
DEETC – Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores

MEIM - Mestrado Engenharia informática e multimédia

# Robótica Móvel

## Trabalho prático 2

**Turma:**

MEIM-2D

**Trabalho realizado por:**

Miguel Távora Nº45102

Bruno Colaço Nº45037

**Docente:**

Jorge Pais

**Data:** 23/07/2022

# Índice

# Índice ilustrações e tabelas

# 1. Introdução

O segundo trabalho de robótica móvel possuí o objetivo de com base na implementação feita no primeiro trabalho construir trajetórias automaticamente para o robô alcançar pontos objetivo tridimensionais tendo por base pontos bidimensionais. Uma trajetória corresponde a um conjunto de comandos, ou seja, duas curvas e uma reta que foram implementadas no primeiro trabalho. Isto permite ao robô ir para uma determinada posição no mundo com uma determinada rotação tendo como referência a sua localização.

Um outro objetivo é ser possível calcular automaticamente pontos objetivo no espaço visível pelo robô e construir uma trajetória automaticamente para o robô conseguir atingir esses pontos objetivo. A partir deste cálculo é possível por o robô a desviar-se de objetos, recolher objetos ou outro tipo de tarefas.

O objetivo final da implementação é conseguir por o robô a uma posição qualquer e sempre que se aproximar de uma parede, ele a siga a uma distância fixa.

Para ser possível esta implementação será necessário realizar alguns ajustes relativamente ao primeiro trabalho, visto que este trabalho possui alguns requisitos extra que na implementação do primeiro trabalho não eram necessários.

# 2. Desenvolvimento

## 2.1 Trajetórias

Uma trajetória no contexto do trabalho prático corresponde a um caminho que o robô tem de percorrer para ir até uma determinada posição no mundo real. Neste sentido existem várias abordagens para resolver este problema, mas a escolhida pelo grupo foi o robô ao centro. Esta metodologia consiste no prossuposto de que o robô é o centro do universo, quer isto dizer que o robô se encontra sempre na posição (0, 0) em x e y. Nesta metodologia quando o robô se move, na verdade o que acontece é que todos os objetos se movem de acordo com o seu movimento, neste caso o robô encontra-se sempre na mesma posição visto que ele é centro do referencial. É necessário definir primeiramente qual é o referencial pois todas as contas realizadas serão baseadas nele, caso se mude o referencial todas as contas deixam de fazer sentido pois deixam de funcionar.

Um exemplo de representação do robô ao centro é o que se segue:

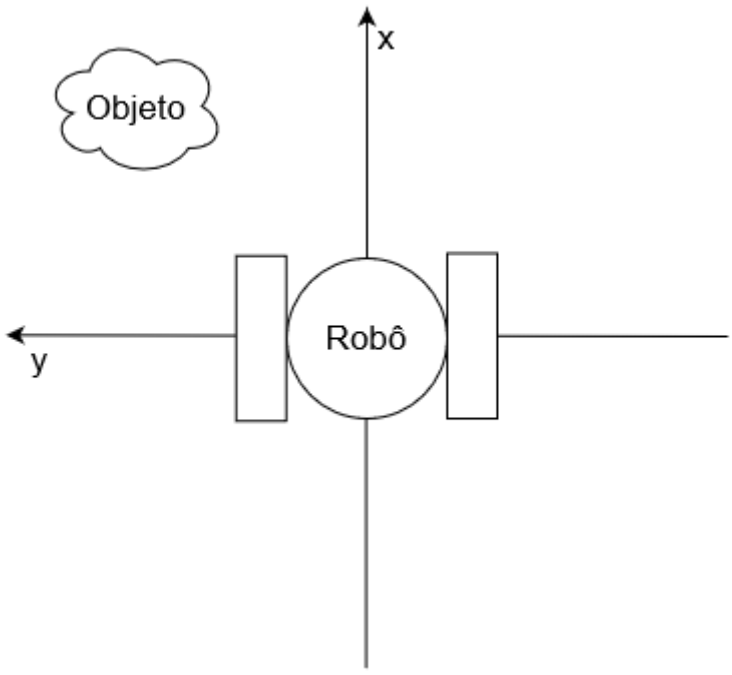


**Figura 1 – Referencial do robô ao centro**

Caso o robô rode 90 graus no sentido dos ponteiros do relógio o resultado é o que se segue:

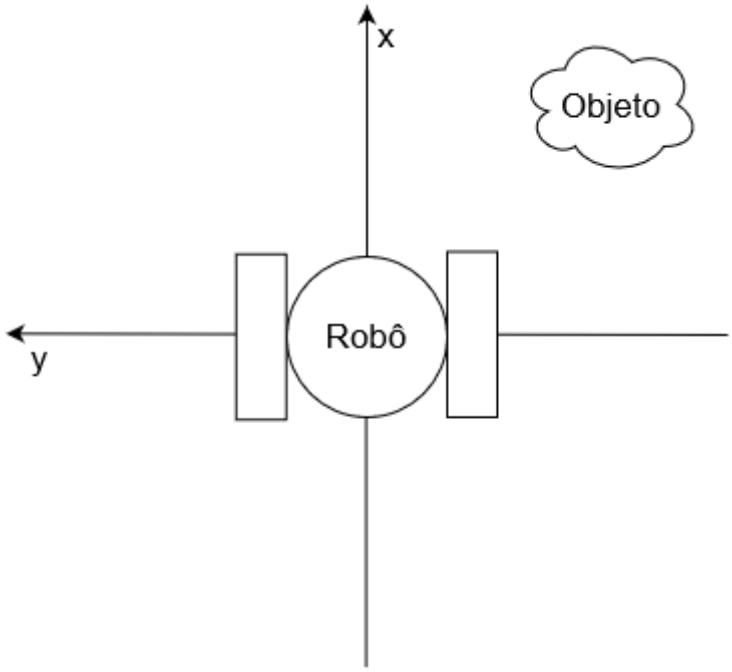


**Figura 2 - Referencial do robô ao centro rodado 90º no sentido dos ponteiros do relógio**

Tendo o referencial do robô é possível definir o ponto objetivo e definir automaticamente uma trajetória para o robô ir para o ponto objetivo. O ponto objetivo é constituído por uma coordenada em x ($Xf$), em y ($Yf$) e uma rotação ($Of$). Os pontos objetivos definidos têm algumas limitações. Estas limitações devem-se ao facto de robô só poder ir para pontos que estejam no seu campo de visão, o seu campo de visão é nomeadamente 45° do centro para a esquerda e 45° do centro para a direita. Quer isto dizer que o valor de $Xf$ tem de ser sempre superior a $Yf$. Um exemplo de diagrama dos pontos possíveis de realizar é o que se segue:

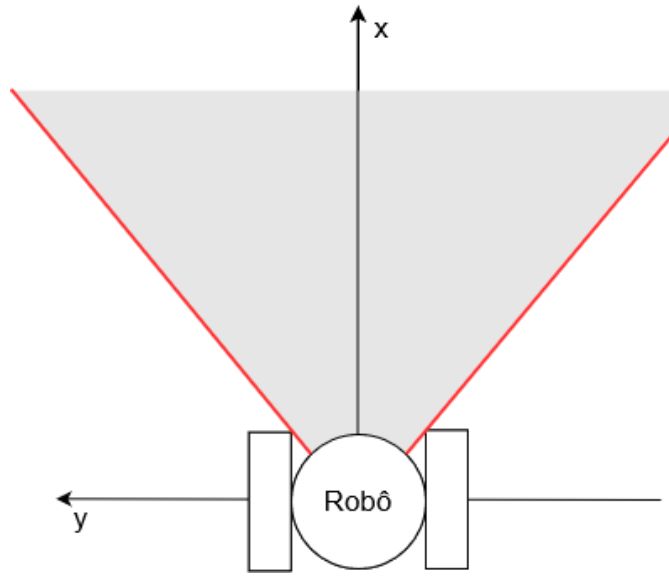


**Figura 3 - Pontos possíveis de realizar**

No diagrama anterior a zona a cinzento delimita os pontos fazíveis de realizar. No cálculo das trajetórias serão realizadas as contas para o robô se deslocar desde o centro do referencial até os 45° para a esquerda. Para ser possível realizar para o lado direito basta definir o valor de $Yf$ como negativo.

Uma trajetória obedece a uma gramática definida pelos comandos de retas e curvas. Uma trajetória corresponde a um conjunto constituído sempre por três comandos, onde o primeiro é sempre uma curva para a direita ou esquerda, seguida de uma reta e outra curva para a direita ou esquerda. No caso do trabalho prático foram implementadas no total três trajetórias distintas.

## 2.1.1 Trajetória 1 teórica

A primeira trajetória é escolhida sempre que $d_{Xif}$ é superior ou igual ao valor $Xf$.

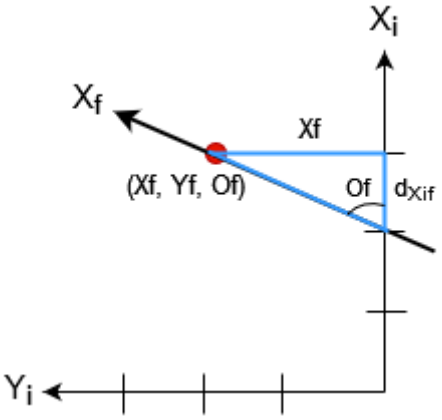


**Figura 4 - Triângulo para cálculo do $d_{xif}$**

Após feito o triângulo é necessário realizar as seguintes fórmulas para obter o $d_{Xif}$.

$$sen(x) = \frac{Cateto\ oposto}{Hipotenusa}$$

$$cos(x) = \frac{Cateto\ adjacente}{Hipotenusa}$$

A partir da fórmula anterior é possível então calcular a hipotenusa do triângulo pela fórmula:

$$Hipotenusa = \frac{Xf}{sen(Of)}$$

A partir da hipotenusa é possível calcular o $d_{Xif}$ através de:

$$d_{Xif} = \frac{\cos{(Of)}}{Hipotenusa}$$

Após calculado o $d_{Xif}$ é preciso começar a procurar por triângulos retângulos para ser possível realizar a triangulação para o robô realizar a trajetória. Primeiro é necessário calcular o $r$, isto é feito da seguinte forma:

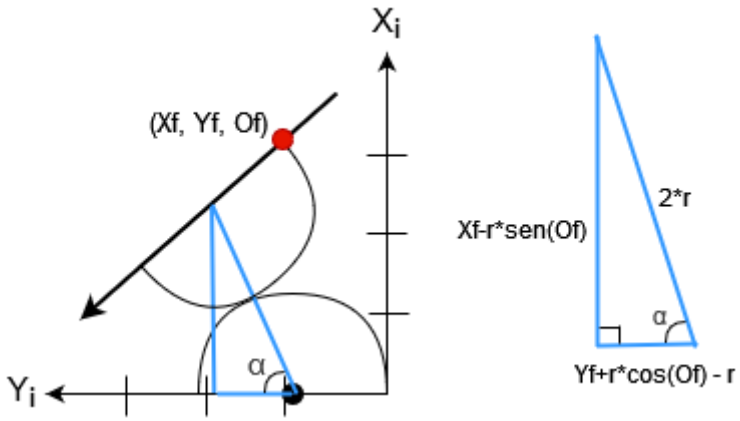


**Figura 5 - Triângulo para cálculo do $r$**

Pelo teorema de Pitágoras temos:

$$(2 * r)^2 = (Xf - sen(Of) * 2)^2 + (Yf + (\cos(Of) - 1) * r)^2$$

Os coeficientes da fórmula resolvente vão ser:

$$a = 2 + 2\cos(Of)$$
$$b = 2 * Yf * (1 - \cos(Of)) + 2 * Xf * sen(Of)$$
$$c = -(Xf^2 + Yf^2)$$

Do resultado obtido o valor correto vai ser sempre o valor positivo, pois não existem medidas negativas. Para calcular o ângulo $\alpha$ é dado por:

$$sen(\alpha) = \frac{Xf - r * sen(Of)}{2 * r}$$

$$\alpha = arcsen(\frac{Xf - r * sen(Of)}{2 * r})$$

## 2.1.2 Trajetória 1 prática

A partir das fórmulas anteriores é possível construir a trajetória, contudo existem certos valores que o robô não consegue realizar devido à manobrabilidade. Se forem enviados os valores das trajetórias calculados existirá uma grande quantidade de erro. Por isso quando se está a calcular as trajetórias é sempre necessário dividir a trajetória em trajetória teórica e trajetória prática. A calculada anteriormente foi a trajetória teórica, mas para minimizar o erro será agora calculada a trajetória prática.

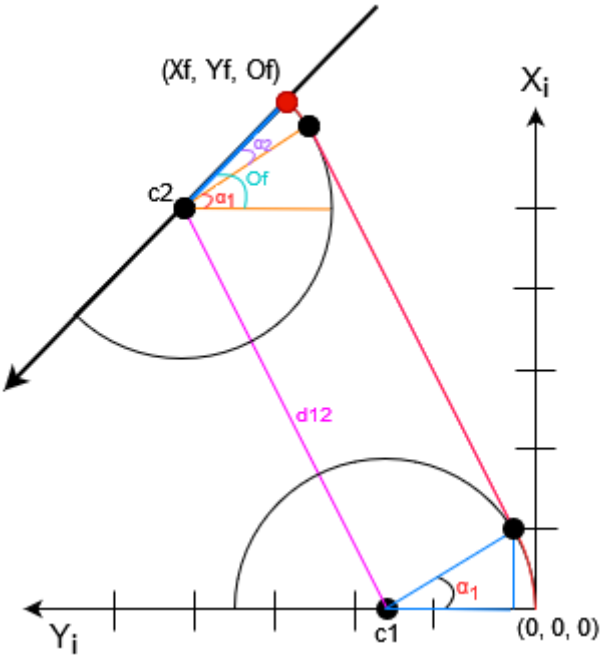


**Figura 6 - Trajetória prática**

A linha com a cor vermelha representa a trajetória que será feita pelo robô. Para obter esta trajetória foram feitos os seguintes cálculos:

O fator será então:

$$f = \frac{\left(r + \frac{distância\ entre\ rodas}{2}\right)}{\left(r - \frac{distância\ entre\ rodas}{2}\right)}$$

A partir do fator é possível calcular o $V_{left}$ que é dado por.

$$V_{left} = \frac{2 * velocidade\ do\ robô}{f + 1}$$

O valor da velocidade tem sempre de ser um número inteiro, por isso o valor de $V_{left}$ é sempre truncado para um valor inteiro.

Dado o $V_{left}$, o $V_{right}$ é então dado pela fórmula:

$$V_{right} = (2 * velocidade\ do\ robô) - V_{left}$$

O valor de $V_{right}$ também tem de ser truncado para um valor inteiro, estes dois valores permitem obter o raio de c1 que é dado por:

$$radius_p = \frac{distância\ entre\ rodas}{2} * (\frac{f + 1}{f - 1})$$

O cálculo do c2 com um dado raio é possível através da seguinte fórmula:

$$x_{c2} = Xf - r * sen(Of)$$

$$y_{c2} = Yf - r * cos(Of)$$

A distância entre c1 e c2 dado pelo comprimento de d12 é possível através do teorema de Pitágoras através da seguinte formula:

$$d_{12} = ((x_{c1} - x_{c2})^2 + (x_{c1} - x_{c2})^2)^{0.5}$$

O cálculo dos ângulos de $\alpha_1$ e $\alpha_2$ é dado por:

$$\alpha_1 = arc\cos(\frac{x_{c2}}{d12})$$

$$\alpha_2 = Of - \alpha_1$$

O resultado da trajetória será então composto pelos seguintes comandos:

- Curva à esquerda -> raio: $radius_p$ ângulo: $\alpha_1$
- Reta -> distância: $d_{12}$
- Curva à esquerda -> raio: $radius_p$ ângulo: $\alpha_2$

Para realizar a trajetória espelhada sobre o eixo do $X_i$, é utilizar exatamente os mesmos resultados, mas em vez de serem curvas para a esquerda seriam curvas para a direita.

### 2.1.3 Trajetória 2 teórica

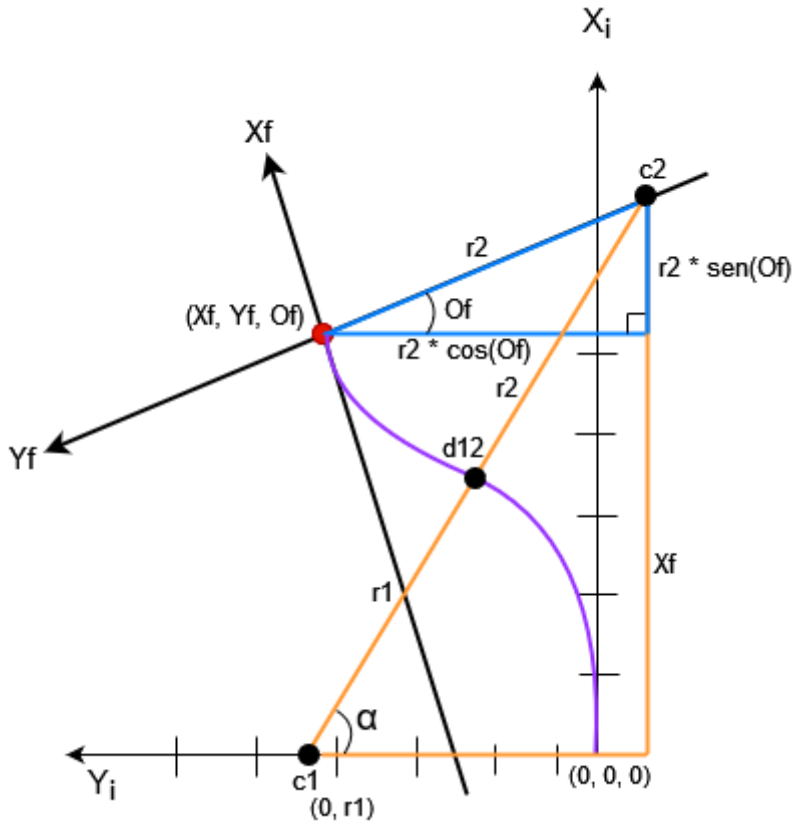


**Figura 7 – Diagrama da trajetória 2 teórica**

Esta trajetória é escolhida sempre que o valor de y do ponto c1 é superior ao y do ponto c2. Em termos matemáticos $y_{c1} > y_{c2}$.

Primeiramente é necessário calcular $r1$ e $r2$ utilizando a distância $d12$ entre os pontos $c1$ e $c2$ que define o centro de pontos de duas coordenadas de cada curva:

$c1 = (x_{c1},\ y_{c1})$

$x_{c1} = 0,\ \ y_{c1} = r1$

$c1 = (0,\ r1)$

$c2 = (x_{c2},\ y_{c2})$

$x_{c2} = Xf + r2 * sen(Of),\quad y_{c2} = Yf - r2 * \cos(Of)$

$c2 = (Xf + r2 * sen(Of),\ Yf - r2 * \cos(Of))$

Então:

$$d12 = \left(\left(Xf + r2 * sen(Of)\right)^2 + \left(Yf - r2 * \cos(Of)\right)^2\right)^{0.5} = r1 + r2$$

Como a equação d12 tem duas variáveis desconhecidas r1 e r2, existem diversas maneiras de resolver, mas a solução escolhida foi:

$$r1 = r2 => (2 * r)^2 = (Xf + r * sen(Of))^2 + (r * (\cos(Of) + 1) - Yf)^2$$

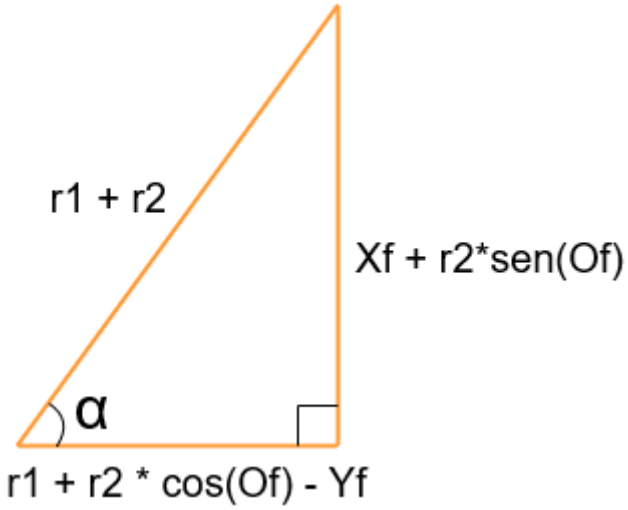


**Figura 8 - Triângulo da trajetória 2**

Cálculo do ângulo $\alpha$:

$$sen(\alpha) = \frac{Xf + r2 * sen(Of)}{r1 + r2}$$

$$\alpha = arc\ sen\left(\frac{Xf + r2 * sen(Of)}{r1 + r2}\right)$$

Para o cálculo do valor de r é utilizado a mesma fórmula da trajetória 1, nomeadamente:

$$a = 2 + 2\cos(Of)$$
$$b = 2 * Yf * (1 - \cos(Of)) + 2 * Xf * sen(Of)$$
$$c = -(Xf^2 + Yf^2)$$

## 2.1.4 Trajetória 2 prática

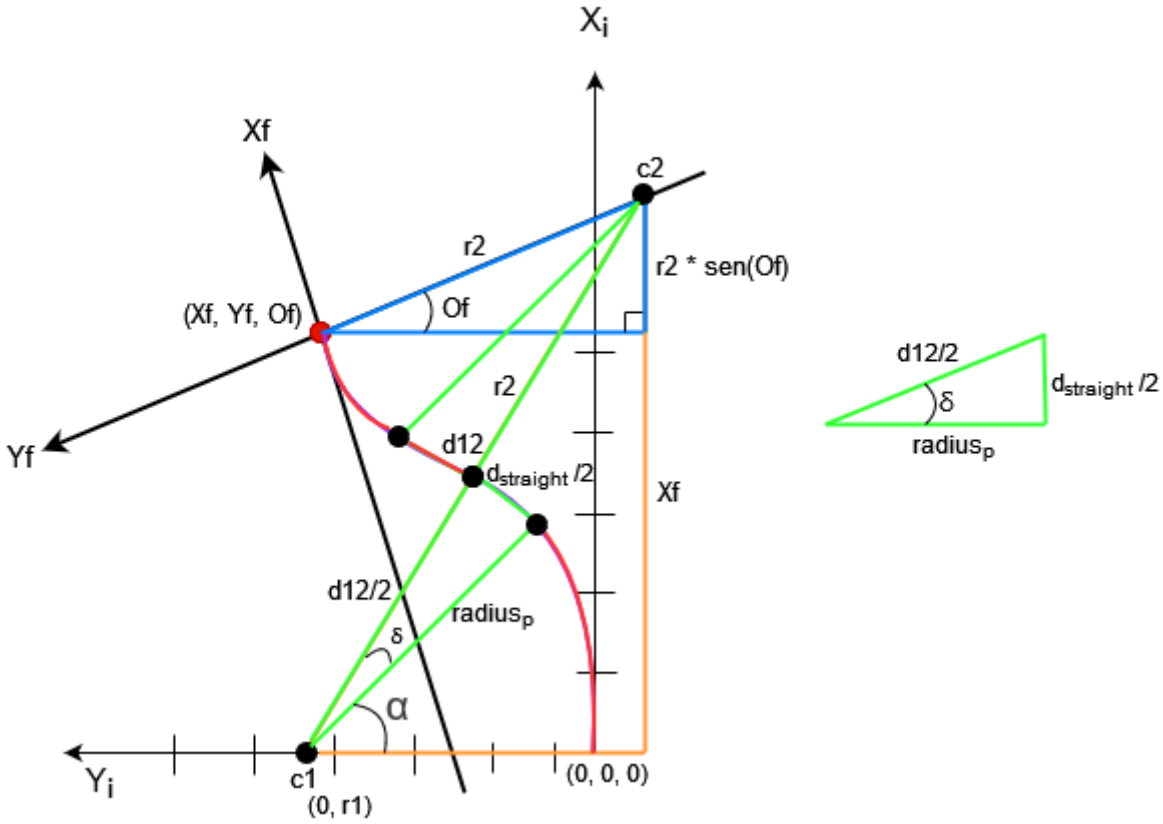


**Figura 9 - Trajetória 2 prática**

A trajetória 2 prática é representada pela linha de cor vermelha, existe uma parte da linha que está a verde para ser percetível o triângulo a verde.

O valor do fator é dado por:

$$f = \frac{v_1}{v_2} = \frac{radius_t + \frac{distância\ entre\ rodas}{2}}{radius_t - \frac{distância\ entre\ rodas}{2}}$$

A velocidade das rodas é então dada por:

$$v_2 = 2 * velocidade\ do\ robô - v_1 = \frac{2}{(f+1) * velocidade\ do\ robô}$$

$$v_1 = 2 * velocidade\ do\ robô - v_2$$

Como não existem velocidades com casas decimais é necessário converter valores os decimais para valores inteiros. Neste caso é necessário truncar os valores decimais, ou seja, ignorar as casas decimais e ficar somente com o valor inteiro.

Agora que se possuí as velocidades podemos calcular a trajetória. Para isso, é necessário saber os valores correspondentes ao triângulo a verde. O raio do círculo é então dado pela seguinte fórmula:

$$radius_p = \frac{distância\ entre\ rodas}{2} * \frac{f-1}{f+1}$$

O $d12$ é dado pela seguinte fórmula:

$$d12 = ((x_{c1} - x_{c2})^2 + (y_{c1} - y_{c2})^2)^{0.5}$$

O ângulo $\delta$ é dado por:

$$\delta = arc\cos\left(\frac{radius_p}{\frac{d12}{2}}\right)$$

A distância de $d_{straight}$ é dado por:

$$d_{straight} = d12 * sen(\delta)$$

O ângulo $\alpha$ prático é dado pela fórmula:

$$\alpha = arc\ sen\left(\frac{x_{c2}}{d12}\right) - \delta$$

O resultado da trajetória será então:

- Curva para a esquerda: raio-> $radius_p$ ângulo-> $\alpha$ prático
- Reta para a frente: distância -> $d_{straight}$
- Curva para a direita: raio-> $radius_p$ ângulo-> $\delta$

Para realizar a trajetória espelhada utiliza-se a mesma metodologia que na trajetória 1, utiliza-se os mesmos valores e troca-se as curvas para esquerda para a direita e vice-versa.

### 2.1.5 Trajetória 3 teórica

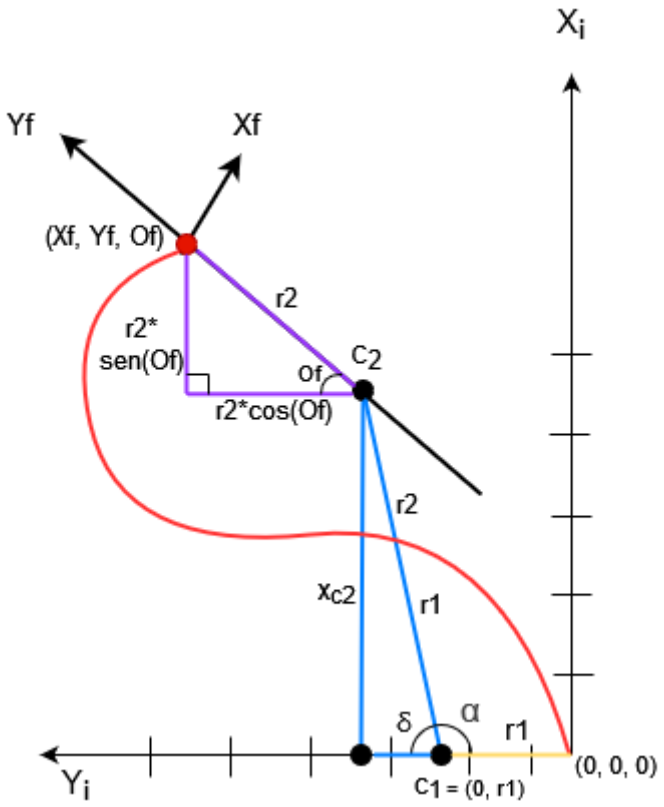


**Figura 10 - Trajetória 3 teórica**

Esta trajetória é escolhida sempre que a coordenada de y do ponto c2 for superior à coordenada y de c1, isto é $y_{c2} > y_{c1}$.

Da mesma maneira que a trajetória 2, primeiro calcula-se $r1$ e $r2$ utilizando a distância $d12$ entre pontos $c_1$ e $c_2$ que define o ponto do centro de duas cordas de curva dados por:

$c1 = (x_{c1},\ y_{c1})$

$x_{c1} = 0,\ \ y_{c1} = r1$

$c1 = (0,\ r1)$

$c2 = (x_{c2},\ y_{c2})$

$x_{c2} = Xf + r2 * sen(|Of|), \quad y_{c2} = Yf - r2 * \cos(|Of|)$

$c2 = (Xf + r2 * sen(|Of|),\ Yf - r2 * \cos(|Of|))$

Então:

$$d12 = \left(\left(Xf + r2 * sen(|Of|)\right)^2 + \left(r1 - Yf + r2 * \cos(|Of|)\right)^2\right)^{0.5} = r1 + r2$$

Como a equação d12 tem duas variáveis desconhecidas r1 e r2, existem diversas, mas a solução escolhida foi:

$$r1 = r2 => (2 * r)^2 = (Xf + r * sen(|Of|))^2 + (r * (\cos(|Of|) + 1) - Yf)^2$$

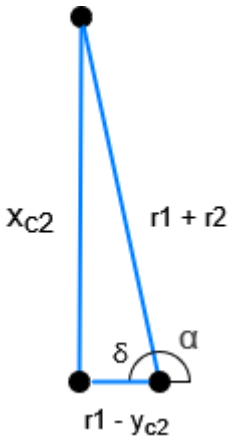


**Figura 11 - Triângulo para calcular a trajetória**

Cálculo do ângulo $\alpha$ é possível através:

$$\alpha = 180 - \delta$$

Temos então a equação:

$$(2 * r)^2 = (Yf - r * (1 + \cos(|Of|)))^2 + (Xf - r * sen(|Of|))^2$$

Para o cálculo do valor de r é utilizado a seguinte fórmula:

$$a = 2 - 2\cos(Of)$$
$$b = 2 * Yf * (1 - \cos(|Of|)) + 2 * Xf * sen(|Of|)$$
$$c = -(Xf^2 + Yf^2)$$

## 2.1.6 Trajetória 3 prática

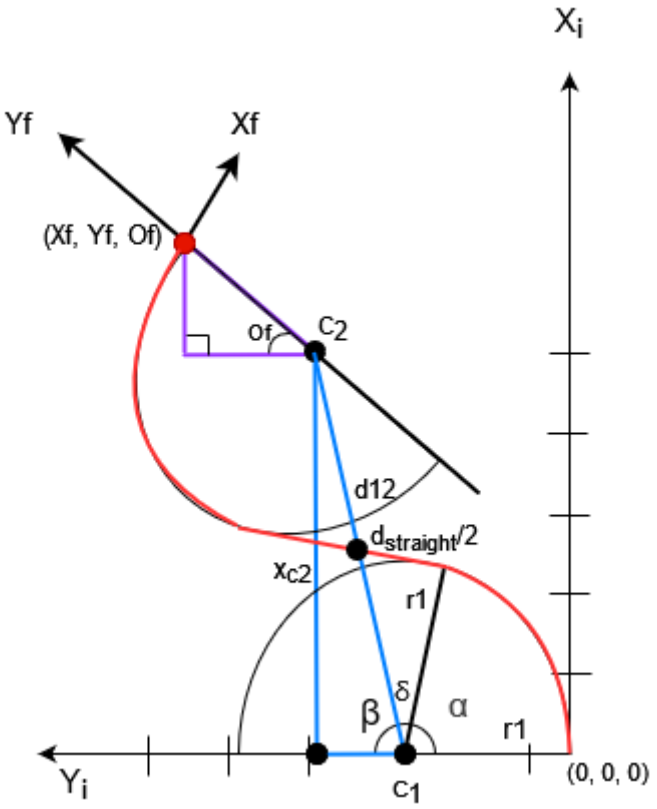


**Figura 12 - Trajetória 3 prática**

O fator é possível de calcular através da seguinte fórmula:

$$f = \frac{v_2}{v_1} = (\frac{radius_t + \left(\frac{distância\ entre\ rodas}{2}\right)}{radius_t - \left(\frac{distância\ entre\ rodas}{2}\right)})$$

A velocidade dos motores das rodas é dada por:

$$v_2 = 2 * Velocidade\ robô - v1 = \frac{2}{f+1} * Velocidade\ robô$$

O raio do círculo é então dado por:

$$radius_p = \frac{distância\ entre\ rodas}{2} * \frac{f+1}{f-1}$$

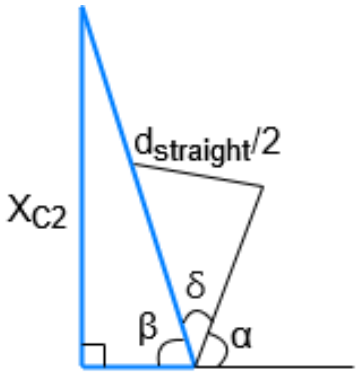


**Figura 13 - Triângulo da trajetória 3**

O cálculo do $d12$ é dado por:

$$d12 = ((x_{c1} - x_{c2})^2 + (y_{c1} - y_{c2})^2)^{0.5}$$

O ângulo $\delta$ é dado por:

$$\delta = arc\cos\left(\frac{radius_p}{\frac{d12}{2}}\right)$$

O ângulo $\beta$ é dado por:

$$\beta = arc\ sen\left(\frac{x_{c2}}{d12}\right)$$

A distância do $d_{straight}$ é dado por:

$$d_{straight} * sen(\delta)$$

O ângulo $\alpha$ é dado por:

$$\alpha = 180 - (\beta + \delta)$$

O resultado da trajetória será então:

- Curva para a esquerda: raio-> $radius_p$ ângulo-> $\alpha$ prático
- Reta para a frente: distância -> $d_{straight}$
- Curva para a direita: raio-> $radius_p$ ângulo-> $\alpha$ prático - $Of$

### 2.1.7 Implementação das trajetórias

Para a implementação das trajetórias foi criada a classe `Trajectories` esta classe é responsável por realizar todos os cálculos das trajetórias teóricas e práticas. O cálculo dos comandos da trajetória 1 são feitos no método `trajectory1()`, os da trajetória 2 no método `trajectory2()` e os da trajetória 3 no método `trajectory3()`. Todos estes métodos retornam os valores correspondentes aos raios e ângulos das curvas e à distância da reta da respetiva trajetória. Estes métodos internamente fazem a afetação dos valores do $y_{c1}$ e $y_{c2}$ da trajetória 2 e da trajetória 3 para atributos com os respetivos métodos de `get()`. Estes valores são importantes para ser possível escolher qual das duas trajetórias será escolhida.

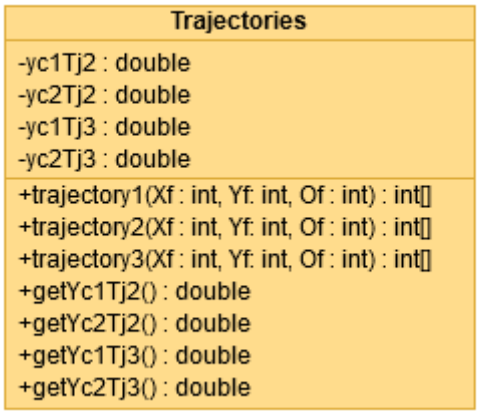


**Figura 14 - Diagrama da classe Trajectories**

Para existir organização na estrutura do código foi criado um gestor que a partir da implementação das trajetórias decide qual das trajetórias será executa de acordo com os valores de $Xf$, $Yf$ e $Of$. A classe de gestão das trajetórias é designada `TrajectoriesGestor` e esta classe além de selecionar a trajetória também é ela que envia os comandos para serem executados no robô e por isso recebe uma referência para a classe `MyRobotLego`. Quando o robô está a realizar uma trajetória ele pode colidir contra um objeto ou contra uma parede e por isso é necessário verificar colisões. Esta verificação também é feita na classe de gestão. A última funcionalidade implementada nesta classe é sempre que é enviado um comando para o robô ele pode ou não de seguida parar e esperar 2 segundos, caso o modo *debug* esteja ativo. Esta funcionalidade serve para *debug* visual dos comandos executados pelo robô.

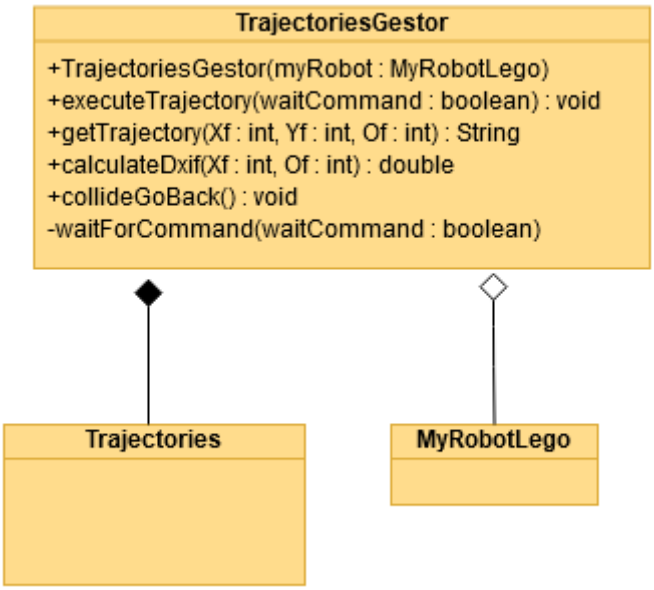


**Figura 15 - Diagrama de classes do TrajectoriesGestor**

Em termos dos resultados obtidos todas as trajetórias foram concluídas com sucesso, onde no máximo o robô no máximo possuía um erro máximo de 10 centímetros em x ou y. A trajetória com maior erro é a terceira trajetória, por isso foi a única que foram arredondados os valores para reduzir o erro.

### 2.2.1 Seguir a parede

O segundo objetivo do trabalho é conseguir que o robô sempre que tenha uma parede do seu lado direito ele consiga seguir a parede conforme ele vai andando. Este comportamento consiste no robô ver qual é o ângulo comparativamente entre ele e a parede, a partir desse valor endireitar-se e ficar sempre a uma distância de aproximadamente 50 centímetros entre ele e a parede. A distância de 50 centímetros obedece a uma equação afim e é designada por linha ideal.

Para este comportamento ser possível é necessário um sensor que consiga medir distâncias relativamente do robô com a parede. No caso do trabalho prático foi utilizado o sensor de ultrassons para medir a distância, contudo este sensor possui alguns erros. Este sensor possui um alcance de no máximo dois metros e meio e um ângulo de visão de 90°. Este sensor vai ser colocado do lado direito do robô para ele conseguir endireitar-se e também seguir a parede do lado direito. Um exemplo do funcionamento é a figura que se segue:

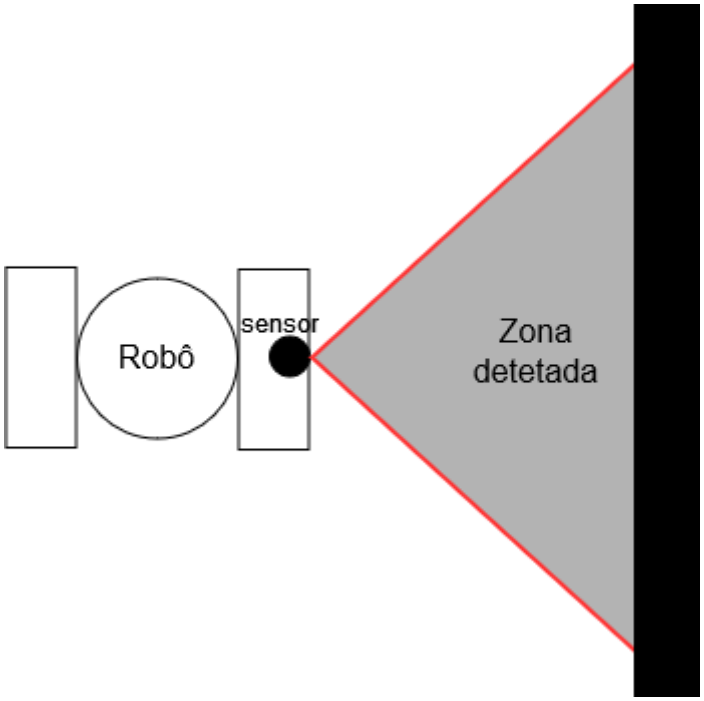


**Figura 16 - Funcionamento do sensor ultrassons**

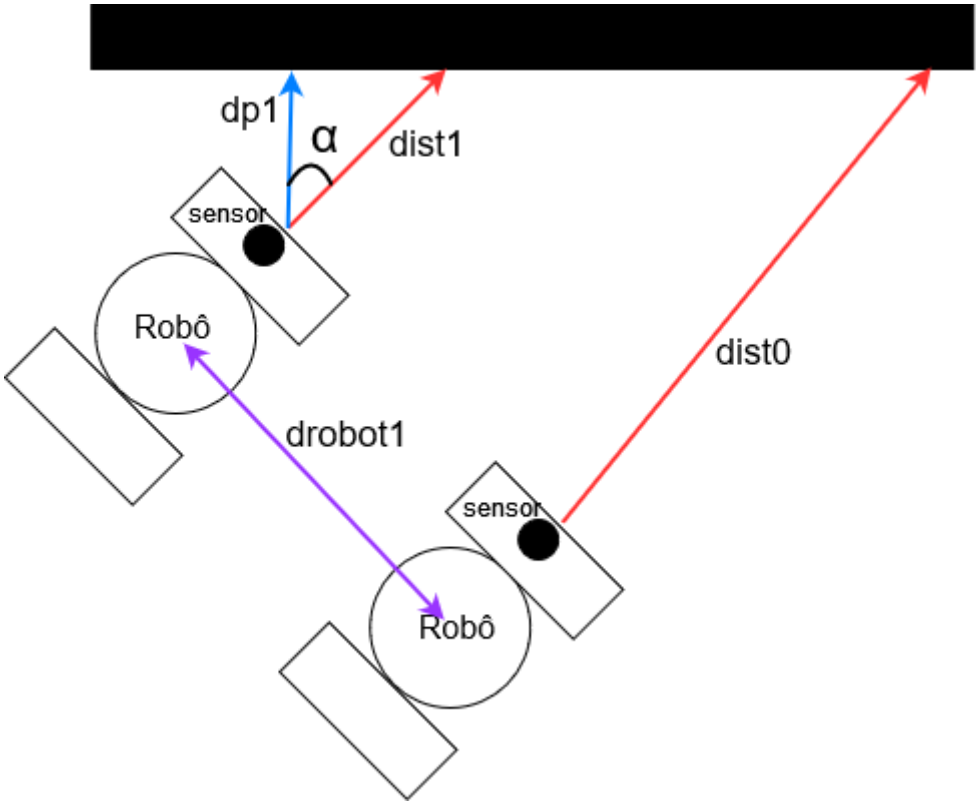


**Figura 17 - Comportamento do robô a seguir a parede**

Para implementar o comportamento de seguir a parede existem duas estratégias, onde ambas começam por primeiro medir o ângulo $\alpha$ relativamente com a parede. Para isso é medida a distância do robô com a parede, de seguida ele anda uma determinada distância, e mede novamente a distância entre ele e a parede. A primeira medida é dada pelo $dist0$ e a segunda medida pelo $dist1$. O que permite obter a distância de um objeto relativamente com o sensor é o método `SensorUS` do `InterpretadorEV3`. A distância percorrida pelo robô entre a medição do $dist0$ e $dist1$ é de 20 centímetros, pois é um valor grande o suficiente para calcular bem as distâncias e evitar colidir com a parede durante o percurso.

Contudo caso o robô colida com uma parede ou objeto ele recua 70 centímetros e curva 90°. Isto é valido em qualquer altura de movimento do robô, tanto quando está a calcular distâncias como quando está a executar a trajetória. Uma trajetória por ser constituída por retas e curvas ele pode colidir tanto numa curva como numa reta e por isso é necessário verificar as colisões em ambos.

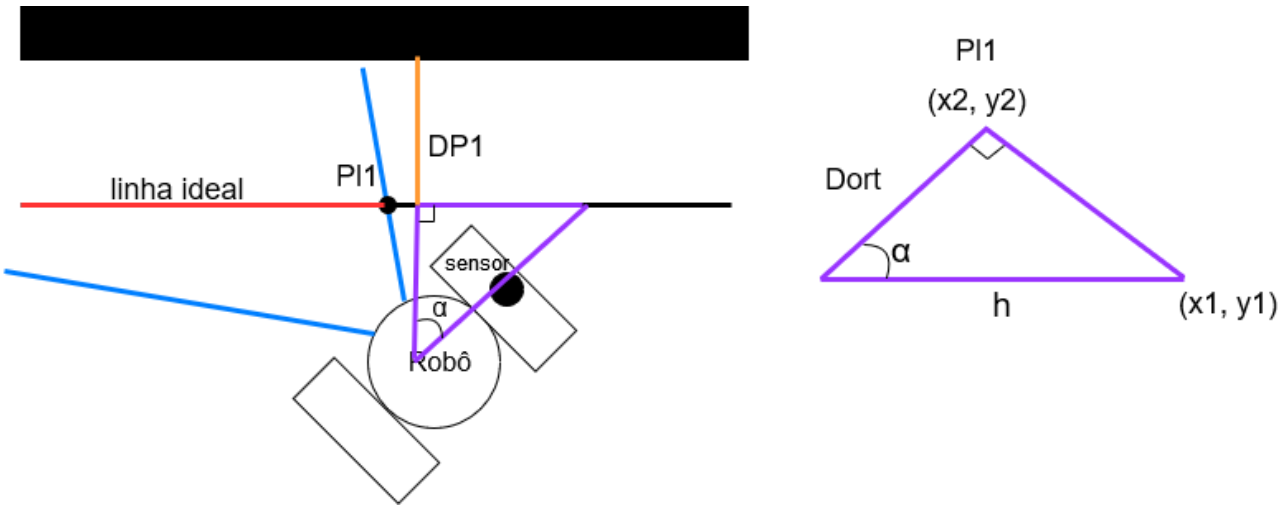


**Figura 18 - Linha ideal vista pelo robô**

Para calcular o ângulo $\alpha$ é utilizada a seguinte fórmula:

$$Tang(a) = \frac{|dist0 - dist1|}{drobot1}$$

$$a = arc\ tang(\frac{|dist0 - dist1|}{drobot1})$$

Para conseguir saber se o robô está mais perto ou mais longe da parede relativamente com a linha ideal é utilizado o $dp1$ que utiliza a seguinte fórmula:

$$\cos(\alpha) = \frac{dp1}{dist1}$$

$$dp1 = dist1 * \cos(a)$$

Sempre que o robô está entre a linha ideal e a parede o dp1 é sempre inferior ou igual à distância da linha ideal, ou seja, $dp1 \leq distância\ ideal$. Caso o robô esteja a uma distância superior a linha ideal então $dp1 > distância\ ideal$.

De seguida temos:

- A primeira estratégia consiste em colocar o robô paralelo com a parede, isto é feito através de uma curva para a esquerda com um determinado raio. No caso do trabalho prático foi escolhido um raio de 10 de forma a robô rodar sobre ele próprio.
- A segunda estratégia é colocar o robô na linha ideal utilizando imediatamente uma trajetória. No caso do trabalho prático foi escolhida esta estratégia pois foram testadas as duas estratégias e produz os melhores resultados.

De seguida é necessário calcular a equação da reta da linha ideal. Para isso utiliza-se primeiro o triângulo para obter as coordenadas dos pontos e calcular a equação e para isso sabe-se que:

$distância\ ideal\ =\ 50$
$x1 = 0$
$y1 = \ -h$
$x2 = Dort * sen(\alpha)$
$y2 = Dort * cos(\alpha)$
$Dort = dp1 - distância\ ideal$

Para calcular a equação de reta é utilizado a seguinte fórmula, onde temos:

$x = 0, y = b = \ -h$

$$y = m * x + b$$

$$m = \frac{y2 - y1}{x2 - x1} = \frac{Dort * \cos(\alpha) + h}{Dort * sen(\alpha)}$$

A partir da equação da reta é possível calcular qualquer ponto na reta e por isso a coordenada escolhida foi sempre o valor do $x2\ + 40$. Foi testado com valores inferiores a 40 e o robô realiza curvas muito apertadas e muitas vezes não conseguia ficar na linha ideal, por isso foi optado por um valor superior. A partir deste valor é calculado o valor de y a partir da equação calculada anteriormente.

A equação da reta em cima parte da premissa que o robô está para lá da linha ideal, ou seja, o $dp1$ é maior que a distância da linha ideal. No caso em que o $dp1$ seja menor que a distância da linha ideal o valor do $b$ da equação passa a ser $h$.

$b = h$

### 2.2.2 Implementação do seguir parede

A implementação do comportamento de seguir a parede foi toda ela realizada na classe `FollowWall`. Esta classe é responsável por calcular as distâncias do robô em relação com a parede e o ângulo relativo entre o robô e a parede. A partir destes valores é possível calcular a equação da reta da linha ideal e assim aplicar a trajetória para um ponto escolhido que coincida com a equação da linha ideal.

Desta maneira a classe `FollowWall` irá então precisar do gestor de trajetórias para conseguir realizar uma trajetória para um determinado ponto e precisa do `MyRobotLego` para conseguir realizar a reta de 40 centímetros para obter as distâncias. O método responsável por obter um ponto em `Xf`, `Yf` e `Of` a partir da equação da reta da linha ideal e utilizar o gestor para calcular uma trajetória para ir para o ponto pretendido é designado `calculate()`. De seguida dentro do ciclo de execução de seguir a parede, feito na interface gráfica, é utilizado o gestor para executar os comandos calculados na trajetória.

Em tempo de execução do comportamento de seguir parede para parar é preciso clicar no botão parar da interface gráfica. Contudo o comportamento só irá acabar após a última trajetória ser realizada.

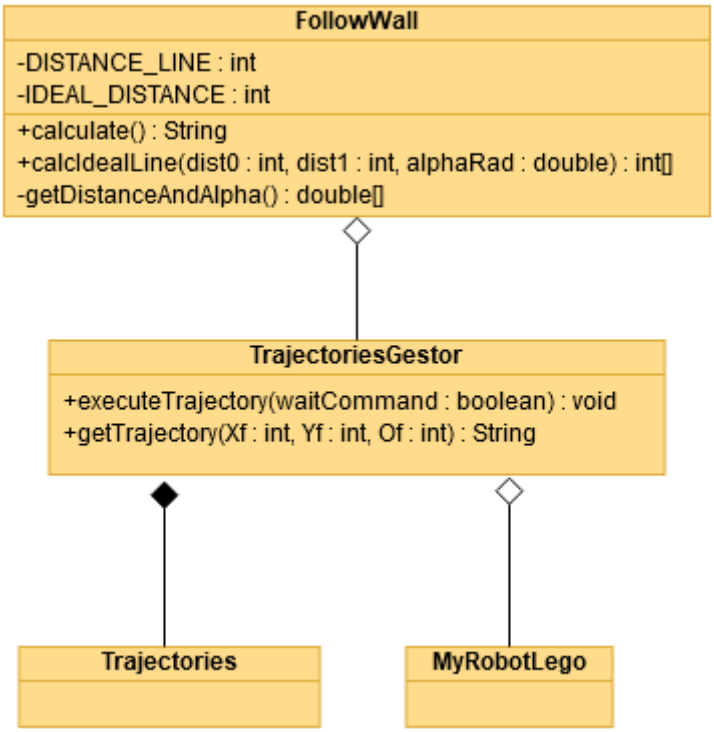


**Figura 19 - Diagrama de classes do seguir parede**

## 2.3 Alterações ao primeiro trabalho

Como o segundo trabalho agora possuí requisitos adicionais foi necessário fazer algumas alterações à implementação realizada no primeiro trabalho. Estas alterações foram feitas tanto na interface gráfica como na própria implementação da classe `MyRobotLego`.

### 2.3.1 Alterações na interface gráfica

Para ser possível realizar trajetórias e também por o robô a calcular pontos para seguir foi necessário adicionar mais componentes na interface gráfica. As componentes são nomeadamente:

- Três caixas de texto que corresponde ao ponto objetivo que é pretendido o robô ir, onde existe um $Xf$ que corresponde à distância que será realizada para a frente do robô. Outro é o $Yf$ que corresponde à distância feita para o lado esquerdo ou direito. Caso a caixa

de texto possua uma coordenada positiva ele movimenta-se para a esquerda senão movimenta-se para a direita. Por fim existe um `Of` que corresponde à rotação do robô relativamente à sua rotação inicial.

- Um botão cinzento que realiza uma trajetória, das três existentes, a partir dos valores que se encontram nas caixas de texto referidas anteriormente. Quando este botão é selecionado todos os botões são desativos e só voltam a ser ativos após concluir a trajetória para prevenir erros.
- Um botão laranja que permite por o robô a explorar o mundo e sempre que encontra uma parede ele começa a seguir a parede. Quando é selecionado o comportamento de seguir a parede ele desativa o próprio botão e das trajetórias para prevenir erros.
- Uma funcionalidade alterada na interface gráfica foi quando o *debug* está ativo o robô executa cada comando e logo de seguida é enviado um comando `stop()` e só envia um novo comando passados dois segundos. Desta forma é possível mais facilmente verificar se o robô está ou não a cumprir os comandos que foram enviados.

O resultado final da interface gráfica obtido foi o da figura que se segue:

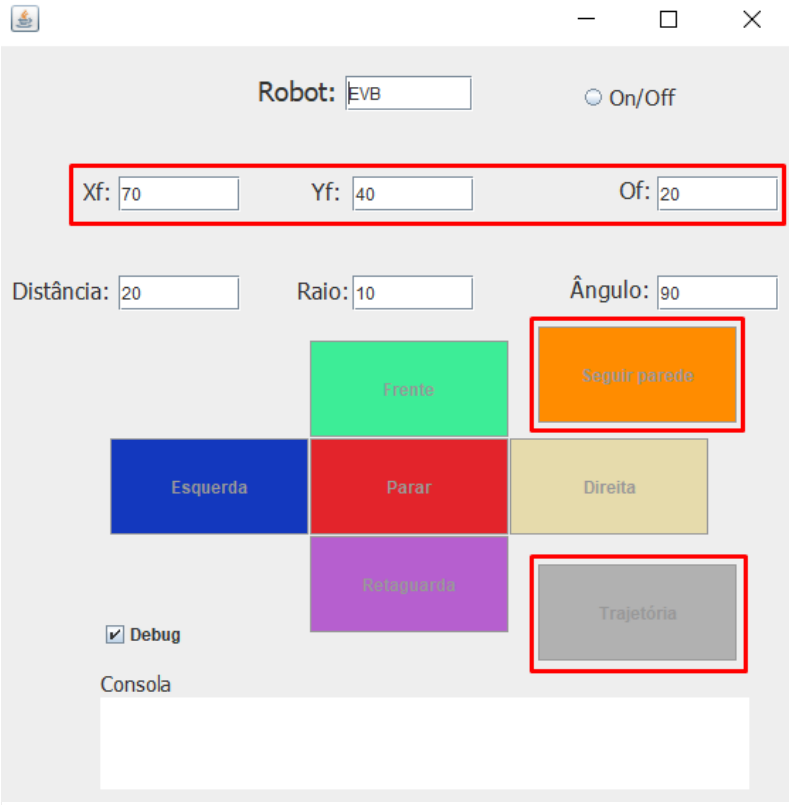


**Figura 20 - Interface gráfica alterada**

Todas as outras componentes mantiveram-se iguais ao que foi implementado no primeiro trabalho.

### 2.3.2 Alterações na implementação da classe responsável por mover o robô (`MyRobotLego`)

O comportamento responsável por seguir a parede requer funcionalidades adicionais que não constavam na implementação feita no primeiro trabalho. As funcionalidades extra necessárias são nomeadamente:

- Quando o robô está a realizar um comando de reta para frente ou para trás ou uma curva para a direita ou esquerda ele pode colidir contra uma parede. Isto deve-se a que o sensor de ultrassons encontra-se do lado direito do robô fazendo com que o robô não consiga ter uma perceção da distância dos objetos à sua frente. Por isso existe um sensor de colisão à frente do robô e foi necessário alterar os métodos `line()` e `curve()` para verificarem se o robô colidiu ou não.
- O robô conseguir executar comandos seguidos sem ser necessário estar sempre a enviar um comando `stop()` para ele parar. Isto poderia logo ter sido implementado no primeiro trabalho, mas o grupo só viu a utilidade desta funcionalidade quando estava a executar o comando de seguir a parede pois o robô não possuía um movimento fluído.

Para implementar a deteção de colisão nos métodos `line()` e `curve()` primeiro foi necessário converter estes métodos de *void* para *boolean* onde o retorno destes métodos é se colidiu ou não. Caso os métodos retornem verdade é porque o robô colidiu caso contrário não colidiu. Uma outra alteração realizada foi nos argumentos onde ambos os métodos agora recebem um booleano designado *sensor* que quando é verdade os métodos verificam se o robô colidiu caso contrário não verifica. Isto deve-se a que só é necessário verificar se o robô colidiu nas trajetórias e não quando se envia comandos isolados. O que permite detetar as colisões do sensor é o método `SensorTouch()` do `InterpretadorEV3`. Caso o robô detete que colidiu o método termina e retorna verdade, senão continua a correr e caso nunca colida termina

quando realiza o número total de rotações pretendidas.

A implementação feita no primeiro trabalho consistia numa implementação que o robô só conseguia realizar um comando de cada vez. Isto porque assumia-se que o tacómetro (contador de rotações) do robô começava sempre em 0. Por isso o robô só conseguia realizar corretamente o primeiro comando, onde o segundo ou não era executado ou era executado uma pequena parte visto que o robô guarda em memória as rotações realizadas. Por isso para enviar um segundo comando era necessário chamar sempre o comando `stop()`. Para resolver este problema logo no início do método são obtidas as rotações atuais do robô e adiciona-se esse valor à quantidade de rotações que serão feitas pelo robô, desta forma o robô consegue realizar múltiplos comandos seguidos sem ser necessário chamar o método `stop()`.

O resultado da máquina de estados do método `line()` foi o seguinte:

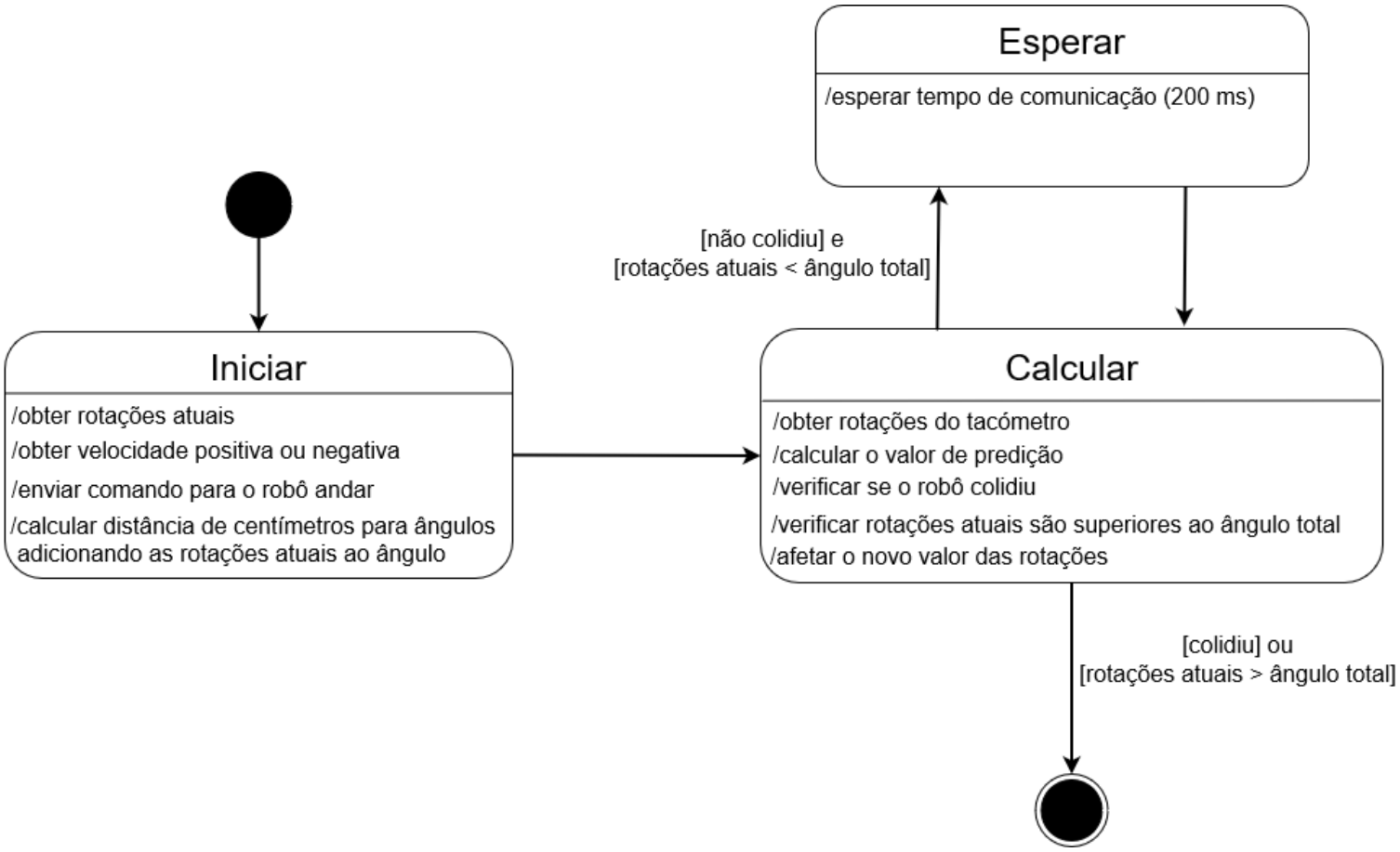


**Figura 21 – Diagrama de estados da máquina de estados do método `line()`**

Além do método `line()` também foi necessário alterar o método `curve()` e os métodos que utilizam o método `curve()`, nomeadamente `curveLeft()` e `curveRight()`. Os ajustes aos métodos foram iguais aos da `line()`, onde foi convertidos todos os métodos para booleanos e recebem como argumento um sensor. O resultado obtido foi:

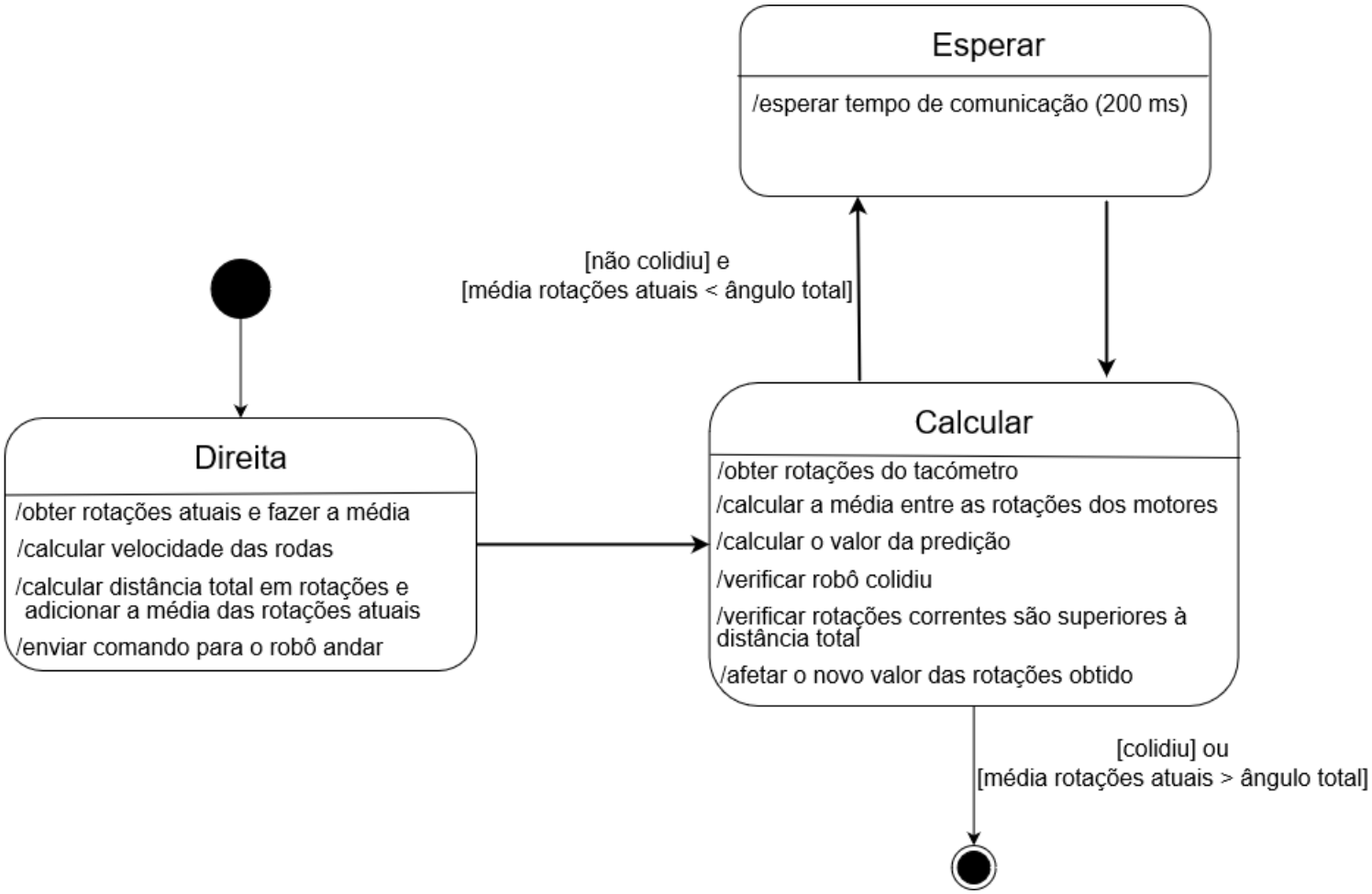


**Figura 22 - Diagrama de estados da máquina de estados do método `curve()`**

## 2.4 Implementação completa

A implementação completa do sistema funciona de tal maneira que existe a interface gráfica que possuí o `main` e existe uma classe de encapsulamento para os diferentes comportamentos designada `Variables`. Esta classe possuí métodos de `get()` para as diferentes classes de forma a estruturar o trabalho. O resultado das classes final é o que se segue:

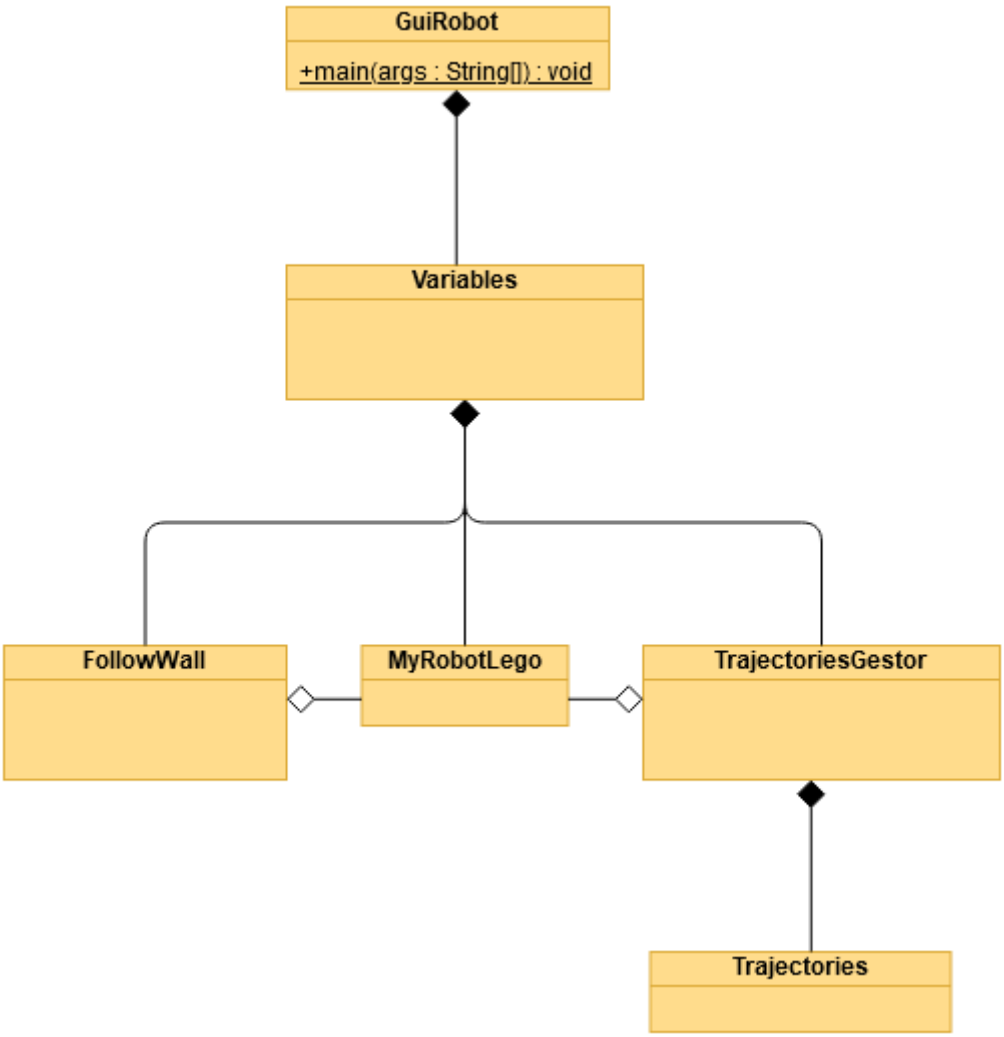


**Figura 23 - Diagrama de classes completo**

# 3. Conclusões

Em suma, com a realização do presente trabalho prático foi possível desenvolver os seguintes conhecimentos:

- Utilização do robô Lego EV3 para desenvolver conhecimentos na área da robótica.
- Utilização de triangulação, nomeadamente procurar triângulos retângulos, para calcular distâncias e ser possível realizar uma trajetória.
- Necessidade de utilização de modelos matemáticos para desenvolver formas de mover o robô para uma determinada coordenada no mundo real.
- Necessidade de utilização de diversas trajetórias para realizar um comportamento mais complexo, de forma a minimizar o erro.
- Realização de um comportamento de seguir parede a partir das trajetórias e cálculos de equação da reta.
- Utilização da biblioteca Swing para desenvolver uma interface gráfica para ligar, desligar e enviar comandos para o robô.
- Utilização de máquina de estados para facilitar a elaboração do código funcional dos comandos do robô.
- Verificação do erro associado às trajetórias para ver se o comportamento está correto.
- Necessidade de possuir um mecanismo de *debug* para os comandos executados para o robô.

# 4. Bibliografia

[1] Folhas Pais J., Introdução à Robótica e Robot EV3 da Lego, ISEL, 2021/22

[2] Folhas Pais J., Trajetória 1 teórica e prática, 2021/22

[3] Folhas Pais J., Trajetória 2 teórica, 2021/22

[4] Folhas Pais J., Trajetória 2 prática, 2021/22

[5] Folhas Pais J., Trajetória 3 teórica, 2021/22

[6] Folhas Pais J., Trajetória 3 prática, 2021/22

[7] Folhas Pais J., Cálculo automático de pontos no espaço, 2021/22

# 5. Anexo

```
import javax.swing.JFrame;
import javax.swing.JButton;

import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.ActionEvent;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

import java.awt.Color;
import java.awt.Font;
import javax.swing.JRadioButton;
import javax.swing.JTextArea;
import javax.swing.JCheckBox;

public class GuiRobot implements Runnable{

	private JFrame frame;

	private boolean connected = false;
	private boolean followWall = false;

	// text fiels to write distances
	private JTextField fieldRobotName;
	private JTextField fieldXf;
	private JTextField fieldYf;
	private JTextField fieldOf;
	private JTextField fieldDistance;
	private JTextField fieldRay;
	private JTextField fieldAngle;

	// text area to show the commands
	private JTextArea areaConsole;
```

```
// buttons to execute commands
private JButton btnFront;
private JButton btnRight;
private JButton btnLeft;
private JButton btnBack;
private JButton btnStop;
private JButton btnTrajectory;
private JButton btnFollowWall;

private JRadioButton onOff;
private JCheckBox checkBoxDebug;

// variables used
private Variables variables;


public static void main(String[] args) {
      // runs the GUI as thread to not block the commands
      GuiRobot gr = new GuiRobot();
      Thread t = new Thread(gr);
      t.start();
}

public GuiRobot() {
      variables = new Variables();
}

public void run() {
      initializeGui();

}

private void initializeGui() {
      frame = new JFrame();
      frame.setBounds(100, 100, 580, 585);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.getContentPane().setLayout(null);
```

```
        // create the elements of the interface
        createButtons();
        createRobotName();
        createDistance();
        createOnOffBtn();
        createRay();
        createAngle();
        createConsole();
        createDebug();


        createXf();
        createYf();
        createOf();

        frame.setVisible(true);
    }


    // create all the buttons
    private void createButtons() {
        // left button
        btnLeft = new JButton("Esquerda");
        btnLeft.setBounds(79, 277, 140, 68);
        btnLeft.setBackground(new Color(19, 56, 190));
        btnLeft.setEnabled(false);
        frame.getContentPane().add(btnLeft);

        // right button
        btnRight = new JButton("Direita");
        btnRight.setBounds(361, 277, 140, 68);
        btnRight.setBackground(new Color(230, 219, 172));
        btnRight.setEnabled(false);
        frame.getContentPane().add(btnRight);

        // stop button
        btnStop = new JButton("Parar");
```

```
btnStop.setBounds(220, 277, 140, 68);
btnStop.setBackground(new Color(227, 36, 43));
btnStop.setEnabled(false);
frame.getContentPane().add(btnStop);

// front button
btnFront = new JButton("Frente");
btnFront.setBounds(220, 208, 140, 68);
btnFront.setBackground(new Color(61, 237, 151));
btnFront.setEnabled(false);
frame.getContentPane().add(btnFront);

// back button
btnBack = new JButton("Retaguarda");
btnBack.setBounds(220, 346, 140, 68);
btnBack.setBackground(new Color(182, 95, 207));
btnBack.setEnabled(false);
frame.getContentPane().add(btnBack);



btnTrajectory = new JButton("Trajetória");
btnTrajectory.setBounds(381, 366, 140, 68);
btnTrajectory.setBackground(new Color(177, 177, 177));
btnTrajectory.setEnabled(false);
frame.getContentPane().add(btnTrajectory);

btnFollowWall = new JButton("Seguir parede");
btnFollowWall.setBounds(381, 198, 140, 68);
btnFollowWall.setBackground(new Color(255,140,0));
btnFollowWall.setEnabled(false);
frame.getContentPane().add(btnFollowWall);

// listeners for the buttons
listenerBtnLeft();
listenerBtnRight();
listenerBtnFront();
listenerBtnBack();
listenerBtnStop();
```

```
            listenerBtnTrajectories();
            listenerBtnFollowWall();

            // listener when close the connection with robot
            listenerOnClose();
      }

      private void createOnOffBtn() {
            // Radio button to turn robot on or off
            onOff = new JRadioButton("On/Off");
            onOff.setFont(new Font("Tahoma", Font.PLAIN, 15));
            onOff.setBounds(410, 25, 109, 23);
            onOff.setSelected(variables.isOnOff());
            frame.getContentPane().add(onOff);

            listenerConnectRobot();
      }

      // create the text and field to write the name
      private void createRobotName() {
            // label to show text of the robot name
            JLabel labelRobot = new JLabel("Robot:");
            labelRobot.setFont(new Font("Tahoma", Font.PLAIN, 19));
            labelRobot.setBounds(183, 13, 67, 37);
            frame.getContentPane().add(labelRobot);

            // field to set the robot name
            fieldRobotName = new JTextField("" + variables.getNomeRobot());
            fieldRobotName.setBounds(245, 21, 90, 25);
            frame.getContentPane().add(fieldRobotName);

            listenerFieldRobotName();
      }

      private void createXf() {
            JLabel label = new JLabel("Xf:");
            label.setFont(new Font("Tahoma", Font.PLAIN, 16));
            label.setBounds(60, 92, 80, 23);
```

```
        frame.getContentPane().add(label);

        fieldXf = new JTextField(""+variables.getXf());
        fieldXf.setBounds(85, 92, 86, 25);
        frame.getContentPane().add(fieldXf);

        listenerXf();
    }

    private void createYf() {
        JLabel label = new JLabel("Yf:");
        label.setFont(new Font("Tahoma", Font.PLAIN, 16));
        label.setBounds(221, 95, 67, 17);
        frame.getContentPane().add(label);

        fieldYf = new JTextField(""+ variables.getYf());
        fieldYf.setBounds(250, 92, 86, 25);
        frame.getContentPane().add(fieldYf);

        listenerYf();
    }

    private void createOf() {
        JLabel labelAngle = new JLabel("Of:");
        labelAngle.setFont(new Font("Tahoma", Font.PLAIN, 16));
        labelAngle.setBounds(438, 92, 58, 20);
        frame.getContentPane().add(labelAngle);

        fieldOf = new JTextField(""+variables.getOf());
        fieldOf.setBounds(465, 92, 86, 25);
        frame.getContentPane().add(fieldOf);

        listenerOf();
    }

    // text and field of distance
    private void createDistance() {
        // label to show text of distância
```

```
        JLabel labelDistance = new JLabel("Dist\u00E2ncia:");
        labelDistance.setFont(new Font("Tahoma", Font.PLAIN, 16));
        labelDistance.setBounds(10, 162, 80, 23);
        frame.getContentPane().add(labelDistance);

        // field to write the distance
        fieldDistance = new JTextField("" + variables.getDistance());
        fieldDistance.setBounds(85, 162, 86, 25);
        frame.getContentPane().add(fieldDistance);

        listenerDistance();
    }

    private void createRay() {
        // text to show raio
        JLabel labelRay = new JLabel("Raio:");
        labelRay.setFont(new Font("Tahoma", Font.PLAIN, 16));
        labelRay.setBounds(211, 165, 67, 17);
        frame.getContentPane().add(labelRay);

        // field to write the ray
        fieldRay = new JTextField("" + variables.getRadius());
        fieldRay.setBounds(250, 162, 86, 25);
        frame.getContentPane().add(fieldRay);

        listenerRay();
    }

    private void createAngle() {
        // text to write ângulo
        JLabel labelAngle = new JLabel("\u00C2ngulo:");
        labelAngle.setFont(new Font("Tahoma", Font.PLAIN, 16));
        labelAngle.setBounds(403, 162, 58, 20);
        frame.getContentPane().add(labelAngle);

        // field to write the angle
        fieldAngle = new JTextField("" + variables.getAngle());
        fieldAngle.setBounds(465, 162, 86, 25);
```

```
		frame.getContentPane().add(fieldAngle);

		listenerAngle();
	}

	private void createConsole() {
		JLabel labelConsole = new JLabel("Consola");
		labelConsole.setFont(new Font("Tahoma", Font.PLAIN, 15));
		labelConsole.setBounds(72, 440, 58, 20);
		frame.getContentPane().add(labelConsole);

		areaConsole = new JTextArea();
		areaConsole.setBounds(72, 460, 458, 65);
		areaConsole.setEditable(false);
		areaConsole.setFont(areaConsole.getFont().deriveFont(14f));
		frame.getContentPane().add(areaConsole);
	}

	private void createDebug() {
		checkBoxDebug = new JCheckBox("Debug");
		checkBoxDebug.setBounds(72, 404, 97, 23);
		checkBoxDebug.setSelected(variables.isDebug());
		frame.getContentPane().add(checkBoxDebug);

		listenerToDebug();
	}

	private void listenerFieldRobotName() {
		fieldRobotName.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				variables.setNomeRobot(fieldRobotName.getText());
				myPrint("Nome:"+ variables.getNomeRobot());
			}
		});
	}

	private void listenerBtnLeft() {
		btnLeft.addActionListener(new ActionListener() {
```

```
                public void actionPerformed(ActionEvent e) {
                      new Thread() {
                            public void run() {
                                  myPrint("Esquerda: " +
String.valueOf(variables.getAngle()) + " "+
String.valueOf(variables.getRadius()));
                                  // TODO
                                  if (variables.getMyImplentation()) {

      variables.getMyRobot().curveLeft(variables.getAngle(),
variables.getRadius(), true);
                                        variables.getMyRobot().stop();


                                  } else {

      variables.getRobot().CurvarEsquerda(variables.getRadius(),
variables.getAngle());
                                        variables.getRobot().Parar(false);
                                  }
                            }
                      }.start();
                }
            });
      }


      private void listenerBtnRight() {
            btnRight.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                      new Thread() {
                            public void run() {
                                  myPrint("Direita: " +
String.valueOf(variables.getAngle()) + " "+
String.valueOf(variables.getRadius()));
                                  // TODO
                                  if (variables.getMyImplentation()) {

      variables.getMyRobot().curveRight(variables.getAngle(),
variables.getRadius(), true);
                                        variables.getMyRobot().stop();
                                  } else {

      variables.getRobot().CurvarDireita(variables.getRadius(),
```

```
variables.getAngle());
                                        variables.getRobot().Parar(false);
                                  }
                            }
                      }.start();
                }
          });
    }

    private void listenerBtnFront() {
          btnFront.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                      // creates a thread that makes the robot goes forward
                      new Thread() {
                          public void run() {
                            myPrint("Frente: " + variables.getDistance());
                                  // TODO
                                  if (variables.getMyImplentation()) {

    variables.getMyRobot().line(variables.getDistance(), false);
                                        variables.getMyRobot().stop();


                                  } else {

    variables.getRobot().Reta(variables.getDistance());
                                        variables.getRobot().Parar(false);
                                  }
                            }
                      }.start();
                }
          });
    }

    private void listenerBtnStop() {
          btnStop.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                      new Thread() {
                            public void run() {
                                  myPrint("Parar");
```

```
                                    // TODO
                                    if (variables.getMyImplentation())
                                          variables.getMyRobot().stop();

                                    else
                                          variables.getRobot().Parar(true);

                                    followWall = false;
                              }
                        }.start();
                  }
            });
      }

      private void listenerBtnBack() {
            btnBack.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        new Thread() {
                              public void run() {
                                    myPrint("Retaguarda: " +
String.valueOf(variables.getDistance()));
                                    // TODO
                                    if (variables.getMyImplentation()) {

      variables.getMyRobot().line(variables.getDistance() * (-1), false);
                                          variables.getMyRobot().stop();
                                    } else {

      variables.getRobot().Reta(variables.getDistance() * (-1));
                                          variables.getRobot().Parar(false);
                                    }
                              }
                        }.start();
                  }

            });
      }

      private void listenerConnectRobot() {
```

```
            onOff.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        if (onOff.isSelected()) {
                              myPrint("Conexão com robô");
                              //TODO
                              if(variables.getMyImplentation()) connected =
variables.getMyRobot().openEV3(variables.getNomeRobot());

                              else connected =
variables.getRobot().OpenEV3(variables.getNomeRobot());

                              // set the interface state
                              setButtinsState(connected);
                        }
                        else {
                              myPrint("Disconexão com robô");
                              // TODO
                              if(variables.getMyImplentation())
variables.getMyRobot().closeEV3();

                              else variables.getRobot().CloseEV3();

                              onOff.setSelected(false);
                              variables.setOnOff(false);
                              connected = false;
                              // set the interface state
                              setButtinsState(false);
                              followWall = false;
                        }
                        try {
                              Thread.sleep(100);
                        } catch (InterruptedException e1) {
                              e1.printStackTrace();
                        }

                  }
            });
      }
```

```
	public void listenerOnClose() {
		frame.addWindowListener(new WindowAdapter() {
		    @Override
		    public void windowClosing(WindowEvent windowEvent) {
		      if(connected) {
				// disconnect the robot
				if(variables.getMyImplentation())
variables.getMyRobot().closeEV3();

				else variables.getRobot().CloseEV3();
				followWall = false;
		      }
		    }
		});
	}

	private void listenerBtnTrajectories() {
		btnTrajectory.addActionListener(new ActionListener() {
		      public void actionPerformed(ActionEvent e) {
				new Thread() {
				      public void run() {
						setButtinsState(false);
						// gets the string with the values of the
commands that will be done
						String trajectories =
variables.getTrajectoriesGestor()
                                                .getTrajectory(variables.getXf(),
variables.getYf(), variables.getOf());


						myPrint(trajectories);


						// executes the commands

	variables.getTrajectoriesGestor().executeTrajectory(checkBoxDebug.isSelect
ed());
						variables.getMyRobot().stop();
						setButtinsState(true);
				      }
				}.start();
```

```
                }
            });
      }

      private void listenerBtnFollowWall() {
            btnFollowWall.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        new Thread() {
                              public void run() {
                                    // disable buttons to prevent errors
                                    btnFollowWall.setEnabled(false);
                                    btnTrajectory.setEnabled(false);
                                    followWall = true;
                                    while(followWall) {
                                          String result =
variables.getFollowWall().calcutate();
                                          myPrint("Follow Wall: "+result);

                                          // executes the commands

      variables.getTrajectoriesGestor().executeTrajectory(checkBoxDebug.isSelect
ed());
                                    }
                                    // enable after the execution
                                    btnFollowWall.setEnabled(true);
                                    btnTrajectory.setEnabled(true);
                              }
                        }.start();
                  }
            });
      }

      private void listenerXf() {
            fieldXf.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        // verifies if the sting only contains numbers, otherwise
shows a message
                        if(fieldXf.getText().matches("[0-9\\-]*")) {
                              int xf = Integer.parseInt(fieldXf.getText());
```

```
                        if(xf <= 400) {
                              variables.setXf(xf);
                              myPrint("Xf:"+ variables.getXf());
                        }
                        else JOptionPane.showMessageDialog(frame, "A
distância só pode ser no máximo 400");//
                  }

                  else JOptionPane.showMessageDialog(frame, "A distância só
pode conter números");//

            }
      });
}

private void listenerYf() {
      fieldYf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                  // verifies if the sting only contains numbers, otherwise
shows a message
                  if(fieldYf.getText().matches("[0-9\\-]*")) {
                        int yf = Integer.parseInt(fieldYf.getText());
                        if(yf <= 400) {
                              variables.setYf(yf);
                              myPrint("Yf:"+ variables.getYf());
                        }
                        else JOptionPane.showMessageDialog(frame, "A
distância só pode ser no máximo 400");//
                  }

                  else JOptionPane.showMessageDialog(frame, "A distância só
pode conter números");//

            }
      });
}

private void listenerOf() {
      fieldOf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```

```
                        // verifies if the sting only contains numbers, otherwise
shows a message
                        if(fieldOf.getText().matches("[0-9\\-]*")) {
                              int of = Integer.parseInt(fieldOf.getText());
                              if(of <= 400) {
                                    variables.setOf(of);
                                    myPrint("Of:"+ variables.getOf());
                              }
                              else JOptionPane.showMessageDialog(frame, "A
distância só pode ser no máximo 400");//
                        }

                        else JOptionPane.showMessageDialog(frame, "A distância só
pode conter números");//

                  }
            });
      }

      private void listenerDistance() {
            fieldDistance.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        // verifies if the sting only contains numbers, otherwise
shows a message
                        if(fieldDistance.getText().matches("[0-9]*")) {
                              int dist =
Integer.parseInt(fieldDistance.getText());
                              if(dist <= 400) {
                                    variables.setDistance(dist);
                                    myPrint("Distancia:"+
variables.getDistance());
                              }
                              else JOptionPane.showMessageDialog(frame, "A
distância só pode ser no máximo 400");//
                        }

                        else JOptionPane.showMessageDialog(frame, "A distância só
pode conter números");//

                  }
            });
```

```
	}

	private void listenerRay() {
		fieldRay.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if(fieldRay.getText().matches("[0-9]*")) {
					int radius = Integer.parseInt(fieldRay.getText());
					if(radius <= 50) {
						variables.setRadius(radius);
						myPrint("Raio:"+ variables.getRadius());
					}
					else JOptionPane.showMessageDialog(frame, "O raio
só pode ser no máximo 50");//
				}

				else JOptionPane.showMessageDialog(frame, "O raio só pode
conter números");//
			}
		});
	}

	private void listenerAngle() {
		fieldAngle.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if(fieldAngle.getText().matches("[0-9]*")) {
					int angle = Integer.parseInt(fieldAngle.getText());
					if(angle <= 720) {
						variables.setAngulo(angle);
						myPrint("Angulo:"+ variables.getAngle());
					}
					else JOptionPane.showMessageDialog(frame, "O angulo
só pode ser no máximo 720");//
				}
				else JOptionPane.showMessageDialog(frame, "O angulo só
pode conter números");//
			}
		});
	}
```

```
	private void listenerToDebug() {
		checkBoxDebug.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				variables.setDebug(checkBoxDebug.isSelected());
			}
		});
	}


	public void myPrint(String str) {
		areaConsole.setText(str);
	}

	public void setButtinsState(boolean result) {
		onOff.setSelected(result);
		variables.setOnOff(result);
		btnLeft.setEnabled(result);
		btnRight.setEnabled(result);
		btnFront.setEnabled(result);
		btnStop.setEnabled(result);
		btnBack.setEnabled(result);
		btnTrajectory.setEnabled(result);
		btnFollowWall.setEnabled(result);
	}

}
```

```
public class Variables {

	// variable used to define if will use the prof library or
	// our implementation
	private final boolean MY_IMPLEMENTATION = true;

	private String robotName;
	private boolean onOff, debug;
	private int radius, angle, distance;
	private int xf, yf, of;
	private RobotLegoEV3 robot;
	private MyRobotLego myRobot;
	private TrajectoriesGestor gestor;
	private FollowWall follow;


	public Variables() {
		robotName= "EVB";
		onOff = false;
		debug = true;
		radius = 10;
		angle = 90;
		distance = 20;
		xf = 70;
		yf = 40;
		of = 20;
		robot = new RobotLegoEV3();
		myRobot = new MyRobotLego();
		gestor = new TrajectoriesGestor(myRobot);
		follow = new FollowWall(myRobot, gestor);

	}

	public String getNomeRobot() {
		return robotName;
	}

	public void setNomeRobot(String nomeRobot) {
```

```
		this.robotName = nomeRobot;
	}

	public boolean isOnOff() {
		return onOff;
	}

	public void setOnOff(boolean onOff) {
		this.onOff = onOff;
	}

	public boolean isDebug() {
		return debug;
	}

	public void setDebug(boolean debug) {
		this.debug = debug;
	}

	public int getRadius() {
		return radius;
	}

	public void setRadius(int radius) {
		this.radius = radius;
	}

	public int getAngle() {
		return angle;
	}

	public void setAngulo(int angle) {
		this.angle = angle;
	}

	public int getDistance() {
		return distance;
	}
```

```
public void setDistance(int distance) {
	this.distance = distance;
}

public int getXf() {
	return this.xf;
}

public void setXf(int Xf) {
	this.xf = Xf;
}

public int getYf() {
	return this.yf;
}

public void setYf(int Yf) {
	this.yf = Yf;
}

public int getOf() {
	return this.of;
}

public void setOf(int Of) {
	this.of = Of;
}

public RobotLegoEV3 getRobot() {
	return this.robot;
}

public MyRobotLego getMyRobot() {
	return this.myRobot;
}

public TrajectoriesGestor getTrajectoriesGestor() {
```

```
            return this.gestor;
      }

      public FollowWall getFollowWall() {
            return this.follow;
      }

      public boolean getMyImplentation() {
            return this.MY_IMPLEMENTATION;
      }

}
```

```
import java.util.concurrent.Semaphore;

public class MyRobotLego {

	public static final int VELOCITY = 50;

	// total distance between wheels
	public static final float WHEEL_CENTER_MASS = 9.5f;

	private InterpretadorEV3 interpretador;

	private final float WHEEL_RADIUS = 2.73f;

	// multiple possible states, to the line and curve
	private enum states {START, LEFT, RIGHT, CALCULATE, WAIT, END};

	// time of bluetooth to make communication
	private final int COMMUNICATION_TIME = 200;



	// obtain the right wheel
	private final int RIGHT_WHEEL = InterpretadorEV3.OUT_C;
	private final int LEFT_WHEEL = InterpretadorEV3.OUT_B;
	private final int RIGHT_LEFT_WHEELS = InterpretadorEV3.OUT_BC;
	private final int TOUCH_SENSOR = InterpretadorEV3.S_1;
	private final int US_SENSOR = InterpretadorEV3.S_2;

	// to prevent multiple threads access the multiple commands simultaneously
	private Semaphore semaphore;

	public MyRobotLego() {
		interpretador = new InterpretadorEV3();
		semaphore = new Semaphore(1);
	}

	// open communication with robot EV3
	public boolean openEV3(String robotName) {
		// because its used multi-threading, only one access at the same
time
```

```
            try {
                  semaphore.acquire();
            } catch (InterruptedException e1) {
                  e1.printStackTrace();
            }
            // connects to the robot
            boolean result = this.interpretador.OpenEV3(robotName);
            // releases the other threads
            semaphore.release();
            return result;
      }

      // close the communication with robot EV3
      public void closeEV3() {
            // because its used multi-threading
            try {
                  semaphore.acquire();
            } catch (InterruptedException e1) {
                  e1.printStackTrace();
            }

            // remove all the commands previously sent
            this.interpretador.ResetAll();
            try {
                  Thread.sleep(COMMUNICATION_TIME);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            this.interpretador.CloseEV3();
            // releases the thread to another thread uses
            semaphore.release();
      }

      // realizes the straight line
      // positive values go forward, negative values goes backwards
      // arg sensor is when we want to know with touch sensor if robot collided
or not
      // return is only useful when sensor is true
      // and returns true when robot collided
```

```
	public boolean line(int distance, boolean sensor) {
		// this way a thread will only start after do a command
		try {
			semaphore.acquire();
		} catch (InterruptedException e1) {
			e1.printStackTrace();
		}
		// current rotations to make the line
		int initialRotations =
this.interpretador.RotationCount(RIGHT_WHEEL);

		// angle that must be done by the robot
		double ang = 0;

		states currentState = states.START;

		// when the distance is positive goes forward
		// if negative goes backwards
		int positiveSignal = (distance > -1) ? 1 : -1;

		// variable used to prediction technique
		int deltaActual = initialRotations;

		// variable used to detect collision
		boolean collided =  false;

		while (currentState != states.END) {
			switch (currentState) {
			case START:
				// the value of the distance must be positive
				// so we use the abs to set positive the value
				ang = ((distance / WHEEL_RADIUS) * 180 /
Math.PI)+initialRotations;

				// sends the message to go forward
				this.interpretador.OnFwd(RIGHT_WHEEL, VELOCITY *
positiveSignal, LEFT_WHEEL, VELOCITY * positiveSignal);

				currentState = states.CALCULATE;
```

```
                        break;

                case CALCULATE:
                        int rotationCounts =
this.interpretador.RotationCount(RIGHT_WHEEL);

                        // prediction value
                        int deltaD = rotationCounts - deltaActual;

                        // verifies if collided or completed the rotations
                        // the last argument is when it goes back the value must
be less than before
                        int stateEnd = this.calculateStateEnd(rotationCounts,
ang, deltaD, sensor, (positiveSignal < 0));

                        // rotation count completes the task
                        // if sensor is on and collides goes back
                        // otherwise keep running
                        collided = (stateEnd == 2) ? true : false;
                        currentState = (stateEnd > 0) ? states.END : states.WAIT;


                        // sets the delta to the current rotations, to predict
the value and gets the
                        // least error possible
                        deltaActual = rotationCounts;
                        break;

                case WAIT:
                        // to not overload the robot with messages
                        // must have a delay
                        try {
                              Thread.sleep(COMMUNICATION_TIME);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                        currentState = states.CALCULATE;
                        break;
```

```
                }
            }

            // after complete of the command sends a release to other thread
starts running
            semaphore.release();

            //if no one sends the command to stop it keep on going until receive
a command to stop
            return collided;
      }

      // angle is in degrees, and radius is in cm
      public boolean curveRight(int angle, int radius, boolean sensor) {
            // this way a thread will only start after do a command
            try {
                  semaphore.acquire();
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }

            // makes the right curve
            boolean collided = curve(true, angle, radius, sensor);


            // after complete the command gives one permission to another thread
is freed
            semaphore.release();

            // if no one sends a stop command it keeps going until send another
command or
            // stop command
            return collided;
      }


      public boolean curveLeft(int angle, int radius, boolean sensor) {
            // this way a thread will only start after do a command
            try {
```

```
                semaphore.acquire();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // makes the left curve
            boolean collided = curve(false, angle, radius, sensor);

            // after complete the command gives one permission to another thread
is freed
            // if no one sends a stop command it keeps going until send another
command or stop command
            semaphore.release();

            return collided;
      }

      // creates a curve with state of curve right and left
      private boolean curve(boolean right, int angle, int radius, boolean
sensor) {

            states currentState = (right) ? states.RIGHT : states.LEFT;

            int[] currentRotations =
this.interpretador.RotationCount(RIGHT_WHEEL, LEFT_WHEEL);
            double meanRotations = Math.round((currentRotations[0] +
currentRotations[1]) / 2);

            // to detect when it collided
            boolean collision = false;

            double ang = 0;
            // variable used to prediction technique
            double deltaActual = meanRotations;

            while (currentState != states.END) {
                switch (currentState) {
                case RIGHT:
                      // calculations used to curve right
                      double radiusRight = radius - (WHEEL_CENTER_MASS / 2);
```

```
                        double radiusLeft = radius + (WHEEL_CENTER_MASS / 2);
                        double factor = radiusLeft / radiusRight;

                        int rightSpeed = (int) Math.round((VELOCITY * 2) / (1 +
factor));
                        int leftSpeed = (int) Math.round((factor * rightSpeed));

                        double totalDist = angle * radius;

                        ang = (Math.abs(totalDist) / WHEEL_RADIUS) +
meanRotations;

                        // the right wheel is C, and left
                        this.interpretador.OnFwd(RIGHT_WHEEL, rightSpeed,
LEFT_WHEEL, leftSpeed);
                        currentState = states.CALCULATE;
                        break;

                  case LEFT:
                        // calculations used to curve left
                        radiusRight = radius + (WHEEL_CENTER_MASS / 2);
                        radiusLeft = radius - (WHEEL_CENTER_MASS / 2);
                        factor = radiusRight / radiusLeft;
                        leftSpeed = (int) Math.round((VELOCITY * 2) / (1 +
factor));
                        rightSpeed = (int) Math.round((factor * leftSpeed));


                        totalDist = angle * radius;

                        ang = (Math.abs(totalDist) / WHEEL_RADIUS) +
meanRotations;

                        // the right wheel is C, and left
                        this.interpretador.OnFwd(RIGHT_WHEEL, rightSpeed,
LEFT_WHEEL, leftSpeed);
                        currentState = states.CALCULATE;
                        break;

                  case CALCULATE:
```

```
                        int[] rotationCounts =
this.interpretador.RotationCount(RIGHT_WHEEL, LEFT_WHEEL);
                        double result = Math.round((rotationCounts[0] +
rotationCounts[1]) / 2);

                        // prediction technique
                        double prediction = result - deltaActual;



                        int resultState = calculateStateEnd((int)result, ang,
(int)prediction, sensor, false);

                        // verify if the robot already complete the curve or not
with prediction
                        // or collided
                        collision = (resultState == 2) ? true : false;
                        currentState = (resultState > 0) ? states.END :
states.WAIT;

                        // used to store the previous value and make movement
prediction
                        deltaActual = result;
                        break;

                  case WAIT:
                        // to not overload the robot with messages
                        // must have a delay
                        try {
                              Thread.sleep(COMMUNICATION_TIME);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                        currentState = states.CALCULATE;
                        break;
                  }
            }

            return collision;
      }
```

```
	// stops the robot
	// it can also uses the Float command, but it gets a bigger error
	// and the first time it as used gets a really high error
	public synchronized void stop() {
		try {
			semaphore.acquire();
			//this.interpretador.Float(RIGHT_LEFT_WHEELS);
			this.interpretador.Off(RIGHT_LEFT_WHEELS);
			Thread.sleep(COMMUNICATION_TIME);
			this.interpretador.ResetAll();

		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		semaphore.release();
	}



	// returns 2 -> if the robot collided
	// returns 1 -> rotation count completed
	// returns 0 -> don't complete the execution
	private int calculateStateEnd(int rotationCounts, double ang, int deltaD,
boolean sensor, boolean reverse) {

		if(sensor) {
			int touched = this.interpretador.SensorTouch(TOUCH_SENSOR);
			if(touched == 1) return 2;

		}

		// when goes back the values must be less
		if(reverse) return (rotationCounts + deltaD < ang ) ? 1 : 0;

		// if the state is end to stops
		// or the rotation count completes the task
		return (rotationCounts + deltaD > ang ) ? 1 : 0;
	}

	public int getSensorUs() {
```

```
            return this.interpretador.SensorUS(US_SENSOR);
      }
}
```

```
public class Trajectories {

	private double yc1Tj2;
	private double yc2Tj2;

	private double yc1Tj3;
	private double yc2Tj3;

	private double alphaTj1;


	public int[] trajectory1(int Xf, int Yf, int Of) {
		//System.out.println("Tragetória 1");
		// must convert to radians, because the calculations is based on
radians
		double radOf = Math.toRadians(Of);


		// quadratic formula with a, b and c
		double a = 2 + 2 * Math.cos(radOf);
		double b = 2*Yf*(1- Math.cos(radOf))+2*Xf*Math.sin(radOf);
		double c = -(Math.pow(Xf, 2) + Math.pow(Yf, 2));

		// calculates the radius
		double radius = distanceBetweenPoints(a, b, c);


		// calculates the alpha
		double alpha = Math.toDegrees(Math.asin((Xf - radius *
Math.sin(radOf)) / (2 * radius)));
		this.alphaTj1 = alpha;
		//System.out.println("a: "+a);
		//System.out.println("b: "+b);
		//System.out.println("c: "+c);

		//System.out.println("x: "+radius);
		//System.out.println("alpha: "+alpha);

		double halfWheelCenterMass = MyRobotLego.WHEEL_CENTER_MASS/2;
```

```
            double factor = ((radius + halfWheelCenterMass) / (radius -
halfWheelCenterMass));
            //System.out.println("factor: "+ factor);

            // without the cast is teorical, with cast is pratical
            double velLeftWheelT = 2*MyRobotLego.VELOCITY / (factor+1);
            int velLeftWheelP = (int) velLeftWheelT;
            double velRightWheelT =  2*MyRobotLego.VELOCITY - velLeftWheelP;
            int velRightWheelP = (int) velRightWheelT;

            // must convert the factor to the pratical one
            factor = (double)velRightWheelP / (double)velLeftWheelP;

            //System.out.println("velLeftWheel: "+ velLeftWheelP);
            //System.out.println("velRightWheel: "+ velRightWheelT);
            //System.out.println("velRightWheel: "+ velRightWheelP);
            //System.out.println("factor: "+ factor);

            double radiusP = (halfWheelCenterMass) * ((factor + 1) / (factor -
1));
            //System.out.println(radiusP);

            double xc2 = Xf - radiusP * Math.sin(radOf);
            double yc2 = Yf + radiusP * Math.cos(radOf);
            //System.out.println("xc2: " + xc2);
            //System.out.println("yc2: " + yc2);

            int xc1 = 0;

            double d12 = Math.sqrt((Math.pow(xc1 - xc2, 2) + Math.pow(radiusP -
yc2, 2)));
            //System.out.println("d12: " + d12);

            double alpha1 =  Math.toDegrees(Math.acos(xc2/d12));
            //System.out.println("alpha1: "+ alpha1);
            double alpha2 = Of - alpha1;
            //System.out.println("alpha2: "+ alpha2);

            int[] results = { (int)alpha1, (int)radiusP, (int)d12,  (int)alpha2,
```

```
(int)radiusP};
            //System.out.println("results: "+Arrays.toString(results));
            return results;

      }

      public int[] trajectory2(int Xf, int Yf, int Of) {
            //System.out.println("Tragetória 2");
            double radOf = Math.toRadians(Of);



            double a  = 2 - (2 * Math.cos(radOf));
            double b = (2 * Yf * (1 + Math.cos(radOf)) ) - (2 * Xf *
Math.sin(radOf));
            double c = -( Math.pow(Xf, 2) + Math.pow(Yf, 2));



            double r = distanceBetweenPoints(a, b ,c);
            //System.out.println("R: "+r);

            double alpha = Math.asin((Xf + r * Math.sin(radOf)) / (2 * r));
            alpha = Math.toDegrees(alpha);
            //System.out.println("alpha: "+alpha);

            double factor = (r + (MyRobotLego.WHEEL_CENTER_MASS / 2)) / (r -
(MyRobotLego.WHEEL_CENTER_MASS / 2));
            //System.out.println("factor: "+factor);
            double vel2 = 2/ (factor + 1) * MyRobotLego.VELOCITY;
            //System.out.println("vel2: "+vel2);
            int tVel2 = (int) vel2;
            //System.out.println("vel2P: "+tVel2);
            int vel1 = 2 * MyRobotLego.VELOCITY - tVel2;
            //System.out.println("vel1: "+vel1);
            // makes the factor pratical
            factor = (double)vel1 / (double)tVel2;
            //System.out.println("factorP: "+factor);
            double radiusP = (MyRobotLego.WHEEL_CENTER_MASS / 2) * ((factor + 1)
/ (factor - 1));
```

```
            //System.out.println("radius P: "+radiusP);



            double[] c1 = {0, radiusP};
            double[] c2 = {radiusP * Math.sin(radOf) + Xf, Yf - (radiusP *
Math.cos(radOf))};
            this.yc1Tj2 = c1[1];
            this.yc2Tj2 = c2[1];
            //System.out.println("c2: " + Arrays.toString(c2));



            double d12 = Math.pow(Math.pow(c1[0] - c2[0], 2) + Math.pow(c1[1] -
c2[1], 2), 0.5);
            //System.out.println("d12: "+d12);



            double delta = Math.acos(radiusP/ (d12 /2 ));
            //double delta = Math.toDegrees(Math.acos(radiusP/ (d12 /2 )));
            //System.out.println("delta: "+delta);

            double dStraight = d12 * Math.sin(delta);
            //System.out.println("dStraight: "+ dStraight);

            double alphaP = Math.toDegrees(Math.asin(c2[0]/d12) - delta);
            //System.out.println("alphaP: "+ alphaP);

            double angleLastCurve = alphaP - Of;

            // results used to move the robot
            int[] results = { (int)alphaP, (int)radiusP, (int)dStraight,
(int)angleLastCurve, (int)radiusP};

            return results;



      }

      public int[] trajectory3(int Xf, int Yf, int Of) {
            //System.out.println("Tragetória 3");
```

```
            double radOf = Math.toRadians(Math.abs(Of));

            double a = 2 - 2 * Math.cos(radOf);
            double b = 2 * Yf * (1 + Math.cos(radOf)) + 2 * Xf *
Math.sin(radOf);
            double c = - (Math.pow(Xf, 2) + Math.pow(Yf, 2));

            double r = distanceBetweenPoints(a, b, c);
            r = (double) Math.round(r * 100) / 100;

            double delta = Math.asin((Xf - r * Math.sin(radOf)) / (r * 2)) ;

            // (double) Math.round(alpha * 100) / 100 rounds 2 decimals places
            double alpha = 180 - Math.toDegrees(delta);
            alpha = (double) Math.round(alpha * 100) / 100;


            double factor = ((r + (MyRobotLego.WHEEL_CENTER_MASS/2)) / (r -
(MyRobotLego.WHEEL_CENTER_MASS/2)));


            double vel2 = (2 / (factor + 1)) * MyRobotLego.VELOCITY;

            int vel2P = (int) vel2;
            int vel1P = 2 * MyRobotLego.VELOCITY - vel2P;


            double factorP = (double)vel1P / (double)vel2P;

            double rPratical = (MyRobotLego.WHEEL_CENTER_MASS/2) * ((factorP +
1) /(factorP - 1));

            double[] c1 = {0, rPratical};
            double[] c2 = {Xf - (Math.sin(radOf) * rPratical), Yf -
(Math.cos(radOf) * rPratical)};
            this.yc1Tj3 = c1[1];
            this.yc2Tj3 = c2[1];

            double d12 = Math.pow(Math.pow(c1[0] - c2[0], 2) + Math.pow(c1[1] -
c2[1], 2), 0.5);
```

```
            double deltaP = Math.acos(rPratical / (d12/ 2));

            double beta = Math.asin(c2[0] / d12);

            double dStraight = d12 * Math.sin(deltaP);


            double alphaP = 180 - (Math.toDegrees(beta) +
Math.toDegrees(deltaP));


            int[] result = {(int)Math.round(alphaP), (int)Math.round(rPratical),
(int)Math.round(dStraight),
                        (int)Math.round((alphaP- Of)),
(int)Math.round(rPratical)};

            return result;
      }


      // only applies the solver formula when it's different of 0
      private double distanceBetweenPoints(double a, double b, double c) {
            // when the a is really low value
            if (a < 0.0009 && a > -0.0009) {
                  double radius = (c * (-1)) / b;
                  return radius;
            }
            else {
                  // makes the calc
                  double sqrt = Math.sqrt((Math.pow(b, 2) - (4 * a * c)));

                  // does the +-
                  double minusB = (b*(-1)) - sqrt;
                  double plusB = (b*(-1)) + sqrt;

                  // must check which one is positive
                  double radius = (minusB > 0) ? minusB / (2* a)  : plusB / (2 *
```

```
a);
                  return radius;
            }
      }

      public double getYc1Tj2() {
            return this.yc1Tj2;
      }

      public double getYc2Tj2() {
            return this.yc2Tj2;
      }

      public double getYc1Tj3() {
            return this.yc1Tj3;
      }

      public double getYc2Tj3() {
            return this.yc2Tj3;
      }

      public double getAlphaTj1() {
            return this.alphaTj1;
      }
}
```

```
public class TrajectoriesGestor {

	private final int NO_TRAJECTORY = 0;
	private final int TRAJECTORY_1 = 1;
	private final int TRAJECTORY_2 = 2;
	private final int TRAJECTORY_3 = 3;

	private Trajectories trajectories;
	private MyRobotLego myRobot;

	// variables used to execute the trajectory
	private int[] commandValues;
	private boolean leftSide;
	private int trajectory;


	public TrajectoriesGestor(MyRobotLego myRobot) {
		this.myRobot = myRobot;
		trajectories = new Trajectories();
	}

	public void executeTrajectory(boolean waitCoomand) {
		switch(this.trajectory) {
			case TRAJECTORY_1:
				executeCommands1(commandValues, leftSide, waitCoomand);
				break;
			case TRAJECTORY_2:
				executeCommands2(commandValues, leftSide, waitCoomand);
				break;
			case TRAJECTORY_3:
				executeCommands3(commandValues, leftSide, waitCoomand);
				break;
		}

		// this way when collide don't do a previous calculated trajectory
		this.trajectory = NO_TRAJECTORY;
	}
```

```
	// returns as string the trajectory chosen
	public String getTrajectory(int Xf, int Yf, int Of) {

		String result = "";
		double dxif = calculateDxif(Xf, Of);

		// variable to mirror the trajectory
		boolean leftSide = (Yf > 0) ? true : false;
		Yf = Math.abs(Yf);
		this.leftSide = leftSide;

		if(dxif <= Xf) {
			int[] results = trajectories.trajectory1(Xf, Yf, Of);

			result = "Trajetória 1 -> Xf: "+Xf+", Yf: "+Yf+", Of: 
"+Of+"\n";
			String curveType = (leftSide) ? "curvarEsquerda(" : 
"curvarDireita(";

			result += curveType+results[0]+", "+results[1]+"); " +
                                "reta("+results[2]+");
"+curveType+results[3]+", "+results[4]+")";

			this.commandValues = results;
			this.trajectory = TRAJECTORY_1;
		}

		else {

			int[] resultsTj2 = trajectories.trajectory2(Xf, Yf, Of);
			int[] resultsTj3 = trajectories.trajectory3(Xf, Yf, Of);

			double yc1Tj2 = trajectories.getYc1Tj2();
			double yc2Tj2 = trajectories.getYc2Tj2();

			double yc1Tj3 = trajectories.getYc1Tj3();
			double yc2Tj3 = trajectories.getYc2Tj3();
```

```
                if(yc2Tj2 < yc1Tj2) {

                        /*System.out.println("ResultsTj2: " +
Arrays.toString(resultsTj2));
                        System.out.println("yc1Tj2: "+yc1Tj2);
                        System.out.println("yc2Tj2: "+yc2Tj2);*/

                        String curveType1 = (leftSide) ? "curvarEsquerda(" :
"curvarDireita(";
                        String curveType2 = (leftSide) ?  "curvarDireita(" :
"curvarEsquerda(";

                        result = "Trajetória 2 -> Xf: "+Xf+", Yf: "+Yf+", Of:
"+Of+"\n";
                        result += curveType1+resultsTj2[0]+", "+resultsTj2[1]+");
" +
                                        "reta("+resultsTj2[2]+");
"+curveType2+resultsTj2[3]+", "+resultsTj2[4]+")";

                        this.commandValues = resultsTj2;
                        this.trajectory = TRAJECTORY_2;
                }

                else if(yc2Tj3 >= yc1Tj3) {

                        /*System.out.println("resultsTj3: " +
Arrays.toString(resultsTj3));
                        System.out.println("yc1Tj3: "+yc1Tj3);
                        System.out.println("yc2Tj3: "+yc2Tj3);*/

                        String curveType1 = (leftSide) ? "curvarEsquerda(" :
"curvarDireita(";
                        String curveType2 = (leftSide) ?  "curvarDireita(" :
"curvarEsquerda(";

                        result = "Trajetória 3 -> Xf: "+Xf+", Yf: "+Yf+", Of:
"+Of+"\n";
                        result += curveType1+resultsTj3[0]+", "+resultsTj3[1]+");
" +
                                        "reta("+resultsTj3[2]+");
"+curveType2+resultsTj3[3]+", "+resultsTj3[4]+")";
```

```
                        this.commandValues = resultsTj3;
                        this.trajectory = TRAJECTORY_3;
                  }

            }
            System.out.println(result);
            return result;

      }

      private double calculateDxif(int Xf, int Of) {
            double radOf = Math.toRadians(Math.abs(Of));

            // sen x = C opposite / hipotenuse
            double hipotenuse = Xf / Math.sin(radOf);

            // cos x = C addjacent / hipotenuse
            double dxif = Math.cos(radOf) * hipotenuse;

            return dxif;
      }

      private void executeCommands1(int[] results, boolean leftSide, boolean
waitCommand) {
            boolean collided = false;

            if(leftSide)
                  collided = curveLeftCollision(results[0], results[1],
waitCommand);

            // we add 10 to the angle to adjust because is does less than the
wanted angle
            else
                  collided = curveRightCollision(results[0], results[1],
waitCommand);



            if(collided) return;
```

```
            // goes with straight line, if collided must end
            collided = straightLineCollision(results[2], waitCommand);
            if(collided) return;



            if(leftSide)
                  curveLeftCollision(results[3], results[4], waitCommand);

            else
                  curveRightCollision(results[3], results[4], waitCommand);

      }


      private void executeCommands2(int[] results, boolean leftSide, boolean
waitCommand) {
            boolean collided = false;

            // when is not mirrored trajectory
            if (leftSide)
                  collided = curveLeftCollision(results[0], results[1],
waitCommand);

            // when is mirrored trajectory
            else
                  collided = curveRightCollision(results[0], results[1],
waitCommand);



            if(collided) return;

            // goes with straight line, if collided must end
            collided = straightLineCollision(results[2], waitCommand);
            if(collided) return;



            if (leftSide)
                  curveRightCollision(results[3], results[4], waitCommand);
```

```
            else
                  curveLeftCollision(results[3], results[4], waitCommand);

      }

      private void executeCommands3(int[] results, boolean leftSide, boolean
waitCommand) {
            boolean collided = false;

            // when is not mirrored trajectory
            if (leftSide)
                  collided = curveLeftCollision(results[0], results[1],
waitCommand);

            // when is mirrored trajectory
            else
                  collided = curveRightCollision(results[0], results[1],
waitCommand);



            if(collided) return;

            // goes with straight line, if collided must end
            collided = straightLineCollision(results[2], waitCommand);
            if(collided) return;

            // when is not mirrored trajectory
            if(leftSide)
                  curveRightCollision(results[3], results[4], waitCommand);

            // when is mirrored trajectory
            else
                  curveLeftCollision(results[3], results[4], waitCommand);

      }

      public void collidedGoBack() {
            // when collides goes back 70cm and curves 90º
```

```
            this.myRobot.line(-70, false);
            this.myRobot.stop();

            this.myRobot.curveLeft(90, 20, false);
            this.myRobot.stop();
      }

      private void waitForCommand(boolean waitCommand) {
            if (waitCommand) {
                  try {
                        Thread.sleep(2000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }

      // method to detect collision on forward and go back 70 cms and curve 90º
      private boolean straightLineCollision(int distance, boolean waitCommand) {
            boolean collided = this.myRobot.line(distance, true);
            if (collided) {
                  collidedGoBack();
                  return true;
            }

            if (waitCommand) {
                  this.myRobot.stop();
                  waitForCommand(waitCommand);
            }
            return false;
      }

      private boolean curveLeftCollision(int angle, int radius, boolean
waitCommand) {
            boolean collided = this.myRobot.curveLeft(angle, radius, true);
            if (collided) {
                  collidedGoBack();
                  return true;
            }
```

```
            if (waitCommand) {
                  this.myRobot.stop();
                  waitForCommand(waitCommand);
            }
            return false;
      }

      private boolean curveRightCollision(int angle, int radius, boolean
waitCommand) {
            boolean collided = this.myRobot.curveRight(angle, radius, true);
            if (collided) {
                  collidedGoBack();
                  return true;
            }
            if (waitCommand) {
                  this.myRobot.stop();
                  waitForCommand(waitCommand);
            }
            return false;
      }

}
```

```
public class FollowWall {

	private final int MIN_ANGLE = 5;
	private final int DISTANCE_LINE = 20;
	private final int IDEAL_DISTANCE = 50;
	private MyRobotLego myRobot;
	private TrajectoriesGestor gestor;

	public FollowWall(MyRobotLego myRobot, TrajectoriesGestor gestor) {
		this.myRobot = myRobot;
		this.gestor = gestor;
	}

	// return the values chosen on trajectory
	public String calcutate() {

		double[] dt1dt2Alpha = this.getDistancesAndAlpha();
		if(dt1dt2Alpha == null) {
			this.gestor.collidedGoBack();
			return "collided!";
		}

		// 1. Method
		// straighten the robot then go to the line
		//straightenRobot((int)dt1dt2Alpha[0], (int)dt1dt2Alpha[1],
dt1dt2Alpha[2]);
		//String result = alternativeLine();

		// 2. Method
		// straighten the robot then get the equation of the ideal line so
that we get a point
		//straightenRobot((int)dt1dt2Alpha[0], (int)dt1dt2Alpha[1],
dt1dt2Alpha[2]);
		//int[] x3y3 = calcIdealLine((int) dt1dt2Alpha[0], (int)
dt1dt2Alpha[1],  Math.toRadians(dt1dt2Alpha[2]));
		//String result = gestor.getTrajectory(x3y3[0], x3y3[1], 0);

		// 3. Method
		// just get the line equation then get the point
```

```
            int[] x3y3 = calcIdealLine((int) dt1dt2Alpha[0], (int)
dt1dt2Alpha[1],  Math.toRadians(dt1dt2Alpha[2]));
            String result = gestor.getTrajectory(x3y3[0], x3y3[1],
(int)dt1dt2Alpha[2]);



            return result;

      }



      public int[] calcIdealLine (int dist0, int dist1, double alphaRad) {
            // resultados mal
            double dp1 = dist1 * Math.cos(alphaRad);
            System.out.println("dp1: "+dp1);

            double dort = Math.abs(dp1 - IDEAL_DISTANCE);
            System.out.println("dort: "+dort);

            double h = dort / Math.cos(alphaRad);
            System.out.println("h: "+h);

            double x2 = dort * Math.sin(alphaRad);
            double y2 = dort * Math.cos(alphaRad);
            double x1 = 0;
            double y1 = h;
            System.out.println("x2: "+x2);
            System.out.println("y2: "+y2);
            System.out.println("x1: "+x1);
            System.out.println("y1: "+y1);

            // this formula is x = mx + b;
            double m = (y1 - y2) / (x2 - x1);

            // if the robot is between the wall and the ideal line b = h
            // if the robot is beyond the ideal line b = -h
            double b;

            if(dp1 > IDEAL_DISTANCE) {
```

```
                b = -h;
            }
            else {
                b = h;
            }


            System.out.println("m: "+m);
            System.out.println("b: "+b);

            //double x3 = (100);
            double x3 = (x2+40);
            double y3 = (m*x3 + b);
            System.out.println("x3: "+x3);
            System.out.println("y3: "+y3);

            int[] result = {(int)x3, (int)y3};

            return result;

      }

      /*private String alternativeLine() {

            int dist = this.myRobot.getSensorUs();
            int distToLine = dist - IDEAL_DISTANCE;
            String result = gestor.getTrajectory(50,distToLine,0);

            return result;
      }*/

      public void straightenRobot(int dist0, int dist1, double alpha) {
            int curve = (dist1 > dist0) ? 1 : 0;

            // it only curves when the radius obtained is bigger than 5
otherwise gets a big error
            if(alpha > MIN_ANGLE) {

                  // when the goes away from the wall curves to the other-side
```

```
                switch(curve) {
                case 0:
                      // the radius of the curve is 7 because this the one with
lowest error
                      myRobot.curveLeft((int) alpha, 10, true);
                      myRobot.stop();
                      //System.out.println("alpha: "+alpha);
                      break;
                case 1:
                      myRobot.curveRight((int) alpha, 10, true);
                      myRobot.stop();
                      break;
                }
          }
     }

     private double[] getDistancesAndAlpha() {
          int dist0 = this.myRobot.getSensorUs();
          System.out.println("dist0: "+dist0);

          // if collided must stop
          boolean collided = myRobot.line(DISTANCE_LINE, true);

          if(collided) return null;

          int dist1 = this.myRobot.getSensorUs();
          System.out.println("dist1: "+dist1);



          double xs = Math.abs(dist0 - dist1);
          double xs2 = xs/ DISTANCE_LINE;

          double a =  Math.atan(xs2);
          //System.out.println("aRad: "+a);

          // round the number adding 0.5
          double aDegrees = Math.toDegrees(a)+0.5;
          System.out.println("aDegrees: "+aDegrees);
```

```
            double[] result = {(double) dist0, (double) dist1, aDegrees};

            return result;
      }

}
```