Exercício de Fixação de Conceitos 2

EA072 - Inteligência Artificial em Aplicações Industriais

Gustavo **CIOTTO PINTON** - **RA 117136**
Campinas, 1 de novembro de 2015

1 Síntese de controle PID

1. Antes de definirmos os operadores de mutação e *crossover*, é necessário diferenciarmos estes dois conceitos. Embora ambos ocorram em organismos biológicos e sejam responsáveis pela alteração do genótipo de um indivíduo, estes mecanismos ocorrem em diferentes etapas da vida do respectivo organismo. O termo *mutação* é utilizado para descrever o processo no qual um alelo de um gene é **aleatoriamente** substituído ou modificado por outro. Em termos matemáticos, sejam os índices $r, \ldots, u$ as posições que sofrerão mutação e que foram determinadas aleatoriamente. Cada posição possui um probabalidade p_m de ser submetida a este processo. Ao final da mutação, os alelos referentes a estes índices serão modificados, no caso de uma codificação em ponto flutuante, segundo uma distribuição *uniforme* ou *não uniforme*. No caso da *não uniforme*, insere-se uma perturbação com distribuição *normal* com média nula na posição escolhida e desvio-padrão decrescente ao longo das gerações, o que garante consequentemente um refinamento da solução à medida que nos aproximamos de resultados ótimos.

   O processo de *crossover*, por sua vez, trata-se de um mecanismo de recombinação genética de dois ou mais cromossomos. Para a codificação de algoritmos evolutivos em que os cromossomos possuem valores em ponto flutuante, destacam-se duas técnicas: o *crossover* aritmético e o uniforme. Para o primeiro, a partir dos cromossomos $\mathbf{x}$ e $\mathbf{y}$, obtém-se uma combinção convexa dos valores dos genes de $\mathbf{x}$ e $\mathbf{y}$, isto é, o cromossomo $a\mathbf{x} + (1-a)\mathbf{y}$, em que $a \in [0, +1]$. Para o segundo, os genes dos cromossomos pais são escolhidos com igual probabilidade para formar um cromossomo filho. Por exemplo, se a probabilidade vale 0.5, então espera-se que o cromossomo resultante seja composto por metade dos genes de $\mathbf{x}$ e metade de $\mathbf{y}$.

   Outra etapa importante para os algoritmos evolutivos é a seleção dos indivíduos mais "adaptados" ao problema (que possuem maior função de *fitness*). Este processo deve garantir que os melhores indivíduos persistam, mas não deve ser extremamente radical a fim de garantir uma diversidade na população. Um exemplo de um operador de seleção é a técnica de *seleção por torneio*. Para selecionar N indivíduos, realizam-se N torneios com p participantes, escolhidos aleatoriamente. Quanto mais alto é p, maior é a pressão seletiva, isto é, para um indivíduo ruim ser escolhido ao menos em um torneio, é necessário que ele compita com $p-1$ indivíduos piores que ele (para p grande, a probabilidade deste fato ocorrer é muito pequena). Cada torneio é vencido pelo indivíduo que apresenta maior *fitness*. No caso deste exercício, será realizado apenas um torneio, que é composto por 3 indivíduos.

2. As constantes rrier apresentadas na tabela 1 a seguir foram definidas no arquivo `prog_PID.m`, disponibilizado pelo professor.

Tabela 1: Constantes definidas em `prog_PID.m`

Atributos	**Valor**
Tamanho da População	100
Número máximo de Gerações	50
Taxa de Mutação	0.4
Taxa de Crossover	0.8

   Observa-se taxas de mutação e *crossover* relativamente altas: para a primeira, um alelo tem probalidade de 40% de ser mutado, isto é, ele possui aproximadamente a metade das chances de ser modificado. Em relação ao tamanho da população e o número de gerações , 100 e 50 são números suficientemente grandes para garantir, respectivamente, diversidade entre os indivíduos e uma boa qualidade no resultado visto a quantidade de recombinações entre os cromossomos que serão realizadas.

   A população inicial é obtida pelo comando `pop = 5*rand(tam_pop,3);`, em que `rand` é uma função do MATLAB que gera números aleatórios segundo uma distribuição uniforme e

`tam_pop` vale 100. Será criada, portanto, uma matriz de 100 linhas, cada uma representando um indivíduo, e 3 colunas, uma para cada constante a ser determinada k_p, k_d e k_i.

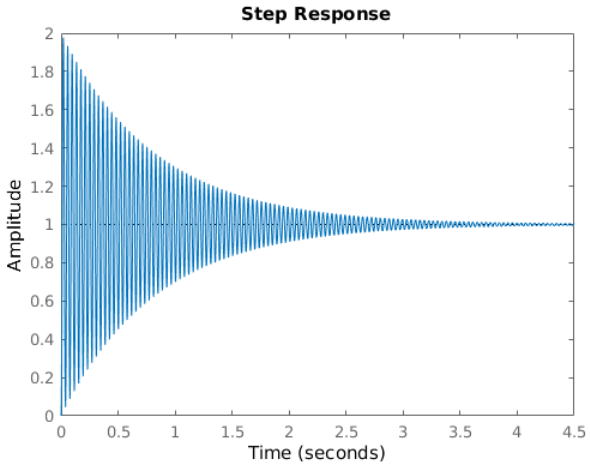
Finalmente, cada operador de *crossover* é escolhido com 50% de probabilidade. Espera-se portanto que o operador aritmétido seja escolhido em metade das oportunidades e o uniforme, também.

3. Após adicionarmos as linhas referentes ao cálculo do *fitness*, as execuções do programa *prog_pid.m* resultam nos dados da tabela 2.

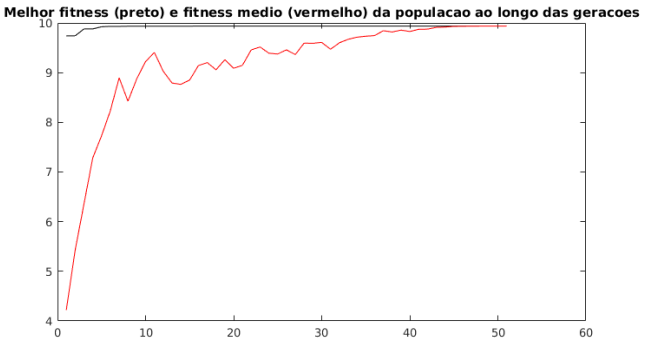
Tabela 2: Resultados das 5 execuções de `prog_PID.m`

Ex.	k_p	k_d	k_i	t_{subida} (s)	$t_{estabilizacao}$ (s)	*Overshoot* (%)	$\Delta\phi$ (°)	*Fitness*
1	4.9935	2.385e-05	4.46633	0.0064396	3.26286	97.7286	1.1935	9.9360
2	4.9960	2.296e-05	3.99303	0.0064393	3.14669	97.6506	1.2340	9.9360
3	4.9923	1.350e-05	3.95399	0.0064397	3.3217	97.7735	1.1687	9.9360
4	4.9940	2.240e-05	4.33452	0.0064395	3.24405	97.7234	1.1960	9.9360
5	4.9990	1.528e-05	4.4244	0.0064348	3.41617	97.8392	1.1346	9.9361

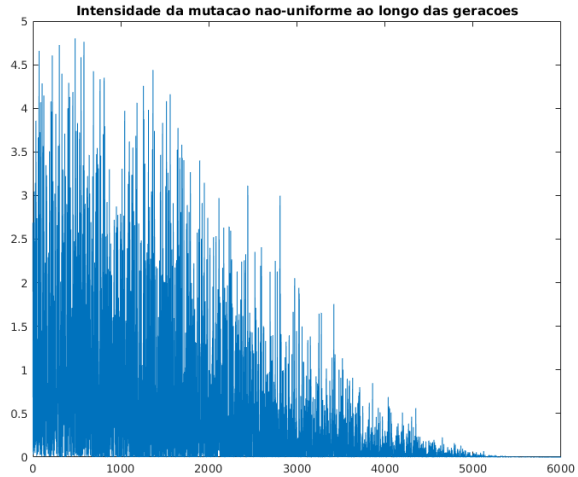
Observa-se que todos os valores k_d relativos à ação derivadora são praticamente nulos e que k_i e k_p encontram-se muito próximos a seus valores máximos, isto é, 5. Isto indica que a ação derivadora não exerce efeitos importantes sobre o tempo de resposta apresentado pelo sistema. Os valores de tempo dos primeiros máximos em todas as execuções concentram-se em torno do valor $t_{subida} = 0.00644$ s. Sabe-se também que este atributo é fortemente ligado à frequência de oscilação que a resposta ao degrau apresentará. Para sistemas de segunda ordem, por exemplo, a relação $w_0 t_m \approx 3$ evidencia o compromisso entre essas duas características. Portanto, para valores muito pequenos de t_{subida}, teremos frequências w_0 elevadas e vice-versa. É justamente esse comportamento que é observado neste caso: pelo fato de que a função de *fitness* só levar em conta o tempo de resposta, obtém-se respostas muito rápidas, porém que oscilam fortemente em torno de 1. A margem de fase ($\Delta\phi$) também possui uma relação com o *overshoot*: pequenas margens de fase produzem sistemas menos estáveis e que apresentam maior erros percentuais entre a entrada e a saída. No nosso caso, como não nos preocupamos com esse parâmetro, obtém-se pequenas margens de fase e, consequentemente, grandes erros antes da estabilização. A figura 1a, que é o resultado de uma das execuções acima, permite a visualização destes comportamentos. A figura 1c, por sua vez, mostra a quantidade de mutações realizadas pelo algoritmo evolutivo no decorrer no programa. Conforme explicado anteriormente, essa quantidade deve ser decrescente, visto que os melhores indíviduos devem ser mantidos nas gerações finais. As figuras 1b e 1d mostram, respectivamente, a evolução dos valores da função de *fitness* e dos parâmetros k_p, k_i e k_d. Para o primeiro, observa-se que o melhor *fitness* permanece sempre próximo de 10 e que o *fitness* médio aproxima-se gradualmente a este valor, indicando que a população esta se "especializando" e adaptando no problema. Para segundo, verifica-se que k_d é praticamente nulo em todas as gerações, k_p se estabiliza a partir da geração 10 e k_i varia fortemente até a geração 40. Estes fatos nos dizem que o parâmetro que mais influencia o tempo do primeiro máximo é a ação integradora.



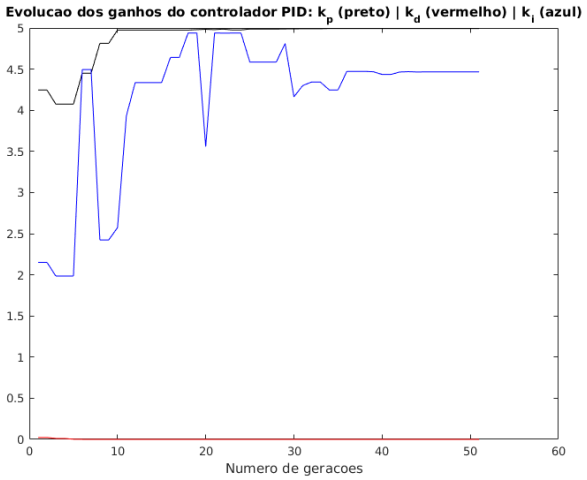
(a) Resposta ao degrau do sistema.



(b) Evolução dos valores da função *fitness*.



(c) Evolução da intensidade de mutações ao longo das gerações.



(d) Evolução dos parâmetros.

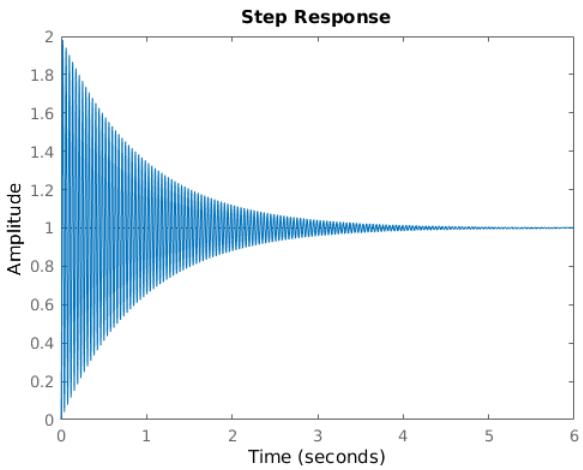
Figura 1: Resultado para a execução 1 do algoritmo evolutivo.

A modificação dos parâmetros da tabela 1 pode influenciar nos resultados acima. Se dobrarmos o número de indivíduos na população, isto é, 200, e aumentarmos as taxas de crossover e mutação para, respectivamente, 0.9 e 0.6, obtém-se os dados contidos na tabela 3.

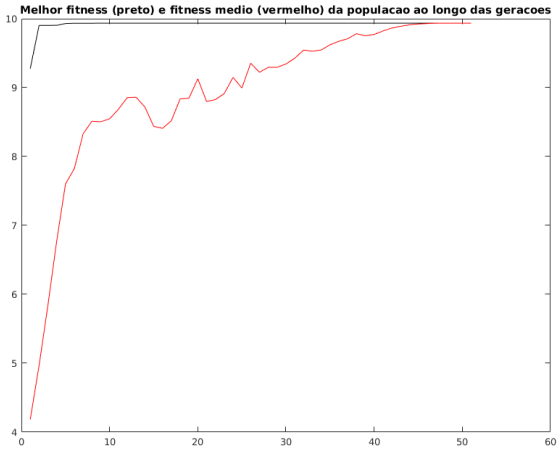
Tabela 3: Resultados da execução com os parâmetros modificados.

k_p	k_d	k_i	t_{subida} (s)	$t_{estabilizacao}$ (s)	*Overshoot* (%)	$\Delta\phi$ (º)	*Fitness*
4.9975	7.80994e-06	4.6839	0.00643281	3.68641	97.9919	1.05405	9.93608

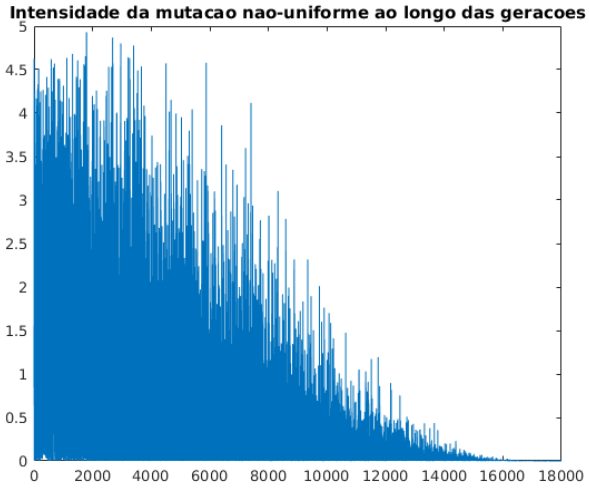
Observa-se que o tempo do primeiro máximo é ligeiramente inferior aos dos casos anteriores e, consequentemente, a função de *fitness* é superior. Os valores dos parâmetros continuam parecidos, sendo que k_d é inferior e k_i superior. De maneira geral, o aumento de recursos computacionais disponibilizados ao programa (aumento de população e quantidade de mutações) não causaram grandes diferenças nos resultados. As figuras 2b e 2d apresentam o mesmo comportamento descrito no parágrafo anterior e a 2c destaca que a quantidade de mutações realizadas foi muito superior aos casos anteriores: observa-se que a presença de quantidades superiores a 10000 quando a taxa de mutação é 0.6.



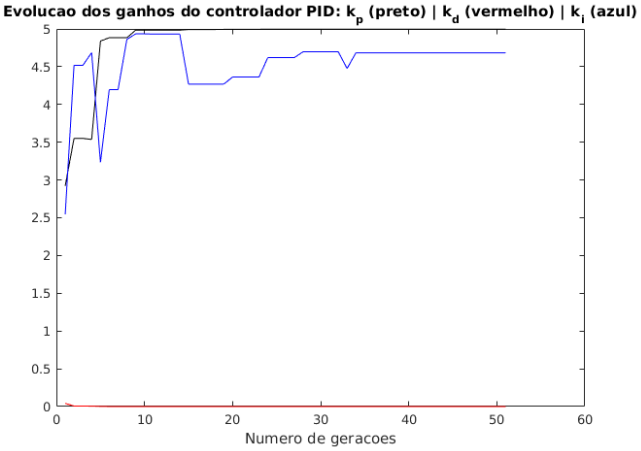
(a) Resposta ao degrau do sistema.



(b) Evolução dos valores da função *fitness*.



(c) Evolução da intensidade de mutações ao longo das gerações.



(d) Evolução dos parâmetros.

Figura 2: Resultado para a execução do algoritmo evolutivo com parâmetros modificados.

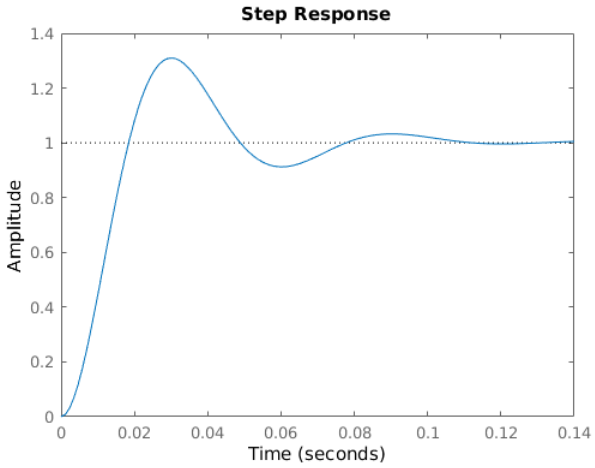
4. Os resultados das 5 execuções com a nova função de *fitness* estão presentes na tabela 4, logo abaixo. Dessa vez, considera-se também a margem de fase.

Tabela 4: Resultados das 5 execuções de `prog_PID.m`

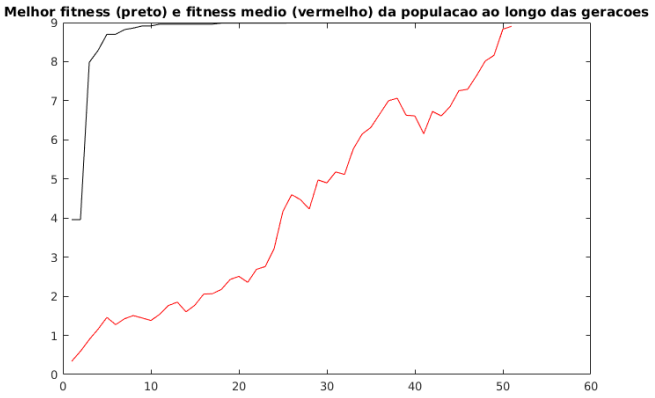
Ex.	k_p	k_d	k_i	t_{subida} (s)	$t_{estabilizacao}$ (s)	*Overshoot* (%)	$\Delta\phi$ (°)	*Fitness*
1	1.22881	0.718353	4.54691	0.705398	12.7319	56.4502	59.9346	5.53763
2	1.15518	0.57906	4.97227	0.605573	10.924	56.4079	59.9872	5.86308
3	0.812793	0.298811	4.76622	0.44429	8.01846	56.4355	59.9522	6.47451
4	1.67979	1.23205	4.94384	0.885861	15.9772	56.398	56.398	5.0356
5	2.36522	0.00523679	1.84369	0.0124718	0.100391	30.9773	59.986	8.98883

Ao contrário do item anterior, verifica-se neste caso uma grande variação nos melhores resultados nas diferentes execuções. As soluções 1 e 2 apresentam soluções muito parecidas em todos os aspectos. A solução 4 foi a pior a ser encontrada e possui o maior valor de k_d encontrado. Isso leva a crer que grandes valores deste parâmetro influenciam negativamente no resultado, principalmente em relação à margem de fase: a pior margem foi encontrada nesta execução. A solução 5 foi aquela que melhor encontrou um compromisso entre tempo do primeiro máximo e estabilidade e, portanto, possui o maior *fitness*. Remarcamos que os tempos de subida e estabilização e o *overshoot* desta resposta foram muito inferiores a aqueles encontrados nas outras

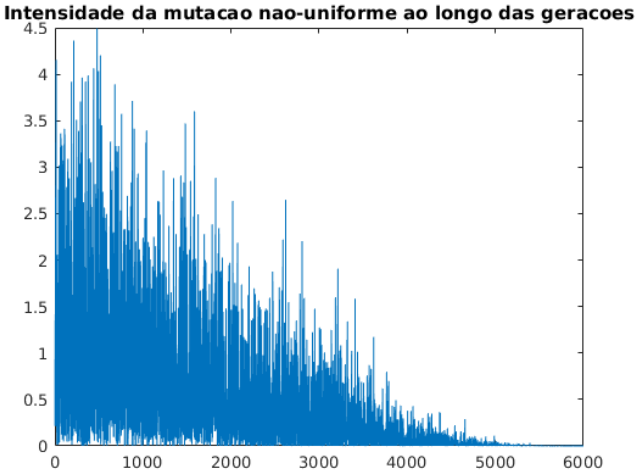
execuções, confirmando ainda mais a qualidade desta solução. Novamente, nos deparamos com o fato de que o ajuste de k_d é o mais sensível: grandes valores deste parâmetro não levam a bons resultados. Observa-se ainda que os outros dois parâmetros, k_p e k_i, também apresentam grandes variações em relação as outras execuções, sendo k_d muito superior e k_i inferior. As imagens a seguir contém os gráficos gerados para a melhor execução.



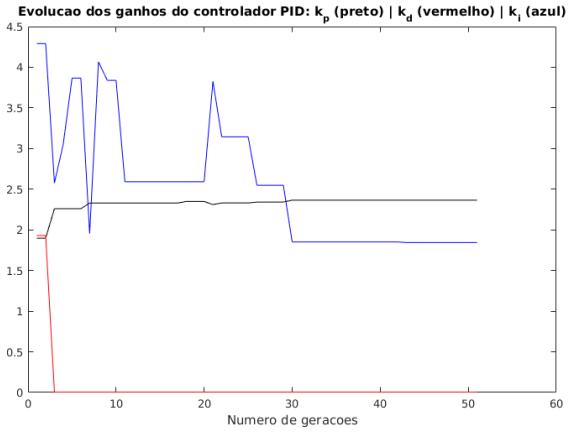
(a) Resposta ao degrau do sistema.



(b) Evolução dos valores da função *fitness*.



(c) Evolução da intensidade de mutações ao longo das gerações.



(d) Evolução dos parâmetros.

Figura 3: Resultado para a melhor execução do algoritmo evolutivo com *fitness* considerando a margem de fase.

A figura 3a representa a resposta ao degrau do sistema com os parâmetros determinados na execução 5, cujas características já foram discutidas no parágrafo anterior. Na imagem 3b, que apresenta a evolução do *fitness*, observa-se que o melhor valor já é encontrado nas primeiras gerações. Entretanto, a quantia média ainda é muito inferior a tal valor, o que implica que a população é muito diversificada até a geração 50, isto é, poucos indivíduos se "adaptaram" tão bem quanto o melhor. A figura 3d revela que o parâmetro que mais variou entre as gerações foi k_p, assim como no caso do item anterior.

Os resultados obtidos com a função *fitness* considerando o tempo de subida e também a margem de fase foram muito superiores em muitos aspectos:

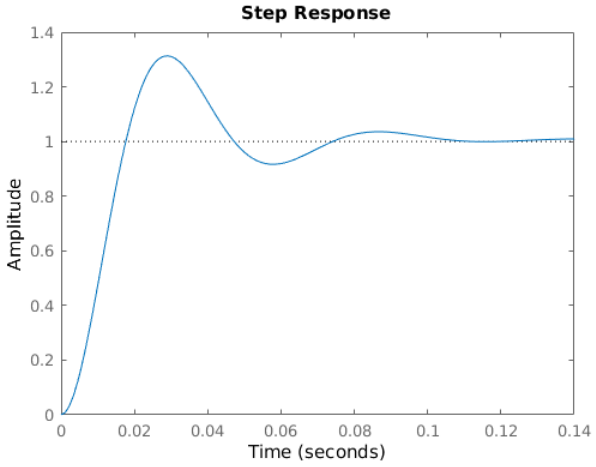
- os valores de *overshoot* no caso anterior eram, em média, de 97%, sendo que neste item, encontra-se 30%;

- a margem de fase no item anterior ($\approx 2^o$) era muito inferior ao caso atual, em que exige-se $\Delta\phi = 60^o$, comprometendo fortemente a estabilidade do sistema;
- o tempo de subida no caso anterior era muito inferior ao atual, porém sua frequência de oscilação em torno de 1 era inaceitável;
- no caso anterior, a resposta demorava muito a se estabilizar, o que pode representar um problema para algumas aplicações. No caso atual, encontra-se tempos de estabilização na ordem de centésimos de segundo;

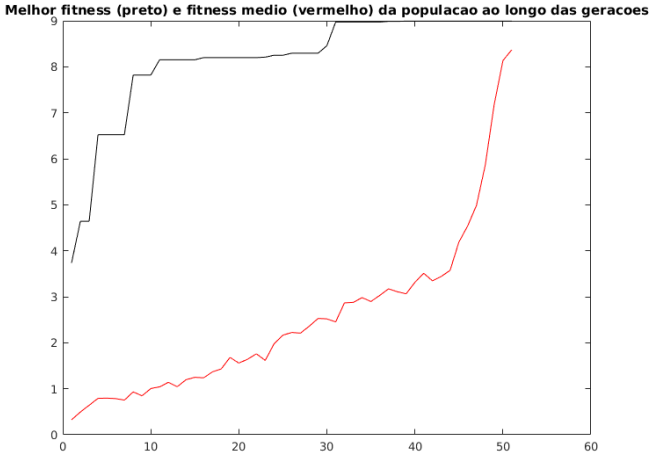
Enfim, efetuando as mesmas modificações realizadas anteriormente no tamanho da população e nas taxas de crossover e mutação, encontra-se os dados da tabela 5 e a imagems a seguir:

Tabela 5: Resultados da execução com os parâmetros modificados.

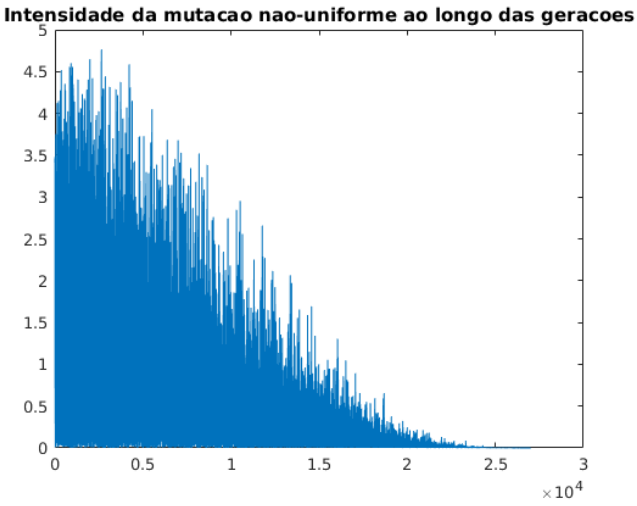
k_p	k_d	k_i	t_{subida} (s)	$t_{estabilizacao}$ (s)	*Overshoot* (%)	$\Delta\phi$ (^{o})	*Fitness*
2.57579	0.005528	3.77055	0.011941	0.09798	31.3347	60.015	8.99308



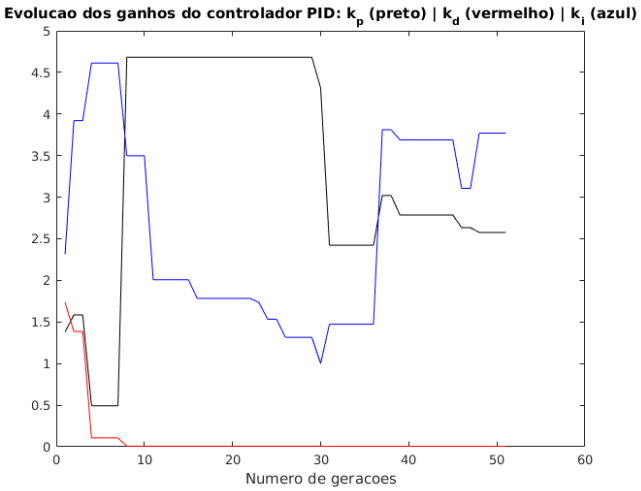
(a) Resposta ao degrau do sistema.



(b) Evolução dos valores da função *fitness*.



(c) Evolução da intensidade de mutações ao longo das gerações.



(d) Evolução dos parâmetros.

Figura 4: Resultado para a melhor execução do algoritmo evolutivo com *fitness* considerando a margem de fase e parâmetros modificados.

Neste caso, como os resultados de cada execução variam bastante, aumentar a população e as taxas de *crossover* e mutação contribuem para a obtenção de uma solução de melhor qualidade,

visto que a diversidade nos indivíduos é maior. Logo, a probabilidade que a solução ótima esteja presente em um dos indivíduos é consequentemente maior. Uma taxa de mutação elevada também garante que mais soluções sejam testadas, à medida que bons indivíduos podem dar origem a indivíduos melhores que eles mesmos através da mutação. Enfim, assim como no caso em que os parâmetros não tinham sido alterados, a figura 4b revela que a média dos valores de *fitness* é muito distante do maior, indicando que nas gerações mais jovens há muitos indivíduos ruins. A figura 4d mostra, por sua vez, que, para populações maiores, todos os parâmetros podem variar fortemente ao passar das gerações, ao contrário que é observado no caso em que tamanho da população é igual a 100. Este fato também é totalmente esperado, visto que, para populações maiores, a chance de um indivíduo encontrar um outro melhor que ele é maior.

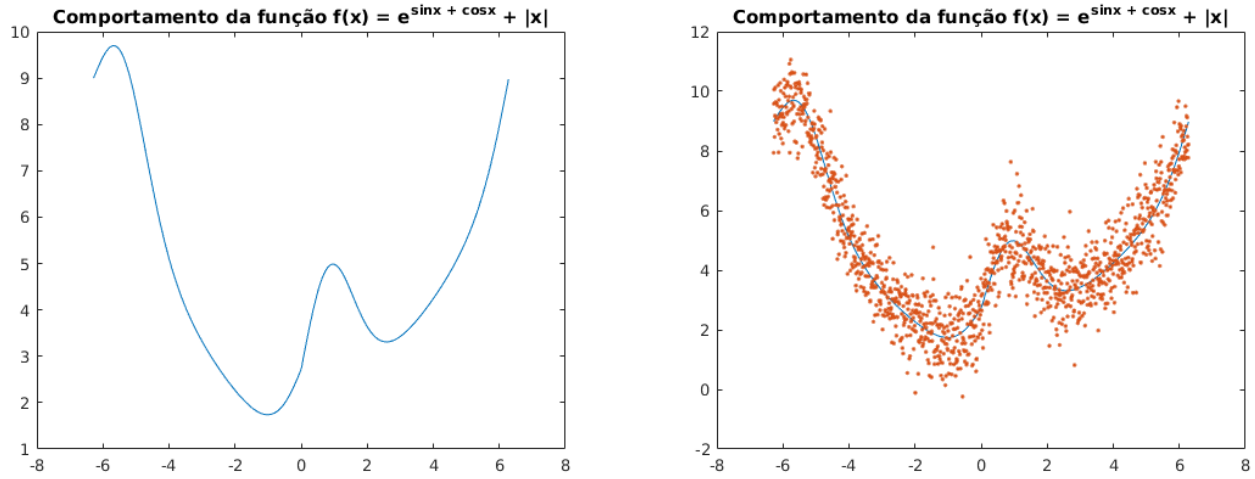
2 Aproximação de funções multidimensionais

2.1 Mapeamento y = f(x)

A função escolhida para o mapeamento $\mathbb{R}^1 \rightarrow \mathbb{R}^1$ está representada logo abaixo. Foi definido que o intervalo de x é $[-2\pi, 2\pi]$ e que este intervalo seria discretizado a cada 0.01, a fim de possuirmos uma quantidade grande de amostras, isto é, 1257.

$$y = \exp(\sin x + \cos x) + |x| \tag{1}$$

Sem a presença de ruídos, este mapeamento produz a imagem 5a. A imagem 8d, por sua vez, apresenta o resultado da soma da função com um ruído de distribuição normal de média nula e variância 0.64. É possível verificar que a adição do ruído prejudica fortemente a identificação do formato inicial do mapeamento.

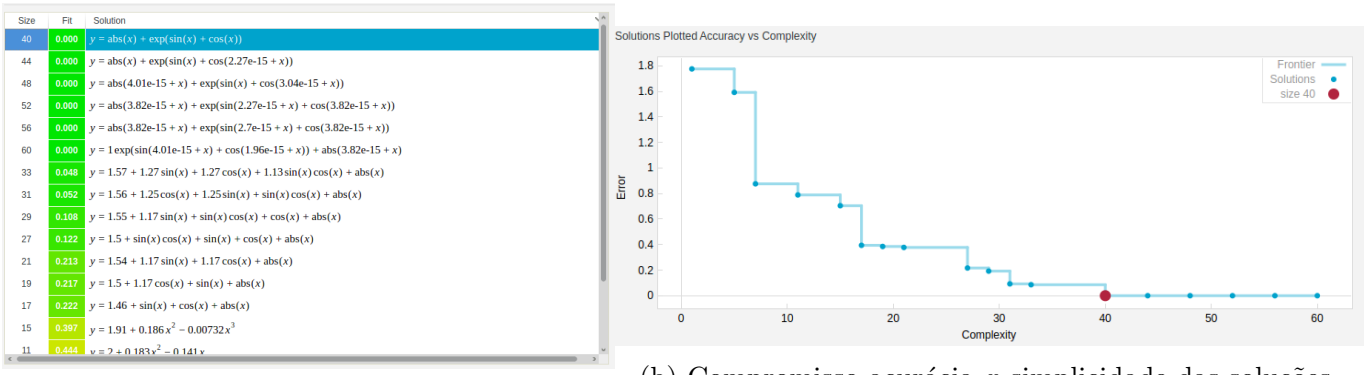


(a) Mapeamento sem ruídos.

(b) Mapeamento com ruídos.

Figura 5: Mapeamento da função 1.

A simulação realizada no *Eureqa*, primeiramente com os dados sem ruídos e habilitando-se as funções-base exp(), sin(), cos(), abs() e as operações básicas, forneceu os resultados contidos nas figuras 6a e 6b. A primeira lista as soluções encontradas que mais se aproximaram dos dados amostrais. Destaca-se que a primeira solução, cujo peso é 40, é justamente aquela que utilizamos para produzir os dados. A segunda imagem ilustra o compromisso entre acurácia (erro) e simplicidade (complexidade da solução). Observa-se que inúmeras soluções apresentam erro praticamente nulo, porém existe apenas uma cuja complexidade é mínima para este caso. Essa função trata-se da solução de tamanho 40 que, como já foi dito, é igual à equação 1. A figura 6c apresenta um resumo da melhor solução encontrada. Destaca-se que o erro podem ser considerados como nulos, visto que o maior erro encontrado foi da ordem de 10^{-14}.



(a) Soluções encontradas pelo software.

(b) Compromisso acurácia x simplicidade das soluções.

Solution Details (calculated on validation data)

Solution	y = abs(x) + exp(sin(x) + cos(x))
R^2 Goodness of Fit	1
Correlation Coefficient	1
Maximum Error	2.30926e-14
Mean Squared Error	5.44583e-29
Mean Absolute Error	5.78929e-15
Coefficients	0
Complexity	40

(c) Resumo da melhor solução encontrada.

Figura 6: Resultados para mapeamento da função 1 sem ruídos.

A simulação executada com os dados com ruído produziu o mapeamento representado pela equação 2. A figura 7 compara o mapeamento produzido nesta fase com aquele utilizado para gerar os dados. Observa-se que ambos os mapeamento estão muito próximos, porém há regiões em que as diferenças são notáveis. Os resultados gerados pelo *software* são, portanto, bastante satisfatórios.

$$y = 1.61175 + 1.830094 * \sin(0.792661 + x) + 0.55764 * \sin(1.99325 * x) + |x| \quad (2)$$

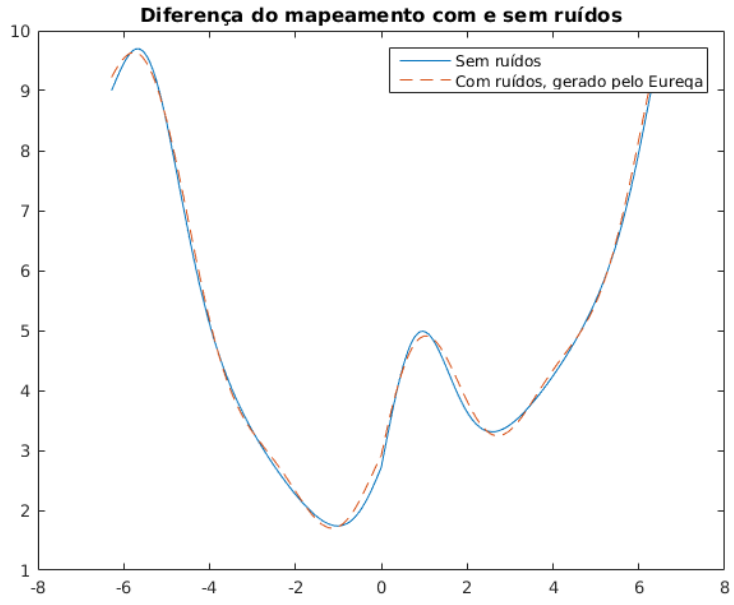
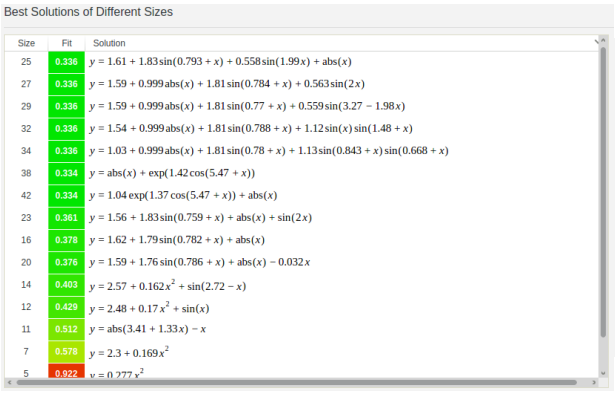
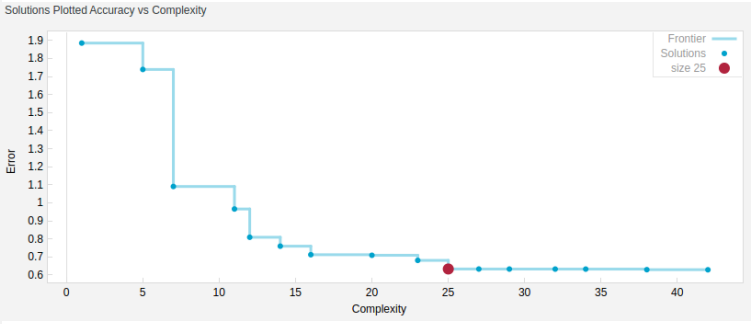


Figura 7: Comparação entre mapeamento com e sem ruídos.

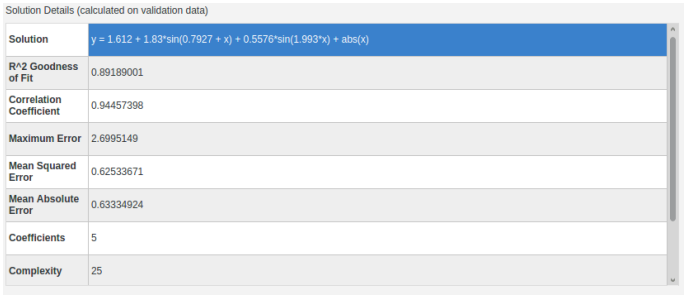
As figuras 8a, 8b, 8c e 8d foram geradas pelo *Eureqa* e possuem informações sobre a solução encontrada. A primeira é a lista de soluções encontradas pelo *software*, a segunda mostra o compromisso acurácia x complexidade (neste caso, a solução apresentada na equação 2 possui melhor compromisso, visto que seu erro é próximo de 0 e sua complexidade, mínima), a terceira contem algumas informações sobre a melhor solução (o erro quadrático médio neste caso é considerável, contrariarmente ao caso sem ruído, e vale 0.625) e a última revela quais dados foram usados para treinamento e validação, assim como o mapeamento encontrado.



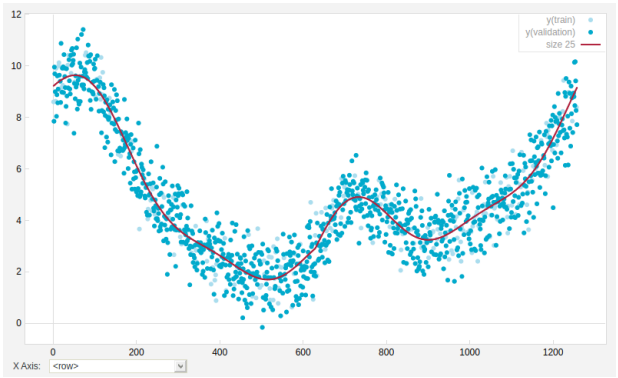
(a) Soluções encontradas pelo software.



(b) Compromisso acurácia x simplicidade das soluções.



(c) Resumo da melhor solução encontrada.



(d) Resumo do mapeamento encontrado e amostras usadas para treinamento e validação.

Figura 8: Resultados para mapeamento da função 1 com ruídos.

2.2 Mapeamento $\mathbf{y = f(x_1, x_2, x_3)}$

A função escolhida para este caso está representada pela equação 3, sendo que $x_1 \in [-3, 1]$, $x_2 \in [1, 5]$ e $x_3 \in [-4, 4]$. As duas primeiras variáveis foram amostradas a cada 0.01 e a segunda, 0.02. Temos, portanto, 401 amostras no total.

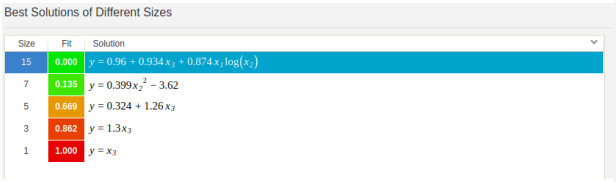
$$y = x_1 * \log_\pi\left(\frac{x_2}{3}\right) + \sqrt{2} * x_3 = x_1 * \frac{\log_{10}\left(\frac{x_2}{3}\right)}{\log_{10}(\pi)} + \sqrt{2} * x_3 \tag{3}$$

O mapeamento produzido pela simulação utilizando os dados sem ruídos foi

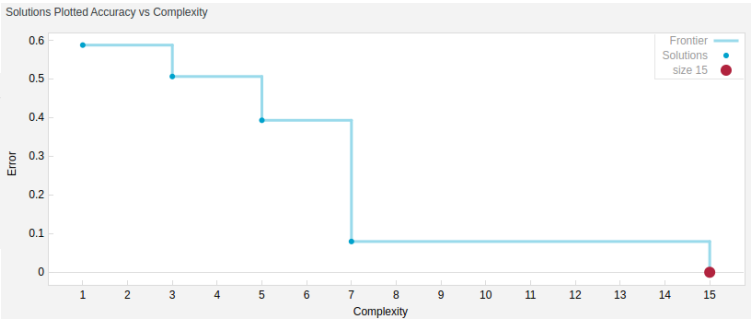
$$y = 0.959713 + 0.93436 * x_3 + 0.873569 * x_1 * \log_{10}(x_2) \tag{4}$$

O mapeamento produzido não corresponde ao utilizado para a geração dos dados, porém os erros associados ao primeiro são praticamente nulos, isto é, o maior erro é da ordem de 10^{-15}, conforme figura 9c. Isto indica que, para os os intervalos considerados para as variáveis, alguns termos podem ser substituídos por outros sem perdas de precisão. Nota-se também, pela figura 9a, que menos soluções foram encontradas em relação ao item anterior. A imagem 9b mostra o compromisso entre

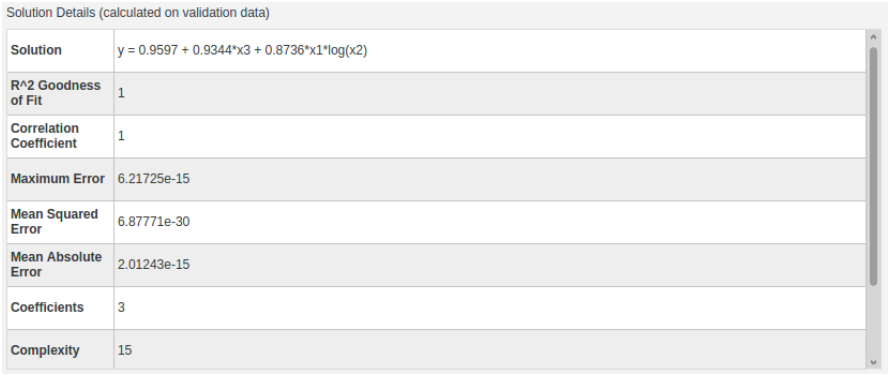
erro e complexidade da melhor solução e é possível concluir, a partir desta imagem, que realmente a solução decomplexidade é 15 supera as demais.



(a) Soluções encontradas pelo software.



(b) Compromisso acurácia x simplicidade das soluções.



(c) Resumo da melhor solução encontrada.

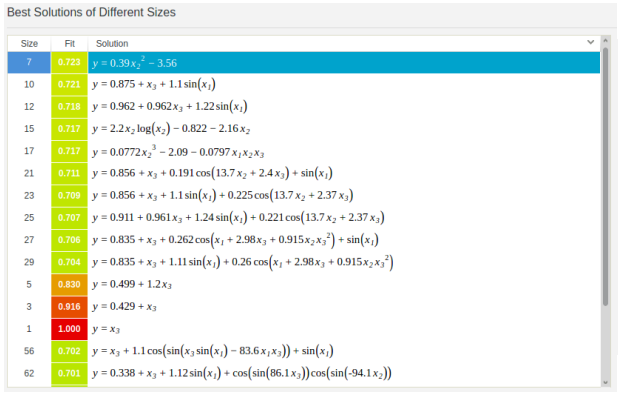
Figura 9: Resultados para mapeamento da função 3 sem ruídos.

Enfim, a execução para os dados com ruídos gera o mapeamento

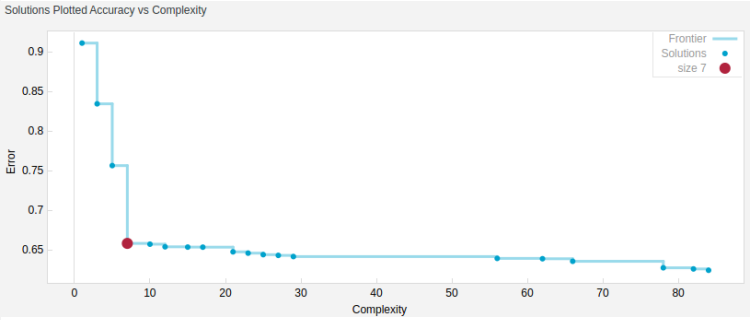
$$y = 0.39025 * x_2^2 - 3.555333 \tag{5}$$

Observa-se que y não depende de x_1 e x_3 neste caso. As figuras a seguir foram geradas pelo *software*. Destacam-se os seguintes fatos:

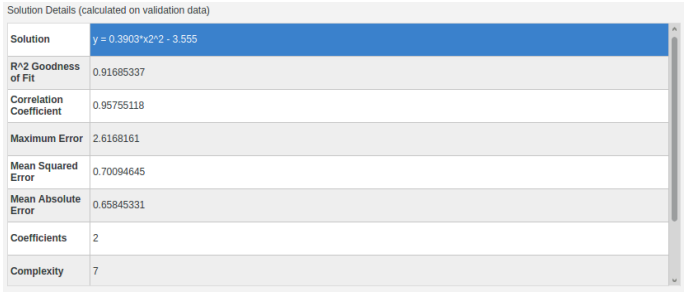
- A figura 10b mostra o compromisso entre erro e complexidade para as melhores soluções. Nota-se que, apesar de a solução não apresentar o menor erro, ela possui uma maior simplicidade comparada às demais. Sendo assim, ela é escolhida como a melhor solução atendendo aos critérios de acurácia e complexidade simultaneamente.
- Conforme figura 10c, o erro quadrático médio é considerável e vale, aproximadamente, 0.70.
- A figura 10d representa o mapeamento e as amostras utilizadas. Observa-se que este mapeamento aproxima-se relativamente bem aos dados.



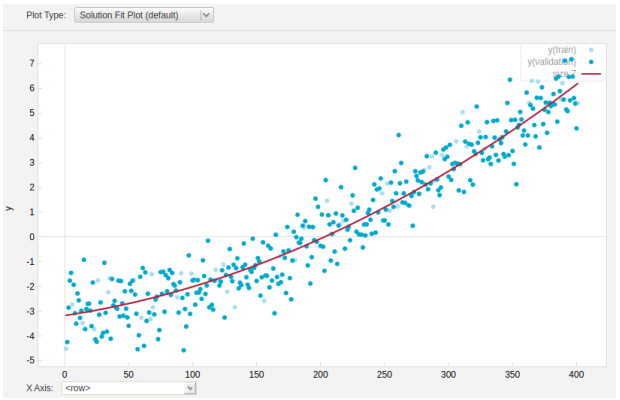
(a) Soluções encontradas pelo software.



(b) Compromisso acurácia x simplicidade das soluções.



(c) Resumo da melhor solução encontrada.



(d) Resumo do mapeamento encontrado e amostras usadas para treinamento e validação.

Figura 10: Resultados para mapeamento da função 3 com ruídos.

3 Controle Nebuloso e Robótica Evolutiva

3.0 Definição dos Consequentes

O primeiro passo no controle do robô é a definição dos consequentes das regras que serão levadas em conta. Considerando que o robô dispõe de 3 sensores (d_1, d_2 e d_3) e que as funções de pertinência associadas às medidas realizadas por eles estão representadas na seção 3 do enunciado, sugestões de consequentes estão representadas na tabela 6, logo abaixo. As regras são da forma **SE** (d_1 **É x**) **E** (d_2 **É y**) **E** (d_3 **É z**) **ENTÃO** ($\Delta\theta$ **É w**).

Tabela 6: Regras que controlarão o robô.

d_1	d_2	d_3	$\Delta\theta$
P	MP	P	MN
P	MP	M	MN
P	MP	G	MN
P	PP	P	Z
P	PP	M	MN
P	PP	G	MN
P	PG	P	Z
P	PG	M	MN
P	PG	G	MN
P	MG	P	Z
P	MG	M	PN
P	MG	G	MN

d_1	d_2	d_3	$\Delta\theta$
M	MP	P	PP
M	MP	M	MP
M	MP	G	MN
M	PP	P	PP
M	PP	M	Z
M	PP	G	PN
M	PG	P	PP
M	PG	M	Z
M	PG	G	PN
M	MG	P	PP
M	MG	M	Z
M	MG	G	PN

d_1	d_2	d_3	$\Delta\theta$
G	MP	P	MP
G	MP	M	MP
G	MP	G	MP
G	PP	P	MP
G	PP	M	MP
G	PP	G	MP
G	PG	P	MP
G	PG	M	PP
G	PG	G	Z
G	MG	P	MP
G	MG	M	PP
G	MG	G	Z

3.1 Controlando o Robô

Uma vez definidas as regras que deverão ser seguidas, o próximo passo é então simular o comportamento do robô através do *software* MATLAB. Para isso, 5 funções foram escritas. A primeira, chamada `trap_pertinencia`, recebe 6 parâmetros, sendo eles a coordenada x, as informações a, b, c e d referentes os trapézios que compõem as funções de pertinência e o valor a ser considerado caso x esteja fora do intervalo $[a, d]$, e retorna o respectivo valor da função de pertinência. O programa 1 abaixo possui a implementação da respectiva função.

Programa 1: `trap_pertinencia.m` - Avalia pertinência de um regra em função de x.

```
%%  FUNCAO trap_pertinencia
%   Funcao responsavel por calcular a pertinencia em uma dada coordenada x.
%   Parametros: x eh a coordenada para qual calcula-se a pertinencia.
%               a, b, c e d sao os parametros referentes ao trapezio.
%               out_of_range representa o resultado que deve ser atribuido
%               caso x esteja fora do intervalo [a, d].
%   Autor: Gustavo CIOTTO PINTON
function pert = trap_pertinencia( x, a, b, c, d , out_of_range)

if (x < a) || (x > d)
    pert = out_of_range;

elseif (x < b)
    if (a ~= b)
        pert = (x - a)/(b -a);
    else
        pert = 1;
    end

elseif (x < c)
    pert = 1;

else
    if (c ~= d)
        pert = (d - x)/(d - c);
    else
        pert = 1;
    end
end
end
```

A segunda função, chamada de `get_D1_D3_Rule` e representada pelo programa 2, calcula a pertinência de cada um dos possíveis estados referentes aos sensores D_1 e D_3. Essa função recebe como parâmetro a distância medida pelo sensor e retorna um vetor com três valores contendo a pertinência para os estados **P**, **M** e **G**.

Programa 2: Avalia pertinência da regras de D_1 e D_3 em função das distâncias medidas.

```
%%  FUNCAO get_D1_D3_Rule
%   Calcula pertinencia para todos os possiveis estados de D1 e D3.
%   Parametros: sensor_capture eh a distancia medida pelo sensor.
%   Retorno   : Vetor com os valores das pertinencias atribuidas a cada
%   regra.
%   Autor: Gustavo CIOTTO PINTON
function S = get_D1_D3_Rule( sensor_capture )

% Calcula pertinencia para P
relevance_P = trap_pertinencia (sensor_capture, 0, 0, 1, 2, (sensor_capture < 0));

% Calcula pertinencia para M
relevance_M = trap_pertinencia (sensor_capture, 1, 2, 3, 4, 0);
```

```
14
15 % Calcula pertinencia para G
16 relevance_G = trap_pertinencia (sensor_capture, 3, 4, 5, 5, (sensor_capture > 5));
17
18 S = [relevance_P relevance_M relevance_G];
19
20 end
```

A função `get_D2_Rule` realiza o mesmo que o descrito anteriormente, mas agora com o sensor D_2. A implementação desta função pode ser encontrada logo abaixo.

Programa 3: Avalia pertinência da regras de D_2 em função das distâncias medidas.

```
1  %%  FUNCAO get_D2_Rule
2  %   Calcula pertinencia para todos os possiveis estados de D2.
3  %   Parametros: sensor_capture eh a distancia medida pelo sensor.
4  %   Retorno   : Vetor com os valores das pertinencias atribuidas a cada
5  %   regra.
6  %   Autor: Gustavo CIOTTO PINTON
7  function S = get_D2_Rule( sensor_capture )
8
9  % Testar para MP
10 relevance_MP = trap_pertinencia (sensor_capture, 0, 0, 1, 2, (sensor_capture < 0));
11
12 % Testar para PP
13 relevance_PP = trap_pertinencia (sensor_capture, 1, 2, 3, 4, 0);
14
15 % Testar para PG
16 relevance_PG = trap_pertinencia (sensor_capture, 3, 4, 5, 6, 0);
17
18 % Testar para MG
19 relevance_MG = trap_pertinencia (sensor_capture, 5, 6, 7, 7, (sensor_capture > 7));
20
21 S = [relevance_MP relevance_PP relevance_PG relevance_MG];
22
23 end
```

A quarta função, `get_Angle_Rule`, implementa as regras descritas na tabela 6. Por ser extensa, a sua implementação foi colocada na seção **Anexos** ao fim deste documento no programa **??**. Em poucas palavras, esta função testa se cada uma das regras está ativa e, caso uma determinada regra esteja, atribui a ela o menor valor dentre as pertinências dos estados que a compõem. Por exemplo, se D_1 é **P** com 50%, D_2 é **MG** com 100% e D_3 é **M** com 33%, então $\Delta\theta$ é **PN** com 33%.

A quinta função, `get_Angle`, é responsável pelo processo de *defuzzyficação* e está representada pelo programa 4. Ela utiliza o método de centro de massa para determinar $\Delta\theta$ que melhor convém dadas as regras ativas.

Programa 4: Avalia pertinência da regras de D_2 em função das distâncias medidas.

```
1  %%  FUNCAO get_Angle
2  %   Funcao responsavel por calcular o desvio no angulo do robo com base na
3  %   regras que foram ativadas pelos sensores.
4  %   Parametros: active_rules possui as regras que foram ativadas e suas
5  %   respectivas pertinencias.
6  %   Retorno:    o desvio que devera ser feito na direcao do robo
7  %   Autor: Gustavo CIOTTO PINTON
8  function angle = get_Angle (active_rules)
9
10 % Discretizacao do intervalo dos possiveis angulos.
11 available_angles = -20:0.5:20;
12
```

```
% c_values eh uma matriz, em que cada componente representa o resultado
% representa o minimo entre a funcao de pertinencia calculada em um ponto e
% o valor da pertinencia de uma regra ativa.  Como ha 5 possiveis funcoes
% de pertinencia (MN, PN, Z, PP e MP), a matriz possui 5 linhas.
c_values = zeros (5, length(available_angles));

% parameters eh uma matriz que contem os parametros que descrevem os
% trapezios relativos a cada uma das possiveis funcoes de pertinencia de
% do desvio angular.
parameters = [ -20 -20 -15 -10 0 ;
               -15 -10  -5   0 0 ;
               -5    0   0   5 0 ;
                0    5  10  15 0 ;
                10  15  20  20 0 ];

% Itera todos os possiveis angulos.
for k = 1: length(available_angles)

    % Para cada angulo e cada funcao de pertinencia, calcula-se o
    % respectivo valor de pertinencia.
    for n = 1:5

        % O valor de pertinencia eh o minimo entre o resultado da n-esima funcao
        % de pertinencia e a pertinencia da regra que corresponde a aquela
        % mesma funcao.
        c_values(n, k) = min (active_rules(n), trap_pertinencia(available_angles(k),
            ...
            parameters(n ,1),  parameters(n ,2), parameters(n ,3), parameters(n ,4),
                ...
            parameters(n ,5)));
    end

end

% Recupera os maximos dos valores de pertinencia entre todas as funcoes de
% pertinencia segundo o metodo do MAXIMO dos MINIMOS.
max_values = max(c_values);

% Calcula o centro de massa.
angle = sum(available_angles.*max_values)/sum(max_values);

% Converte o resultado em radianos.
angle = angle*pi/180;

end
```

Enfim, a sexta função, `robot_control`, realiza o controle do robô conforme regras especificadas acima. Sua implementação está representada pelo programa 5.

Programa 5: Automatiza controle do robô.

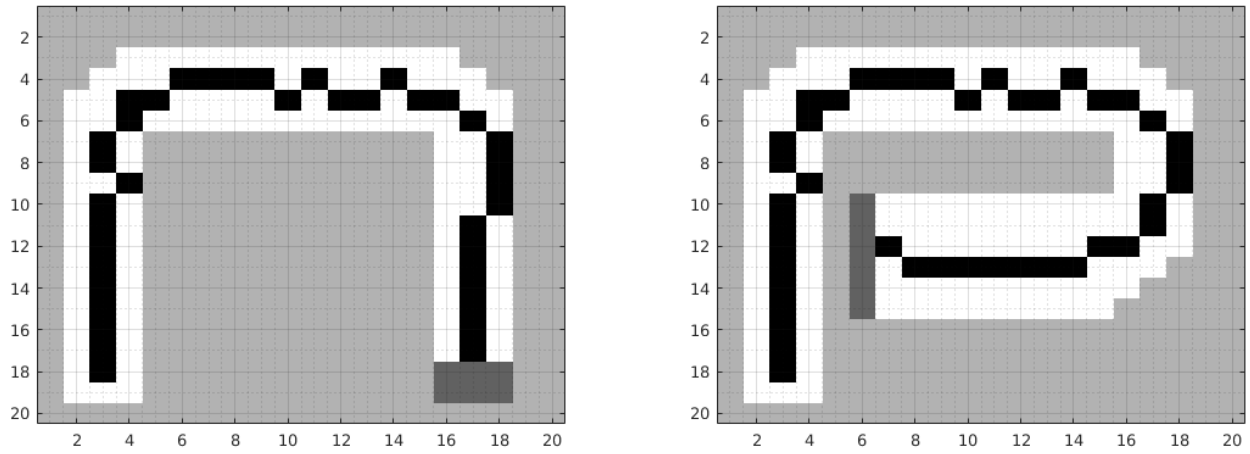
```
%%  FUNCAO robot_control
%   Funcao responsavel por controlar o robo segundo as regras formalizadas
%   na secao '3.0 Definicao dos Consequentes'.
%   Parametros: matrix eh o labirinto onde o robo sera colocado.
%               xStart e yStart sao as coordenadas iniciais do robo.
%               rDirection eh a direcao inicial do robo.
%               speed eh a velocidade do robo.
%   Autor: Gustavo CIOTTO PINTON
function robot_control(matrix, xStart, yStart, rDirection, speed)

labyrinth_matrix = matrix;
robot_direction = rDirection;
```

```
13 x_robot = xStart;
14 y_robot = yStart;
15
16 m_length = length (labyrinth_matrix);
17
18 % Movimenta o robo ate achar um ponto de destino, isto e, 'F'
19 while (labyrinth_matrix (round(m_length + 1 - y_robot), round(x_robot)) ~= 'F')
20
21     labyrinth_matrix (round(m_length + 1 - y_robot), round(x_robot)) = 'r';
22
23     % Procura o o obstaculo mais proximo na direcao d1.
24     % Itera ate encontrar tal obstaculo.
25     x_d1 = x_robot + cos(robot_direction + pi/4 );
26     y_d1 = y_robot + sin(robot_direction + pi/4 );
27     while (round(x_d1) >= 1) && (round(x_d1) <= length(labyrinth_matrix (1,:)) ) ...
28             && (round(m_length + 1 - y_d1) >= 1) ...
29             && (round(m_length + 1 - y_d1) <= m_length) ...
30             && (labyrinth_matrix (round(m_length + 1 - y_d1), round(x_d1)) ~= '#')
31
32         x_d1 = x_d1 + cos(robot_direction + pi/4);
33         y_d1 = y_d1 + sin(robot_direction + pi/4);
34
35     end
36
37     % Calcula distancia ao obstaculo mais proximo encontrado.
38     distance_d1 = sqrt((x_d1 - x_robot)^2 + (y_d1 - y_robot)^2);
39
40     % Procura o o obstaculo mais proximo na direcao d2.
41     % Itera ate encontrar tal obstaculo.
42     x_d2 = x_robot + cos(robot_direction);
43     y_d2 = y_robot + sin(robot_direction);
44     while (round(x_d2) >= 1) && (round(x_d2) <= length(labyrinth_matrix (1,:)) ) ...
45             && (round(m_length + 1 - y_d2) >= 1) ...
46             && (round(m_length + 1 - y_d2) <= m_length) ...
47             && (labyrinth_matrix (round(m_length + 1 - y_d2), round(x_d2)) ~= '#')
48
49         x_d2 = x_d2 + cos(robot_direction);
50         y_d2 = y_d2 + sin(robot_direction);
51
52     end
53
54     % Calcula distancia ao obstaculo mais proximo encontrado.
55     distance_d2 = sqrt((x_d2 - x_robot)^2 + (y_d2 - y_robot)^2);
56
57     % Procura o o obstaculo mais proximo na direcao d3.
58     % Itera ate encontrar tal obstaculo.
59     x_d3 = x_robot + cos(robot_direction - pi/4);
60     y_d3 = y_robot + sin(robot_direction - pi/4);
61     while (round(x_d3) >= 1) && (round(x_d3) <= length(labyrinth_matrix (1,:)) ) ...
62             && (round(m_length + 1 - y_d3) >= 1) ...
63             && (round(m_length + 1 - y_d3) <= m_length) ...
64             && (labyrinth_matrix (round(m_length + 1 - y_d3), round(x_d3)) ~= '#')
65
66         x_d3 = x_d3 + cos(robot_direction - pi/4);
67         y_d3 = y_d3 + sin(robot_direction - pi/4);
68
69     end
70
71     % Calcula distancia ao obstaculo mais proximo encontrado.
72     distance_d3 = sqrt((x_d3 - x_robot)^2 + (y_d3 - y_robot)^2);
73
74     % Verifica valores das funcoes de pertinencia para cada sensor.
75     d1_rules = get_D1_D3_Rule(distance_d1);
```

```
76      d2_rules = get_D2_Rule(distance_d2);
77      d3_rules = get_D1_D3_Rule(distance_d3);
78
79      % Verifica quais regras foram ativadas.
80      active_rules = get_Angle_Rule(d1_rules, d2_rules, d3_rules);
81
82      % Processo de defuzzyficacao. A partir das regras ativas, calcula-se o
83      % desvio a ser feito na direcao do robo.
84      d_angle = get_Angle (active_rules);
85
86      robot_direction = robot_direction + d_angle;
87
88      % Calcula nova posicao.
89      x_robot = x_robot + speed*cos(robot_direction);
90      y_robot = y_robot + speed*sin(robot_direction);
91
92  end
93
94  % Imprime matriz do labirinto.
95  figure
96  imagesc(labyrinth_matrix);
97  colormap(flipud(gray));
98  grid on;
99  grid minor;
100 end
```

3.2 Resultados

A fim de testar o controlador implementado na seção anterior, dois mapas foram criados. Os resultados das execuções estão representados na figuras 11a e 11b, em que a trajetória está representada em preto, a parede em cinza claro, os pontos de destino em cinza escuro e as posições livres, em branco. Para ambos, a direção inicial do robô é $\frac{\pi}{2}$, as coordenadas iniciais são $(3, 3)$ e a velocidade é $v = 1$ *pixel/iteração*. Verifica-se que o controle é bem-sucedido, sendo que o robô é capaz de realizar as curvas presentes nos mapas. É importante comentar que os cantos *suavizados* possuem uma grande importância para o bom resultado, uma vez que o controle nas áreas de "ponta" não é tão eficiente visto que, quando o controlador verifica a proximidade a uma parede, ele pode não saber para que lado virar.



(a) Controle do robô no mapa 1.

(b) Controle do robô no mapa 2.

Figura 11: Resultados para o controle nebuloso do robô para dois mapas distintos.

O *script* utilizado nos testes pode ser encontrado no programa 6.

Programa 6: *Script* de teste..

```
clear all;
close all;

% Constroi mapa 1.
labyrinth_matrix = zeros(20,20);
labyrinth_matrix (1:2, :) = '#';
labyrinth_matrix (20, :) = '#';
labyrinth_matrix (:, 1) = '#';
labyrinth_matrix (:, 19:20) = '#';
labyrinth_matrix (7:20, 5:15) = '#';
labyrinth_matrix (2:4, 2:4) = fliplr(triu(ones(3)))*'#';
labyrinth_matrix (2:5, 16:19) = triu(ones(4))*'#';
labyrinth_matrix (18:19, 16:18) = 'F';

robot_control(labyrinth_matrix, 3,3, pi/2, 1);

% Constroi mapa 2.
labyrinth_matrix = zeros(20,20);
labyrinth_matrix (1:2, :) = '#';
labyrinth_matrix (20, :) = '#';
labyrinth_matrix (:, 1) = '#';
labyrinth_matrix (:, 19:20) = '#';
labyrinth_matrix (7:20, 5:15) = '#';
labyrinth_matrix (16:20, 5:20) = '#';
labyrinth_matrix (2:4, 2:4) = fliplr(triu(ones(3)))*'#';
labyrinth_matrix (2:5, 16:19) = triu(ones(4))*'#';
labyrinth_matrix (12:15, 16:19) = flipud(triu(ones(4)))*'#';
labyrinth_matrix (10:15, 6:15) = 0;
labyrinth_matrix (10:15, 6) = 'F';

robot_control(labyrinth_matrix, 3,3, pi/2, 1);
```

3.3 Controle via Rede Neural MLP