

# Tarefa 3

## MO431 - 1s2021

G. M. Perrotta  
173846

R. F. Prudencio  
186145

S. Chenatti  
177065

April 25, 2021

## 1 Setup

```
import sys
import time
from typing import Optional

try:
    from functools import cached_property
    from time import perf_counter_ns
except ImportError:
    cached_property = property

    def perf_counter_ns():
        return time.perf_counter() * 10e9

import timeit

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pybobyqa
from IPython.display import set_matplotlib_formats
from mpl_toolkits.mplot3d import Axes3D
from scipy.optimize import line_search, minimize

assert sys.version_info.major == 3
assert sys.version_info.minor >= 6
!python --version

# Output pngs and pdfs for matplotlib artifacts
set_matplotlib_formats("svg", "pdf")

# Use TeX fonts for plots
mpl.rc("text", usetex=True)
mpl.rc("text.latex", preamble=r"\usepackage{amssymb}")

# Set default figsize
mpl.rcParams["figure.figsize"] = (5, 5)
```

```

# Optimization parameters
TOL = 1e-5
INITIAL_XY = 4.0, 4.0
INITIAL_SIMPLEX_NM = [(-4, 4), (-4, 1), (4, -1)]
NUMBER_OF_EXECUTIONS = 100

# Stores information about the algorithms execution
EXEC_INFO = {}

```

Matplotlib is building the font cache; this may take a moment.

Python 3.8.9

## 2 Função de Himmelblau

A função de Himmelblau é uma função não-convexa em 2D com 4 mínimos globais definida por

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$

Nesta tarefa vamos experimentar com diferentes algoritmos de otimização para encontrar um ponto de mínimo da função.

```

class Himmelblau:
    def __init__(self):
        self.xs = []
        self.steps = 0
        self.grad_steps = 0

    def __call__(self, xs):
        """Evaluate the function at given point."""
        self.xs.append(xs)
        self.steps += 1
        x, y = xs
        return (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2

    def _gradx(self, xs):
        """Gradient with respect to x."""
        x, y = xs
        return 4 * x * (x ** 2 + y - 11) + 2 * (x + y ** 2 - 7)

    def _grady(self, xs):
        """Gradient with respect to y."""
        x, y = xs
        return 4 * y * (y ** 2 + x - 7) + 2 * (y + x ** 2 - 11)

    def grad(self, x):
        """Return a 2-tuple with the gradient with respect to x and y."""
        self.grad_steps += 1
        return self._gradx(x), self._grady(x)

    @cached_property
    def x_dom(self):

```

```

    """Define the x domain for plotting."""
    return np.linspace(-4.5, 4.5, 1000)

    @cached_property
    def y_dom(self):
        """Define the y domain for plotting."""
        return np.linspace(-4.5, 4.5, 1000)

    @cached_property
    def levels(self):
        """Define contour levels."""
        low_levels = [10, 25, 40, 60, 80]
        high_levels = list(np.arange(100, 300, 40))
        return low_levels + high_levels

    def reset(self):
        self.xs = []
        self.steps = 0
        self.grad_steps = 0

```

```

def plot_surf(ax, fn):
    """Make a surface plot of `fn`."""
    xs, ys = np.meshgrid(fn.x_dom, fn.y_dom)
    zs = fn(np.stack([xs, ys]))
    surf = ax.plot_surface(xs, ys, zs, alpha=0.8, cmap="plasma")
    ax.set_xlabel("$x$")
    ax.set_ylabel("$y$")
    ax.set_zlabel("$f(x, y)$")
    ax.view_init(azim=-45)

def plot_cont(ax, fn):
    """Make a contour plot of `fn`."""
    xs, ys = np.meshgrid(fn.x_dom, fn.y_dom)
    zs = fn(np.stack([xs, ys]))
    cs = ax.contour(xs, ys, zs, levels=fn.levels, cmap="plasma")
    ax.clabel(cs, fmt="%d")
    ax.set_xlabel("$x$")
    ax.set_ylabel("$y$")

def plot_opt_path(ax, fn, xs):
    """Plot the optimization path along a 2D or 3D ax."""
    x = np.clip(xs[0], np.min(fn.x_dom), np.max(fn.x_dom))
    y = np.clip(xs[1], np.min(fn.y_dom), np.max(fn.y_dom))

    if isinstance(ax, Axes3D):
        zs = fn(np.stack([x, y]))
        coords = (x, y, zs)
    else:
        coords = (x, y)

    # Plot the optimization path
    ax.plot(*coords, color="k")

    # Plot the initial and final estimates of the parameters x.

```

```

ax.scatter(*map(lambda c: c[0], coords), color="r", marker="^", label=r"\mathbf{x_0}")
ax.scatter(*map(lambda c: c[-1], coords), color="r", marker="*", label=r"\mathbf{x_*}")
ax.legend()

def plot_surf_with_opt_path(fn, xs):
    """Helper function to plot the function's surface with the optimization path."""
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    plot_surf(ax, fn)
    plot_opt_path(ax, fn, xs)
    fig.tight_layout()

def plot_cont_with_opt_path(fn, xs):
    """Helper function to plot the function's surface with the optimization path."""
    fig, ax = plt.subplots()
    plot_cont(ax, fn)
    plot_opt_path(ax, fn, xs)
    fig.tight_layout()

```

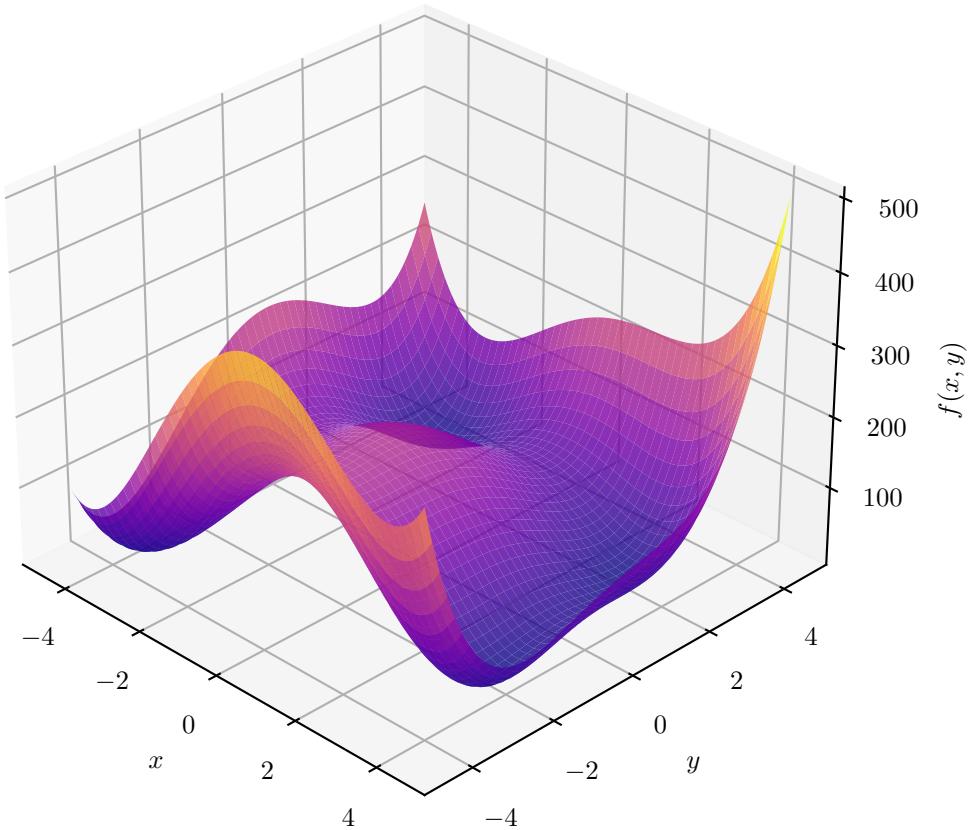
A seguir, podemos visualizar a superfície da função.

```

himmelblau = Himmelblau()

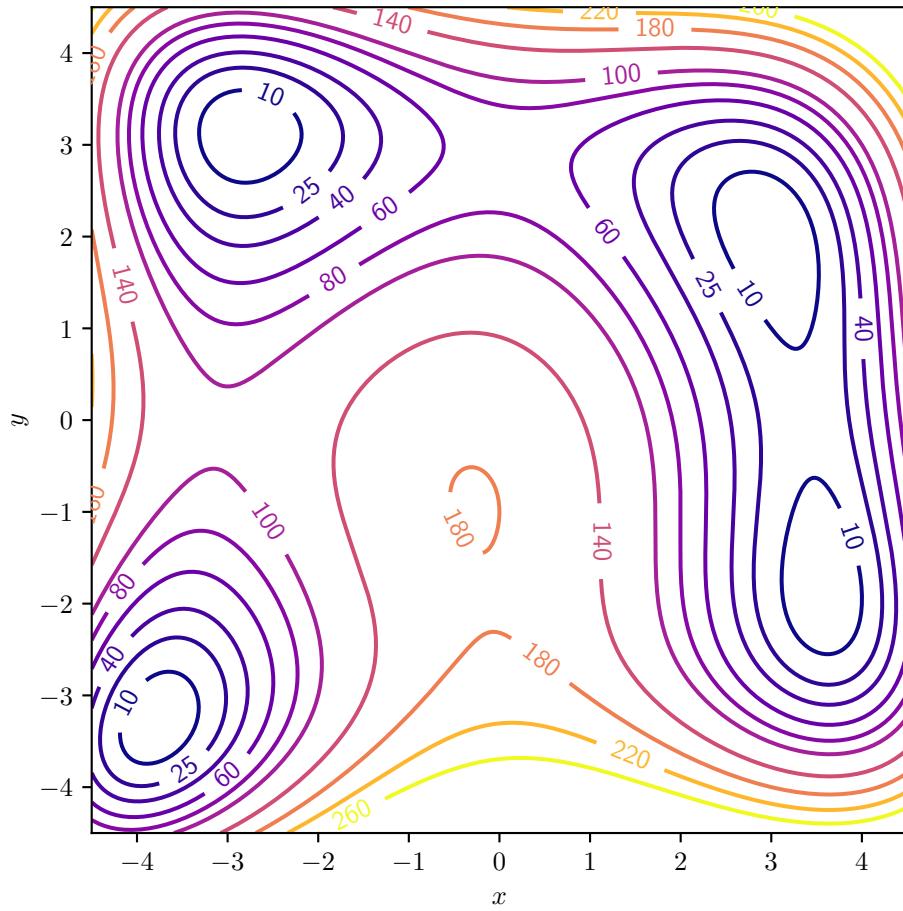
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
plot_surf(ax, himmelblau)
fig.tight_layout()

```



Note como a função possui quatro pontos de mínimo para o intervalo  $x, y \in [-4, 5, -4, 5]$  do domínio. Isso fica ainda mais claro pelo gráfico de contorno dado a seguir.

```
fig, ax = plt.subplots()
plot_cont(ax, himmelblau)
fig.tight_layout()
```



### 3 Métodos de otimização

Nesta seção vamos apresentar os resultados de 5 métodos de otimização. A seguir, definimos duas funções de utilidade para imprimir informações a respeito de cada método.

```

def print_aligned(tokens_dict, sep="|"):
    """Print the aligned keys and values of `tokens_dict`."""
    max_len = max(map(len, tokens_dict))
    for k, v in tokens_dict.items():
        print(f"{k:{max_len}s} {sep} {v}")

def opt_info(func):
    """Optimization info decorator.

    Prints information about optimization algorithm including:
    * The minimum point found;
    * The value of the optimized function at the minimum point;
    * The number of calls made to the function during optimization;
    * The elapsed time of the algorithm.
    """

```

```

"""
def wrapper_opt_info(fn, *args, **kwargs):

    fn.reset()
    mean_time = timeit.timeit(lambda: func(fn, *args, **kwargs), number=NUMBER_OF_EXECUTIONS)
    mean_time /= NUMBER_OF_EXECUTIONS
    mean_steps = fn.steps / NUMBER_OF_EXECUTIONS
    mean_grad_steps = fn.grad_steps / NUMBER_OF_EXECUTIONS

    # Runs one more time to acquire the results
    fn.reset()
    x_sol = func(fn, *args, **kwargs)
    steps = fn.steps
    grad_steps = fn.grad_steps

    info = {
        "Mínimo encontrado": f"f(x={x_sol[0]:.4f}, y={x_sol[1]:.4f}) = {fn(x_sol):.4f}",
        "Tempo médio de execução": f"{mean_time * 1e3:.4f}ms",
        "Número de chamadas para f(x)": steps,
        "Número de chamadas para df(x)/dx": grad_steps,
    }

    print_aligned(info)

    return np.stack(fn.xs, axis=-1), (mean_time, mean_steps, mean_grad_steps)

return wrapper_opt_info

```

### 3.1 Conjugado gradiente

A seguir apresentamos o método do conjugado gradiente para encontrar um mínimo da função.

```

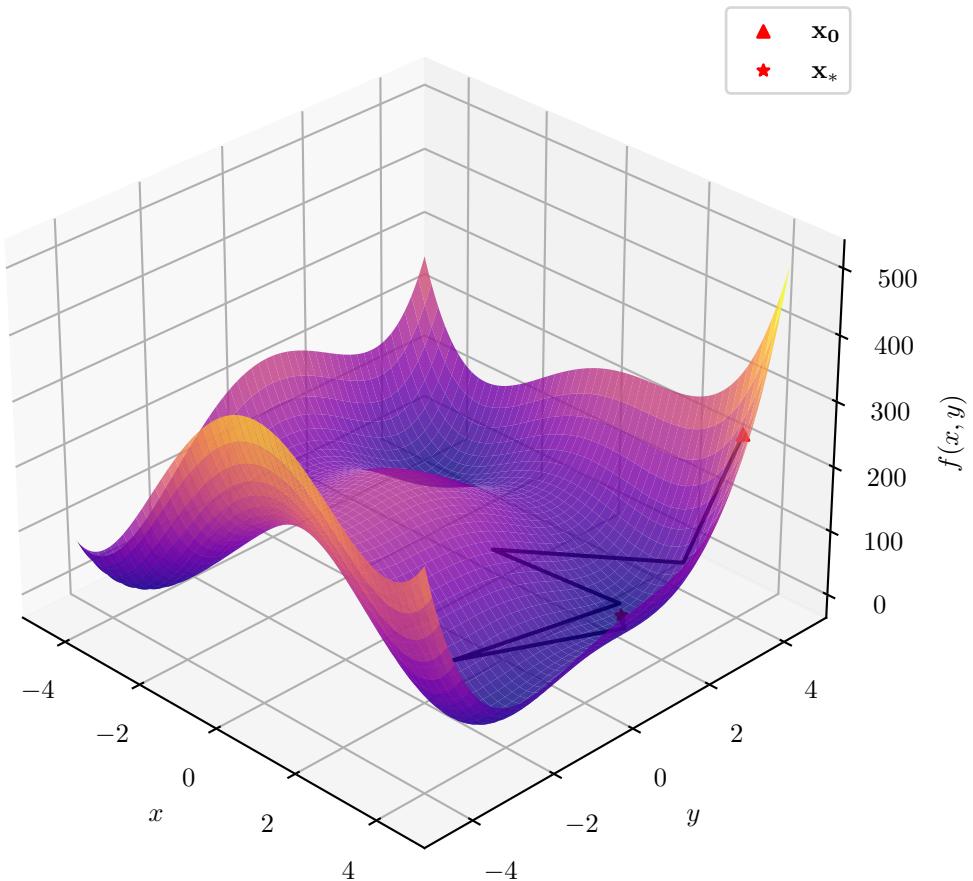
@opt_info
def minimize_cg(fn, x0):
    opt_res = minimize(fn, x0, jac=fn.grad, method="CG")
    return opt_res.x

xs, EXEC_INFO["Gradiente conjugado"] = minimize_cg(himmelblau, INITIAL_XY)

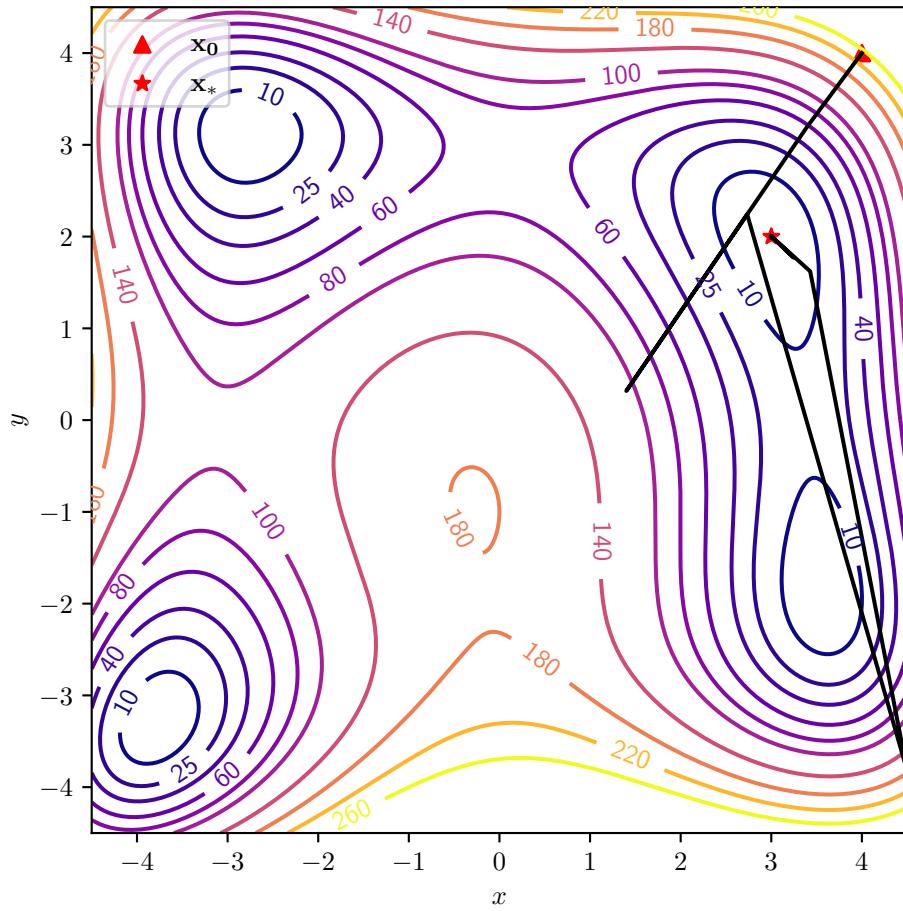
```

Mínimo encontrado	f(x=3.0000, y=2.0000) = 0.0000
Tempo médio de execução	1.4506ms
Número de chamadas para f(x)	17
Número de chamadas para df(x)/dx	17

```
plot_surf_with_opt_path(himmelblau, xs)
```



```
plot_cont_with_opt_path(himmelblau, xs)
```



### 3.2 Descida do gradiente com busca em linha

```
@opt_info
def minimize_line_search(fn, x0, tol):
    xk = np.array(x0)
    pk = -np.array(himmelblau.grad(xk))
    old_fval = np.inf
    new_fval = fn(xk)

    while np.abs(new_fval - old_fval) > tol:
        alpha, _, _, new_fval, old_fval, new_pk = line_search(fn, fn.grad, xk, pk)

        if alpha is None:
            return xk

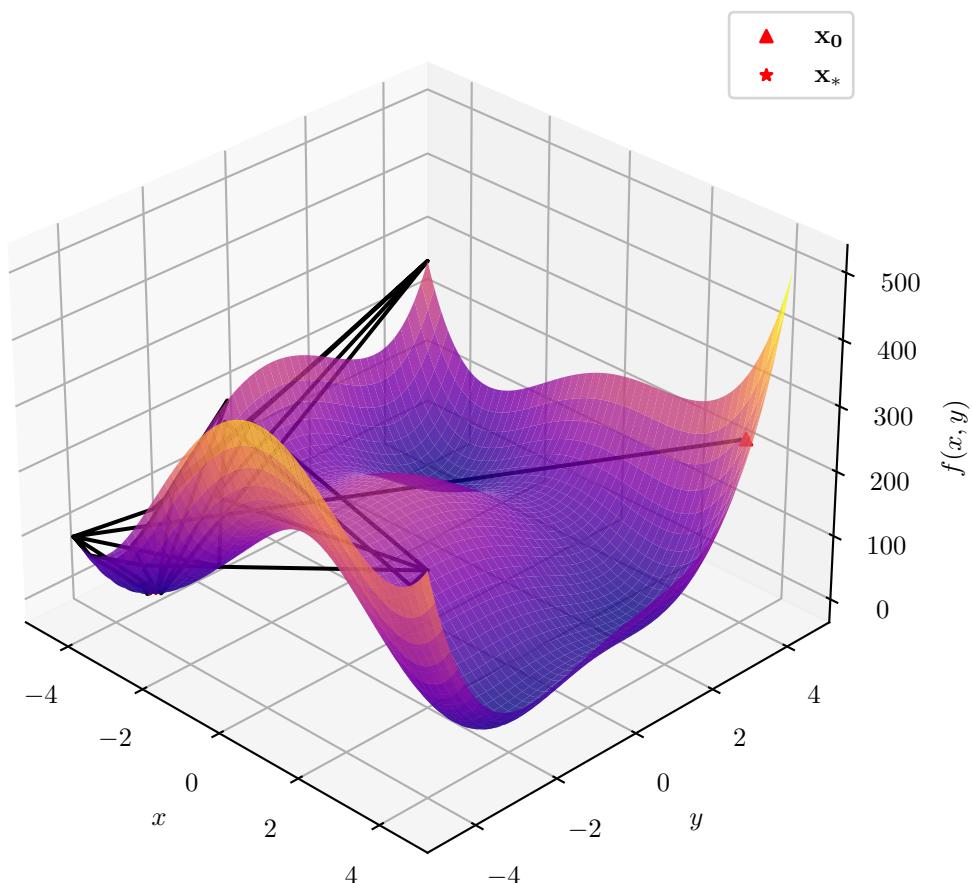
        xk = xk + alpha * np.array(pk)
        pk = -np.array(new_pk)

    return xk
```

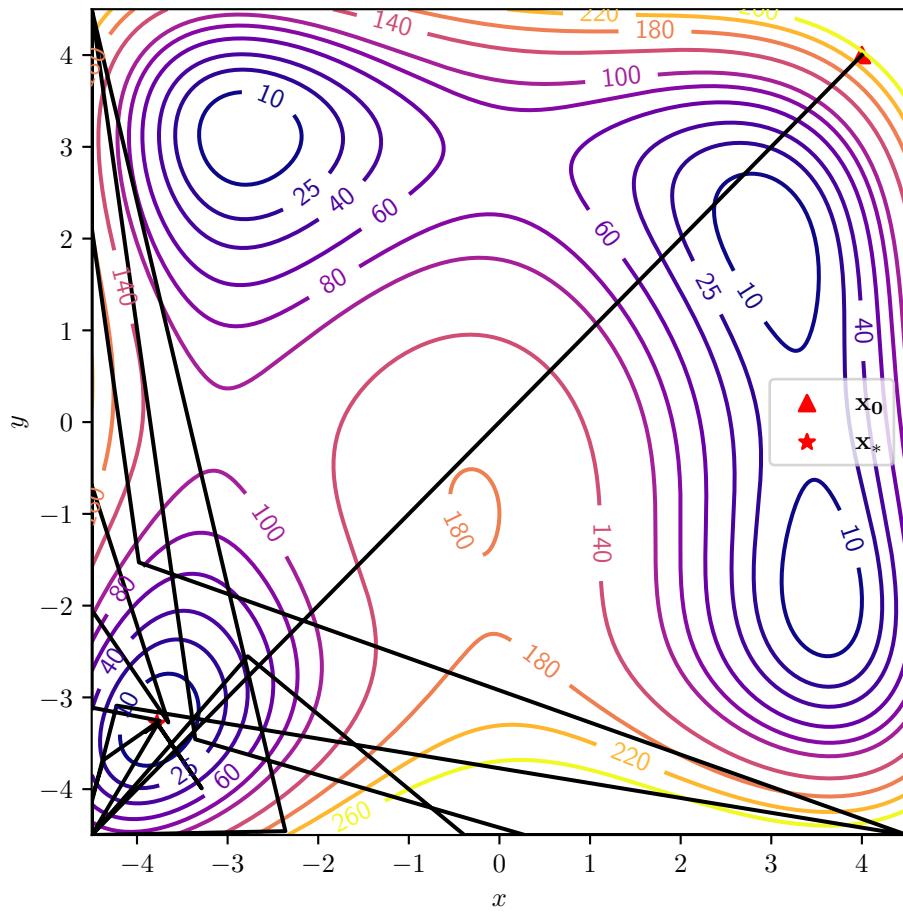
```
xs, EXEC_INFO["Busca em linha"] = minimize_line_search(himmelblau, INITIAL_XY, TOL)
```

Mínimo encontrado	$f(x=-3.7792, y=-3.2831) = 0.0000$
Tempo médio de execução	3.2781ms
Número de chamadas para $f(x)$	77
Número de chamadas para $df(x)/dx$	23

```
plot_surf_with_opt_path(himmelblau, xs)
```



```
plot_cont_with_opt_path(himmelblau, xs)
```



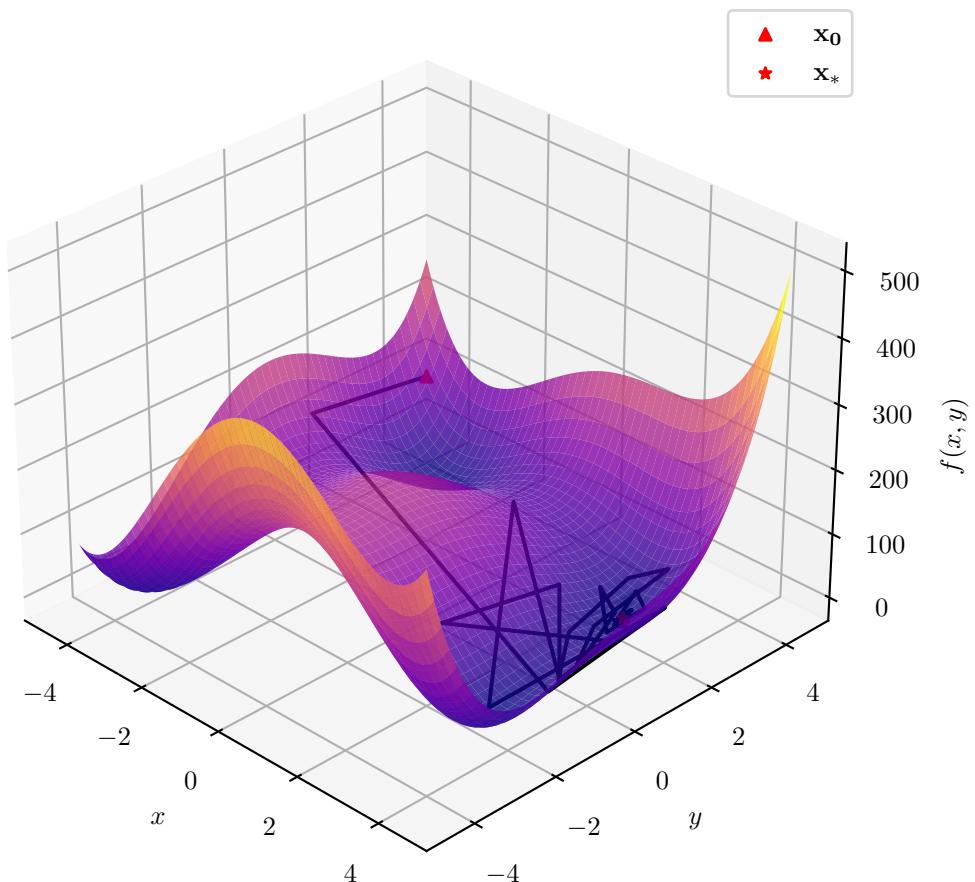
### 3.3 Nelder-Mead

```
@opt_info
def minimize_nelder_mead(fn, x0):
    opt_res = minimize(fn, x0=np.empty(2), method="Nelder-Mead", options={"initial_simplex": x0})
    return opt_res.x

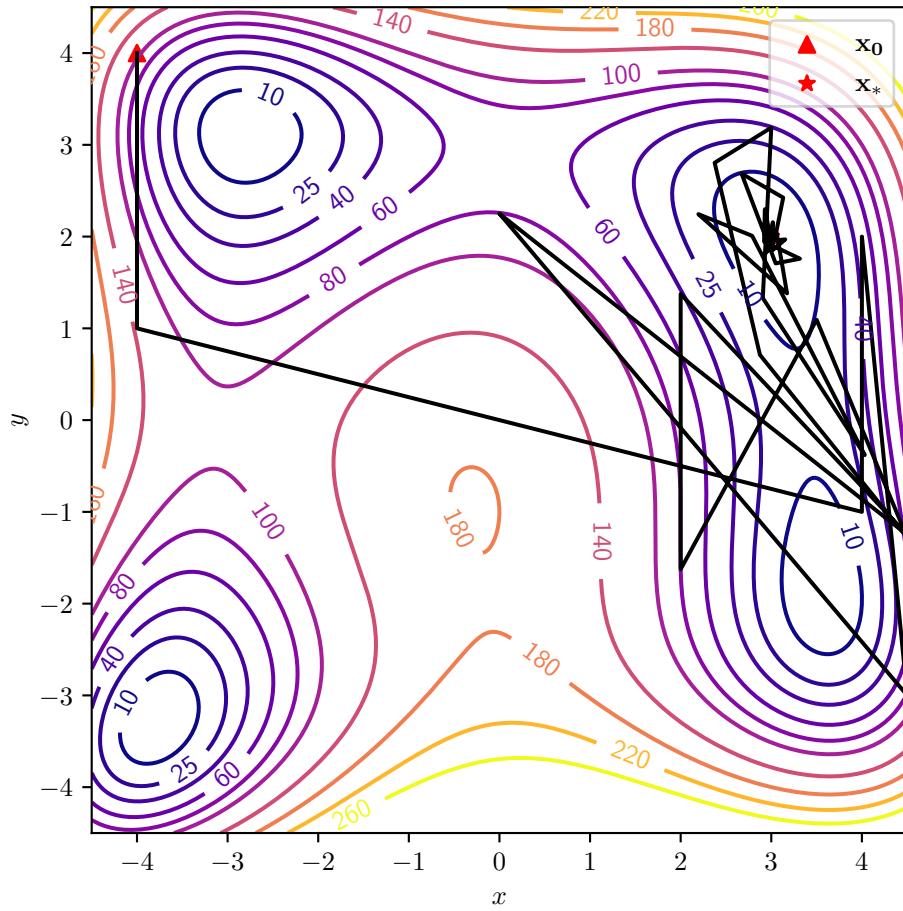
xs, *nelder_mead_info = minimize_nelder_mead(himmelblau, INITIAL_SIMPLEX_NM)
```

Mínimo encontrado	$f(x=3.0000, y=2.0000) = 0.0000$
Tempo médio de execução	2.1710ms
Número de chamadas para $f(x)$	76
Número de chamadas para $df(x)/dx$	0

```
plot_surf_with_opt_path(himmelblau, xs)
```



```
plot_cont_with_opt_path(himmelblau, xs)
```



### 3.4 Broyden-Fletcher-Goldfarb-Shanno (BFGS)

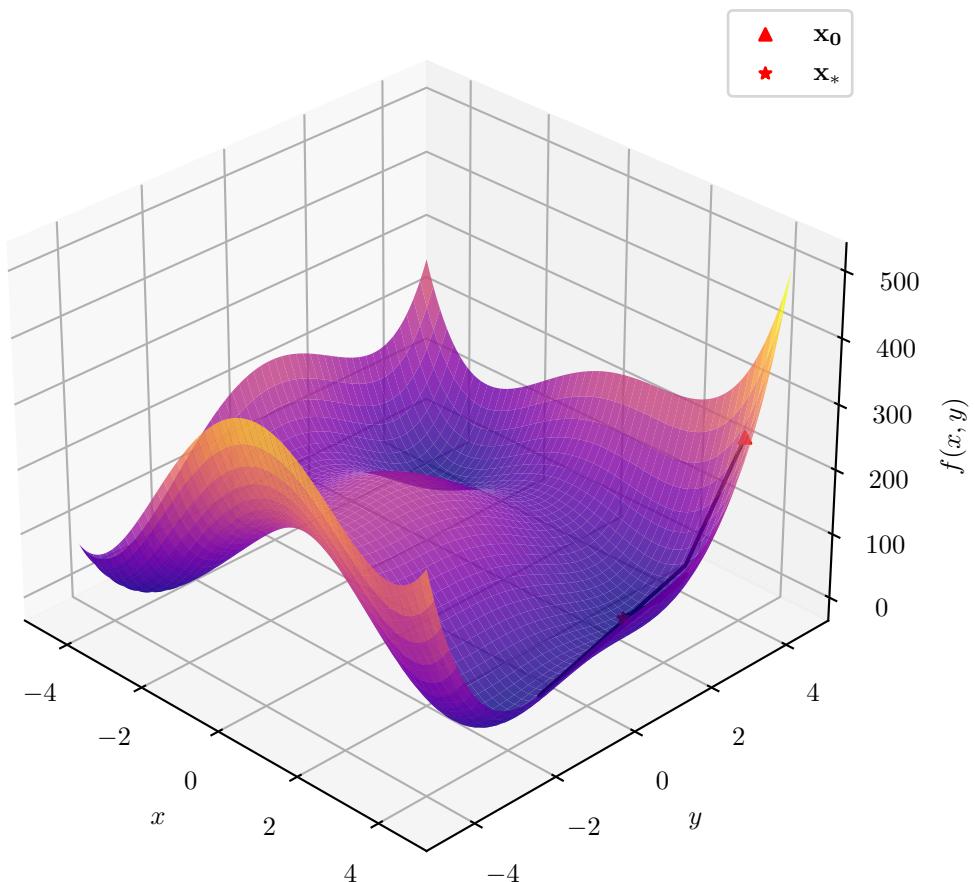
```
@opt_info
def minimize_bfgs(fn, x0, with_grad=False):
    opt_res = minimize(fn, x0, method="BFGS", jac=fn.grad if with_grad else None)
    return opt_res.x
```

#### 3.4.1 BFGS sem gradiente

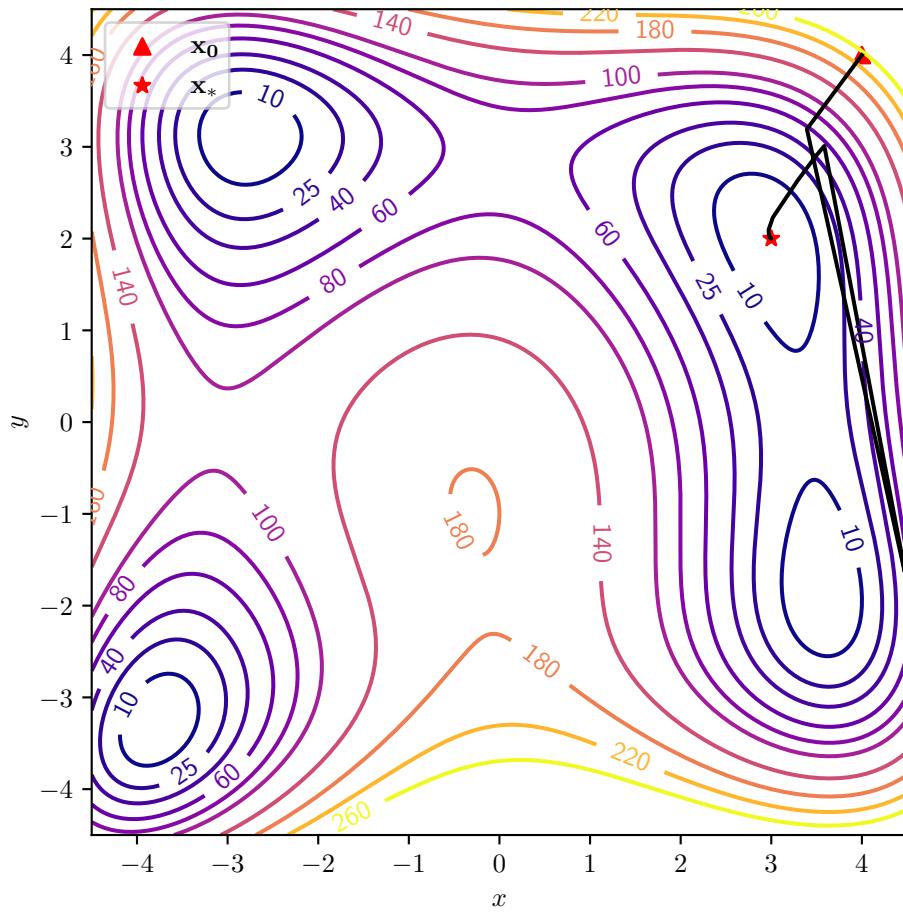
```
xs, EXEC_INFO["BFGS sem gradiente"] = minimize_bfgs(himmelblau, INITIAL_XY, with_grad=False)
```

Mínimo encontrado	$f(x=3.0000, y=2.0000) = 0.0000$
Tempo médio de execução	4.6606ms
Número de chamadas para $f(x)$	48
Número de chamadas para $df(x)/dx$	0

```
plot_surf_with_opt_path(himmelblau, xs)
```



```
plot_cont_with_opt_path(himmelblau, xs)
```

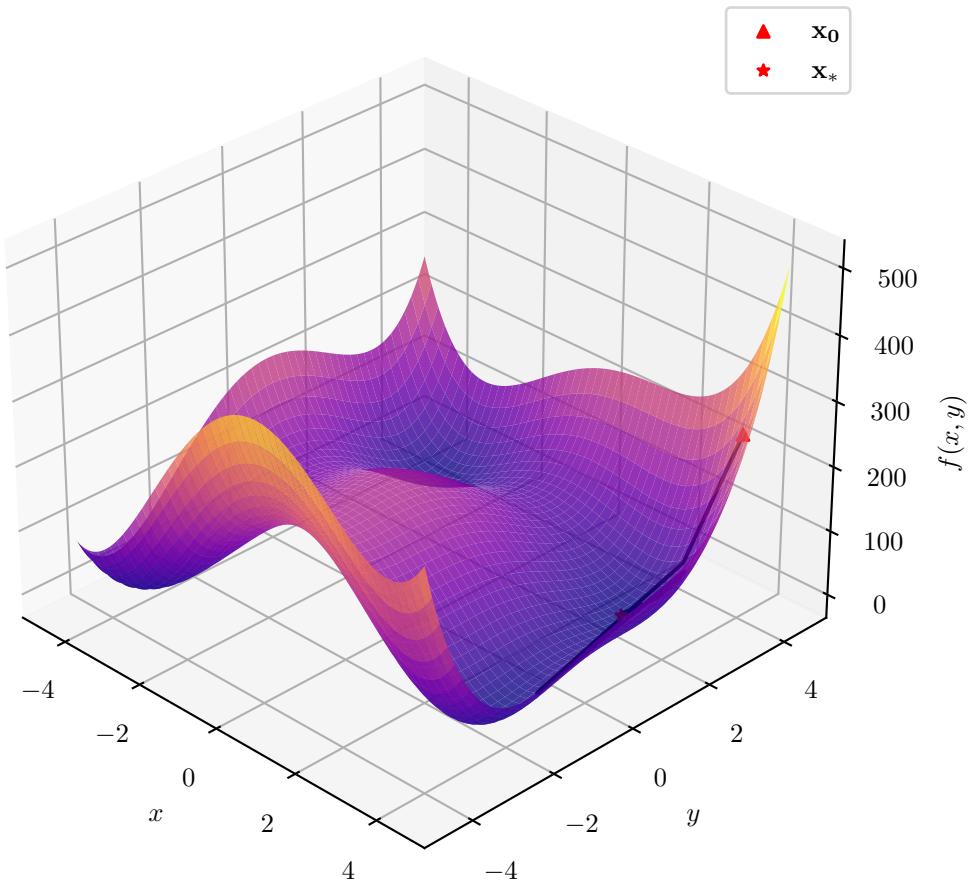


### 3.4.2 BFGS com gradiente

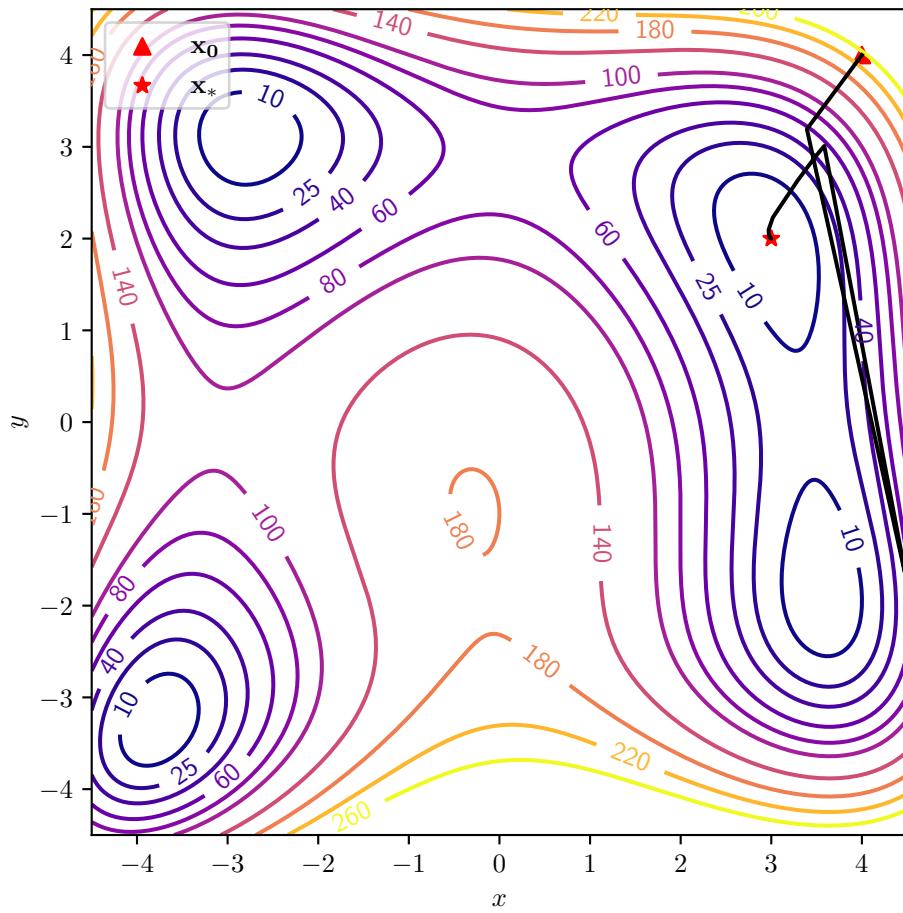
```
xs, EXEC_INFO["BFGS com gradiente"] = minimize_bfgs(himmelblau, INITIAL_XY, with_grad=True)
```

Mínimo encontrado	$f(x=3.0000, y=2.0000) = 0.0000$
Tempo médio de execução	1.8325ms
Número de chamadas para $f(x)$	16
Número de chamadas para $df(x)/dx$	16

```
plot_surf_with_opt_path(himmelblau, xs)
```



```
plot_cont_with_opt_path(himmelblau, xs)
```



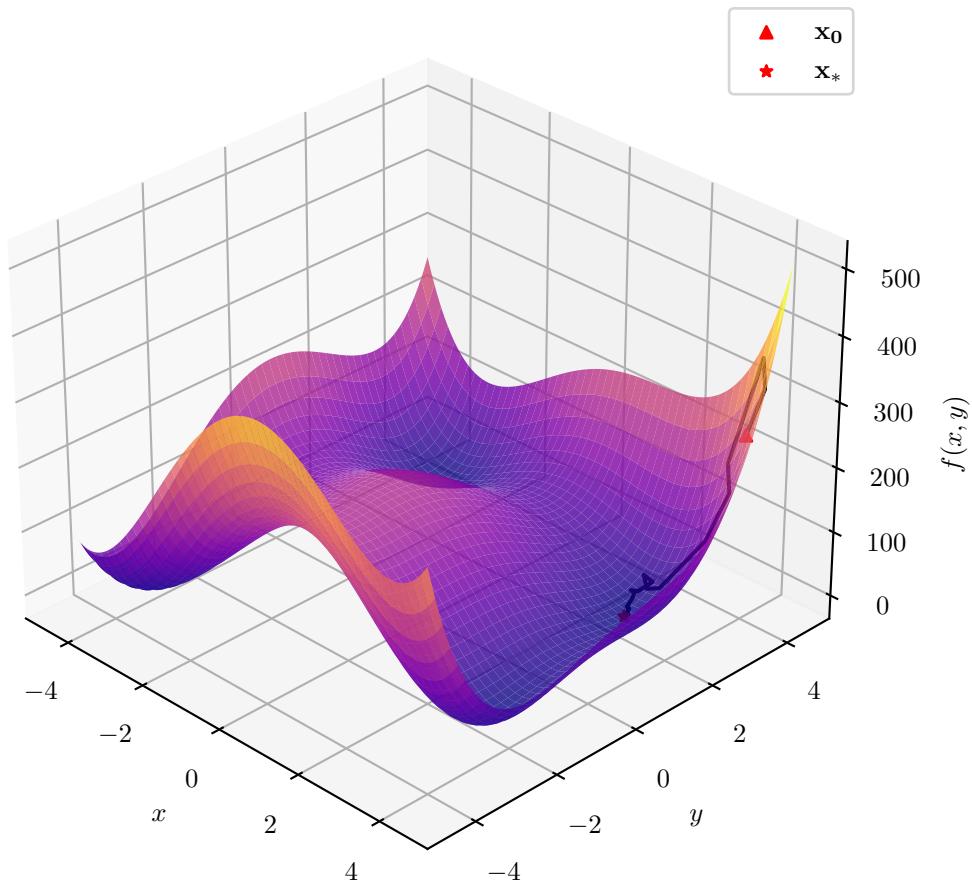
### 3.5 BOBYQA

```
@opt_info
def minimize_bobyqa(fn, x0=None):
    opt_res = pybobyqa.solve(fn, np.array(x0))
    return opt_res.x
```

```
xs, EXEC_INFO["BOBYQA"] = minimize_bobyqa(himmelblau, INITIAL_XY)
```

Mínimo encontrado	$f(x=3.0000, y=2.0000) = 0.0000$
Tempo médio de execução	123.9193ms
Número de chamadas para $f(x)$	58
Número de chamadas para $df(x)/dx$	0

```
plot_surf_with_opt_path(himmelblau, xs)
```



## 4 Comparação entre os métodos

Como observado, todos os algoritmos foram capazes de encontrar um mínimo global. De qualquer forma, cada algoritmo, de acordo com as suas peculiaridades, apresenta um desempenho diferente.

A seguir, comparamos os métodos de otimização a partir do tempo médio de execução em 100 iterações e o número médio de chamadas realizadas para  $f(x, y)$  e  $\nabla f(x, y)$  por cada algoritmo.

```
def plot_methods_performance(algorithms_info):
    """Compares the algorithms' performance."""

    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
    for method_name, (time, steps, grad_steps) in algorithms_info.items():
        axes[0].plot(time, steps, "o", label=method_name)
        axes[1].plot(time, grad_steps, "o", label=method_name)
```

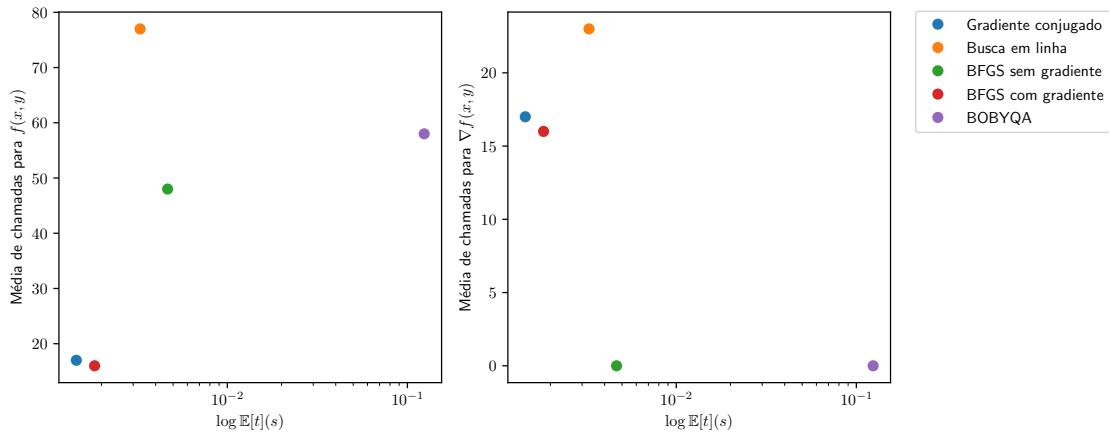
```

axes[0].set_ylabel("Média de chamadas para  $f(x, y)$ ")
axes[1].set_ylabel("Média de chamadas para  $\nabla f(x, y)$ ")
for ax in axes:
    ax.set_xlabel("$\log \mathbb{E}[t] (s)$")
    ax.set_xscale("log")

handles, labels = ax.get_legend_handles_labels()
fig.legend(handles, labels, loc="upper left", bbox_to_anchor=(1.0, 0.97))
fig.tight_layout()

plot_methods_performance(EXEC_INFO)

```



A partir do gráfico acima, observamos que o BOBYQA é o algoritmo mais lento. Para o BOBYQA, foram feitos testes com duas bibliotecas: pdf o e pybobqa, mas as duas apresentaram resultados similares, onde o tempo de execução do BOBYQA é cerca de 2 ordens de grandeza maior dos demais algoritmos. Ainda assim, resta a dúvida se a discrepância no tempo de execução é fruto da implementação, já que para os demais algoritmos usamos o `scipy`, uma biblioteca muito mais consolidada na comunidade.

Mesmo apresentando o pior desempenho em tempo de execução, o BOBYQA não é o algoritmo que realiza mais chamadas à função. Pelo gráfico acima, podemos ver que a busca em linha realiza o maior número de chamadas, tanto para  $f(x, y)$  como  $\nabla f(x, y)$ . Exceto pela busca em linha, podemos ver que os algoritmos que utilizam o gradiente, como o BFGS com gradiente e o gradiente conjugado, realizam muito menos passos para chegar no mínimo.

Apesar do tempo de execução ser a métrica de maior interesse para uma dada função  $f$  que queremos minimizar, o número de passos é um bom indicativo da performance do algoritmo caso a função  $f$  seja mais cara de avaliar (e.g. realizar inferência com uma rede neural). Por isso, os algoritmos com gradiente podem ser mais atraentes para funções mais complexas de avaliar.