Tarefa 2
MO431 - 1s2021

Guilherme Mendeleh Perrotta
173846

Rafael Figueiredo Prudencio
186145

Samuel Chenatti
177065

April 18, 2021

1 Setup

[1]:
```
import sys
from functools import cached_property
from typing import Sequence, Tuple

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import torch
from matplotlib import ticker

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

# Use TeX fonts for plots
mpl.rc("text", usetex=True)

# Rosenbrock function parameters
A = 1
B = 100

# Optimization parameters
TOL = 1e-5
MAX_STEPS = 5e4
XY_INITIAL = 0.0, 0.0
```

```
Python 3.8.3
```

2 Função de Rosenbrock

A função de Rosenbrock é uma função não-convexa em 2D definida a partir de dois parâmetros a e b como

$$f(x, y) = (a - x)^2 + b(y - x^2)^2.$$

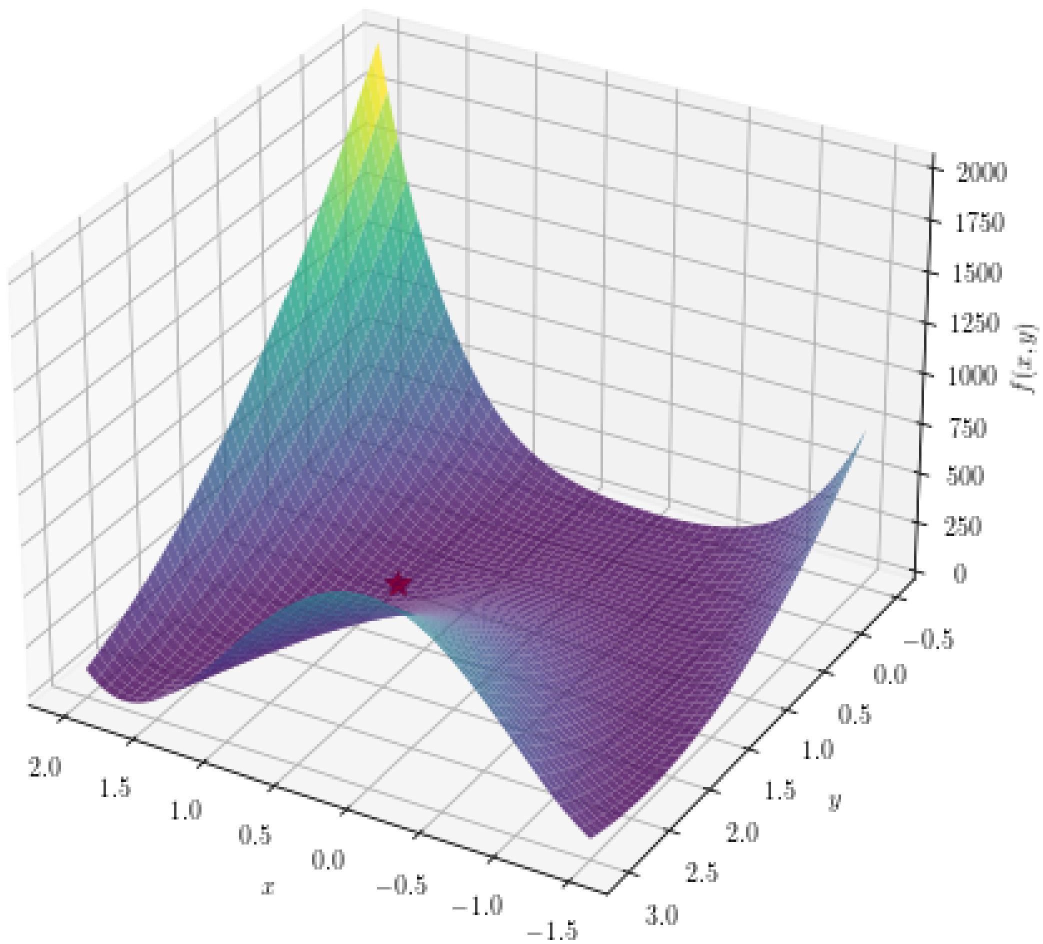
Nesta tarefa vamos minimizar a função de Rosenbrock para $a = 1$ e $b = 100$, que possui ponto de mínimo global em $(x, y) = (a, a^2) = (1, 1)$ e é dada por

[2]:
```
class Rosenbrock:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self, x, y):
        """Compute the value of the function at given point."""
        return (self.a - x) ** 2 + self.b * (y - x ** 2) ** 2

    def _gradx(self, x, y):
        """Gradient with respect to x."""
        return -2 * (self.a - x) - 4 * self.b * x * (y - x ** 2)

    def _grady(self, x, y):
        """Gradient with respect to y."""
        return 2 * self.b * (y - x ** 2)

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

    @property
    def x_min(self):
        """Return the x position of the global minimum."""
        return self.a

    @property
    def y_min(self):
        """Return the y position of the global minimum."""
        return self.a ** 2

    @cached_property
    def min(self):
        """Return the minimum value of the function."""
        return self(self.x_min, self.y_min)

    @cached_property
    def xs(self):
        """Define the x domain for plotting."""
        return np.linspace(-1.5, 2, 1000)

    @cached_property
    def ys(self):
        """Define the y domain for plotting."""
```

```
        return np.linspace(-0.5, 3, 1000)
```

Em seguida, podemos visualizar a superfície da função.

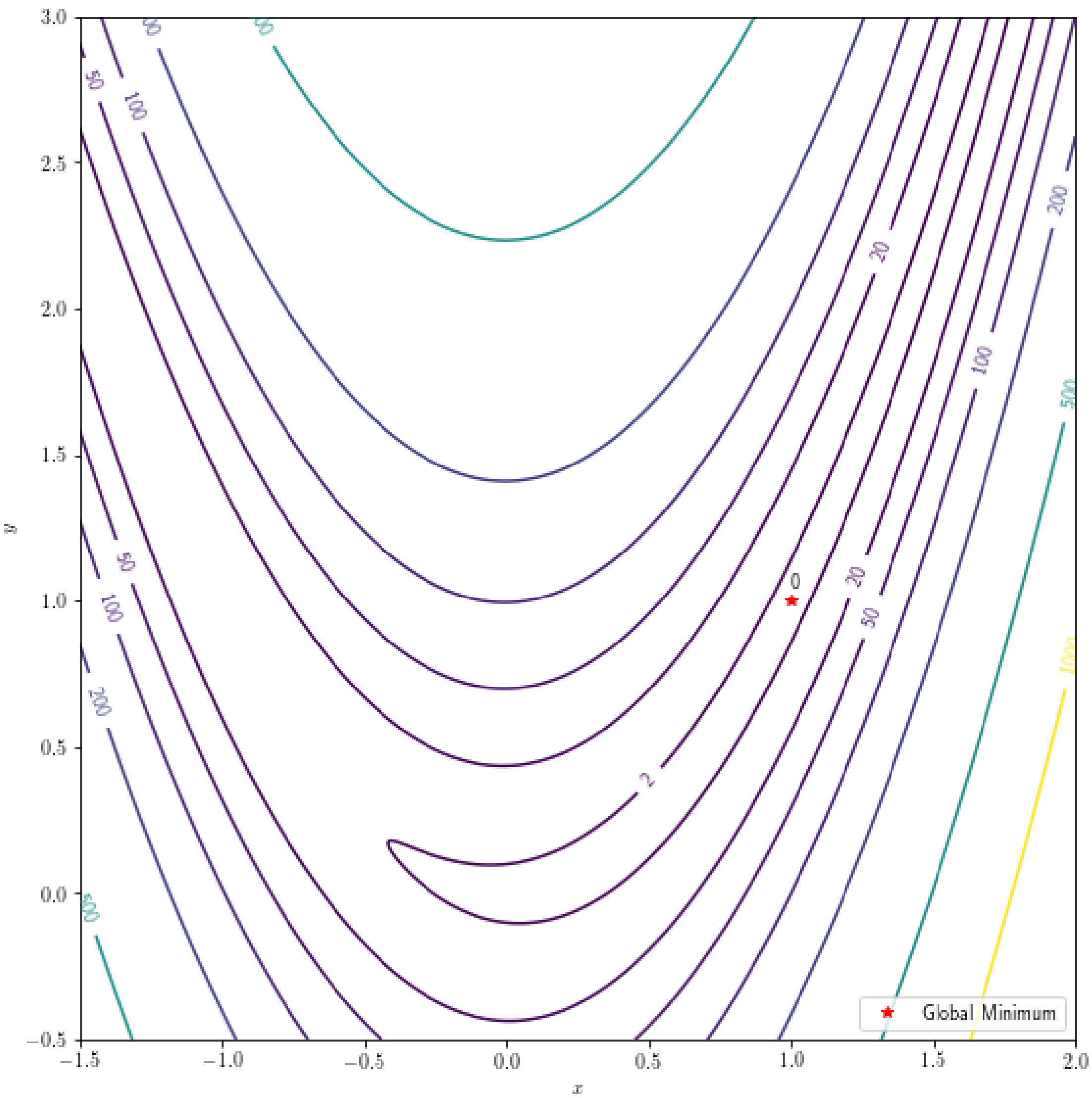
[3]:
```
def plot_surface(fig, ax, fn):
    xs, ys = np.meshgrid(fn.xs, fn.ys)
    zs = fn(xs, ys)
    surf = ax.plot_surface(xs, ys, zs, alpha=0.8, cmap="viridis")
    ax.plot(fn.x_min, fn.y_min, fn.min, "r*", markersize=10, label="Global␣
 ↪Minimum")
    ax.set_xlabel("$x$")
    ax.set_ylabel("$y$")
    ax.set_zlabel("$f(x, y)$")
    ax.view_init(azim=120)


rosenbrock = Rosenbrock(a=A, b=B)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection="3d")
plot_surface(fig, ax, rosenbrock)
fig.tight_layout()
```



Note como o gradiente da função decai rapidamente próximo do mínimo global bem como em uma grande região no formato de U em torno dele, dificultando o processo de minimização. A seguir, também plotamos as curvas de nível da função.

[4]:
```
def plot_contour(ax, fn: Rosenbrock):
    locator = ticker.IndexLocator(10, 50)
    xs, ys = np.meshgrid(fn.xs, fn.ys)
    zs = fn(xs, ys)
    cs = ax.contour(xs, ys, zs, levels=[2, 20, 50, 100, 200, 500, 1000])
    ax.clabel(cs, fmt="%d")
    ax.plot(fn.x_min, fn.y_min, "r*", label="Global Minimum")
    ax.annotate(fn.min, (fn.x_min, fn.y_min + 0.05))
    ax.set_xlabel("$x$")
    ax.set_ylabel("$y$")
    ax.legend(loc="lower right")
```

```
    return fig, ax


fig, ax = plt.subplots(figsize=(8, 8))
plot_contour(ax, rosenbrock)
fig.tight_layout()
```



3 Descida do gradiente com gradiente explícito

[5]:
```
def has_converged(fn, xy_old, xy_new, tol):
    """Verify if the function has converged according to the absolute␣
↪difference criteria."""
    return np.abs(fn(*xy_old) - fn(*xy_new)) < tol


def grad_step(fn, xy, lr):
    """Perform a gradient descent step by computing the updated values of x an␣
↪y."""
    x, y = xy
    gradx, grady = fn.grad(x, y)
    return x - lr * gradx, y - lr * grady


def grad_descent(fn, max_steps, xy_initial, lr, tol, schedule: float = 1):
    """Find the parameters that minimize `fn`.

    Args:
        max_steps: The maximum number of gradient steps before terminating.
        xy_initial: 2-tuple with the initial values of x and y.
        lr: Learning rate.
        tol: The minimum absolute difference tolerated between two successive␣
↪values of `fn`.

    Returns:
        A np.ndarray with shape (S, 2) with all of the xy positions traveled to␣
↪reach the minimum.
    """
    step = 1
    xy_old = xy_initial
    xy_new = grad_step(fn, xy_old, lr)

    xys = [xy_old, xy_new]

    try:
        while not has_converged(fn, xy_old, xy_new, tol) and step < MAX_STEPS:
            xy_old = xy_new
            xy_new = grad_step(fn, xy_old, lr)
            xys.append(xy_new)

            lr *= schedule

            step += 1

    except OverflowError:
```

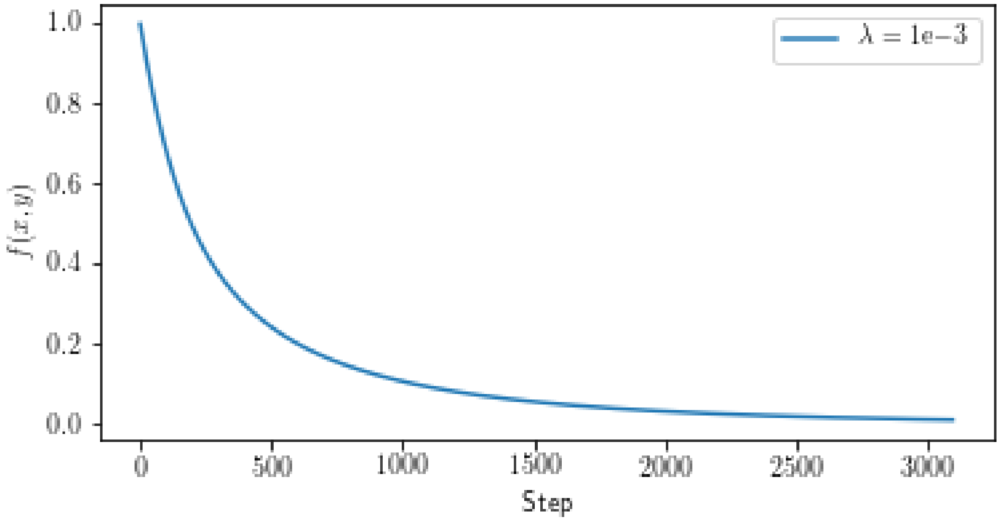
```
        print(
            "Houve um erro de Overflow (possivelmente causado pelo tamanho do␣
↪step) e o algoritmo "
            "foi interrompido"
        )

    return np.array(xys)


def inspect_grad_descent(xy_path):
    """Print information about the optimization path."""
    print(f"Num steps: {len(xy_path) - 1}")
    print(f"Final xy: {xy_path[-1][0]:.4e}, {xy_path[-1][1]:.4e}")


def plot_grad_descent_path(ax, fn, xy_path, step_diff=1):
    """Plot the optimization path in the function's surface."""
    xs, ys = xy_path[::step_diff, 0], xy_path[::step_diff, 1]
    xs = np.clip(xs, np.min(fn.xs), np.max(fn.xs))
    ys = np.clip(ys, np.min(fn.ys), np.max(fn.ys))
    zs = fn(xs, ys)
    ax.plot(xs, ys, zs, color="k")


def plot_fn(fn: Rosenbrock, xy_paths: Sequence[np.ndarray], labels:␣
↪Sequence[str]):
    """Plot the function values per time step."""
    fig, ax = plt.subplots(figsize=(6, 3))
    for xy_path, label in zip(xy_paths, labels):
        zs = fn(xy_path[:, 0], xy_path[:, 1])
        ax.plot(zs, label=label)
    ax.set_ylabel("$f(x, y)$")
    ax.set_xlabel("Step")
    ax.legend()
```

3.1 Learning rate $\lambda = 1\mathrm{e}{-3}$

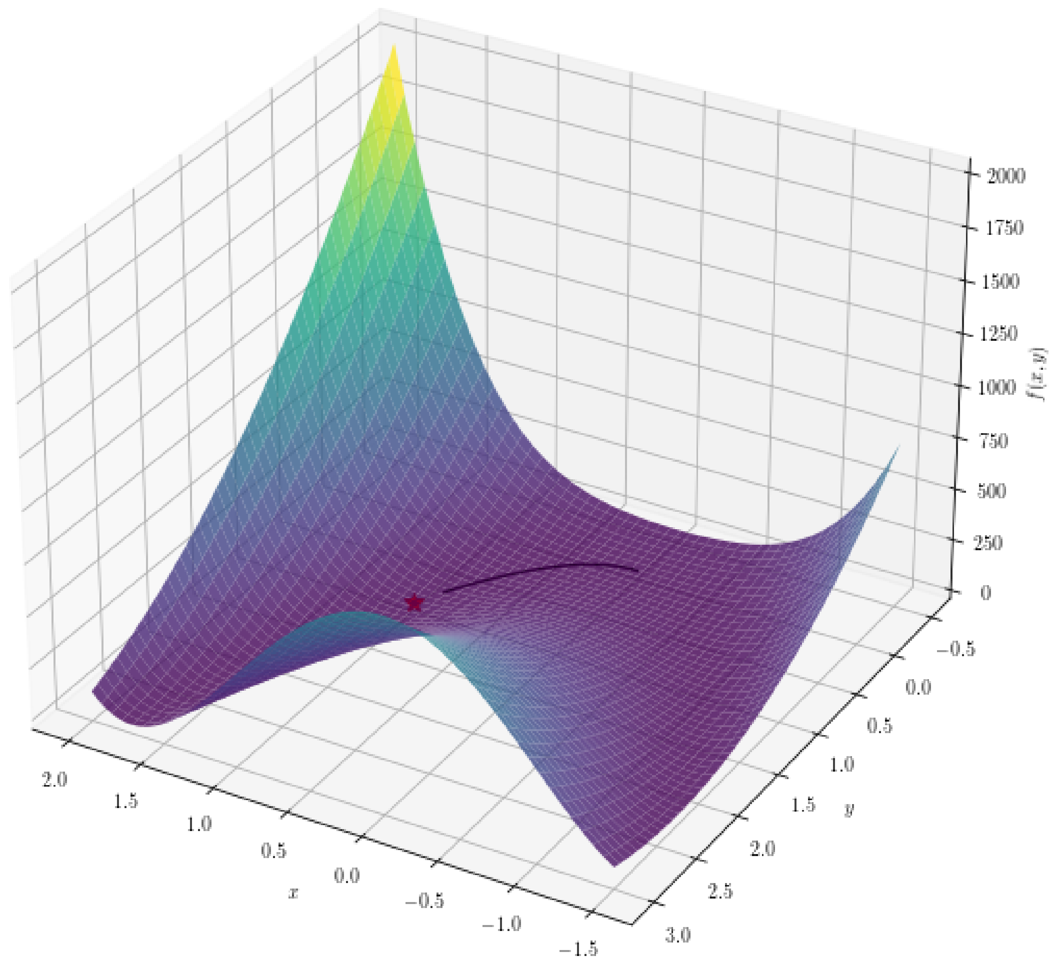
Para um $\lambda = 1\mathrm{e}{-3}$, o otimizador converge de forma bem suave até satisfazer o critério de parada da diferença de valore successivos de $f(x, y)$ abaixo da tolerância.

[6]:
```
xy_path = grad_descent(rosenbrock, max_steps=MAX_STEPS, xy_initial=XY_INITIAL,␣
↪tol=TOL, lr=1e-3)
inspect_grad_descent(xy_path)
plot_fn(rosenbrock, [xy_path], ["$\lambda = 1\mathrm{e}{-3}$"])
```

```
Num steps: 3096
Final xy: 8.9737e-01, 8.0483e-01
```



[7]:
```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection="3d")
plot_surface(fig, ax, rosenbrock)
plot_grad_descent_path(ax, rosenbrock, xy_path, step_diff=1)
fig.tight_layout()
```

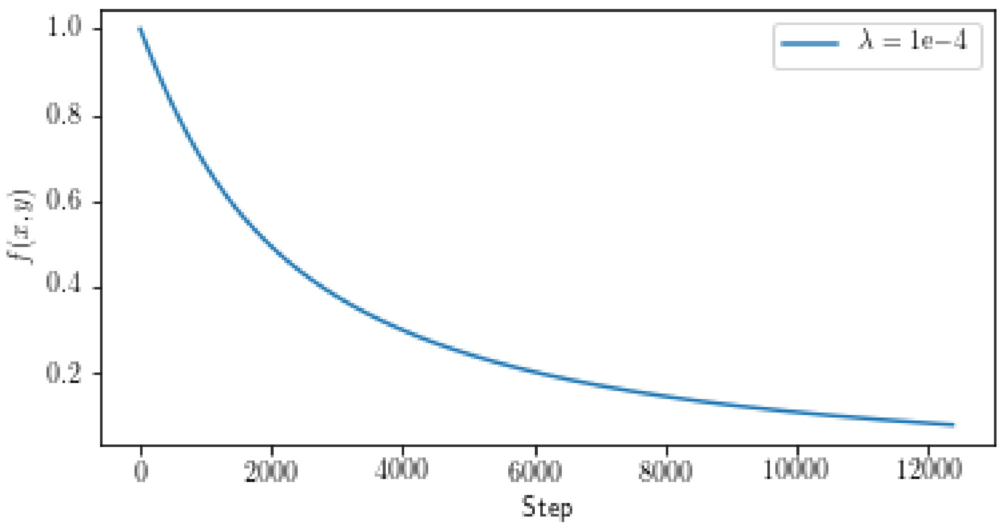


3.2 Learning rate $\lambda = 1\mathrm{e}{-4}$

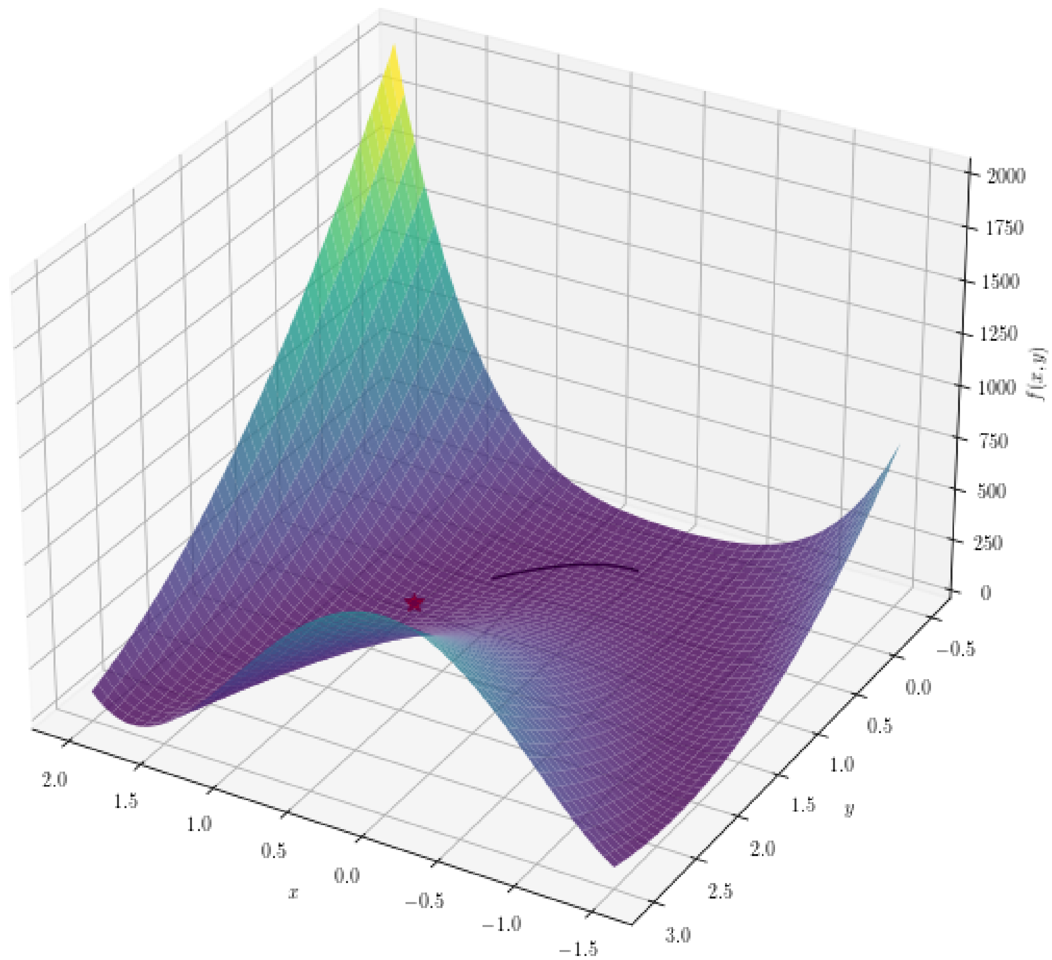
Usando um learning rate menor o otimizador aproxima-se mais lentamente do mínimo global. Devido à convergência mais lenta, o critério de parada é satisfeito para valores de x e y mais distantes dos valores ótimos que para $\lambda = 1\mathrm{e}{-3}$.

[8]:
```
xy_path = grad_descent(rosenbrock, max_steps=MAX_STEPS, xy_initial=XY_INITIAL,␣
 ↪tol=TOL, lr=1e-4)
inspect_grad_descent(xy_path)
plot_fn(rosenbrock, [xy_path], ["$\lambda = 1\mathrm{e}{-4}$"])
```

```
Num steps: 12384
Final xy: 7.2225e-01, 5.2035e-01
```



[9]:
```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection="3d")
plot_surface(fig, ax, rosenbrock)
plot_grad_descent_path(ax, rosenbrock, xy_path, step_diff=100)
fig.tight_layout()
```



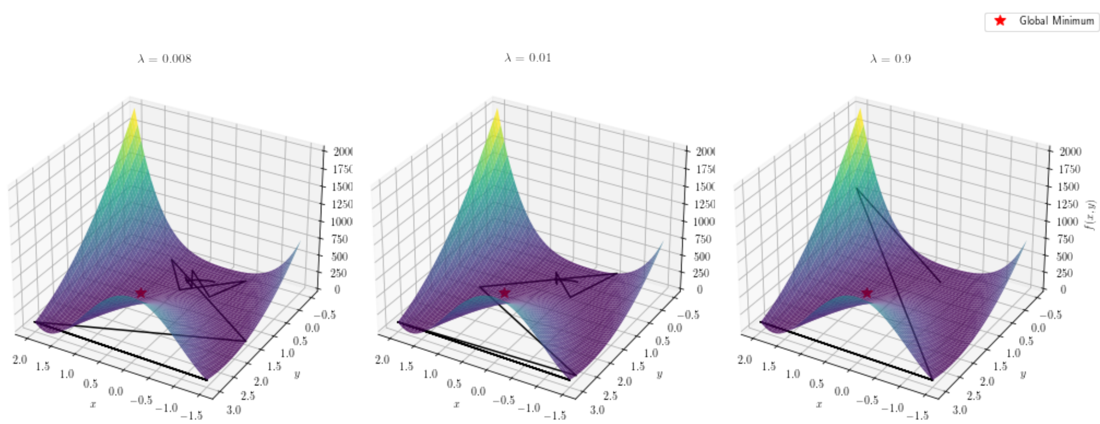
3.3 Learning rates altos

Para os learning rates altos, utilizamos $\lambda \in \{8\text{e}{-3}, 1\text{e}{-2}, 9\text{e}{-1}\}$

[10]:
```
high_lr = [8e-3, 1e-2, 9e-1]

fig = plt.figure(figsize=(14, 6))

for idx, lr in enumerate(high_lr):
    print("Learning rate:", lr)

    xy_path = grad_descent(rosenbrock, max_steps=MAX_STEPS,␣
 ↪xy_initial=XY_INITIAL, tol=TOL, lr=lr)
    inspect_grad_descent(xy_path)
```

```python
    ax = fig.add_subplot(1, len(high_lr), idx + 1, projection="3d")
    plot_surface(fig, ax, rosenbrock)
    plot_grad_descent_path(ax, rosenbrock, xy_path, step_diff=1)
    ax.set_title(f"$\lambda = {lr}$")

    print()

handles, labels = ax.get_legend_handles_labels()
fig.legend(handles, labels)
fig.tight_layout()
```

```
Learning rate: 0.008
Houve um erro de Overflow (possivelmente causado pelo tamanho do step) e o
algoritmo foi interrompido
Num steps: 66
Final xy: 1.0101e+133, 3.4429e+88

Learning rate: 0.01
Houve um erro de Overflow (possivelmente causado pelo tamanho do step) e o
algoritmo foi interrompido
Num steps: 40
Final xy: 1.4533e+178, 4.7268e+118

Learning rate: 0.9
Houve um erro de Overflow (possivelmente causado pelo tamanho do step) e o
algoritmo foi interrompido
Num steps: 5
Final xy: 8.4459e+122, 3.1781e+82
```

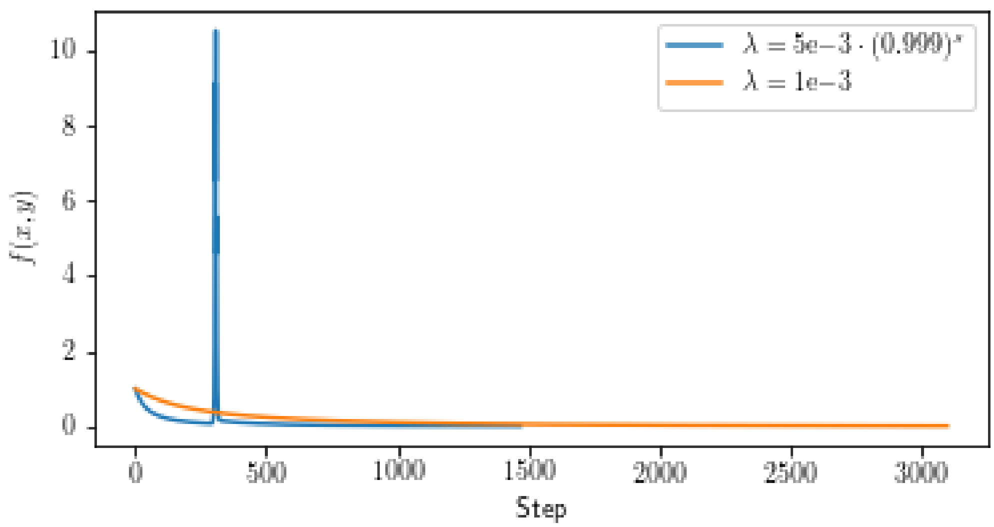


3.4 Política de redução do Learning Rate

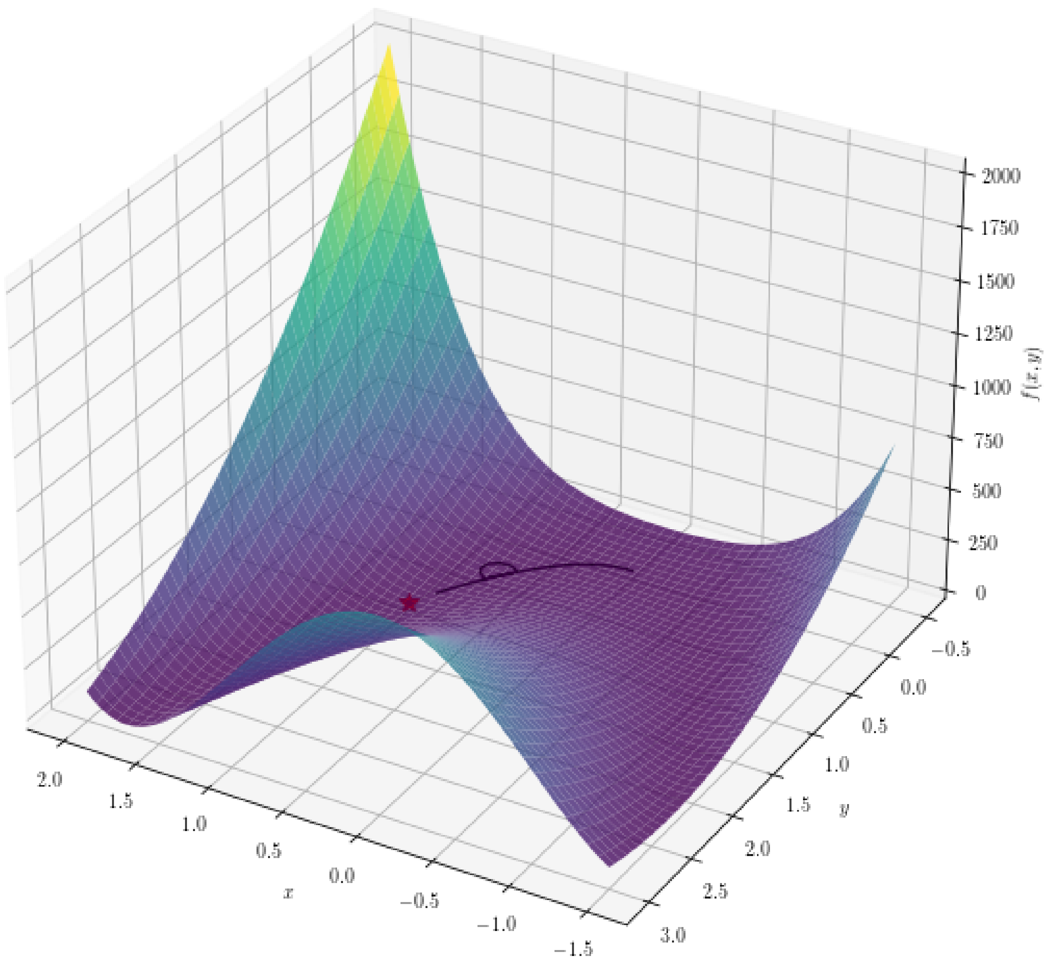
Usando a política de redução de learning rate notamos que apesar da maior instabilidade, foi possível chegar em um mínimo mais próximo do mínimo global. Além disso, o otimizador convergiu em cerca de metade dos passos que o otimizador sem a política de redução de learning rate.

[11]:
```
xy_path = grad_descent(rosenbrock, max_steps=MAX_STEPS, xy_initial=XY_INITIAL,␣
 ↪tol=TOL, lr=1e-3)
xy_path_lr = grad_descent(
    rosenbrock, max_steps=MAX_STEPS, xy_initial=XY_INITIAL, tol=TOL, lr=5e-3,␣
 ↪schedule=0.999
)
inspect_grad_descent(xy_path_lr)
plot_fn(
    rosenbrock,
    [xy_path_lr, xy_path],
    ["$\lambda = 5\mathrm{e}{-3} \cdot (0.999)^s$", "$\lambda =␣
 ↪1\mathrm{e}{-3}$"],
)
```

```
Num steps: 1468
Final xy: 9.0391e-01, 8.1665e-01
```



[12]:
```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection="3d")
plot_surface(fig, ax, rosenbrock)
plot_grad_descent_path(ax, rosenbrock, xy_path_lr, step_diff=2)
fig.tight_layout()
```



4 Derivação automática

4.1 Implementação

A seguir, nós sobreescrevemos o método `grad` da função Rosenbrock implementada anteriormente e implementamos o cálculo das derivadas no ponto utilizando o pacote PyTorch

[13]:
```
class RosenbrockAutoGrad(Rosenbrock):
    """Reimplementation of the Rosenbrock function computing the gradient using␣
↪PyTorch autograd."""

    def grad(self, x: float, y: float) -> Tuple[float, float]:
        """Computes the gradient through PyTorch.Autograd.
```

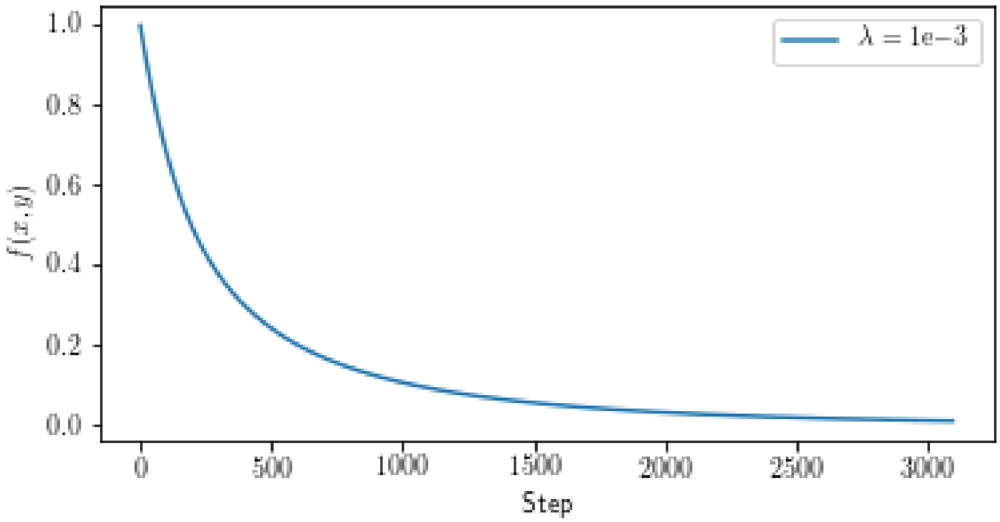
```
        Args:
            x: The point on x axis where the gradient will be evaluated.
            y: The point on y axis where the gradient will be evaluated.

        Returns:
            A tuple containing the gradient on x and y axis, respectively.
        """
        # First, we need to convert the original values into tensors
        # This way PyTorch can record every operation made to them in order to␣
↪assemble the
        # computational graph
        # Also, we explicitly ask PyTorch to record the gradient during forward/
↪backward passess
        x = torch.tensor(x, requires_grad=True)
        y = torch.tensor(y, requires_grad=True)

        # In torch language, we make a forward pass (that is, we evaluate the␣
↪function at the given
        # point while assembling the computational graph) and a backward pass,␣
↪computing the
        # gradient for the input tensors
        forward: torch.tensor = self(x, y)
        forward.backward()

        # Note that the grads are also tensors, so we use item() in order to␣
↪retrieve the value as
        # an scalar
        return x.grad.item(), y.grad.item()
```

4.2 Experimentos

Como observado anteriormente, o algoritmo de descida do gradiente obtém os melhores resultados (isto é, se aproxima mais do ponto ótimo computado analitcamente) quando $\lambda = 1\mathrm{e}{-3}$. Note como usando o gradiente automático computado pelo PyTorch o otimizador convergiu ao mesmo ponto e no mesmo número de passos que com o gradiente analítico.

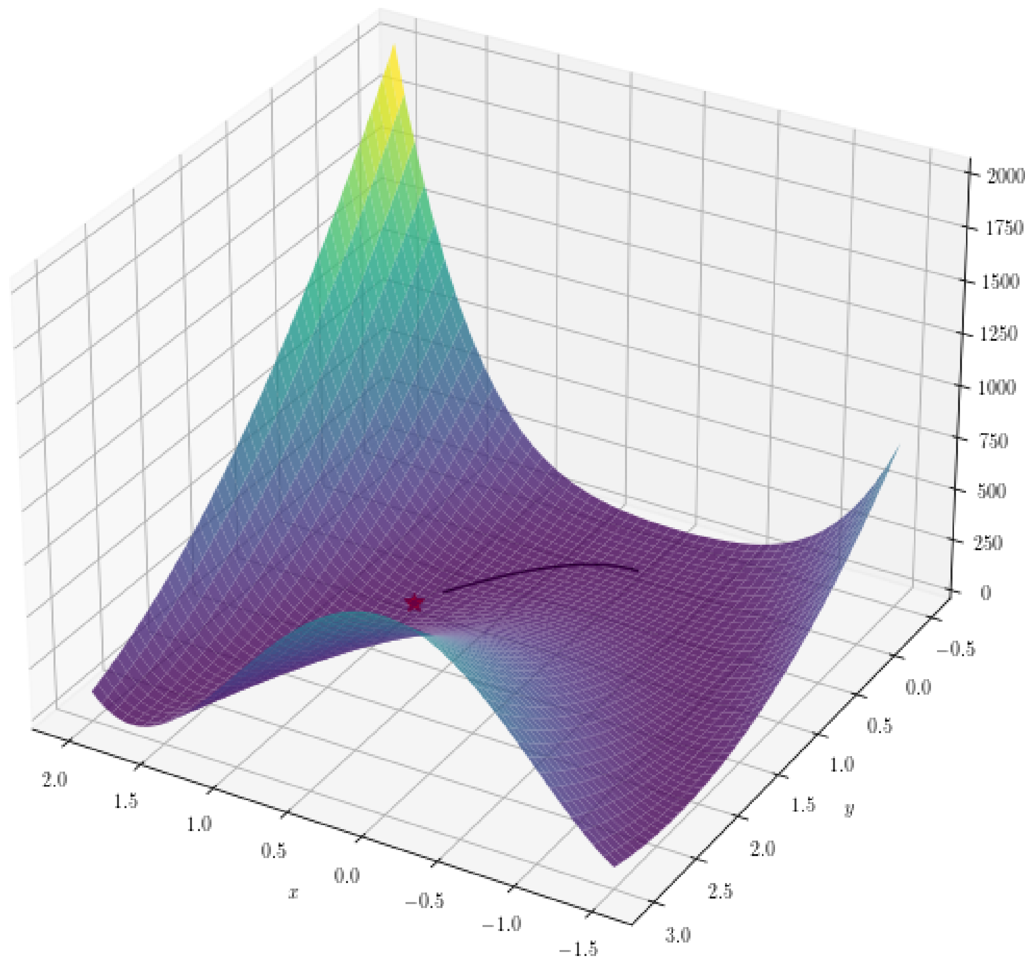
[14]:
```
rosenbrock_autograd = RosenbrockAutoGrad(a=A, b=B)

xy_path = grad_descent(
    rosenbrock_autograd,
    max_steps=MAX_STEPS,
    xy_initial=XY_INITIAL,
    tol=TOL,
    lr=1e-3,
)
inspect_grad_descent(xy_path)
plot_fn(rosenbrock_autograd, [xy_path], ["$\lambda = 1\mathrm{e}{-3}$"])
```

```
Num steps: 3096
Final xy: 8.9737e-01, 8.0483e-01
```



[15]:

```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection="3d")
plot_surface(fig, ax, rosenbrock)
plot_grad_descent_path(ax, rosenbrock, xy_path, step_diff=2)
fig.tight_layout()
```



2000
1750
1500
1250
1000
750
500
250
0
$f(x,y)$
2.0
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
x
-0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
y