UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA
CURSO DE ENGENHARIA MECÂNICA

MÁRIO JORGE LIMA SEIXAS AGUIAR

SISTEMA DE CLASSIFICAÇÃO E SEPARAÇÃO DE PEÇAS POR VISÃO COMPUTACIONAL USANDO O BRAÇO ROBÓTICO COMAU NS 16 1.65

Recife
2019

MÁRIO JORGE LIMA SEIXAS AGUIAR

SISTEMA DE CLASSIFICAÇÃO E SEPARAÇÃO DE PEÇAS POR VISÃO COMPUTACIONAL USANDO O BRAÇO ROBÓTICO COMAU NS 16 1.65

Trabalho de Conclusão de Curso apresentado ao Departamento de Engenharia Mecânica da Universidade Federal de Pernambuco como exigência para obtenção de grau de Bacharel em Engenharia Mecânica.

Área de concentração: Mecatrônica

Orientador: Prof. Dr. João Paulo Cerquinho Cajueiro

Recife

2019

Catalogação na fonte
Bibliotecária Maria Luiza de Moura Ferreira, CRB-4 / 1469

A282s Aguiar, Mário Jorge Lima Seixas.
Sistema de classificação e separação de peças por visão computacional usando o braço robótico COMAU NS 16 1.65 / Mário Jorge Lima Seixas. - 2019.
105 folhas, il.

Orientador: Prof. Dr. João Paulo Cerquinho Cajueiro.

TCC (Graduação) – Universidade Federal de Pernambuco. CTG. Departamento de Engenharia Mecânica, 2019.
Inclui Referências e Apêndices.

1. Engenharia Mecânica. 2. Visão computacional. 3. Braço robótico. 4. Python. 5. OpenCv. 6. Template matching. I. Cajueiro, João Paulo Cerquinho (Orientador). II. Título.

UFPE

621 CDD (22. ed.) BCTG/2019-314

MÁRIO JORGE LIMA SEIXAS AGUIAR

SISTEMA DE CLASSIFICAÇÃO E SEPARAÇÃO DE PEÇAS POR VISÃO COMPUTACIONAL USANDO O BRAÇO ROBÓTICO COMAU NS 16 1.65

Trabalho de Conclusão de Curso apresentado ao Departamento de Engenharia Mecânica da Universidade Federal de Pernambuco como exigência para obtenção de grau de Bacharel em Engenharia Mecânica.

Aprovada em: 03/07/2019.

BANCA EXAMINADORA

Profº. João Paulo Cerquinho Cajueiro (Orientador)

Universidade Federal de Pernambuco

Profº. Jacinaldo Balbino de Medeiros Junior (Examinador Interno)

Universidade Federal de Pernambuco

Profº. João Marcelo Teixeira (Examinador Interno)

Universidade Federal de Pernambuco

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE PERNAMBUCO
DEPARTAMENTO DE ENGENHARIA MECÂNICA
PROGRAMA DE GRADUAÇÃO EM ENGENHARIA MECÂNICA

DECLARAÇÃO

Declaramos, para os devidos fins, que no dia 03 de julho de 2019, durante a cerimônia da Defesa de Trabalho de Conclusão de Curso do aluno **Mário Jorge Lima Seixas Aguiar**, intitulada "SISTEMA DE CLASSIFICAÇÃO E SEPARAÇÃO DE PEÇAS POR VISÃO COMPUTACIONAL USANDO O BRAÇO ROBÓTICO COMAU C5G", houve a sugestão de modificação do título do referido Trabalho de Conclusão de Curso para "SISTEMA DE CLASSIFICAÇÃO E SEPARAÇÃO DE PEÇAS POR VISÃO COMPUTACIONAL USANDO O BRAÇO ROBÓTICO COMAU NS 16 1.65", o que foi acatado, em comum acordo, pelos membros titulares da banca, doutores **João Paulo Cerquinho Cajueiro** (orientador), **João Marcelo Teixeira,** e mestre **Jacinaldo Balbino de Medeiros Junior**.

Recife, 26 de julho de 2019.

Prof. José Maria Barbosa
Coordenador de Trabalho de Conclusão de Curso – TCC
Curso de Graduação em Engenharia Mecânica – CTG/EEP-UFPE

AGRADECIMENTOS

Agradeço primeiramente a Deus, por tudo e todos que por Sua vontade me fez chegar a esse ponto do espaço-tempo.

Ao meu pai, Mário Jorge, em quem eu sempre me espelhei, agradeço por todo o apoio e motivação ao longo da vida; cada sacrifício, conversa e suporte me tornaram a pessoa que sou. E à minha madrasta, Giovanna Seixas, por nossa criação e estímulos para educação; fundamental para estabilidade e tranquilidade necessária para meu desenvolvimento.

Gratidão imensurável a Victor Gomes Cardoso por todo suporte técnico, tempo, paciência e ideias no desenvolvimento do projeto; sem dúvida, a segurança que passou me fez superar as incertezas ao longo do projeto.

A todos que estiveram presentes ao longo da minha graduação, aqui representados por Maria Júlia; sou muito grato por toda ajuda, motivação e troca de conhecimentos.

Aos professores que contribuíram para minha formação e que de alguma forma melhoram a universidade através de projetos práticos, em especial o Profº. João Paulo, orientador e idealizador do presente projeto, só tenho a agradecer por todo suporte, diretrizes e aprendizados proporcionados.

RESUMO

No presente trabalho foi desenvolvido um sistema integrado de visão computacional aplicado à manipulação de um braço robótico industrial. O sistema de visão, responsável pela classificação de três peças distintas e identificação de suas posições em um campo de trabalho restrito por cores diferentes, foi projetado para agrupar as informações da imagem referentes às peças e enviá-las ao braço robótico para fazê-lo atuar. A técnica de processamento principal utilizada no reconhecimento dos objetos foi o *Template Matching*, tornando necessária a utilização de padrões de imagens das peças para o funcionamento. Para este trabalho foram utilizadas as linguagens de programação *Python*, auxiliada das bibliotecas *OpenCv* e *Socket*, responsáveis pelo processamento de imagem e envio de dados, respectivamente; e a linguagem PDL2, responsável pelo *script* de movimentações do braço baseado nas informações recebidas do *Python*. Os *hardwares* utilizados para esta aplicação foram um *notebook*, um *smartphone* e a central de comandos do braço robótico. O objetivo do projeto é validar fisicamente o funcionamento do sistema classificador e separador de peças através da visão computacional, tornando possível a aplicação em indústrias de diferentes maneiras, com as devidas adaptações. Apesar de sempre existirem questões a serem melhoradas, os resultados obtidos foram satisfatórios para atender ao objetivo principal de classificar, localizar e fazer um braço robótico movimentar-se até a localização obtida, capturando a peça e levando-a a uma posição final associada ao tipo de peça identificada.

Palavras-chave: Visão computacional. Braço robótico. *Python*. *OpenCv*. *Template matching*.

ABSTRACT

In the present work an integrated computer vision system applied to the manipulation of an industrial robotic arm was developed. The vision system, which is responsible for classifying objects and identifying their positions in a different color-restricted work area, is designed to group the image information about the objects and send them to the robotic arm to make it move. The main processing technique used in object recognition was Template Matching, making it necessary to use image patterns of the objects for the operation. For this work we used Python programming language, supported by the OpenCv and Socket libraries, responsible for image processing and sending of data, respectively; and the PDL2 language, responsible for the script of arm movements based on information received from Python. The hardware used for this application was a notebook, a smartphone and the robotic arm command center. The goal of the the project is to physically validate the operation of the classifier system and objects separator through computer vision, making it possible to apply to industries in different ways with appropriate adaptations. Although there are always issues to be improved, the results obtained were satisfactory to meet the main objective of classifying, locating and making a robotic arm move until the location obtained by capturing the part and taking it to a final position associated with the identified part type.

Keywords: Computer vision. Robotic arm. *Python*. *OpenCv*. *Template Matching*.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Em meados do século XVIII, a Revolução Industrial barateou em muito vários bens de consumo, aumentando o poder de compra da população e melhorando a qualidade de vida de boa parte do mundo. Essa ampliação foi conquistada com o passar dos anos à medida que as indústrias se desenvolviam e a população crescia. Todo esse desenvolvimento gerado pela revolução fez surgir processos industriais de maior rapidez e produção.

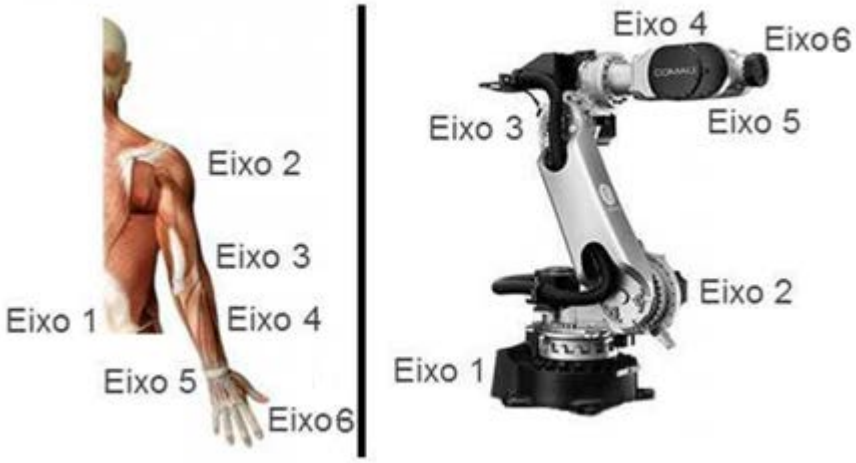
Naquela época, a revolução trouxe ao mundo as máquinas. Nos tempos atuais, mais uma vez motivado pela satisfação dos clientes, a exigência para se ter maior precisão e qualidade em suas produções vem fazendo as engenharias trabalharem para cada vez mais trazerem ao mundo as máquinas inteligentes.

A história da humanidade mostra que ao longo dos anos as civilizações se desenvolveram em paralelo as diversas descobertas tecnológicas que diminuíam de alguma forma a necessidade do esforço físico humano na realização de tarefas. Hoje a busca por novas tecnologias tem gerado uma forte ascensão no desenvolvimento de robôs. Inspirados em aparições mitológicas, literárias e em filmes, suas aplicações têm se adaptado desde serviços industriais, domésticos e militares até aplicações para o entretenimento (CHOSET e LYNCH, 2005).

Dentro da diversidade existente de robôs industriais, encontram-se os chamados braços robóticos. Esses braços são máquinas manipuladoras, controladas automaticamente, que podem ter uma base fixa ou móvel para aplicação em automação industrial. A arquitetura mecânica desses braços é comumente utilizada para posicionar uma específica mão-ferramenta que é o que realiza de fato o trabalho útil do robô (ORMINDO e ALVES, 2014).

A flexibilidade desse tipo de robô deve-se muito a suas características antropomórficas, isto é, semelhantes às do ser humano, com em média seis graus de liberdade, determinados por seus eixos rotativos (COMAU, 2016). Normalmente essa associação é feita com relação aos cinco eixos biológicos humanos (tronco, ombro, braço, antebraço e punho), podendo ser expandida para o quanto se queira, caso levados em consideração garras ou mesmo quando associados aos robôs humanóides. Essa semelhança com as articulações de membros superiores humanos é o que justifica a caracterização deste tipo de robô ao seu nome "braço" robótico. A associação pode ser observada na Figura 01 a seguir.

Figura 01 - Associação entre eixos articulados humanos e robóticos.



Fonte: Adaptado de COMAU, 2016

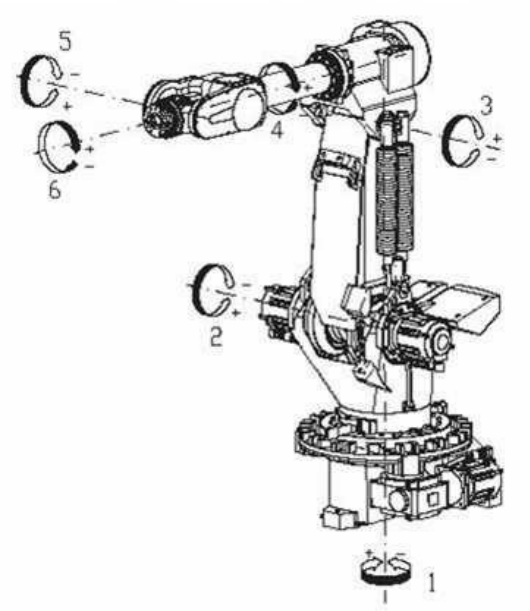
A aplicação da robótica nas indústrias brasileiras é uma realidade gradativa que iniciou com a chegada das primeiras indústrias automotivas ao país, garantindo segurança, produtividade e precisão de forma nunca antes vista. Entre tantos braços robóticos utilizados em indústrias, o presente trabalho foi baseado em um braço da COMAU, fabricante especializada nesse segmento, por ser o que temos à disposição.

Os braços robóticos da COMAU são desenvolvidos de forma a se moverem para qualquer posição, dentro dos limites geométricos, a partir do movimento de seus eixos de rotação. Dentre as possibilidades de aplicação, as ferramentas de solda e os mecanismos de garras para captura de objetos e realização de trabalhos são os mais utilizados dentro das indústrias.

A ideia desenvolvida neste trabalho é a de localizar e classificar peças a serem manipuladas pelo braço robótico usando visão computacional e passar suas coordenadas para o sistema de controle do braço robótico. Atualmente, existem empresas que já oferecem esse tipo integração disponível em forma de pacotes de acessórios para venda, entretanto, para o robô utilizado nesse trabalho essa integração não foi disponibilizada. É conseguida, portanto, a partir da utilização de uma câmera e *hardware* externos ao sistema do braço que gerem os resultados do sensoriamento por imagem enviando informações de localização e classificação das peças para guiar os movimentos.

O braço robótico e seus eixos nos quais se tentará implementar o sistema de visão computacional pode ser visto na Figura 02.

Figura 02 - Ilustração dos seis eixos articulados do braço robótico.



Fonte: COMAU, 2016

A manipulação de robôs como este normalmente é feita baseada em um conjunto de sistemas de coordenadas definidas pelo fabricante ou pelo usuário. Em trabalhos usuais, o robô executa um ciclo de movimentações entre pontos pré-programados do sistema escolhido. Em trabalhos recentes, esta programação vem sendo substituída por outra forma de definição de posições dentro do sistema de coordenadas escolhido: a visão computacional.

Visão computacional ou visão de máquinas é um conjunto de técnicas que permite aos computadores interpretar a imagem que está sendo capturada, seja ela estática ou em tempo real, e acionar saídas programáveis como se queira (GONZALEZ e WOODS, 2002). Baseia-se inevitavelmente, portanto, nas técnicas de processamento digital de imagens.

Aplicações com utilização de visão computacional são inúmeras em qualquer ramo da engenharia, e por vezes, da medicina. E apesar de parecer algo muito recente e sofisticado têm sido usadas desde a década de 70, em satélites de mapeamento terrestre, por meio do processamento digital da imagem (SCHOWENGERDT, 2000).

Inúmeros projetos são desenvolvidos na atualidade com o intuito de integração da visão computacional com algum mecanismo robótico atuador. No trabalho desenvolvido por Nascimento, Santos e Sorrentino (2016), por exemplo, um robô autônomo foi projetado para se locomover através da identificação de um objeto específico, no caso, bolas de tênis, como forma de sinalização para seus movimentos. O robô seguia a imagem sinal com algoritmos

desenvolvidos na mesma linguagem de programação e as mesmas bibliotecas almejadas para o presente trabalho.

Existem ainda projetos que abordam as técnicas de rastreamento e acompanhamento de objeto baseado em visão computacional, novamente trazendo a integração entre os campos de conhecimento. A arquitetura trazida por estes trabalhos, como mostrado no trabalho exemplo de Pestana (2014), permite que o usuário especifique um objeto o qual o robô deverá seguir a partir de uma distância aproximada, simulando casos onde há perda de rastreamento de imagem, para o aguardo da recuperação de rastreamento de imagem ou a segunda detecção.

A integração entre os campos da visão computacional e a automação, por meio de braços robóticos, tem crescido no âmbito industrial nos últimos anos. As aplicações normalmente referem-se a investimentos para otimização de processos e redução de custos das empresas, mas são usados também para a qualidade dos produtos.

O desenvolvimento e aplicação dessas novas tecnologias de automação das indústrias com o intuito de maiores controles dos processos fez surgir o novo conceito de funcionamento de indústrias ao redor do mundo, a indústria 4.0. Essa recente ideia, responsável por integrar as principais inovações tecnológicas dos campos de automação, controle e tecnologia da informação, aplicadas aos processos industriais, é o principal motivo dos estudiosos estarem considerando o fato como um novo período no contexto das grandes revoluções industriais.

De acordo com Palma e Bueno (2017), os cinco princípios da indústria 4.0 são: capacidade de operação em tempo real, virtualização, descentralização, orientação a serviços e modularidade. A nova revolução industrial traz consigo diversas mudanças operacionais dos sistemas industriais. Uma dessas mudanças é a integração das tecnologias para adaptação de processos, tal como o controle de braços robóticos a partir de a visão computacional.

Desta forma, soluções semelhantes as dos projetos citados podem servir para possíveis aplicações em indústrias que estejam vislumbrando a ideia de Indústria 4.0, como forma de melhoria de seus processos. Projetos como estes surgem todos os dias, e são aplicados sempre que uma oportunidade é visualizada nas indústrias, girando o fluxo de desenvolvimento das tecnologias no mundo.

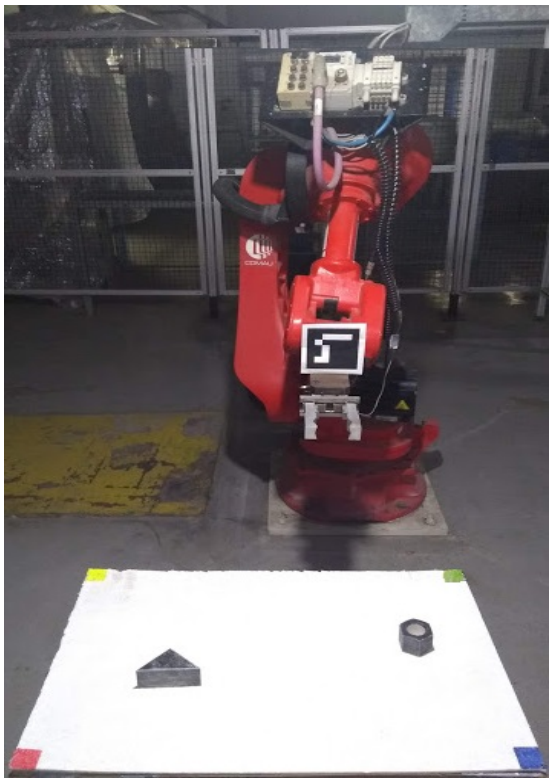
A motivação no desenvolvimento de novas soluções é a de aumentar a produtividade de linhas industriais, a fim de reduzir tempos de *setup* de máquinas e custos com pessoal e

energia. Isso otimizaria linhas e diversificaria a utilização da máquina, tal como preconiza a Indústria 4.0.

Este trabalho foca mais especificamente em implementar a funcionalidade de classificador e separador de peças, que automaticamente identifica pela geometria das peças que tipo de peça está na área de trabalho e já correlaciona o tipo à sua posição final. Tal sistema também identifica a posição e orientação da peça, gerando os dados necessários para que o braço mova-se para as coordenadas em que foi identificada a peça, pegue-a e guarde-a na posição final, definida para cada tipo de peça.

A solução que aqui será abordada foi desenvolvida a partir de um braço robótico específico, o braço robótico COMAU NS 16 1.65 instalado no galpão do Departamento de Engenharia Mecânica da Universidade Federal de Pernambuco (Figura 03). Com o sucesso da pesquisa, o trabalho poderá expandir-se para outras máquinas como esta, instaladas em setores industriais diversos.

Figura 03 - Braço robótico do departamento de eng. mecânica da UFPE.



Fonte: Elaborado pelo Autor, 2018.

O equipamento está instalado em um ambiente de aproximadamente $6m^2$ e funciona como tantos outros braços robóticos através de programações com posições e movimentações pré-definidas ou em modo manual através de um terminal de programação - TP.

Para explicitar a ideia do trabalho, os tópicos a seguir foram separados para melhor descrever os objetivos gerais e específicos que deverão ser concluídos com êxito ao final do projeto.

1.1 Objetivo geral

Desenvolver um sistema de controle da movimentação de um braço robótico com coordenadas obtidas por visão computacional.

1.2 Objetivos específicos

- Desenvolver um algoritmo de visão computacional que identifique e classifique objetos e retorne sua posição e orientação atual em um sistema de coordenadas estabelecido;
- Definir sequência de ações para cada possibilidade de peça identificada;
- Desenvolver algoritmo na linguagem computacional do braço para efetuar a sequência de ações de acordo com as informações recebidas pelo sistema de visão computacional;
- Estabelecer comunicação entre o sistema de visão computacional e braço robótico.

Este texto está organizado da seguinte maneira: o capítulo 2 apresenta a fundamentação teórica do projeto, tanto da parte do braço robótico quanto da parte de identificação por imagem, o capítulo 3 apresenta a metodologia utilizada no desenvolvimento do projeto, no capítulo 4 descrevem-se os resultados obtidos com o sistema, no capítulo 5 a conclusão do trabalho e por fim as referências utilizadas como base teórica para o desenvolvimento e apêndices com alguns detalhes evidenciados do projeto.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão apresentados e revisados princípios e aplicações dos conceitos necessários para realização do presente trabalho. Em primeira análise, uma apresentação sobre o funcionamento do braço robótico. Em seguida uma apresentação detalhada de aspectos e conceitos relevantes para a aplicação de visão computacional e validação do projeto realizado, com discussão das técnicas de visão computacional relevantes a este trabalho.

2.1 Braços robóticos

Essa secção foi subdividida em dois tópicos principais: o primeiro referente ao funcionamento do braço robótico NS 16 1.65, especificando acessórios e comando principais para o funcionamento, e o segundo trazendo algumas tecnologias e aplicações desenvolvidas.

2.1.1 Funcionamento do braço robótico NS 16 1.65

O braço robótico NS 16 1.65 como citado anteriormente funciona essencialmente devido à existência e integração à unidade de controle, equipamento criado especificamente para o controle do braço. Essa central é a responsável por toda a administração do sistema, potência e periféricos e é a partir dela que se consegue armazenar e programar *scripts* de movimentações (COMAU, 2016).

Como a maioria dos braços robóticos, o NS 16 1.65 funciona nos moldes de um manipulador, formado por corpos rígidos em série unidos por articulações, formando uma cadeia cinemática aberta (CABRAL, 2010).

Para movimentar-se tanto a cinemática direta quanto a inversa podem ser aplicadas neste tipo de robô. Apesar disso, a parametrização direta às angulações dos eixos não é utilizada nesse trabalho, excluindo assim a aplicação por cinemática direta. Desta forma, a partir de cinemática inversa são fornecidos posições e velocidades de um ponto do corpo rígido do braço, normalmente sua extremidade atuadora (efetuador), obtendo as posições e velocidades correspondentes necessárias para os eixos articulados (CABRAL, 2010).

Toda movimentação por cinemática inversa tem por objetivo, portanto, calcular os valores angulares θ_1, θ_2, … , θ_n de posição dos eixos motores, estabelecendo sistemas de equações com esses outros parâmetros referentes às posições finais. Normalmente, esse sistema de equações são não-lineares, podendo ter múltiplas soluções ou mesmo não possuir soluções (MACHARET, 2010).

Por ser um sistema com 6 graus de liberdade o efetuador consegue alcançar posições e orientações arbitrárias no espaço 3D, atribuição limitada em sistemas com menos que essa quantidade de graus de liberdade disponíveis (MACHARET, 2010).

Tratando-se de um equipamento desenvolvido para aplicações industriais, todo o cálculo da cinemática inversa para atuação dos motores de robôs como o NS 16 1.65 já é realizado por completo dentro da central de controle do robô de forma transparente. Assim, basta fornecer ao sistema a posição desejada final dentro do alcance permitido para o efetuador que a central de comando calcula as possíveis soluções e efetua o movimento utilizando uma das maneiras de resolver a trajetória.

Considerando que os parâmetros desejados de posição resultem em um sistema de equações sem solução, um erro é gerado no robô informando a impossibilidade da ação. Normalmente isso ocorre em casos de extrapolação do espaço alcançável ou restrições por sobreposição de corpos rígidos da própria cadeia cinemática e por regiões de colisões com o a superfície do piso.

Dessa forma, para que o usuário consiga enviar comandos de movimentação ou posição desejada ao robô, o terminal de programação, TP, é o mais indicado. Além disso, seguindo a ideia do *Open Concept* da COMAU, conexões USB e Ethernet podem ser utilizadas para o desenvolvimento de aplicações que possibilitem envio de comandos de outras formas.

O TP é a interface mais usual de operação, seu uso está relacionado ao controle manual dos movimentos do robô, a programação *in loco* de *scripts* na linguagem *PDL2*, a execução desses *scripts* comando por comando e sua edição. Tal operação é ainda mais facilitada devido à presença de LEDs de alerta, botões de emergência, display e botões de navegação e controle. Esses itens permitem ao usuário a identificação imediata da posição da ponta ferramenta no display do TP, em termo de coordenadas do sistema referencial escolhido. O TP usado no braço que usamos neste trabalho pode ser visto na Figura 04.

O TP conta com uma chave seletora que define o modo de funcionamento da unidade de controle entre Programação, Automático ou Remoto. No modo de programação, o usuário consegue ter um maior controle das ações, realizar manobras de determinação de ferramenta, escrever e executar códigos de programação, além de determinar posições e fazer o braço mover-se automaticamente calculando a melhor trajetória possível para movimentação. Esse modo permite ao usuário atuar de maneira direta, movendo eixos ou de maneira indireta,

executando *scripts* de movimentos predefinidos. Neste modo, a velocidade é reduzida (com máxima de 250mm/s).

O modo automático, por sua vez, ao executar ciclos de programas bem definidos, normalmente salvos na memória da central de comando, movimenta-se à velocidade plena, seguindo as sequências dos *scripts*.

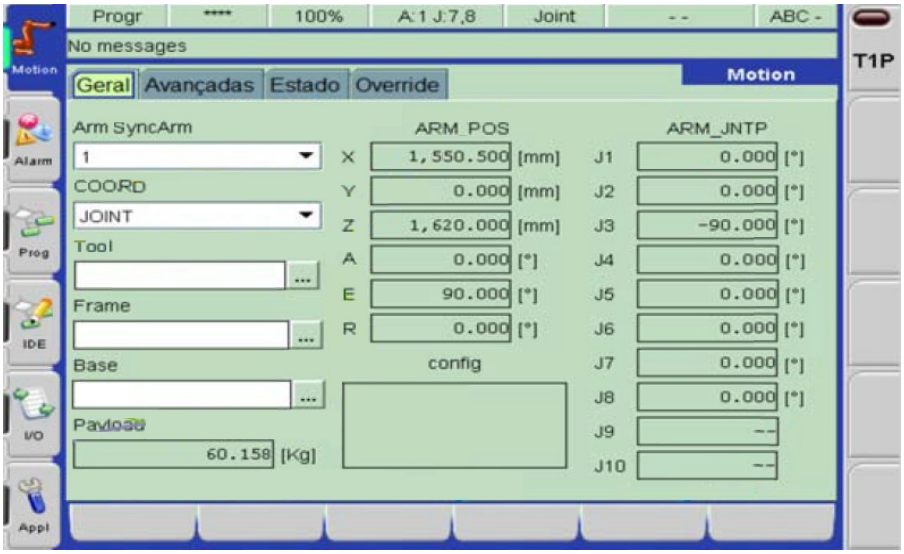
O modo remoto permite ao operador o domínio dos códigos de maneira remota, por meio de um computador ou CLP exterior ao braço.

Figura 04 - TP (terminal de programação) do braço robótico COMAU NS 16 1.65.

Fonte: COMAU, 2016.

A tela do TP, que funciona como IHM para esse robô, mostra tudo o que está acontecendo com o robô no momento. Na Figura 05, as informações relacionadas ao sistema de referência (*COORD*), ferramenta utilizada (*Tool*), coordenadas da posição atual (*ARM POS*), espaço de trabalho (*Frame*), entre outros podem ser observadas em *print* da tela do TP.

Figura 05 - Coordenadas de posicionamento do braço robótico, exibido pelo TP.

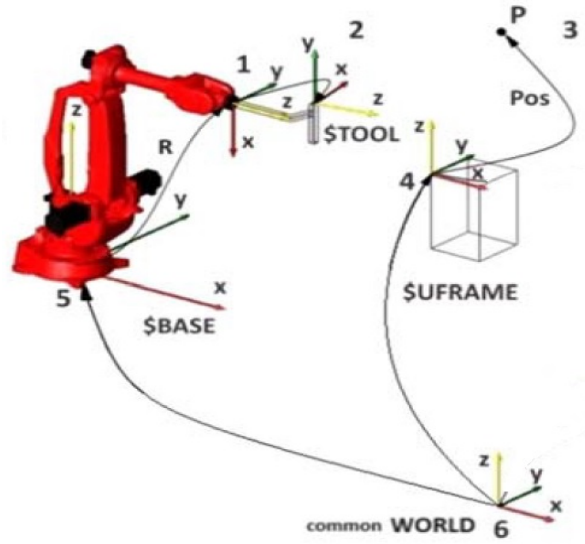


Fonte: COMAU, 2016.

2.1.1.1 Sistemas de referência

Os sistemas de referência do braço baseiam-se em coordenadas cartesianas ângulos dos eixos do robô. Os pontos cartesianos desses sistemas são usados para guiar as ações de movimentação. Existem três tipos de sistemas de referência principais: *Base Frame* ($BASE), *Tool Frame* ($TOOL) e *User Frame* ($UFRAME). Assim, para cada movimento que se deseja fazer é importante determinar a que sistema de referência as coordenadas de ação estão referenciadas (COMAU, 2013).

Figura 06 - Sistemas de referência.



Fonte: Adaptado de COMAU, 2013.

Como pode ser observado na Figura 06, o sistema *Base Frame* tem como referência os eixos X, Y e Z atribuídos à base do braço robótico; o sistema *Tool Frame* toma por base os eixos cartesianos atribuídos à ponta ferramenta; e por último o sistema *User Frame*, que pode estar definido em qualquer ponto de alcance do braço robótico, sendo portanto uma definição escolhida pelo usuário do braço (CARDOSO, 2018).

Este último sistema (*User Frame*) é de grande interesse em trabalhos onde os planos de trabalho não coincidem com nenhum dos dois outros sistemas de referência ou nos casos onde se queira maiores praticidades para definição de movimentos em planos de trabalhos deslocados da base de origem, por exemplo.

2.1.1.2 Principais variáveis e comandos

A linguagem utilizada na programação de *scripts* para o braço robótico é a PDL2, conforme citado anteriormente. Essa linguagem permite definições de variáveis e solicitação de comandos de forma bastante simples de modo a se adequar a possibilidades de movimentação do braço robótico.

As variáveis de movimentação normalmente utilizadas para o armazenamento das grandezas espaciais são denominadas: POSITION, XTNDPOS e JOINTPOS. A POSITION, talvez a mais interessante entre elas, descreve posições espaciais em termos do próprio sistema de coordenadas definido, isto é, 6 coordenadas: X, Y e Z definindo o ponto do espaço e os ângulos A, E e R definindo a rotação em graus para orientação, formando assim o sistema com as seis coordenadas (X, Y, Z, A, E, R). As variáveis do tipo XTNDPOS caracterizam-se por ser uma versão extendida do tipo de variável POSITION, podendo ter, por exemplo, parâmetros da ferramenta acoplada ao braço.

A variável JOINTPOS representa o estado do sistema a partir das posições angulares em graus de cada junta mecânica. Associando esta variável ao tipo de cinemática de mecanismos, fica claro que a sua utilização corresponde a um tipo de cinemática direta, em que se pede uma angulação dos eixos e de imediato os eixos motores do robô obedecem à solicitação sem a necessidade de cálculos por parte da central de comandos.

Ainda considerando a cinemática dos robôs, as variáveis do tipo POSITION atuam diferente das do tipo JOINTPOS, isso porque elas estão associadas à utilização da cinemática inversa, em que ao se solicitar que a ponta ferramenta do braço robótico seja movimentada

para uma posição (POSITION) bem definida no sistema de coordenadas cartesianas estipulado, a central de comandos calcula o ângulo das articulações do robô de maneira que o ponto da posição final seja alcançado.

Com relação aos comandos de movimentação, o mais comum é o MOVE TO, onde pode ser indicado para onde ir a ponta ferramenta do braço robótico ou como devem se adequar suas juntas motoras. Esse comando pode estar associado também à forma como o braço irá para a posição desejada. Por exemplo, em casos de desejo por movimentos lineares simplesmente escreve-se MOVE LINEAR TO [POSITION].

Os *scripts* em PDL2 seguem um padrão que envolve: definição do programa (PROGRAM), declaração de constantes (CONST) e variáveis (VAR), definição de rotinas de trabalho ou manipulações (ROUTINE) e em seguida a definição do *script* principal - *main* - inicializado por BEGIN CYCLE.

2.1.2 Tecnologias desenvolvidas e aplicações em indústrias

O desenvolvimento de braços robóticos por empresas e trabalhos acadêmicos vem crescendo com o avanço das tecnologias. Protótipos criados em impressoras 3D, sistemas embarcados e servo motores têm dominado aulas, pesquisas e projetos da área da robótica. Além disso, o desejo das empresas de automatizar seus processos torna-se cada vez maior para se adequar à revolução das indústrias 4.0.

A COMAU, uma das grandes fabricantes de braços robóticos, desenvolve diferentes linhas de série, como é o caso da linha NS 16 1.65. Em projeto as definições dos parâmetros físicos e digitais normalmente variam com a carga pretendida para ser suportada, o alcance máximo do braço e sua repetibilidade. De um modo geral, cada tipo de braço é projetado pensando também no tipo de indústria e em sua aplicação em campo.

As aplicações são inúmeras: mensuração e testes, corte de plasma e jato de água, polimento, prensa e dobra, usinagem de madeira e vidro, solda de ponto e arco, paletização, montagem, selagem cosmética, distribuição, fundição, manuseio de embalagens e aplicações com sistemas de visão.

A aplicação de sistemas de visão aos braços robóticos é viável segundo o *Open Concept* criado pelos fabricantes desses tipos de braço. Em equipamentos criados seguindo esse conceito, a unidade de controle permite a computadores externos acesso simplificado para controle e integração com sensores externos, como é o caso das câmeras dos sistemas

de visão. Essa abertura permite a inovação das estratégias de movimentação para o braço (COMAU, 2016).

Além de aplicações mais robustas desenvolvidas pelos próprios fabricantes do braço robótico, inúmeros sistemas têm sido integrados a braços robóticos pelo mundo, exemplo disso são observados em projetos de manipulação de corpos de prova, a partir da visão computacional conforme desenvolvido em Correia (2013).

Encontram-se exemplos desta aplicação também em trabalhos como o de Oliveira e Rodrigues (s.d.), cujo objetivo é desenvolver o protótipo de um braço robótico para funcionamento unicamente por meio das coordenadas de um sistema de visão computacional.

2.2 Visão computacional

Essa secção foi subdividida em dois tópicos principais: o primeiro caracterizando e descrevendo conceitos referente a visão computacional, importantes para o processamento das imagens, e o segundo tópico trazendo as etapas possíveis de um processamento.

2.2.1 Caracterização e conceitos

Esse tópico aborda conceitos iniciais, porém essenciais para análise e manipulação de imagens digitais. O tópico está subdividido em: imagem digital, representação de imagem, modelos de cores e a inicialização dos conceitos de processamento digital de imagem no campo da visão computacional.

2.2.1.1 Imagem digital

É caracterizada como imagem digital, toda imagem constituída de elementos discretos da figura chamados de *pixels*. O *pixel* representa o menor ponto de formação de uma imagem; para cada um deles, portanto, é atribuída uma cor que pode variar dentre o infinito universo das paletas de cores (SCHOWENGERDT, 2000).

A atribuição de cores aos *pixels* depende da profundidade de cor da imagem digital, isto é a quantidade de *bits* por *pixel* (*bpp*) que a imagem possui. Quanto maior for este parâmetro, maior é a quantidade de cores disponíveis, vide Tabela 1.

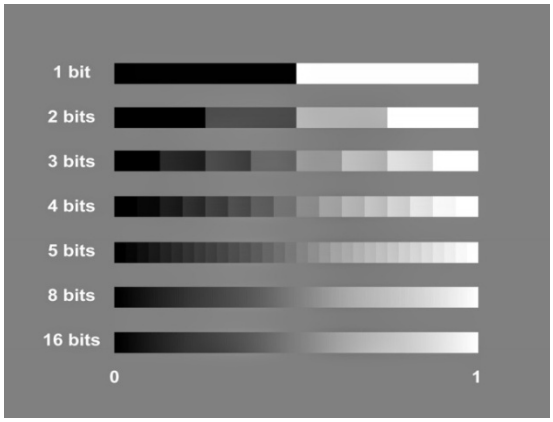
Tabela 01 - Padrões gráficos com respectivas quantidades de cores e de bits por pixel.

Profundidade de cor (nº de *bits)*	Nº de cores produzidas	Qualidade de cor	Padrão gráfico
1	$2^1 = 2$	Preto e Branco	Monocromática
2	$2^2 = 4$	Cores de 2 *bits*	CGA (*Color Graphics Adapter*)
4	$2^4 = 16$	Cores de 4 *bits*	EGA (*Enhanced Graphics Adapter*)
8	$2^8 = 256$	Cores de 8 *bits*	VGA (*Video Graphics Adapter*)
16	$2^{16} = 65.536$	Cores de 16 *bits (High color)*	XGA (*Extended Graphics Array*)
24	$2^{24} = 16.777.216$	Cores de 24 *bits (True color)*	SVGA (SuperVGA)

Fonte: Adaptado de: http://alexgomes0515.blogspot.com/2017/11/modelos-de-cor.html.

Dessa forma, para cada universo de cor produzido pelo respectivo número de *bpp* pode-se detalhar que existem para cada uma das cores produzida (matiz) um valor equivalente (tonalidade). Essa tonalidade, ou valor equivalente, costuma ser representada em escalas de cinza, ignorando as variações de brilho, podendo ser melhor observada na Figura 07.

Figura 07 - Tonalidades por bits.

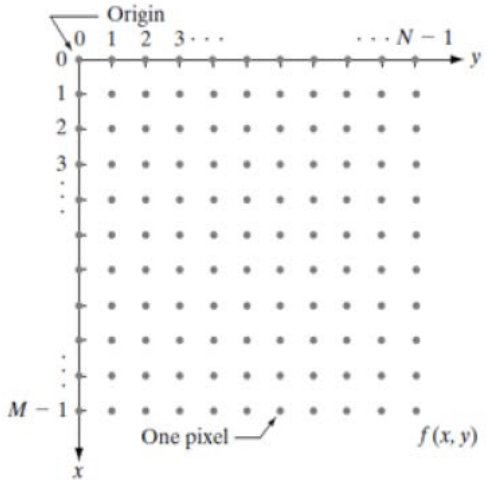


Fonte: Página da Focus, Escola de Fotografia. Disponível em: https://focusfoto.com.br/profundidade-de-bit-e-tonalidade/. Acesso em: 05 de janeiro 2019.

2.2.1.2 Representação de imagem

A representação de uma imagem real em imagem digital parte do princípio da discretização da imagem em uma matriz de *pixels*. Assume-se que essa matriz, por sua vez, deve possuir um número limitado de linhas M e um número limitado de colunas N. Dessa forma consegue-se ter de forma discreta uma matriz como a representada na Figura 08. A representação seguindo essa matriz pode ser feita a partir da convenção de eixos cartesianos definidos em sua primeira posição.

Figura 08 - Referência para sistema cartesiano em imagem digital.



Fonte: GONZALEZ e WOODS, 2002.

Assim, cada ponto (x,y) representa uma posição na imagem, e para cada posição um resultado da função básica da matriz f(x,y) é retornado. A natureza básica da função f(x,y) é uma composição da quantidade de luz incidente na cena (iluminação), definida por i(x,y) e do quanto de luz é refletida naquele ponto (refletância), definida por r(x,y) (GONZALEZ e WOODS, 2002).

Essa definição matemática de uma imagem, através de funções, tem sua importância entendida quando existe algum intuito com a imagem, seja extração de informação ou manipulação da imagem. Assim, quando necessários trabalhos com imagem as mesmas são utilizadas como matrizes e manipuladas por transformadas matriciais que impactam na imagem final.

Para cada manipulação é aplicado algum tipo de técnica (em geral, transformadas) que gere um resultado na imagem. Para imagens coloridas se faz necessário conhecer um pouco sobre os modelos de cores que podem ser adotados para representação e análise dos *pixels*.

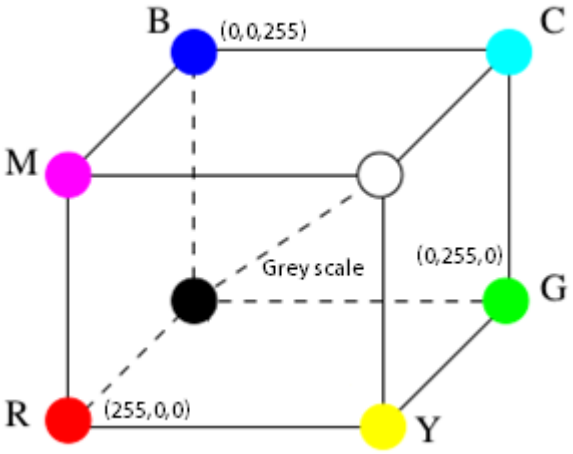
2.2.1.3 Modelos de cores

Para facilitar a especificação de cores e transformações de imagens em frentes de trabalho diferentes e condições particulares, modelos de cores foram desenvolvidos com diversas parametrizações, cada qual com seu modelo de coordenadas (KELDA, 2014).

Os modelos de cores mais usuais são: *RGB*, *CMY*, *HSV*, *HSB*, *HSL*, *YUV* e *HSI*. Aqui serão detalhados dois dos modelos de cores mais utilizados em projetos de visão computacional: *RGB* e *HSV*.

O modelo de cor *RGB* (*red*, *green*, *blue* – vermelho, verde, azul), refere-se ao espaço de cor formado pela sensação da soma ponderada dos componentes *red* (R), *green* (G) e *blue* (B) (MARTINS, 2010). Esses componentes variam de 0 a um máximo de 255 (para 8 bits por componente), o que corresponde à intensidade das três cores primárias (vermelho, verde e azul) usadas nas telas de dispositivos eletrônicos (TVs, computadores, celulares e projetores). O esquema RGB pode ser visualizado como um espaço cartesiano, onde cada componente é um eixo, como pode ser observado na Figura 09 (GONZALEZ e WOODS, 2002).

Figura 09 - Coordenadas cartesianas *RGB*.



Fonte: Adaptado de MARTINS, 2010.

Neste sistema, cada combinação das três coordenadas resulta em um ponto de cor único. Uma particularidade notável consiste do resultado da igualdade das três coordenadas cartesianas: uma cor em escala de cinza, diagonal do cubo da Figura 09 (CATTIN, 2016).

Uma propriedade prática deste modelo de cor caracteriza-se pela invariância da cor normalizada para mudanças de orientação da superfície de objetos (SKARBEK e KOSCHAN, 1994). Apesar dessa vantagem, uma restrição é observada quando leva-se em consideração

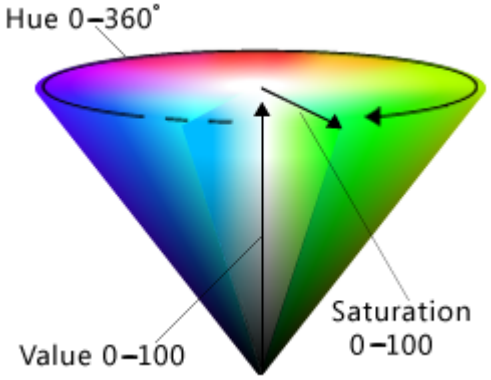
variações de iluminação do ambiente: normalmente até mesmo pequenas variações podem resultar em variações significativas nas coordenadas de cores, tornando-a não indicada para ambientes com variação de iluminação (KELDA, 2014).

Outro espaço de cor usualmente utilizado é o *HSV* (*hue*, *saturation*, *value* – tonalidade ou matiz, saturação e valor ou brilho), normalmente representado por coordenadas cilíndricas dos parâmetros que o nomeiam. Tais parâmetros são melhor definidos a seguir:

- *Hue* (tonalidade ou matiz): define a cor pura, variando de 0 a 360º.
- *Saturation* (saturação): define a vibração ou medida do grau de diluição de uma cor pura pela luz branca, variando de 0 a 100%.
- *Value* (valor ou brilho): define o brilho de uma cor, variando de 0 a 100% (CATTIN, 2016).

A representação desse modelo de cor rearranja a geometria do *RGB* na tentativa de ser mais intuitiva e percentualmente relevante do que a representação cartesiana (cubo) (KELDA, 2014). A sua representação trata-se de um cone de base circular onde a altura é o *value*, a distância do eixo central é o *saturation* e o ângulo é o *hue*, tal como mostra a Figura 10.

Figura 10 - Esquema de representação do modelo de cor *HSV*.



Fonte: Pagina da Microsoft. Disponível em: https://msdn.microsoft.com/en-us/library/windows/desktop/dn742482(v=vs.85).aspx

Acesso em: 02 julho 2017.

Segundo Shuhua e Gaizhi (2010), em projetos reais o *HSV* deve ser o modelo de cor utilizado porque as descrições em termos de tonalidade, saturação e matiz são frequentemente mais relevantes, facilitando a discriminação das superfícies. Como os componentes R, G e B da cor estão todos correlacionados com a quantidade de luz que a atinge uma superfície, a

análise de imagem utilizando RGB dificulta a discriminação das cores onde há variação de iluminação no ambiente, tornando o modelo *HSV* preferido para a aplicação.

2.2.1.4 Processamento digital de imagens

O trabalho com imagens digitais, portanto, pode significar a manipulação dentro de uma imagem discreta de milhares de *pixels* por imagem. Isso pode parecer muito para uma primeira análise, mas ao longo dos anos foi sendo facilitado e contornado por meio das técnicas de manuseio com imagens digitais que surgiram e ainda surgem nos dias atuais (PRATT, 2007).

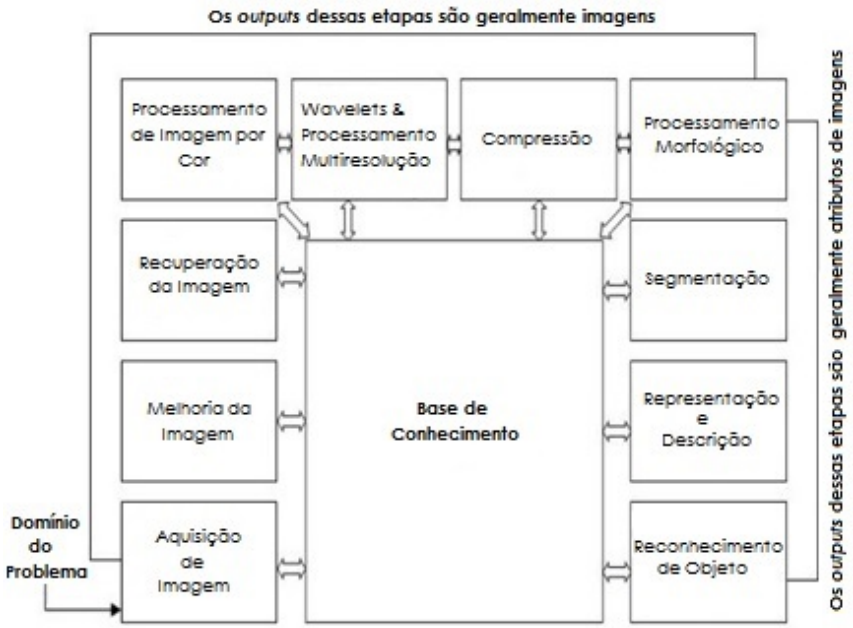
Assim, conhecendo a mínima partícula definidora de uma imagem, consegue-se facilmente entender os tipos e técnicas de manipulação de imagens, a partir de contrastes, filtros espaciais, supressão de ruídos, manipulação de geometrias e identificação de cores (SCHOWENGERDT, 2000). Essa interpretação e/ou manipulação de imagens digitais por meio da associação dessas diversas técnicas recebe o nome de processamento digital de imagens.

2.2.2 Etapas do processamento digital de imagens

O processamento digital de imagens é normalmente guiado por metodologias desenvolvidas e/ou em desenvolvimento que seguem uma sequência de manipulações da imagem dinâmica de acordo com o propósito do processamento. As metodologias aplicadas ao longo das etapas podem conduzir para resultados em que tanto a entrada quanto a saída do sistema sejam imagens, como podem também gerar informações e/ou atributos extraídos da imagem de entrada (GONZALEZ e WOODS, 2002).

O fluxograma apresentado na Figura 11 mostra de forma sintética os processos que podem ser aplicados a uma determinada imagem de entrada a fim de obter informações ou imagens processadas de saída, não sendo exaustivo até porque outras técnicas vão sendo desenvolvidas a cada momento.

Figura 11 - Processos possíveis de aplicação para o processamento de imagem digital.



Fonte: Adaptado de GONZALEZ e WOODS, 2002.

Serão descritos a seguir alguns desses processos, apresentando técnicas ou metodologias importantes em suas aplicações. Apenas os considerados de maior relevância para o projeto serão citados, entretanto isso não exclui a importância dos não citados em outras aplicações.

2.2.2.1 Aquisição de imagem

Este processo corresponde normalmente à etapa inicial de qualquer sistema de processamento de imagem. A captura, o armazenamento e a transmissão da imagem comumente são iniciados com a conversão de um campo de imagem contínua para sua forma equivalente em âmbito digital (PRATT, 2007).

O processo de transmissão caracteriza-se pela etapa de envio de imagens do sensor de imagem para o *hardware* responsável pela visão computacional. Normalmente isso é feito de forma contínua nos chamados *frames* aguardando a captura para sua interrupção.

O processo de captura leva em conta uma série de fatores, desde o tipo de sensor para a aquisição (ultrassom, ressonância magnética, câmera digital), a quantidade de sensores do sistema, seu posicionamento diante ao campo que se quer observar, a resolução da captura e a iluminação do ambiente em análise (para sensores luminosos). Todos esses fatores são parâmetros de projeto e devem ser escolhidos para melhor atender às aplicações específicas (JUNIOR, 2010).

Normalmente a imagem a ser analisada pode estar em algum formato padrão, tais como **.jpg, *.png, *.bmp*, etc, ou estar apenas armazenada na memória dinâmica do sistema, em forma de uma variável e compactada aos moldes de uma matriz de *pixels*.

Um ponto importante da aquisição de imagens é a sua calibração. Garantir uma boa perspectiva para a imagem digital, quando comparada à imagem real, ajuda a obter medições mais confiáveis do ambiente real reproduzido em imagem digital.

2.2.2.2 Recuperação de Imagem

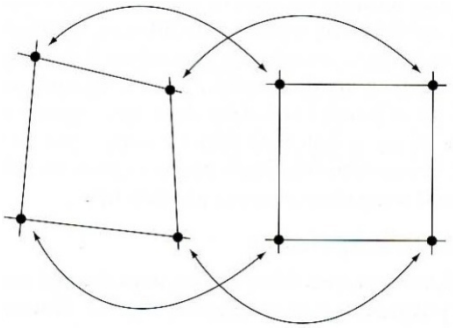
O processo de recuperação de imagem tenta reconstruir ou recuperar uma imagem degradada como tentativa de torná-la apta para novos processos e análises. Trata-se de um pré-processamento com intuito de melhorar a eficiência dos processos seguintes (GONZALEZ e WOODS, 2002).

Dentre as inúmeras abordagens, a transformação espacial ou transformação de perspectiva é uma das técnicas mais utilizadas para correções da perspectiva das imagens distorcidas. O princípio de funcionamento desta técnica esta relacionado à recriação de projeções paralelas da imagem de entrada em outro plano (KIRIAN e MURALI, 2013).

Esta técnica, também chamada de homografia, quando utiliza o plano homográfico, atua computando as posições relativas de quatro pontos da imagem distorcida de entrada para recriar essas posições relativas em outro plano, neste caso o plano homográfico (KIRIAN e MURALI, 2013).

Na Figura 12, pode-se observar a associação de pontos entre dois segmentos de imagens de perspectivas diferentes. Normalmente, essa correspondência é utilizada na conversão de imagens com algum tipo de profundidade para o plano da tela.

Figura 12 - Correspondência de pontos em dois segmentos de imagem.



Fonte: GONZALEZ e WOODS, 2002.

2.2.2.3 Processamento de cor da imagem

A utilização de processamento de imagem por cor é normalmente motivada pela simplicidade de descrição das cenas, facilitando a identificação e extração de objetos em uma imagem. O processamento por cor só é possível devido à identificação computacional quase que imediata das cores que cada pixel exibe, a partir de um modelo de cor definido (GONZALEZ e WOODS, 2002).

Através desta técnica consegue-se - possivelmente associada a outra técnica - detectar e classificar com sucesso objetos de diferentes cores. Na Figura 13 observa-se uma aplicação exemplo deste processo, com reconhecimento de cores em uma imagem estática de esferas coloridas.

Figura 13 - Reconhecimento digital por cor.

Fonte: Página da *Pyimagesearch*. Disponível em: http://www.pyimagesearch.com/2014/08/04/opencv-python-color-detection/. Acesso em: 20 junho 2017.

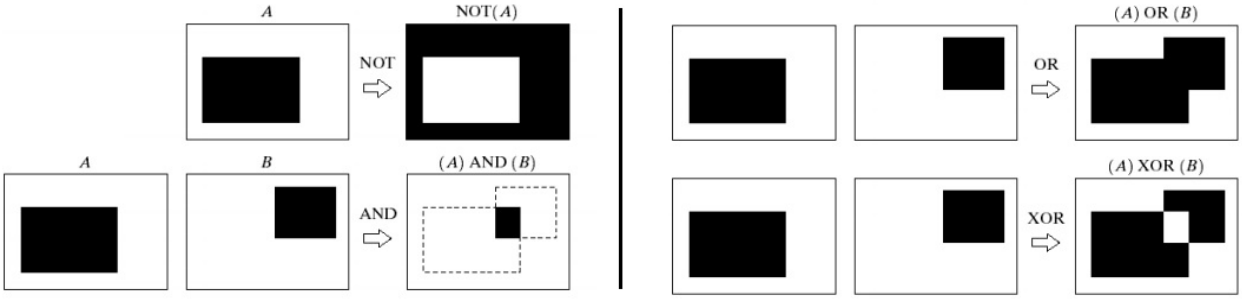
2.2.2.4 Processamento de morfologia

O processamento de morfologia denota na visão computacional a utilização das técnicas envolvidas na morfologia matemática como uma ferramenta para extração ou descrição de componentes das imagens ou regiões de interesse (GONZALEZ e WOODS, 2002).

Usado para extração dos componentes úteis da imagem para posterior análise, o processamento por morfologia pode ser observado de forma aplicada em casos como a extração de contornos, desbaste da imagem (erosão) ou dilatação, operações lógicas diversas, entre outros (RHODY, 2005).

Um dos princípios de funcionamento desse processo como um todo são as operações lógicas entre imagens binárias. Operações conhecidas do tipo AND, OR, XOR, entre outras são presentes também quando se trata de *pixel* de imagens. A Figura14 a seguir ilustra essas operações.

Figura 14 - Operações lógicas básicas entre imagens.



Fonte: Adaptado de GONZALEZ e WOODS, 2002.

Este tipo de operação é aplicada normalmente em manipulações entre imagens para obtenção por segregação de determinadas regiões de interesse.

Os processos de erosão e dilatação por sua vez, não visam obtenção desse tipo de segregação. A sua aplicação tem por objetivo maior a manipulação das imagens para obtenção de melhores, ou mais úteis regiões da imagem original.

Esses processos conseguem diminuir partículas, eliminam componentes menores que o elemento estruturante, aumentam espaços e permitem a separação de componentes conectados, no caso da erosão. E de forma inversa, aumentam partículas, preenchem espaços e conectam componentes próximos no caso da dilatação (MAINTZ, 2005).

2.2.2.5 Segmentação

A segmentação que pode ser entendida como manipulação de contraste é uma transformação radiométrica de *pixel* por *pixel* que é desenvolvida para discriminar e realçar visualmente as regiões de imagens de baixos contrastes. Nesse caso, cada nível de cinza dos *pixel*s é mudado pela transformação especificada sem levar em conta os *pixels* da vizinhança (SCHOWENGERDT, 2000). É em termos gerais uma técnica de rápida aplicação.

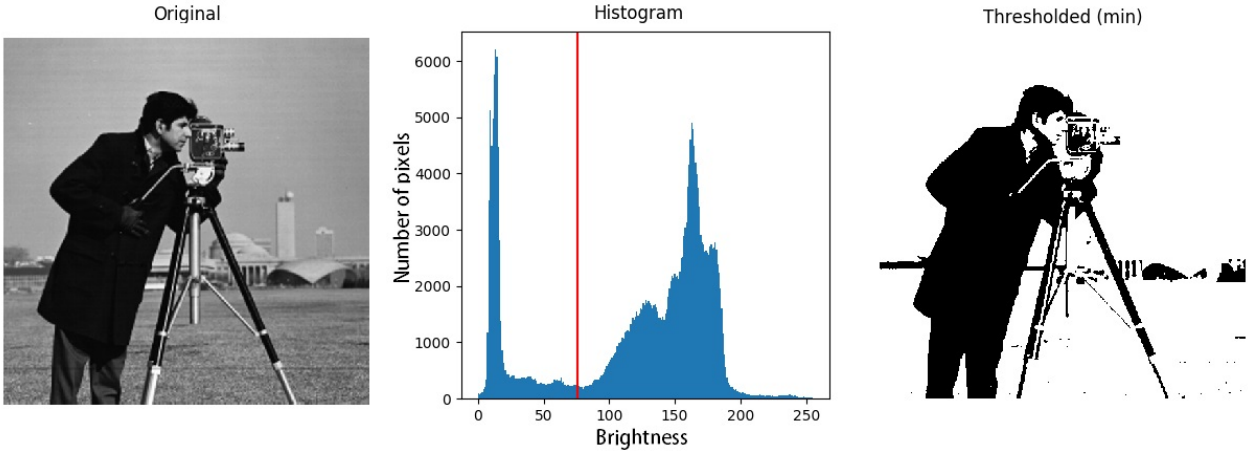
Essa técnica possui assim como as outras subdivisões de possíveis procedimentos, que vão desde histograma de níveis de cinza (definir níveis de cinza), contraste de realce e o *thresholding* (limiarização). Esse último é o mais comumente utilizado e o qual este trabalho tomará como base para início de desenvolvimento.

Thresholding ou binarização é um tipo de manipulação que transforma cada *pixel* da imagem em um binário, em que só se tem o preto ou branco. *Pixels* de tonalidade mais escura

são transformados em *pixels* pretos e *pixels* de tonalidade mais clara são transformados em *pixels* brancos (SCHOWENGERDT, 2000).

Dessa forma, estipulando um limite como critério de seleção do que será preto e do que será branco, consegue-se facilmente segmentar uma imagem e captar objetos em espaços que possuam bons contrastes de imagem, tal como exemplifica a Figura 15.

Figura 15 - Segmentação por *thresholding* e histograma de avaliação. Esquerda: imagem original, centro: histograma mostrando o nível escolhido para binarização, direita: imagem binarizada.



Fonte: Adaptado da página da *Scikit-image*. Disponível em: https://scikit-image.org/docs/0.13.x/auto_examples/xx_applications/plot_thresholding.html. Acesso em: 15 de Maio 2019.

Como pode ser observado no centro da Figura 15, é uma forma de representar a quantidade de *pixels* observados na imagem correspondente a cada brilho ou tonalidade em níveis de cinza, de 0 a 255. Na imagem, a reta de vermelho (na posição T=150) representa o limite de limiarização definidor do que será tomado como preto (a esquerda do limite) e como branco (a direita do limite).

2.2.2.6 Reconhecimento de objeto

Em visão computacional, a etapa de reconhecimento de objetos pode ser entendida pelo processo de classificação e obtenção de localização de objetos em imagens digitais (AHUJA e TULI, 2013). Em sua maioria, para essa identificação o uso de padrões de imagens são utilizados; sendo assim, para iniciar esse processo, as ideias de *pattern* (padrão), *pattern class* (classe de padrão) e *features* (características) precisam ser entendidas.

As características de um objeto ou algo a ser detectado são chamadas no âmbito da visão computacional de *features* ou *descriptors*. Essas características normalmente são

escolhidas pelo desenvolvedor para melhor representar/classificar um objeto a ser detectado. A combinação de várias características resulta em um padrão (*pattern*); uma família desses padrões é chamada de classe de padrões (*pattern class*) (GONZALEZ e WOODS, 2002).

Como citado anteriormente, as características definidoras de um padrão podem ser qualquer particularidade do objeto a ser detectado. Isso significa que tanto características quantitativas (como largura e altura dos objetos), como características estruturais (abordagem comumente utilizada para as impressões digitais) podem ser utilizadas na classificação da imagem. A escolha dessas características, portanto, define a abordagem que será dada ao sistema de visão computacional (GONZALEZ e WOODS, 2002).

As abordagens resultantes das escolhas dos padrões são distribuídas em dois diferentes modelos de método: os métodos teóricos de decisão e o método estrutural. Como o método estrutural não tem serventia para este trabalho, apenas os métodos teórico de decisão são detalhados a seguir.

A abordagem teórica de decisão caracteriza-se pela utilização das características não estruturais, normalmente quantitativas das imagens. Os mais conhecidos métodos teóricos de decisão são as abordagens por '*matching'*, classificadores estatísticos ótimos e as redes neurais. Esses métodos não possuem restrições definitivas de escolha para aplicação, eles simplesmente podem ser escolhidos em projeto de acordo com o grau de precisão e/ou recursos disponíveis para o desenvolvimento do sistema de visão computacional. A seguir, tais técnicas podem ser melhor entendidas.

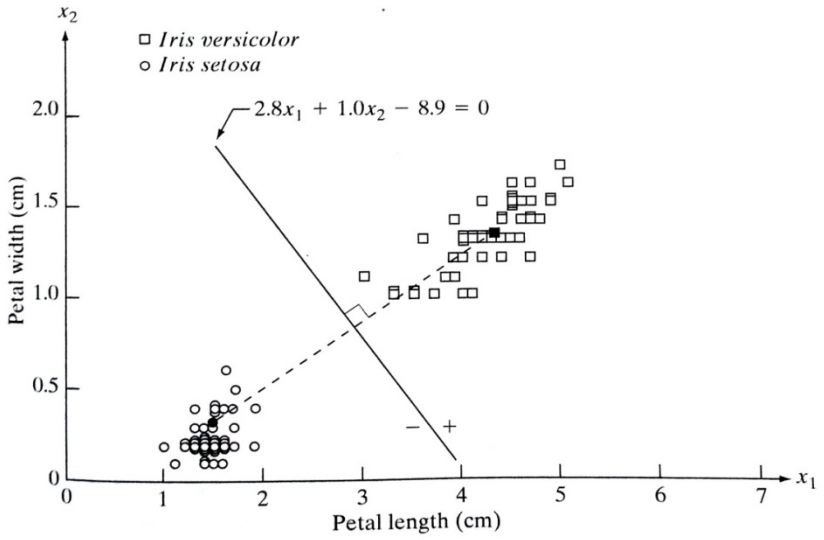
A técnica de *Matching* (correspondência) é uma das mais utilizadas técnicas na visão computacional. Ela está baseada na comparação de porções de imagens (sub-imagens) com uma imagem principal de entrada, a fim de encontrar as similaridades entre elas. Essa comparação pode estar relacionada a uma abordagem baseada na área (*template*) ou a uma abordagem baseada nas características (*features* ou *descriptors*) (MAHALAKSHMI e MUTHAIAH, 2012).

A abordagem baseada nas características (*Featured-based approach*) é melhor aplicada quando tanto as sub-imagens quando a imagem de referência possui características mais precisas e de maior controle, isso inclui pontos, curvas ou modelos de superfície que precisam ser correspondidos (MAHALAKSHMI e MUTHAIAH, 2012). No geral essa abordagem se utiliza de classificadores de distância mínima (limite de decisão) para classificar os padrões

detectados (técnica também chamada de análise de discriminante, *discriminant analysis*) (FISHER, 1936).

No Figura 16 pode ser melhor compreendido os limites estabelecidos por Fisher (1936) para a classificação das detecções de pétalas de flores, de acordo com sua largura e comprimento, em seu estudo.

Figura 16 - Limite de decisão de classificadores de distância mínima representado por reta.



Fonte: GONZALEZ e WOODS, 2002.

Como pode ser observado, as regiões do gráfico são separadas por um limite de decisão, em que a característica (*feature*) é classificada pela região em que se encontra. No exemplo anterior esse limite é dado em forma de reta.

A segunda abordagem relativa ao *Matching* é baseada em área (*Area-based approach*) e é mais conhecida como *Template Matching*. Essa abordagem é baseada em Correlações - entre imagem de referência e padrões - e é naturalmente aplicada a objetos que não possuam características (*features*) fortes o suficiente para classificar um objeto em uma imagem por meio da abordagem anterior (MAHALAKSHMI e MUTHAIAH, 2012). A seguir pode-se verificar a forma mais simples do cálculo de correlação entre uma imagem *f(x,y)* e uma sub-imagem *w(x,y)* para aplicação da técnica:

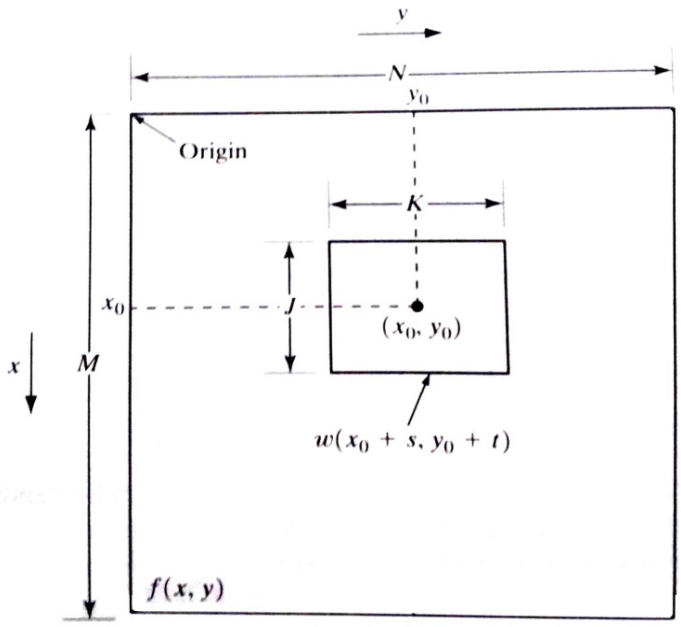
$$c(x,y)\ \sum_s\ \sum_t\ f(s,t)\ w(x+s, y+t) \qquad (1)$$

Conforme citado em tópico anterior *f(x,y)* e *w(x,y)* são as funções naturais básicas de uma imagem; *f(x,y)* relativo à imagem principal e *w(x,y)* relativo à sub-imagem padrão

(*template)*. À medida que x e y variam, a função correlação *c(x,y)* tende a encontrar as regiões similares entre elas imagens, até que ao final indique a posição (x,y) de maior correlação (AHUJA e TULI, 2013).

A ideia principal é fazer com que uma sub-imagem (*pattern* ou padrão, aqui chamadas de *template*) percorra por completo a imagem de referência, da esquerda para direita, de cima para baixo e com da origem convencionada calculando a partir da fórmula anterior a correlação para cada ponto da imagem determinado por (x, y), conforme mostrado na Figura 17 (GONZALEZ e WOODS, 2002).

Figura 17 - Representação de padrão percorrendo ponto a ponto a imagem de entrada.



Fonte: GONZALEZ e WOODS, 2002.

Nesta aplicação, inúmeras são as maneiras de escrever a equação de correlação a partir de coeficientes. As manipulações, como a normalização, por exemplo, se dão usualmente para diminuir efeitos de amplitudes entre imagem de referência e sub-imagem padrão (MAHALAKSHMI e MUTHAIAH, 2012).

Um ponto negativo desta técnica é que a correspondência para imagens ou objetos rotacionados da imagem não geram resultados positivos para o reconhecimento. Nos casos em que já se sabe que o objeto a ser identificado na imagem pode estar rotacionado é necessário fazer o *template* rotacionar também, a fim de calcular a correlação com a imagem principal também rotacionada (GONZALEZ e WOODS, 2002).

A segunda técnica relativa aos métodos teóricos de decisão são os Classificadores Estatísticos Ótimos. Como o próprio nome sugere, essa técnica de reconhecimento de objetos é baseada em resultados probabilísticos gerados a partir de classificadores sofisticados de imagem. Nessa técnica, atributos referentes à intensidade dos pixels são escolhidos para representar objetos ou regiões das imagens em um arquivo classificador (LILESAND e KIEFER, 1987).

O mais conhecido classificador supervisionado desta técnica é o classificador bayesiano. Ele se utiliza de dados prévios de treinamento para sua criação, isto é, exemplos de classe de padrões a serem reconhecidas (RIPLEY, 2008). Desta forma, um classificador funcional se utiliza de padrões como entrada a fim de retornar decisões a partir de probabilidades, quantificando as incertezas e tornando-as inferências (SANTOS, 2003).

Um exemplo de aplicação desta técnica é o classificador *Haar Cascade* desenvolvido por Viola e Jones (2001) para reconhecimento de rostos humanos. Essa aplicação tem uma grande importância no âmbito da visão computacional e tem sido intensamente replicada para outras aplicações.

De maneira geral a criação de classificadores para aplicação desta técnica necessita de um grupo de imagens positivas, isto é, imagens que representam bem o objeto que se quer rastrear, isto é, uma classe de padrões. E um grupo de imagens negativas, que podem ser imagens quaisquer, exceto imagens da classe de padrão definida. Esses grupos de imagens devem ser adquiridos em grandes quantidades (milhares), em uma média de 1 positiva para 2 negativas.

Os classificadores estatísticos ótimos podem ser sofisticados a ponto de pré-treinar um sistema a fim de realizar um reconhecimento, mas não são tão avançados a ponto de permitir que o sistema consiga se auto treinar à medida que identifica positivos e negativos numa imagem como em um aprendizado de máquina (GONZALEZ e WOODS, 2002).

O auto-treinamento de classificadores formados por classes de padrões é conseguido na última das técnicas de métodos teóricos de decisão aqui apresentadas: as Redes Neurais. Essa técnica traz aos sistemas de visão computacional modelos matemáticos inspirados nas estruturas neurais de organismos inteligentes, adquirindo conhecimento através da experiência (TEODORO, 2003).

A aplicação dessa técnica é considerada como um ramo da inteligência artificial, isso se reflete da sua capacidade de aprendizado, estabelecendo relações complexas entre diversas variáveis sem que seja imposto qualquer modelo pré-estabelecido (TEODORO, 2003). Essas variáveis no ramo da visão computacional são as características (*features*) dos padrões de imagens (*pattern*).

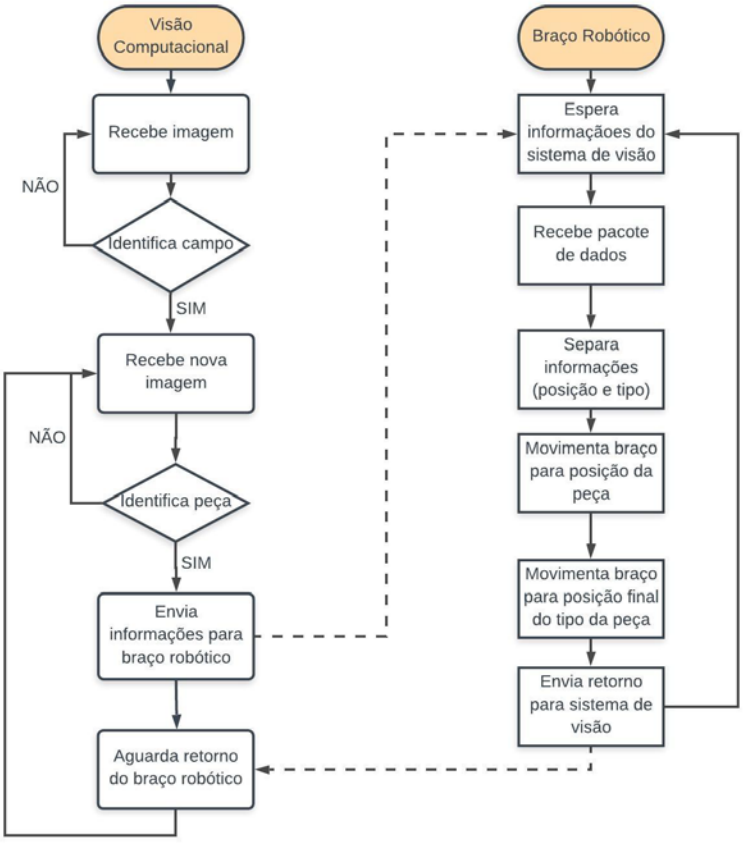
3 METODOLOGIA

Será detalhada nessa secção a metodologia seguida para elaboração do trabalho. Para isso, a secção foi dividida nos subtópicos a seguir: fluxo dos sistemas, configurações iniciais, desenvolvimento do algoritmo de visão computacional, comunicação *hardware*-braço robótico e desenvolivimento do algoritmo de rotas.

3.1 Fluxo dos sistemas

Com o intuito de cumprir com o objetivo geral, respeitando as condições de contorno e definições iniciais do sistema, foi detalhado o fluxograma das informações e ações necessárias a serem desenvolvidas para tudo correr conforme planejado. A Figura 18 apresenta de maneira macro a sequência de informações e ações do sistema:

Figura 18 - Sistemas em integração.



Fonte: Elaborado pelo Autor, 2019.

Da parte da visão computacional, o fluxo do sistema é iniciado ao receber uma imagem de entrada obtida através de uma câmera. Essa imagem é processada para identificar os pontos definidores das extremidades do campo de trabalho por meio do processamento de imagem

por cor. Os pontos obtidos deste processo definem a região de buscas por padrões nas etapas seguintes.

O fluxo entra a partir de então em um *loop* para buscas de objetos dentro do campo de trabalho. Uma imagem da situação atual do campo de trabalho é recebida novamente. Dessa vez por meio da homografia e dos pontos definidos anteriormente uma imagem delimitada, apenas a região de interesse é criada. A partir dessa imagem gerada, os cálculos de correlação por *Matching* se iniciam para localização e classificação de possíveis peças em campo. Uma vez que as peças são identificadas, o sistema envia para o braço robótico as informações de local e tipo por meio de conexão TCP/IP, via *socket*. O algoritmo de visão aguarda um retorno referente à finalização dos movimentos do braço e reinicia o ciclo recebendo uma nova imagem da situação atual do campo de trabalho.

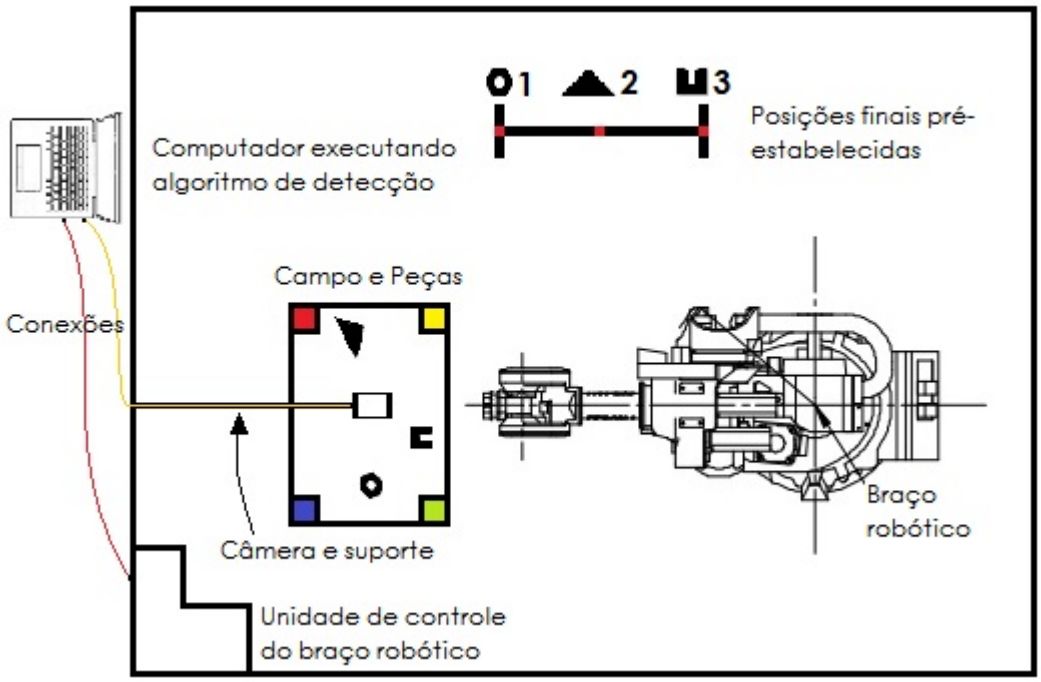
Por parte do braço robótico, nenhum movimento é iniciado até que se recebam informações do sistema de visão computacional referente à peça. Ao conectar-se com o sistema de visão, o pacote de dados recebido é organizado em variáveis do sistema, segregando a informação referente à localização da peça, da informação referente ao tipo peça. As variáveis criadas nesta separação são as responsáveis pelas etapas seguintes.

O movimento do braço se inicia em direção à localização da peça, definida pelas variáveis de localização, citadas anteriormente. Ao realizar a captura, o segundo movimento é iniciado em direção à posição pré-definida para a peça coletada, ação que só é possível com a utilização da variável de tipo de peça também recebida do processo de visão computacional.

Ao finalizar a entrega da peça em sua posição correspondente, o braço robótico envia um retorno ao sistema de visão computacional informando a finalização dos movimentos daquele ciclo, permitindo o reinício da identificação de novas peças.

Na Figura 19, adiantando um pouco a disposição dos itens no sistema, pode ser observado a vista superior do ambiente para melhor entendimento do fluxo do sistema.

Figura 19 - Vista superior da disposição dos elementos no ambiente.



Fonte: Elaborado pelo Autor, 2019.

Para o desenvolvimento dos sistemas integrados de identificação por visão computacional e execução das ações pelo braço robótico conforme apresentado na Figura 19 precisou-se seguir as macro-etapas de desenvolvimento a seguir:

- Configurações iniciais: Como início do processo de desenvolvimento do projeto, as condições de contorno para o sistema são nesta etapa configuradas e ajustadas para o melhor andamento das atividades. Tanto as características físicas de instalação, ambiente de trabalho, posições, número de objetos, tipo de objetos, fluxograma de ações, etc., quanto as características digitais de *hardwares, softwares,* pré-definições de variáveis, são estabelecidas para o problema.
- Desenvolvimento do algoritmo de visão computacional: É a etapa de estruturação, codificação e testes do sistema de visão computacional identificador de objetos e posições em espaços pré-estabelecidos. É a criação do sensor do sistema, e por isso talvez uma das mais importantes e complexas etapas do projeto.
- Desenvolvimento do algoritmo de rotas do braço robótico: Etapa em que é estruturado o comportamento do robô em todos os momentos do ciclo de funcionamento do sistema, desde sua inicialização, ações por resposta às variáveis de entrada e a finalização da sequência de movimentos. É, portanto, o planejamento de

todas as rotas para uma árvore de possibilidades pré-definidas nas condições de contorno.

- Comunicação *hardware*-braço robótico: Nesta etapa, é realizada a integração principal das duas unidades processadoras necessárias ao projeto, *hardware* independente, responsável pelo sensoriamento por visão computacional; e unidade de controle do robô, responsável pela execução dos movimentos. Essa integração é feita a partir de configurações simples de inicialização e também a partir de algoritmos em linguagens de programação distintas entre os *hardware*s, cada um com a sua.
- Integração e detalhamento dos algoritmos: Consiste da finalização da criação, ajustes e detalhes para os sistemas funcionarem de forma integrada, com facilidades e sem erros operacionais.

Neste trabalho, o foco principal se manteve na integração do sistema de visão computacional ao sistema de comandos do braço robótico. Desta forma, os programas desenvolvidos aqui consistem apenas do essencial para a correta sequência de funcionamento e respostas às entradas fornecidas.

A ordem de execução das macro-etapas citadas não necessariamente seguiu a ordem em que foram descritas aqui. Em paralelo, por exemplo, a etapa de comunicação *hardware*-braço robótico foi executada com a integração dos algoritmos e configurações iniciais.

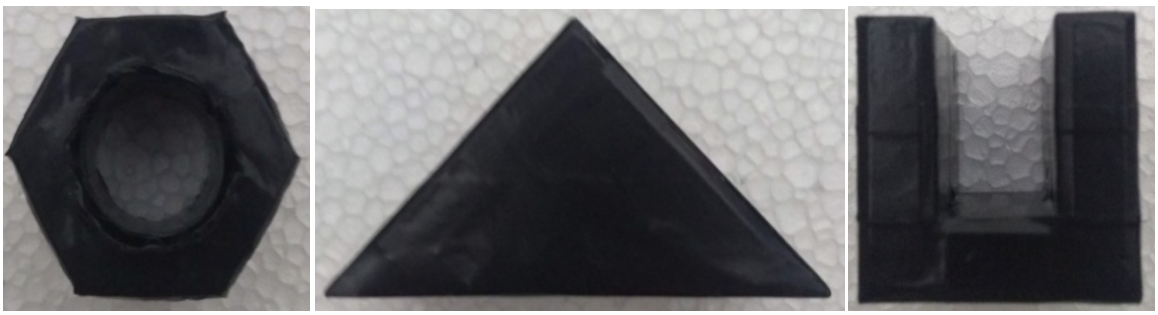
3.2 Configurações iniciais

Neste tópico são estabelecidas configurações iniciais e condições de contorno para restrição do problema proposto e organização da estrutura de trabalho. Assim, define-se a seguir as peças utilizadas, o arranjo físico do sistema, sensor de imagem, ferramenta de captura e espaço de trabalho.

3.2.1 Definição de peças

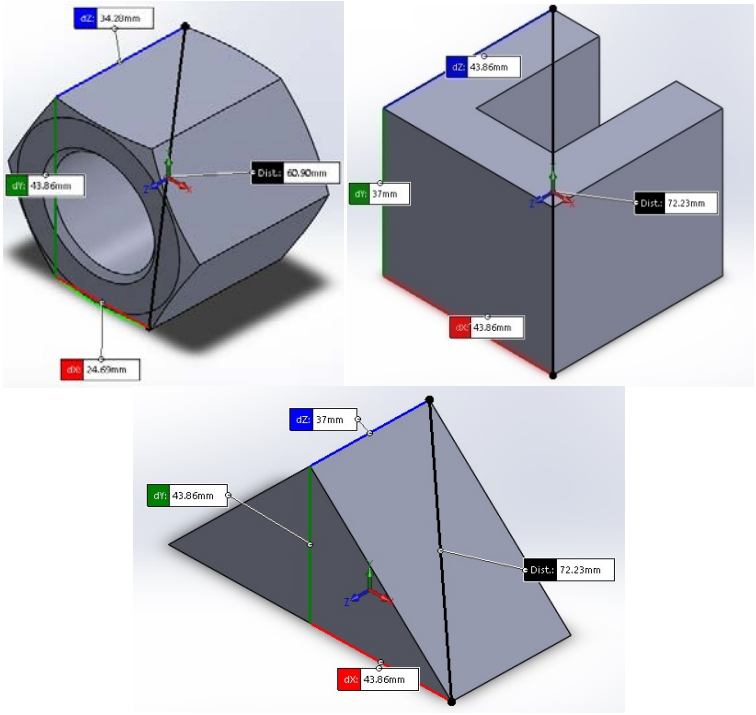
Inicialmente foram definidas, modeladas e confeccionadas em impressora 3D as peças que seriam usadas na identificação por imagem ao longo do projeto. Foram escolhidos os três modelos de objetos mostrados nas Figuras 19 e 20 a seguir:

Figura 20 - Peças escolhidas para o projeto. Esquerda: porca; Centro: prisma triangular; Direita: U.



Fonte: Elaborado pelo Autor, 2019.

Figura 21 - Modelagem em *software*.



Fonte: Elaborado pelo Autor, 2019.

As peças foram escolhidas garantindo uma diversificação de formas geométricas e uma homogeneidade de cor, de forma que não fosse utilizado no algoritmo de visão computacional o reconhecimento por cores para identificação das peças. Isso justifica o fato delas terem sido pintadas e cobertas com fita isolante de cor preta.

As dimensões escolhidas para cada peça foram ajustadas para atender a amplitude máxima e mínima da garra do braço robótico em questão, mostrada em mais detalhe na Figura 21, garantindo uma boa proporcionalidade entre elas.

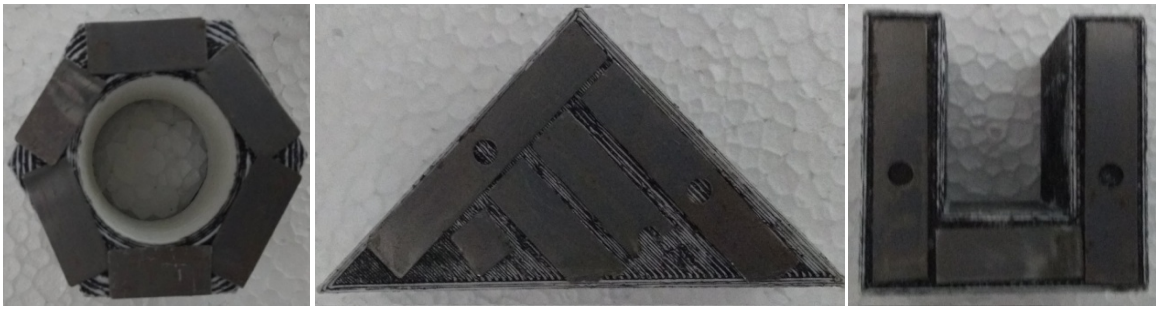
Figura 22 - Vista superior da garra do braço robótico.

Fonte: Elaborado pelo Autor, 2019.

As peças foram enumeradas para fazer com que a identificação das peças em *software*, descrita mais adiante, fosse feita por números inteiros, desta forma, as seguintes atribuições foram dadas: peça 1: porca; peça 2: prisma triangular; peça 3: U.

Nas superfícies inferiores e superiores das peças foram inseridas (no interior das fitas isolantes) placas metálicas. Essa aplicação será justificada mais adiante na seção de definição da ferramenta de trabalho. Na Figura 22 consegue-se observar essa aplicação:

Figura 23 - Placas metálicas posicionadas no interior das peças.



Fonte: Elaborado pelo Autor, 2019.

3.2.2 Arranjo físico

A arquitetura do ambiente de trabalho foi definida para facilitar a identificação das peças e garantir uma aplicação a mais ao sistema de visão computacional. Um campo confeccionado de isopor branco foi posicionado em cima de uma mesa suporte à aproximadamente 0,75 metros de altura em relação ao chão.

Cortado com dimensões de 635 x 411 mm, o isopor teve as suas extremidades pintadas de cores diferentes, mostrado na Figura 23. Amarelo, vermelho, azul e verde caracterizam, portanto, os limites do espaço de trabalho onde as peças são posicionadas para a identificação.

Figura 24 - Campo de isopor com extremidades pintadas.

Fonte: Elaborado pelo Autor, 2019.

Assim, conforme mostrado em capítulo anterior, dentre as técnicas de processamento de imagem, a de reconhecimento de cores foi a escolhida para restringir o campo de trabalho a fim de melhorar os resultados de busca por peças no ambiente.

3.2.3 Definição do sensor de imagem

Em um suporte acima do campo de trabalho foi posicionado um *smartphone* com câmera com resolução de 2592x1944 pixels para fotos e 1280x720 *pixels* para vídeos para funcionar como sensor de imagem para o projeto. O *smartphone Android*, executando o aplicativo *DroidCamApp*, que se tornou o responsável pela transmissão das imagens da câmera para o computador. Esse aplicativo – junto com outro instalado no computador - permite a conexão entre aparelhos por meio de *Wi-Fi* ou USB.

Pela praticidade, o modo de operação utilizando o *Wi-Fi* foi previamente escolhido, tendo que ser substituído a seguir pelo USB devido alguns problemas de conexão com a rede.

3.2.4 Linguagem de programação

Como citado anteriormente, para atingir o objetivo principal foi necessário desenvolver dois sistemas diferentes, um de visão computacional e outro de movimentação do braço robótico.

A linguagem utilizada para o processo de identificação dos objetos foi *Python*, por ser uma linguagem dinâmica, de alto nível e orientada a objetos e por ser *open source*

demonstrando facilidade nas buscas por bibliotecas, trechos de programas e documentação completa das funções pré-definidas.

Uma das bibliotecas utilizadas, e talvez a mais importante, foi o módulo *OpenCV (Open Source Computer Vision Library)*. Essa biblioteca de visão computacional contém grande quantidade de processos e técnicas implementados em funções pré-definidas, caracterizando-se quase que essencial para trabalhos com visão computacional.

Outras bibliotecas como o *Numpy*, *Imutils*, *Math*, S*truct* e *Socket* também foram utilizadas pontualmente para específicas operações ao longo do desenvolvimento. Dentre essas é necessário dar um destaque ao *Numpy,* que se mostra de extrema importância para o manuseio com matrizes.

A linguagem utilizada para o acesso ao braço robótico, por sua vez, é a *PDL2,* utilizada pela COMAU em seus braços robóticos, permitindo a realização de movimentos e conexões conforme as necessidades aqui pretendidas.

3.2.5 Definição de ferramenta

Para utilização do braço robótico de forma mais precisa e aplicável ao sistema em desenvolvimento, uma calibração inicial da ponta ferramenta precisou ser feita. Para isso, uma caneta fixada à garra do braço robótico foi utilizada para definir um ponto de referência para a calibração.

A ideia da calibração é fazer com que a ponta ferramenta fixada à garra do braço robótico alcance precisamente a um ponto de referência fixo no ambiente acessível ao braço. Isso, entretanto, deve ser conseguido por meio de variações da forma com que a ferramenta chega ao ponto de referência para a calibração.

Essa definição de ferramenta foi salva na variável interna *Tool 2* e tem sua maior importância no mapeamento do campo de trabalho apresentado na seção seguinte.

As Figuras 24 e 25 detalham etapas desse ajuste inicial.

Figura 25 - Detalhe do posicionamento para definição da ponta ferramenta.

Fonte: Elaborado pelo Autor, 2018.

Figura 26 - Visão macro da definição da ponta ferramenta.

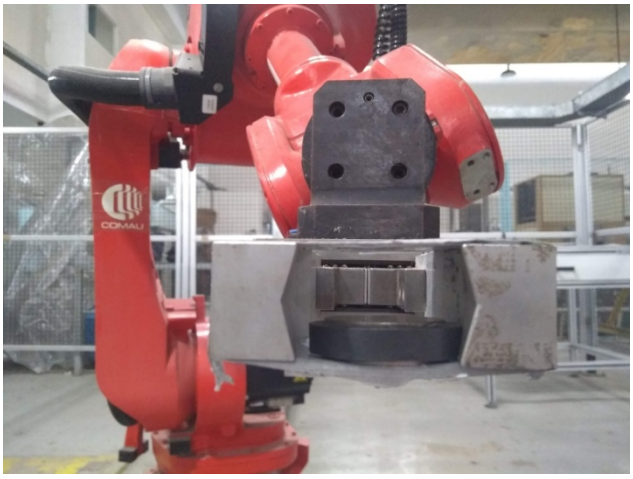
Fonte: Elaborado pelo Autor, 2018.

Como o sistema de acionamento pneumático da garra do braço robótico não pôde ser utilizado, um ímã foi fixado no interior das pinças da garra, conforme pode ser observado na

Figura 26. Essa utilização de um ímã como sistema de captura das peças justifica o fato da aplicação de placas metálicas nas superfícies das peças impressas.

Apesar da precisão conseguida na definição de uma ferramenta, a utilização de ímã como mecanismo de captura das peças tornou tamanha precisão desnecessária para a aplicação. Desta forma uma nova configuração de ponta ferramenta foi desconsiderada, sendo utilizada no programa apenas a ponta ferramenta definida anteriormente *(Tool 2)* associada a um *off-set* para pequeno ajuste nas posições finais.

Figura 27 - Ímã posicionado entre as pinças da garra do braço robótico.



Fonte: Elaborado pelo Autor, 2019.

3.2.6 Definição de espaço de trabalho

Para haver possibilidade de manipulação do braço robótico a partir de entradas de dados referenciados ao sistema de coordenadas definido pelo plano do isopor (campo de trabalho), o sistema de coordenadas que o braço robótico precisaria seguir deveria coincidir com o sistema real do arranjo físico. Desta forma um *workspace* (plano de trabalho) novo precisou ser definido para facilitar o trabalho.

Definido como *Frame 2*, o plano de trabalho criado para o projeto utilizou como base a ponta ferramenta definida na seção anterior (*Tool 2*) para estabelecer as coordenadas de origem de um sistema cartesiano.

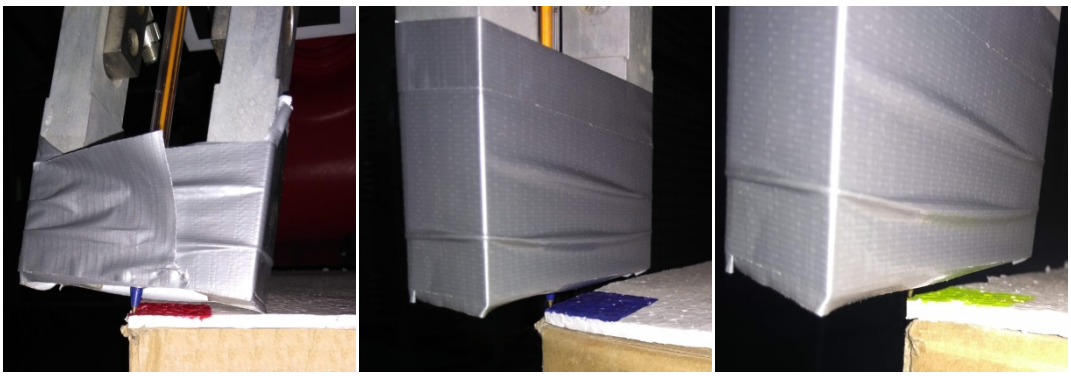
A definição dos eixos de coordenadas do plano de trabalho digital do braço robótico foi realizada em três etapas: a primeira delas consistiu em tocar a ponta ferramenta na origem do sistema de coordenadas; em segundo lugar tocar um ponto do eixo X do plano de trabalho; e

por último tocar um ponto do eixo Y. A definição deste sistema de coordenadas é fixa, portanto foi utilizada ao longo de todo o trabalho, qualquer movimento de reposicionamento da mesa resultaria na necessidade de uma nova definição como esta. O desenvolvimento de aplicações onde esse sistema de referência possa ser restabelecido automaticamente fica como sugestão para futuros trabalhos. Nas Figuras 27 e 28 a seguir é possível observar melhor a definição realizada dos eixos e etapas desse processo para definição do plano de trabalho.

Figura 28 - Sistema de coordenadas definido para o campo de trabalho.

Fonte: Elaborado pelo Autor, 2019.

Figura 29 - Definição do sistema de coordenadas (origem, eixo x e eixo y).



Fonte: Elaborado pelo Autor, 2018.

Definido o campo de trabalho pôde-se definir as posições finais - uma para cada tipo de peça - e a posição inicial para a ponta ferramenta. Essas posições foram configuradas de forma a estarem fora do campo de trabalho e distantes umas das outras a fim de melhor observar as diferenças de posição.

A cada um dos pontos finais - salvos como variáveis internas nomeadas pos_final_1, pos_final_2, pos_final_3 no sistema - foram fixados ímãs para a captura das peças trazidas pelo braço robótico.

3.3 Desenvolvimento do algoritmo de visão computacional

O desenvolvimento do algoritmo de identificação das peças por visão computacional passa por blocos de processamentos conforme mostrado na Figura 18. Para transcorrer de maneira correta, foram implementados códigos em *Python* cada qual com sua funcionalidade para atender aos blocos do fluxograma.

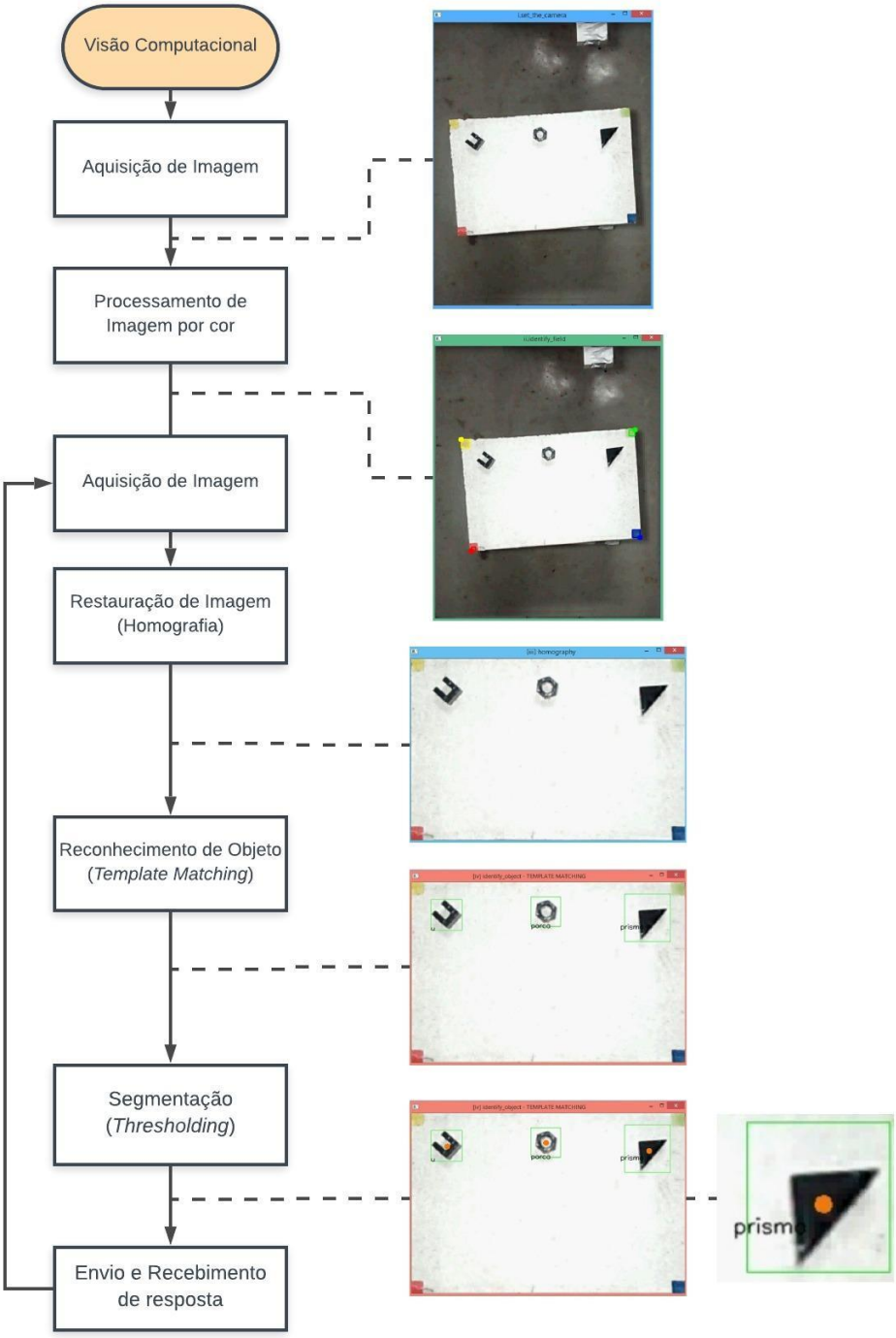
Desta forma, os códigos são executados sequencialmente para cumprirem suas funções específicas no programa, recebendo entradas geradas pelo bloco anterior e retornando saídas que são entradas para os blocos subsequentes. Um bloco de código nomeado *main.py* foi desenvolvido para funcionar como o responsável por garantir esse fluxo de dados, chamando outros blocos para gerarem retornos necessários.

Por questão de organização os arquivos foram nomeados seguindo a sequência em que são requisitados, entretanto suas funções dentro da sequência do programa é o que é importante e por isso estão especificadas a seguir.

O sistema de visão computacional, ilustrado na Figura 29, tem início a partir da aquisição de imagens do ambiente dentro do campo de visão do sensor de imagem. A captura neste ponto é feita e uma imagem de entrada é gerada. A partir de então é iniciado o processamento por cor da imagem de entrada, com o intuito de identificar as extremidades do campo de trabalho. Uma vez identificado o processo de recuperação de imagem por meio de homografia é realizado gerando uma imagem sem distorções para análises. O processo de reconhecimento de objeto é iniciado por meio de *template matching* e as regiões de correlação calculadas superior a um limite definidor são marcadas identificando as peças. Os centróides dessas peças são então definidos por meio dos contornos gerados após aplicação de *thresholding* dentro das regiões obtidas pelo processo de *template matching.*

Concluído este sequenciamento, o ciclo se reinicia com aquisição de nova imagem e buscas por peças em seu interior, repetindo os processos de processamento utilizados até que não haja mais nenhuma peça em campo.

Figura 30 - Processos utilizados na identificação por imagem dos objetos em campo.



Fonte: Elaborado pelo Autor, 2019.

3.3.1 Aquisição da imagem

Como citado anteriormente o processo de aquisição de imagem passa por dois mecanismos descritos a seguir: a transmissão e a caputra.

3.3.1.1 Transmissão

Como citado anteriormente nas configurações iniciais, a transmissão da imagem da câmera ao computador se deu com o auxílio do aplicativo *DroidCamApp* via USB. Esse aplicativo precisa estar aberto a todo momento enquanto os códigos de captura estiverem requisitando imagens.

A transmissão de imagens neste sistema ocorre de forma contínua até a solicitação de captura da imagem a se analisar, que interrompe o fluxo de imagens para o computador.

3.3.1.2 Captura de imagem

Ao ser executado, o programa aguarda uma fração de segundos para fazer a captura de imagem de maneira correta. Até esta requisição de imagem a se analisar, uma série de *frames* vão sendo transmitidos e exibidos para garantir que a câmera tenha sido inicializada sem erros.

Esse processo não depende de variáveis externas para ser executado, e portanto não recebe nenhum tipo de variável advinda da base de conhecimento do programa.

Para o sequenciamento da identificação, esse processo pode ser observado mais de uma vez, isso porque ao se inicializar o programa, antes de entrar para o ciclo de identificação de peças, o programa precisa definir a região (campo) de trabalho, que só é requisitada uma vez a cada inicialização. A seguir, para cada ciclo de identificação de peça é solicitada uma nova aquisição de imagem da situação atual do campo.

3.3.2 Identificação do campo

A identificação do campo de trabalho passa por dois processos distintos envolvendo imagem: a identificação por cor da região do campo de trabalho e a correção de perspectiva e ajuste da imagem por homografia.

3.3.2.1 Identificação por cor

A identificação por cor é utilizada aqui para a determinação da região de trabalho, a partir das extremidades coloridas do isopor. O código responsável por essa etapa foi nomeado *ii_identify_field.py*.

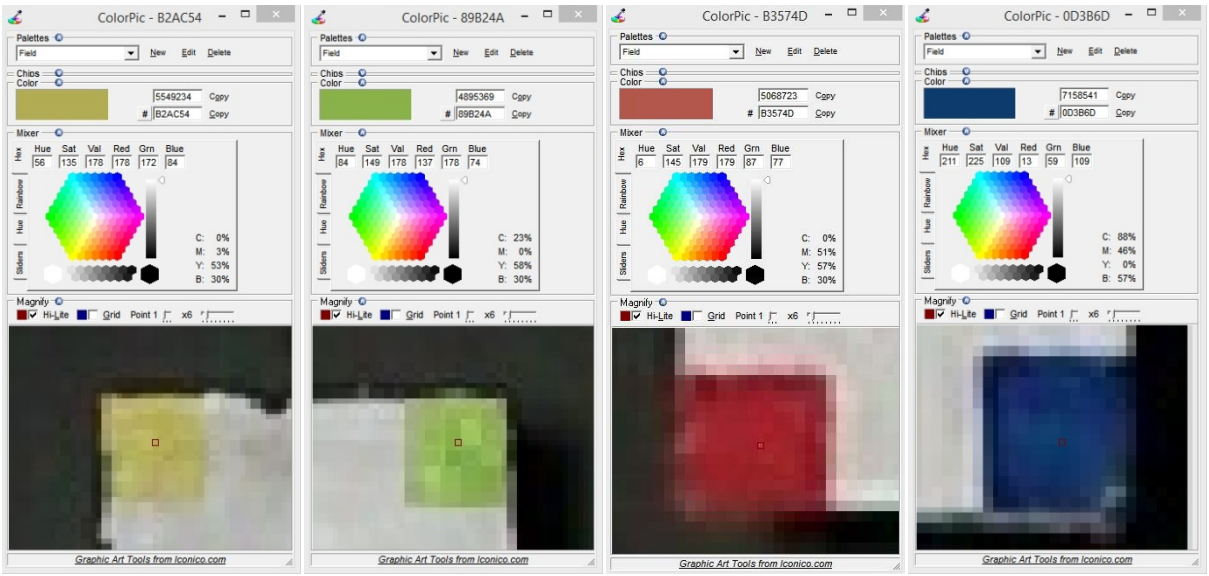
Nesse processo é realizada de forma direta a classificação por cor das regiões do campo, conforme as técnicas de processamento de imagem por cor, citadas anteriormente. Com o intuito de selecionar o campo de trabalho para posterior busca por peças, o código busca pelas

cores previamente definidas (azul, amarelo, vermelho e verde) para selecionar a região desejada.

Esse bloco de código recebe do processo anterior (aquisição da imagem) a entrada da imagem capturada. Ao identificar as regiões de cores nesta imagem, ele calcula os centróides dos marcadores para em seguida retornar ao programa principal a matriz de posições x e y referentes às extremidades das regiões retangulares que englobam todo o contorno de cor.

Com base em coleta *in loco* das coordenadas de cores HSV nas extremidades do isopor, utilizando o *software ColorPic*, foram estabelecidos limites inferiores e superiores das cores. Posteriormente, algumas extrapolações dos resultados foram utilizadas para abranger um intervalo maior de cores, diminuindo as interferências por iluminação do ambiente. Na Figura 30 pode-se ver o registro da coleta:

Figura 31 - Obtenção do range de cores das extremidades do campo utilizando o *ColorPic*.



Fonte: Elaborado pelo Autor, 2018.

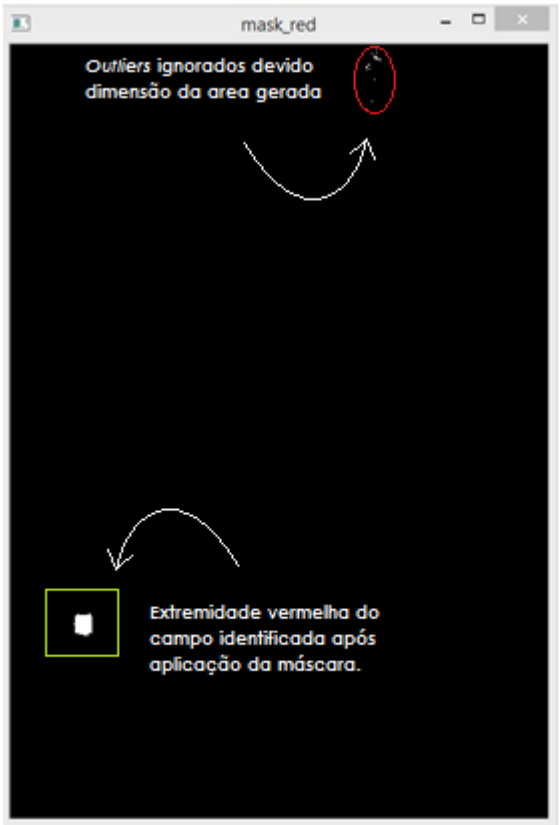
Após coleta realizada em três dias e horários diferentes, conforme observado nos Apêndices A, B, C e D, foram definidos os vetores de coordenadas mínimo e máximo para cada cor no modelo HSV. Os limites estabelecidos foram:

```
boundaries = [
([1/2, 139, 116], [11/2, 209, 179]),        # red
([210/2, 80, 41], [243/2, 236, 126]),       # blue
([47/2, 47, 165], [70/2, 198, 199]),        # yellow
([73/2, 77, 157], [98/2, 149, 196])]        # green
```

Os parâmetros da primeira posição dos vetores, correspondente ao *Hue* (tonalidade ou matiz) em coordenadas HSV variam de 0º a 360º, entretanto a formatação das funções no *OpenCv* do *Python* aceitam esses valores com a variação de 0º a 180º. Isso justifica a divisão por 2 desses parâmetros no código acima.

Com a imagem de origem e esses vetores aplicados à função *cv2.inRange( )* consegue-se criar imagens máscaras definidoras das regiões onde as cores do vetor são presentes. A partir disso, com uso da função *cv2.bitwise_and( )* a imagem de origem e as máscaras criadas passam pela conjunção *bit* a *bit* desses *array's,* gerando uma imagem com a posição exata do *range* de cor bem discriminada. A Figura 31 exemplifica uma das máscaras criadas no processo.

Figura 32 - Máscara criada a partir da restrição do *range* de cores.



Fonte: Elaborado pelo Autor, 2019.

Como observado na Figura 31, possíveis *outliers* são tratados ignorando-se regiões de área muito menor ou muito maior que a área gerada por uma extremidade do campo. Para ter alguma base a definição do intervalo de áreas aceitável passou por uma verificação da media das áreas geradas por essas extremidades, dada a altura da câmera e manipulações da imagem.

3.3.2.2 Homografia

Como pode ser percebido na imagem anterior, o retorno gerado pela identificação das extremidades do campo por meio de processamento de cor não é o suficiente para a criação de uma região delimitada e de perspectiva adequada para os trabalhos de identificação da peça e correta referência para o braço robótico posteriormente.

Sendo assim, o bloco de código denominado *iii_homography.py* foi criado para corrigir a perspectiva da imagem e delimitar a imagem de trabalho. Utilizando a matriz de pontos das extremidades do campo de trabalho consegue-se, utilizando a técnica de homografia citada anteriormente, criar uma imagem de perspectiva corrigida, restringida por meio de cortes na imagem original.

As regiões desnecessárias para futura identificação das peças no campo são desprezadas e a região de interesse é corrigida e planificada, retornando à base de conhecimentos principal um imagem mais objetiva formada apenas com a região do campo de trabalho.

Para a aplicação da homografia na imagem original uma série de micro processos são necessários, eles são viabilizados a partir da utilização das funções pré-definidas do *OpenCv*: *cv2.findHomography( )* e *cv2.warpPerspective( )*.

A primeira das funções encontra uma transformação de perspectiva entre dois planos, correlacionando a matriz de coordenadas dos quatros pontos obtidos do bloco de código anterior - definidor da região de interesse - a quatro pontos de referência, alvos para destino dos primeiros. Dessa forma, a função aloca as coordenadas às novas posições para planificar e corrigir a imagem retornando uma matriz de transformação de perspectiva h entre os planos procurados e de destino.

A função *cv2.warpPerspective( )* portanto aplica a transformada de perspectiva à imagem de origem, gerando como retorno do bloco de código a imagem corrigida de perspectiva e recortada apenas com a região de interesse.

```
def perspective_correction(image,vector):
    print "\n\n   [iii_homography]"
    size = (635, 411, 3)
    im_dst = np.zeros(size, np.uint8)
    pts_dst = np.array( [[0,0],
                        [size[0] - 1, 0],
                        [size[0] - 1, size[1] -1],
                        [0, size[1] - 1 ]
                        ], dtype=float
                        )
    pts_src = np.array( [
                        [vector[2][0], vector[2][1]],
                        [vector[3][0], vector[3][1]],
                        [vector[1][0], vector[1][1]],
                        [vector[0][0], vector[0][1]]
                        ], dtype=float
                        )
    h, status = cv2.findHomography(pts_src, pts_dst)
    im_dst = cv2.warpPerspective(image, h, size[0:2])
```

3.3.3 Identificação dos objetos

Dentro da etapa de identificação dos objetos são necessárias dois dos processos comentando anteriormente. São eles: a segmentação e o reconhecimento de objeto.

3.3.3.1 Reconhecimento de objeto

A identificação dos objetos é realizada no bloco de código nomeado *iv_identify_object.py*. A partir da imagem corrigida resultante das manipulações anteriores, consegue-se aplicar as técnicas de reconhecimento de objetos para buscas na imagem pelas peças anteriormente citadas.

Para isso, a técnica de *template matching* citada na seção de reconhecimento de objetos é utilizada. Conforme apresentado, a aplicação dessa técnica depende da utilização de *templates* dos objetos - imagens em recorte objetivamente da peça que se quer buscar.

Foram preparados 360 *templates* de cada tipo de peça, rotacionados grau a grau para que cada *template* correspondesse a um ângulo de rotação da peça em relação ao sistema referencial do campo. Essa abordagem permitiu a identificação do tipo de peça, sua localização no campo e seu ângulo de rotação - apesar de posteriormente, com a substituição das garras do robô por captura por ímãs, esse ultimo parâmetro tornar-se desnecessário vez que para qualquer angulação a captura seria feita da mesma forma.

O código de reconhecimento, portanto, funciona testando a aplicação da técnica de *template matching* a cada *template* criado. Assim, a uma taxa de aproximadamente 50 *templates* por segundo em um *hardware* de processador Intel® Core™ i3-5005U CPU 2.00GHz de 4GB de RAM, cada um dos 360 *templates* criados, de cada peça, é testado ao longo da imagem de busca (da direita para a esquerda, de cima para baixo) em busca do melhor valor de correspondência. O melhor valor de correspondência indica, então, a posição da região onde encontra-se a peça (indicada em coordenadas cartesianas da imagem de busca), a angulação da peça e o tipo de peça que foi identificada (ambos correspondentes ao melhor *template* identificado).

3.3.3.2 Segmentação

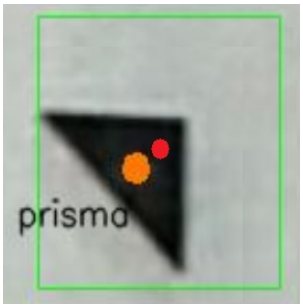
Uma vez que se é definida dentro da imagem do campo a região de maior precisão para a localização de uma determinada peça, busca-se precisar fielmente a exata posição da daquele tipo de peça dentro da região encontrada. Isto porque nem sempre a região encontrada tem como centro o centróide da peça buscada.

Para precisar a exata posição da peça, uma etapa de segmentação é necessária. Assim, utilizando a segmentação por *thresholding* dentro apenas da região de correspondência selecionada pelo *template matching* anteriormente consegue-se gerar um contorno gerado pela peça em contraste ao fundo branco do isopor.

Isso permite a precisão na obtenção do centróide da peça em coordenadas cartesianas da imagem que é facilmente convertida de *pixels* para milímetros e salvas em um vetor final.

Na Figura 32 pode ser observada a diferença entre o centróide da peça obtido com a aplicação de *thresholding* após delimitação da região da peça por *template matching* (de laranja) e o centróide da peça caso considerado o centro da região obtida pelo processo de *template matching* como centro da peça (de vermelho).

Figura 33 - Comparação entre centróide da peça e da região.



Fonte: Elaborado pelo Autor, 2019.

Desta forma, ao final desses dois processos de identificação de imagem, é obtido um único vetor contendo as informações definidas anteriormente de posição, angulação e tipo de peça. A representação deste vetor resultante pode ser observada a seguir:

$$vetor = [\, X, Y, \hat{A}ngulo, Tipo\ de\ peça \,] \quad (2)$$

Sendo as coordenadas X e Y dadas em milímetros, a coordenada angular dada em graus e o tipo de peça definido entre 1, 2 ou 3, correspondendo à peça identificada. De modo que ao serem enviadas para o braço robótico nenhum tipo de conversão seja necessária.

O último bloco de código desenvolvido na linguagem *Python* está descrito na seção a seguir por estar relacionado ao envio de informações entre os *hardwares*.

3.4 Comunicação *hardware*-braço robótico

A troca de informações entre os sistemas só foi possível a partir do protocolo de comunicação TCP/IP citado anteriormente. Uma comunicação por meio de protocolo TCP exige que se estabeleça uma conexão entre as máquinas e para isso foi desenvolvido o último bloco de código em *Python*, descrito a seguir.

O envio de informações se deu através do código *v_socket.py*, sendo o principal responsável pelo envio do vetor de informações analisadas em imagem para o código em *PDL2*. Para isso a conexão TCP/IP é estabelecida por meio da definição de IP e porta de acesso idêntica entre as máquinas.

Feito isso a informação de saída – vetor posição da peça – é empacotada e enviada em um só pacote de dados para a máquina servidora que ao receber associa as variáveis de se seu sistema.

Após o envio do vetor, para cumprir corretamente o fluxograma de ações estabelecido anteriormente o programa aguarda um retorno por parte do sistema em *PDL2*, que é enviado ao finalizar o ciclo de movimentos, e retornar *True* confirmando o fim da sequência de movimentos do ciclo realizado pelo braço robótico.

3.5 Desenvolvimento do algoritmo de rotas

O algoritmo de rotas do braço robótico foi desenvolvido por completo em linguagem *PDL2*. Ao todo foram utilizados cinco subprogramas, chamados rotinas (ROUTINE), além de um programa principal MAIN onde as rotinas são chamadas para integralizar a sequência de

ações do braço robótico. As rotinas utilizadas foram: *tcp_accept*, *ru_get*, *ToolFrame*, *RmtToolFrame* e *move_to_object* (desenvolvida pelo Autor) e serão explicadas a seguir.

O programa principal MAIN inicia suas linhas chamando a rotina *ToolFrame*, associado aos parâmetros (2, 2). Essa rotina associada à rotina interna *RmtToolFrame* é responsável pela inicialização de duas configurações importantes para o sistema, a primeira, selecionada pelo parâmetro "2", é a ponta ferramenta definida para esta aplicação; a segunda também descrita pelo parâmetro "2" é o *frame* de trabalho configurado, isto é, o espaço de trabalho pré-definido nas configurações iniciais.

Em sequência, a primeira movimentação do braço é requisitada, a linha de código descrita por "MOVE TO pos_inicial" solicita a movimentação do braço robótico para a posição inicial do braço no início do ciclo de funcionamento definida pela variável "pos_inicial". Esta variável foi pré-definida nas configurações iniciais diretamente na memória do sistema.

Por questões de logística para o melhor funcionamento do sistema de visão computacional, a posição inicial da extremidade do braço (ponta ferramenta) foi definida longe do campo de visão da câmera.

A inicialização das ações objetivas é dada pela linha de código que chama a principal rotina de integração do braço com o programa desenvolvido em *Python* para identificação de peças. A rotina nomeada por *move_to_object*, desenvolvida como parte do projeto, executa ações de recebimento de informações, movimentações em direção à peça e retorno do tipo de peça identificada.

Ao ser executada, a rotina *move_to_object* chama outras duas rotinas (*tcp_accept e ru_get)*, aproveitadas do trabalho de Gomes e Teixeira (2018) para o recebimento e envio de informações ao programa em *Python* via conexão TCP/IP. Desta forma, o recebimento do vetor empacotado via *socket* preenche a variável "XYAT", definida no início do código como um *array* de quatro posições. Os valores são atualizados em *loop* pela leitura do arquivo *lun_tcp_rs* (variável de conexão que recebe os pacotes de informação) e alocados nos espaços de *array* da variável "XYAT".

Como citado anteriormente, uma variável do tipo POSITION para ser definida precisa dos parâmetros (X, Y, Z, A, E, R) preenchidos com valores possíveis dentro das limitações do braço robótico. Desta forma os valores não relacionados às informações recebidas no vetor

são estabelecidas diretamente na programação da rotina. Os ângulos E e R recebem respectivamente 90º e 0º e a coordenada cartesiana Z (altura referenciada no campo de trabalho) recebe o valor provisório de 200 mm.

Assim, ao se concluir o preenchimento da variável XYAT criada, consegue-se alocar os valores de X, Y e A à variável POSITION e transcrever a posição da peça em coordenadas tridimensionais do braço robótico, completando as seis coordenadas necessárias.

A última posição da variável XYAT, no entanto, como citado na seção de código de identificação de objetos refere-se ao tipo de peça identificado. Armazenado como um inteiro no código em PDL2, essa variável é atribuída como retorno de toda a rotina *move_to_object*.

Definida a posição da peça na variável *pos_object* e do tipo de peça em *my_object*, a ferramenta do braço robótico é enviada para a posição da peça ainda a uma altura Z de 200 mm estabelecida no início da rotina. E só então, um decremento linear de altura até Z igual a 55 mm é implementado. Essa etapa anterior à chegada à peça é necessária devido a possíveis colisões entre a ponta ferramenta e outras peças ou campo.

Após a ponta ferramenta chegar a seu ponto mínimo e tocar a peça com o ímã em sua extremidade os dois corpos estarão magneticamente atraídos e a captura estará concluída. Com a peça "em mãos" o braço inicia uma nova subida devido ao incremento de coordenada Z (até 300 mm), finalizando a rotina *move_to_object*.

Voltando ao programa principal MAIN, o destino final do braço robótico com a peça é indicado pelo retorno em número inteiro da rotina *move_to_object* finalizada. Três casos foram implementados, um para cada tipo de peça, cada qual enviando o braço robótico a uma posição final previamente definida, pos_final_1, pos_final_2, pos_final_3. Essas posições foram previamente definidas nas configurações iniciais do sistema, salvas como variáveis internas na central de comandos conforme citado. Algumas linhas de códigos foram escritas posteriormente para suavizar a chegada ao destino final, fazendo com que o braço primeiramente chegasse a uma posição superior à posição final, e só então pudesse descer linearmente para as posições finais.

Ao chegar à posição final, a peça é atraída pelo ímã presente nesta posição, liberando o braço robótico para novo ciclo. Quando isso ocorre o ciclo de ações se encerra enviando um sinal para o *Python* informando a sua finalização. Ao receber este pacote o código em *Python* reinicia o ciclo de identificação de peças para a coleta de possíveis peças ainda em campo.

4 RESULTADOS

Nesta seção são apresentadas as respostas do sistema ao longo dos testes de implementação do algoritmo de visão e integralização ao braço robótico, além de citar problemas que surgiram ao longo do desenvolvimento. É importante destacar que nem sempre as condições do ambiente eram favoráveis aos testes, e que detalhes técnicos ao bom funcionamento da identificação foram sendo resolvidos à medida que os problemas apareciam até que se chegasse a uma aplicação funcional da integração da visão computacional ao braço robótico.

Desta forma, o objetivo principal desta seção é a validação da integração dos sistemas desenvolvidos neste trabalho e apresentação dos problemas e soluções que o mesmo apresentou no decorrer do processo.

Para validação funcional do sistema, foi filmado de dois pontos de vista diferentes um ciclo completo de atuação dos sistemas integrados: uma filmagem do ambiente físico com as movimentações do braço robótico; e uma filmagem da tela do computador utilizado para rodar o programa de visão computacional em *python*. Essas filmagens estão no *link*: https://www.dropbox.com/sh/n9urzgr5gf90dxe/AADv7YJM7PDm47rz6l_cgkM7a?dl=0.

4.1 Aquisição da imagem

A aquisição da imagem feita por meio do bloco de código *i_camera_setting.py* e câmera posicionada acima do campo de trabalho contou com detalhes não previstos, que precisou de tomadas de ações pontuais para o melhor funcionamento.

O primeiro ajuste realizado foi na transmissão da imagem da câmera para o computador. Apesar da ideia inicial de utilizar a *Wi-Fi* para isso, as circunstâncias *in loco* dificultaram essa forma de transmissão, que por praticidade acabou sendo adaptada para funcionar via cabo USB. Para isso foram necessários pequenos ajustes no código *Python* e no aplicativo *DroidCamApp* - instalado tanto no *smartphone* quando no computador.

Um ponto limitador quanto a utilização do sistema refere-se a necessidade de inicialização manual desse aplicativo no começo do processo, tanto no *smartphone* quanto no computador. Desta forma, tanto a aplicação por *Wi-Fi* quanto a alguma alternativa de inicialização automática do aplicativo dentro do próprio código *Python* ficam como sugestão para o desenvolvimento de futuros trabalhos.

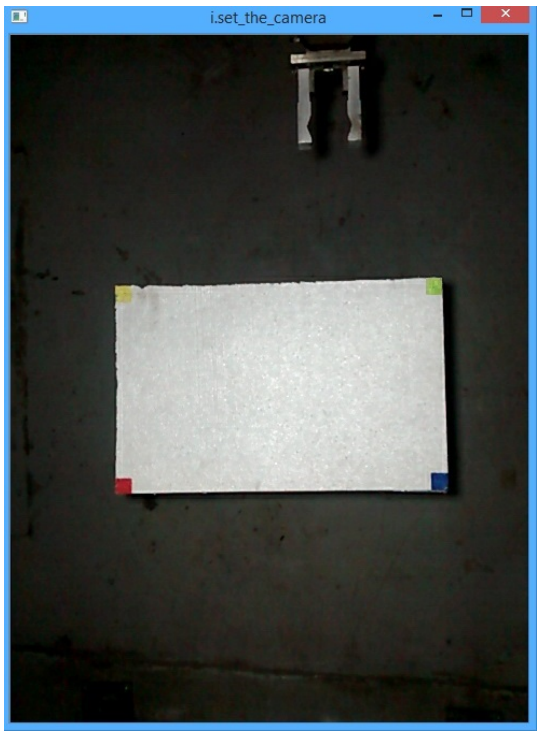
O segundo detalhe corrigido foi a modificação de rotação da imagem capturada (Figura 34). Essa mudança foi necessária uma vez que os sistemas seguintes de identificação de campo foram implementados para atender a uma imagem com a cor azul posicionada ao canto inferior direito da imagem. Como o sistema de suporte foi montado para ter o *smartphone* posicionado em posição única, modificações no código foram preferidas para rotacionar digitalmente a imagem, a fim de não modificar a estrutura montada.

Apesar destes detalhes a aquisição das imagens desta forma ocorreu sem problemas, sem perda de qualidade ou resolução das imagens obtidas. Na Figura 33 pode ser observado como a imagem bruta é obtida para início de processamento com a utilização do *software DroidCam Client*.

Figura 34 - Imagem bruta obtida pelo *DroidCam Client*, antes de qualquer manipulação.

Fonte: Elaborado pelo Autor, 2019.

Figura 35 - Imagem rotacionada gerada pelo código de inicialização da câmera.



Fonte: Elaborado pelo Autor, 2019.

4.2 Identificação do campo

A identificação do campo de trabalho por meio do reconhecimento das cores das extremidades do isopor possui suas limitações de operação. Como citado anteriormente o código *ii_identify_field.py* identifica as cores pré-fixadas (azul, amarelo, vermelho e verde), entretanto apesar desse processo ocorrer da forma desejada, em certas condições a identificação é prejudicada devido a uma série de fatores: superfícies do ambiente de cores semelhantes no campo de visão da câmera, iluminação do ambiente, restrição de área dos contornos obtidos, etc.

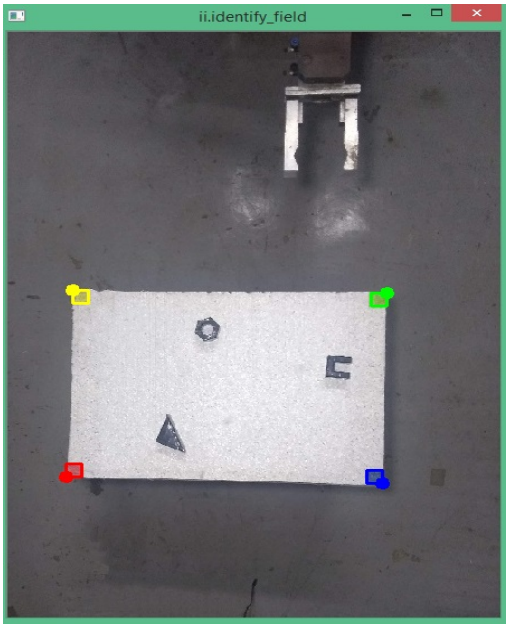
De início uma confusão relacionada ao vermelho do braço robótico foi observada. Devido à presença dessa superfície no campo de visão da câmera, a identificação da extremidade vermelha com campo era prejudicada. A solução simples para esse problema foi a configuração da posição inicial do braço robótico longe do campo de visão da câmera.

Uma segunda fonte de confusão para o sistema era o cabo *ethernet* (azul ou vermelho) que por vezes ficam dentro do campo de visão da câmera. O isolamento da área para impedir a presença de qualquer outro objeto fora do contexto solucionou definitivamente e de maneira simples essas situações.

Uma terceira limitação quanto à identificação das cores diz respeito ao reconhecimento do amarelo do campo de trabalho. Este detalhe se refere também a problemas recorrentes devido à iluminação do ambiente. Os testes realizados quando a luz do sol ainda se fazia presente terminavam sempre gerando erros devido a variação da iluminação natural. Normalmente, feixes de luz do sol que adentram pelo galpão onde localiza-se o braço robótico, refletiam no chão do ambiente espectros de tonalidade amarela que confundiam a identificação da extremidade amarela do campo. Este detalhe, por sua vez, é de difícil solução, isto porque sob essas condições a cada minuto a iluminação natural pode se comportar de maneira completamente diferente do minuto anterior, impedindo a definição fixa de uma *range* de coordenadas de cores que definam as máscaras. Este problema foi resolvido ao limitar a realização dos testes à mesma uma faixa de horário após o pôr do sol, utilizando uma iluminação própria sempre.

Feito isso, pôde-se fixar as coordenadas de cores *HSV* definidoras dos quadrados das extremidades do campo de trabalho gerando melhores resultados, independente do dia que estaria sendo testado. A Figura 35 mostra o resultado após identificação do campo:

Figura 36 - Identificação de cores no campo.



Fonte: Elaborado pelo Autor, 2019.

Apesar da melhor eficiência dos testes durante o período da noite, com iluminação forçada, em exceções ocorriam algumas falhas na identificação de cores. No geral essas falhas decorreram do insuficiente número de coleta das coordenadas *HSV* nas regiões extremas do campo, configurando um intervalo que não abrangia por inteiro as cores das extremidades

com certas variações de iluminação. Esses problemas foram contornados extrapolando o intervalo de cores permitidas para cada cor.

Com o aumento da permissividade de cores em uma máscara, surgiram *outliers* que precisavam ser tratado de alguma maneira para não serem confundidos com a região da extremidade do campo. O tratamento desses *outliers* foi feito utilizando a área que cada contorno gerava, conforme citado anteriormente. Dada a altura de posicionamento da câmera, se muito maior ou muito menor que a área esperada de uma extremidade do campo, ignorava-se o *outlier*.

A restrição por área foi eficaz e fundamental para ignorar a fonte de erro que eram os *outliers,* mas fez surgir uma limitação na identificação das regiões. Em casos de intensas variações de iluminação, onde o intervalo de cor estabelecido não fosse suficiente para discriminar parte da própria extremidade colorida, a restrição da área poderia fazer ignorar uma extremidade completa devido à pequena área identificada com a cor.

Na Figura 37 é possível observar que na extremidade superior direita, a região verde conseguiu ser identificada, mesmo perdendo parte de sua área devido ao intervalo incompleto estabelecido cores. Essa região mesmo reduzida, corretamente, não foi considerada um *outlier*- que precisaria ser bem menor para ser ignorada.

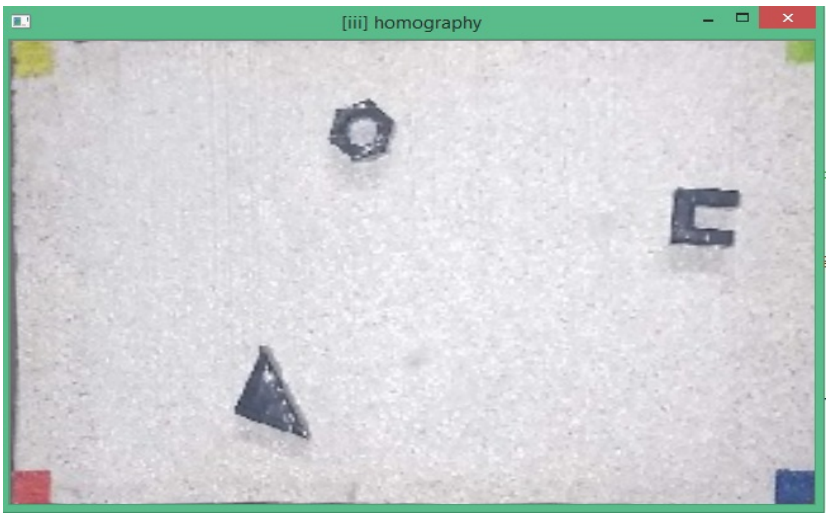
O ideal seria que em todos os testes, todas as regiões estivessem com suas áreas identificadas por completo, o que acontecia na maioria das vezes. Entretanto, em casos que a porcentagem de área perdida fosse elevada, problemas na calibração poderiam aparecer isto por que a associação entre os sistemas de referências, da imagem digital e do *UserFrame* mapeado anteriormente, poderiam perderiam suas calibrações erros sistemáticos no apontamento da localização de peças.

Apesar de não ser o ideal, mas como forma de fazer os testes seguirem sem a parada dos códigos devido a geração de erros, alguns *off-sets* foram pré-estabelecidos para melhor correspondência das extremidades às posições reais.

4.3 Homografia

A homografia para correção de perspectiva da imagem, através do bloco *iii_homography.py* gerou resultados esperados, sem nenhum tipo de problema quando recebidos os pontos da imagem de entrada corretamente do bloco de código anterior. A seguir se consegue observar o resultado (imagem de saída) gerado para as próximas etapas:

Figura 37 - Correção de perspectiva



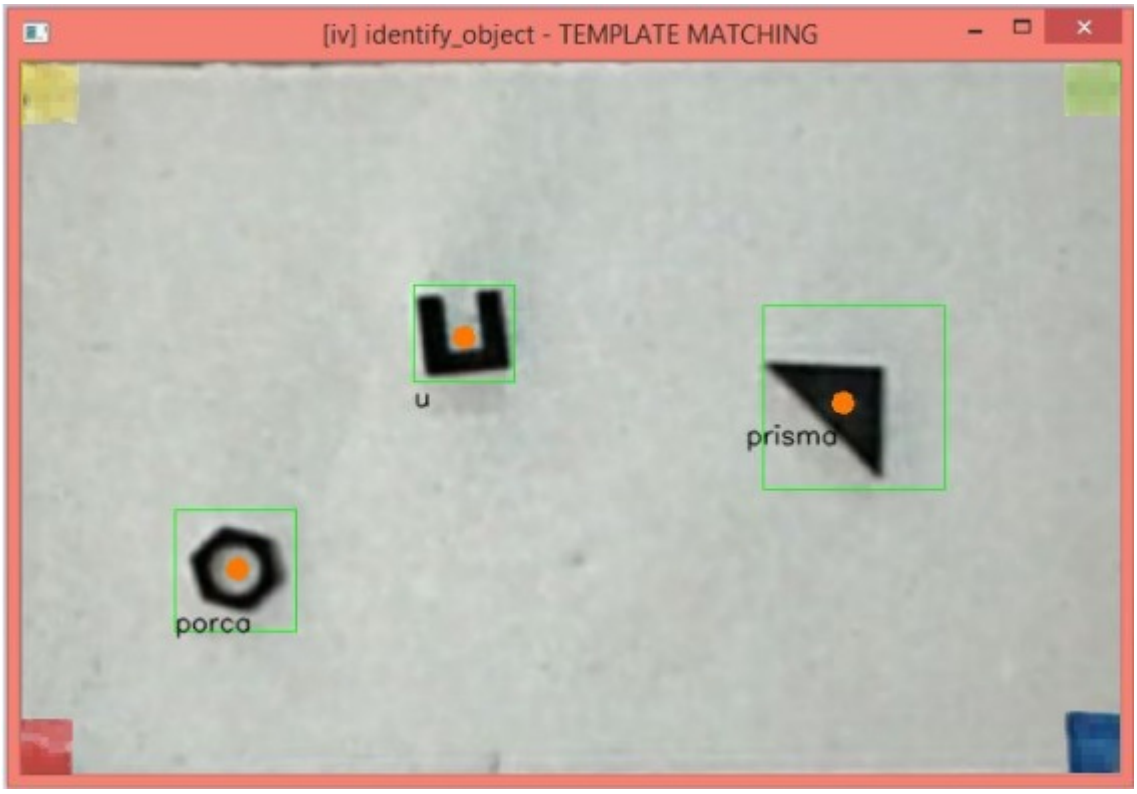
Fonte: Elaborado pelo Autor, 2019.

Como citado anteriormente, o ideal é que os pontos fornecidos pela identificação de cores sejam exatamente os mesmos das extremidades do campo de trabalho. Como nem sempre isso ocorre como o esperado devido ao problema citado para identificação de cores, pequenos ajustes com *off-sets* foram implementados, resultando em melhores resultados para o sistema.

4.4 Identificação do objeto

O bloco de código *iv_identify_object.py,* mais longo dentre todos, se comportou bem após implementação da técnica de *template matching*. Os vetores de saída, contendo as coordenadas X e Y, o ângulo de rotação da peça e o seu tipo foram gerados conforme previsto, modificado apenas a conversão entre sistema de coordenadas (sistema da imagem para sistema do campo de trabalho do braço). A Figura 37 mostra um dos resultados obtidos ao longo dos testes deste processo.

Figura 38 - Identificação de peças.



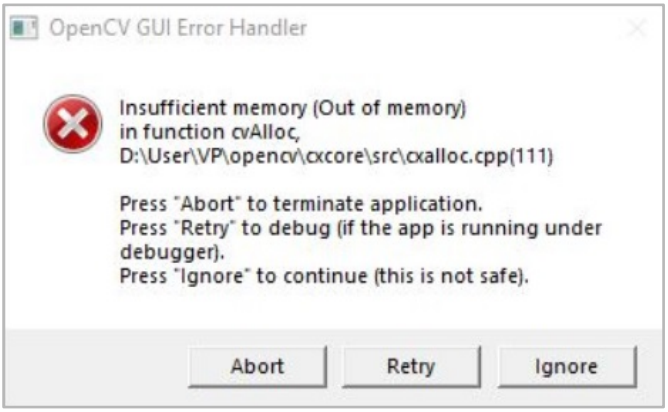
Fonte: Elaborada pelo Autor, 2019.

Esta etapa de implementação foi a que mais sofreu modificações ao longo do projeto e por isso, foi a que mais demorou a ser finalizada. As modificações giraram em torno da busca pela melhor técnica de identificação para o caso proposto. As técnicas utilizadas correram os diferentes níveis de estratégias, desde as muito simples, como identificação por número de arestas de uma peça até técnicas mais sofisticadas como o *Haar Cascade*.

A identificação por número de arestas chegou a gerar resultados satisfatórios em um primeiro momento de descoberta do universo da visão computacional, entretanto as taxas de erro devido a falsos positivos, isto é, confusão entre peças, descartou a utilização da estratégia para integração ao braço robótico.

A tentativa de utilização do método *Haar Cascade* foi o responsável pelo atraso na finalização do projeto como um todo. O tempo elevado necessário para criação de bons classificadores reduziu a perspectiva de atingir o objetivo da identificação por imagem fazendo necessária a busca por novas técnicas. Além disso, a necessidade de um processador melhor para a geração desses classificadores limitou a amplitude de atuação com o método, como pode ser observado na Figura 38.

Figura 39 - Erro por falta de memória em tentativa de criação de classificador.



Fonte: Elaborado pelo Autor, 2018.

Como citado, bons classificadores precisam de milhares de imagens positivas e outras milhares de imagens negativas para gerarem boa classificação de peças. Além disso, é importante lembrar que três classificadores precisavam ser criados, um para cada tipo de peça.

Devido à insuficiência de memória do processador do computador, as tentativas com milhares de imagens como sugerido pelo método não foram possíveis. Limitado por um máximo de 300 imagens positivas e 600 negativas, que precisavam de 4 horas para o treinamento, os classificadores gerados não foram suficientes para garantir uma boa acurácia.

O sucesso da classificação de peças só se deu após tentativa de utilização da técnica do *Template Matching*. Em primeira tentativa de utilização do *template matching* já gerou resultados com 100% de acurácia e 0% de falsos positivos, isso com apenas 30 minutos de trabalhos preliminares com a imagem *template*, sendo escolhido, portanto para a continuação do trabalho.

Apesar da captura dos objetos ocorrer por meio magnético, e por isso a identificação do ângulo de rotação da peça não ser necessário, o algoritmo de reconhecimento utilizando *template matching* foi desenvolvido para obter essa informação. A solução para obtenção deste dado, visto que a técnica não atende bem esse reconhecimento, foi a criação do banco de dados de *templates* rotacionados grau a grau, conforme Figura 39 e Apêndices E, F e G.

Figura 40 - *Templates* da peça U rotacionados de 0º, 30º, 45º, 60º e 90º respectivamente.

Fonte: Elaborado pelo Autor, 2019.

Vale ressaltar que para o sistema de identificação criado, as correlações calculadas entre os *templates* e a imagem de entrada eram feitos para todos os 360 *templates* de cada um dos três tipos de peça. Isso significa que para processar uma imagem de entrada o sistema varre 1080 vezes a imagem principal, percorrendo-a com os *templates* e calculando as correlações para cada ponto da imagem. Essa abordagem resultou em um tempo de processamento relativamente elevado (20 segundos) para a identificação por completo de todos os objetos no campo.

Esse tempo de processamento poderia ter sido reduzido se fosse diminuído também a precisão na identificação da angulação das peças. Isso significa fornecer ao sistema menos *templates*, com rotações mais espaçadas entre os ângulos. Um bom exemplo seria aumentar para 15 em 15º a variação de angulação da rotação dos *templates*; com isso, o sistema estaria varrendo a imagem de entrada com apenas 24 *templates* por peça ao em vez dos 360 utilizados no projeto. Isso reduziria de 1080 para 72 o número de vezes em que *templates* estariam varrendo a imagem de entrada, resultando em tempos de identificação bastante reduzidos.

Para reduzir ainda mais esse tempo de processamento, melhorias no código podem ser implementadas para diminuir ainda mais o número de *templates* utilizados. Uma das ideias para isso seria um melhor aproveitamento das peças simétricas, que não foi feito no caso da peça 1 (porca) por praticidade de formulação de um código generalista, que servisse para qualquer tipo de peça.

Por fim, após o correto funcionamento do sistema de classificação, a necessidade da inserção de um sistema dentro da região identificada de interesse caracterizou-se como um detalhe não previsto que precisou ser implementado e assim o foi de maneira eficiente. A identificação do centróide da peça através da utilização da técnica de *thresholding* completou o objetivo final por parte da visão computacional.

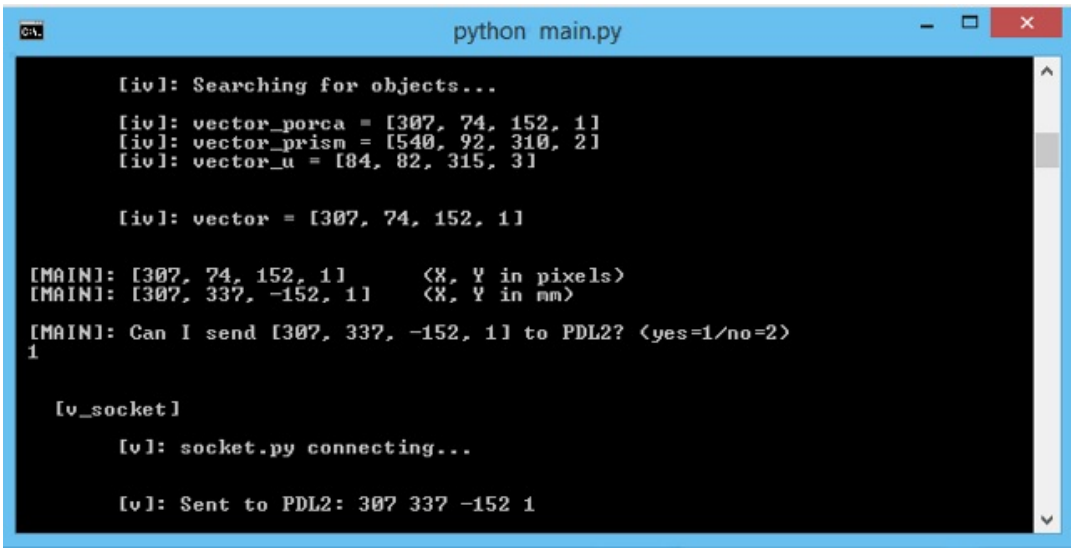
É importante destacar que esse sistema sofre de uma limitação para uma situação específica de posicionamento das peças. Essa situação de possíveis falhas na identificação corresponde ao caso em que as peças estão muito próximas uma das outras. Soluções para contornar essa limitação podem ser pensado e implementado em futuros trabalhos.

4.5 Envio de informações (*socket*)

O envio da informação via comunicação TCP/IP, utilizando o pacote *socket* do *Python* foi bem sucedido. As informações classificadas e armazenadas no vetor gerado no processo anterior foram compactadas de forma correta para o envio. O pacote de informações enviado ao braço robótico consegue ser captado corretamente e ao ser descompactado é alocado para as variáveis corretas seus valores.

Nas Figuras 40, a seguir consegue-se ver um exemplo de impressão dos valores enviados pelo código em *Python* ao código executado simultaneamente em *PDL2*.

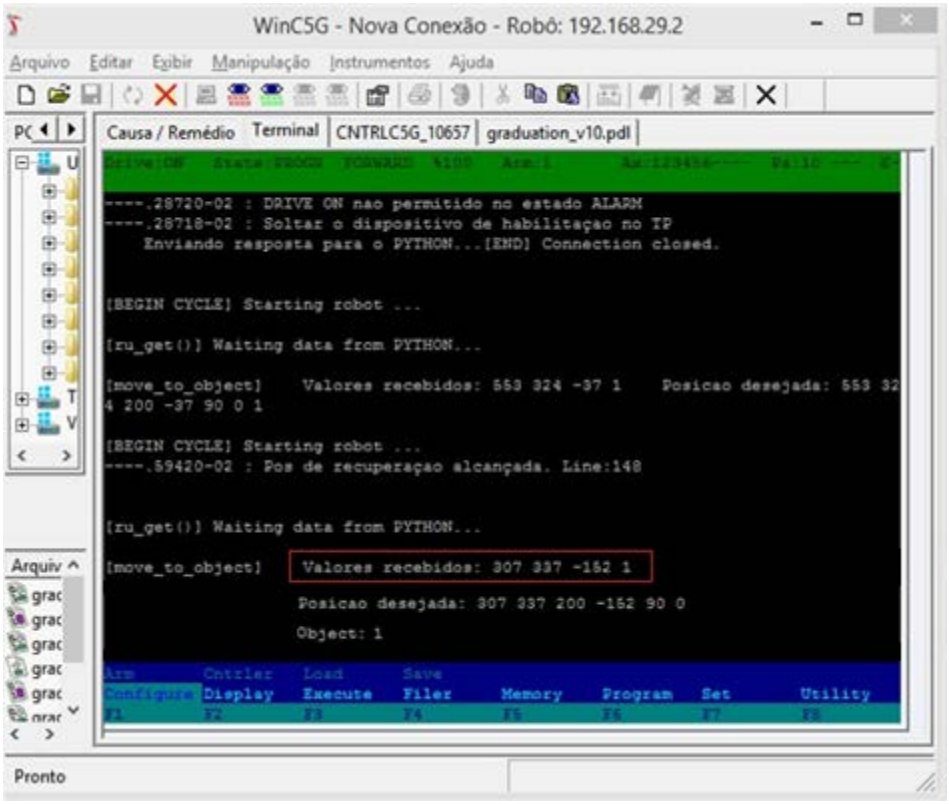
Figura 41 - Envio do vetor posição com sucesso do *Python* para o robô.



Fonte: Elaborado pelo Autor, 2019.

O recebimento das informações por parte do braço robótico pode ser observado na Figura 41. As movimentações conduzidas por essas informações fazem o ciclo do sistema ser finalizado de maneira correta, conforme vídeo disponível no *link* anteriormente citado. As limitações neste sentido refere-se à solução para robustez da movimentação, que para esse trabalho não foi implementada. Soluções para possível interferência ou colisão ao longo dos movimentos ficam, portanto, como oportunidade de melhoria para desenvolvimento em futuros trabalhos.

Figura 42 - Recebimento do vetor posição com sucesso pelo robô.



Fonte: Elaborado pelo Autor, 2019.

5 CONCLUSÃO

No presente trabalho, foi desenvolvido um sistema de visão computacional utilizando a técnica de *Template Matching*, para classificação e localização de peças em um campo de trabalho. Tal sistema foi integrado a um braço robótico industrial para a separação e organização dessas peças fisicamente. Conforme citado anteriormente essa aplicação foi validada, concluindo o objetivo geral com sucesso.

De um modo geral sistemas como esse sempre podem ser melhorados e adaptados para outras realidades de aplicação. As técnicas utilizadas sofrem variações de projeto a projeto, sendo utilizadas das mais simples às mais sofisticadas, dependendo do grau de precisão requerido, singularidade dos objetos a serem capturados, ambientes, entre outros fatores. Além disso, melhorias no pré-processamento da imagem podem gerar melhores resultados tanto quanto utilizar técnicas mais complexas.

De qualquer forma, aplicações com a integração desses sistemas contribuem para o crescimento tecnológico e podem melhorar a capacidade de operação em tempo real, virtualizando decisões e orientando os serviços, de acordo com os princípios da indústria 4.0.

O projeto abre uma série de outros possíveis desenvolvimentos vinculados ao mesmo braço robótico utilizado. Como sugestão de futuros trabalhos que envolvam a integração da visão computacional ao braço robótico, a utilização de embarcados, como uma *Raspberry*, para executar o reconhecimento por imagem, a utilização da *Wi-Fi* com *smartphones*, ou mesmo a simplificação utilizando uma câmera USB convencional, são ideias que podem ser utilizadas para as pesquisas em prol do desenvolvimento da estrutura tecnológica disponível.

Variações maiores do sistema também podem gerar boas e diversificadas aplicações. As sugestões nesse sentido são: a ampliação do campo de visão da câmera para a área limite total do braço robótico; variação da quantidade de câmeras presente no sistema; posição da câmera em diagonais ou até na ponta ferramenta, entre outros.

Este trabalho desempenhou bem o seu objetivo de desenvolver um sistema de controle da movimentação de um braço robótico baseado em coordenadas obtidas por visão computacional, de três tipos diferentes de peças em um determinado campo de trabalho, em que foi possível classificar e localizar as peças levando-as a uma posição específica associada ao tipo da peça. Concluindo, a integração entre esses tipos de sistemas pode gerar bons resultados em indústrias, abrindo possibilidades para desenvolvimento de novos trabalhos que desenvolvam essa interação a partir de novas aplicações.

REFERÊNCIAS

AHUJA, Kavita; TULI, Preeti. Object Recognition by Template Matching Using Correlations and Phase Angle Method. **International Journal of Advanced Research in Computer and Communication Engineering** (IJARCCE), Raipur, India, v.3, ed.3, p. 1368-1373, March 2013. Disponível em: https://www.ijarcce.com/upload/2013/march/11-kavita%20ahuja%20-%20object%20recognition-c.pdf. Acesso em: 19 set. 2018.

BRITO, Cândido Regis. **Reconhecimento de Objetos Utilizando Redes Neurais Artificiais e Geometria Fractal.** 2011. Dissertação (Mestrado em Modelagem Computacional e Tecnologia Industrial) - SENAI CIMATEC, Salvador, 2011.

BUSCARIOLLO, Paulo Henrique. **Sistema de Posicionamento Dinâmico Baseado em Visão Computacional e Laser.** 2008. Tese (Doutorado em Engenharia Naval) - Escola Politécnica da Universidade de São Paulo, São Paulo, 2008.

CABRAL, E. L. L. **Análise de Robôs,** Cinemática da Posição de Robôs Manipuladores. São Paulo: USP. *E-book*. Disponível: http://sites.poli.usp.br/p/eduardo.cabral/Cinem%C3%A1tica%20Direta.pdf. Acesso em: 21 mai. 2019.

CARDOSO, Victor Gomes. **Manual de Usuário**. Robô C5G (COMAU). Manual de Instruções - Departamento de Engenharia Mecânica, Universidade Federal de Pernambuco (UFPE), Recife, 2018.

CATTIN, Phillippe. **Digital Image Fundamentals:** Introduction to Signal and Image Processing. Basiléia, Suíça: MIAC/University of Basel, 2016. *E-book*. Disponível em: https://miac.unibas.ch/SIP/pdf/SIP-02-Fundamentals.pdf. Acesso em: 18 nov. 2018.

CHOSET, Howie; LYNCH, Kevin; HUTCHINSON, Seth. **Principles of Robot Motion:** Theory, Algorithms and Implementation. Cambridge, Massachusetts and London, England: A Bradford Book, The MIT Press, 2005.

COLE, Luke; AUSTIN, David; COLE, Lance. **Visual Object Recognition using Template Matching.** Proceedings of Australian Conference on Robotics and Automation, Canberra 2004. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.9548&rep=rep1&type=pdf. Acesso em: 05 jan. 2019.

COLORPIC. Aplicativo para Windows. Disponível em: https://colorpic_1.pt.downloadastro.com/baixar/. Acesso em: 10 jun. 2018.

COMAU. **Uso da Unidade de Controle, Software de Sistema Rel. 1.18:** Acesso ao sistema (ativação, desligamento, login, logout), interface usuário, Terminal de Programação, comandos de sistema, WinC5G, início e fim de ciclo, configuração de E/S, opções de software. Betim: Comau Robotics, 2013.

COMAU. **Treinamento Básico de Robôs C5G.** Betim: Comau Robotics, 2016.

COMAU. Site Oficial COMAU: **Comau SpA VAT no. 00952120012.** Disponível em: https://www.comau.com/PT. Acesso em: 01 abr. 2019.

CORREIA, Diego Anisio. **Sistema de Visão Para Manipuladores Robóticos.** 2013. Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) - Universidade do Vale do Itajaí, São José, 2013.

DROIDCAM, Client. Aplicativo para Windows e Android. Disponível em: http://www.dev47apps.com/droidcam/windows/. Acesso em: 20 set. 2018.

FISHER, R. A. The Use of Multiple Measurements in Taxonomic Problems. **Annals of Human Genetics**, v.7, ed.2, p.179-188, 1936. (Also in Contributions to Mathematical Statistics, John Wiley & Sons, New York, 1950). Disponível em http://syllabus.cs.manchester.ac.uk/pgt/2018/COMP61021/reference/Fisher-DA.pdf. Acesso em: 17 mai. 2019.

GEORGIEVA, Lidiya; DIMITROVA, Tatyana; ANGELOV, Nicola. **RGB And HSV Colour Models In Colour Identification Of Digital Traumas Images.** International Conference on Computer Systems and Technologies – CompSysTech, v.12, p.12.1-12.6, Varna, Bulgaria, Junho 2005. Disponível em: http://ecet.ecs.uni-ruse.bg/cst05/Docs/cp/sV/V.12.pdf. Acesso em: 10 dez. 2019.

GOMES, José; TEIXEIRA, João Marcelo; BARROS, Gutenberg; TEICHRIEB, Veronica. **Controle de Braço Robótico Industrial Através de Miniatura.** 2018. Artigo (Trabalho de Iniciação Científica) - Universidade Federal de Pernambuco (UFPE), Recife, 2018. Disponível em: https://ssl4799.websiteseguro.com/swge5/PROCEEDINGS/PDF/CBA2018-0833.pdf. Acesso em: 20 dez. 2018.

GONZALEZ, Rafael C.; WOODS, Richard E. **Digital Image Processing.** 2. ed., Upper Saddle River, New Jersey: Prentice Hall, 2002.

GREGGIO, N. Real-Time 3D Stereo Tracking And Localizing Of Spherical Objects With The Icub Robotic Platform. **Journal of Intelligent Robotic Systems.** Pontedera, Italy, v.63, ed. 3-4, p. 417-446, September 2011. Disponível em: http://vislab.isr.ist.utl.pt/wp-content/uploads/2012/12/11-jirs-greggio.pdf Acesso em 19 jun. 2018.

GUDLA, Arun Gowtham. **A Methodology to Determine the Functional Workspace Of a 6R Robot Using Forward Kinematics And Geometrical Methods.** 2012. Dissertation (Master of Applied Science) - University of Windsor, Ontário, Canadá, 2012.

JOHNSON, R. A.; WICHERN, D. W. **Applied Mutilvariate Statistical Analysis**. 6. ed., Upper Saddle River, New Jersey: Pearson & Prentice-Hall, 1998.

JUNIOR, Arildo Dirceu Cordeiro. **Sistema de Visão Robótica Aplicado a Tarefas de Manipulação.** 2013. Trabalho de Conclusão de Curso (Graduação em Tecnologia em Mecatrônica Industrial) - Departamentos Acadêmicos de Eletrônica e Mecânica, Universidade Tecnológica Federal do Paraná (UTFPR), Paraná, 2013.

JUNIOR, Julio C. S. Jacques. **Processamento de Imagens:** Fundamentos. Material Didático (Disciplina de Simulação) - Escola Politécnica PUCRS, Rio Grande do Sul, 2010. Disponível em: https://www.inf.pucrs.br/~smusse/Simulacao/PDFs/aula_02_Fundamentos_PI.pdf. Acesso em: 18 jun. 2018.

KELDA, Harmeet Kaur. A Review: Color Models in Image Processing. **International Journal of Computer Technology and Applications.** Amritsar, Pb Índia, v.5, ed.2, p. 319-322, March-April 2014. Disponível em: https://pdfs.semanticscholar.org/08e3/393e627311a204419f3b94f5519958bfab2f.pdf. Acesso em: 13 jan. 2019.

KIRIAN, Geetha; MURALI, S. Automatic Retification of Perspective Distortion From a Single Image Using Plane Homography. **International Journal on Computational Sciences & Applications (IJCSA).** Karnataka, India, v.3, ed.5, p. 47-58, October 2013. Disponível em: https://pdfs.semanticscholar.org/f66a/30a8633ff61aae0293345d6cb273c331fa24.pdf. Acesso em: 05 jun. 2019.

KOGA, Marcelo Li. **Classificadores Bayesianos:** Aplicados a análise sintática da língua portuguesa. 2011. Trabalho Acadêmico (Técnicas de Raciocínio Probabilístico em Inteligência Artificial - Departamento de Engenharia de Computação e Sistemas Digitais - PCS), Escola Politécnica da Universidade de São Paulo, São Paulo, 2011.

LILESAND, T.M.; KIEFER, R.W. **Remote Sensing And Image Interpretation.** 2. ed., New York: Johm Wiley & Sons, 1987.

MACHARET, Douglas G. **Introdução à Robótica:** Cinemática Inversa. 2016. Material Didático (Disciplina de Introdução à Robótica) - Universidade Federal de Minas Gerais, 2016. Disponível em: https://homepages.dcc.ufmg.br/~doug/cursos/lib/exe/fetch.php?media=cursos:introrobotica:2016-1:aula10-cinematica-inversa.pdf. Acesso em: 27 set. 2018.

MAHALAKSHMI, T.; MUTHAIAH, R.; SWAMINATHAN, P. An Overview of Template Matching Technique in Image Processing. **Research Journal of Applied Sciences Engineering and Technology.** Tamil Nadu, India, v.4, ed.24, p. 5469- 5473, December 2012**.** Disponível em: https://pdfs.semanticscholar.org/98f4/8619f4bf1b6dd3eabcf06d4c5f5fec75f1ca.pdf. Acesso em: 04 fev. 2019.

MAINTZ, Twan. **Digital and Medical Image Processing.** Utrecht, Netherland: UU.nl, 2005. *E-book*. Disponível em: http://www.cs.uu.nl/docs/vakken/ibv/reader/readerINFOIBV.pdf. Acesso em: 03 fev. 2019.

MARTINS, Samuel Botter. **Introdução ao Processamento Digital de Imagens:** Definições Básicas, Espaço de Cores e Histogramas. São Paulo: Instituto de Computação/UNICAMP, 2010. *E-book*. Disponível em: https://www.ic.unicamp.br/~ra144681/misc/files/ApostilaProcDeImagesParteI.pdf. Acesso em: 24 ago. 2018.

MILLER, Gregory. **Lowering False Positive Detection Rates Using Multiple Haar Classifiers.** Research Project (Introduction to Artificial Intelligence) – Departament of Computer Science, Florida State University, Tallahessee. Disponível em: http://www.cs.fsu.edu/~cop4601p/project/students/gregory-milner/HaarClassifiers_milner.pdf Acesso em: 20 dez. 2019.

NASCIMENTO, Daniel Lemos de Almeida; SANTOS, Isabella Cristina Campos; SORRENTINO, Nathan Matta. **Visão Computacional Aplicada A Robôs Autônomos de Baixo Custo.** 2016. Trabalho de Conclusão de Curso (Graduação em Sistema de Informação) - Instituto Federal Fluminense, Campos dos Goytacazes, Rio de Janeiro, 2016.

NEHEMY, Gabriella Furlan; GONÇALVES, Paulo José Paupitz. **Uso de Visão Computacional para Controle de Protótipo de Braço.** 2014. Trabalho Acadêmico-Artigo (Congresso de Iniciação Científica da Unesp - Graduação em Engenharia Mecânica) - Faculdade de Engenharia de Bauru, São Paulo, 2014.

OLIVEIRA, Luiz Eduardo. **Processamento de Imagens:** Morfologia Matemática Binária. Material Didático (Processamento de Imagem) - Departamento de Informática, Universidade Federal do Paraná (UFPR), Paraná. Disponível em: http://www.inf.ufpr.br/lesoliveira/download/morfologia.pdf. Acesso em: 11 fev. 2019.

OLIVEIRA, Nádia Raquel; RODRIGUES, Hernandes Erick; SANTOS, Flávio Alves; TAVARES, Francisco Machezan; ALMEIDA, Jhoisnáyra Vitória; ROCHA, Francisco Vinícius; FILHO, Antônio Edson; ARAÚJO, Francisco Marcelino. **Prototipação de um Braço Robótico Controlado Remotamente com Processamento Digital de Imagens para Detecção e Manuseio de Objetos.** Artigo (Mostra Nacional de Robótica - MNR) - Departamento de Indústria, Segurança e Produção Cultural – Instituto Federal do Piauí, Piauí.

OPENCV. **Opencv Documetation Python Tutorials.** Disponível: https://docs.opencv.org/3.0- beta/do c/ py_tutorials/py_tutorials.html. Acesso em 02 jul. 2017.

ORMINDO, Tamara do Vale; ALVES, Rafael Carlos Nogueira; FRAGOSO, Paulo Eduardo, VIDAL, Leonardo Carvalho. **Aplicações de Robôs Industriais com Garras Mecânicas.** Trabalho Acadêmico (Graduação de Engenharia de Produção Automotiva). Faculdade de Engenharia de Resende, Rio de Janeiro.

PALMA, J. M. B.; BUENO, U. S.; STOROLLI, W. G.; SCHIAVUZZO, P. L.; CESAR, F. I. G.; MAKIYA, I. K. **Os Princípios da Indústria 4.0 e os Impactos na Sustentabilidade da Cadeia de Valor Empresarial.** International Workshop Advances in Cleaner Production. São Paulo, ed. 6, Maio 2017. Disponível em: http://www.advancesincleanerproduction.net/sixth/files/sessoes/6B/4/palma_jmb_et_al_academic.pdf. Acesso em: 25 mai. 2019.

PEREIRA, Rafael Cardoso. **Técnica de Rastreamento e Perseguição de Alvo Utilizando o Algoritmo Haar Cascade Aplicada a Robôs Terrestres com Restrições de Movimento.** 2017. Dissertação (Mestrado em Engenharia Mecânica) - Universidade Federal do Rio Grande do Norte, Natal, 2017.

PESTANA, J.; LOPEZ, Jose Luis Sanchez. Computer Vision Based General Object Following For GPS-Denied Multirotor Unmanned Vehicles. **American Control Conference.** Portland, OR, USA, June 2014. Disponível em: https://www.researchgate.net/publication/269293737_Computer_vision_based_general_object_following_for_GPS-denied_multirotor_unmanned_vehicles. Acesso em: 15 nov. 2018.

PRATT, W. K. **Digital Image Processing,** PICKS Scientific Inside. 4. ed., New York: A John Wiley & Sons, Inc., Publication, 2007.

QUEIROZ, José E. R.; GOMES, Herman M.. Introdução ao Processamento Digital de Imagens. **Revista de Informática Teórica e Aplicada – RITA**. Rio Grande do Sul, Brasil, v.8, ed.1, p.1-31, 2001. Disponível em: http://www.dsc.ufcg.edu.br/~hmg/disciplinas/graduacao/vc-2014.1/Rita-Tutorial-PDI.pdf. Acesso em: 27 nov. 2018.

REZAEI, Mahdi. **Creating a Cascade of Haar-Like Classifiers:** Step by Step. 2015. Tutorial (Cascade of Classifiers Tutorial) - Department of Computer Science, the University of Auckland, 2015. Disponível em: https://www.cs.auckland.ac.nz/~m.rezaei/Tutorials/Creating_a_Cascade_of_Haar-Like_Classifiers_Step_by_Step.pdf. Acesso em 3 jul. 2018.

RHODY, Harvey. **Basic Morphological Image Processing.** 2005. Material Didático (Digital Image Processing Lecture) - Center for Imaging Science, Rochester Institute of Technology, Rochester, Nova York, 2005. Disponível em: https://www.cis.rit.edu/class/simg782/lectures/lecture_03/lec782_05_03.pdf. Acesso em: 14 set. 2018.

RIOS, Marcel Leite. **Visão Computacional Aplicada ao Monitoramento de Robôs Móveis em Cenários de Robótica Educacional.** 2017. Dissertação (Pós-Graduação em Informática) - Instituto de Computação, Universidade Federal do Amazonas, Manaus, 2017.

RIPLEY, B D. **Pattern Recognition And Neural Networks.** 1. ed., New York, USA: Cambridge University, Press, 2008.

ROCHA, Carlos Diego França. **Aplicação do Algoritmo Haar Cascade em um Sistema Embarcado para Detecção de Ovos do Mosquito Aedes Aegypti em Palhetas de Ovitrampas.** 2018. Trabalho de Conclusão de Curso (Graduação em Análise e

Desenvolvimento de Sistemas) - Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte (IFRN), 2018.

SANTOS, Eduardo. **O Uso de Visão Computacional Para o Controle De Um Manipulador Robótico.** Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) - Universidade Tecnológica Federal do Paraná (UTFPR), Pato Branco, Paraná, 2014.

SANTOS, Rosângela Leal; OHATA, Arlete Tieko; QUINTANILHA, José Alberto. **A Análise Bayesiana na Classificação Supervisionada de Imagens:** Aplicação na Determinação de Classes de Uso do Solo (Mogi das Cruzes - SP). Simpósio Brasileiro de Sensoriamento Remoto – SBSR. Belo Horizonte, Brasil, ed.6, p. 2119-2122, Abril 2003. Disponível em: http://marte.sid.inpe.br/attachment.cgi/ltid.inpe.br/sbsr/2002/11.18.00.34/doc/15_363.pdf. Acesso em: 24 jan. 2019.

SCHOWENGERDT, R. **Techniques for Image Processing and Classification in Remote Sensing**. 3. ed., London, United Kingdom: Academic Press, 2000.

SCURI, Antonio Escaño. **Fundamentos da Imagem Digital.** Rio de Janeiro: Tecgraf/PUC-Rio, 2002. *E-book*. Disponível em: https://webserver2.tecgraf.puc-rio.br/~scuri/download/fid.pdf. Acesso em: 2 abr. 2019.

SKARBEK, Wladyslaw; KOSCHAN, Andreas. **Colour Image Segmentation**: A Survey. Technischer Bericht 94-32 (Technische Universit at Berlin) – Berlin, Deutschland, 1994. Disponível em: https://pdfs.semanticscholar.org/12b7/bf3f137fb06c87952a8ac6d1d490c35f5c81.pdf Acesso em: 17 mar. 2019.

SHUHUA, Li; GAIZHI, Guo. The application of improved HSV color space model in image processing. **International Conference: Future Computer and Communication (ICFCC).** Wuhan, China, v.2, ed.2, 2010.

TEODORO, Edivaldo. Redes Neurais. **Network Technologies, Nova Odessa.** São Paulo, v.1-2, ed.1-2, p. 57-100, 2002-2003. Disponível em: http://www.luzimarteixeira.com.br/wp-content/uploads/2009/09/redes-neurais.pdf. Acesso em: 10 mai. 2019.

VIJAYARANI, S.; SAKILA, A. Template Matching Technique for Searching Words in Document Images. **International Journal on Cybernetics & Informatics (IJCI).** Coimbatore, India, v.4, ed.6, p. 25-35, December 2015. Disponível em: http://aircconline.com/ijci/V4N6/4615ijci03.pdf. Acesso em: 30 abr. 2019.

VIOLA, Paul; JONES, Michael. Rapid Object Detection using a Boosted Cascade of Simple Features. **Conference on Computer Vision And Pattern Recognition**. Mitsubishi Electric Research Labs. Cambridge, Massachusetts, 2011.

APÊNDICE A - COORDENADAS HSV VERMELHA DO CAMPO

VERMELHO								
H (*Hue*)			S (*Saturation*)			V (*Value*)		
1ª	2ª	3ª	1ª	2ª	3ª	1ª	2ª	3ª
3	8	2	196	164	151	159	165	136
5	4	1	195	183	167	136	155	160
5	2	7	203	175	171	148	160	168
8	4	10	190	176	165	158	161	137
8	10	3	186	161	173	148	169	148
9	4	1	202	174	153	160	163	161
2	2	11	187	173	139	131	164	159
3	7	4	153	170	180	148	172	118
5	3	3	185	173	173	143	159	149
6	4	5	209	168	172	150	168	174
8	6	9	195	173	173	156	165	180
4	3	2	146	171	167	145	171	179
3	7	4	204	170	169	116	167	167
3	8	4	161	164	147	141	168	158
7	2	1	182	173	173	133	170	160
7	3	5	139	169	175	165	171	175

LIMITE INFERIOR		
H	S	V
1	139	116

LIMITE SUPERIOR		
H	S	V
11	209	179

1ª Coleta às 17:30.

2ª Coleta às 18:00.

3ª Coleta às 18:30.

APÊNDICE B - COORDENADAS HSV AZUL DO CAMPO

AZUL								
H (*Hue*)			S (*Saturation*)			V (*Value*)		
1ª	2ª	3ª	1ª	2ª	3ª	1ª	2ª	3ª
223	232	218	161	166	213	92	89	102
220	243	218	210	135	209	107	85	106
212	233	220	223	164	191	110	95	72
215	237	210	222	166	141	108	83	65
217	233	224	217	181	210	108	86	80
223	237	220	217	172	198	88	80	94
229	228	229	202	149	191	91	111	107
225	240	226	148	150	202	81	78	63
225	233	224	236	185	209	80	84	89
220	237	218	214	168	220	105	82	87
218	233	215	208	149	255	126	82	41
220	229	220	235	143	223	51	102	88
223	234	210	142	154	133	81	95	92
196	236	221	167	160	80	81	81	82
218	237	223	158	181	201	116	85	89
206	233	220	95	168	202	126	82	92

LIMITE INFERIOR		
H	S	V
210	80	*41*

LIMITE SUPERIOR		
H	S	V
243	236	126

1ª Coleta às 17:30.

2ª Coleta às 18:00.

3ª Coleta às 18:30.

APÊNDICE C - COORDENADAS HSV AMARELA DO CAMPO

AMARELO								
H (*Hue*)			S (*Saturation*)			V (*Value*)		
1ª	2ª	3ª	1ª	2ª	3ª	1ª	2ª	3ª
61	68	54	136	189	120	173	169	182
57	70	52	105	171	100	182	167	183
57	70	56	126	186	132	182	177	178
64	68	52	90	177	100	189	171	184
59	69	55	133	198	124	175	173	183
59	68	50	124	179	133	189	174	180
58	68	54	104	191	116	182	172	184
52	70	53	47	153	49	175	180	182
70	68	60	52	178	72	175	170	177
61	67	55	117	171	118	165	169	193
65	68	50	67	187	81	186	175	189
58	68	54	89	162	104	178	173	189
55	67	52	118	123	100	169	189	183
51	65	56	104	177	94	199	169	188
47	69	53	93	144	118	180	180	177
56	67	55	137	157	106	177	171	187

LIMITE INFERIOR		
H	S	V
47	47	165

LIMITE SUPERIOR		
H	S	V
70	198	199

1ª Coleta às 17:30.

2ª Coleta às 18:00.

3ª Coleta às 18:30.

APÊNDICE D - COORDENADAS HSV VERDE DO CAMPO

VERDE								
H (*Hue*)			S (*Saturation*)			V (*Value*)		
1ª	2ª	3ª	1ª	2ª	3ª	1ª	2ª	3ª
90	79	77	87	82	102	177	176	177
83	76	80	86	95	126	182	168	180
88	75	83	93	90	129	165	157	182
83	87	79	91	76	130	157	178	177
86	98	88	85	90	149	179	163	173
89	81	82	98	85	121	174	180	188
90	77	79	83	86	114	166	158	179
77	82	80	96	100	128	183	194	169
80	73	83	101	92	131	176	172	196
82	90	83	96	119	130	164	164	180
84	75	77	93	85	102	181	176	177
81	73	83	91	79	140	159	168	171
90	73	79	86	110	85	195	179	165
89	81	79	89	103	95	188	164	158
86	86	80	77	111	98	177	187	177
87	75	76	86	97	79	173	176	177

LIMITE INFERIOR		
H	S	V
73	77	157

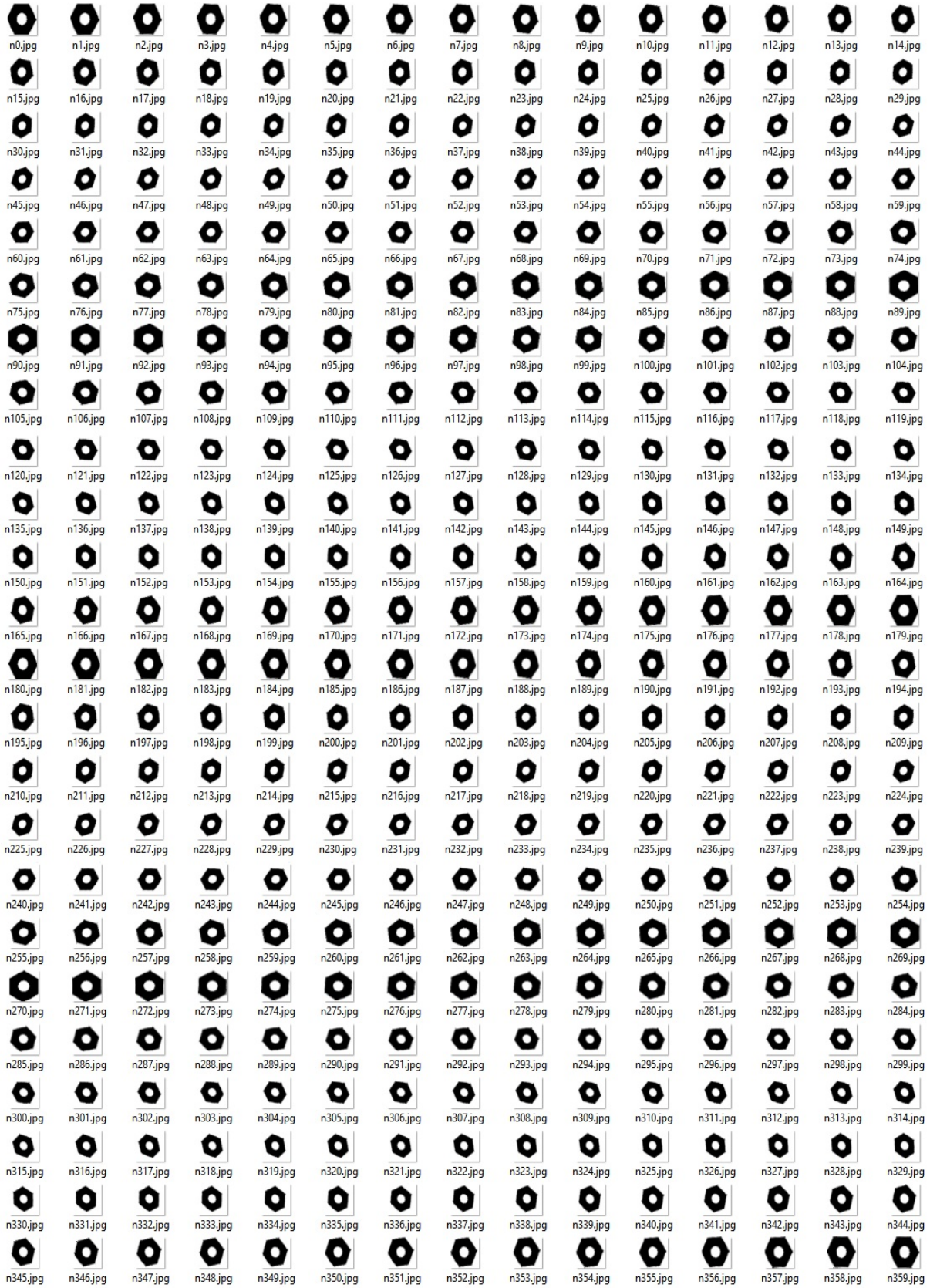
LIMITE SUPERIOR		
H	S	V
98	149	196

1ª Coleta às 17:30.

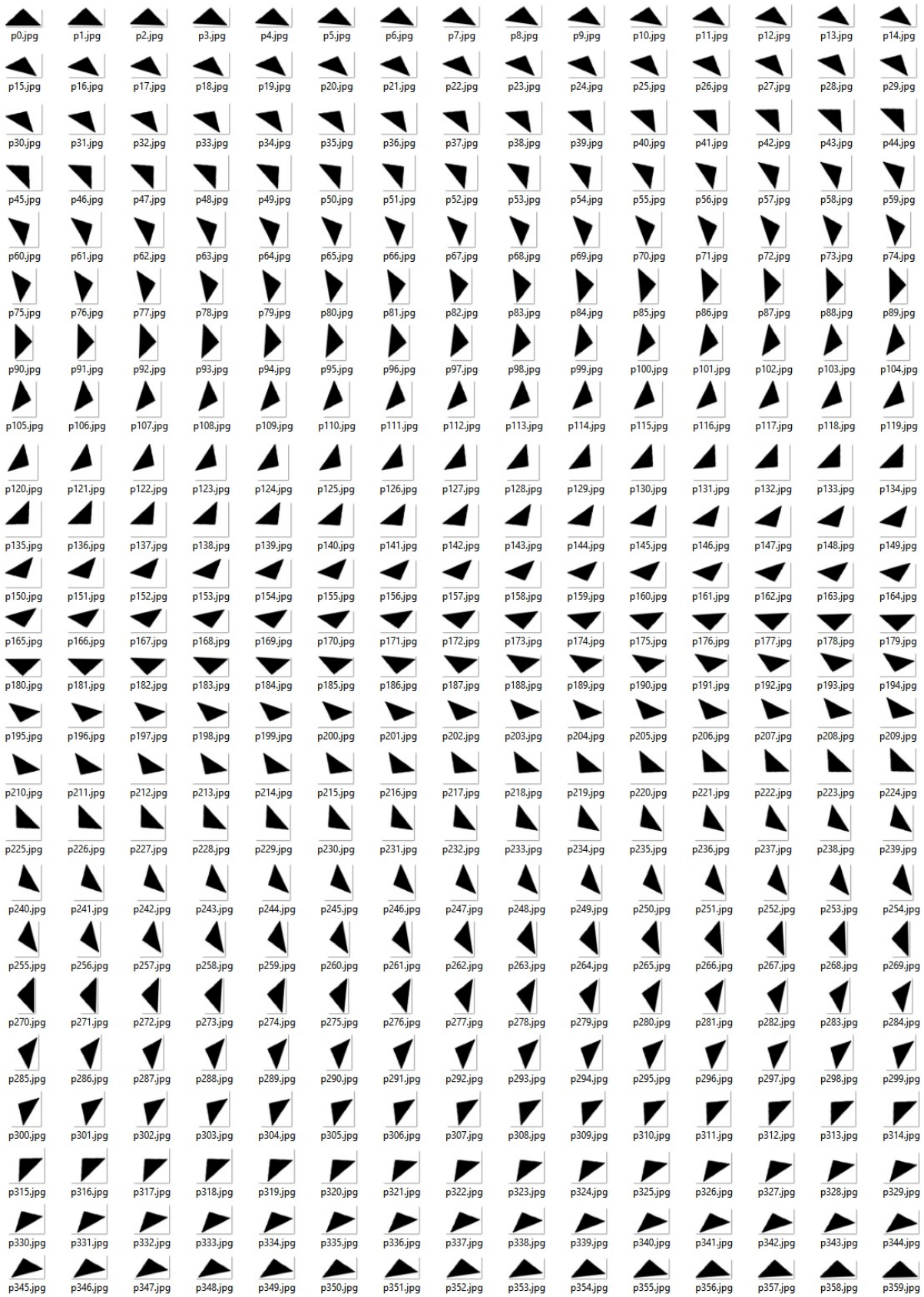
2ª Coleta às 18:00.

3ª Coleta às 18:30.

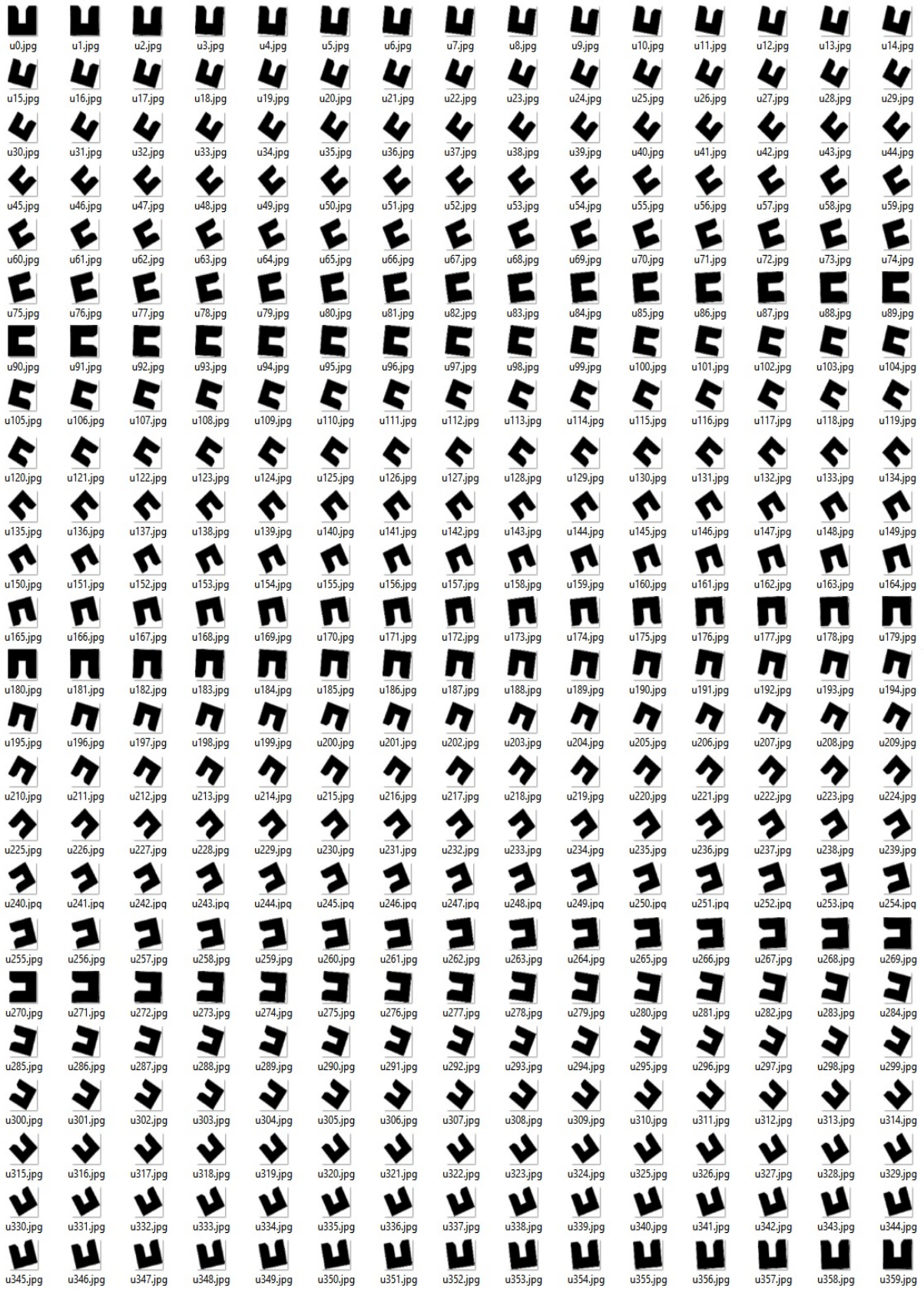
APÊNDICE E - TEMPLATES DA PEÇA TIPO PORCA



APÊNDICE F - TEMPLATES DA PEÇA TIPO PRISMA TRIANGULAR



APÊNDICE G - TEMPLATES DA PEÇA TIPO U



APÊNDICE H - CÓDIGO EM PYTHON PARA CLASSIFICAÇÃO DAS PEÇAS

```
def set_the_camera_from_usb():
        print "\n\n\n   [i_camera_setting]"
        # print "\n [i]: Set the camera from USB and press 'q' !!"
        cap = cv2.VideoCapture(0)
        wait = time.time()+10
        n = 0
        while(n != 1):
                ret, frame = cap.read()
                rotate_img = rotate(frame,-90)
                frame = imutils.resize(rotate_img, 635, 411)
                #frame = cv2.flip(frame, 1)
                cv2.imshow("i.set_the_camera_from_usb", frame)
                if (cv2.waitKey(100) & 0xFF == ord('q')) or (time.time() > wait):
                        n = 1
                        cv2.destroyAllWindows()
                        break

        print "\n       [i]: Image has been defined!!"
        return frame


def vector_field(img):
        print "\n\n\n   [ii.identify_field]\n"
        error_contour = 0
        error_area = [1,1,1,1]
        error_div = [1,1,1,1]
        # Boundaries list of HSV colors
        boundaries = [
        ([1/2, 139, 116], [11/2, 209, 179]),            # red
        ([210/2, 80, 41], [243/2, 236, 126]),           # blue
         ([47/2, 47, 165], [70/2, 198, 199]),           # yellow
        ([73/2, 77, 157], [98/2, 149, 196])]            # green
        image = img
```

```
        hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        # create NumPy arrays from the boundaries
        lower_red = np.array(boundaries[0][0], dtype = "uint8")         #red
        upper_red = np.array(boundaries[0][1], dtype = "uint8")         #red
        lower_blue = np.array(boundaries[1][0], dtype = "uint8")        #blue
        upper_blue = np.array(boundaries[1][1], dtype = "uint8")        #blue
        lower_yellow = np.array(boundaries[2][0], dtype = "uint8")      #yellow
        upper_yellow = np.array(boundaries[2][1], dtype = "uint8")      #yellow
        lower_green = np.array(boundaries[3][0], dtype = "uint8")       #yellow
        upper_green = np.array(boundaries[3][1], dtype = "uint8")       #yellow

        # find colors within the specified boundaries and apply the mask
        mask_red = cv2.inRange(hsv_image, lower_red, upper_red)
        mask_blue = cv2.inRange(hsv_image, lower_blue, upper_blue)
        mask_yellow = cv2.inRange(hsv_image, lower_yellow, upper_yellow)
        mask_green = cv2.inRange(hsv_image, lower_green, upper_green)

        # apply the mask in real image
        output_red = cv2.bitwise_and(hsv_image, hsv_image, mask = mask_red)
        output_blue = cv2.bitwise_and(hsv_image, hsv_image, mask = mask_blue)
        output_yellow = cv2.bitwise_and(hsv_image, hsv_image, mask = mask_yellow)
        output_green = cv2.bitwise_and(hsv_image, hsv_image, mask = mask_green)

        # find contours in each output
        img_red, contours_red, hierarchy_red =
cv2.findContours(mask_red,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)
        img_blue, contours_blue, hierarchy_blue =
cv2.findContours(mask_blue,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)
        img_yellow, contours_yellow, hierarchy_yellow =
cv2.findContours(mask_yellow,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)
        img_green, contours_green, hierarchy_green =
cv2.findContours(mask_green,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)

        if contours_red != []:
                for C_red in contours_red:
```

```
                    if cv2.contourArea(C_red) >= 10:
                        error_area[0] = 0; M_red = cv2.moments(C_red)
                        if M_red["m00"] != 0:
                            error_div[0] = 0
                            cx_red = int(M_red["m10"] / M_red["m00"]); cy_red =
int(M_red["m01"] / M_red["m00"])
                            w_red = h_red =
int(math.sqrt(cv2.contourArea(C_red))) + 1
                            cv2.circle(image, (cx_red-w_red/2,cy_red+h_red/2), 3,
(0,0,255), 5)
                            cv2.rectangle(image, (cx_red-w_red/2,cy_red-
h_red/2),(cx_red+w_red/2,cy_red+h_red/2), (0,0,255), 2)
        else: error_contour = 1; print "          [ii]: red []"

        if contours_blue != []:
            for C_blue in contours_blue:
                if cv2.contourArea(C_blue) >= 100:
                    error_area[1] = 0; M_blue = cv2.moments(C_blue)
                    if M_blue["m00"] != 0:
                        error_div[1] = 0
                        cx_blue = int(M_blue["m10"] / M_blue["m00"]);
cy_blue = int(M_blue["m01"] / M_blue["m00"])
                        w_blue = h_blue =
int(math.sqrt(cv2.contourArea(C_blue))) + 1
                        cv2.circle(image,
(cx_blue+w_blue/2,cy_blue+h_blue/2), 3, (255,0,0), 5)
                        cv2.rectangle(image, (cx_blue-w_blue/2,cy_blue-
h_blue/2),(cx_blue+w_blue/2,cy_blue+h_blue/2), (255,0,0), 2)
        else: error_contour = 1; print "          [ii]: blue []"

        if contours_yellow != []:
            for C_yellow in contours_yellow:
                if cv2.contourArea(C_yellow) >= 100:
                    error_area[2] = 0; M_yellow = cv2.moments(C_yellow)
                    if M_yellow["m00"] != 0:
                        error_div[2] = 0
```

```
                                        cx_yellow = int(M_yellow["m10"] / M_yellow["m00"]);
cy_yellow = int(M_yellow["m01"] / M_yellow["m00"])
                                        w_yellow = h_yellow =
int(math.sqrt(cv2.contourArea(C_yellow))) + 1
                                        cv2.circle(image, (cx_yellow-w_yellow/2,cy_yellow-
h_yellow/2), 3, (0,255,255), 5)
                                        cv2.rectangle(image, (cx_yellow-
w_yellow/2,cy_yellow-h_yellow/2),(cx_yellow+w_yellow/2,cy_yellow+h_yellow/2),
(0,255,255), 2)
        else: error_contour = 1; print "        [ii]: yellow []"


        if contours_green != []:
                for C_green in contours_green:
                        if cv2.contourArea(C_green) >= 50:
                                error_area[3] = 0; M_green = cv2.moments(C_green)
                                if M_green["m00"] != 0:
                                        error_div[3] = 0
                                        cx_green = int(M_green["m10"] / M_green["m00"]);
cy_green = int(M_green["m01"] / M_green["m00"])
                                        w_green = h_green =
int(math.sqrt(cv2.contourArea(C_green))) + 1
                                        cv2.circle(image, (cx_green+w_green/2,cy_green-
h_green/2), 3, (0,255,0), 5)
                                        cv2.rectangle(image, (cx_green-w_green/2,cy_green-
h_green/2),(cx_green+w_green/2,cy_green+h_green/2), (0,255,0), 2)
        else: error_contour = 1; print "        [ii]: green []"


        if error_contour==0 and max(error_area)==0 and max(error_div)==0:
                vector = [(cx_red-w_red/2,cy_red+h_red/2),
(cx_blue+w_blue/2,cy_blue+h_blue/2), (cx_yellow-w_yellow/2,cy_yellow-h_yellow/2),
(cx_green+w_green, cy_green-h_green)]
                print "\n        [ii]: Field has been defined!!"
        else:
                vector = []
                print "  [ii]: error_area =", error_area
                print "  [ii]: error_div =", error_div


        cv2.imshow("ii.identify_field", image)
```

```
        cv2.waitKey(2500)
        cv2.destroyAllWindows()
        return vector

def perspective_correction(image,vector):
        print "\n\n   [iii_homography]"
        size = (635, 411, 3)
        im_dst = np.zeros(size, np.uint8)
        pts_dst = np.array( [[0,0],
                                        [size[0] - 1, 0],
                                        [size[0] - 1, size[1] -1],
                                        [0, size[1] - 1 ]
                                        ], dtype=float
                                        )
        pts_src = np.array( [
                                        [vector[2][0], vector[2][1]],
                                        [vector[3][0], vector[3][1]],
                                        [vector[1][0], vector[1][1]],
                                        [vector[0][0], vector[0][1]]
                                        ], dtype=float
                                        )
        h, status = cv2.findHomography(pts_src, pts_dst)
        im_dst = cv2.warpPerspective(image, h, size[0:2])

        print "\n          [iii]: Homography has been done!"
        cv2.imshow("[iii] homography", im_dst)
        cv2.waitKey(1500)
        cv2.destroyAllWindows()
        if cv2.waitKey(0) & 0xFF == ord('q'):
                cv2.destroyAllWindows()


        return im_dst

def vector_object_template(image):
```

```
        print "\n\n\n   [iv_identify_object]\n"
        ANG = 0; nut = []; prism = []; u=[]; k = 0
        vector_nut = []; vector_prism = []; vector_u = []
        res_nut = []; res_prism = []; res_u = []
        compare_nut = []; compare_prism = []; compare_u = []
        max_val_nut = []; max_val_prism = []; max_val_u = []
        inside = [0,0,0]

        resized = imutils.resize(image, 635, 411)
        gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
        blur = cv2.GaussianBlur(gray, (5, 5), 20)
        _, thresh = cv2.threshold(blur,50,255,cv2.THRESH_TOZERO+cv2.THRESH_OTSU)
#cv2.THRESH_TRUNC
        # # teste = cv2.adaptiveThreshold(blur,255,cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY,11,2)
        # plt.hist(resized.ravel(),256,[0,256]); plt.show()

        while (k<=358):
                k=k+1
                nut.append(cv2.imread("templates/ima_nut/n%d.jpg" %k,0))
                prism.append(cv2.imread("templates/ima_prism/p%d.jpg" %k,0))
                u.append(cv2.imread("templates/ima_u/u%d.jpg" %k,0))

        methods = ['cv2.TM_CCORR_NORMED'] # others in the end
        method = eval(methods[0])

        # # Apply template Matching
        print "  [iv]: Searching for objects...\n"
        for a in range (359):
                res_nut.append(cv2.matchTemplate(thresh,nut[a],method))
                res_prism.append(cv2.matchTemplate(thresh,prism[a],method))
                res_u.append(cv2.matchTemplate(thresh,u[a],method))

        for b in range (len(res_nut)):
                # (min_val, max_val, min_loc, max_loc)
```

```
            compare_nut.append(cv2.minMaxLoc(res_nut[b]))
            compare_prism.append(cv2.minMaxLoc(res_prism[b]))
            compare_u.append(cv2.minMaxLoc(res_u[b]))

        for c in range (len(compare_nut)):
            max_val_nut.append(compare_nut[c][1])
            max_val_prism.append(compare_prism[c][1])
            max_val_u.append(compare_u[c][1])

        for d in range (len(max_val_nut)):
            if max(max_val_nut)==max_val_nut[d] and max(max_val_nut) >= 0.95:
#100000.0:
                inside[0] = 1
                w_nut, h_nut = nut[d].shape[::-1]
                top_left_nut = compare_nut[d][3]
                bottom_right_nut = (top_left_nut[0]+w_nut, top_left_nut[1]+h_nut)

        cv2.rectangle(image,(top_left_nut[0],top_left_nut[1]),(bottom_right_nut[0],bottom_ri
ght_nut[1]),(0,255,0),1)

        cv2.putText(image,'porca',(top_left_nut[0],top_left_nut[1]+70),cv2.FONT_HERSHE
Y_SIMPLEX,0.50,(0,0,0),1,cv2.LINE_AA)
                TYPE=1;ANG = d #; cx=top_left_nut[0]+w_nut/2;
cy=top_left_nut[1]+h_nut/2
                crop_nut =
thresh[top_left_nut[1]:bottom_right_nut[1],top_left_nut[0]:bottom_right_nut[0]]
                img_nut, contours_nut, hierarchy_nut =
cv2.findContours(crop_nut,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)
                for C_nut in contours_nut:
                    M_nut = cv2.moments(C_nut)
                    if M_nut["m00"] != 0:
                        cx_nut = int(M_nut["m10"] / M_nut["m00"]); cy_nut =
int(M_nut["m01"] / M_nut["m00"])
                        # cv2.circle(image,
(top_left_nut[0]+w_nut/2,top_left_nut[1]+h_nut/2), 3, (0,0,255), 5)
                        cv2.circle(image,
(cx_nut+top_left_nut[0],cy_nut+top_left_nut[1]), 3, (0,120,255), 5)
                        break
```

```
                print "  [iv]: vector_porca =",
[cx_nut+top_left_nut[0],cy_nut+top_left_nut[1], ANG, TYPE]
                vector_nut = [cx_nut+top_left_nut[0],cy_nut+top_left_nut[1], ANG,
TYPE]


            if max(max_val_prism)==max_val_prism[d] and max(max_val_prism) >=
0.9500: #10000.0:
                inside[1] = 1
                w_prism, h_prism = prism[d].shape[::-1]
                top_left_prism = compare_prism[d][3]
                bottom_right_prism = (top_left_prism[0]+w_prism,
top_left_prism[1]+h_prism)

        cv2.rectangle(image,(top_left_prism[0],top_left_prism[1]),(bottom_right_prism[0],bot
tom_right_prism[1]),(0,255,0),1)
                # cv2.circle(image,
(top_left_prism[0]+w_prism/2,top_left_prism[1]+h_prism/2), 3, (0,0,255), 5)
                cv2.putText(image,'prisma',(top_left_prism[0]-
10,top_left_prism[1]+80),cv2.FONT_HERSHEY_SIMPLEX,0.50,(0,0,0),1,cv2.LINE_AA)
                TYPE=2; ANG = d; #cx=top_left_prism[0]+w_prism/2;
cy=top_left_prism[1]+h_prism/2
                crop_prism =
thresh[top_left_prism[1]:bottom_right_prism[1],top_left_prism[0]:bottom_right_prism[0]]
                img_prism, contours_prism, hierarchy_prism =
cv2.findContours(crop_prism,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)
                for C_prism in contours_prism:
                        M_prism = cv2.moments(C_prism)
                        if M_prism["m00"] != 0:
                                cx_prism = int(M_prism["m10"] / M_prism["m00"]);
cy_prism = int(M_prism["m01"] / M_prism["m00"])
                                # cv2.circle(image,
(top_left_prism[0]+w_nut/2,top_left_prism[1]+h_nut/2), 3, (0,0,255), 5)
                                cv2.circle(image,
(cx_prism+top_left_prism[0],cy_prism+top_left_prism[1]), 3, (0,120,255), 5)
                                break
                print "  [iv]: vector_prism =",
[cx_prism+top_left_prism[0],cy_prism+top_left_prism[1], ANG, TYPE]
                vector_prism =
[cx_prism+top_left_prism[0],cy_prism+top_left_prism[1], ANG, TYPE]
```

```
            if max(max_val_u)==max_val_u[d] and max(max_val_u) >= 0.850:
                inside[2] = 1
                w_u, h_u = u[d].shape[::-1]
                top_left_u = compare_u[d][3]
                bottom_right_u = (top_left_u[0]+w_u, top_left_u[1]+h_u)

    cv2.rectangle(image,(top_left_u[0],top_left_u[1]),(bottom_right_u[0],bottom_right_u[
1]),(0,255,0),1)

    cv2.putText(image,'u',(top_left_u[0],top_left_u[1]+70),cv2.FONT_HERSHEY_SIMP
LEX,0.50,(0,0,0),1,cv2.LINE_AA)
                TYPE=3; ANG = d;
                cx=top_left_u[0]+w_u/2; cy=top_left_u[1]+h_u/2
                crop_u =
thresh[top_left_u[1]:bottom_right_u[1],top_left_u[0]:bottom_right_u[0]]
                img_u, contours_u, hierarchy_u =
cv2.findContours(crop_u,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)
                for C_u in contours_u:
                    M_u = cv2.moments(C_u)
                    if M_u["m00"] != 0:
                        cx_u = int(M_u["m10"] / M_u["m00"]); cy_u =
int(M_u["m01"] / M_u["m00"])
                        # cv2.circle(image,
(top_left_u[0]+w_u/2,top_left_u[1]+h_u/2), 3, (0,0,255), 5)
                        cv2.circle(image,
(cx_u+top_left_u[0],cy_u+top_left_u[1]), 3, (0,120,255), 5)
                        break
                print "  [iv]: vector_u =", [cx_u+top_left_u[0],cy_u+top_left_u[1],
ANG, TYPE]
                vector_u = [cx_u+top_left_u[0],cy_u+top_left_u[1], ANG, TYPE]

    if vector_nut == []: print "     [iv]: Nothing detected in PORCA template matching:
vector_porca = ()"
    if vector_prism == []: print "  [iv]: Nothing detected in PRISMA template matching:
vector_prism = ()"
    if vector_u == []: print "       [iv]: Nothing detected in u template matching: vector_u
= ()"
    print "\n"
```

```
        if vector_nut != [] and inside[0]==1:
                vector = vector_nut

        elif vector_prism != [] and inside[1]==1:
                vector = vector_prism

        elif vector_u != [] and inside[2]==1:
                vector = vector_u

        else:
                vector = []

        print "  [iv]: vector =", vector

        cv2.imshow('[iv] identify_object - TEMPLATE MATCHING', image)
        cv2.waitKey(1500)
        cv2.destroyAllWindows()
        # if cv2.waitKey(1) & 0xFF == ord('q'):
        #         cv2.destroyAllWindows()
        #         cv2.waitKey(0)
        return vector

        ANG = 0; nut = []; prism = []; u=[]; d = 0
        vector_nut = []; vector_prism = []; vector_u = []
        res_nut = []; res_prism = []; res_u = []
        compare_nut = []; compare_prism = []; compare_u = []
        max_val_nut = []; max_val_prism = []; max_val_u = []
        top_left_nut = 0; top_left_prism = 0; top_left_u = 0
        bottom_right_nut = 0; bottom_right_prism = 0; bottom_right_u = 0

def convert_vector_object(vector):
        converted_vector_object = vector

        # X=635 mm  Y=411mm
```

```
        if vector != []:
                k = 635/635 # mm/pixel # denominador definido com base
                h = 411/411     # mm/pixel      # no que foi definido em ii

                converted_vector_object[0] = vector[0] * k
                converted_vector_object[1] = 411 - (vector[1] * h)

                if converted_vector_object[2] > 180:
                        converted_vector_object[2] = -1*(converted_vector_object[2] - 180)
                else:
                        converted_vector_object[2] = -1*converted_vector_object[2]
        else:
                print "  [iv]: vector = [] (convert_vector_object)"
        return converted_vector_object

def send_to_pdl2(vector):
        print "\n\n   [v_socket]\n"
        serverHost = '192.168.29.2' #IP Server #'localhost'
        serverPort = 1214
        error = 1
        # Criamos o socket e o conectamos ao servidor
        print "  [v]: socket.py connecting...\n\n"
        while(error == 1):
                error = 0
                sockobj = socket(AF_INET, SOCK_STREAM)
                try:
                        sockobj.connect((serverHost, serverPort))
                except:
                        error = 1
                        print "  [v]: Connection failed \n\n"
                        print "  [v]: Trying again... \n\n"

                if error == 0:
                        data = struct.pack('iiii',vector[0], vector[1], vector[2], vector[3])
                        sockobj.sendall(data)
```

```
                    time.sleep(10)
                    print "  [v]: Sent to PDL2:", vector[0], vector[1], vector[2], vector[3]


                    # Espera uma resposta do servidor
                    data = sockobj.recv(4)
                    unpacked_data = struct.unpack('i', data)
                    print "  [v]: Received from PDL2: ", unpacked_data

                    # Fechamos a conexao
                    sockobj.close()

        return True

print "\n [MAIN]"
while(vector_field == []):
        ## Set the Moto G1 camera
        image = i.set_the_camera_from_usb()
        #image = i.set_the_camera_from_wifi()
        # image = cv2.imread("img_for_test/CAMPO CERTO/teste (%d).jpg" %n)
#PORCA+PRISMA+U/

        ## Find the user workspace    # yellow,red,blue,green
        vector_field = ii.vector_field(image)
        if vector_field == []: print "\n\n\n [MAIN]: Color identification goes wrong! \n      Try
again or Modify boundaries in [ii]!"

        while (vector_field != []):
                n = n + 1
                print "\n\n [MAIN]: Starting Object Detection loop!!"
                ## Set the Moto G1 camera again
                image = i.set_the_camera_from_usb()
                # image = cv2.imread("img_for_test/CAMPO CERTO/teste (%d).jpg" %n)
#PORCA+PRISMA+U/

                ## Correct perspective from image
```

```
            image = iii.perspective_correction(image, vector_field)

            ## Find the object
            vector_object = iv.vector_object_template(image)
            if vector_object != []:
                print "\n\n [MAIN]:", vector_object, "          (X, Y in pixels)"
                converted_vector_object = iv.convert_vector_object(vector_object)
                print " [MAIN]:", converted_vector_object, "          (X, Y in mm)"

                ## Send vector_object to PDL2
                print "\n [MAIN]: Can I send", converted_vector_object,"to PDL2?
(yes=1/no=2)"
                s_n = input(' ')
                if s_n == 1: send_position = v.send_to_pdl2(converted_vector_object)
                else: print "\n [MAIN]: Nothing sent"

                # send_position = v.send_to_pdl2(converted_vector_object)

                print "\n [MAIN]: End loop!!"
                #cv2.putText(image,str(converted_vector_object),(x,y-
30),cv2.FONT_HERSHEY_SIMPLEX,0.50,(0,0,0),1,cv2.LINE_AA)

            else: print "\n\n\n [MAIN]: Object identification goes wrong! \n      Try again
or Modify efficiency in [iv]!"
```

APÊNDICE I - CÓDIGO EM PDL2 PARA SEPARAÇÃO DAS PEÇAS

```
PROGRAM graduation_v10 HOLD -- Programa com comandos de movimento;
 ------------------------------------------
----DECLARANDO CONSTANTES --
------------------------------------------
CONST
        MY_PORT = 1214
        VERBOSE = 0

        DV_TCP_GET = 27          -- documentazione comau
        DV_TCP_ACCEPT = 29       -- documentazione comau (open a server)
        DV_TCP_CONNECT = 30      -- documentazione comau (open a client)
        DV_TCP_DISCONNECT = 31   -- documentazione comau

------------------------------------------
----DECLARANDO VARIAVEIS ---
------------------------------------------
VAR
        pos_inicial, pos_final_1, pos_final_2, pos_final_3 : POSITION
        pos_final_1_1, pos_final_2_1, pos_final_3_1 : POSITION
        feedback, lun_tcp_rs, my_object : INTEGER

------------------------------------------
----DECLARANDO ROUTINES ------
------------------------------------------
ROUTINE move_to_object : INTEGER EXPORTED FROM graduation_v10 -- Rotina
principal do codigo

ROUTINE tcp_accept(port, vi_netlun, vi_scrlun, verbose : INTEGER) EXPORTED FROM
graduation_v10 -- Adicionando a "routine" tcp_accpet a "routine" status_decode2;

ROUTINE ru_get(vi_netlun, vi_scrlun, verbose : INTEGER) EXPORTED FROM
graduation_v10

ROUTINE ToolFrame(ai_tool, ai_frame, ai_arm : INTEGER()) EXPORTED FROM
TT_TOOL GLOBAL -- Definidor de Tool e Frame
```

```
ROUTINE RmtToolFrame(ai_tool, ai_frame, ai_arm : INTEGER()) EXPORTED FROM
TT_TOOL GLOBAL
-----------------------------------------------
ROUTINE ru_get(vi_netlun, vi_scrlun, verbose : INTEGER)
VAR ls_session, ls_accept, ls_connect : STRING[40]
 li_remote, li_local, li_options, li_linger : INTEGER
BEGIN
 -- DV_CNTRL Build in Routine fuer get information ----------------------
 WRITE LUN_CRT (NL,NL,'[ru_get()] Waiting data from PYTHON...')
 DV_CNTRL(DV_TCP_GET, (vi_netlun), ls_session, ls_accept, ls_connect, li_remote,
li_local, li_options, li_linger)
 IF $DV_STS = 0 THEN
  IF verbose = 1 THEN
   IF vi_scrlun = -1 THEN
       WRITE LUN_CRT ('    [ru_get()]: ')
     WRITE LUN_CRT ('    Session: ', ls_session, ' Accept: ', ls_accept)
     WRITE LUN_CRT ('    Connect: ', ls_connect, ' Remote Port: ', li_remote, ' Local: ',
li_local, NL )
   ELSE
       WRITE vi_scrlun ('    [ru_get()]: ')
     WRITE vi_scrlun ('    Connect: ', ls_connect, ' Port: ', li_remote, ' Local: ', li_local, NL )

   ENDIF
  ENDIF
 ENDIF
END ru_get

-----------------------------------------------

ROUTINE tcp_accept(port, vi_netlun, vi_scrlun, verbose : INTEGER)
BEGIN

 IF verbose = 1 THEN
  IF vi_scrlun = -1 THEN
   WRITE LUN_TP (NL,NL,'[tcp_accept()] Waiting for connections...')
```

```
  ELSE
    WRITE vi_scrlun (NL,NL,'[tcp_accept()] Waiting for connections...')
  ENDIF
 ENDIF

 DV_CNTRL(DV_TCP_ACCEPT, (vi_netlun), (port), '0.0.0.0')

 IF $DV_STS = 0 THEN
  ru_get((vi_netlun), -1, verbose)
  IF verbose = 1 THEN
   IF vi_scrlun = -1 THEN
    WRITE LUN_TP ('[tcp_accept()] Connected.')
   ELSE
    WRITE vi_scrlun ('[tcp_accept()] Connected.')
   ENDIF
  ENDIF
 ELSE
  IF vi_scrlun = -1 THEN
   WRITE LUN_TP ('[tcp_accept()] Error! $DV_STS=', $DV_STS , NL)
  ELSE
   WRITE vi_scrlun ('[tcp_accept()] Error! $DV_STS=', $DV_STS , NL)
  ENDIF
 ENDIF

END tcp_accept

-------------------------------------------------

ROUTINE move_to_object : INTEGER
CONST

        E = 90  -- VERTICAL 180
        R = 0   -- Angulo com a horizontal
VAR
        pos_object, pos_object_pluZ: POSITION      -- POS(X, Y, Z, ANG1, ANG2, ANG3)
```

```
    XYAT : ARRAY[4] OF INTEGER              -- [X, Y, ANG1, TYPE]
    i, Z : INTEGER

BEGIN
    Z := 200
  FOR i := 1 TO 4 DO
        READ lun_tcp_rs(XYAT[i]::4)         -- Valores recebidos do Python;
  ENDFOR

  WRITE LUN_CRT (NL,NL,'[move_to_object]')
    WRITE LUN_CRT ('    Valores recebidos:',XYAT[1],XYAT[2],XYAT[3],XYAT[4])

    pos_object := POS(XYAT[1], XYAT[2], Z, XYAT[3], E, R)
    my_object := XYAT[4]

    WRITE LUN_CRT ('    Posicao desejada:',XYAT[1],XYAT[2], Z, XYAT[3],E,R,
XYAT[4])
    -- OPEN HAND 1
    MOVE LINEAR TO pos_object

    Z := 55
    MOVE LINEAR TO POS(XYAT[1]-30, XYAT[2], Z, XYAT[3] ,E, R)
    -- CLOSE HAND 1
    Z := 300
    MOVE LINEAR TO POS(XYAT[1]-30, XYAT[2], Z, XYAT[3] ,E, R)

    feedback := 5
    WRITE lun_tcp_rs(feedback)-- Resposta a ser enviada pelo socket.

    RETURN (my_object)                  -- Retorno da funcao
END move_to_object

-----------------------------------------------
```

```
------------------
---- MAIN ----
------------------
BEGIN CYCLE
        WRITE LUN_CRT (NL,NL,'[BEGIN CYCLE] Starting robot ...')
        ToolFrame(2, 2)         -- Define Tool and Frame
        --pos_inicial := POS(490,470,190,-90,90,0)
        --pos_final_1 := POS(-320,220,140,-90,90,0)
        --pos_final_2 := POS(950,145,140,-90,90,0)
        --pos_final_3 := POS(115,530,140,-90,90,0)

        MOVE LINEAR TO pos_inicial       -- MOVE 1: Pointing to door

        -- WRITE LUN_CRT (NL, 'Starting robot ...', NL)
        -- WRITE LUN_CRT (NL, '[ROS_COMAU_robot_status] Connection starting...',
NL)

        OPEN FILE lun_tcp_rs ('NETT:', 'rw'),
        WITH $FL_BINARY = TRUE,
        ENDOPEN

        tcp_accept (MY_PORT, lun_tcp_rs, LUN_CRT, VERBOSE)
        -- WRITE LUN_CRT (NL,'Starting cycle...',NL)

        move_to_object

        SELECT my_object OF
                CASE (1):
                MOVE LINEAR TO pos_final_1_1
                MOVE LINEAR TO pos_final_1
                MOVE LINEAR TO pos_final_1_1
                CASE (2):
                MOVE LINEAR TO pos_final_2_1
                MOVE LINEAR TO pos_final_2
                MOVE LINEAR TO pos_final_2_1
```

```
            CASE (3):
            MOVE LINEAR TO pos_final_3_1
            MOVE LINEAR TO pos_final_3
            MOVE LINEAR TO pos_final_3_1
        ELSE:
            WRITE LUN_CRT (NL, ' my_error: something went wrong', NL)
        ENDSELECT
        -- OPEN HAND 1

        WRITE LUN_CRT('    Enviando resposta para o PYTHON...')
        DV_CNTRL(31, (lun_tcp_rs))          -- Envia para o Python que ciclo acabou
        DELAY 1000
        CLOSE FILE lun_tcp_rs
        WRITE LUN_CRT('[END] Connection closed.', NL)

END graduation_v10
```