INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA - CAMPUS ITABAIANA

CURSO TÉCNICO EM AUTOMAÇÃO INDUSTRIAL

Modalidade presencial

MODELO DE MOBILIDADE MECATRÔNICA INTELIGENTE

Joás de Brito Ferreira Filho

Orientador(a): Msc. Márcia Fernanda da Silva Santiago

Itabaiana – PB, Novembro de 2019

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA - CAMPUS ITABAIANA

CURSO TÉCNICO EM AUTOMAÇÃO INDUSTRIAL

Modelo de Mobilidade Mecatrônica Inteligente

Uma abordagem teórica utilizando Redes Neurais Artificiais e Mapeamento Cartesiano

Joás de Brito Ferreira Filho

Orientador(a): Msc. Márcia Fernanda da Silva Santiago

Trabalho apresentado ao Instituto Federal de Educação, Ciência e Tecnologia da Paraíba (IFPB) - *Campus* Itabaiana, como requisito para conclusão do Curso Técnico em Automação Industrial, modalidade integrado.

Itabaiana – PB, Novembro de 2019

RESUMO

Este trabalho apresenta o modelo teórico de um algoritmo de controle de direção para robôs industriais (Modelo de Mobilidade Mecatrônica Inteligente, ou 3MI), elemento crucial e indispensável na automação do processo de transporte de cargas. No panorama atual da indústria, algumas empresas pioneiras já vêem a logística de transporte como uma variável determinante para um fluxo constante e ininterrupto de artigos industriais, adotando, assim, robôs que eliminam problemas como congestionamento em meio à fábrica, acidentes de natureza ergonômica, entre outros. Entende-se que para se implementar um protótipo funcional é necessário que haja uma catalogação de especificações características de cada planta industrial, motivo que inviabiliza a construção de um único robô, para o qual seriam necessários parâmetros físicos reais, o que caracterizaria este trabalho como um estudo de caso, não sendo este o objetivo. Apresenta-se, portanto, um esquema abstrato de organização e processamento matemático que, como é mostrado, pode ser caracterizado como um modelo algorítmico passível de aplicação em sistemas reais. O algoritmo é dividido em três módulos: o módulo de mapeamento cartesiano; o módulo de tomada de decisão e o módulo principal, que coordena os dois primeiros. É apresentada uma visão geral dos requisitos teóricos (em otimização matemática e em geometria analítica) para o desenvolvimento do sistema e um modelo de implementação que busca máxima otimização em trabalhos futuros, cuja linha de pensamento é melhor aprofundada no Apêndice, utilizando como abordagem de avaliação de eficiência uma modelagem matemática simulada computacionalmente.

Palavras chave: sistemas autônomos; redes neurais artificiais; robôs industriais.

ABSTRACT

This paper presents a theoretical driving control model for industrial robots (Intelligent Mechatronic Mobility Model), a crucial and indispensable element in the automation of the loads transport process. In the current situation of the industry, some pioneers companies actually see the transport logistic as a determinant variable for a constant and ininterruptible flow of industrial items, adopting, therefore, robots that eliminates problems like congestion inside the factory, ergonomic accidents, and so on. It is a consense that to implements functional prototype it is necessary to catalogate the specifications of each industrial plant, being it what makes it unfeansible to constructs a single robot, for which would be necessary real physics parameters, what would characterize this paper as a case study, not being this the intent. It is shown an abstract scheme of organization and mathematical processment that, as it is shown, can be characterized as an algorithmic model that can be applied in real systems. The algorithm is divided in three modules: the cartesian mapping module; the decision making module and the main module, that controls the two others. It is presented a general view of the theoretical requirements (about mathematical optimization and analytics geometry) for the system's development and an implementation model that seeks the better optimization in future papers, which details are covered in the Appendix, using as evaluation approach a mathematical modeling computationally simulated.

Keywords: autonomous systems; artificial neural networks; industrial robots.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

1. INTRODUÇÃO

No contexto industrial é evidente a necessidade do transporte de cargas, seja no momento de aquisição de materiais, de escoamento da produção ou ainda na manutenção logística da planta industrial. A tarefa geralmente é desempenhada por operários, utilizando a própria força ou equipamentos de transporte, para cargas mais pesadas. Assim, o processo de transporte de cargas pode ser descrito como uma rota que vai de uma origem a um destino, descrevendo um percurso semelhante a cada ciclo de transporte.

O avanço das tecnologias da informação e a aplicação destas na emergente Indústria 4.0 trazem ao panorama laboral a automação de tarefas, *a priori* realizadas por seres humanos. O emprego dos Robôs Industriais em tais tarefas vem criando um ambiente onde o operário tem pouca ou nenhuma influência nos robôs dentro da indústria. No transporte de cargas, a abordagem automática mostra-se uma maneira eficiente de reduzir custos com mão de obra e evitar eventuais danos (FEB UNESP, 2016) à saúde dos operários.

Para que um robô transportador possa operar com segurança e eficiência razoável, independente da supervisão humana, este deve ser capaz de seguir uma rota conhecida, da origem ao destino, desviando de objetos, sejam eles móveis ou estáticos. A eficiência de um robô autoguiado é medida segundo o custo total do robô, o tempo gasto em uma rota e a capacidade de peso que pode ser transportada por ciclo origem-destino, sendo estas variáveis comparadas com o transporte puramente humano (quando o operário realiza o transporte utilizando a própria força) e com o transporte auxiliado por máquinas e carrinhos, o que nos concede uma visão geral dos pontos positivos e negativos deste tipo de sistema.

Este trabalho descreve o modelo teórico de um sistema de autonomia mecatrônica, isto é, uma estrutura lógica para robôs autônomos. O modelo é descrito por partes, de fato, os módulos que compõem o Modelo de Mobilidade Mecatrônica Inteligente (3MI). É dada uma base teórica para facilitar o entendimento das Redes Neurais Artificiais e porquê esta tecnologia compõe o núcleo do sistema aqui proposto. Ao longo de todo o trabalho é utilizada uma metodologia descritiva que traz os elementos do sistema à discussão científica, a partir da exposição matemática dos conceitos que relacionam estes elementos; uma seção (3.1.4.) é dedicada à exposição

dos parâmetros de compatibilidade com *hardware* e, por fim, é apresentada uma discussão sobre a eficácia de um sistema baseado neste modelo.

1.1. OBJETIVOS

1.1.1. Objetivo Geral

Apresentar a proposta de implementação de um sistema capaz de automatizar o processo de transporte de cargas no contexto industrial.

1.1.2. Objetivos Específicos

- Descrever os principais componentes do 3MI;
- Colaborar com o trabalho de futuros pesquisadores engajados na automação de robôs industriais;
- Aplicar tecnologias de diversas áreas do conhecimento na automação de um processo com riscos inerentes, diminuindo assim injúrias no meio industrial;
- Automatizar robôs *a priori* controlados por humanos, diminuindo, assim, os erros;

2. REFERENCIAL TEÓRICO

Diversas abordagens para o problema de orientação espacial automática giram em torno de uma programação linear, com casos específicos estritamente codificados pelo desenvolvedor. Este tipo de sistema tem suas limitações, no sentido de que torna inviável a aplicação em ambientes demasiadamente distintos, apresenta muitos detalhes que devem ser levados em conta na implementação do algoritmo, o que maximiza a presença de erros e dificulta a conclusão de um sistema totalmente utilizável. A abordagem utilizada neste projeto tem como diferencial uma tecnologia emergente, que apresenta versatilidade e porcentagem de erro desprezível; redes neurais artificiais permitem ao desenvolvedor reduzir passos necessários simplificando o algoritmo a uma única função, que requer apenas um número suficiente de dados amostrais; este caminho leva a uma implementação simples e eficiente do algoritmo de desvio de objetos.

2.1. REDES NEURAIS ARTIFICIAIS

As *Artificial Neural Networks* (ANN), ou Redes Neurais Artificiais, são um modelo matemático baseado na estrutura do cérebro (HAYKIN, 1994). Redes Neurais Artificiais são consideradas aproximadores universais de funções, ou seja, dados dois conjuntos D e I (domínio e imagem), uma rede neural pode descrever uma relação R entre estes dois conjuntos, com o treinamento adequado e um *dataset* (conjunto de dados) com informações suficientemente numerosas para explicitar o problema à ANN. Assim, formalmente definindo as redes neurais, estas são uma estrutura matricial em camadas formada por pesos; recebe entradas e as transmite ao longo das próximas camadas, repetindo o processo até a camada de saída.

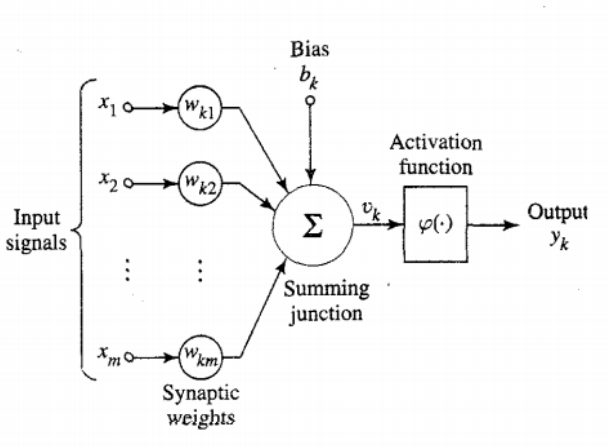
2.1.1. ESTRUTURA DO NEURÔNIO ARTIFICIAL

Um neurônio pode ser caracterizado como uma função, no sentido em que este é constituído de um número fixo de entradas. Cada entrada possui um peso w, que define a importância daquela entrada na próxima camada de neurônios. Assim, sendo as entradas do neurônio representadas pela matriz $X_{n,1}$, os pesos pela matriz $W_{1,n}$ e a saída

do neurônio pelo número real z, temos que o neurônio (representado na Figura 1) é matematicamente expresso pela Equaçaõ 1:

$$z = X \cdot W + b \quad . \tag{1}$$

Figura 1 – Representação do Neurônio Artificial



Fonte: (HAYKIN, 1994)

Onde o elemento b, denominado bias, trata-se de uma variável que influencia a rede neural em duas situações: *underfitting* e *overfitting*, atuando, desta forma, como um neurônio auxiliar, que pode indicar a acurácia do treinamento da rede; esse pode ser incluído como o elemento W[0], tornando necessária a atribuição do elemento X[0] como sendo igual a 1, o que significa que a entrada do neurônio bias será constante em 1. A equação expandida toma a forma

$$z = (1 \cdot b) + (X[1] \cdot W[1]) + (X[2] \cdot W[2]) + \ldots + (X[n] \cdot W[n]) = X \cdot W \tag{2}$$

A saída gerada por um dado neurônio pode assumir valores de grandezas demasiado discrepantes, dificultando as computações subsequentes pelas outras camadas da rede neural. Assim, um último elemento passa a constituir o neurônio artificial, a chamada função de ativação. Existem diversas funções que podem desempenhar este papel, sendo a mais popular delas a função sigmóide (*Sigmoid Function)*. Esta função, cuja projeção cartesiana lembra a forma do caractere S, tem duas propriedades que tornam-na especialmente adequada para a aplicação em redes neurais artificiais. A primeira, o enquadramento de qualquer valor gerado como saída

em um intervalo de 0 a 1, o que facilita o manuseio destes valores; a segunda é a aplicabilidade desta função no algoritmo de aprendizagem automática, que será explorado *a posteriori*, na Seção 2.1.2. A Equação 3 descreve a função sigmóide e tem a forma:

$$f(x)=\frac{1}{1+e^{-x}} \quad . \tag{3}$$

A saída do neurônio, agora com a função de ativação (generalizada por f) é dada pela Equação 4:

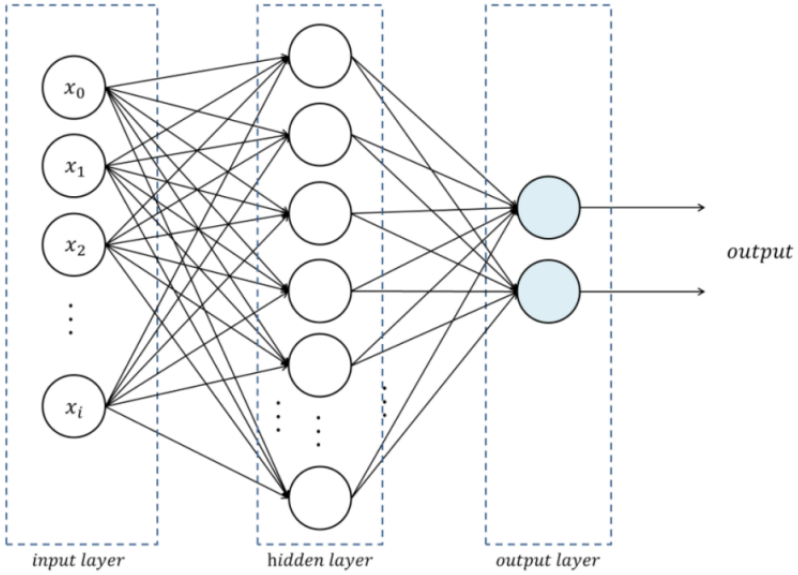
$$\tilde{Y}=f(\sum_{i=0}^{n} x_i w_i) \tag{4}.$$

A função acima (Equação 4) descreve o elemento neural mais simples, com apenas um neurônio: O Perceptron. Esta estrutura cognitiva é limitada a problemas simples, chamados lineares. Isso significa que um problema um pouco mais complexo, como a simulação da porta lógica XOR, não pode ser solucionada por este tipo de estrutura. As redes neurais podem variar segundo sua estrutura, número de neurônios, forma de conexão, funções de ativação e algoritmo de treinamento, o que torna denso o estudo de todas as combinações de neurônios. Para fins de análise matemática, seja dada uma rede neural com as seguintes características:

- Uma camada oculta (*hidden layer*)
- Neurônios totalmente conectados
- Função sigmóide em todas as camadas

Esta é uma *feed-forward neural network* (rede neural multicamada), também conhecida como MLP (Multilayer Perceptron), e pode ser representada graficamente como segue, na Figura 2.

Figura 2 – Exemplo de Rede Neural do tipo MLP



Fonte: https://www.cc.gatech.edu/~san37/post/dlhc-fnn/

2.1.2. TREINAMENTO POR MEIO DE OTIMIZAÇÃO MATEMÁTICA (Backpropagation / Gradient Descent / Chain Rule)

Para que uma rede neural funcione corretamente, *i.e.*, gere saídas que determinem a aproximação de uma dada função "ideal", é necessário que haja uma configuração particular de pesos. As redes neurais têm uma intrínseca relação com os processos de Otimização Matemática, conjunto de conhecimentos que visa alcançar valores denominados ótimos; e no caso das redes neurais, o algoritmo de otimização mais utilizado é conhecido como *gradient descent* (ou descida de gradiente), e funciona segundo princípios (STRANG) do cálculo diferencial (Regra da Cadeia) para encontrar o mínimo local de uma função (ROBBINS; MONRO, 1951) de erro. Este processo é efetuado em todas as camadas, no inverso à propagação entrada-saída, o que dá ao

método o nome *Backpropagation* (*backward propagation of errors*). O algoritmo surgiu com a necessidade vista pelos pesquisadores de que Perceptrons Multicamadas encontrassem sozinhos a melhor estrutura interna (Brilliant.org, 2019). O algoritmo de *backpropagation* requer três elementos para funcionar:

- Um conjunto de dados no formato $X = \{(x_1,y_1), (x_2,y_2), ... (x_n,y_n)\}$, onde x_i é uma entrada e y_i é a saída esperada para aquela entrada.
- Uma *feedforward neural network*, cujos parâmetros, denominados pelo conjunto Θ, são: w_{ij}^k , o peso entre o neurônio (ou nó) j na camada l_k e o nó i na camada l_{k-1}; b_i^k , o elemento bias do nó i na camada l_k. Os nós em uma mesma camada não são conectados entre si, e as camadas são totalmente conectadas, ou seja, todos os neurônios de uma camada estão conectados com todos os neurônios das camadas vizinhas.
- Uma *error function* $E(X, \Theta)$, que expressa o erro (discrepância matemática) entre a saída desejada y_i e a saída calculada $\hat{y}_i$ para uma dada combinação dos parâmetros Θ sobre uma entrada x_i, utilizando-se um dado par (x_i, y_i).

A função de erro (*cost function, error function*) calcula o quão próxima a saída de uma rede está da saída que o treinador espera, servindo como um feedback do processo de treinamento da rede neural (NIELSEN, 2015); há diversas funções que podem servir para este fim, sendo a função *mean squared error* a mais utilizada para o treinamento de redes neurais artificiais. Esta função é necessária para que se obtenha o valor E_{total} (erro total), que é obtido a partir da soma dos erros de cada unidade, como pode ser visto na Equação V

$$E_{total}(target, output) = \sum \frac{1}{2}(target_i - output_i)^2, \quad (5)$$

sendo *target* a saída esperada e *output* a saída propriamente dita, calculada pela rede neural; $target_i$ e $output_i$ indicam os pares correspondentes a cada unidade *i*.

Todos os hiperparâmetros da rede (pesos, biases) devem ser alterados de acordo com o E_{total}, utilizando uma técnica de derivação conhecida como Regra da Cadeia, que nos permite calcular a derivada de uma função composta do tipo f(g(x)), como é mostrado na Equação VI:

$$\frac{dy}{dx}[f(g(x))] = f'(g(x))\,g'(x) \quad . \quad (6)$$

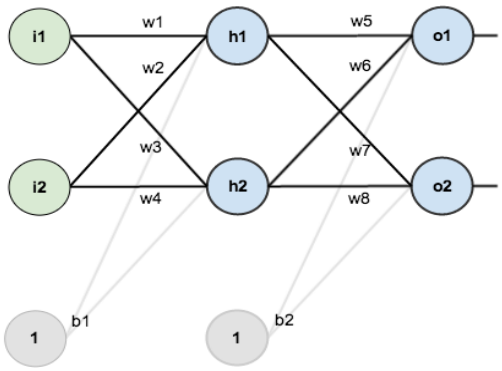
A Regra da Cadeia enuncia, portanto, que a derivada de uma função composta f(g(x)) com relação a x é igual ao produto das derivadas de f(x) com relação a g(x) e de g(x) com relação a x, isto é,

$$\frac{df(g(x))}{dx} = \frac{df(x)}{dg(x)} \cdot \frac{dg(x)}{dx} \quad . \tag{7}$$

Para alterar um peso, por exemplo, na última camada de uma rede neural 2x2x2, como a representada na Figura 3 (MAZUR, 2015), precisamos saber o quanto este peso afeta E_{total}, *i.e.*, procuramos o valor da derivada parcial de E_{total} com relação a w_i. Aplicando a Regra da Cadeia para o peso w_5, temos que

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} \cdot \frac{\partial out_{o_1}}{\partial \imath_{o_1}} \cdot \frac{\partial \imath_{o_1}}{\partial w_5} \quad , \tag{8}$$

Figura 3 – Exemplo de Rede Neural



Fonte: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

no que segue, pelas regras da diferenciação parcial (STRANG), que

$$\frac{\partial E_{total}}{\partial out_{o_1}} = -1\left(2\frac{1}{2}\left(target_{o1} - out_{o1}\right)^{2-1}\right) + 0 = out_{o1} - target_{o1} \quad . \tag{9}$$

Nos resta calcular o quanto a saída de o_1 varia com relação à entrada de o_1, ou seja, procura-se a derivada parcial de out_{o1} com respeito a in_{o1}, dada por

$$\frac{\partial out_{o_1}}{\partial \imath_{o_1}} = out_{o1}(1 - out_{o1}) \quad , \tag{10}$$

nos restando um último elemento, a derivada parcial de in_{o1} com respeito a w_5, que é dada por

$$\frac{\partial \imath_{o_1}}{\partial w_5} = out_{h1} \quad ; \tag{11}$$

Assim, substituindo em VII as equações X, IX e VIII, temos, na forma da Equação 12, que

$$\frac{\partial E_{total}}{\partial w_5} = \left(out_{o1} - target_{o1}\right) \cdot \left(out_{o1}\left(1 - out_{o1}\right)\right) \cdot \left(out_{h1}\right) \quad . \qquad (12)$$

Utiliza-se este valor para diminuir o erro, isto é, alterando w_5. O procedimento é feito subtraindo-o de w_5, opcionalmente multiplicando $\frac{\partial E_{total}}{\partial w_5}$ por uma taxa de aprendizado, neste caso, $\propto$. Finalmente, temos que o novo peso w_5 assumirá o valor dado por

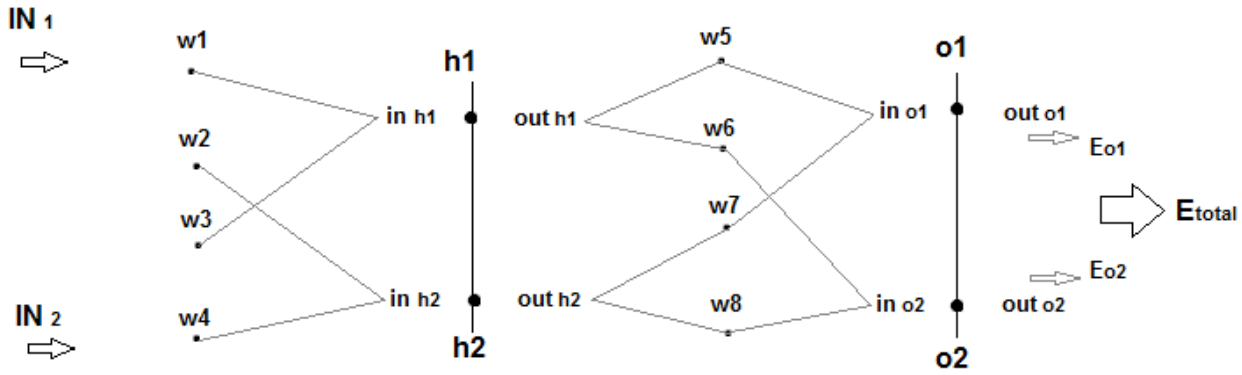
$$5 + ¿ = w_5 - \propto \frac{\partial E_{total}}{\partial w_5} \quad , \qquad (13)$$

$$w_{¿}$$

sendo necessário repetir este processo para todos os pesos da rede, camada por camada. É importante ressaltar que os valores utilizados no *backpropagation* são pertencentes à primeira configuração das camadas da rede, devendo-se aplicar as correções nos pesos apenas dispondo do valor de todos os pesos; isso quer dizer que os valores utilizados em um dado ciclo de *backpropagation* correspondem aos valores que geraram a saída do ciclo *feedforward* que imediatamente o precede.

Se observarmos com atenção, os cálculos do algoritmo de treinamento funcionam seguindo uma trilha, como no último exemplo, indo de E_{total} até w_5; as operações foram baseadas em derivar parcialmente o par [*elemento da direita, elemento da esquerda*], como se pode observar no grafo presente na Figura 4. Este diagrama nos dá base suficiente para elaborar o algoritmo de backpropagation para treinar automaticamente uma rede neural.

Figura 4 – Diagrama de Trilha (*backpropagation*)



Fonte: O autor

Apesar de não estarem representados, os *biases* são alterados seguindo o mesmo processo dos pesos convencionais. Para concluir, considere-se o seguinte exemplo: deseja-se modificar o valor do peso w_1; seguindo a trilha representada na figura 7, temos de ir de E_{total} até w_1. O algoritmo capaz de obter, automaticamente, todos os erros dos nós da camada oculta (para uma rede de três camadas) é obtido através da generalização (Equação 15) dos cálculos necessários ao desenvolvimento da Equação 14, e fica como contribuição do autor, no módulo neural_network.py, disponível no Anexo III.

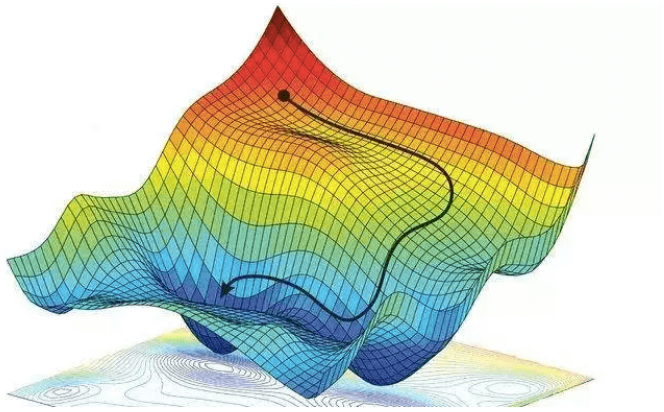
$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \cdot \frac{\partial out_{h1}}{\partial ¿_{h1}} \cdot \frac{\partial ¿_{h1}}{\partial w_1} \tag{14}$$

$$\frac{\partial E_{total}}{\partial w_h} = \sum \left[\left(\left(out_o - target_o \right) \cdot \left(out_o \left(1 - out_o \right) \right) \cdot w_{ho} \right) \right] \cdot \left[out_h \left(1 - out_h \right) \right] \cdot \left[¿_h \right] \tag{15}$$

A Equação 14 descreve, de maneira generalizada, o cálculo da derivada parcial do erro total com respeito a um dado peso da camada oculta. Estabelece, portanto, que este valor está relacionado com todos os neurônios da camada de saída, valor representado pelo somatório dos produtos na forma $\left[\left(\left(out_o - target_o \right) \cdot \left(out_o \left(1 - out_o \right) \right) \cdot w_{ho} \right) \right]$, onde o representa um neurônio da camada oculta e w_{ho} representa o peso que liga este neurônio ao neurônio que utiliza o peso w_h em questão.

Em suma, acompanhar o algoritmo de backpropagation em funcionamento nos deixa cientes de como o ato de mudar os pesos em uma rede muda a saída gerada por ela, e esse é o núcleo do processo de aprendizagem automática: ajustar os parâmetros internos da rede neural de maneira a obter o menor erro, ou seja, a função que melhor aproxima a rede neural da *função ideal*.

Figura 5 – Ilustração do Processo de Otimização *Gradient Descent*



Fonte: https://easyai.tech/en/ai-definition/gradient-descent/

3. METODOLOGIA

A metodologia deste trabalho deu-se com a implementação dos módulos constituintes do 3MI, seguida de uma análise sobre a relação entre a complexidade do espaço representado por uma rede neural e o número de entradas desta rede. Os códigos foram escritos na linguagem de programação Python (python.org), por motivos expostos na Seção 3.1.4.

Para que um sistema de controle de locomoção funcione adequadamente não basta que este permita ao robô identificar obstáculos e desviar deles, é necessário que haja um controle de percurso eficiente, que não permita ao robô desviar-se de sua rota original. Integrar, em um único sistema, um controle de rota e a capacidade de evitar colisões com objetos define um robô que faz uso deste sistema como autonomamente guiado.

Para implementar um sistema de controle de percurso é necessário, antes de tudo, implementar uma representação adequada do que seria um percurso. Uma maneira de alcançar este fim seria representar um percurso linear como uma equação, e um percurso misto como uma série de equações. Outra alternativa, igualmente válida, seria

implementar um sistema cartesiano e representar as rotas como pares de pontos neste plano. De toda forma, o sistema necessitaria de uma memória, à qual faria consultas em tempo real.

A comunicação entre a rede neural e o sistema de controle de rota se daria através de interrupções de processamento. Isso significa, em poucas palavras, que a rota pré-programada seria seguida, de maneira que o sistema motor do robô estaria sendo controlado pelos algoritmos de controle de percurso, ao passo que a rede neural estaria ciente de todas as variações no ambiente possíveis de se captar por seus sensores, e tomando o controle do sistema motor assim que detectasse um objeto, realizando a tomada de decisão e desviando a rota. Assim que o desvio fosse concluído, o percurso feito pelo robô neste processo estaria armazenado e seria utilizado como constante para calcular os passos necessários ao retorno do robô à rota principal. Isto descreve um sistema que alterna o núcleo de processamento entre evitar colisões e corrigir uma rota.

Em suma, o percurso de um dado robô seria representado, no sistema, como uma série de passos, que descreveriam segmentos de retas por pontos, os diversos vértices onde a direção do movimento muda; assim, toda a rota seria composta por instruções do tipo “mova-se para o ponto (x,y)”.

3.1. FUNCIONAMENTO GERAL DO SISTEMA

Esta seção tem como objetivo descrever estrutural e funcionalmente um sistema teórico de controle de locomoção voltado a robôs industriais. Primeiramente, é preciso responder a certas questões que dizem respeito às características gerais do objeto de pesquisa (robôs autônomos) e às particularidades do modelo apresentado neste artigo.

Robôs industriais autônomos seguem uma linha de funcionamento muito semelhante. Há um sensoriamento local, um processamento, que pode ser local ou remoto e uma reação em tempo real. Recentemente a empresa BMW passou a utilizar robôs autônomos dentro de suas fábricas, no transporte de peças automotivas. Em um artigo publicado pela empresa BMW (BMW Group, 2016), foi descrito o funcionamento geral do projeto: “Rodeado de radiotransmissores e equipado com um mapa digital, o robô dirige independentemente até o seu destino. Quando trens rebocadores cruzam o seu caminho, um sensor identifica o obstáculo e para o robô autoguiado [...]” (tradução do autor).

Neste trabalho, é apresentado, como sugestão de implementação, um sistema baseado no 3MI composto, fisicamente (ver seção 3.1.4), por doze sensores ultrassônicos HC-SR04 e uma placa Raspberry PI. Estes seriam os elementos necessários para suportar as especificações de processamento de informações e resposta em tempo real. Torna-se necessário, no desenvolvimento de um robô autônomo, definir a lógica por trás do comportamento deste robô. Assim, este trabalho procura desenvolver um protótipo em ambiente simulado (ver seção 3.3) utilizando o módulo *graphics.py*, compatível com a linguagem Python e necessário para visualização do funcionamento do 3MI para assim comprovar, metodicamente, a eficiência dos mecanismos empregados na implementação do 3MI.

O modelo teórico aqui descrito pode ser visto como um passo crucial e requisito indispensável para a construção de um robô autoguiado, formando, em uma única rotina, todo o algoritmo de decisão necessário para a implementação de um robô autônomo. Dessa forma, o sistema será composto de dois núcleos de processamento: o algoritmo de desvio de objetos, neste caso uma rede neural artificial que detectará anomalias no percurso (como objetos transitando ou postos no meio do caminho) e o algoritmo de rotas, que ficará encarregado de "dirigir" o robô utilizando o mapa em sua memória e sendo auxiliado pelo algoritmo de desvio.

3.1.1. ABSTRAÇÃO GEOMÉTRICA

Para que seja capaz de seguir uma rota, um sistema necessita apenas de poucas informações, armazenadas no formato de segmentos de retas. A equação geral da reta ($Ax + By + C = 0$) é capaz de descrever qualquer reta no plano cartesiano. Assim, para cada segmento de linha reta presente no percurso de um dado robô, seu sistema de controle só precisa ter acesso a uma série de constantes `way[n] = {line, origin, destiny}`, onde `line` é a equação que perpassa aquela reta e os elementos `origin` e `destiny` vêm a ser os pontos que delimitam o segmento de reta; o *array* de segmentos `way` descreve o percurso completo do robô. Uma maneira de obter esta representação é implementar uma *class* que defina um padrão para as linhas. Esta classe deteria métodos para, por exemplo, calcular a distância até um ponto; além disso, um método que permitiria calcular retas que corrigissem um desvio da reta principal. O código que alcança este objetivo, escrito na linguagem Python 3, pode ser encontrado no Anexo I,

assim como todas as classes envolvidas na análise geométrica das rotas. Para explicar, de maneira geral, o papel de cada classe de objetos geométricos, foi organizada uma tabela (Tabela 1) contendo os métodos, atributos e objetivos de cada uma das classes implementadas.

TABELA 1 – Classes do módulo de mapeamento geométrico

Point(self, X, Y)	**Line**(self, A, B, C)	**Segment**(self, line, origin, destiny)	**Route**(self)
Define a entidade geométrica *ponto*, sendo descrito por coordenadas cartesianas.	Define a entidade *reta*, descrita pela função geral da reta, formada pelas constantes A, B e C.	Define a entidade *segmento de reta*, uma linha delimitada por dois pontos.	Define a entidade rota, uma lista de segmentos de reta que visam conectar dois pontos no plano.
self.X, self.Y (coordenadas do ponto no plano cartesiano)	self.A, self.B, self.C (constantes da equação $Ax + By + C = 0$)	self.line (equação que suporta o segmento), self.origin, self.destiny (pontos que delimitam o segmento)	self.way[] (lista de segmentos que descrevem o percurso completo do robô)
through(self, point): gera uma linha reta que passa por dois pontos	**show**(self): exibe a equação no monitor do dispositivo		**appendSegment**(self, segment): adiciona um segmento ao percurso; **nextStep**(self, matched_points): solicita o próximo

			segmento do percurso; **newRoute**(self, matched_points, end): indica desvio de rota; **setRight**(self, matched_points, wrng_segment): corrige o desvio de rota.

3.1.2. REDE NEURAL

O diferencial deste modelo de implementação é o módulo de tomada de decisão, que é composto por funções que criam, treinam e executam uma rede neural de três camadas (uma única *hidden layer*). A arquitetura recomendada para um sistema a ser construído fisicamente é baseada em uma *feedforward backpropagation neural network* com esquema de 12 nós na camada de entrada, 10 nós na camada oculta e 6 nós na camada de saída. Esta arquitetura (FFNN) é a mais recomendada por

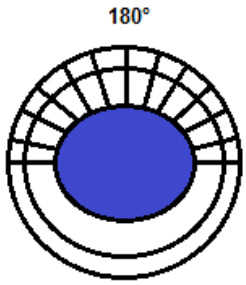
1. Mostrar-se eficiente no reconhecimento de padrões;
2. Ser de fácil implementação;
3. Doze sensores para as entradas, seis direções para as saídas (ver seção 3.1.4).

A rede neural deve ser capaz de receber as distâncias medidas pelos sensores e com base nesta entrada decidir que direção tomar. A nova rota será determinada quando, seguindo nesta direção, o robô encontrar-se em uma posição que torne viável a correção de rota por meio do módulo geométrico.

Um esquema de funcionamento pode ser observado no Anexo II. Seguindo a ordem das figuras, consideramos o ponto vermelho como o robô autônomo e o ponto

verde como o destino deste. Pode-se observar que o algoritmo em si corrige as rotas criando novas linhas, que vão de encontro com os pontos finais das linhas das quais o robô primeiramente desviou-se. Assim, a cada chamada ao módulo de tomada de decisão, a rede neural, previamente treinada naquele ambiente, dirá qual a melhor direção a se tomar, enquanto o módulo geométrico corrigirá o desvio efetuado. Abaixo, na Figura 4, pode-se observar o esquema físico, ou o arranjo dos sensores em um robô teórico.

Figura 6 – Esquema de disposição dos sensores ultrassônicos



Fonte: O autor

É de extrema importância ressaltar o nível de especificidade deste módulo, sabendo que a rede neural pode ter diversas arquiteturas e ainda assim resolver o problema com eficiência razoável. Agora apresentado o mecanismo que proporciona ao robô o desvio, é necessário que haja um módulo principal, que gerencie o comportamento do robô revezando o controle de ação entre o módulo de tomada de decisão e o módulo de mapeamento cartesiano.

Com um número suficiente de exemplos (*samples*) no formato [[s1, s2, ..., s12], [d1, d2, ..., d6]] (referindo-se, respectivamente, às saídas obridas e às saídas desejadas), é possível treinar uma rede neural, avaliá-la e, assim, qualificar a eficiência deste tipo de sistema. Dessa forma, é preciso implementar, além de uma arquitetura de rede neural, um algoritmo de treinamento (ver Seção 2.1.2). O código do MTD está disponível no Anexo III.

3.1.3. MÓDULO DE CONTROLE

O módulo de controle será responsável por coordenar os dois outros módulos, gerenciando os recursos e funções destes; assim, a criação de um único módulo de controle, que estabeleça uma interface entre sistemas distintos, deve levar em conta as particularidades dos elementos que serão controlados. Dessa forma, o módulo de controle está disposto hierarquicamente como a função *main*, *i.e.*, todas as outras funções utilizadas no 3MI serão chamadas pelo módulo de controle.

A importância do módulo principal é tanta que o algoritmo por trás de seu código descreve justamente o comportamento do robô, no mais alto nível de abstração. Para facilitar o entendimento do código, escrito em Python 3, o pseudocódigo do módulo principal está escrito logo abaixo

```
SE (MODO DE OPERAÇÃO == APRENDIZADO)
{
    DEFINIR SEGMENTO
    SEGMENTO = ROTA_MANUAL( )
}

SE (MODO DE OPERAÇÃO == PADRÃO)
DEFINIR SEGMENTO
REPETIR
{
    SE (NeuralNetwork(AMBIENTE) != FRENTE)
    {
            SEGMENTO = NOVA_ROTA( )
            CORRIGIR_ROTA (SEGMENTO)
    }
}
```

Como visto no algoritmo acima, a condição para invocar o método de correção de desvio, presente no módulo de mapeamento cartesiano, é que a rede neural gere uma saída diferente da correspondente ao comando “siga em frente” (assumida como saída padrão), isso quer dizer que enquanto o robô não se deparar com uma situação que possa ocasionar em colisão, a rota do robô permanecerá inalterada. O código deste módulo pode ser encontrado no Anexo IV.

3.1.4. ESQUEMA DE HARDWARE PROPOSTO

Para determinar o esquema de hardware de um protótipo sensitivo baseado no 3MI é necessário considerar uma plataforma que suporte a linguagem Python, linguagem em que foi desenvolvido o 3MI, além disso, é preciso levar em conta que este protótipo seria responsável por traçar rotas válidas, desviando de obstáculos e levando cargas até os pontos determinados; dessa forma, o sistema deve estar rodando

em um processador que, além de suportar o funcionamento de uma rede neural (especificamente uma do tipo *feedforward backpropagation multilayer perceptron*), deve ser capaz de gerar respostas em tempo real; estudos no campo dos veículos autônomos demonstraram sucesso com a plataforma Raspberry PI (WHITE, 2018).

Juntamente com o Raspberry, seriam empregados 12 sensores ultrassônicos HC-SR04 tendo em vista que, de acordo com as especificações presentes no datasheet do sensor (ELEC FREAKS, 2013) o sensor mede distâncias em um intervalo de 2cm a 400cm, em um ângulo de 15°. Dessa forma, serão dispostos 180/15 = 12 sensores dispostos como na Figura 4.

Um elemento lógico, presente no módulo principal, que até então não foi descrito, é o módulo *machine.py*. Este seria o módulo responsável pela comunicação das entradas e saídas físicas do robô com o sistema de controle. Idealmente, este módulo implementa uma classe Machine, com atributos de entrada e saída, além de parâmetros que possam ser necessários de acordo com a particularidade de cada sistema de hardware. É importante ressaltar que este é o único módulo específico, visto que o sistema pode ser aplicado a qualquer robô capaz de se orientar nas oito direções, o que torna necessário desenvolver um módulo machine.py para cada hardware, individualmente e segundo as suas especificações.

O módulo machine.py fica encarregado de suportar variáveis associadas a botões, métodos de varredura, métodos de movimentação e todos os elementos envolvidos na comunicação hardware-software. Sem este conjunto de códigos seria inviável desenvolver o sistema proposto neste trabalho.

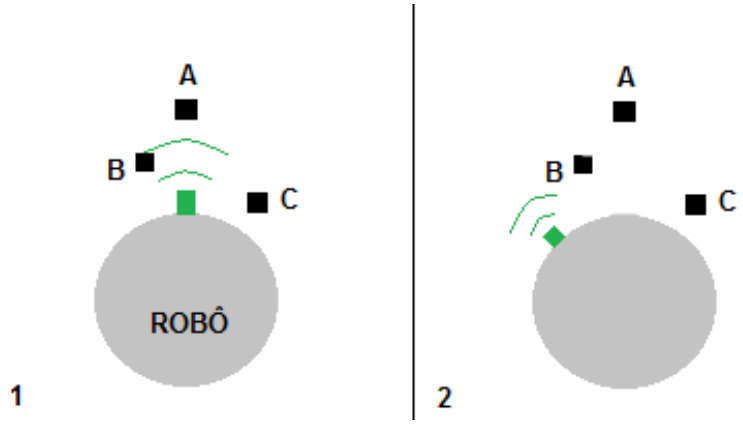
3.2. RELAÇÃO ENTRE A COMPLEXIDADE DO ESPAÇO REPRESENTADO E A EFICÁCIA DO SISTEMA AUTÔNOMO

Levando em conta todos os elementos do Modelo de Mobilidade Mecatrônica Inteligente até então apresentados, a saber, os módulos de Controle, Tomada de Decisão e Principal, além dos requisitos técnicos para implementação do módulo "*machine.h*", necessário como drive de comunicação com o hardware de um dado protótipo, podemos nos questionar sobre as limitações e perspectivas de aperfeiçoamento do 3MI. Em se tratando de qualquer modelo teórico é possível variar parâmetros construtivos de determinado sistema e, talvez assim, obter um funcionamento mais sofisticado e

eficiente (abordagem evidenciada no uso de redes neurais como módulo de tomada de decisão). Neste caso, é cabível discutir dois parâmetros deste modelo; um físico (o número de sensores dispostos no perímetro de um dado robô) e um lógico (a precisão do sistema de direcionamento, ocasionando em um melhor desempenho do sistema). É possível, segundo um raciocínio muito simples, chegar a uma relação válida de correspondência entre estes dois parâmetros.

Supondo um modelo mais simples, com menos sensores. É necessário, para entender as implicações desta simplificação, recapitular o fato de que, como o número de unidades na camada de entrada da nossa rede neural é correspondente com o número de sensores alojados no circuito do robô, um modelo mais simples tem como fim facilitar o estudo do desempenho do sistema sob diferentes arquiteturas. Desta forma, um robô com apenas um sensor é representado na Figura 7.

Figura 7 – Comportamento do sistema sob um único sensor



Fonte: O autor

Seguindo o esquema acima, em 1 temos a representação de um robô com seu único sensor ultrassônico conectado ao 3MI e mais três objetos, A, B e C, logo à frente. Sendo detectado o objeto A, o sistema indicará o desvio a ser feito para evitar a colisão com A. Estando o sensor livre de objetos em seu campo de sensibilidade, o sistema não

indicará a colisão com o objeto B, aparente em 2. A partir deste exemplo é possível afirmar que um sistema contando com apenas um sensor vem a ser ineficiente, inviabilizando seu emprego na indústria.

Levando em consideração a situação hipotética discutida no último parágrafo, podemos relacionar a ineficiência do sistema (por não ter sido capaz de indicar a colisão com o objeto B) com o número pequeno de sensores; de fato, um único sensor não é capaz de transmitir representações espaciais demasiado complexas. Assim, temos que o número de comportamentos para os quais um sistema baseado no 3MI está apto a apresentar é diretamente proporcional ao número de padrões que podem ser representados na camada de entrada da rede neural deste sistema. A isto equivale dizer, de forma abstrata: quanto mais pontos na figura representativa do perímetro circular, mais confiável é o comportamento do robô, visto que a resolução do sistema aumenta.

4. CONSIDERAÇÕES FINAIS

A automação industrial vem, aos poucos, sendo empregada em diversos problemas enfrentados nas fábricas. Um destes problemas, como foi apresentado, é o de transporte logístico, setor que apresenta riscos tanto ergonômicos, quando se emprega trabalhadores humanos nesta tarefa, quanto econômicos, quando se pondera a respeito da importância de uma planta industrial com uma logística fluida, ágil e em constante otimização; todos estes fatores motivaram empresas como a BMW a adotar robôs industriais de transporte, comprovando o potencial desta tecnologia.

Com o que foi apresentado até aqui, espera-se que, em trabalhos futuros, o sistema possa ser implementado com êxito e atingindo os resultados esperados, tendo sido seguidos os parâmetros descritos neste trabalho. O modelo integra conhecimentos das áreas de otimização matemática, inteligência artificial, robótica, ciência da computação e geometria analítica, o que garante a solidez da base teórica desta tecnologia que tem tudo para se tornar indispensável em todos os setores industriais nos

próximos anos. Esta solidez mostra-se compatível com os requisitos de operação dentro do ambiente industrial e vê-se alcançada pelo rigor matemático empregado na descrição deste modelo, refletido na estrutura dos módulos enfim apresentados.

5. REFERÊNCIAS BIBLIOGRÁFICAS

BMW GROUP PRESSCLUB. **BMW Group introduces self-driving robots in Supply Logistics**. Disponível em:<https://www.press.bmwgroup.com/global/article/detail/T0257786EN/bmw-group-introduces-self-driving-robots-in-supply-logistics?language=en> Acesso em: 17, Agosto de 19.

BRILLIANT. **Backpropagation**. Disponível em: <https://brilliant.org/wiki/backpropagation/> Acesso em: 17, Agosto de 19.

ELEC FREAKS. **Ultrasonic Ranging Module HC - SR04**. Disponível em: <https://www.electroschematics.com/wp-content/uploads/2013/07/HCSR04-datasheet-version-1.pdf> Acesso em: 20, Outubro de 2019.

FEB UNESP. **Análise ergonômica no transporte manual de cargas: um estudo de caso em uma empresa de produção de cimento** (2016). Disponível em: <https://revista.feb.unesp.br/index.php/gepros/article/viewFile/1627/766> Acesso em: 20, Outubro de 2019.

HAYKIN, Simon. **Neural Networks - A Comprehensive Foundation** (1994). Disponível em: <https://cdn.preterhuman.net/texts/science_and_technology/artificial_intelligence/Neural%20Networks%20-%20A%20Comprehensive%20Foundation%20-%20Simon%20Haykin.pdf> Acesso em: 20, Outubro de 2019.

NIELSEN, Michael. **How the backpropagation algorithm works**. Disponível em: <http://neuralnetworksanddeeplearning.com/chap2.html> Acesso em: 17, Agosto de 19.

ROBBINS, Herbert; MONRO, Sutton. **A Stochastic Aproximation Method.** Publicado por: The Annals of Mathematical Statistics (1951). Disponível em: <https://pdfs.semanticscholar.org/34dd/d8865569c2c32dec9bf7ffc817ff42faaa01.pdf> Acesso em: 20, Outubro de 2019.

SANJEEVI, Madhu. **Chapter 7: Artificial Neural Networks With Math**. Disponível em: <https://medium.com/deep-math-machine-learning-ai/chapter-7-artificial-neural-networks-with-math-bb711169481b> Acesso em: 17, Agosto de 2019.

WHITE, Toby. **Computational Efficiency: Can Something as Small as a Raspberry Pi Complete the Computations Required to Follow the Path?**. Disponível em: <https://www.intechopen.com/online-first/computational-efficiency-can-something-as-small-as-a-raspberry-pi-complete-the-computations-required> Acesso em: 20, Outubro de 2019.

MAZUR, Matt. **A step by step backpropagation example**. Disponível em: <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/> Acesso em: 22, Outubro de 2019.

STRANG, Gilbert. **Calculus**. Disponível em: <https://ocw.mit.edu/ans7870/resources/Strang/Edited/Calculus/Calculus.pdf> Acesso em: 23, Outubro de 2019.

6. APÊNDICE – Modelagem Matemática do Problema

Esta seção apresenta as primeiras etapas no desenvolvimento de uma simulação computacional que seria capaz de avaliar a eficiência do 3MI por meio da modelagem matemática do problema de dar autonomia a um robô transportador. Foi empregado o módulo *graphics.h* (), por ser compatível com a linguagem em que é apresentado o 3MI e por permitir a criação de formas geométricas na tela (como em um plano cartesiano). Os passos necessários estão listados como segue.

1. Tornar os módulos escritos em Python compatíveis com o módulo *graphics.py*, a fim de criar visualizações gráficas do sistema em funcionamento;
2. Treinar a rede neural com *dataset* obtido manualmente, por meio da interface gráfica;
3. Avaliar o desempenho do 3MI simulando seu funcionamento em mapas distintos;

Para tornar compatíveis os módulos do 3MI e do graphics.py é necessário decidir o que precisa ser implementado, já que o graphics.py possui diversos métodos que dispensam métodos correspondentes nos módulos do 3MI. Dessa forma, deve-se levar em conta que a simulação receberá um mapa e retornará o mesmo mapa com um caminho traçado. As iterações de desvio de colisões e correção de rotas serão efetuadas matematicamente, sendo necessário que ao final do caminho percorrido os segmentos de reta estejam armazenados para análise futura.

O treinamento da rede neural deve extrair situações (isto é, configurações de entradas) diretamente dos mapas simulados. Para tanto, leva-se em conta que o sensor simulado tem um range de 15° e alcance de 4 metros. Estes valores deverão ser adaptados segundo uma escala viável, que permita traçar rotas consideráveis em mapas de complexidade razoável. A simulação do sensor pode ser feita como um segmento de reta que é medido até o primeiro ponto de intersecção ou até o limite do sensor. Este treinamento se dará com a seleção de um ou mais mapas seguida pela seleção manual de pontos neste mapa, o que deverá coletar pares de vetores (input, target) correspondentes à situação em que cada clique foi efetuado; em seguida, com pares suficientes, a rede neural será treinada até que se alcance um nível de eficácia considerável.

Com a rede neural treinada, esta será avaliada quando simulada em uma coleção de mapas não inclusos na fase de treinamento. Os resultados devem ser correspondentes à

eficácia do treinamento, o que está diretamente associado ao número de exemplos (input, target) que foram coletados. Dessa forma, a fase de treinamento vem a ser não só mais exaustiva, mas decisiva na avaliação de desempenho do 3MI.

Adaptação dos Módulos para Simulação

Módulo de Tomada de Decisão: O funcionamento do Módulo de Tomada de Decisão (MTD) será idêntico ao que seria utilizado em um sistema físico, com a diferença na maneira como são obtidos e tratados os dados que modelarão a configuração da rede neural. Virtualmente, o sistema não dispõe de medições reais, efetuadas por sensores em um ambiente industrial, o que torna necessária a modelagem matemática de um ambiente que seria considerado, de acordo com a disposição de obstáculos, correspondente ao que seria uma planta industrial real. Esta planta simulada seria composta apenas por um espaço bidimensional ocupado, em posições estratégicas, por obstáculos retangulares. O robô seria uma entidade com uma seta de orientação e doze feixes de comprimento específico. As medições que, no mundo real, seriam interpretadas por sensores, corresponderiam aos feixes do robô disponíveis no espaço navegável, *i.e.*, os comprimentos dos feixes seriam limitados pelos obstáculos.

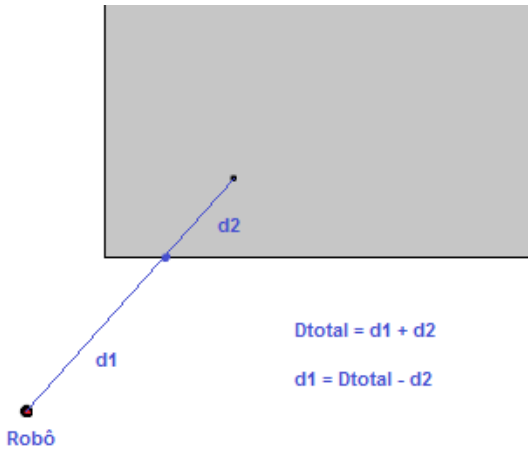
Apesar desta diferença, o MTD não necessita ser alterado, tendo em vista que sua implementação permite uma chamada simples com uma série de vetores distribuídos ao longo do tempo (neste caso, distribuídos um por ciclo) e sua saída é uma entre seis possíveis e esse minimalismo presente tanto na entrada quanto na saída do MTD dispensa alterações no código referente.

Módulo de Mapeamento Cartesiano: O cartesius.py é constituído de funções para representação geométrica do percurso de um robô rodando o 3MI, mas, no caso de uma simulação, deverão ser feitas alterações para representar, também, o espaço simulado. A simulação de um sistema de navegação com um personagem autônomo lembra a implementação de uma *game engine*, mas, neste caso, não haverá processos de animação; todos os movimentos deverão ser calculados e armazenados em estruturas de dados passíveis de constante alteração; neste caso, o cartesius.py deve ser capaz de representar uma entidade móvel, de onde emanam doze feixes de comprimento definido; o espaço simulado onde circulará esta entidade é constituída de uma

combinação de dois tipos diferentes de campo: o espaço navegável e o espaço obstáculo. A ideia é medir o quanto cada feixe invade o espaço obstáculo, obtendo, assim, doze medições. Além de representações, o cartesius.py deve dispor de funções que efetuem transformações rígidas nos elementos geométricos, tendo em vista que o robô deverá ser capaz de girar sobre o próprio eixo em um ângulo definido pela saída da rede neural. A adaptação do cartesius.py para a simulação computacional está disponível no Anexo VI.

A medição de cada feixe se dará através de uma rotina que primeiramente determina se o ponto externo do feixe está no espaço obstáculo, em seguida medindo a distância desse ponto até o limite do obstáculo (para simplificar a simulação, só se lidará com retângulos); subtraindo-se do comprimento total do feixe este comprimento, obtém-se uma medição simulada. Para que seja possível efetuar tal procedimento é preciso conhecer exatamente onde se localiza o ponto de intersecção entre o feixe e o limite do obstáculo. É possível conhecer esse ponto por meio da aplicação da ordenada conhecida na equação geral da reta que perpassa o feixe simulado. Resolvendo-se a equação encontra-se a outra ordenada do ponto, o que nos permite, então, medir a distância deste até o ponto final do feixe. Uma representação esquemática pode ser vista na Figura 6, logo abaixo.

Figura 8 – Medição Simulada

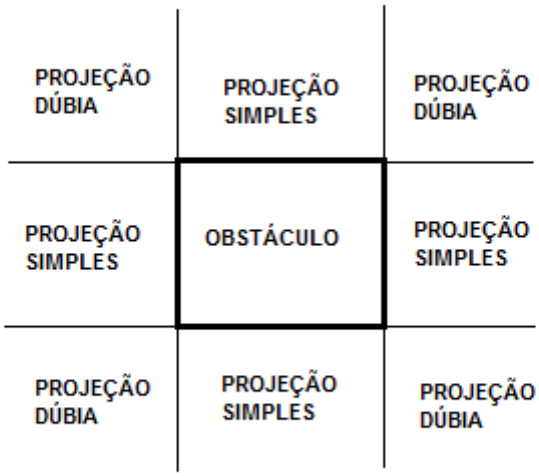


Fonte: O autor

Entretanto, encontrar o ponto de interseção que permite medir o comprimento útil do feixe não é uma tarefa trivial. Como se pode observar na Figura 7, os obstáculos simulados dividem o espaço em oito regiões, sendo quatro delas regiões onde as

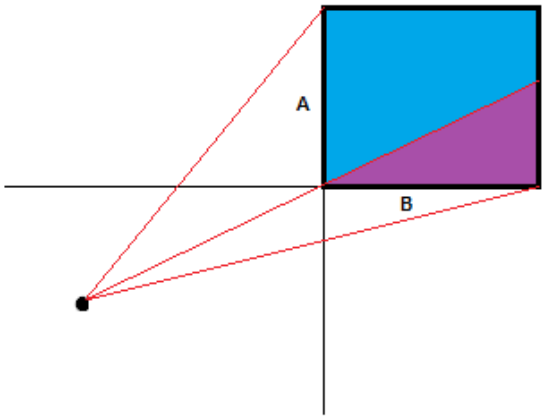
projeções de possíveis intersecções são facilmente determináveis, e outras quatro, denominadas externas ao obstáculo, que não encontram projeções em uma das faces do obstáculo. Dessa forma, é necessário, para se encontrar o ponto de intersecção do feixe com o objeto, determinar que face do objeto é interceptada pelo feixe. Em regiões facilmente determináveis esta face é intuitivamente encontrada, mas em regiões externas ao obstáculo é necessário, como indica a Figura 8, determinar regiões secundárias, delimitadas por segmentos de reta com origem no robô e com fim nos três vértices mais próximos do robô. O quadrilátero resultante determina duas regiões, onde, agora sim, é possível determinar em que face um dado feixe intercepta tal objeto.

Figura 9 – Regiões de Projeção do Obstáculo



Fonte: O autor

Figura 10 – Projeções Dúbias



Fonte: O autor

Como pode-se observar, os três segmentos de reta delimitam duas regiões no obstáculo. Se o ponto final de um dado feixe está contido na região em azul, a interseção que procuramos está na face A, de maneira análoga se esse ponto está na região em roxo, o que significaria que a intersecção está na face B. Com estas implicações geométricas foram implementados métodos na classe SimulatedRobot (Anexo V) que computam estas informações funcionando ativamente com as classes da biblioteca graphics.py medições simuladas, no Módulo de Mapeamento Geométrico Simulado (cartesiusim.py), disponível no Anexo V.

Outra propriedade da classe SimulatedRobot, a saber, a disposição de doze pontos (representando o alcance máximo do sensor simulado) posicionados de modo que sejam separados por 15°, deve ser ressaltada como indispensável na modelagem matemática de um dado robô e seus sensores. A obtenção dos pontos é dada pela utilização de coordenadas polares, que serão aplicadas em cada ponto e, enfim, convertidas para o plano R^2 . Tais operações podem ser efetuadas sob a biblioteca numpy.py (numpy.org).

Vale ressaltar que, como um apêndice, esta seção não trata de uma metodologia concluída, mas de uma proposta de implementação para o que seria a efetivação científica das redes neurais artificiais como aparato eficiente na automação de robôs industriais. As classes presentes neste módulo carecem de correção e conclusão, de maneira que não devem ser consideradas senão como motivação, ou mesmo como exemplo, e não como um projeto concluído.

7. ANEXOS

ANEXO I: Forma generalizada da equação 13

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \cdot \frac{\partial out_{h1}}{\partial ¿_{h1}} \cdot \frac{\partial ¿_{h1}}{\partial w_1} \quad (14)$$

$$\frac{\partial E_{total}}{\partial w_1} = \left(\frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}\right) \cdot \frac{\partial out_{h1}}{\partial ¿_{h1}} \cdot \frac{\partial ¿_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \left(\left(\frac{\partial E_{o1}}{\partial ¿_{o1}} \cdot \frac{\partial ¿_{o1}}{\partial out_{h1}}\right) + \left(\frac{\partial E_{o2}}{\partial ¿_{o2}} \cdot \frac{\partial ¿_{o2}}{\partial out_{h1}}\right)\right) \cdot \frac{\partial out_{h1}}{\partial ¿_{h1}} \cdot \frac{\partial ¿_{h1}}{\partial w_1}$$

$$\frac{\partial E_{¿tal}}{\partial w_1} = \left[\left(\frac{\partial E_{o1}}{\partial ¿_{o1}} \cdot w_5\right) + \left(\frac{\partial E_{o2}}{\partial ¿_{o2}} \cdot w_6\right)\right] \cdot \left[out_{h1}(1 - out_{h1})\right] \cdot [¿_1]$$

$$\frac{\partial E_{total}}{\partial w_1} = \left[\left(\frac{\partial E_{o1}}{\partial out_{o1}} \cdot \frac{\partial out_{o1}}{\partial ¿_{o1}} \cdot w_5\right) + \left(\frac{\partial E_{o2}}{\partial out_{o2}} \cdot \frac{\partial out}{\partial ¿_{o1}} \cdot w_6\right)\right] \cdot \left[out_{h1}(1 - out_{h1})\right] \cdot [¿_1]$$

$$\frac{\partial E_{total}}{\partial w_1} = \left[\left(\left(out_{o1} - target_{o1}\right) \cdot \left(out_{o1}(1 - out_{o1})\right) \cdot w_5\right) + \left(\left(out_{o2} - target_{o2}\right) \cdot \left(out_{o2}(1 - out_{o2})\right) \cdot w_6\right)\right] \cdot \left[out_{h1}(1 -\right.$$

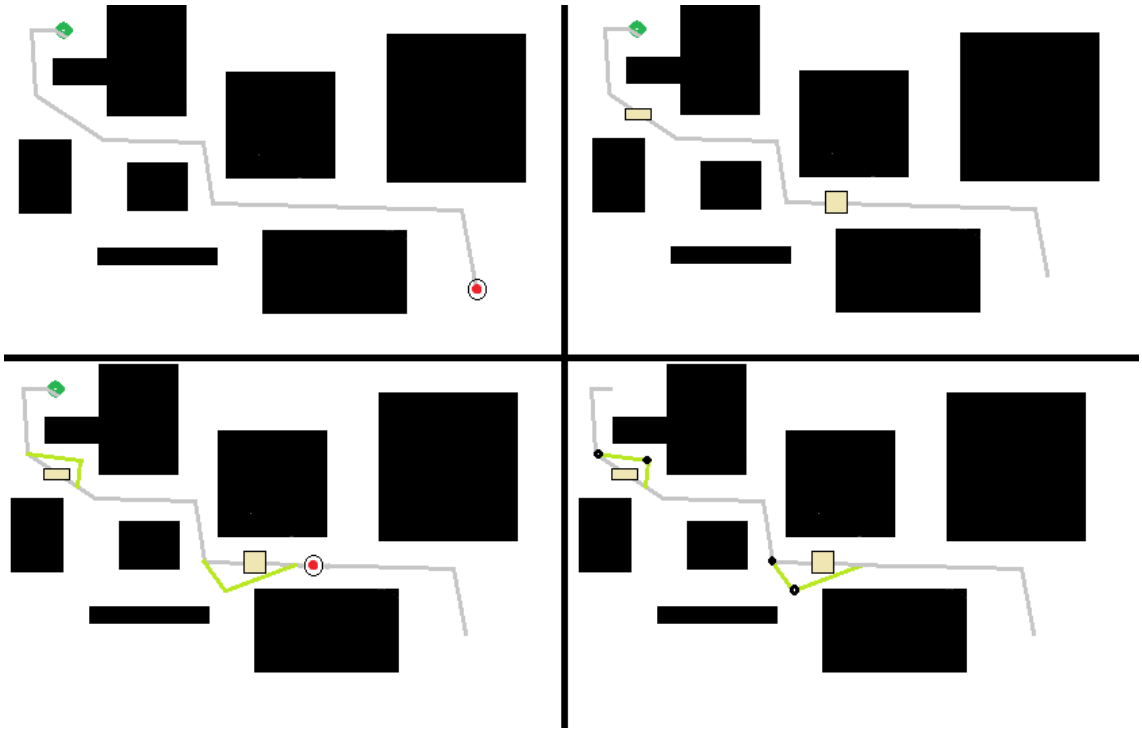
$$\frac{\partial E_{total}}{\partial w_h} = \sum \left[\left(\left(out_o - target_o\right) \cdot \left(out_o(1 - out_o)\right) \cdot w_{ho}\right)\right] \cdot \left[out_h(1 - out_h)\right] \cdot [¿_h] \quad (15)$$

ANEXO II: cartesius.py

```
import math

class Point:
    def __init__(self, X, Y):
        self.x = X
        self.y = Y

    def through(self, point):
        line_through_points = Line((self.y - point.y), (point.x - self.x),
((self.x * point.y) - (point.x * self.y)))
        return line_through_points

    def distance_to(self, point):
        deltaX = self.x - point.x
        deltaY = self.y - point.y
        s = (deltaX * deltaX) + (deltaY * deltaY)
        distance = math.sqrt(s)
        return distance

    def set(self, x, y):
        self.x = x
        self.y = y

    def get(self):
        return [self.x, self.y]

    def X(self):
        return self.x

    def Y(self):
        return self.y


class Line:

    def __init__(self, A, B, C):
        self.A = A
        self.B = B
        self.C = C
        self.show()

    def f(self, x):
        return -(self.C + (self.A * x)) / self.B

    def get_x(self, y):
        return -(self.C + (self.B * y)) / self.A

    def show(self):
        print(self.A, 'x + ', self.B, 'y + ', self.C, '= 0')


class Segment:
    def __init__(self, line, origin, destiny):
        self.line = line
        self.origin = origin
        self.destiny = destiny


class Route:
    def __init__(self, way):
        self.way = way

    def appendSegment(self, segment):
        self.way.append(segment)

    def nextStep(self, matched_points):
```

```
        return self.way[matched_points]

    def deviation(self, matched_points, end, unknown_point):
        new_route = Segment(unknown_point.through(end), unknown_point, end)
        self.setRight(matched_points, new_route)

    def setRight(self, matched_points, wrng_segment):
        next_point = self.nextStep(matched_points).destiny
        right_line = wrng_segment.destiny.through(next_point)
        right_segment = Segment(right_line, wrng_segment.destiny, next_point)
        return right_segment
```

ANEXO III: Funcionamento Geral



ANEXO IV: MTD.py

```
## Módulo de Tomada de Decisão do 3MI
## Autor: Joás de Brito
#
# Possui funções que criam, treinam e executam FFNNs com uma camada oculta
#
# A rede neural é representada como uma lista de camadas;
# Cada camada é representada como uma lista de neurônios;
# Cada neurônio é representado como uma lista de pesos;
# Cada peso é representado como um número real.
#
# neuron  == weights list
# layer   == neurons list
# network == layers list
#
#


from math import exp
import random


# retorna uma lista de listas no formato network = [[hidden_layer],
[output_layer]]
# os pesos são inicializados aleatoriamente no range [0,1]
# os biases são inicializados como b = 0
def generate_network(in_n, hid_n, out_n):  # número de neurônios nas camadas
    hidden_layer = []
    for h_neuron in range(hid_n):
        neuron = []

        for h_weight in range(in_n):
            neuron.append(random.random())
        neuron.append(0)  # bias weight
        hidden_layer.append(neuron)

    output_layer = []
    for o_neuron in range(out_n):
        neuron = []
        for o_weight in range(hid_n):
            neuron.append(random.random())
        neuron.append(0)  # bias weight
        output_layer.append(neuron)

    network = [hidden_layer, output_layer]
    print("HIDDEN Layer: ")
    print(hidden_layer)
    print("OUTPUT Layer: ")
    print(output_layer)
    return network


# calcula a entrada de um dado neurônio
def weighted_sum(weights, inputs):
    output = weights[-1]  # bias
    for i in range(len(weights) - 1):
        output += inputs[i] * weights[i]
    return output


def sigmoid(x):
    return 1 / (1 + exp(-x))


# calcula a saída de uma rede para uma dada entrada
def forward_propagation(network, inputs):
    h_output = []
    for neuron in network[0]:
```

```
        h_output.append(sigmoid(weighted_sum(neuron, inputs)))
    output = []
    for neuron in network[1]:
        output.append(sigmoid(weighted_sum(neuron, h_output)))
    return output


# backpropagation: para uma FFNN de arquitetura AxBxC, retorna uma nova
configuração de pesos e biases
def backward_propagation(network, inputs, targets, alpha):
    n_of_samples = len(targets)
    _h_in = list()  #
    _h_out = list()  # variaveis
    _o_in = list()  #
    _o_out = list()  #
    _err = list()  #

    _w_ih = network[0]  # input_to_hidden weights
    _w_ho = network[1]  # hidden_to_output weights
    counter = 0

    new_network_setup = network
    averaged_new_network_setup = network
    for layer in range(2):
        for neuron in range(len(new_network_setup[layer])):
            for weight in range(len(neuron)):
                averaged_new_network_setup[layer][neuron][weight] = 0.0

    for _target in targets:  # para cada sample do batch
        _input = inputs[counter]
        outputs = forward_propagation(network, _input)
        for neuron in network[0]:
            weighted_sum = weighted_sum(neuron, _input)
            _h_in.append(weighted_sum)
            _h_out.append(sigmoid(weighted_sum))

        for neuron in network[1]:
            weighted_sum = weighted_sum(neuron, _h_out)
            output = sigmoid(weighted_sum)
            _o_in.append(weighted_sum)
            _o_out.append(output)

        for i in range(len(_target)):
            _err.append(error_function(_o_out[i], _target[i], i))
            print("Output Neuron ", i, "Error: ", _err[i])

        print("Total network output error: ",
error_function(forward_propagation(network, _input), _target))

        # output layer
        j_range = len(_o_out)
        k_range = len(network[1][0])
        for j in range(j_range):  # six neurons
            for k in range(k_range):  # eleven weights
                partial_derivative_E_ho = (_o_out[j] - _target[j]) *
(_o_out[j] * (1 - _o_out[j])) * (
                _h_out[j])  # hidden_to_output weights
                new_network_setup[1][j][k] -= alpha * partial_derivative_E_ho
# weight update

        # hidden layer
        j_range = len(_h_out)
        k_range = len(network[0][0])
        l_range = len(network[1])  # six output neurons == six weights
        partial_derivative_E_ih = 0.0
        for j in range(j_range):  # ten hidden neurons
            for k in range(k_range):  # thirty weights
                aux = (_h_out[j] * (1 - _h_out[j]) * weighted_sum(_w_ih[j], ))
```

```
                o_sum = 0.0
                for l in range(l_range):
                    o_sum += (_o_out[l] - _target[l]) * (_o_out[l] * (1 -
_o_out[l])) * (_w_ho[j][l])
                partial_derivative_E_ih = aux * o_sum
                new_network_setup[0][j][k] -= alpha * partial_derivative_E_ih
# weight update

        for layer in range(2):
            for neuron in range(len(new_network_setup[layer])):
                for weight in range(len(neuron)):
                    averaged_new_network_setup[layer][neuron][weight] +=
new_network_setup[layer][neuron][weight]

        counter = counter + 1

    for layer in range(2):
        for neuron in range(len(new_network_setup[layer])):
            for weight in range(len(neuron)):
                averaged_new_network_setup[layer][neuron][weight] /=
n_of_samples

    return averaged_new_network_setup


# squared error function
def error_function(output, target, index):
    error = 0.0
    for i in range(len(output)):
        if (index == i and index != -1):
            return pow((output[i] - target[i]), 2) / 2
        error += pow((output[i] - target[i]), 2) / 2


def train(network, dataset, alpha, epochs):
    batch = 0
    new_network = network
    print("Training with alpha = ", alpha)

    for epoch in range(epochs):
        for batch in range(len(dataset)):
            print("batch: ", epoch)
            inputs = dataset[batch][0]
            targets = dataset[batch][1]
            new_network = backward_propagation(new_network, inputs, targets,
alpha)
    print("Training completed!")
    return new_network
```

ANEXO V: Módulo Principal

```
# importa os módulos de mapeamento cartesiano, tomada de decisão e integração
com hardware
import cartesius, neural_network, machine


# função principal
def main(ROUTE, operation_mode, machine):
    if (operation_mode == 'standard'):
        # inicialização (ponto zero)
        matched_points = 0

        # enquanto o botão de ligar estiver pressionado (sinal = 1)
        while (machine.input.ON_OFF()):

            # escanear ambiente
            SCAN = neural_network.run(machine.input.scan())

            # enquanto a rede neural não indicar rota de colisão
            while (SCAN != 0):

                # seguir a rota armazenada
                machine.output.move(ROUTE.nextStep(matched_points)
                matched_points = matched_points + 1

                # desviar da rota de colisão
                wrong_segment = machine.output.moveWithDeviation(SCAN)

                # corrigir o desvio efetuado
                machine.output.move(ROUTE.setRight(matched_points,
wrong_segment))
                matched_points = matched_points + 1

                if (operation_mode == 'learning'):
                # enquanto a rota não for salva

                while (machine.input.save_route == False):
                    # adicionar segmento à rota
                    ROUTE.appendSegment(machine.input.lastSegment())


if (__name__ == '__main__'):
    main()
```

ANEXO VI: cartesiusim.py

```
## Módulo de Mapeamento Geométrico do 3MI simulado
## Autor: Joás de Brito
#
#
import graphics as g
import math
import numpy as np
import neural_network

def cart2pol(x, y):
    rho = np.sqrt(x**2 + y**2)
    phi = np.arctan2(y, x)
    return(rho, phi)

def pol2cart(rho, phi):
    x = rho * np.cos(phi)
    y = rho * np.sin(phi)
    return [x, y]

def rad(angle):
    if(angle == 0):
        return 0
    return (math.pi * float(angle))/(180)

class Point:
    def __init__(self, X, Y):
        self.x = X
        self.y = Y

    def through(self, point):
        line_through_points = Line((self.y - point.y), (point.x - self.x),
((self.x * point.y) - (point.x * self.y)))
        return line_through_points

    def distance_to(self, point):
        deltaX = self.x - point.x
        deltaY = self.y - point.y
        s = (deltaX * deltaX) + (deltaY * deltaY)
        distance = math.sqrt(s)
        return distance

    def set(self, x, y):
        self.x = x
        self.y = y

    def get(self):
        return [self.x, self.y]

    def X(self):
        return self.x

    def Y(self):
        return self.y


class Line:

    def __init__(self, A, B, C):
        self.A = A
        self.B = B
        self.C = C
        self.show()

    def f(self, x):
```

```
        return -(self.C + (self.A * x)) / self.B

    def get_x(self, y):
        return -(self.C + (self.B * y)) / self.A

    def show(self):
        print(self.A, 'x + ', self.B, 'y + ', self.C, '= 0')


class Segment:
    def __init__(self, line, origin, destiny):
        self.line = line
        self.origin = origin
        self.destiny = destiny


################################################################################
######################

class Space:
    def __init__(self, extension):
        self.height = extension[0]
        self.width = extension[1]
        self.objects = []

    def insert_object(self, pointA, pointC):  # todo objeto é um retângulo
descrito por quatro vértices
        pointB = Point(pointA.X, pointC.Y)
        pointD = Point(pointC.X, pointA.Y)
        self.objects.append([pointA, pointB, pointC, pointD])

#define atributos e métodos de um robô simulado
class SimulatedRobot:
    def __init__(self, initial_position, unidade, window):
        self.position = initial_position   # posição inicial do robô
        self.direction = Point(initial_position.X(), (initial_position.Y() +
(7*unidade)))  # ponto que indica a orientação
        self.angles = []
        self.drawable_points = []
        self.drawable = []
        self.first_p = 0  # 12 pontos, onde o primeiro é o self.angle[0]

        self.w = window
        for i in range(24):
            cartPoint = pol2cart((5*unidade), rad(7.5 + (i*15.0)))
            x = self.position.X() + cartPoint[0]
            y = self.position.Y() + cartPoint[1]
            self.angles.append(Point(x, y))
            self.drawable_points.append(g.Point(x, y))
        center = g.Point(self.position.X(), self.position.Y())
        circle = g.Circle(center, 5)
        circle.setFill("red")
        direction = g.Point(self.direction.X(), self.direction.Y())
        self.drawable = [circle, direction]
        self.d_angles = []
        for i in range(12):
            p = pol2cart(7 * unidade, rad(i * 30))
            p = g.Point(p[0] + self.position.X(), p[1] + self.position.Y())
            print(p)
            self.d_angles.append(p)
        if (i == 3):
            self.d_angles[3].setOutline("green")
            self.drawable[1] = self.d_angles[3]
        d = []
        for i in range(12):
            d.append(self.d_angles[i])
        self.drawable.append(d)
        self.draw_robot()
```

```
    def get_range(self, op):
        points = []
        start = self.first_p
        end = start + 12
        plus = 0

        #ciclic from 0 to 23
        if(end > 23):
            end -= 24
            if(end < start):
                plus = end
                end = 24
        if(op == 0):
            return [range(start, end), plus]
        else:
            for i in range(start, end+1):
                points.append(self.angles[i])
            return points

    #determina, em um objeto, a face interceptada pelo sensor
    def get_spacial_projection(self, rect_points, point):
        robot = self.position
        left_up = rect_points[0]
        left_down = rect_points[1]
        right_down = rect_points[2]
        right_up = rect_points[3]

        if (robot.Y() >= left_up.Y() and robot.Y() <= left_down.Y()):  #
left/right
            if (robot.X() < right_down.X()):
                return [left_up, left_down]
            else:
                return [right_up, right_down]

        elif (robot.X() >= left_up.X() and robot.X() <= right_down.X()):  #
above/below
            if (robot.Y() < left_up.Y()):
                return [left_up, right_up]
            else:
                return [left_down, right_down]

        #left_up
        if(robot.X() < left_up.X() and robot.Y() > left_up.Y()):
            limit = self.position.through(left_up)  # linha que delimita os
dois campos de possibilidade
            if (point.Y() == limit.f(point.X())):
                return [left_up, left_up]
            if(point.Y() < limit.f(point.X())):
                return [left_up, left_down]
            else:
                return [left_up, right_up]

        #left_down
        if(robot.X() < left_down.X() and robot.Y() < left_down.Y()):
            limit = self.position.through(left_down)  # linha que delimita os
dois campos de possibilidade
            if (point.Y() == limit.f(point.X())):
                return [left_down, left_down]
            if(point.Y() < limit.f(point.X())):
                return [left_down, right_down]
            else:
                return [left_up, left_down]

        #right_down
        if(robot.X() > right_down.X() and robot.Y() < right_down.Y()):
            limit = self.position.through(right_down)
            if (point.Y() == limit.f(point.X())):
```

```
                return [right_down, right_down]
            if(point.Y() < limit.f(point.X())):
                return [left_down, right_down]
            else:
                return [right_up, right_down]

        # right_up
        if(robot.X() > right_up and robot.Y() > right_up):
            limit = self.position.through(right_up)
            if(point.Y() == limit.f(point.X())):
                return [right_up, right_up]
            if(point.Y() < limit.f(point.X())):
                return [right_up, right_down]
            else:
                return [left_up, right_up]

    #retorna o valor obtido pelo sensor
    def measure(self, point, space):
        intersection = point
        for i in range(len(space.objects)):  # procura o objeto em que está
situado a ponta do feixe
            if (point.Y() >= space.objects[i][0].Y() and point.Y() <=
space.objects[i][1].Y()):  # verifica se o ponto do feixe
                if (point.X() >= space.objects[i][0].X() and point.X() <=
space.objects[i][1].X()):  # está contido no objeto
                    line = self.position.through(point)  # equação que contêm
o feixe
                    intercepted_face =
self.get_spacial_projection(space.objects[i], point)  # face interceptada pelo
feixe
                    if(intercepted_face[0] == intercepted_face[1]): #  a face
é diagonal
                        intersection = intercepted_face[0]
                    elif(intercepted_face[0].X == intercepted_face[1].X):  # a
face é vertical
                        x = intercepted_face[0].X()
                        y = line.f(x)
                        intersection = Point(x, y)
                    elif(intercepted_face[0].Y() == intercepted_face[1].Y()):
# a face é horizontal
                        y = intercepted_face[0].Y()
                        x = line.get_x(y)
                        intersection = Point(x, y)
                    break
        if(intersection == point):#caso o feixe não encontre nenhum objeto
            return 0
        distance = self.position.distance_to(intersection)
        return distance

    #retorna uma lista com as medições dos doze sensores
    def get_measurements(self, space):
        measurements = []
        points = self.get_range()
        for i in range(12):
            m = self.measure(points[i], space)
            measurements.append(m)

    def draw_robot(self):
        self.drawable[0].draw(self.w)
        self.drawable[1].draw(self.w)
        for i in range(12):
            self.drawable[2][i].draw(self.w)

        r = self.get_range(0)
        for i in r[0]:
            self.drawable_points[i].setOutline("red")
        for i in range(r[1]):
            self.drawable_points[i].setOutline("red")
```

```
        for i in range(24):
            self.drawable_points[i].draw(self.w)

    def undraw_robot(self):
        for i in range(24):
            self.drawable_points[i].undraw()
        self.drawable[0].undraw()
        self.drawable[1].undraw()
        for i in range(12):
            self.drawable[2][i].undraw()

    #rotaciona os feixes do robô de acordo com units
    def rotate(self, units):
        self.undraw_robot()
        d = self.direction
        self.direction = Point(self.d_angles[3+units].getX(),
self.d_angles[3+units].getY())

        self.drawable[1] = g.Point(d.X(), d.Y())

        self.first_p += units
        if(self.first_p > 23):
            self.first_p -= 23

        self.draw_robot()

    def move(self, deltaX, deltaY):
        self.undraw_robot()
        d = self.direction
        self.position.set(self.position.X()+deltaX, self.position.Y()+deltaY)
        self.direction = Point(d.X() + deltaX, d.Y() + deltaY)

        for i in range(12):
            p = self.drawable[2][i]
            self.drawable[2][i] = g.Point(p.getX(), p.getY())

        for i in range(24):
            p = self.angles[i]
            self.angles[i] = Point(p.X() + deltaX, p.Y() + deltaY)
            dp = self.drawable_points[i]
            self.drawable_points[i] = g.Point(dp.getX()+deltaX, dp.getY()
+deltaY)

        drawable = self.drawable
        x = drawable[0].getCenter().getX() + deltaX
        y = drawable[0].getCenter().getY() + deltaY
        new_center = g.Point(x, y)
        circle = g.Circle(new_center, 5)
        circle.setFill("red")
        x = drawable[1].getX() + deltaX
        y = drawable[1].getY() + deltaY
        direction = g.Point(x, y)
        self.drawable = [circle, direction]

        self.draw_robot()

    #def set_direction(self, ):

################################################################################

class Route:
    def __init__(self, way):
        self.way = way

    def appendSegment(self, segment):
        self.way.append(segment)
```

```
    def nextStep(self, matched_points):
        return self.way[matched_points]

    def deviation(self, matched_points, end, unknown_point):
        new_route = Segment(unknown_point.through(end), unknown_point, end)
        self.setRight(matched_points, new_route)

    def setRight(self, matched_points, wrng_segment):
        next_point = self.nextStep(matched_points).destiny
        right_line = wrng_segment.destiny.through(next_point)
        right_segment = Segment(right_line, wrng_segment.destiny, next_point)
        return right_segment
```