

Sistemas Autónomos Inteligentes

1º Ano

Mestrado em Engenharia Eletrónica Industrial e Computadores

Universidade do Minho

2023/2024



Grupo 5

António Serra PG53665

Marcelo Ribeiro PG54028

Paulo Castro PG54130

Pedro Teixeira PG54135

Pedro Loureiro PG54143

Prof. Estela Bicho

Prof. Luís Louro

Prof. Sérgio Monteiro

Índice

Conteúdo

1. Introdução	1
1.1. Objetivos do trabalho	1
1.2. Estrutura do braço	2
2. Simulação em CoppeliaSim	3
2.1. Criação de um cenário	3
2.2. Controlo do braço	4
2.3. Visão por Computador	5
2.3.1 Cor	5
2.3.2 Forma	6
2.3.3 Posição	7
2.4. Redes neuronais	8
2.5. Campos dinâmicos	10
2.6. Resultados	16
3. Implementação no Real	17
3.1 Comunicação com o braço	17
3.2 Visão por computador no teste real	19
3.3 Redes neuronais	21
3.4 Resultados	21
4. Conclusão	27
5. Referências	28

1. Introdução

1.1. Objetivos do trabalho

No âmbito da Unidade Curricular de Sistemas Autónomos Inteligentes foi proposto pela equipa docente desenvolver um sistema autónomo inteligente de armazenamento e organização de objetos, com o intuito de aprofundar os conhecimentos de visão por computador, redes neurais, campos neuronais dinâmicos, ROS e ambiente de simulação em CoppeliaSim.

O robô utilizado para o projeto é o “PincherX 100 RobotArm” cuja tarefa é colocar um determinado objeto numa posição predefinida que é determinada pelas características deste. Para este caso de estudo foram utilizados três objetos: Um cubo vermelho, um cubo verde e um cilindro verde. Deste modo existem duas características visuais diferentes que servem para distinguir onde colocar o objeto. Para este projeto, a visão por computador é crucial para determinar a posição de cada objeto e consequentemente possibilitando que, através do treino de uma rede neuronal, cada junta do braço tome valores que permitem o manuseamento destes.

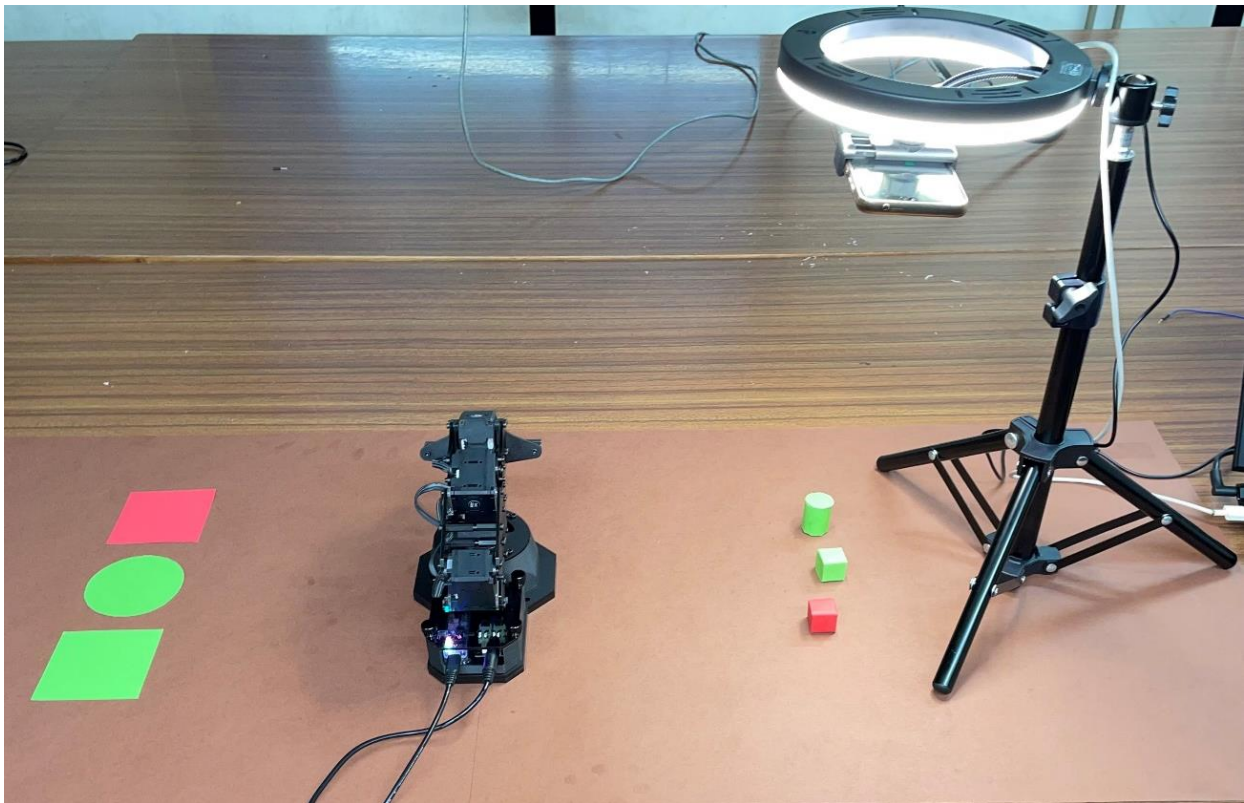


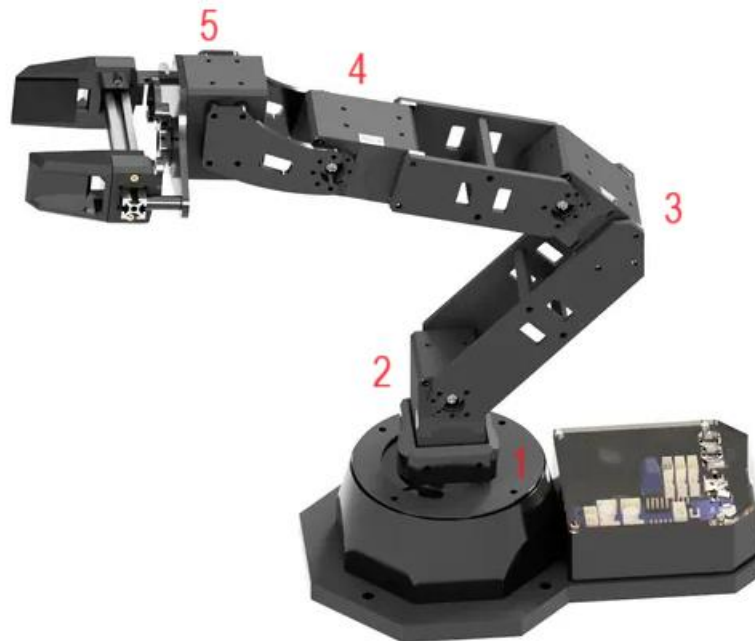
Figura 1-Projeto desenvolvido.

1.2. Estrutura do braço

O braço robótico (PincherX 100 Robot Arm) possui as seguintes características:

Degrees of Freedom	4
Reach	300mm
Span	600mm
Repeatability	5mm
Working Payload	50g
Weight	770g

Este é composto por cinco servo motores, dos quais podem ser controlados com limitações relativamente aos valores dos ângulos atribuídos a cada uma das juntas.



	Nome	Range angular
1	Waist	-179.99°↔180.00°
2	Shoulder	-111.00°↔107.00°
3	Elbow	-121.00°↔92.00°
4	Wrist	-100.00°↔123.00°
5	Gripper	Close/Open

2. Simulação em CoppeliaSim

2.1. Criação de um cenário

Antes de qualquer implementação utilizando o braço real, foi desenvolvido em CoppeliaSim um cenário o mais semelhante possível ao ambiente de testes que foi depois implementado na realidade. Deste modo criou-se um *digital twin* em ambiente de simulação onde foram testadas e validadas todas as componentes individuais deste projeto. Os elementos principais desta simulação são:

Braço robótico “px100” – Responsável pelo manuseamento dos objetos. Os valores de ângulos para as suas juntas serão determinados por uma rede neuronal;

Sensor de visão – Captação de imagens que são processadas para obter informação acerca do ambiente;

Cubos e Cilindro – Objetos que devem ser colocados em posições diferentes dependendo da sua forma e cor. Esta informação é obtida através do processamento das imagens do sensor de visão;

Bases – Posições predefinidas onde os objetos devem ser colocados;

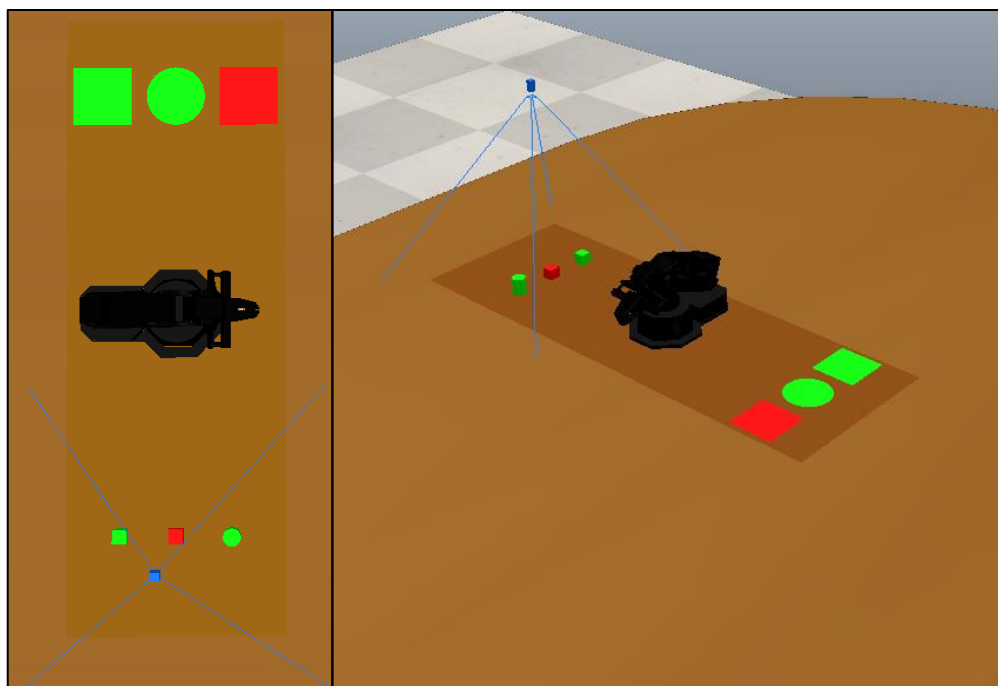


Figura 2- Cenário desenvolvido em CoppeliaSim

Tal como mostra a figura 2, o braço foi posicionado em cima de uma mesa onde também se encontram os objetos e as bases onde estes serão colocados. As posições das bases são fixas significando que não necessitam de ser determinadas pelo sistema de visão. A posição dos objetos pode mudar e, portanto, o sensor de visão deve cobrir toda a área onde é possível que os objetos se encontrem.

2.2. Controlo do braço

O controlo de todo o sistema foi feito em MATLAB. Assim sendo, através de uma comunicação entre este programa e o CoppeliaSim, foi possível desenvolver todo o código responsável pela execução das tarefas que cumprem o objetivo inicialmente proposto.

Fazendo uso das funções de API disponibilizadas pelo Coppelia Robotics para MATLAB, foi possível controlar as juntas do braço. Em primeiro lugar, foi necessário obter os handles para cada motor. Isto foi implementado numa função de inicialização, chamada no início do programa. Após este passo, foram desenvolvidas funções que permitem atribuir, individualmente, ângulos a cada junta do robô e também ler qual o ângulo atual de uma determinada junta. É importante mencionar que, em simulação, todos os motores são controlados por posição e não por PWM.

```
function [error] = setJointPosition(obj,position,MotorNumber)

    res = obj.vrep.simxSetJointTargetPosition(obj.clientID, obj.MotorHandle{MotorNumber},deg2rad(position), obj.vrep.simx_opmode_streaming);
    if (res ~= obj.vrep.simx_return_ok && res ~= obj.vrep.simx_return_novalue_flag)
        disp('ERROR: Failed sending angle value!');
        error = 1;
        return;
    end
end
function [error,JointAngle] = getJointAngle(obj,MotorNumber)

    [res,JointAnglerad] = obj.vrep.simxGetJointPosition(obj.clientID, obj.MotorHandle{MotorNumber}, obj.vrep.simx_opmode_streaming);
    JointAngle=rad2deg(JointAnglerad);
    if (res ~= obj.vrep.simx_return_ok && res ~= obj.vrep.simx_return_novalue_flag)
        disp('ERROR: Failed receiving angle data!');
        error = 1;
        return;
    end
end
```

Figura 3 - Funções de escrita e leitura nas juntas

Tal como demonstrado na figura 3, as funções recebem como parâmetro qual o valor da junta, de 1 a 4, para a qual é pretendido atribuir ou ler um valor de ângulo. Os números correspondem a “waist”, “shoulder”, “elbow” e “wrist_angle” respetivamente. Para a função “setJointPosition”, deve ainda ser indicado qual o ângulo pretendido. No caso da função “getJointAngle”, esta retorna o valor, convertido para graus, do ângulo que a junta mencionada possui naquele momento.

```
function [error] = close_gripper(obj)

    res = obj.vrep.simxSetJointTargetPosition(obj.clientID,obj.MotorHandle{5},0.015, obj.vrep.simx_opmode_streaming);
    if (res ~= obj.vrep.simx_return_ok && res ~= obj.vrep.simx_return_novalue_flag)
        disp('ERROR: Failed sending close command to gripper(left finger)!');
        error = 1;
        return;
    end
    res = obj.vrep.simxSetJointTargetPosition(obj.clientID,obj.MotorHandle{6},-0.015, obj.vrep.simx_opmode_streaming);
    if (res ~= obj.vrep.simx_return_ok && res ~= obj.vrep.simx_return_novalue_flag)
        disp('ERROR: Failed sending close command to gripper(right finger)!');
        error = 1;
        return;
    end
end

function [error] = open_gripper(obj)

    res = obj.vrep.simxSetJointTargetPosition(obj.clientID,obj.MotorHandle{5},0.047, obj.vrep.simx_opmode_streaming);
    if (res ~= obj.vrep.simx_return_ok && res ~= obj.vrep.simx_return_novalue_flag)
        disp('ERROR: Failed sending open command to gripper(left finger)!');
        error = 1;
        return;
    end
    res = obj.vrep.simxSetJointTargetPosition(obj.clientID,obj.MotorHandle{6},-0.047, obj.vrep.simx_opmode_streaming);
    if (res ~= obj.vrep.simx_return_ok && res ~= obj.vrep.simx_return_novalue_flag)
        disp('ERROR: Failed sending open command to gripper(right finger)!');
        error = 1;
        return;
    end
end
```

Figura 4 - Funções de fecho e abertura da gripper

Por fim, para que o robô possa mudar a posição dos objetos para a correspondente às suas características, a “*gripper*” deve abrir e fechar permitindo largar e pegar nos objetos. No caso da simulação, estas ações são controladas por posição, o que significa que para abrir a “*gripper*” deve ser-lhe atribuída o range máximo de posição linear para os “*fingers*” (0.047 e -0.047) e para fechar, deve ser-lhes atribuído a posição mínima (0.015 e -0.015). Ambas as funções estão demonstradas na figura 4.

2.3. Visão por Computador

Para obter informação acerca dos objetos (posição, cor e forma), para que o braço possa alcançá-los e colocá-los no sítio certo, foi utilizado um *vision sensor* que capta imagens do ambiente de simulação que serão processadas, extraindo todos os dados necessários. A função do remote API utilizada para obter a imagem do ambiente de simulação foi a “*simxGetVisionSensorImage2*” que retorna, para além da resolução, uma imagem que é atribuída à variável “img”. Esta é uma matriz de 256x256 com três dimensões onde cada dimensão corresponde a uma cor do código RGB.

2.3.1 Cor

De modo a saber qual a cor de um determinado objeto, utilizou-se a função “*imsplit*”, disponível na *toolbox* de visão do MATLAB (*Image Processing Toolbox*). Esta função retorna 3 imagens diferentes em que cada uma corresponde a uma escala de cinzentos determinada por uma das cores RGB. Assim sendo, obtém uma imagem para o vermelho, outra para o verde e outra para o azul. Para este caso, apenas interessam as imagens do vermelho e do verde. Depois utilizou-se a função “*im2bw*” que devolve uma outra imagem com apenas duas cores, preto e branco. O parâmetro *Level*, que é

fornecido como parâmetro para esta função, determina qual o threshold para que um certo tom de cinzento seja considerado branco ou preto, portanto este deve ser afinado de modo que apenas os objetos da cor pretendida fiquem brancos e tudo o resto fique preto. O resultado deste algoritmo está exemplificado na figura 5.

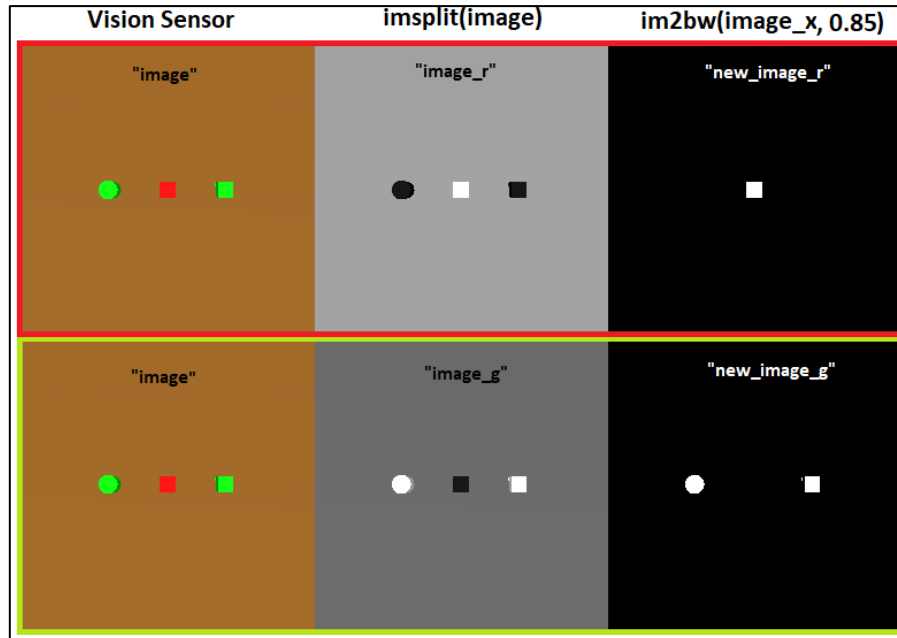


Figura 5 - Separação da imagem por cores

2.3.2 Forma

A outra característica que distingue os objetos é a forma, que pode ser cubo ou cilindro. Esta deve, tal como a cor, ser identificada através do sistema de visão. Para isso, e fazendo uso das imagens obtidas para a distinção das cores, utilizou-se uma outra função disponível na *toolbox* de visão. Esta função é a “*edge*” em que são dados como parâmetro a imagem obtida anteriormente e também o método que será utilizado para a criação de uma linha nos limites dos objetos. Depois de criar este limite, é chamada a função “*regionprops*” que recebe como parâmetro a imagem criada pela função anterior e retorna várias informações acerca de diferentes propriedades da forma criada pelos limites dos objetos. Para identificar qual a forma, comparam-se os valores da área da figura criada pelos limites do objeto com a propriedade chamada “*Convex Area*” que corresponde à área de um quadrado que engloba toda a figura.

Tal como mostra a figura 6, no caso dos cubos, a área da figura originada pelos seus limites (figuras do centro) é igual à área do quadrado criado à volta destes (quadrado azul nas figuras da direita). Quanto ao cilindro, a área da figura criada pelos seus limites será sempre menor do que a área do quadrado criado à sua volta. Assim sendo comparando estes valores, é possível distinguir qual a forma do objeto.

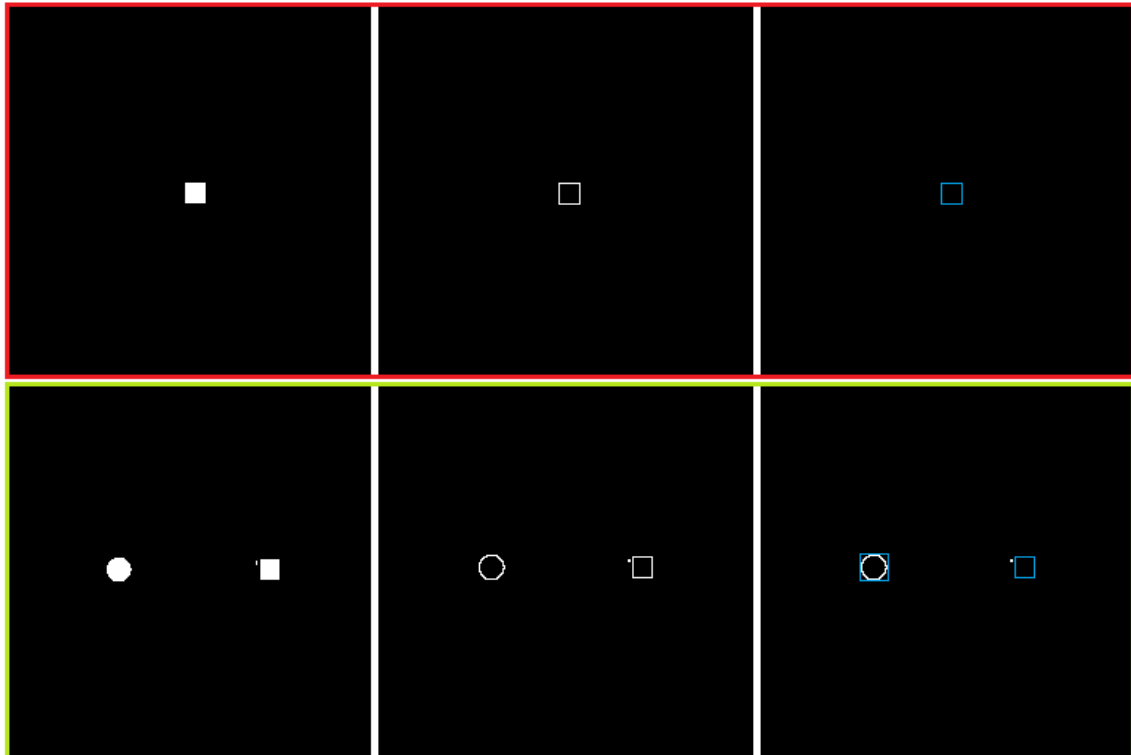


Figura 6 - Delimitação dos objetos identificados por cor

2.3.3 Posição

Depois de identificada a cor e a forma do objeto, é importante saber onde é que este se encontra no espaço para que a sua posição seja convertida em ângulos para as juntas do braço robótico. Assim sendo, foi utilizado um método simples que transforma as coordenadas, dos centros das formas, na imagem fornecida pelo sensor em coordenadas do ambiente de simulação. Para isso foram colocados vários objetos em cima da mesa, dentro do raio de visão do sensor, e foi feita uma relação para ambas as coordenadas x e y, entre as coordenadas na imagem e as coordenadas na simulação. A figura 7 mostra as retas de regressão para este efeito.

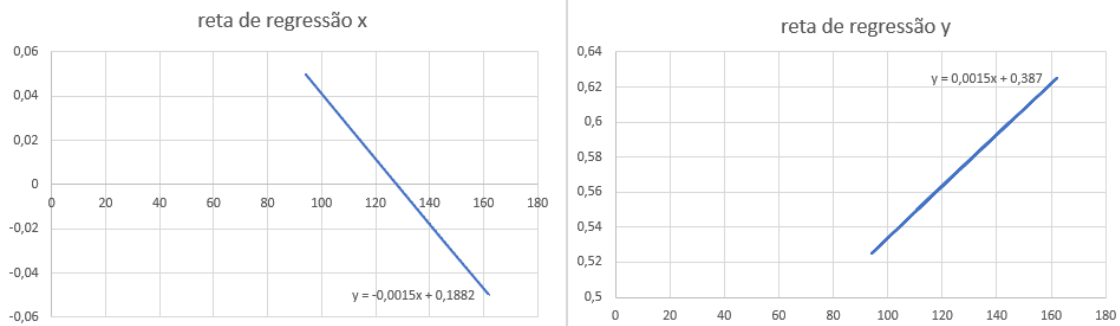


Figura 7 - Equações que relacionam as coordenadas da imagem com as coordenadas do CoppeliaSim

2.4. Redes neuronais

Para determinar a posição no espaço para um conjunto de ângulos das juntas, e para determinar os ângulos das juntas para uma certa posição, temos de implementar duas redes neuronais sendo classificadas como rede neuronal da cinemática direta e rede neuronal da cinemática inversa respetivamente, para isso foi utilizada a “*Deep Learning Toolbox*” do MATLAB.

Para treinar uma rede neuronal temos de primeiro criar as matrizes de dados de entrada e de saída, neste caso as matrizes “angles” e “posição” vão servir ambas como matriz de entrada ou de saída dependendo de qual rede estamos a treinar. Estas matrizes foram criadas através de um ciclo onde, a cada iteração é colocado um novo conjunto de ângulos de interesse nas juntas e são extraídas as coordenadas do centro da “*gripper*” no espaço. Para que estas sejam as mais corretas possíveis, foi utilizado um “*Dummy*” colocado entre os “*fingers*” (fig.8 e fig.9).

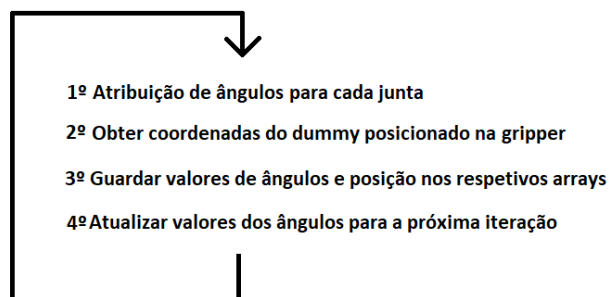


Figura 8 - Algoritmo utilizado para a obtenção de dados de treino

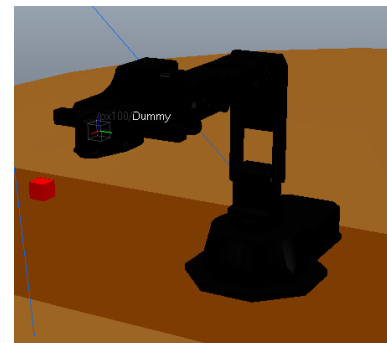


Figura 9 - Dummy utilizado para obter as coordenadas do centro da gripper

O segundo passo é selecionar o método de treino e número de camadas ocultas. Para a escolha do método são disponibilizadas três opções “*Bayesian Regularization Training*”, “*Levenberg-Marquardt Training*” e “*Scaled Conjugate Gradient Training*”.

O método de treino “*Levenberg-Marquardt*” é rápido para problemas bem definidos, o método “*Scaled Conjugate Gradient*” é eficaz numa grande variedade de problemas incluindo problemas não lineares, porém o método escolhido foi o “*Bayesian Regularization*” que, apesar de ser lento, foi o que apresentou resultados mais satisfatórios no contexto do nosso problema.

A quantidade de “*hidden layers*” escolhida foi de 30 unidades. Este valor foi selecionado após testes com várias quantidades sendo esta a mais adequada pois apresenta um treino rápido e eficiente. Por último temos de selecionar a percentagem dos dados fornecidos para treinar, validar e testar a rede. Estes parâmetros foram ajustados até obter um resultado satisfatório.

Para criar e treinar estas redes foi desenvolvido o seguinte código (fig.10 e fig.11)

```
x = angles'; % Dados de entrada
t = posicao'; % Dados de saída

trainFcn = 'trainbr'; % Método de treino escolhido (treino Bayesiano Regularizado)
hiddenLayerSize = 30; % Número de hidden layers
net2 = fitnet(hiddenLayerSize, trainFcn); % Criação da rede neuronal

% Escolha da percentagem dos dados fornecidos usados para treinar, validar
% e testar a rede criada

net2.divideParam.trainRatio = 70/100; % 70% dos dados para treino
net2.divideParam.valRatio = 15/100; % 15% dos dados para validação
net2.divideParam.testRatio = 15/100; % 15% dos dados para teste

[net2,tr] = train(net2,x,t); % Treino da rede

% Teste da rede

y = net2(x); % Realiza previsões usando a rede treinada nos dados de treinamento
e = gsubtract(t, y); % Calcula o erro entre as saídas reais e as previsões
performance = perform(net2, t, y); % Calcula a performance da rede
```

Figura 10 - Código implementado para a criação da rede neuronal direta

```
x = posicao'; % Dados de entrada
t = angles'; % Dados de saída

trainFcn = 'trainbr'; % Método de treino escolhido (treino Bayesiano Regularizado)
hiddenLayerSize = 30; % Número de hidden layers
net2 = fitnet(hiddenLayerSize, trainFcn); % Criação da rede neuronal

% Escolha da percentagem dos dados fornecidos usados para treinar, validar
% e testar a rede criada

net2.divideParam.trainRatio = 70/100; % 70% dos dados para treino
net2.divideParam.valRatio = 15/100; % 15% dos dados para validação
net2.divideParam.testRatio = 15/100; % 15% dos dados para teste

[net2,tr] = train(net2,x,t); % Treino da rede

% Teste da rede

y = net2(x); % Realiza previsões usando a rede treinada nos dados de treinamento
e = gsubtract(t, y); % Calcula o erro entre as saídas reais e as previsões
performance = perform(net2, t, y); % Calcula a performance da rede
```

Figura 11 - Código implementado para a criação da rede neuronal inversa

Após criar e treinar a rede neuronal da cinemática inversa analisamos os gráficos referentes à performance, erro e regressão uma vez que esta é a rede que será usada (fig.12).

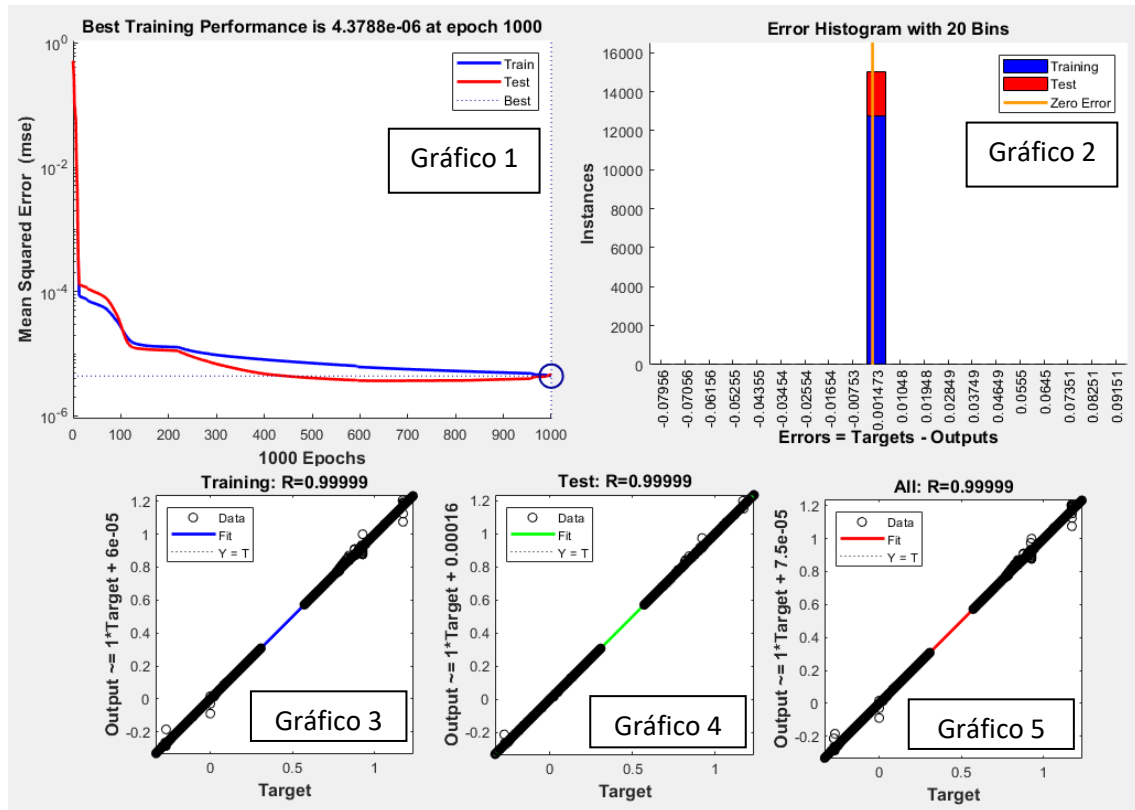


Figura 12 - Gráficos resultantes do treino da rede

O gráfico 1 mostra a diminuição do erro ao longo do processo de treino e de teste, como se pode observar o erro na fase de treino tende para 0 e na fase de teste essa tendência mantém-se. O gráfico 2 mostra a diferença entre o valor esperado e o valor obtido, estando este valor compreendido num pequeno intervalo em torno de 0. Os gráficos 3, 4 e 5 mostram as retas de regressão para a fase de treino, de teste e o processo completo, que representam o quão bem a rede se ajustou aos dados, como o valor R é muito próximo de 1 significa que a rede tem um bom desempenho.

2.5. Campos dinâmicos

Para cumprir o objetivo proposto, o sistema deve ser capaz de tomar uma decisão acerca de onde colocar os objetos, dependendo da informação adquirida pelo sistema de visão. Neste projeto, existem duas propriedades que distinguem os objetos: cor e forma. Dois dos objetos são cubos e apenas um deles é cilíndrico. Para além disso, dois são verdes e somente um é vermelho. Quanto aos sítios onde os objetos vão ser colocados, as bases, estes têm sempre o mesmo valor de posição no espaço. Deste modo, apenas é necessário atribuir um ângulo à “waist” capaz de rodar todo o braço na direção onde está cada uma das bases. Os ângulos determinados para tal efeito foram: 70 graus – Base Quadrada Vermelha; 90 graus – Base Circular Verde; 110 graus – Base Quadrada Verde (fig. 13).

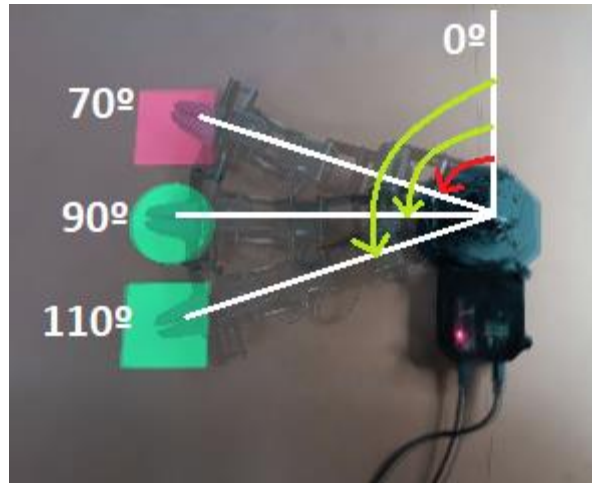


Figura 13 - Ângulos da "waist" para cada base

Analisando as possibilidades de ângulos da "waist" em função das propriedades, foi construída a seguinte tabela:

		Forma	
		Cubo	Cilindro
Cor	Vermelho	70	N/A
	Verde	110	90

Figura 14 - Correspondência entre ângulo e propriedades do objeto

Existem então duas propriedades que têm duas possibilidades para o ângulo da "waist". Estas são a cor verde e a forma cúbica. Deste modo, a verificação da propriedade complementar a estas é crucial para que o sistema tome a decisão final quanto ao ângulo a ser atribuído, uma vez que, tanto para o vermelho como para a forma cilíndrica, só existe um output possível.

No sistema desenvolvido, a decisão acerca de onde pousar um determinado objeto é feita por campos dinâmicos que recebem como input uma gaussiana centrada no valor que corresponde à informação obtida, pelo sistema de visão, para uma determinada característica. Uma vez que os campos têm eixos contínuos, foi necessário selecionar um intervalo de valores para cada uma das características dos objetos. Para a cor foi utilizado o espectro de luz visível. Assim sendo, o eixo x representa o comprimento de onda correspondente à cor (fig. 15). O range de valores para o vermelho é 625-740 e para o verde é 500 a 565. Uma vez que não existe nenhum intervalo contínuo de valores que represente forma, para esta característica decidiu-se utilizar um intervalo de 0 a 500 em que o valor 100 representa o cilindro e 300 representa o cubo (fig. 16).

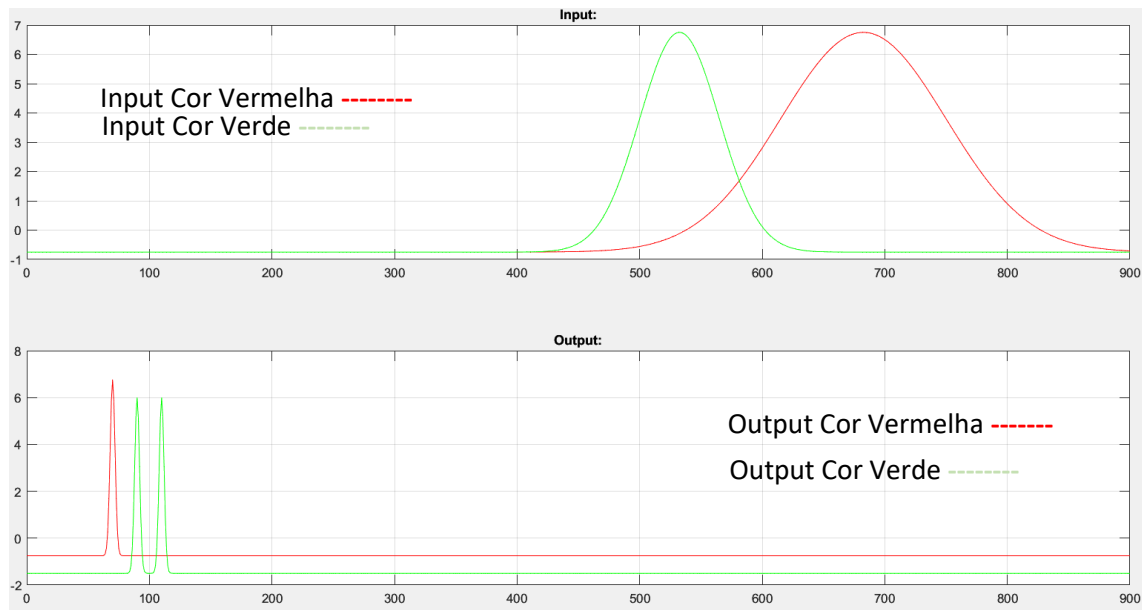


Figura 15 – Propriedade: Cor; Gaussianas de input de comprimento de onda (gráfico de cima) e respetivos outputs para ângulo da “waist” (gráfico de baixo).

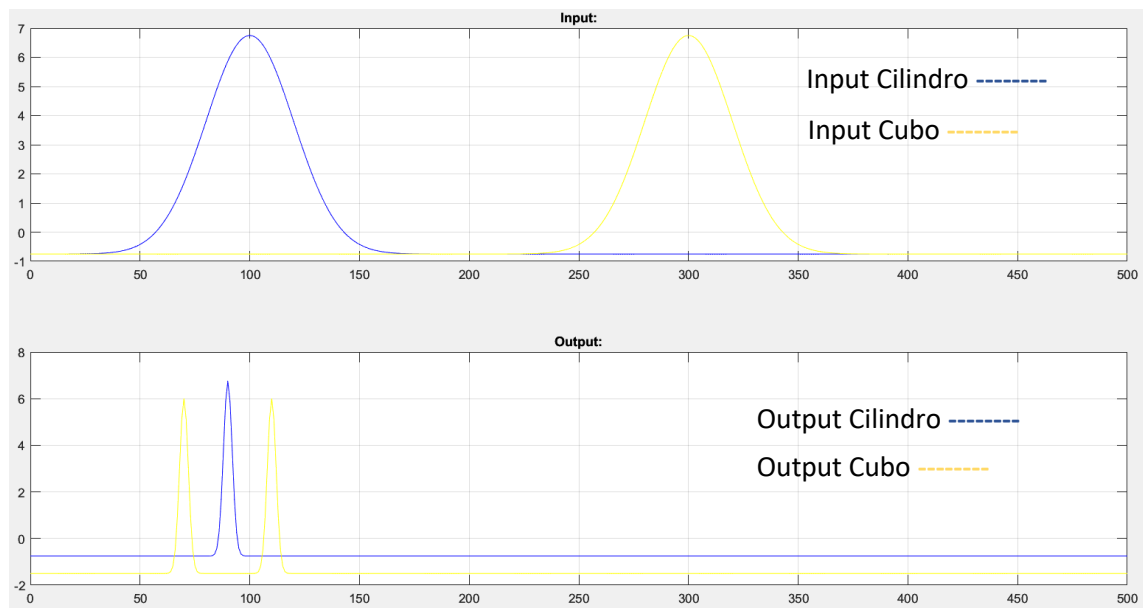


Figura 16-Propriedade: Forma; Gaussianas de input da forma (gráfico de cima) e respetivos outputs para ângulo da “waist” (gráfico de baixo).

O output dos campos dinâmicos é obtido através da equação de Amari. Esta equação permite tomar uma decisão acerca de qual o output correto para um ou mais inputs.

$$\tau \frac{\partial u(\psi, t)}{\partial t} = -u(\psi, t) + h + S(\psi, t) + \int w(\psi - \psi') g(u(\psi', t)) d\psi' + noise$$

Analisemos o caso do cubo vermelho. Serão utilizadas duas equações Amari para obter os inputs e tal como já foi mencionado, foram criadas gaussianas para cada possibilidade de input. Neste caso o input para a cor será o vermelho e o input para a forma será o cubo. O output de cada um destes campos será uma nova gaussiana com valores positivos centrados nos valores correspondentes à característica do objeto e os restantes valores tomam valores negativos. Assim sendo este output deve ser filtrado mantendo os valores positivos e tornando os valores negativos iguais a zero. Após esta filtragem, a nova gaussiana é multiplicada pela matriz dos pesos de ligação entre os neurónios de entrada e saída fazendo um somatório para obter o ângulo (ou ângulos) correspondente à característica (fig. 17).

```
for i=1:nx_cor_saida
    for j=1:nx_cor
        S_cor_p(i)=S_cor_p(i)+(y_cor(j)*W_cor(i,j));
    end
end
```

Figura 17 - Algoritmo para obter a o ângulo correspondente a uma característica. Somatório da multiplicação entre input e peso das ligações.

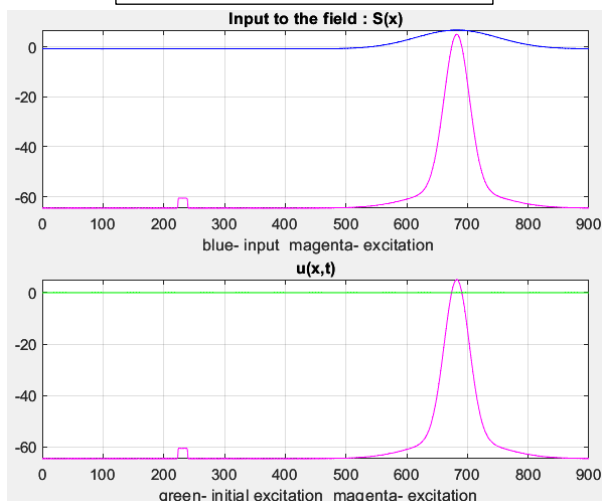
Para obter os pesos das ligações entre os neurónios de input e output, utilizou-se a regra de aprendizagem Hebbian. Segundo esta regra, neurónios que disparam seguidamente estão conectados de alguma forma sendo esta uma regra de aprendizagem por associação. Na fase de treino é necessário indicar o input, mas também o respetivo output. Por esta razão, foram também criadas as gaussianas correspondentes aos ângulos tal como mostra a figura anterior. A figura 18 mostra o algoritmo utilizado para obter a matriz de pesos das ligações.

```
alfa=0.01;
W_cor=zeros(M_cor,N_cor);
for k=1:2
    for i=1:M_cor %each output neuron i
        for j=1:N_cor % for each input xj
            dW_cor = alfa*S_saida_cor(i,k)*S_entrada_cor(j,k);
            W_cor(i,j)=W_cor(i,j)+dW_cor; %update synaptic weight from input unit j to output neuron i
        end
    end
end
```

Figura 18 - Algoritmo de treino para obter os pesos das ligações. Algoritmo para a cor.

Por fim, as gaussianas de posição obtidas, tanto para a cor como para a forma correspondente, são atribuídas ao parâmetro S de uma nova equação de Amari que corresponde ao campo dinâmico que tomará a decisão final acerca de qual ângulo deve ser atribuído à “waist” tendo em conta as características do objeto. Para este caso, o vermelho apenas tem uma possibilidade de output. No entanto a forma cubo tem duas possibilidades. Assim sendo, haverá um output com o dobro da amplitude em relação ao outro, que corresponderá ao que é comum ao vermelho e ao cubo. Em casos mais complexos, o facto de haver múltiplos outputs possíveis poderá criar ambiguidade. Para o caso estudado neste projeto tal não aconteceu, sendo o campo dinâmico capaz de reconhecer sempre qual o ângulo que deve ser atribuído à “waist”. O seguinte esquema ilustra todo este procedimento para o caso do cubo vermelho:

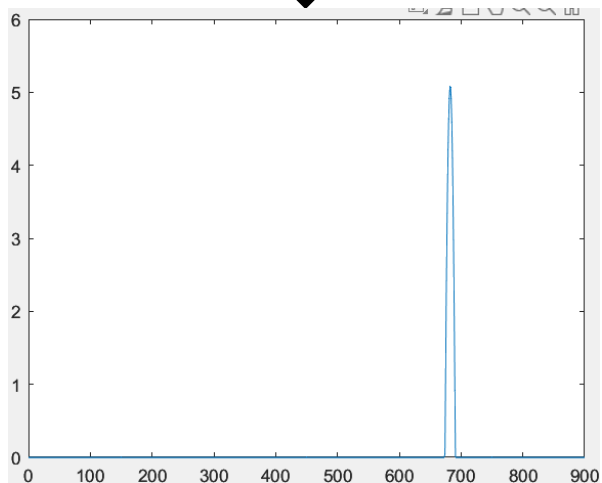
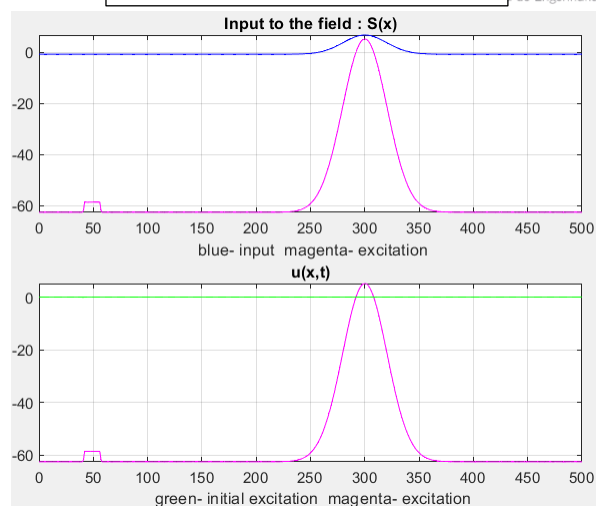
Equação de Amari da cor



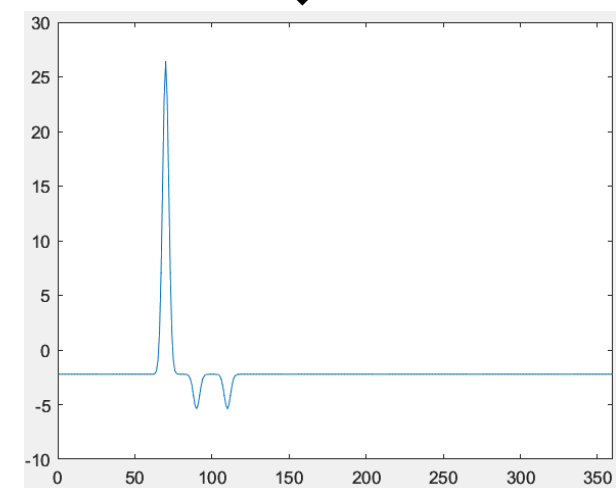
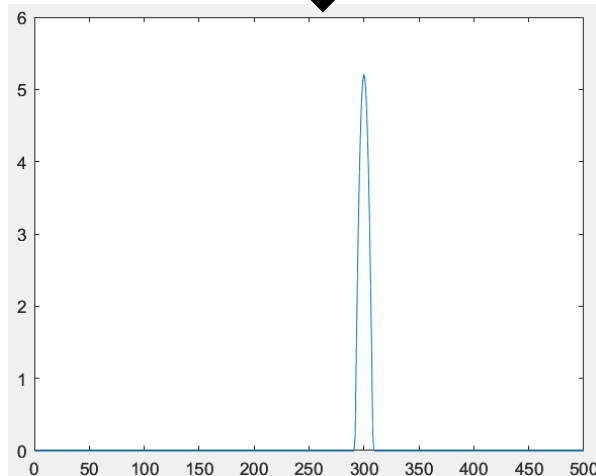
Cubo vermelho

Input nas
equações de
Amari da Cor
e Forma

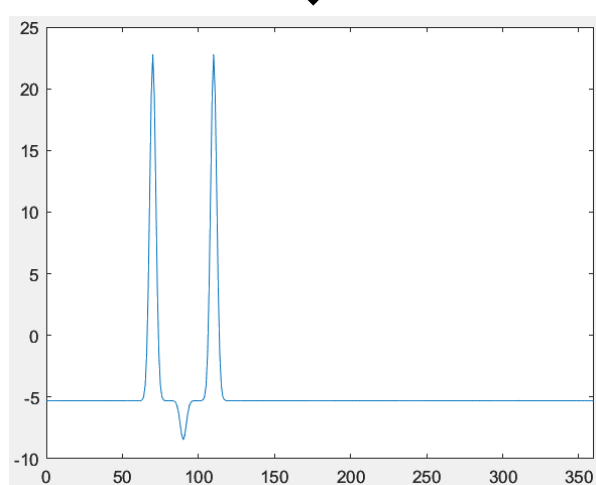
Equação de Amari da posição

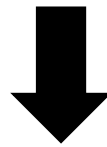


Saída da função
que filtra valores
negativos e
mantém os
valores positivos
do output do
campo anterior

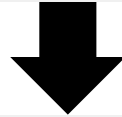
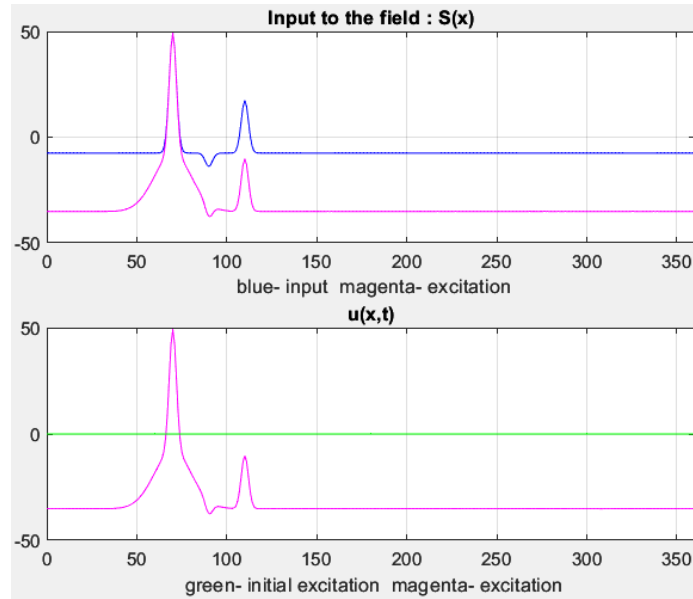


Somatório dos
pesos das
ligações
calculadas pelo
método de
Hebbian com o
output da
função anterior.

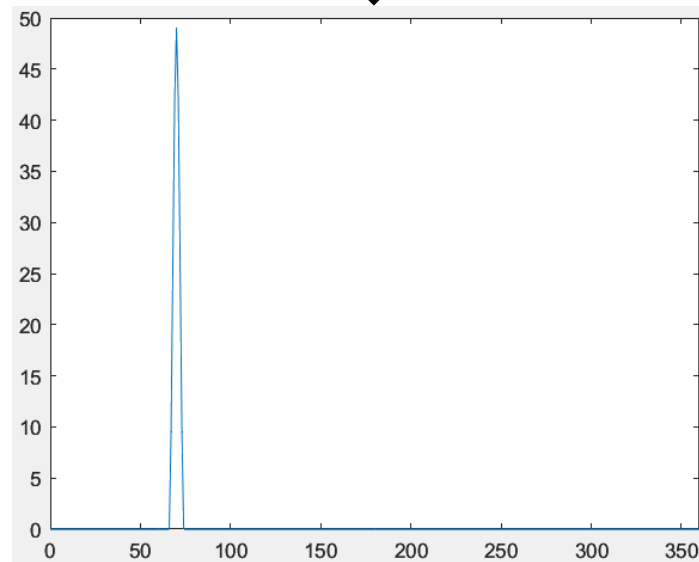




Input de ambos os somatórios no campo da posição




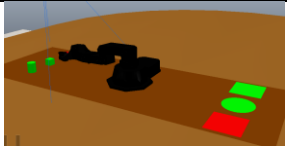


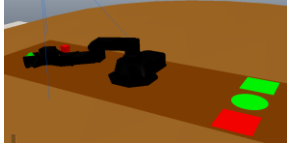
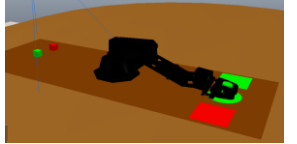

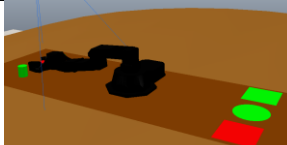


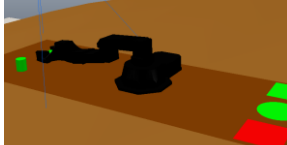

Output do campo neuronal que determina qual o ângulo a atribuir à "waist" verificando qual o ângulo que corresponde ao máximo da função



De modo a implementar os campos neuronais dinâmicos neste projeto, foram criadas funções que recebem como parâmetro as propriedades do objeto retornando um valor para o ângulo da "waist" correspondente à base onde este deve ser colocado.

2.6. Resultados

Depois de validar todas as componentes anteriormente faladas desenvolveu-se um programa capaz de cumprir o objetivo inicial. Num primeiro momento é pedido ao utilizador a indicação de qual objeto deve ser colocado na respetiva base e após obter essa informação o “*vision sensor*” identifica cada objeto presente no espaço retornando a sua posição. Esta serve como entrada para a rede neuronal de cinemática inversa que a converte para valores de ângulos de juntas que possibilitam alcançar o objeto. Por fim são usados os campos dinâmicos para identificar em qual das bases deve ser colocado o objeto selecionado.

Testes	Posição inicial	Captura do objeto	Entrega do objeto
1			
2			
3			
4			

Para comprovar o bom funcionamento de todo o sistema realizaram-se os testes apresentados na tabela acima. O teste 1 mostra a captura do cubo vermelho, selecionado pelo utilizador, e a entrega na sua respetiva base. No teste 2 manteve-se as posições iniciais dos objetos, porém o objeto selecionado foi o cilindro. Para o teste 3 foram invertidas as posições dos cubos garantindo que o sistema de visão distingue a cor independentemente da posição inicial. Por fim no teste 4 a posição inicial do cubo verde foi ligeiramente alterada para demonstrar que a rede consegue dar valores corretos para o ângulo das juntas desde que a posição inicial esteja ao alcance do braço.

3.Implementação no Real

Após validar todos as componentes do projeto em ambiente de simulação, procedeu-se à implementação com o robô real. Tal como esperado, a transição entre simulação e real trouxe novos desafios que não foram necessários ter em conta em Coppeliasim. Estas diferenças vão ser exploradas neste capítulo.

Foram usadas duas cartolinas castanhas para servir de apoio a todo o projeto. Os 3 objetos foram criados utilizando esferovite e cartolina, verde ou vermelha, de onde também se cortaram as bases para os objetos. A figura 19 mostra todo o cenário desenvolvido.

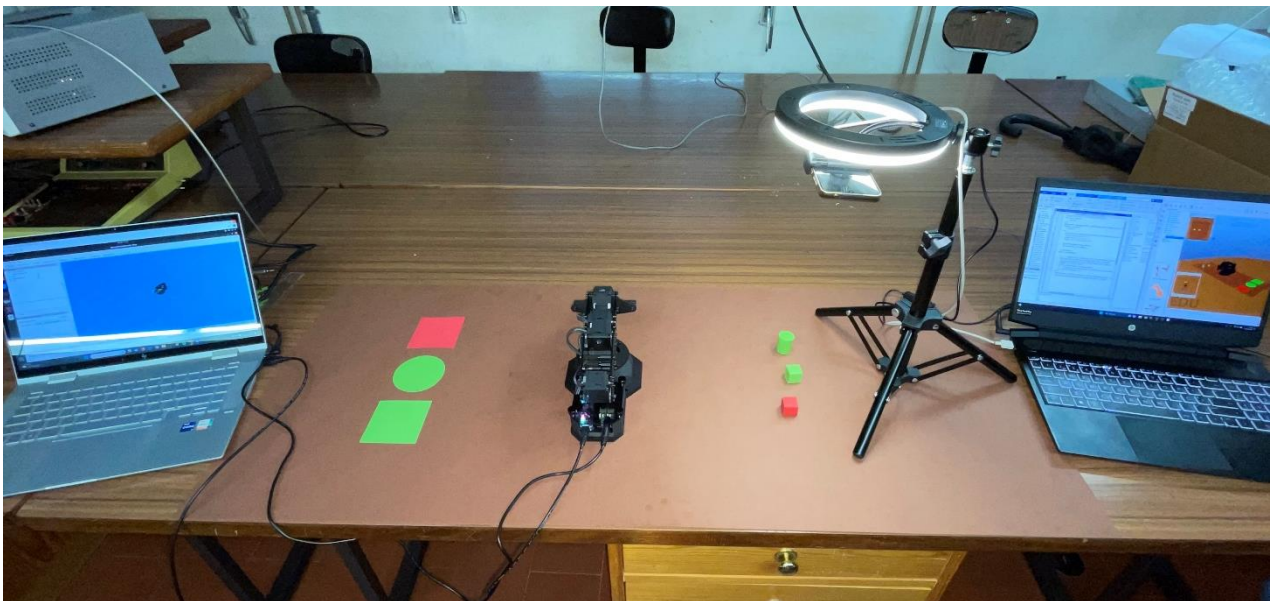


Figura 19 - Cenário implementado para o projeto

3.1 Comunicação com o braço

A comunicação com o braço para lhe enviar instruções foi feita utilizando o ROS1. Uma vez que este apenas funciona em Linux, utilizou-se uma máquina virtual em VMware a correr num PC com Windows 11. Visto que se fez proveito do código já anteriormente desenvolvido em MATLAB para a simulação, apenas acrescentando as funções do ROS, foi necessário estabelecer uma comunicação entre o PC que corre o MATLAB e a máquina virtual que está no outro PC onde o braço está conectado por ligação USB. Para isso, era obrigatório que ambos os computadores estivessem ligados à mesma rede, e uma vez que a rede da universidade introduz vários problemas no que toca à ligação da máquina virtual à mesma, utilizou-se a rede móvel de um telemóvel para ligar ambos os PCs. Depois de iniciado o servidor de ROS na máquina virtual, o computador do MATLAB conecta-se podendo então começar a publicar tópicos para comandar o braço. Tudo isto está representado na imagem 20.

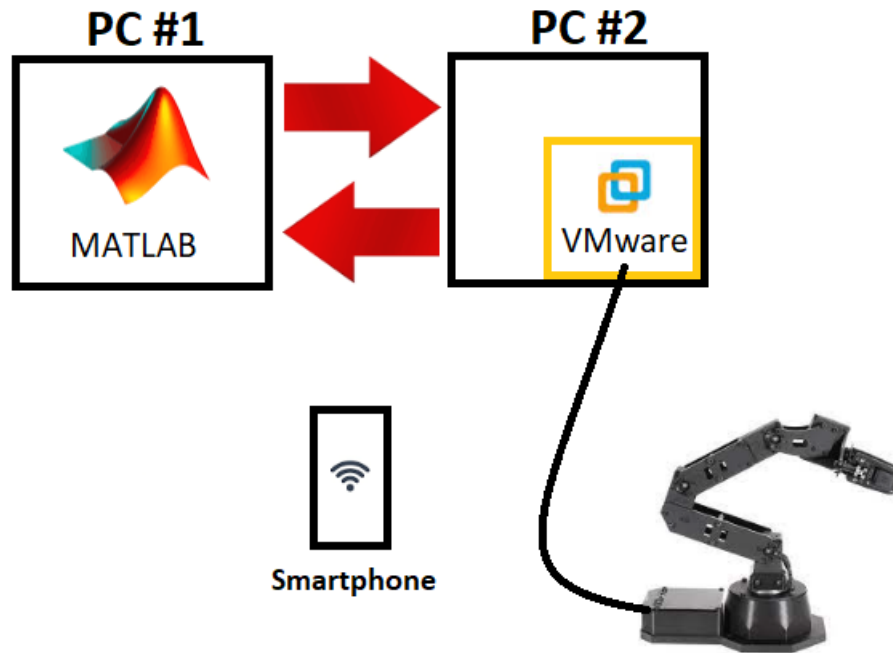


Figura 20 - Esquema de ligações entre computadores e braço

Tal como referido anteriormente, o código utilizado na simulação foi também utilizado para a implementação com o braço real, acrescentando, nas funções de inicialização, as funções que devolvem apontadores para os tópicos. Na função onde se altera o ângulo de uma das juntas, foi colocada a publicação no tópico “px100/commands/joint_single” que envia uma *string* com o nome de qual junta se pretende alterar e também o valor em radianos do ângulo pretendido (fig. 21). No caso das funções que controlam a “gripper”, uma vez que esta, no braço real, é controlada por PWM, para além da *string* “gripper” é também enviado o valor 100 para abri-la e o valor -100 para fechá-la. Para ler valores das juntas, tal como acontecia na simulação, acrescentou-se à função “getJointAngle” a leitura do valor das juntas do braço através da subscrição do tópico “joint_states” do ROS (fig. 22).

```
function [error] = setJointPosition(obj,position,MotorNumber,pub_joint_single,msg_joint_single)

msg_joint_single.Name= obj.Motor_Name{MotorNumber};
msg_joint_single.Cmd= [deg2rad(position)];
send(pub_joint_single, msg_joint_single);
```

Figura 21 - Função de publicação no tópico “joint_single”

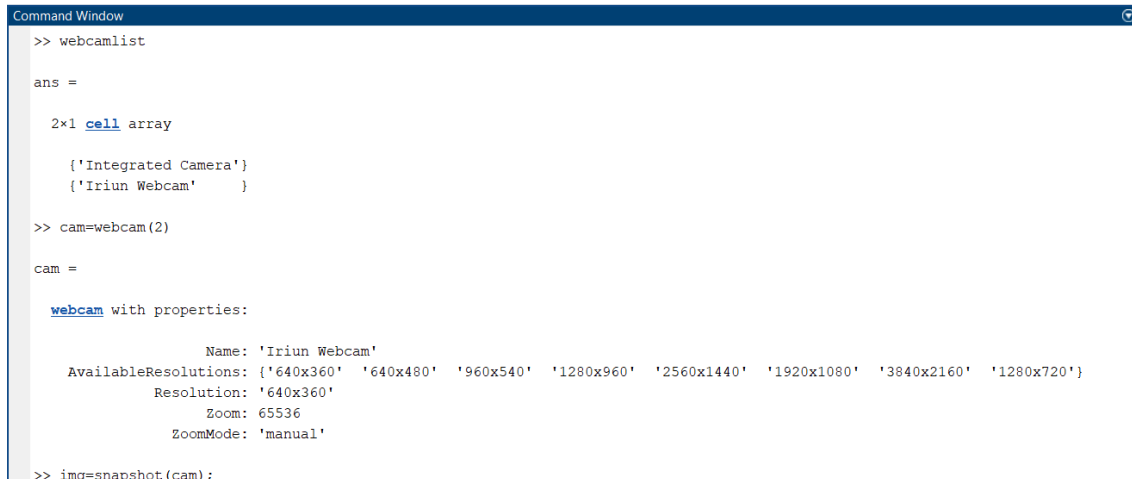
```
function [error,JointAngle,juntas] = getJointAngle(obj,MotorNumber,sub_joint_position)

msg = receive(sub_joint_position);
juntas = rad2deg(msg);
```

Figura 22 - Função de leitura do tópico “joint_states”

3.2 Visão por computador no teste real

Na simulação foi utilizado um *vision sensor* para obter imagens do ambiente. Ao passar para a implementação na realidade foi necessário utilizar um dispositivo com a capacidade de captar imagens e enviá-las para o MATLAB para que estas sejam processadas. De maneira a reduzir os custos do projeto, o grupo utilizou um telemóvel como webcam. Neste caso foi utilizado um dispositivo Android onde foi instalada a aplicação “Iriun Webcam” pelo Google Play. No computador que corre o MATLAB foi instalado o software, com o mesmo nome, para Windows. Este permite simular uma webcam do PC através do telemóvel, enviando um *live feed* da câmara do smartphone para o computador por Wi-Fi ou por USB, que foi o método escolhido pelo grupo. No MATLAB, através da *toolbox* “MATLAB Support Package for USB Webcams” é possível aceder ao *live feed* do telemóvel e obter imagens. Na figura 23 é exemplificado como utilizar as funções desta *toolbox* para obter imagens da câmara do smartphone.



```
Command Window
>> webcamlist

ans =

    2x1 cell array

    {'Integrated Camera'}
    {'Iriun Webcam'      }

>> cam=webcam(2)

cam =

    webcam with properties:
        Name: 'Iriun Webcam'
    AvailableResolutions: {'640x360' '640x480' '960x540' '1280x960' '2560x1440' '1920x1080' '3840x2160' '1280x720'}
        Resolution: '640x360'
        Zoom: 65536
        ZoomMode: 'manual'

>> img=snapshot(cam);
```

Figura 23 - Procedimento para obter uma imagem da câmara do telemóvel em MATLAB

A maior diferença em termos de visão, ao transitar da simulação para o real, foi o lidar com sombras e imperfeições nas imagens obtidas pela câmara. Para tentar minimizar estes problemas e também para fixar a câmara numa posição semelhante á do *vision sensor* na simulação, foi utilizado *ring light*. Desta maneira, foi possível obter uma imagem com menos sombras e sempre da mesma perspetiva. A figura 24 mostra como foi implementado.



Figura 24 - Estrutura para obter imagens do cenário real

Em termos de processamento de imagem, foram feitas algumas alterações, sendo uma delas o *resize* da imagem original captada pela câmara para uma resolução mais baixa de forma a conseguir reduzir as imperfeições dos objetos. Para distinguir os objetos por cor e forma utilizou-se exatamente os mesmos procedimentos, porém com novos parâmetros (fig. 25).



Figura 25 - Processamento da imagem para a cor verde

Para obter a posição dos objetos foi necessário gerar novas equações para as coordenadas uma vez que a posição da câmara não corresponde exatamente à posição do *vision sensor* na simulação (fig. 26).

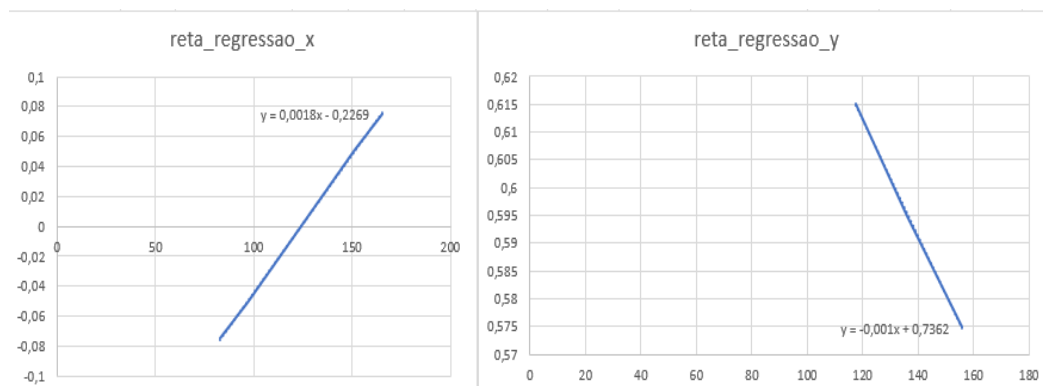


Figura 26 - Equações que relacionam as coordenadas da câmara com as coordenadas reais

3.3 Redes neuronais

A rede já treinada em simulação não apresentou resultados satisfatórios ao utilizá-la no cenário real. Para contornar esse problema foi criada uma rede neuronal de cinemática inversa nova. O método de treino desta foi exatamente igual ao utilizado nas redes anteriores apenas diferenciando a forma como foram recolhidos os dados.

Para a recolha da matriz de entrada, matriz das posições, foram colocadas várias referências de posição no campo de visão da câmara (fig. 27) e as suas coordenadas foram convertidas para coordenadas reais utilizando a reta de regressão calculada acima.



Figura 27 - Referências utilizadas para obter as posições para treinar a rede neuronal

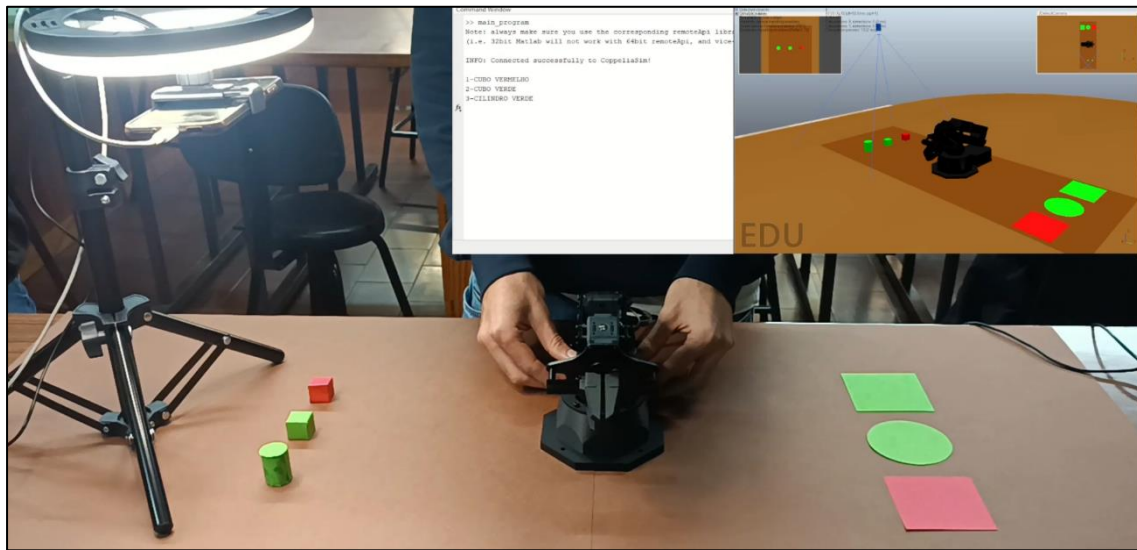
Quanto à matriz de saída, matriz dos ângulos, os seus valores foram obtidos colocando o braço real na posição de cada referência e guardado os valores dos ângulos para cada junta, para isto foi utilizada a função “*getJointAngle*” que obtém esses valores a partir da subscrição do tópico “*joint_states*” do ROS.

Com esta nova rede neuronal os resultados foram muito satisfatórios.

3.4 Resultados

Após a transição, com sucesso, de todas as componentes em simulação para o real, efetuaram-se testes semelhantes aqueles que foram realizados em CoppeliaSim.

O primeiro teste consiste em colocar todos os objetos na respetiva base, sendo o utilizador quem indica a ordem pela qual estes vão ser manipulados. Em simultâneo com o movimento do braço, foram monitorizados o *digital twin*, a *command window* do MATLAB e os gráficos dos campos neuronais.



O braço inicia de uma posição de repouso com a “*gripper*” fechada. De seguida é comandado pelo programa que este assuma uma posição, “*ready*”, que equivale a ter todas as juntas com ângulo 0 graus. O sistema de visão informa o utilizador quais são os objetos que está a detetar sendo esperado que o utilizador selecione qual deles deve ser colocado na sua base. Neste caso o utilizador escolheu o terceiro objeto detetado pelo sistema que é o cilindro verde.

```

Command Window

>> main_program
Note: always make sure you use the corresponding remoteApi libr
(i.e. 32bit Matlab will not work with 64bit remoteApi, and vice

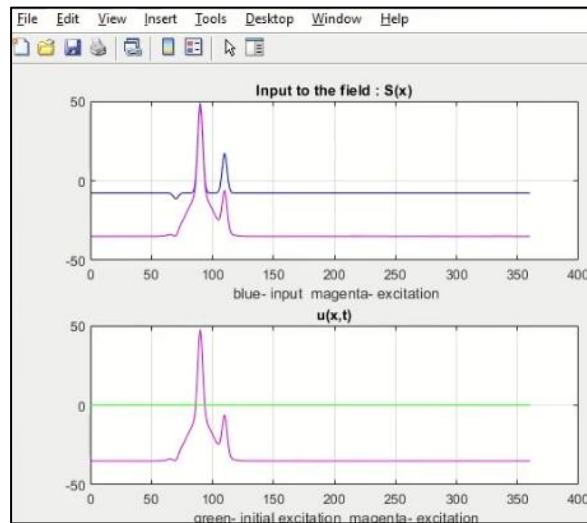
INFO: Connected successfully to CoppeliaSim!

1-CUBO VERMELHO
2-CUBO VERDE
3-CILINDRO VERDE
SELECIONE OBJETO: 3

objeto_selecionado =

    3
  
```

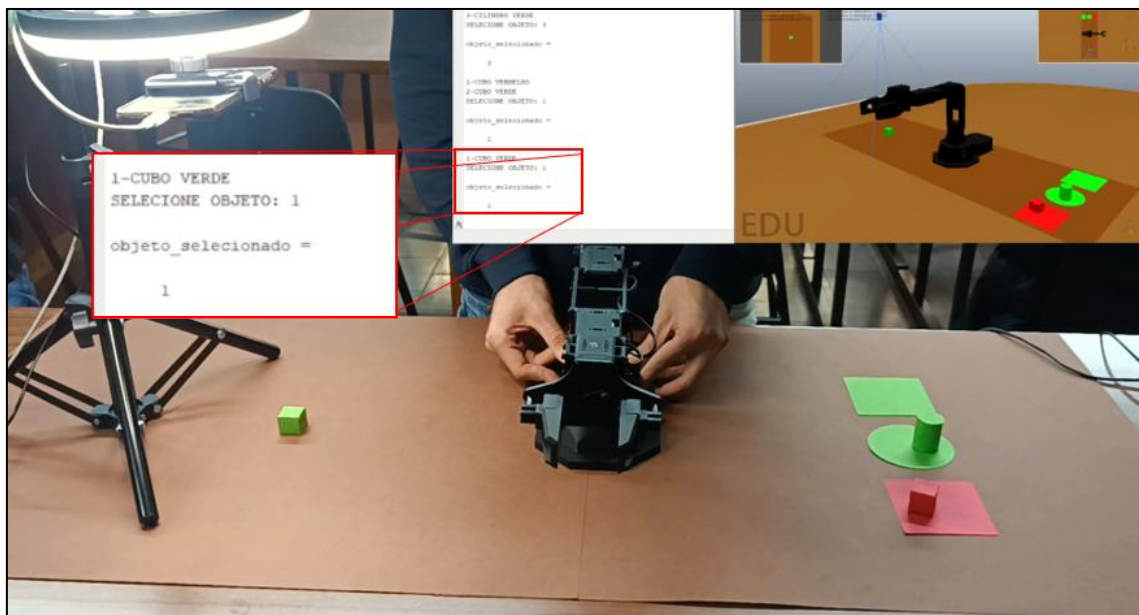
Após o programa saber qual o objeto que foi escolhido pelo utilizador, as características da sua cor e forma são atribuídas como parâmetros para as funções que calculam o ângulo da “*waist*” correspondente à base correta. Para além disso o sistema de visão informa à rede neuronal de cinemática inversa qual a posição do objeto escolhido. Por sua vez, esta retorna valores, para todas as juntas do braço, que permitem alcançá-lo.



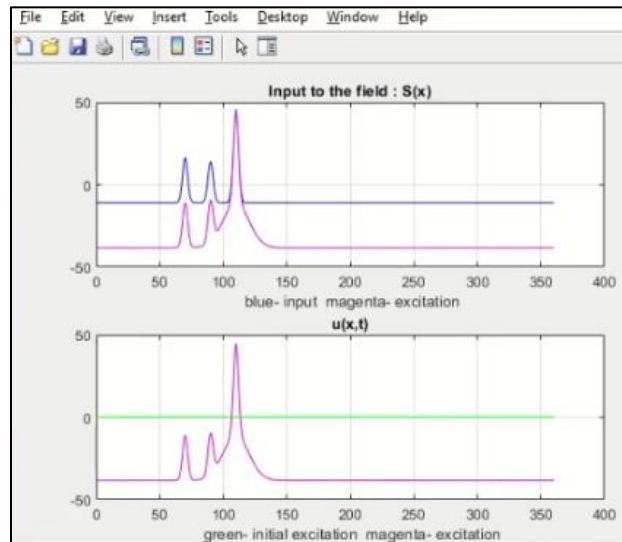
Os gráficos acima mostram o output dos campos para o cilindro verde. É possível observar que existem dois picos centrados em dois ângulos distintos, sendo que isto deve-se ao facto da cor verde ter duas possibilidades para ângulo da “waist”. No entanto, uma vez que a forma cilindro apenas tem uma possibilidade, a que é comum aos dois destaca-se e no final esse foi o único pico que se encontra acima de zero, sendo então reconhecido como output (linha magenta do gráfico).

Depois de pousar o cilindro no sítio correto, o utilizador escolheu um novo objeto. Neste caso, foi o cubo vermelho. Esta situação é muito parecida à do cilindro verde, sendo a única diferença o facto da ambiguidade estar presente na sua forma e não na sua cor.

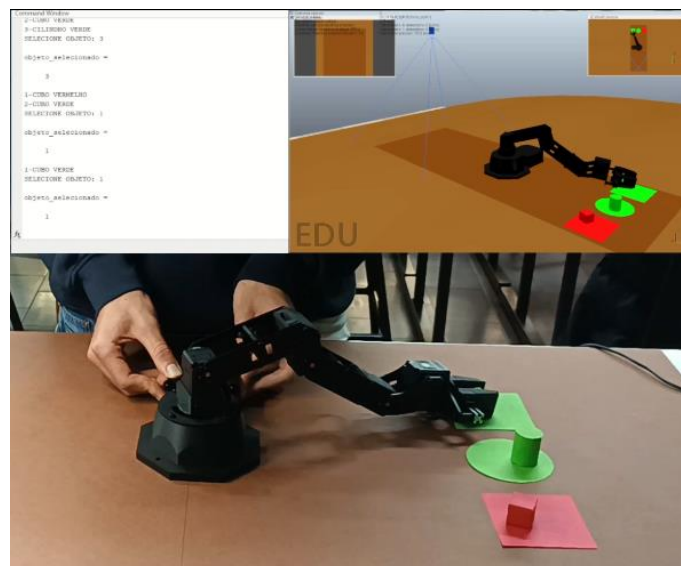
Por fim, o único objeto disponível para o braço manipular foi o cubo verde.



Este, ao contrário dos anteriores, apresenta ambiguidade em ambas as características pois existem dois objetos verdes e dois objetos com forma cúbica. Isto irá criar um pico em cada uma das possibilidades para ângulo da waist. No entanto, tal como nas situações anteriores, o ângulo que é uma das possibilidades em ambas as características, terá um pico com o dobro da amplitude. Desta maneira, o campo dinâmico identifica corretamente qual o ângulo para este objeto.



Na imagem acima é possível observar os três picos criados em cada ângulo possível para as juntas. Conclui-se que, uma vez que o campo dinâmico identificou acertadamente o output sempre que existe uma gaussiana com maior amplitude em relação às outras, isso será suficiente para tomar uma decisão.



Durante este teste foi verificado o funcionamento do *digital twin* criado em Coppeliasim. Na imagem acima observa-se que o movimento do braço é espelhado na simulação

Este teste ficou concluído assim que se verificou que todos os objetos foram corretamente identificados e colocados nos sítios certos.



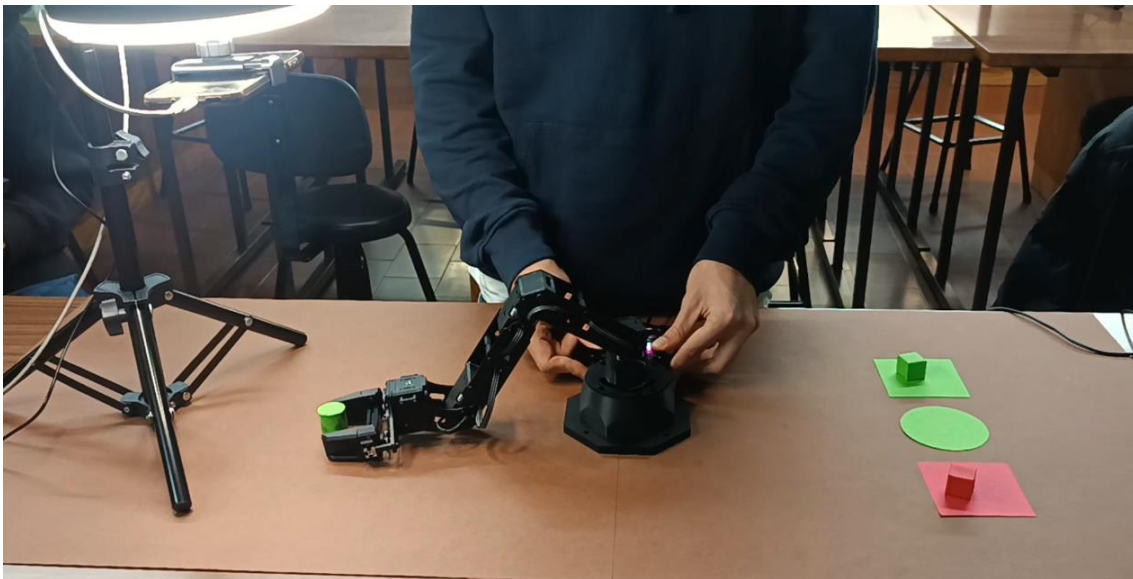
Como segundo teste, decidiu-se alterar a posição de um dos objetos enquanto o braço coloca um dos outros na base correta. Isto tem o objetivo de garantir que a visão é atualizada de cada vez que o braço fica disponível e também para testar se a rede neuronal de cinemática inversa atribui valores corretos às juntas para uma posição para a qual não foi treinada.



Acima, vê-se que o braço alcança o cubo verde, a pedido do utilizador. Nesta situação o cubo vermelho já foi colocado no sítio correto.



Quando o braço se dirige para a base do cubo verde, alterou-se a posição do cilindro verde.



No final, verificou-se que o braço foi capaz de ir buscar o cilindro á sua nova posição, significando que o sistema é capaz de se adaptar a estas situações.

Com estes dois testes concluídos com sucesso, o grupo deu por terminado o projeto.

4. Conclusão

Com este projeto foi possível colocar em prática os conhecimentos adquiridos durante as aulas de Sistemas Autónomos Inteligentes. Nomeadamente, foram consolidados os conhecimentos em redes neuronais, campos neuronais dinâmicos, visão por computador e ROS. Para além disso, o grupo aprendeu com detalhe a utilizar tanto o CoppeliaSim como também várias ferramentas do MATLAB que se mostraram bastante úteis.

O maior desafio neste projeto foi o treino da rede de cinemática inversa pois existem vários conjuntos de valores para juntas que permitem ao braço alcançar a mesma posição. Outro desafio foi a construção do cenário quando se iniciou a implementação com o braço real. Foi necessário garantir que todas as posições relativas entre o braço e a visão se mantinham as mesmas durante toda a fase de testes.

Achamos que as soluções encontradas pelo grupo para mitigar este problema foram adequadas, principalmente a utilização do tripé com iluminação pois permitiu diminuir bastante o ruído das imagens e manter a câmara na mesma posição. Houve ainda alguns problemas de conexão ao servidor ROS que o grupo conseguiu contornar apesar de, ainda assim, a ligação falhar esporadicamente.

Concluimos que existem alguns aspetos a melhorar sendo o principal aspeto o cenário da implementação no real. Este deveria estar montado numa base onde se fixaria o braço, pois como é possível ver nos testes efetuados, um elemento do grupo estava a segurá-lo. Nesta base também se deveria prender o dispositivo com o qual se captura as imagens sendo este, idealmente, uma webcam. Para este projeto foi utilizado um smartphone com o intuito de reduzir custos, no entanto, isto torna-se uma desvantagem pois o smartphone tem de ser retirado no final dos testes, ao contrário do que aconteceria com uma webcam dedicada para o projeto.

Quanto aos campos neuronais dinâmicos, obtiveram-se bons resultados, porém, neste projeto o problema que era solucionado por eles era simples, servindo apenas para consolidar os conceitos básicos deste tópico.

Por fim, o grupo ficou satisfeito com os resultados, agradecendo aos professores por toda a ajuda dada ao longo do desenvolvimento deste projeto.

5.Referências

- <https://www.trossenrobotics.com/pincherx-100-robot-arm.aspx>
- https://www.mathworks.com/help/images/index.html?s_tid=CRUX_topnav
- <https://www.mathworks.com/help/deeplearning/ref/fitnet.html>
- <https://www.mathworks.com/matlabcentral/fileexchange/45182-matlab-support-package-for-usb-webcams>
- Sandamirskaya, Y. (2014). Dynamic neural fields as a step toward cognitive neuromorphic architectures.
- Material de estudo da Unidade Curricular Sistemas Autónomos Inteligentes.