <libcpro.h>

#include "generic.h"
#include "us.h"
#include "executor.h"
#include "serial.h"
#include "emerg.h"
#include "headers.h"

#define PORT "SLB"

static SERIAL serial;

extern int kill;
extern US_DATA sens;
extern POSTURE pos;
extern MOVES mov;
extern EMERGENCY emerg;


void decode(void) {
  char category, type, aux[50];
  char param[3][30], *token, *tmp;
  int p, n_param;

  if (strlen(serial.buff_in)<=2) {      /* Se existem menos de dois caracteres */
    strcpy(serial.buff_out,"error\r");
    serial.send=TRUE;                   /*   envia msg de erro */
        return;
  }

  token=serial.buff_in;

  category=*token;      /* Retira a CATEGORIA */
  token++;
  type=*token;          /* Retira o TIPO */
  token++;

  p=0;                              /* Retirar parametros, */
  while (*token!=TERMINATOR) {      /*    se existirem.    */
    tmp=aux;
    do {
      *tmp=*token;
      tmp++; token++;
    } while( (*token!=TERMINATOR) && ( ( (*token!=SEPARATOR1) && (*token!=SEPARATOR2) ) || (type==IN_ALBATROS) )
);
    *tmp='\0';
    strcpy(param[p],aux);
    if(*token!=TERMINATOR)
         token++;
       p++;
    n_param=p;
  }

/*  for(p=0;p<n_param;p++)
       printf("\nparam[%d]= %s",p+1,param[p]);*/

  switch(category) {
    case MSGC_DATAQUERY:
      switch(type) {
        case IN_POST:
          sprintf(aux,"S %ld %ld %ld %d %ld\r",pos.x,pos.y,pos.theta,mov.status,get_time());
          strcpy(serial.buff_out,aux);
```

```
        serial.send=TRUE;
        break;
      case IN_US:
        strcpy(serial.buff_out,"U");
        for (p=0; p<24; p++) {
          sprintf(aux," %ld",sens.values[p]);
          strcat(serial.buff_out,aux);
        }
        sprintf(aux," %d %ld\r",mov.status,get_time());
        strcat(serial.buff_out,aux);
        serial.send=TRUE;
        break;
      case IN_POSTUS:
        sprintf(serial.buff_out,"V %ld %ld %ld %d",pos.x, pos.y, pos.theta, mov.status);
        for (p=0; p<24; p++) {
          sprintf(aux," %ld",sens.values[p]);
          strcat(serial.buff_out,aux);
        }
        sprintf(aux," %ld\r",get_time());
        strcat(serial.buff_out,aux);
        serial.send=TRUE;
        break;
      case IN_CLOCK:
        sprintf(serial.buff_out,"C %ld\r",get_time());
        serial.send=TRUE;
        break;
      case IN_USTOUT:
        sprintf(serial.buff_out,"T %d\r", sens.time_out);
        serial.send=TRUE;
              break;
      default:
        strcpy(serial.buff_out,"error\r");
        serial.send=TRUE;
        return;
    }
    break;

  case MSGC_UNIDIRCOM:
    switch(type) {
      case IN_VELOCITY:
        if (n_param!=2) {
          strcpy(serial.buff_out,"error\r");
          serial.send=TRUE;
          return;
        }
        mov.left=atoi(param[0]);
        mov.right=atoi(param[1]);
        emerg.rtm=TRUE;
        strcpy(serial.buff_out,"ack\r");
        serial.send=TRUE;
        break;
      case IN_SERV:
        if ( (strcmp(param[0],"F")!=0) && (strcmp(param[0],"N")!=0) ) {
          strcpy(serial.buff_out,"error\r");
          serial.send=TRUE;
          return;
        }
        if(strcmp(param[0],"N")==0)
          mov.status=1;
        else
          mov.status=0;
        strcpy(serial.buff_out,"ack\r");
        serial.send=TRUE;
        break;
      case IN_ODOM:
              sendf("ODOS C=%d,%d,%d\r",atoi(param[0]),atoi(param[1]),atoi(param[2]));
        strcpy(serial.buff_out,"ack\r");
        serial.send=TRUE;
              break;
      case IN_ALBATROS:
        sendf("%s",param[0]);
        strcpy(serial.buff_out,"ack\r");
        serial.send=TRUE;
        break;
```

```
      case IN_KILL:
        strcpy(serial.buff_out,"ack\r");
        serial.send=TRUE;
              s_close(serial.desc);
        kill=TRUE;
        break;
      default:
        strcpy(serial.buff_out,"error\r");
        serial.send=TRUE;
        return;
    }
    break;

  case MSGC_USACTION:
    switch(type) {
      case IN_USTABLE:
        if (strlen(param[0])<24) {
          strcpy(serial.buff_out,"error\r");
          serial.send=TRUE;
          return;
        }
        token=param[0];
        for (p=0; p<24; p++) {
          sens.active_sensors[p]=*token-48;
          token++;
        }
        emerg.us_change=TRUE;
        strcpy(serial.buff_out,"ack\r");
        serial.send=TRUE;
        break;
      case IN_NDELAY:
        if (strlen(param[0])==0) {
          strcpy(serial.buff_out,"error\r");
          serial.send=TRUE;
          return;
        }
        sens.node_delay=atoi(param[0]);
        strcpy(serial.buff_out,"ack\r");
        serial.send=TRUE;
        break;
      case IN_GNDELAY:
        sprintf(serial.buff_out,"N %d\r",sens.node_delay);
        serial.send=TRUE;
        break;
      case IN_STAT:
              if (*param[0]=='N') {
                   sprintf(serial.buff_out,"S %ld\r",sens.stat.node_mean_time);
              }
              else if (*param[0]=='S') {
                   sprintf(serial.buff_out,"S %ld\r",sens.stat.scan_time);
              }
              else
        strcpy(serial.buff_out,"error\r");
        serial.send=TRUE;
              break;
      case IN_SUSTOUT:
              sens.time_out=atoi(param[0]);
              sprintf(serial.buff_out,"ack\r");
              serial.send=TRUE;
              break;
      case IN_TOGSENS:
        if (strlen(param[0])==0) {
          strcpy(serial.buff_out,"error\r");
          serial.send=TRUE;
          return;
        }
              if (param[0][0]=='0')
                emerg.auto_toggle=FALSE;
        else if (param[0][0]=='1')
                 emerg.auto_toggle=TRUE;
        else {
          strcpy(serial.buff_out,"error\r");
          serial.send=TRUE;
          return;
```

```
          }
                sprintf(serial.buff_out,"ack\r");
                serial.send=TRUE;
                break;
        default:
          strcpy(serial.buff_out,"error\r");
          serial.send=TRUE;
          return;
      }
      break;
    default:
      strcpy(serial.buff_out,"error\r");
      serial.send=TRUE;
      return;
  }
}


int init_serial_spy(void) {

  init_port();

  if ( (serial.desc=s_open(PORT,O_RDWR,P2NOECHO)) <= 0) {
    printf("\nERROR opening serial port!\n\r");
    return(ERROR);;
  }

  emerg.time_stamp=get_time();
  printf("\nSerial_spy started.");
  return(OK);
}


void close_serial_spy(void) {

  s_close(serial.desc);
  kill_port();
}

void serial_spy(void) {
  static int first_time=TRUE;
  static char *token;
  static char ch;
  int size;

  if (first_time) {
    token=serial.buff_in;
        first_time=FALSE;
  }

  size=s_read(serial.desc,&ch,1);
  if (size==0)
        return;
  else if (ch!=TERMINATOR) {
    printf("%c",ch);
    *token=ch;
    token++;
        return;
  }

  *token=TERMINATOR;
  token++;
  *token='\0';
  emerg.time_stamp=get_time();
  decode();
  token=serial.buff_in;
  printf("\n\r->");

  if (serial.send) {
    size=strlen(serial.buff_out);
    s_write(serial.desc,serial.buff_out,size);
    serial.send=FALSE;
  }
}
```

us.c

```
#include <libcrobu.h>
#include <libcalbl.h>
#include "us.h"

#include "generic.h"
#include "headers.h"

US_DATA sens;
static us_conf_desc *us_conf;
static ACTIVE_TABLE table;

int get_sensor_node(int number) {
  return((us_conf+number-1)->sensor/10);
}

int get_sensor_number(int node,int number)
{
  int j=0;

  while ( (us_conf+j)->sensor != node*10+number )
    j++;
  return(j+1);
}

void read_node(int node)
{
  static long temp[6][6];
  static int last_node=0;
  static int first_time=TRUE;
  long ret;
  int i;

  if (last_node!=node) {
    last_node=node;
    delay_ticks(sens.node_delay);
  }

  if (first_time==TRUE) {
    sendf("READ C=%d,%s O=%lx\r",node,table.sens[node-1],temp[node-1]);
    first_time=FALSE;
  }

  do {
    ret=sendf("READ C=%d,%s O=%lx\r",node,table.sens[node-1],temp[node-1]);
    sens.node_times[node-1]=get_time();
  } while ( ret!=0);

  for (i=0; i<4; i++) {
    sens.values[get_sensor_number(node,i+1)-1]=temp[node-1][i];
  }
}

void init_read_sensors(void) {
  int i, j;
  sens.stat.node_mean_time=0;


  /* Initialize AST (defined in US.H) */
  for (i=0;i<6;i++) {
    table.node[i]=0;
    for (j=0;j<6;j++)
      table.sens[i][j]='0';
    table.sens[i][6]='\0';
  }

  for (i=0; i<24; i++) {              /* Make all sensors active */
    sens.active_sensors[i]=1;
    sens.values[i]=0;
  }
  sens.change_flag=TRUE;              /* TRUE if the 'active_sensors' array was changed */
```

```
  sens.node_delay=DEFAULT_N_DELAY;     /* Default node firing delay */
  sens.time_out=DEFAULT_TIME_OUT;      /* Default time-out */

  send("SSET S=1 N=13\r");             /* Because sensors 2 and 24 positions */
  send("SSET S=23 N=52\r");            /*   was changed                      */

  send("READ M=S R=D T=33 V=300\r");  /* Single-shot mode, timeout of 33 ms and return distance */
  printf("\nSingle-shot Mode Start.");
  us_conf = get_us_conf(1);
}

void read_sensors()
{
  int node,number,i;
  long start, end;
  long scan_start, scan_end;
  int scan, ret;
  static total_read_time=0;
  static numb_read=0;
  static int last_time_out=DEFAULT_TIME_OUT;

  /* Fill in AST, if necessary */
  if (sens.change_flag==TRUE) {
    for (i=0;i<24;i++) {
      node=(us_conf+i)->sensor/10;
      number=(us_conf+i)->sensor-10*node;
      if (sens.active_sensors[i]!=table.sens[node-1][number-1]-48){
        if (sens.active_sensors[i]==ACTIVE) {
          table.node[node-1]++;
          table.sens[node-1][number-1]='1';
        }
        else {
          table.node[node-1]--;
          table.sens[node-1][number-1]='0';
          sens.values[i]=0;
        }
      }
    }
    sens.change_flag=FALSE;
  }

  if (sens.time_out!=last_time_out) {
        do {
      ret=sendf("READ M=S R=D T=%d V=300\r", sens.time_out);
        } while (ret!=0);
        last_time_out=sens.time_out;
  }

  /* Read nodes, if necessary */

  /* Readings without statistical data */
  /*
  for (i=1;i<=6;i++)
    if (table.node[i-1] > 0)
      read_node(i);
  */

  /* Readings with statistical data */
  scan=0;
  scan_start=get_time();
  for (i=1;i<=6;i++) {
    if (table.node[i-1] > 0) {
      scan++;
      start=get_time();
      read_node(i);
      end=get_time();
      numb_read++;
      total_read_time=total_read_time+end-start;
      sens.stat.node_mean_time=total_read_time/numb_read;
    }
  }
  scan_end=get_time();
  if (scan==6)
```

```
    sens.stat.scan_time=scan_end-scan_start;
}
```

emerg.c

```
#include <libcrobu.h>
#include <libcalbl.h>

#include "emerg.h"
#include "generic.h"
#include "executor.h"
#include "serial.h"
#include "us.h"
#include "headers.h"


EMERGENCY emerg;

extern MOVES mov;
extern POSTURE pos;


int left_vel=0, right_vel=0;

int get_us_limit() {
  int max;

  if (abs(mov.left) > abs(mov.right))
    max=abs(mov.left);
  else
    max=abs(mov.right);

  if (max<22)
     return(220);
  return(max*10);
}


int sub24(int a, int b) {
  int r;

  r=a-b;
  if (r<1)
    r=24+r;
  return r;
}


int add24(int a, int b) {
  int r;

  r=a+b;
  if (r>24)
    r=r-24;
  return r;
}


void set_table(int *a, int *b) {
  int n;
  int *source, *dest;

  dest=a;
  source=b;

  for (n=1; n<=24; n++) {
    *dest=*source;
    dest++;
    source++;
  }
```

```
}

void table_or(int *a, int *b) {
  int n;
  int *source, *dest;

  dest=a;
  source=b;

  for (n=1; n<=24; n++) {
    if ( (*dest==ACTIVE) || (*source==ACTIVE) )
      *dest=ACTIVE;
    else
      *dest=NOT_ACTIVE;
    dest++;
    source++;
  }
}


int get_bumper_state() {
  static int first_time=TRUE;
  static long *pl;

  if (first_time==TRUE) {
    pl=get_log_input(1);
    first_time=FALSE;
  }
  return(*pl & 1);
}


int get_main_sensor(void) {
  int left, right;
  double r;

  /* Nao tem velocidades armazenadas */
  if ( (left_vel==0) && (right_vel==0) ) {
    left=mov.left;
    right=mov.right;
  }
  else {
    left=left_vel;
    right=right_vel;
  }

  if (left==right) {
    if (left==0)
      return(0);                    /* No motion */
    else if (left<0)
      return(13);                   /* Backward motion */
    else
      return(1);                    /* Forward motion */
  }

  r = (left+right)*1.0/(left-right);
  r = r*DBW/2;

  if (fabs(r) > LIMIT12) {          /* Straight motion */
    if ( (left>0) && (right>0) ) {
      return(1);
    }
    else {
      return(13);
    }
  }
  else if (fabs(r) > LIMIT23) {     /* Very soft curve */
    if ( (left>0) && (right>0) ) {
      if (r > 0)
        return(2);
      else
        return(24);
    }
```

```
    else {
      if (r > 0)
        return(12);
      else
        return(14);
    }
  }
  else if (fabs(r) > LIMIT34) {     /* Soft curve */
    if ( (left>0) && (right>0) ) {
      if (r > 0)
        return(3);
      else
        return(23);
    }
    else {
      if (r > 0)
        return(11);
      else
        return(15);
    }
  }
  else if (fabs(r) > LIMIT45) {     /* Medium curve */
    if ( (left>0) && (right>0) ) {
      if (r > 0)
        return(4);
      else
        return(22);
    }
    else {
      if (r > 0)
        return(10);
      else
        return(16);
    }
  }
  else if (fabs(r) > LIMIT56) {     /* Hard curve */
    if ( (left>0) && (right>0) ) {
      if (r > 0)
        return(5);
      else
        return(21);
    }
    else {
      if (r > 0)
        return(9);
      else
        return(17);
    }
  }
  else if (r==0) {                  /* Rotation */
    if (left>0)
      return(7);
    else
      return(19);
  }
  else if (left>right) {            /* Very hard curve */
    if (fabs(left) > fabs(right))
      return(6);
    else
      return(18);
  }
  else {
    if (fabs(left) > fabs(right))
      return(8);
    else
      return(20);
  }
}

void emergency_stop(void)
  {
  send("ODOM OF\r");
  send("SERV OF\r");
  send("ODOM ON\r");
```

```
  sendf("ODOS C=%ld,%ld,%ld\r", pos.x, pos.y, pos.theta);
  }


void init_emergency(void) {
  send("BUMP OF\r");
  emerg.flag=NONE;
  emerg.rtm=FALSE;
  emerg.us_change=FALSE;
  emerg.auto_toggle=FALSE;
  emerg.time_stamp=get_time();
  emerg.on=BUMPER+US+COM;
}


void emergency(void) {
  static int wds[5];
  static int emergency_active_sensors[24];
  static int last_auto_toggle=FALSE;
  static int last_m_sensor=0;
  int main_sensor, n, i;

  extern US_DATA sens;

  main_sensor=get_main_sensor();

/*********************************************************
*********************************************************
Testa e actualiza os sensores US se esse modulo esta activo
*********************************************************
*********************************************************/
if ( (emerg.on&US)==US )
  {
/************************************************
Actualiza a tabela de sensores de emergencia activos
************************************************/
  if (main_sensor!=last_m_sensor) {

    if (main_sensor==7) {
      wds[0]=7;
      wds[1]=5;
      wds[2]=6;
      wds[3]=17;
      wds[4]=18;
    }
    else if (main_sensor==19) {
      wds[0]=19;
      wds[1]=20;
      wds[2]=21;
      wds[3]=8;
      wds[4]=9;
    }
    else if (main_sensor==6) {
      wds[0]=6;
      wds[1]=5;
      wds[2]=7;
      wds[3]=18;
      wds[4]=24;
    }
    else if (main_sensor==8) {
      wds[0]=8;
      wds[1]=7;
      wds[2]=9;
      wds[3]=14;
      wds[4]=20;
    }
    else if (main_sensor==20) {
      wds[0]=20;
      wds[1]=21;
      wds[2]=19;
      wds[3]=2;
      wds[4]=8;
    }
    else if (main_sensor==18) {
```

```
      wds[0]=18;
      wds[1]=17;
      wds[2]=19;
      wds[3]=12;
      wds[4]=6;
    }
    else if (main_sensor==0) {
      wds[0]=0;
      wds[1]=0;
      wds[2]=0;
      wds[3]=0;
      wds[4]=0;
    }
    else {
      wds[0]=main_sensor;
      wds[1]=sub24(main_sensor,1);
      wds[2]=add24(main_sensor,1);
      wds[3]=sub24(main_sensor,2);
      wds[4]=add24(main_sensor,2);
    }

    for (n=1; n<=24; n++) {           /* Constructs emergency_active_sensors */
      i=0;
      while ( (n!=wds[i]) && (i<5) )
        i++;
      if (i==5)
        emergency_active_sensors[n-1]=NOT_ACTIVE;
      else
        emergency_active_sensors[n-1]=ACTIVE;
    }

    if (emerg.auto_toggle==TRUE)
      set_table(sens.active_sensors,emergency_active_sensors);
    else
      table_or(sens.active_sensors,emergency_active_sensors);

    sens.change_flag=TRUE;
    last_m_sensor=main_sensor;
    last_auto_toggle=emerg.auto_toggle;
  }

  if ( (last_auto_toggle!=emerg.auto_toggle) || (emerg.us_change==TRUE) ) {
    if (emerg.auto_toggle==TRUE)
      set_table(sens.active_sensors,emergency_active_sensors);
    else
      table_or(sens.active_sensors,emergency_active_sensors);

    emerg.us_change=FALSE;
    sens.change_flag=TRUE;
    last_auto_toggle=emerg.auto_toggle;
  }

/*************************************/
/* Testa as emergencias nos sensores US */
/*************************************/
  if (main_sensor!=0) {
    i=0;
    while ( (sens.values[wds[i]-1]>0) && (i<5) )
      i++;
    if (i<5) {
      if ( (left_vel==0) && (right_vel==0) ) {
        left_vel=mov.left;
        right_vel=mov.right;
      }
      if (mov.status!=0) {
        mov.left=0;
        mov.right=0;
        mov.flag=TRUE;
      }
      printf("\nWARNING!  Not enough US information (missing sensor no. %d).",wds[i]);
      return;
    }
    else if ( (left_vel!=0) || (right_vel!=0) ) {
      mov.left=left_vel;
```

```
      mov.right=right_vel;
      left_vel=0;
      right_vel=0;
    }
    i=0;
    while ( (sens.values[wds[i]-1] > get_us_limit()) && (i<5) )
      i++;
    if (i<5) {
      emerg.flag=emerg.flag|US;     /* Sinaliza uma emergencia devida aos US */
          emergency_stop();
          left_vel=0;
          right_vel=0;
          emerg.rtm=FALSE;
      printf("\nWARNING!  Object too close (see sensor no. %d = %ld).",wds[i],sens.values[wds[i]-1]);
      /* printf("\nAge = %ld  -  Actual time = %ld",sens.node_times[get_sensor_node(wds[i])-1], get_time()); */
      return;
    }
    emerg.flag=NONE;
    if (emerg.rtm==TRUE) {
      mov.flag=TRUE;
      emerg.rtm=FALSE;
    }
  }

  }

/***********************************/
/* Testa as emergencias nos bumpers */
/***********************************/
if ( (emerg.on&BUMPER)==BUMPER )
{
  if (get_bumper_state()==PRESSED) {
    emerg.flag=emerg.flag|BUMPER;     /* Sinaliza uma emergencia devida ao bumper */
    emergency_stop();
    printf("\nWARNING! See bumper!");
    return;
  }
    else
      emerg.flag=NONE;
}

/*****************************************************************************/
/* Se nao houve emergencia e ha um pedido de movimento, permite o movimento */
  if (emerg.rtm==TRUE) {    /* Request to stop */
     mov.flag=TRUE;
     emerg.rtm=FALSE;
  }

/***************************************/
/* Testa as emergencias nas comunicacoes */
/***************************************/
if ( (emerg.on&COM)==COM )
{
  if (get_time()-emerg.time_stamp > COMM_LIMIT) {
    emerg.flag=emerg.flag|COM;       /* Sinaliza uma emergencia devida as comunicacoes */
    emergency_stop();
    printf("\nWARNING!  I am alive. What about you?");
    return;
  }
}

  emerg.flag=NONE;
}
```

executor.c

```
#include <libcrobu.h>
#include <libcalbl.h>

#include "emerg.h"
#include "executor.h"
```

```
#include "generic.h"
#include "headers.h"


MOVES mov;
POSTURE pos;

extern EMERGENCY emerg;


void init_execute(void) {
  mov.flag=FALSE;
  mov.status=0;
  mov.left=0;
  mov.right=0;

  pos.x=0;
  pos.y=0;
  pos.theta=0;
  send("ODOM ON\r");

  printf("\nExecutor started.");
}


void close_execute(void) {
  send("SERV OF\r");
}


void execute(void) {
  static int serv_stat=0;
  static long int temp[6];

  sendf("ODOM O=%lx\r",temp);
  pos.x=temp[0];
  pos.y=temp[1];
  pos.theta=temp[2];

/*reset in case of emergency*/
  if (emerg.flag>0) {
    mov.left=0;
    mov.right=0;
    mov.status=0;
    serv_stat=0;
    mov.flag=FALSE;
    return;
  }

/*Actualization of status if necessary*/
  if (mov.status!=serv_stat) {
    if (mov.status==1)
      send("SERV ON T=BV\r");
    else {
      send("ODOM OF\r");
      send("SERV OF\r");
      mov.left=0;
      mov.right=0;
      send("ODOM ON\r");
      sendf("ODOS C=%ld,%ld,%ld\r", pos.x, pos.y, pos.theta);
    }
    serv_stat=mov.status;
  }

/*Realization of movements if possible*/
  if ( (mov.flag==TRUE) && (emerg.flag==FALSE) ) {
    if (mov.status!=1) {
      send("SERV ON T=BV\r");
      mov.status=1;
      serv_stat=1;
    }
    sendf("MOVE V AC=%d,%d\r",mov.left,mov.right);
    mov.flag=FALSE;
  }
```

```
}
```

kernel.c

```
/*************************************************/
/* Projecto de 5 Ano                             */
/*    Programa de Navegacao e Comunicacoes       */
/*    para Robuter III da Robosoft               */
/*                                               */
/*                12218 Paulo Miguel de Jesus Dias */
/*                10580 Emanuel Amaral Oliveira  */
/*************************************************/

#include <libcstd.h>
#include <libcalbl.h>
#include <libcpro.h>

#include "us.h"
#include "serial.h"
#include "generic.h"
#include "executor.h"
#include "emerg.h"
#include "headers.h"


int kill=FALSE;

/*                                    period,   priority,   offset*/
Procs_To_Add tab1={4, read_sensors,       4,          2,        6,
                     serial_spy,          2,          3,        0,
                     execute,            10,          4,        2,
                     emergency,          10,          5,        4 };

Procs_To_Remove tab2={4, read_sensors, serial_spy, execute, emergency};


void main()
{
/* Initializations of processes */
  init_emergency();
  init_execute();
  init_serial_spy();
  init_read_sensors();
  kill=FALSE;

  init_procit();

  while(add_procit(&tab1));

  while(kill==FALSE);

  printf("\nTerminar processos\n");
  while(rem_procit(&tab2));
  close_serial_spy();
  close_execute();
}
```

generic.c

```
#include <libcalbl.h>

#include "generic.h"
#include "headers.h"

long get_time()
{
  static long *ret;
  static int first_time=TRUE;

  if (first_time==TRUE) {
    sendf("TSYS C=5 O=%lx",&ret);
    first_time=FALSE;
  }
```

```
  return(*ret);
}


void delay_ticks(long ticks)
{
  long start;

  start=get_time();

  while( (get_time()-start) <= ticks);
}


void delay(int ms)
{
  static long start;

  start=get_time();

  while( (get_time()-start) <= ms/10);
}
```

ANEXO II.2 – *Código C*

ptu.c

```
#include <libcstd.h>
#include <libport.h>
#include <libcalbl.h>
#include <libcpro.h>
#include <libport.h>

#include "ptu.h"
#include "gate.h"
#include "serial.h"
#include "generic.h"
#include "headers.h"
#include "us.h"

#define PORT "SLD"

static SERIAL ptu_serial;


long pt_values[MAX_PTU_SENS];
long fvalues[MAX_PTU_SENS];
long dvalues[MAX_PTU_SENS];

int ptu_on;

static int angle;
int ccc;
int local_angle;


int div10(int value) {
  char aux[6];

  if (value == 0)
    return(0);

  sprintf(aux, "%d", value);
  if (strlen(aux) < 2)
    return(0);

  aux[strlen(aux)-1]='\0';
  return(atoi(aux));
}



void filter(void) {
  int i;

  for (i=0; i<70; i++) {
    if ( (pt_values[i]>MAX) && (pt_values[i+1]<MAX-1000) && (pt_values[i+2]>MAX) )
      fvalues[i+1]=(pt_values[i]+pt_values[i+2])/2;
    else
      fvalues[i+1]=pt_values[i+1];;
  }

  if ( (pt_values[70]>MAX) && (pt_values[71]<MAX-1000) && (pt_values[0]>MAX) )
    fvalues[71]=(pt_values[70]+pt_values[0])/2;
  else
    fvalues[71]=pt_values[71];;

  if ( (pt_values[71]>MAX) && (pt_values[0]<MAX-1000) && (pt_values[1]>MAX) )
    fvalues[0]=(pt_values[71]+pt_values[1])/2;
  else
    fvalues[0]=pt_values[0];;
}
```

```
void diff(void) {
  int i;

  filter();

  for (i=0; i<71; i++)
    dvalues[i]=fvalues[i+1]-fvalues[i];

  dvalues[71]=fvalues[0]-fvalues[71];
}



void read_pt(void) {
  static long temp[6];
  long ret;
  char aux[5];
  int pos;
  static int first_time=TRUE;

  local_angle=angle;

  sprintf(aux, "%d", local_angle);

  switch (aux[strlen(aux)-1]) {
    case '4':
    case '9':
      if(local_angle>0)
        local_angle++;
      else
        local_angle--;
      break;
    case '6':
    case '1':
      if(local_angle>0)
        local_angle--;
      else
        local_angle++;
      break;
  }

  sprintf(aux, "%d", local_angle);


  if ( (aux[strlen(aux)-1] == '5') || (aux[strlen(aux)-1] == '0') ) {
    pos=2*(local_angle+90);
    pos=pos/10;

  if (first_time == TRUE) {
      sendf("READ C=7,110000 T=%d V=%d O=%lx\r", PAN_TIME_OUT, SOUND_SPEED, temp);
      first_time=FALSE;
    }

    do {
      ret=sendf("READ C=7,110000 T=%d V=%d O=%lx\r",PAN_TIME_OUT, SOUND_SPEED ,temp);
    } while ( ret!=0);

/* Values greather than time out are equal to time_out */
    if(temp[0]>PAN_TIME_OUT_DIST)
      temp[0]=PAN_TIME_OUT_DIST;
    if(temp[1]>PAN_TIME_OUT_DIST)
      temp[1]=PAN_TIME_OUT_DIST;


/* Os valores sao corrigidos para uma velocidade do som de 335 */
    pt_values[pos]=(temp[1]-46)/0.96;
    pt_values[pos+36]=(temp[0]-46)/0.96;

  }

}
```

```
void send2ptu(char *msg) {
  char aux[50];
  char ch;

  strcpy(aux, msg);
  strcat(aux, "\r");

  s_write(ptu_serial.desc,aux,strlen(aux));
  do
    s_read(ptu_serial.desc,&ch,1);
  while(ch!=13);

}


int init_ptu(void) {
  int n;

  if ( (ptu_serial.desc=s_open(PORT,O_RDWR) ) <= 0) {
    printf("\nERROR opening PTU serial port!\n\r");
    return(ERROR);
  }

  printf("\nPTU started.\n");

  send2ptu("ED");
  send2ptu("FT");
  send2ptu("I");
  send2ptu("PA6000");

  move_ptu(-1755,SPEED);

  for (n=0; n<MAX_PTU_SENS; n++) {
    pt_values[n]=0;
    fvalues[n]=0;
    dvalues[n]=0;
  }

  ptu_on=FALSE;

  return(OK);
}


void close_ptu(void) {
  printf("\n--> %d", ccc);
  s_close(ptu_serial.desc);
}


void move_ptu(int pos, int speed) {
  static int last_speed=0;


  /* Set the speed of the pan & tilt unit */
  if (speed != last_speed) {
    sprintf(ptu_serial.buff_out,"ps%d\r",speed);
    send2ptu(ptu_serial.buff_out);
    last_speed=speed;
  }

  /* Send the movement to the monitor of the pan & tilt unit */
  sprintf(ptu_serial.buff_out,"pp%d\r",pos);
  send2ptu(ptu_serial.buff_out);
}

void ptu(void)
{
  int size;
  static char *token;
  static int state=1;
  static int first_time=TRUE;
```

```
  static int position=0;
  int n;

  char ch;

  if (ptu_on) {
    diff();
    if (first_time) {
      s_write(ptu_serial.desc,"PP\r",3);
      ccc=0;
      first_time=FALSE;
      return;
    }

    if ( (state==0) && (position==1655) ) {
      state=1;
      move_ptu(-1755,SPEED);
      do
        s_read(ptu_serial.desc,&ch,1);
      while(ch!=13);
    }

    if ( (state==1) && (position==-1755) ) {
      move_ptu(1655,SPEED);
      state=0;
      do
        s_read(ptu_serial.desc,&ch,1);
      while(ch!=13);
    }

    size=s_read(ptu_serial.desc,ptu_serial.buff_in,20);
    if (size==0)
      return;
    token=ptu_serial.buff_in;
    while (*token!=13)
      token++;
    *token='\0';

    token=ptu_serial.buff_in+2;
    position=atoi(token);
    angle=position*185/3600;

    read_pt();
    s_write(ptu_serial.desc,"PP\r",3);
  }
  else {
    first_time=FALSE;
    for (n=0; n<MAX_PTU_SENS; n++)
     pt_values[n]=0;
  }
}
```

gate.c

```
#include "gate.h"

#include "executor.h"
#include "generic.h"
#include "headers.h"

#define Va    5

extern long pt_values[72];
extern long fvalues[72];
extern long dvalues[72];

int spot_on;

extern MOVES mov;
```

```
int add71(int a, int b) {
  int r;

  r=a+b;
  if (r>71)
    r=r-72;
  return r;
}



void init_gate(void) {
  spot_on=FALSE;
  printf("SPOT process started.\n");
}


int get_corners(int *left, int *right, long *left_value, long *right_value) {
  int p, n, pos;
  int corner[20];
  int in_door, lock;

  pos=0;                                  /* Detecta o numero de transicoes */
  for (n=0; n<72; n++) {
    if (dvalues[n] > MARGIN) {
      corner[pos]=n;
      pos++;
    }
  }

  p=0;
  while ( (lock==FALSE) && (p<pos) ) {  /* Procura por uma passagem */
    *right=corner[p];
    n=*right;
    in_door=0;
    do {
      n=add71(n,1);
      in_door++;
    } while (dvalues[n]>-MARGIN);        /* Isto pode ficar mais eficiente */
    *left=n+1;
    if (in_door<8)
      lock=TRUE;
    else
      lock=FALSE;;
    p++;
  }

  if (lock == TRUE) {                   /* Se foi encontrada uma passagem */
    *left_value=fvalues[*left];
    *right_value=fvalues[*right];
  }
  else {                                /* Passagem nao encontrada */
    *left=0;
    *right=0;
    *left_value=0;
    *right_value=0;
    printf("\n");
  }
  return(lock);
}


int get_center(int left, int right) {
  int center;

  if (left<right) {
    left=left+72;
    center = (left+right)/2-72;
  }
  else
    center = (left+right)/2;

  return(center);
```

```
}

void gate(void) {
  int left, right;
  long left_value, right_value;
  static int first_time=TRUE;
  static int start_time;
  static int first_not_locked;
  int center;

  if (spot_on == FALSE) {
    first_not_locked=TRUE;
    return;
  }

  if (spot_on == TRUE) {
    if (first_time) {
      printf("\nSpot finder started!");
      first_not_locked=TRUE;
      first_time=FALSE;
    }
  }

  if (get_corners(&left, &right, &left_value, &right_value) == TRUE) {

    if ( (left_value > MIN) && (right_value > MIN) ) {

      if ( abs(left_value-right_value) < 300) {                  /* Robot is centered */
        center =  get_center(left, right);
       if ( (center<3) || (center>69) ) {
          printf("\nRobot is centered.");
          mov.left=Va;
          mov.right=Va;
          mov.flag=TRUE;
          first_not_locked=TRUE;
          return;
       }
       if ( (center>=3) && (center<18) ) {
         printf("\nC-Moving left");
         mov.left=-Va/2;
         mov.right=Va/2;
         mov.flag=TRUE;
         first_not_locked=TRUE;
         return;
       }
       if ( (center<=69) && (center>54) ) {
         printf("\nC-Moving right");
         mov.left=Va/2;
         mov.right=-Va/2;
         mov.flag=TRUE;
         first_not_locked=TRUE;
         return;
       }
      }

      if ( left_value > right_value+300 ) {
        if ( (left>54) && (left<69) ) {
          printf("\nGate at right. (%d)", left);
          mov.left=0;
          mov.right=0;
          mov.flag=TRUE;
          spot_on=FALSE;
          first_not_locked=TRUE;
          return;
        }
        if ( (right>2) && (right<18) ) {
          printf("\nMoving left %ld (%d) %ld (%d)", left_value, left, right_value, right);
          mov.left=-Va/2;
          mov.right=Va/2;
          mov.flag=TRUE;
          first_not_locked=TRUE;
          return;
        }
```

```
        printf("\nMoving to left corner %d %d", left, right);
        mov.left=Va;
        mov.right=Va;
        mov.flag=TRUE;
        first_not_locked=TRUE;
        return;
      }
      if ( right_value > left_value+300 ) {
        if ( (right>3) && (right<18) ) {
          printf("\nGate at left. (%d)", right);
          mov.left=0;
          mov.right=0;
          mov.flag=TRUE;
          spot_on=FALSE;
          first_not_locked=TRUE;
          return;
        }
        if ( (right<70) && (right>54) ) {
          printf("\nMoving right");
          mov.left=Va/2;
          mov.right=-Va/2;
          mov.flag=TRUE;
          first_not_locked=TRUE;
          return;
        }
        printf("\nMoving to right corner");
        mov.left=Va;
        mov.right=Va;
        mov.flag=TRUE;
        first_not_locked=TRUE;
        return;
      }
      printf("\nI don't know what to do! (%d %d)", left, right);
    }
    else {
      printf("\nI'm close!");
      mov.left=0;
      mov.right=0;
      mov.flag=TRUE;
      spot_on=FALSE;
    }
  }
  else {
    /* printf("\nNot locked"); */
    mov.left=0;
    mov.right=0;
    mov.flag=TRUE;
    if (first_not_locked) {
      start_time=get_time();
      first_not_locked=FALSE;
    }
    if (get_time()-start_time>TIMEOUT) {
      spot_on=FALSE;
      first_not_locked=TRUE;
    }
  }
}
```

doors.c

```
#include <libcstd.h>
#include <libport.h>
#include <libcalbl.h>
#include <libcpro.h>
#include <libport.h>

#include "doors.h"
#include "serial.h"
#include "generic.h"
#include "headers.h"
#include "ptu.h"
#include "executor.h"
```

```
#include "emerg.h"
#include "us.h"

extern long pt_values[MAX_PTU_SENS];
extern MOVES mov;
extern EMERGENCY emerg;
extern US_DATA sens;
extern ptu_on;

int door;
long fvalues[MAX_PTU_SENS];
long min_read[MAX_PTU_SENS];

/* Funcao que calcula as distancias minimas para embate em funcao da geametria do robot e da posicao da PTU */
/*********************/
void init_doors(void)
/*********************/
{
  int x1,x2,x3,x4,i;

  door=OFF;

/* Compute minimal readings for each angle of the PTU  */
  x1=PAN_X_POSITION-BOX_WIDTH/2+SAFETY_DIST;
  x2=ROBOT_LENGTH-PAN_X_POSITION-BOX_WIDTH/2+SAFETY_DIST;
  x3=PAN_Y_POSITION-BOX_WIDTH/2+SAFETY_DIST;
  x4=ROBOT_WIDTH-PAN_Y_POSITION-BOX_WIDTH/2+SAFETY_DIST;

  min_read[0]=x1;
  min_read[36]=x2;

  for(i=1;i<=5;i++)
    min_read[i]=x1/cos((5*3.14/180)*i);

  for(i=6;i<=18;i++)
    min_read[i]=x3/sin((5*3.14/180)*i);

  for(i=19;i<=27;i++)
    min_read[i]=x3/sin((5*3.14/180)*(36-i));

  for(i=28;i<=35;i++)
    min_read[i]=x2/cos((5*3.14/180)*(36-i));

  for(i=37;i<=45;i++)
    min_read[i]=x2/cos((5*3.14/180)*(i-36));

  for(i=46;i<=54;i++)
    min_read[i]=x4/sin((5*3.14/180)*(i-36));

  for(i=55;i<=66;i++)
    min_read[i]=x4/sin((5*3.14/180)*(72-i));

  for(i=67;i<=71;i++)
    min_read[i]=x1/cos((5*3.14/180)*(72-i));


  for(i=0;i<=69;i=i+3)
    printf("\nposicao %d = %ld       posicao  %d = %ld        posicao %d =
%ld",i,min_read[i],i+1,min_read[i+1],i+2,min_read[i+2]) ;

}


/**************/
void doors(void)
/**************/
  {
  static int count=0;
  int i;
  int turn_left,turn_right;


  if (door==ON)
    {
```

```
  {
  static int count=0;
  int i;
  int turn_left,turn_right;


  if (door==ON)
    {
    emerg.on=BUMPER;
    /* Na primeira iteracao, garante que na posicao 0graus
      o caminho esta desempedido (robot mais ou menos centrado
      na porta) */

/*************************
    if(center==FALSE)
    {

    if( (pt_values[0]>MAX_DIST)||(side==0) )
      center=TRUE;
      else
      {
      if(side==2)
        {
        mov.left=0;
        mov.right=-1;
        mov.flag=TRUE;
        printf("\n DIREITA");
        }
      else if(side==1)
        {
        mov.left=-1;
        mov.right=0;
        mov.flag=TRUE;
        printf("\n ESQUERDA");
       }
      return;
      }
    }
*****************************/

  /* Desvia-se dos obstaculos relativamente as distancias minimas
    definidas em min_read */
    turn_left=FALSE;
    turn_right=FALSE;
    for(i=1;i<=17;i++)
      if(pt_values[i]<min_read[i])
       {
      printf("\n ver posicao %d(%ld)",i,pt_values[i]);
      turn_right=TRUE;
      }
    for(i=54;i<=70;i++)
      if(pt_values[i]<min_read[i])
      {
      printf("\n ver posicao %d(%ld)",i,pt_values[i]);
      turn_left=TRUE;
      }

    if((turn_left==FALSE)&&(turn_right==FALSE))
      {
      count=0;
      mov.left=2;
      mov.right=2;
      mov.flag=TRUE;
      }
    else if((turn_left==TRUE)&&(turn_right==FALSE))
      {
      count=0;
      mov.left=-1;
      mov.right=0;
      mov.flag=TRUE;
      }
    else if((turn_left==FALSE)&&(turn_right==TRUE))
      {
      count=0;
```

```
      mov.left=0;
      mov.right=0;
      mov.flag=TRUE;
      if (count>((TIME_BEFORE_MOVE)/10))
        {
        door=OFF;
        ptu_on=OFF;
        emerg.on=BUMPER+US+COM;
        printf("Can't cross the door");
        }
      }

  /* Se todos os valores medidos de um dos lados sao elevados, desliga o modo de passar portas */
    for(i=1;i<=18;i++)
      if(pt_values[i]<(MAX_DIST))
        {
        for(i=54;i<=71;i++)
          if(pt_values[i]<(MAX_DIST))
              return;
          door=OFF;
          ptu_on=OFF;
          mov.left=0;
          mov.right=0;
          mov.flag=TRUE;
          emerg.on=BUMPER+US+COM;
          printf("\nDoor crossed successfully");
          return;
        }
      door=OFF;
      ptu_on=OFF;
        mov.left=0;
        mov.right=0;
      mov.flag=TRUE;
      emerg.on=BUMPER+US+COM;
      printf("\nDoor crossed successfully");
  }
}
```

ANEXO II.1 – *headerfiles*

ptuhc

```
#define ERROR       -1
#define OK           1

#define SPEED 1500
#define PAN_TIME_OUT 15
#define PAN_TIME_OUT_DIST 2491

#define MAX_PTU_SENS 72


/* Ultra-sound data filter */
void filter(void);


/* Detects high variations in ultra-sound data */
void diff(void);

/* Move the pan axis to the absolute position pos at the speed desired */
void move_ptu(int pos,int speed);


/* Process that coordinates PTU movements and read angle value */
void ptu(void);


/* Send message to PTU unit */
void send2ptu(char *);


int init_ptu();


void close_ptu();
```

gate.h

```
#define MARGIN 300
#define MIN    700
#define MAX    2300

#define TIMEOUT  1000


void init_gate(void);


/* Find passage corners and their values */
int get_corners(int *, int *, long *, long *);


/* Get center given left and right corner */
int get_center(int, int);


/* Process that put the robot in front of the door */
void gate(void);
```

doors.h

```
#define TIME_BEFORE_MOVE 500
#define MAX_DIST 1500
#define ROBOT_LENGTH 1025
#define ROBOT_WIDTH 700
/* Distance to the front */
#define PAN_X_POSITION 725
/* Distance to the left side */
#define PAN_Y_POSITION 350
#define BOX_WIDTH 100
#define SAFETY_DIST 35


/* inicilization of process for crossing doors */
void init_doors(void);

/* Process that controls doors crossing */
void doors(void);
```

ANEXO III - *headerfile e Código C*

dem.h

```
#define DISTX 12000
#define DISTY 10000
#define RAIO 10000


/* function that coordinates moviment */
void dem(void);
```

dem.c

```
#include <libcstd.h>
#include <libport.h>

#include "generic.h"
#include "us.h"
#include "executor.h"
#include "emerg.h"
#include "dem.h"

extern int kill;
extern US_DATA sens;
extern POSTURE pos;
extern MOVES mov;
extern EMERGENCY emerg;

static int left=0,right=0;

int orientacao(void) {
    static int d,temp,state=0;

    if(state==0) {
      printf("\nINICIO CALIBRACAO");
      if(sens.values[17]>sens.values[19])
      state=1;
      if(sens.values[17]<sens.values[19])
      state=2;
    }
    switch(state) {
      case 1: if(sens.values[17]>sens.values[19]){
                mov.left=1;
                mov.right=0;
                emerg.rtm=TRUE;
                return(FALSE);
              }
              else {
                state=3;
                d=(2*RAIO+DISTY-sens.values[17]*10)/2;
                return(FALSE);
              }
      case 2: if(sens.values[17]<sens.values[19]){
                mov.left=-1;
                mov.right=0;
                emerg.rtm=TRUE;
                return(FALSE);
              }
              else {
                state=3;
                d=(2*RAIO+DISTY-sens.values[17]*10)/2;
                return(FALSE);
              }
      case 3: if(sens.values[0]*10>DISTX) {
                 mov.left=10;
                 mov.right=10;
                 emerg.rtm=TRUE;
```

```
      else {
        state++;
        return;
      }
    case 4: if(pos.y>10000) {
        left=20;
        right=20;
        break;
      }
      else {
        state++;
        return;
      }
    case 5: if(pos.theta<0) {
        left=(20*1.0)/(((RAIO+3000)*1.0/(RAIO-3000)));
        right=20;
        break;
      }
      else {
        state++;
        return;
      }
    case 6: if(pos.x<5000) {
        left=10;
        right=10;
        break;
      }
      else {
        state=0;
        orient=TRUE;
        left=0;
        right=0;
        n++;
        if(n>5)
          kill=TRUE;
          break;
        }
  }
  if((mov.left!=left)||(mov.right!=right)) {
    if ((mov.left==0)||(mov.right==0))
      if (count<20) {
        count++;
        return;
        }
      count=0;
      mov.left=left;
      mov.right=right;
      emerg.rtm=TRUE;
    }
}
}
```