ExerciciosLista1

October 18, 2018

1 Exercícios 4.2, 4.3 e 4.4 do livro texto

2018/3 Aluno: Pedro Bandeira de Mello Martins Disciplina: CPE773 - Otimização Convexa Professor: Wallace A. Martins PEE/COPPE - UFRJ

Minimizar $f(x)$ no intervalo dado com incerteza menor que 10^{-5} com os métodos:

1. Fibonacci Search
2. Golden-Section Search
3. Quadratic Interpolaton Method
4. Cubic Interpolation Method
5. Davies, Swann and Campey algorithm
6. Backtracking Line Search
7. Brute Force (implementação do scipy)

O algoritmo de força bruta foi utilizado para compararmos a quantidade necessária de avaliações para se chegar ao mesmo resultado.

Os pacotes utilizados nesses exercícios são:

```
In [1]: import sys
        if '..' not in sys.path:
            sys.path.append('..')
        import matplotlib.pyplot as plt
        from matplotlib import cm
        from mpl_toolkits.mplot3d import Axes3D
        import numpy as np
        import pandas as pd
        import time
        from copy import copy

        from scipy.optimize import brute
        from functions import order5_polynomial, logarithmic, sinoid, order4_polynomial
        from functions import functionObj
        from models.optimizers import DichotomousSearch, \
                                      FibonacciSearch, GoldenSectionSearch, \
                                      QuadraticInterpolationSearch, \
                                      CubicInterpolation, DaviesSwannCampey, \
                                      BacktrackingLineSearch, InexactLineSearch
```

Para os exercícios 4.2, 4.3 e 4.4, a função do arquivo `run_exercises.py` é rodada, onde se entrega a função a ser minimizada pelos métodos acima e ela retorna um *dataframe* com todas as informações obtidas durante as minimizações.

```
In [2]: from run_exercises import run_exercise
```

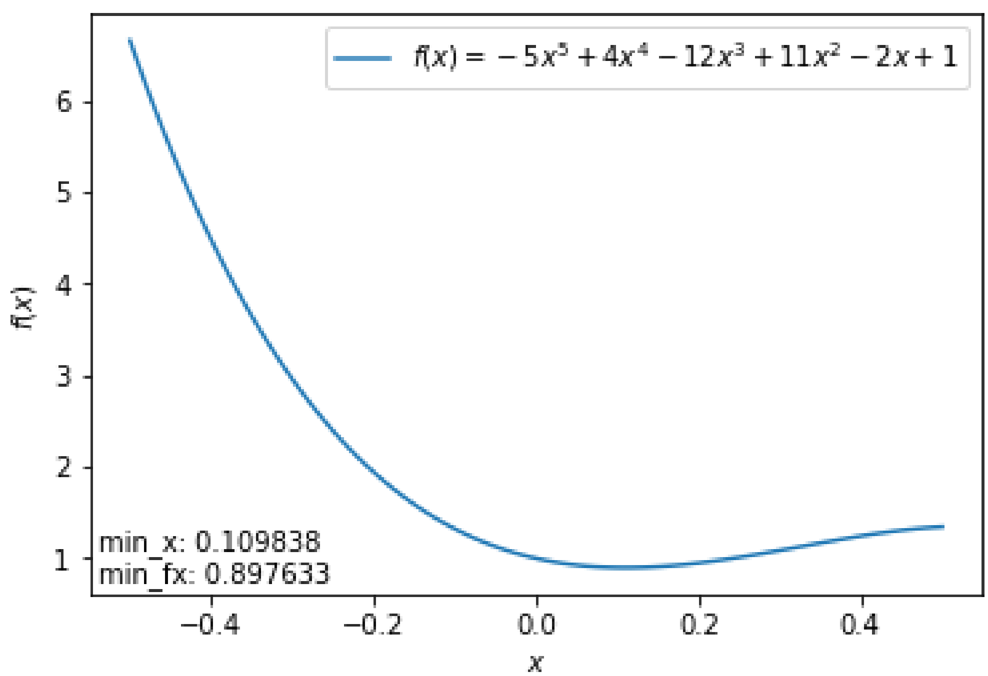
Os gráficos são gerados pela função:

```
In [3]: def show_chart(df):
            fig, axes = plt.subplots(4, 2, figsize = (13, 10))
            for algorithm, ax in zip(df.index, axes.flatten()):
                ax.plot(range(1, df['fevals'][algorithm] + 1), df['all_evals'][algorithm])
                ax.set_title(algorithm)
                ax.ticklabel_format(axis = 'y', style = 'plain')
                ax.set_xlabel('Function evaluations')
                ax.set_ylabel('$f(x)$')
            plt.tight_layout()
            plt.show()
```

1.1 Exercício 4.2

$f(x) = -5x^5 + 4x^4 - 12x^3 + 11x^2 - 2x + 1$

```
In [4]: results_42 = run_exercise(order5_polynomial, f_string = '$f(x) = -5x^5+4x^4-12x^3+11x^2
```



1.1.1 Resultados

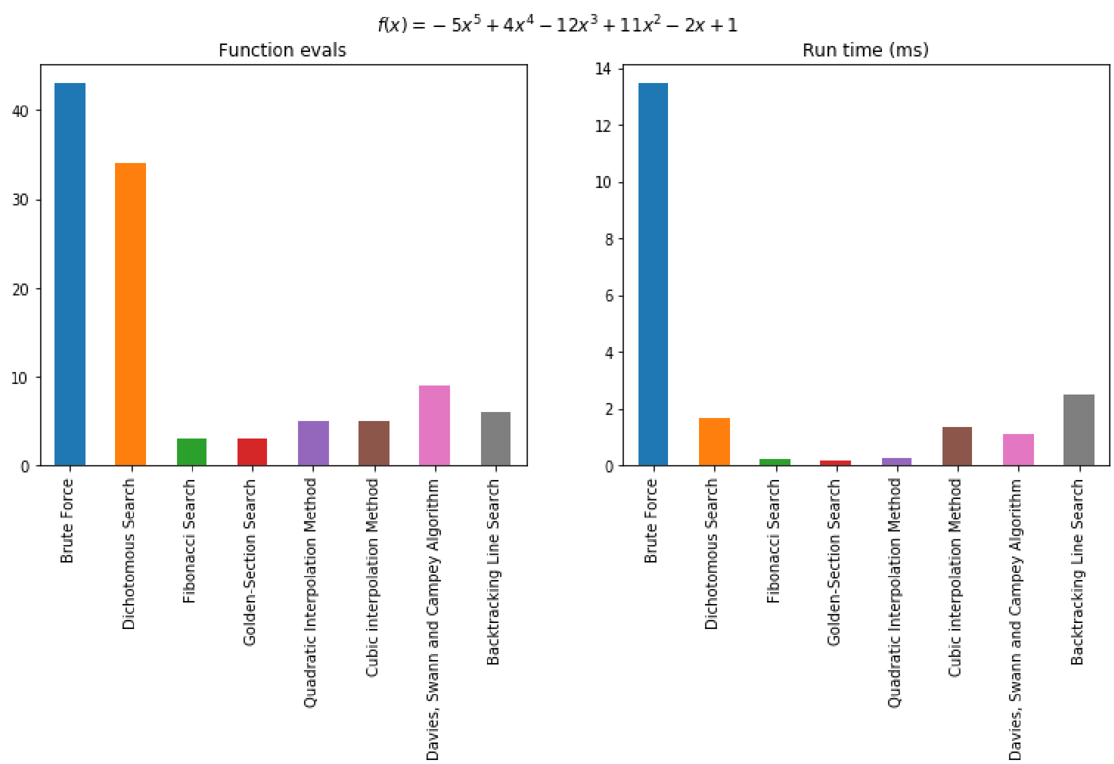
```
In [5]: results_42[['best_f', 'best_x', 'fevals', 'run_time (s)']]
```

Out[5]:

	best_f	best_x	fevals	run_time (s)
Brute Force	0.897633	0.109838	43	0.013442
Dichotomous Search	0.897633	0.109862	34	0.001680
Fibonacci Search	0.897633	0.109860	3	0.000207
Golden-Section Search	0.897633	0.109860	3	0.000160
Quadratic Interpolation Method	0.897633	0.109860	5	0.000275
Cubic interpolation Method	0.897633	0.109860	5	0.001358
Davies, Swann and Campey Algorithm	0.897633	0.109861	9	0.001077
Backtracking Line Search	0.897633	0.109860	6	0.002515

1.1.2 Eficiência computacional em avaliações de funções e tempo de execução.

```
In [6]: fig, axes = plt.subplots(1,2, figsize=(13,5))
        fig.suptitle('$f(x) = -5x^5+4x^4-12x^3+11x^2-2x+1$')
        results_42['fevals'].plot.bar(title = 'Function evals', ax= axes[0])
        (results_42['run_time (s)']*1e3).plot.bar(title = 'Run time (ms)', ax=axes[1])
        plt.show()
```

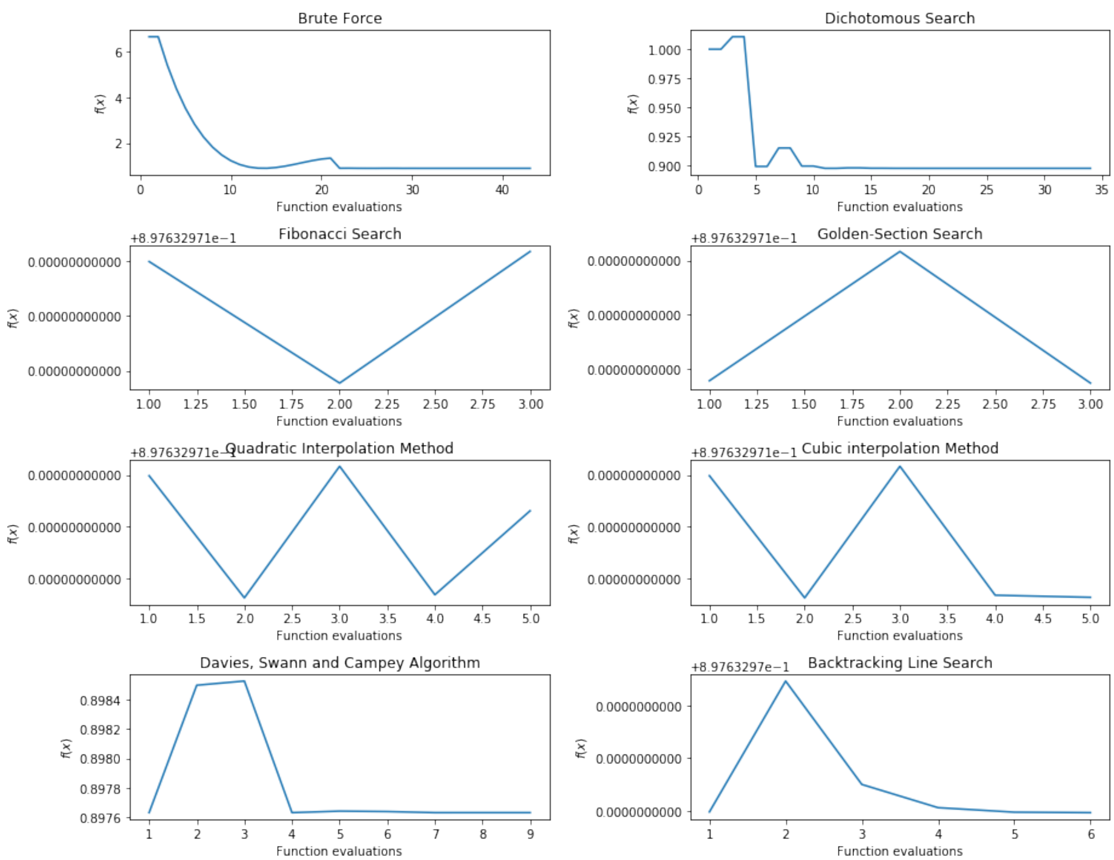


A tabela de resultados e os gráficos de eficiência computacional demonstram que a força bruta é a minimização de maior custo computacional tanto em número de avaliações de funções quanto em tempo de execução.

Apesar da Dichotomous Search ser o segundo algoritmo a utilizar mais avalições de funções, possui um tempo de execução menor do que a busca por retrocesso (*Backtracking Line Search*). Isso provavelmente acontece porque o *Backtracking Line Search* precisa estimar o gradiente da função objetivo. O gradiente é calculado com a biblioteca autograd.
Neste exercício a busca de seção dourada (*Golden-Section Search*) foi o algoritmo de menor tempo de execução e avaliação de funções.

1.1.3 Gráficos de f(x) por avaliações.

```
In [7]: show_chart(results_42)
```

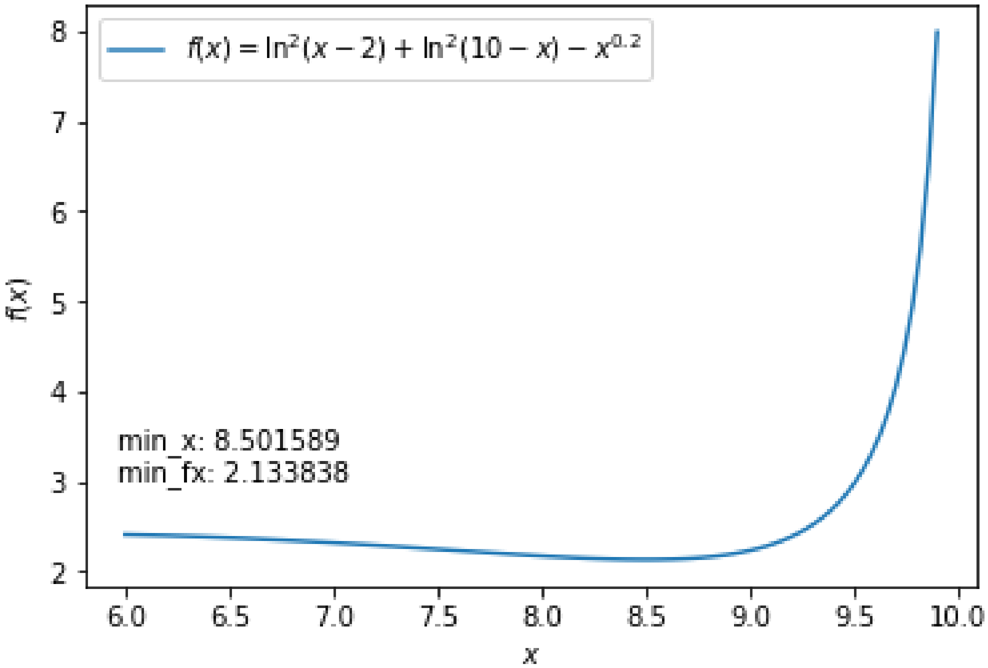


Pelos gráficos de $f(x)$ por avaliações de funções, podemos ver que os algoritmos *Fibonacci Search*, *Golden-Section Search*, *Quadratic Interpolation Method*, *Cubic Interpolation Method* e *Backtracking Line Search* encontraram o resultado logo na primeira avaliação de função, porém continuaram a rodar até entregarem o resultado. Isso acontece visto que a condição de parada deve ser encontrada, o que não necessariamente é na primeira interação que encontramos o mínimo.

1.2 Exercício 4.3

$f(x) = \ln^2(x-2) + \ln^2(10-x) - x^{0.2}$

```
In [8]: results_43 = run_exercise(logarithmic, f_string = '$f(x) = \ln ^2 (x-2) + \ln ^2(10-x)
                              seed =  9,
                              textpos = (12,40),
                              interval = [6, 9.9])
```



1.2.1 Resultados

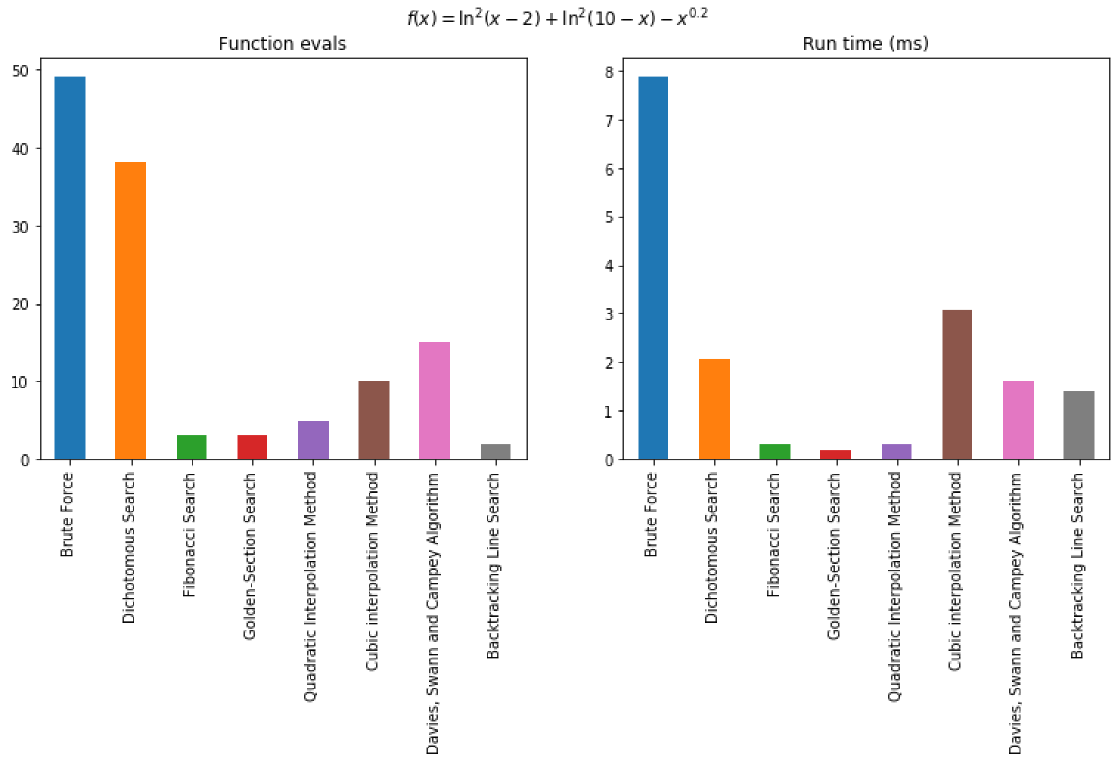
```
In [9]: results_43[['best_f', 'best_x', 'fevals', 'run_time (s)']]
```

Out[9]:

	best_f	best_x	fevals	run_time (s)
Brute Force	2.133838	8.501589	49	0.007866
Dichotomous Search	2.133838	8.501585	38	0.002073
Fibonacci Search	2.133838	8.501586	3	0.000308
Golden-Section Search	2.133838	8.501586	3	0.000176
Quadratic Interpolation Method	2.133838	8.501587	5	0.000296
Cubic interpolation Method	2.133838	8.501587	10	0.003075
Davies, Swann and Campey Algorithm	2.133838	8.501585	15	0.001615
Backtracking Line Search	2.133838	8.501587	2	0.001401

1.2.2 Eficiência computacional em avaliações de funções e tempo de execução.

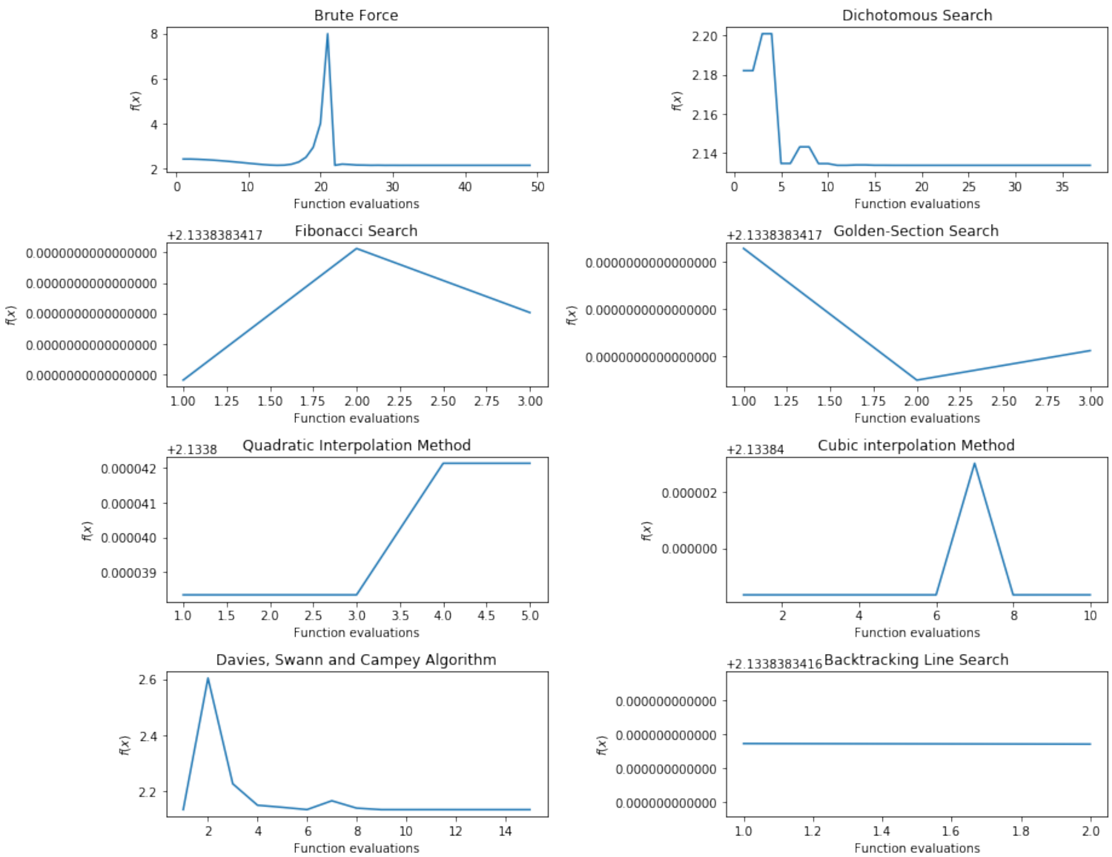
```
In [10]: fig, axes = plt.subplots(1,2, figsize=(13,5))
         fig.suptitle('$f(x) = \ln ^2 (x-2) + \ln ^2(10-x) - x^{0.2}$')
         results_43['fevals'].plot.bar(title = 'Function evals', ax= axes[0])
         (results_43['run_time (s)']*1e3).plot.bar(title = 'Run time (ms)', ax=axes[1])
         plt.show()
```



No exercício 4.3 observa-se que a força bruta continua sendo o algoritmo de menor eficiência computacional. Os algoritmos que utilizam gradiente da função também se mostraram menos eficientes computacionalmente em tempo de execução, apesar de terem efetuado poucas avaliações de funções.
Como no exercício 4.2, o Dichotomous Search permaneceu em segunda pior avaliação em *Function Evals* e *Run time (ms)*.

1.2.3 Gráficos de f(x) por avaliações.

```
In [11]: show_chart(results_43)
```

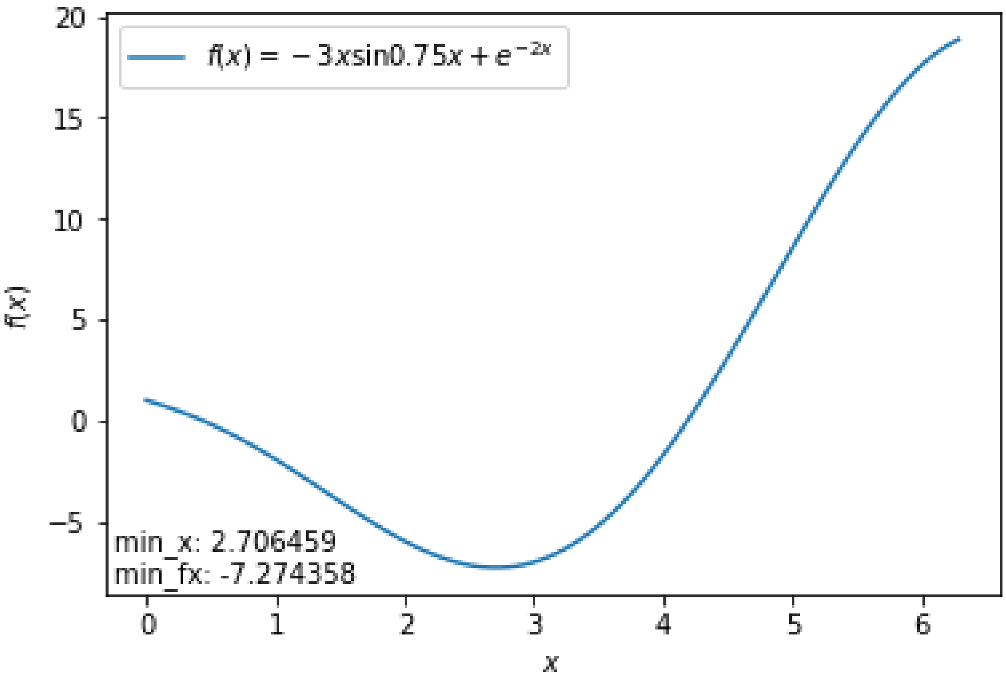


Nos gráficos de $f(x)$ por avaliações do exercício 4.3, é interessante notar que o *Dichotomous Search* obteve o mesmo comportamento do exercício 4.2, apesar de atingir valores diferentes de $f(x)$. Os algoritmos *Fibonacci Search*, *Golden-Section Search*, *Cubic Interpolation Method* e *Backtracking Line Search* atingiram o mínimo global na primeira iteração. Apesar do algoritmo *Quadratic Interpolation Method* ter variado mais do que os já citados, ele permaneceu dentrou da tolerância de erro durante as 5 iterações.

1.3 Exercício 4.4

$f(x) = -3x \sin 0.75x + e^{-2x}$

```
In [12]: results_44 = run_exercise(sinoid, f_string = '$f(x) = -3x\sin 0.75 x + e ^{-2x}$',
                                   seed =  9,
                                   interval = [0, 2*np.pi])
```



1.3.1 Resultados

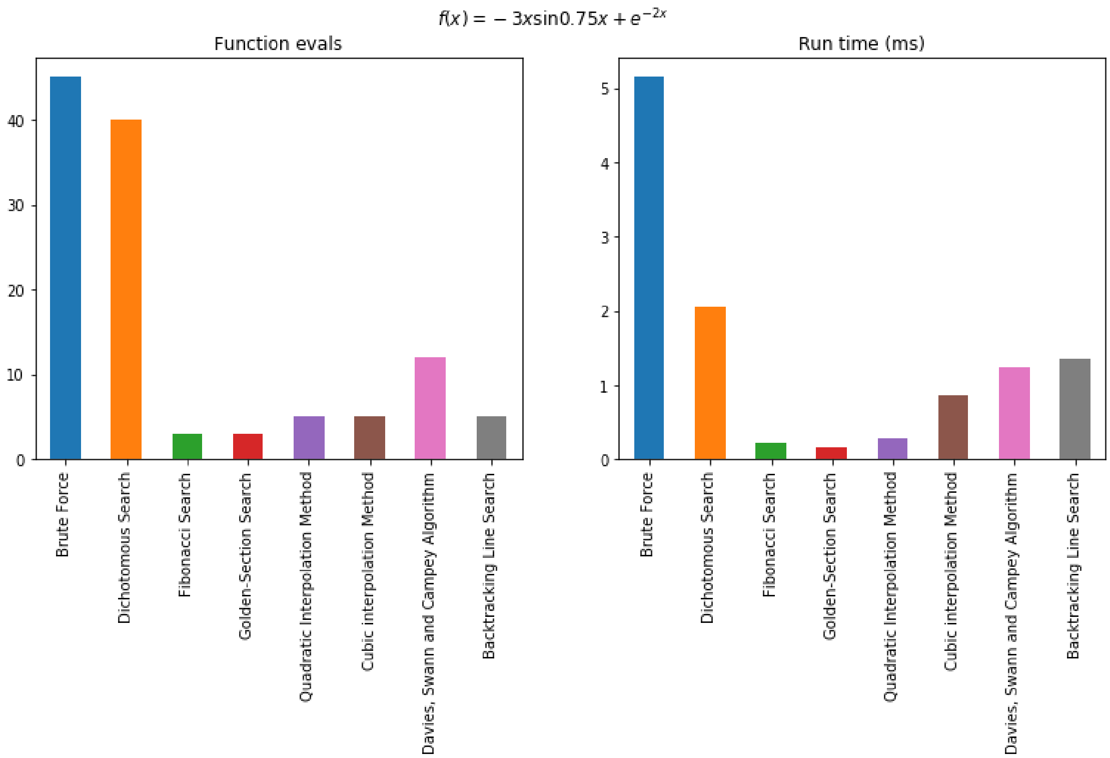
```
In [13]: results_44[['best_f', 'best_x', 'fevals', 'run_time (s)']]
```

Out[13]:

	best_f	best_x	fevals	run_time (s)
Brute Force	-7.274358	2.706459	45	0.005145
Dichotomous Search	-7.274358	2.706477	40	0.002044
Fibonacci Search	-7.274358	2.706476	3	0.000214
Golden-Section Search	-7.274358	2.706476	3	0.000165
Quadratic Interpolation Method	-7.274358	2.706476	5	0.000280
Cubic interpolation Method	-7.274358	2.706476	5	0.000867
Davies, Swann and Campey Algorithm	-7.274358	2.706475	12	0.001239
Backtracking Line Search	-7.274358	2.706476	5	0.001361

1.3.2 Eficiência computacional em avaliações de funções e tempo de execução.

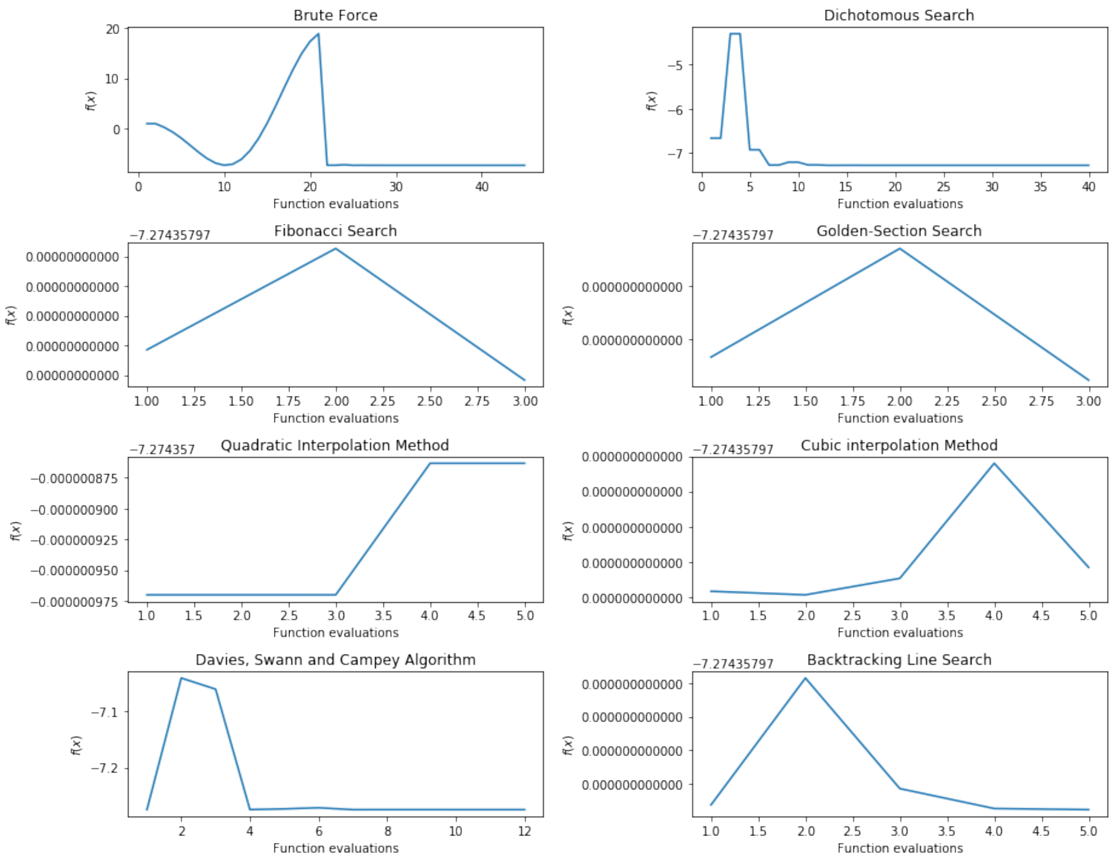
```
In [14]: fig, axes = plt.subplots(1,2, figsize=(13,5))
         fig.suptitle('$f(x) = -3x\sin 0.75 x + e ^{-2x}$')
         results_44['fevals'].plot.bar(title = 'Function evals', ax= axes[0])
         (results_44['run_time (s)']*1e3).plot.bar(title = 'Run time (ms)', ax=axes[1])
         plt.show()
```



Os resultados obtidos no exercício 4.4 foram próximos dos resultados do exercício 4.3. Após avaliação dos três primeiros exercícios, a *Golden-Section Search* se mostrou o melhor algoritmo de minimização para os três problemas.

1.3.3 Gráficos de f(x) por avaliações.

```
In [15]: show_chart(results_44)
```



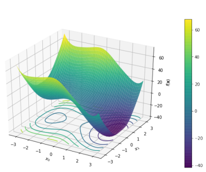
No exercício 4.4, o *Dichotomous Search* apresentou um comportamento diferente dos exercícios 4.2 e 4.3.

1.4 Exercício 4.11

$f(\mathbf{x}) = 0.7x_1^4 - 8x_1^2 + 6x_2^2 + \cos(x_1 x_2) - 8x_1$

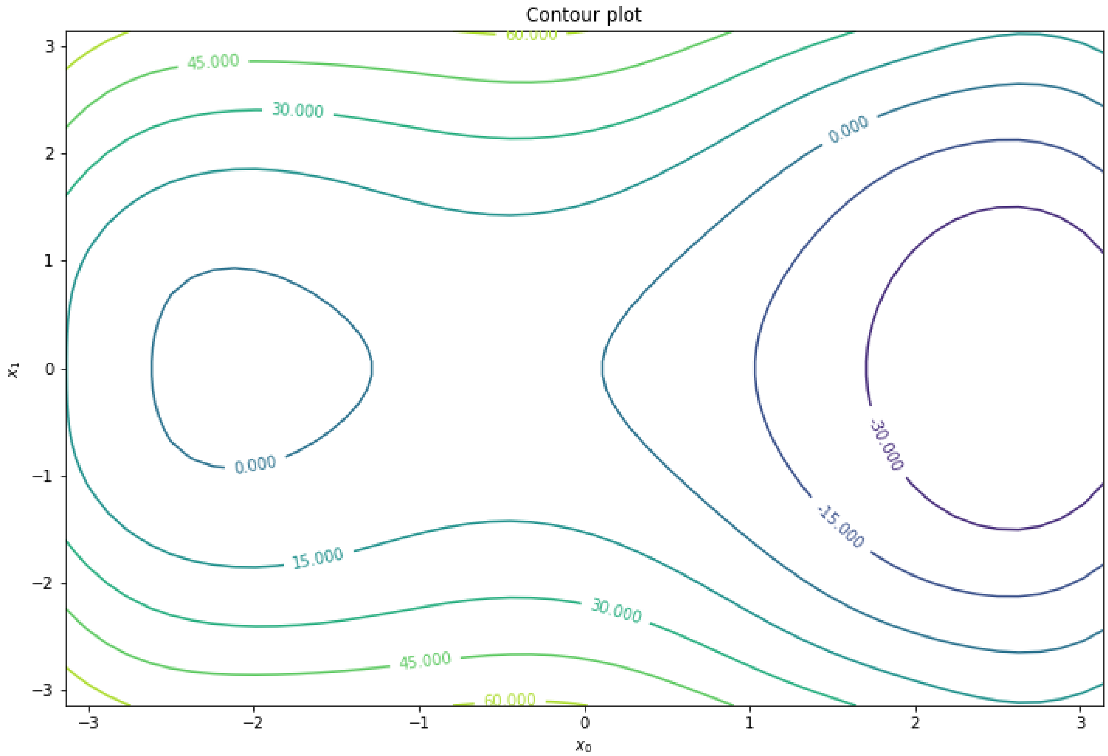
1.4.1 Item a)

```
In [16]: from run_exercises import plot_surface
         plot_surface()
```



1.4.2 Item b)

```
In [17]: from run_exercises import plot_contour
         plot_contour()
```



1.4.3 Item d)

Para o item d, rodou-se os algoritmos de Backtracking e Fletcher's Inexact Line Search. Os dois encontraram αs a partir de $\mathbf{x}_0 = [-\pi, \pi]$ e $\mathbf{d}_0 = [1.0, -1.3]$. A solução de Fletcher encontrou um custo menor que a solução de Backtracking, como visto na próxima célula.

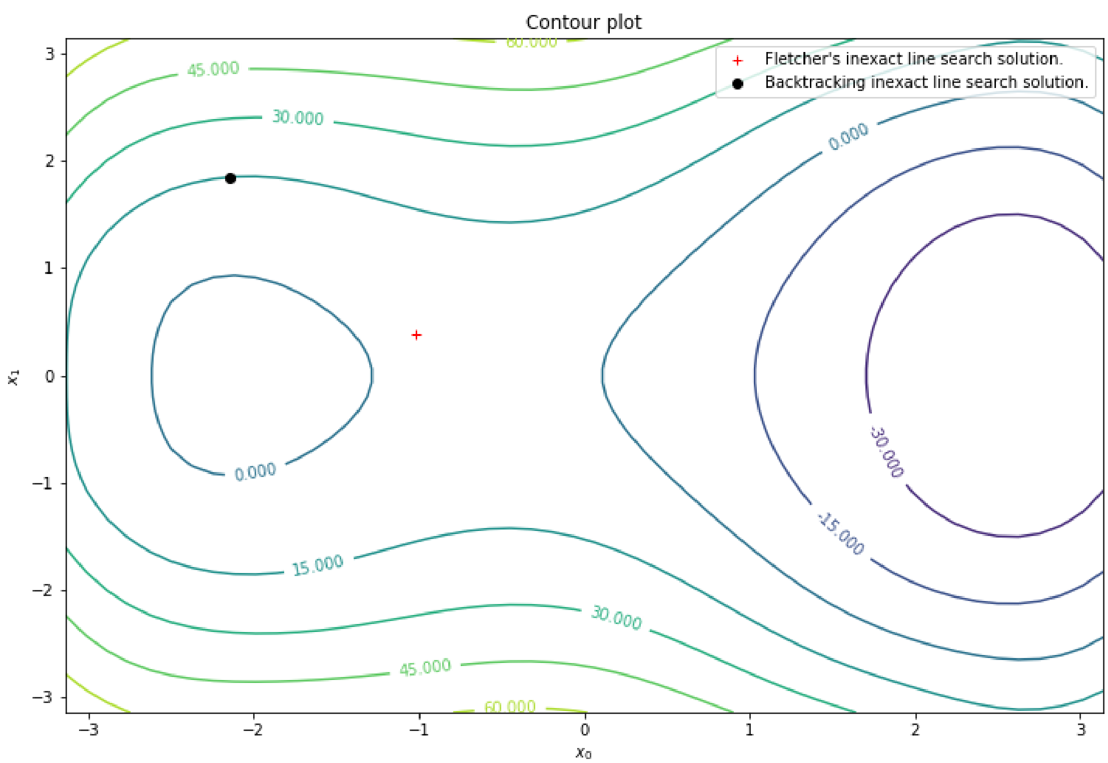
```
In [18]: from run_exercises import plot_func_alpha
         x_0 = np.array([-np.pi, np.pi])
         d_0 = np.array([1.0, -1.3])
         func = functionObj(order4_polynomial)

         item_d_optimizer = InexactLineSearch(func, x_0, d_0)
         backtracking_opt = BacktrackingLineSearch(func, x_0, d_0)
         alpha_f, f0_f = item_d_optimizer._line_search()
         alpha_b, f0_b = backtracking_opt._backtracking_line_search(func.grad(x_0))
         print('Inexact Line Search Methods:')
         print(' - Fletcher solution\n   '+u'\u00B7' + u'\u03B1'+': %.7f\n   '%alpha_f + u'\u0
         print(' - Backtracking solution\n   '+u'\u00B7'+ u'\u03B1'+': %.7f\n   '%alpha_b + u''
```

```
Inexact Line Search Methods:
 - Fletcher solution
   ů: 2.1191411
   ůf: 2.4014783
 - Backtracking solution
   ů: 1.0000000
   ůf: 14.8198176
```

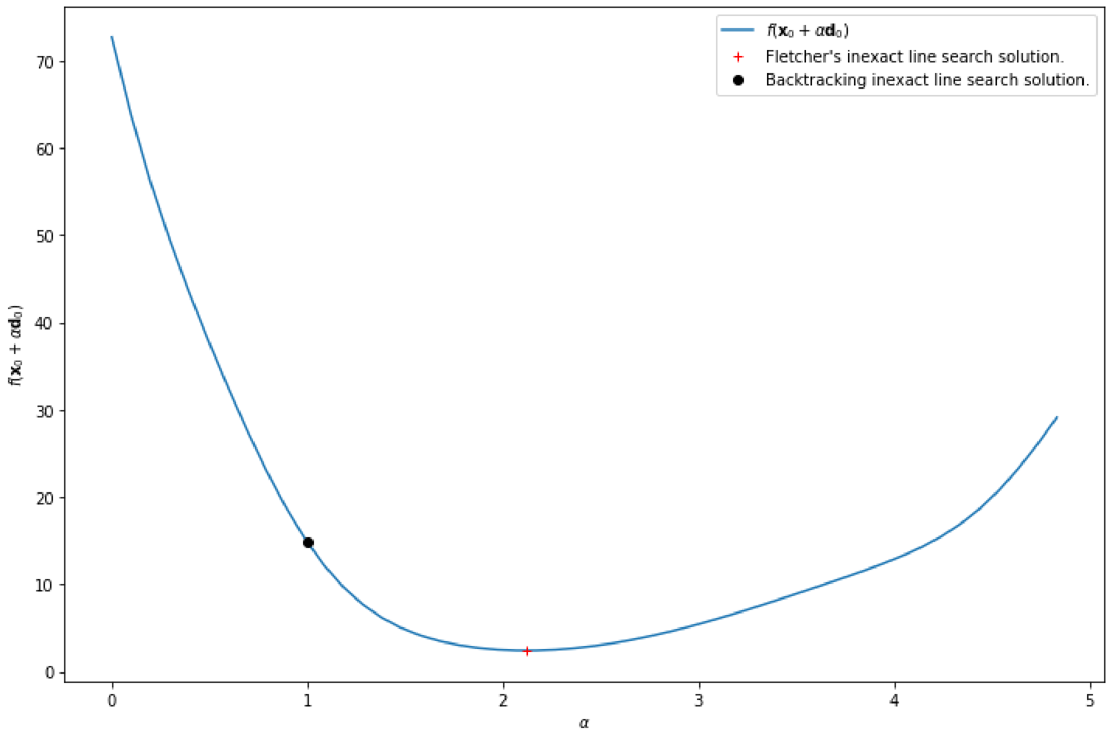
Observando o gráfico a seguir, pode-se ver que o $\mathbf{x}$ encontrado pela solução de Fletcher permaneceu na região $C = \{x | 0.0 \geq f(x) \leq 15.0\}$ mais próxima da região de mínimo local $C_{minlocal} = \{x | f(x) \leq 0.0\}$. O Backtracking encontrou uma solução na borda do nível $C_{15} = \{x | f(x) = 15.0\}$.

```
In [19]: plot_contour(x_0 + alpha_f*d_0, x_0 + alpha_b*d_0)
```



O gráfico a seguir demonstra que o Fletcher's Inexact line Search encontrou um α mínimo da função $f(\mathbf{x}_0 + \alpha \mathbf{d}_0)$ melhor do que o Backtracking inexact line search.

```
In [20]: alphas = np.linspace(0, 4.8332)
         plot_func_alpha(x_0, d_0, alphas, alpha_f, f0_f, alpha_b, f0_b)
```



1.4.4 Item e)

Para o item e), o Fletcher's Inexact Line Search encontrou uma solução melhor do que no item d). Isso se deve pelo fato do vetor direção $\mathbf{d}_0$ apontar para o mínimo global da função custo. O backtracking line search permaneceu encontrando o $\alpha = 1.0$, isso se deve ao fato a busca inexata não ter entrado na condição do *while* na primeira iteração, permanecendo assim o α como fora inicializado no código.

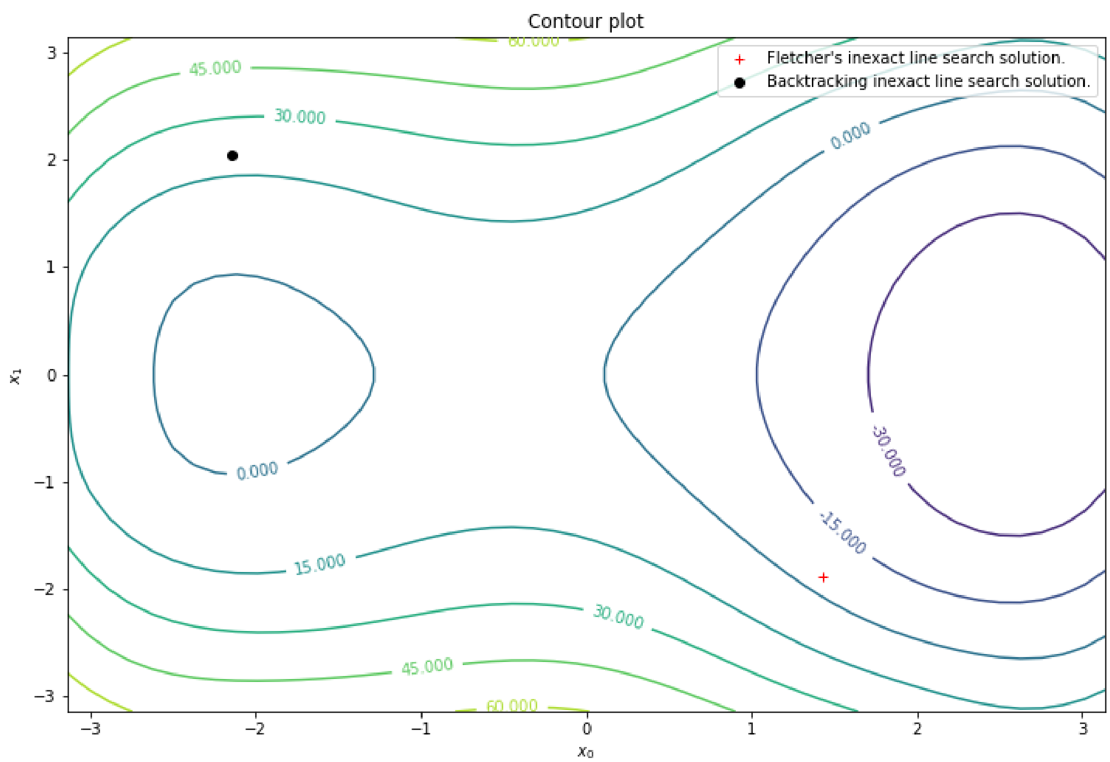
```
In [21]: x_0 = np.array([-np.pi, np.pi])
         d_0 = np.array([1.0, -1.1])
         func = functionObj(order4_polynomial)
         item_d_optimizer = InexactLineSearch(func, x_0, d_0)
         backtracking_opt = BacktrackingLineSearch(func, x_0, d_0)
         alpha_f, f0_f = item_d_optimizer._line_search()
         alpha_b, f0_b = backtracking_opt._backtracking_line_search(func.grad(x_0))
         print('Inexact Line Search Methods:')
         print(' - Fletcher solution\n    '+u'\u00B7' + u'\u03B1'+': %.7f\n    '%alpha_f + u'\u00
         print(' - Backtracking solution\n    '+u'\u00B7'+ u'\u03B1'+': %.7f\n    '%alpha_b + u'
```

```
Inexact Line Search Methods:
 - Fletcher solution
   ů: 4.5730216
   ůf: -4.4062606
 - Backtracking solution
```

```
ů: 1.0000000
ůf: 19.8410511
```

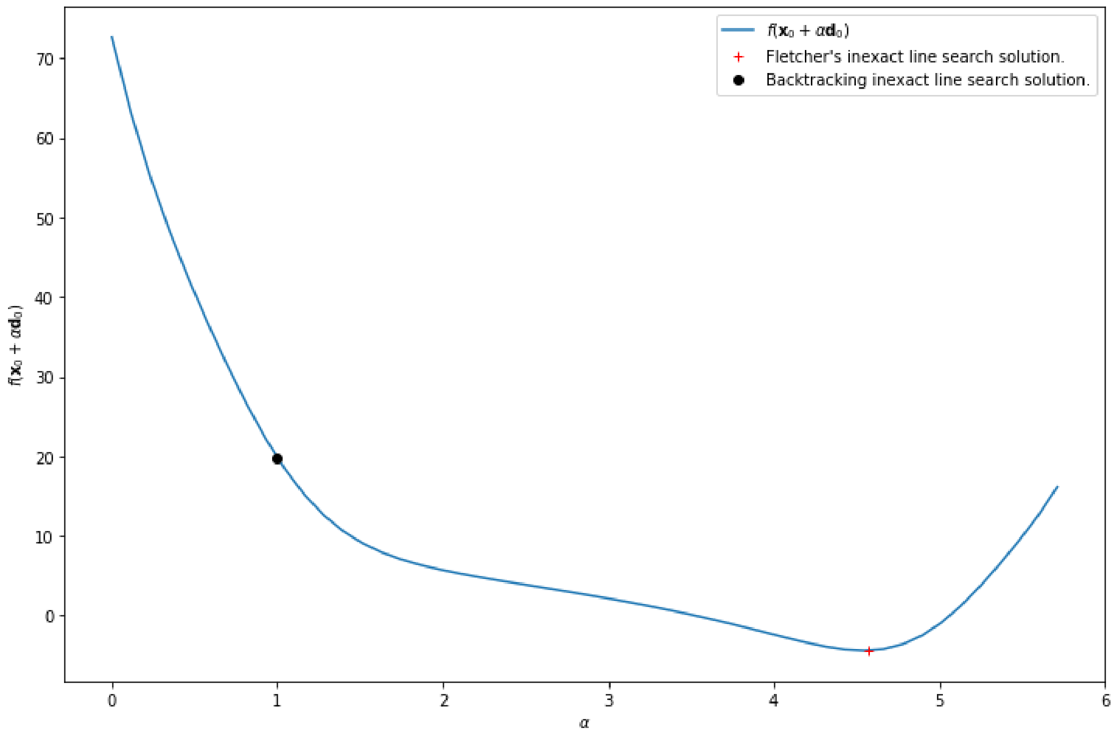
O gráfico a seguir mostra que a solução para o α encontrado pelo inexact line search do Fletcher está mais próxima do mínimo global. No caso do Backtracking, como a condição de entrada do *while* o α permaneceu igual a 1.0, como a direção inicial $\mathbf{d}_0$ aponta para o mínimo global, o resultado encontrado pelo $\alpha = 1.0$ se mostrou pior que no item d).

```
In [22]: from run_exercises import plot_func_alpha
         plot_contour(x_0 + alpha_f*d_0, x_0 + alpha_b*d_0)
```



Novamente é possível ver que o a solução da busca em linha de Fletcher encotnrou um α mínimo da função $f(\mathbf{x}_0 + \alpha\mathbf{d}_0)$

```
In [23]: alphas = np.linspace(0, 5.7120)
         plot_func_alpha(x_0, d_0, alphas, alpha_f, f0_f, alpha_b, f0_b)
```



$f(\mathbf{x}_0 + \alpha\mathbf{d}_0)$
Fletcher's inexact line search solution.
Backtracking inexact line search solution.
70
60
50
40
30
20
10
0
0
1
2
3
4
5
6
α
$f(\mathbf{x}_0 + \alpha\mathbf{d}_0)$