Regressão Logística nos dados covtype de LIBSVM

Iremos treinar os dados para classificação binária usando os dados `covtype` da biblioteca LIBSVM. Veja também Kaggle-LIBSVM.

Queremos minimizar

$$\min_{w\in\mathbb{R}^{d\times 1}} f(w) = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i\log(s(x_i^\top w)) + (1-y_i)\log(1-s(x_i^\top w))\right) + \frac{\gamma}{2}\|w\|^2,$$

onde $x_i \in \mathbb{R}^{d\times 1}$, $y_i \in \{0,1\}$, $s(z) = \frac{1}{1+\exp(-z)}$ é a função sigmóide. O gradiente é dado por

$$\nabla f(w) = \frac{1}{n}\sum_{i=1}^{n} x_i(s(x_i^\top w) - y_i) + \gamma w.$$

Esta é uma função suave com constante de Lipschitz $L = L_0 + \gamma$ onde

$$L_0 = \frac{1}{4}\lambda_{\max}\left(\frac{X^\top X}{n}\right),$$

onde $X \in \mathbb{R}^{n\times d}$ é a matriz de dados cuja i-ézima linha é o vetor $x_i^\top$ e $\lambda_{\max}(\cdot)$ denota o maior autovalor.

In [1]:
```
# Importação de módulos necesserários:

import matplotlib
import numpy as np
import scipy
import seaborn as sns
import matplotlib.pyplot as plt
import numpy.linalg as la
from sklearn.datasets import load_svmlight_file
from sklearn.utils.extmath import safe_sparse_dot
```

In [2]:
```
# Lendo os dados do arquivo covtype e guardando na matriz de dados X e vetor de labels y:

data = load_svmlight_file('./datasets/covtype.bz2')
X, y = data[0].toarray(), data[1]
if (np.unique(y) == [1, 2]).all():
    # Devemos garantir que os labels estão em {0, 1}
    y -= 1
```

```python
n, d = X.shape # tamanho da amostra, número de features
```

Exercício 1: Funções auxiliares

1. Construa uma função `smoothness(X)` que toma a matriz de dados `X` e retorna a constante L_0.
2. Construa uma função `f(w, X, y, l2)` que toma o iterado `w`, os dados `X, y` e a penalização γ (= `l2`) e retorna o valor funcional $f(w)$.
3. Construa uma função `df(w, X, y, l2)` toma o iterado `w`, os dados `X, y` e a penalização γ (= `l2`) e retorna o gradiente $\nabla f(w)$.

In [4]:
```python
#Escreva código aqui
def smoothness(X:np.ndarray) -> float:
    n = X.shape[0]
    XtX = (X.T @ X) / n
    lambda_max = la.eigvalsh(XtX).max()
    L0 = lambda_max / 4
    return L0

def sig(z:np.ndarray) -> float:
    return 1/(1+ np.exp(-z))

def f(w:np.ndarray, X:np.ndarray, y:np.ndarray, l2:float) -> float:
    z = X @ w
    s = sig(z)
    log_loss = - np.mean(y * np.log(s) + (1 - y) * np.log(1 - s))
    reg_term = (l2 / 2) * np.linalg.norm(w)**2
    return log_loss + reg_term

def df(w:np.ndarray, X:np.ndarray, y:np.ndarray, l2:float) -> np.ndarray:
    n = X.shape[0]
    z = X @ w
    s = sig(z)
    grad = (1 / n) * X.T @ (s - y) + l2 * w
    return grad
```

Inicialização

Fixaremos:

In [5]:
```python
L0 = smoothness(X)
l2 = L0 / n
```

```
L = L0 + l2
w0 = np.zeros(d)         # ponto inicial
it_max = 10000           # número de iterações

#Função de desempenho é a norma de Frobenius:
J = lambda w: la.norm(w)

#Função que retorna gradiente (para X,y,l2) já declarados.
def Df(w):
    return df(w, X, y, l2)

l2
```

Out[5]:
```
array(8.67636767)
```

Exercício 2: Método do gradiente

Construa uma função `gd(df, w0, la=1, numb_iter=100)` que toma como entrada as funções `J()` `df()`, o ponto inicial `w0`, o passo `la` e o número de iterações `numb_iter` e implementa o método gradiente iniciando de `w0`. Esta função deve retornar a sequência de valores da função $\|\nabla f(w)\|_2$ em cada um dos iterados w. A função também deve retornar o último iterado.

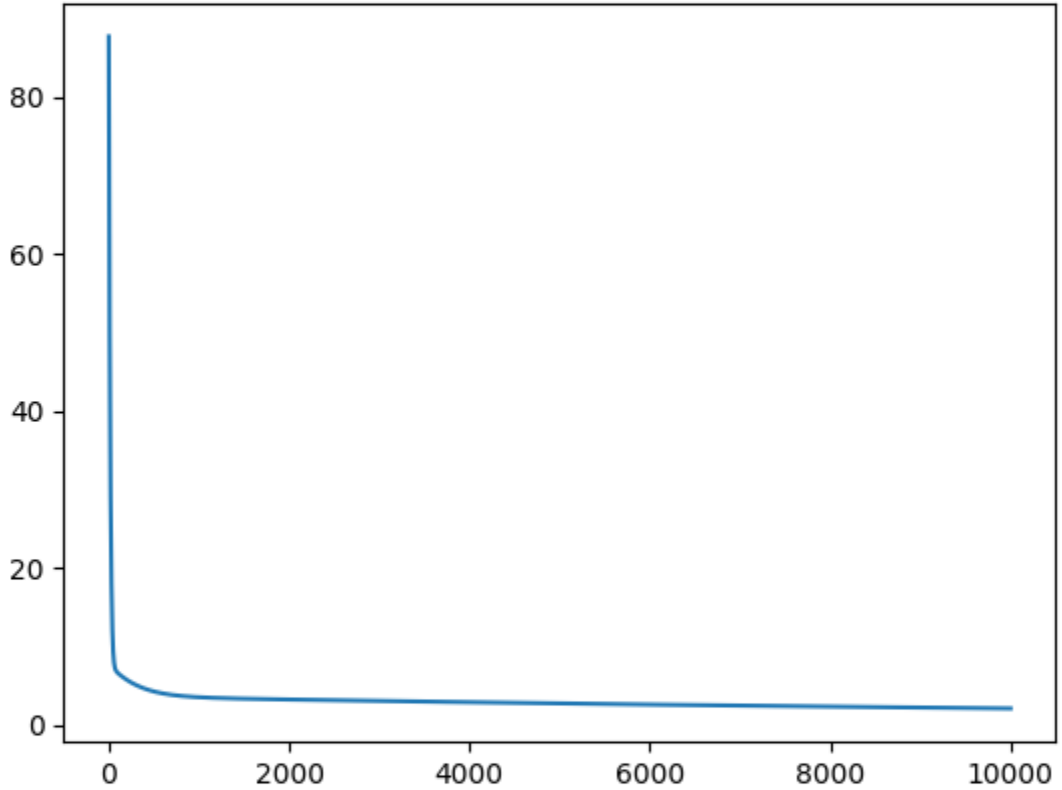
Implemente a função com passo `la=1./L`

In [6]:
```
#Escreva código aqui
def gd(J, df, w0:np.ndarray, la:float=1, numb_iter:int=100) -> tuple[list[float], np.ndarray]:
    w = w0.copy()
    grad_norms = list()

    for i in range(numb_iter):
        grad = df(w)
        grad_norm = J(grad)
        grad_norms.append(float(grad_norm))

        w -= la * grad

    return grad_norms, w
```

In [7]:
```
# gradient descent
f1 = gd(J, Df, w0, la=1./L, numb_iter=it_max)
```

In [8]:
```
plt.plot(f1[0])
```

Out[8]: [<matplotlib.lines.Line2D at 0x77672a1084d0>]



Exercício 3: Método do gradiente acelerado

Construa uma função `accel_gd(J, df, w0, la=1, numb_iter=100)` que toma como entrada as funções `J()`, `df()`, o ponto inicial `w0`, o passo `la` e o número de iterações `numb_iter` e implementa o método gradiente com aceleração de Nesterov iniciando de `w0` e $t_0 = 1$:

$$
\begin{aligned}
w_{k+1} &:= y_k - la\nabla f(y_k),\\
t_{k+1} &:= \frac{1+\sqrt{1+4t_k^2}}{2},\\
y_{k+1} &:= w_{k+1} + \frac{t_k - 1}{t_{k+1}}(w_{k+1} - w_k).
\end{aligned}
$$

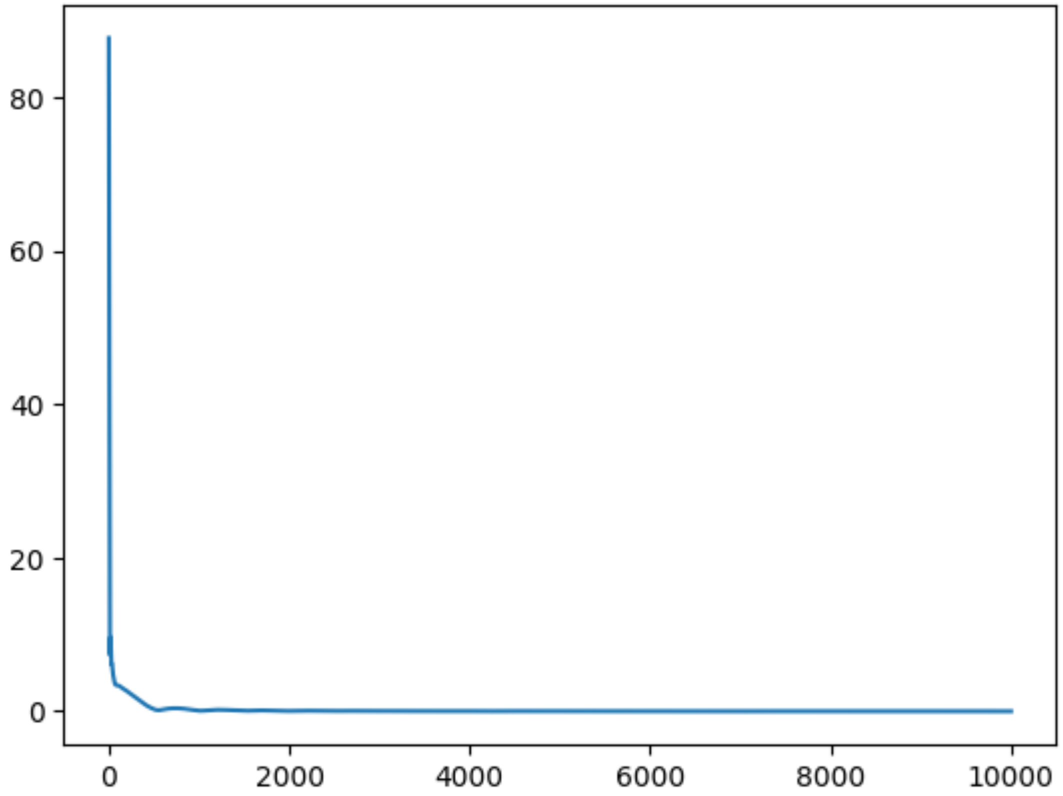
Esta função deve retornar a sequência de valores da função `J(df(y))` em cada um dos iterados `y`, isto é, a sequência das normas dos gradientes de y_k ao longo da trajetória do método. A função também deve retornar o último iterado.

Implemente a função com passo `la=1./L`.

In [9]:
```python
#Escreva código aqui
def accel_gd(J, df, w0:np.ndarray, la:float=1, numb_iter:int=100) -> tuple[list[float], np.ndarray]:
    grad_norms = list()
    w = w0.copy()
    y = w0.copy()
    t = 1

    for k in range(numb_iter):
        grad = df(y)
        grad_norms.append(float(J(grad)))

        w_next = y - la * grad

        t_next = (1 + np.sqrt(1 + 4 * t**2)) / 2

        y = w_next + ((t - 1) / t_next) * (w_next - w)

        w, t = w_next, t_next

    return grad_norms, w
```

In [10]:
```python
# Nesterov accelerated gradient descent
f2 = accel_gd(J, Df, w0, la=1./L, numb_iter=it_max)
plt.plot(f2[0])
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x776727b46190>]
```



Exercício 4: Método do gradiente acelerado com γ

Construa uma função `accel_gd_sc(J, df, w0, la=1, kappa, numb_iter=100)` que toma como entrada as funções `J()`, `df()`, o ponto inicial `w0`, o passo `la` e o número de iterações `numb_iter` e implementa o método gradiente com aceleração de Nesterov iniciando de `y0=w0`:

$$
\begin{aligned}
w_{k+1} &:= y_k - la\nabla f(y_k),\\
y_{k+1} &:= w_{k+1} + \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}(w_{k+1} - w_k).
\end{aligned}
$$

Àcima, $\kappa := L/\gamma$. Esta função deve retornar a sequência de valores da função `J(df(y))` em cada um dos iterados `y`, isto é, a sequência das normas dos gradientes de y_k ao longo da trajetória do método. A função também deve retornar o último iterado.

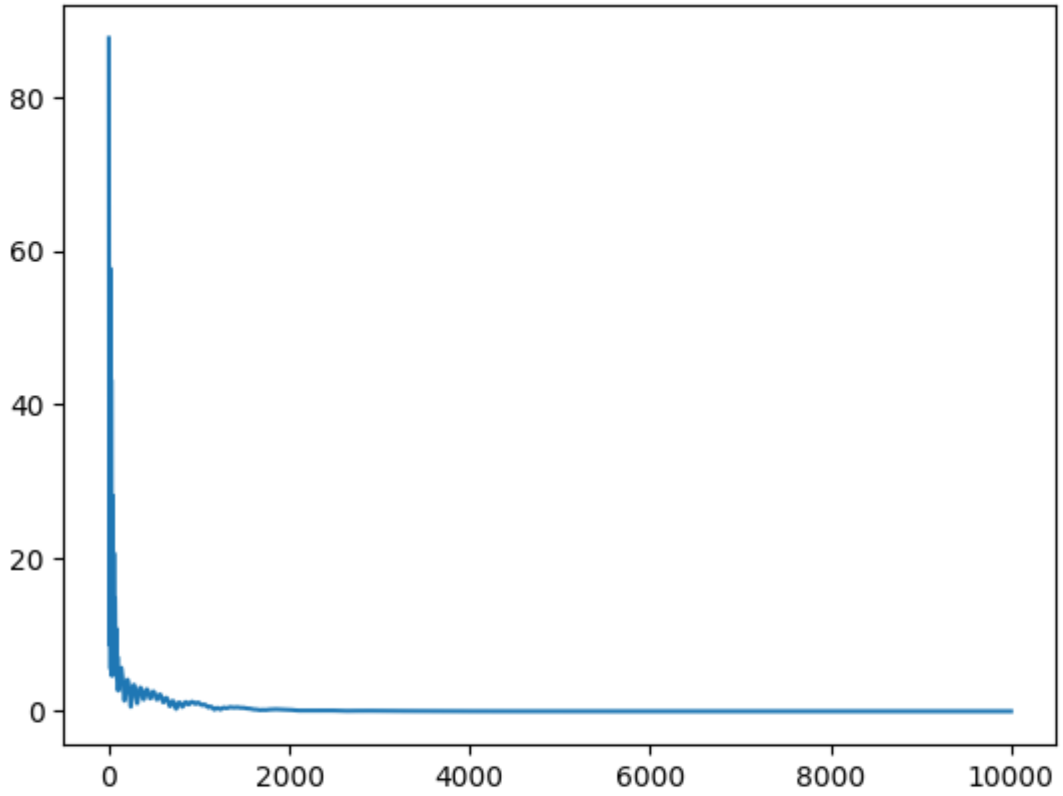
Implemente a função com passo `la=1./L`.

In [11]:
```
#Escreva código aqui
def accel_gd_sc(J, df, w0:np.ndarray, la:float, kappa, numb_iter:int=100) -> tuple[list[float], np.ndarray]:
```

```python
    grad_norms = list()
    w = w0.copy()
    y = w0.copy()
    theta = (np.sqrt(kappa) - 1) / (np.sqrt(kappa) + 1)

    for k in range(numb_iter):
        grad = df(y)
        grad_norms.append(float(J(grad)))

        w_new = y - la * grad

        y = w_new + theta * (w_new - w)

        w = w_new

    return grad_norms, y
```

In [12]:
```python
# Nesterov accelerated gradient descent for strongly convex functions
f3 = accel_gd_sc(J, Df, w0, la=1./L, kappa=L/l2, numb_iter=it_max)
plt.plot(f3[0])
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x776727a217d0>]
```



Exercício 5: Adagrad-Norm

Construa uma função `ad_grad_norm(J, Df, w0, b0=0.5, eta=1, numb_iter=100)` que toma como entrada as funções `J()`, `Df`, o ponto inicial `w0`, parametros positivos `b0` e `eta` e o número de iterações `numb_iter` e implementa o método Adagrad-Norm iniciando de `w0`:

$$w_{k+1} := w_k - \frac{\eta}{\sqrt{b_0^2 + \sum_{j=1}^{k} \|\nabla f(w_j)\|_2^2}} \nabla f(w_k).$$

Esta função deve retornar a sequência de valores da função `J(Df(w))` em cada um dos iterados `w`, isto é, a sequência das normas dos gradientes de X_k ao longo da trajetória do método. A função também deve retornar o último iterado.

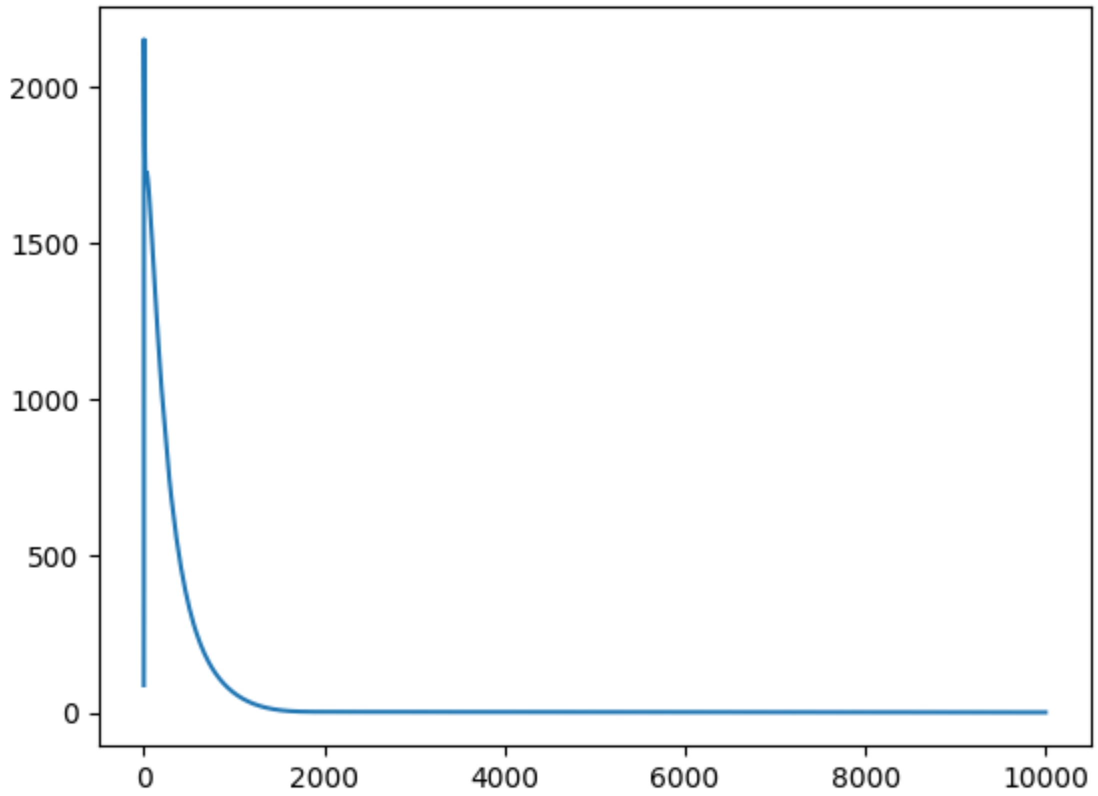
Implemente a função com `b0=0.5` e `eta=1`.

In [13]:
```
#Escreva código aqui
def ad_grad_norm(J, Df, w0:np.ndarray, b0:float=0.5, eta:float=1, numb_iter:int=100) -> tuple[list[float], np.ndarray]:
    grad_norms = list()
    w = w0.copy()
```

```
    sum_grad_sq = 0

    for k in range(numb_iter):
        grad = Df(w)
        grad_norm = J(grad)
        grad_norms.append(float(grad_norm))

        sum_grad_sq += grad_norm**2
        step_size = eta / np.sqrt(b0**2 + sum_grad_sq)

        w -= step_size * grad

    return grad_norms, w
```

In [14]:
```
# Adagrad-Norm
f4 = ad_grad_norm(J, Df, w0, b0=0.5, eta=0.01, numb_iter=it_max)
plt.plot(f4[0])
```

Out[14]: [<matplotlib.lines.Line2D at 0x776727a673d0>]



Exercício 6: Adam

Construa uma função `adam(Df, w0, alpha, beta1, beta2, epsilon, numb_iter)` que toma como entrada as funções `Df`, o ponto inicial `w0`, parametros positivos `alpha`, `beta1`, `beta2`, `epsilon` e o número de iterações `numb_iter` e implementa o método Adam iniciando de `w0`, $m_0 = 0$, $v_0 = 0$ e $k = 0$: para cada $j$ézima coordenada:

$$
\begin{aligned}
m_{k+1}[j] &:= \beta_1 \cdot m_k[j] + (1-\beta_1)\cdot \nabla f(w_k)[j],\\
v_{k+1}[j] &:= \beta_2 \cdot v_k[j] + (1-\beta_2)\cdot (\nabla f(w_k)[j])^2,\\
\hat{m}_{k+1}[j] &:= \frac{1}{1-\beta_1^{k+1}} m_{k+1}[j],\\
\hat{v}_{k+1}[j] &:= \frac{1}{1-\beta_2^{k+1}} v_{k+1}[j],\\
w_{k+1}[j] &:= w_k[j] - \frac{\alpha}{\sqrt{\hat{v}_{k+1}[j]}+\epsilon}\hat{m}_{k+1}[j].
\end{aligned}
$$

Esta função deve retornar a sequência de valores da função `J(Df(w))` em cada um dos iterados `w`, isto é, a sequência das normas dos gradientes de w_k ao longo da trajetória do método. A função também deve retornar o último iterado.

Implemente a função com `alpha=0.001`, `beta1=0.9`, `beta2=0.999`, `epsilon=10**(-8)`.

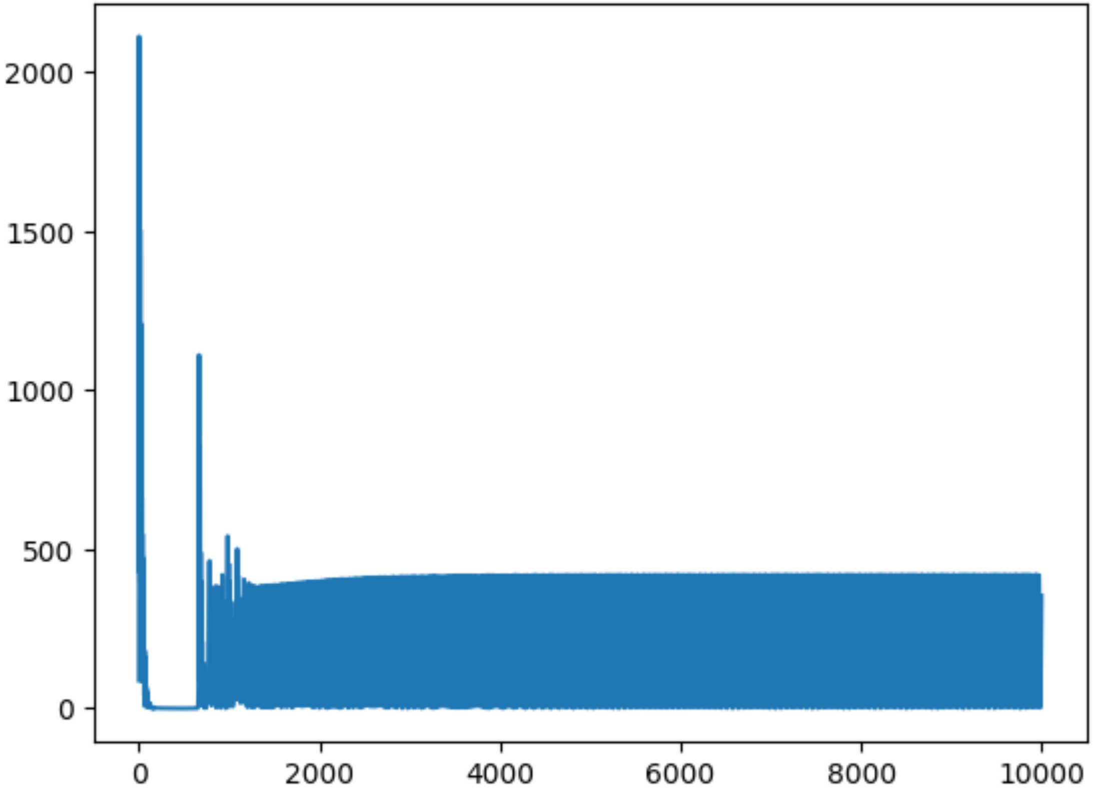
In [15]:

```
def adam(Df, w0: np.ndarray, alpha: float = 0.001, beta1: float = 0.9, beta2: float = 0.999, epsilon: float = 1e-8, numb_iter:
    grad_norms = list()
    w = w0.copy()
    m = np.zeros_like(w)
    v = np.zeros_like(w)

    for k in range(1, numb_iter + 1):
        grad = Df(w)
        grad_norm = J(grad)
        grad_norms.append(float(grad_norm))

        m = beta1 * m + (1 - beta1) * grad
        v = beta2 * v + (1 - beta2) * (grad ** 2)

        m_hat = m / (1 - beta1 ** k)
        v_hat = v / (1 - beta2 ** k)

        w -= alpha * m_hat / (np.sqrt(v_hat) + epsilon)
```

```
    return grad_norms, w
```

In [16]:
```
# Adam
f5 = adam(Df, w0, alpha=0.001, beta1=0.9, beta2=0.999, epsilon=10**(-8), numb_iter=it_max)
plt.plot(f5[0])
```

Out[16]: [<matplotlib.lines.Line2D at 0x776727af73d0>]

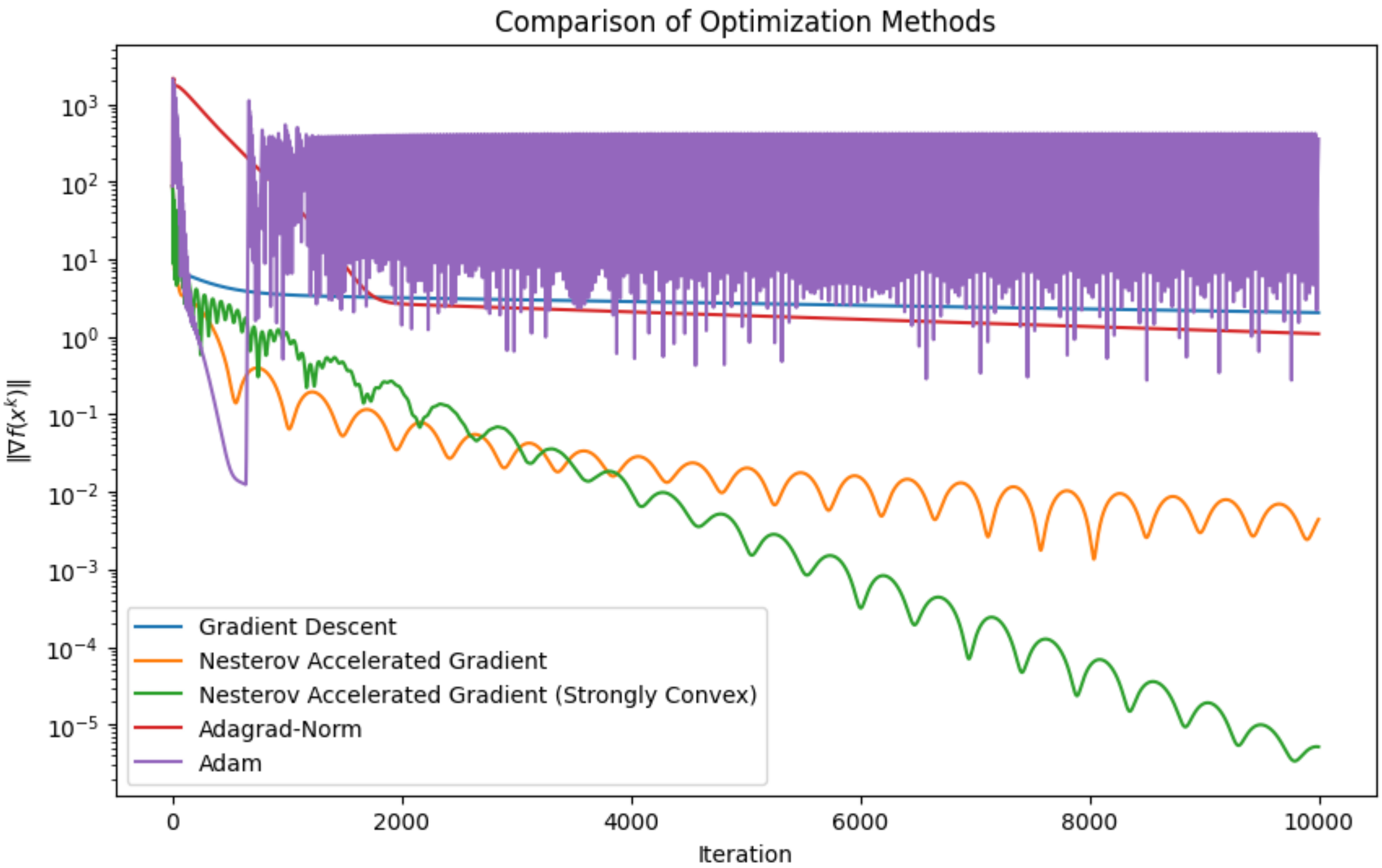


Exercício 7:

Implemente num mesmo gráfico os erros $\|\nabla f(w_k)\|$ de cada método em função no número de iterações.

In [17]:
```
#Escreva código aqui
plt.figure(figsize=(10, 6))
plt.plot(f1[0], label='Gradient Descent')
plt.plot(f2[0], label='Nesterov Accelerated Gradient')
plt.plot(f3[0], label='Nesterov Accelerated Gradient (Strongly Convex)')
plt.plot(f4[0], label='Adagrad-Norm')
plt.plot(f5[0], label='Adam')
```

```python
plt.xlabel('Iteration')
plt.yscale('log')
plt.ylabel(r'$\Vert\nabla f(x^k)\Vert$')
plt.legend()
plt.title('Comparison of Optimization Methods')
plt.show()
```

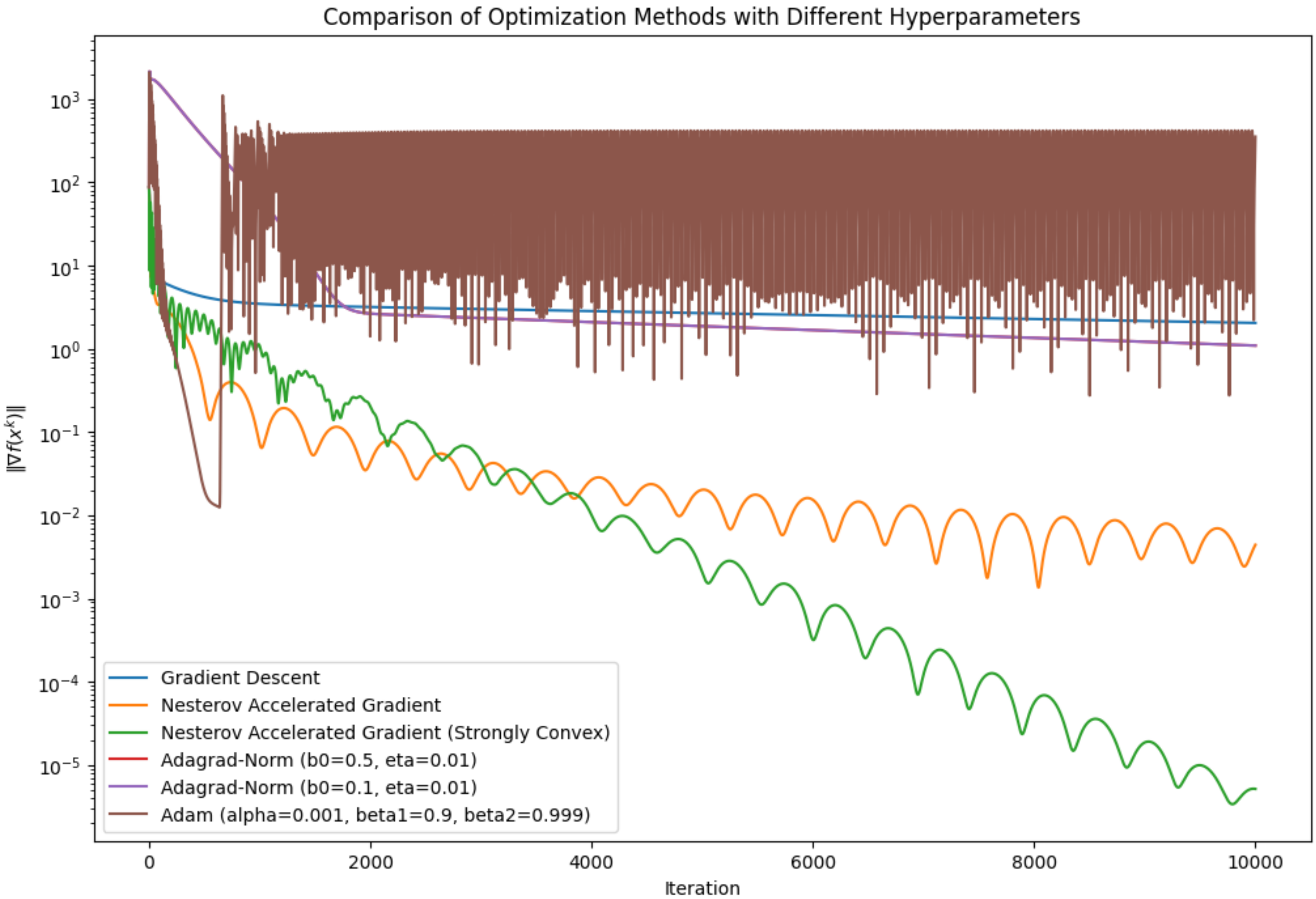


Exercício 8:

Experimente com os hyper-parâmetros de Adagrad-Norm and Adam para ver se eles podem chegar perto ou superar a performance de GD e Nesterov. Plote o gráfico como no Exercício 7.

In [18]:
```
# Experimenting with hyperparameters for Adagrad-Norm and Adam

f4_1 = ad_grad_norm(J, Df, w0, b0=0.1, eta=0.01, numb_iter=it_max)
f4_2 = ad_grad_norm(J, Df, w0, b0=1.0, eta=0.01, numb_iter=it_max)
f4_3 = ad_grad_norm(J, Df, w0, b0=0.5, eta=0.1, numb_iter=it_max)

f5_1 = adam(Df, w0, alpha=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8, numb_iter=it_max)
f5_2 = adam(Df, w0, alpha=0.01, beta1=0.9, beta2=0.999, epsilon=1e-8, numb_iter=it_max)
f5_3 = adam(Df, w0, alpha=0.001, beta1=0.95, beta2=0.999, epsilon=1e-8, numb_iter=it_max)
```

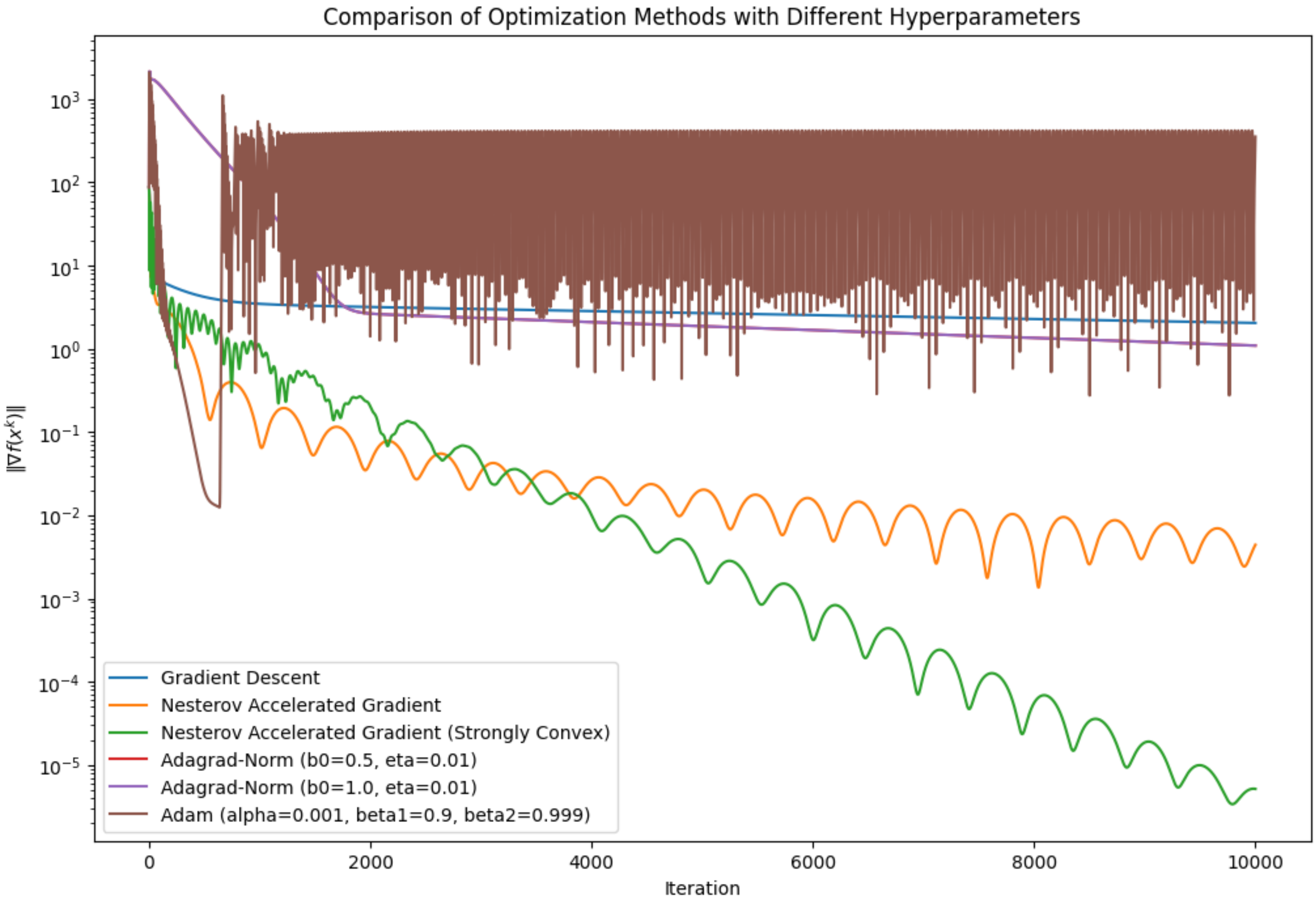
In [22]:
```
plt.figure(figsize=(12, 8))
plt.plot(f1[0], label='Gradient Descent')
plt.plot(f2[0], label='Nesterov Accelerated Gradient')
plt.plot(f3[0], label='Nesterov Accelerated Gradient (Strongly Convex)')
plt.plot(f4[0], label='Adagrad-Norm (b0=0.5, eta=0.01)')
plt.plot(f4_1[0], label='Adagrad-Norm (b0=0.1, eta=0.01)')
plt.plot(f5[0], label='Adam (alpha=0.001, beta1=0.9, beta2=0.999)')
plt.xlabel('Iteration')
plt.yscale('log')
plt.ylabel(r'$\Vert\nabla f(x^k)\Vert$')
plt.legend()
plt.title('Comparison of Optimization Methods with Different Hyperparameters')
plt.show()
```



In [23]:

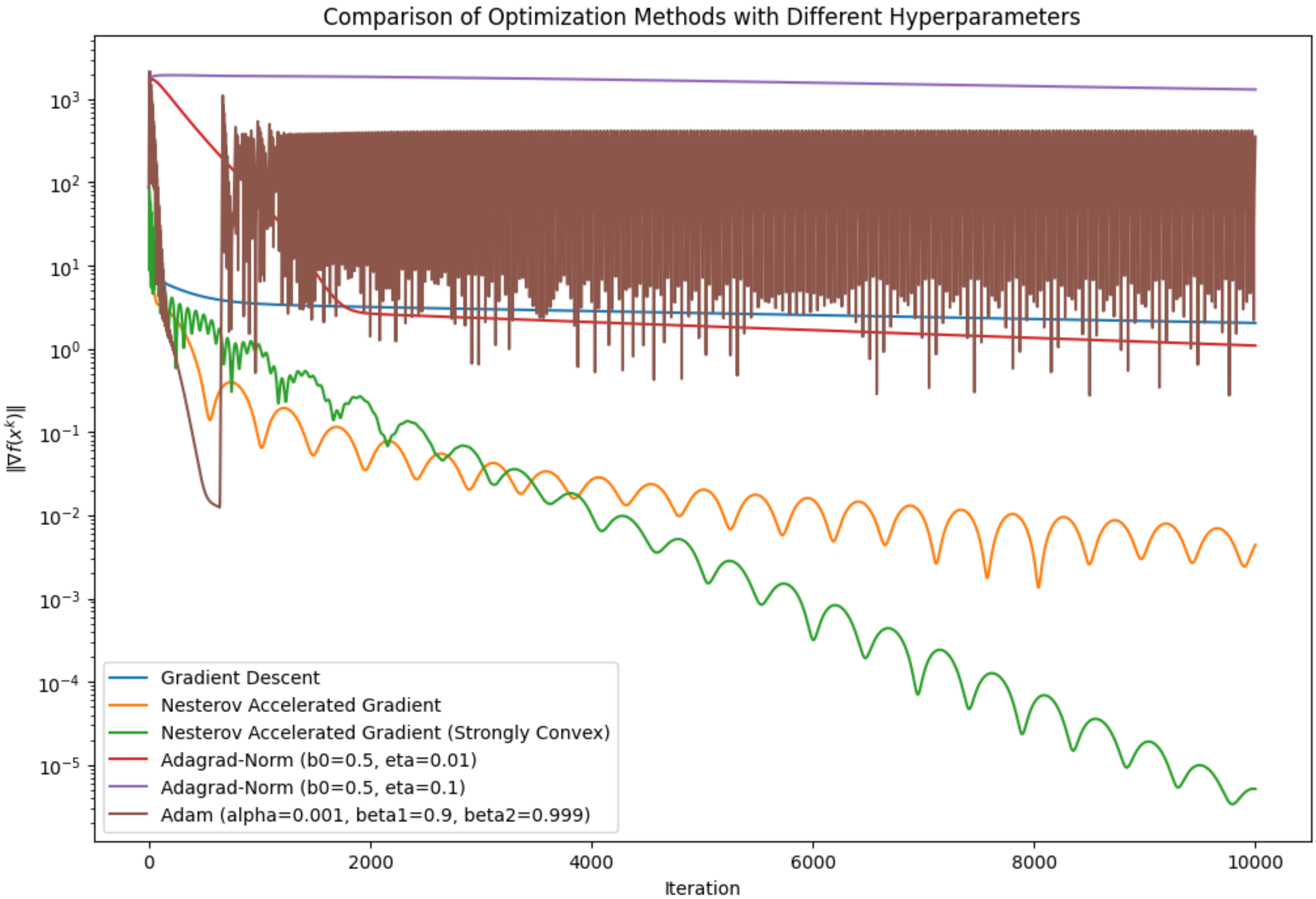
```
plt.figure(figsize=(12, 8))
plt.plot(f1[0], label='Gradient Descent')
plt.plot(f2[0], label='Nesterov Accelerated Gradient')
plt.plot(f3[0], label='Nesterov Accelerated Gradient (Strongly Convex)')
```

```python
plt.plot(f4[0], label='Adagrad-Norm (b0=0.5, eta=0.01)')
plt.plot(f4_2[0], label='Adagrad-Norm (b0=1.0, eta=0.01)')
plt.plot(f5[0], label='Adam (alpha=0.001, beta1=0.9, beta2=0.999)')
plt.xlabel('Iteration')
plt.yscale('log')
plt.ylabel(r'$\Vert\nabla f(x^k)\Vert$')
plt.legend()
plt.title('Comparison of Optimization Methods with Different Hyperparameters')
plt.show()
```



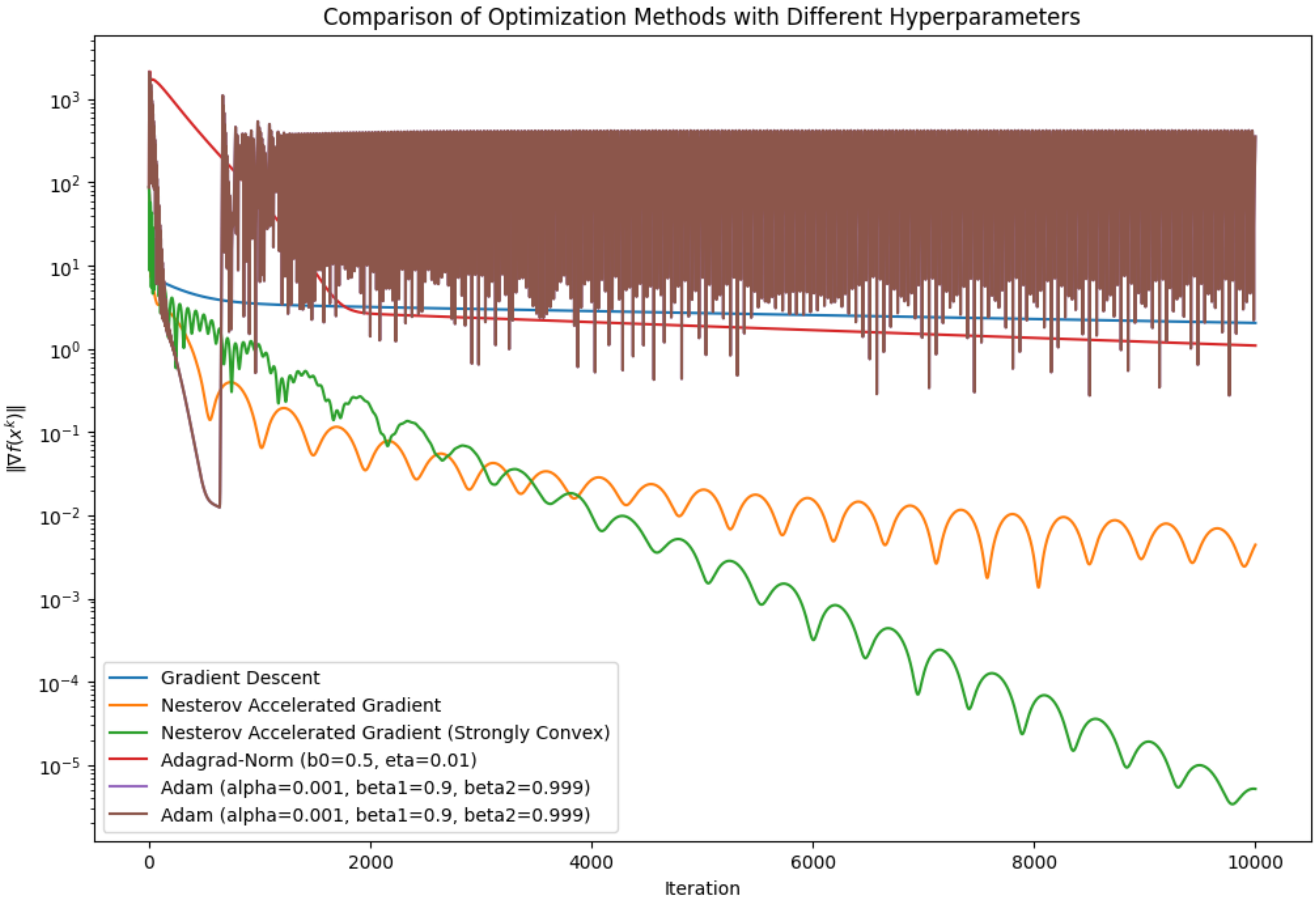
In [24]:
```
plt.figure(figsize=(12, 8))
plt.plot(f1[0], label='Gradient Descent')
plt.plot(f2[0], label='Nesterov Accelerated Gradient')
plt.plot(f3[0], label='Nesterov Accelerated Gradient (Strongly Convex)')
```

```
plt.plot(f4[0], label='Adagrad-Norm (b0=0.5, eta=0.01)')
plt.plot(f4_3[0], label='Adagrad-Norm (b0=0.5, eta=0.1)')
plt.plot(f5[0], label='Adam (alpha=0.001, beta1=0.9, beta2=0.999)')
plt.xlabel('Iteration')
plt.yscale('log')
plt.ylabel(r'$\Vert\nabla f(x^k)\Vert$')
plt.legend()
plt.title('Comparison of Optimization Methods with Different Hyperparameters')
plt.show()
```



In [25]:
```
plt.figure(figsize=(12, 8))
plt.plot(f1[0], label='Gradient Descent')
plt.plot(f2[0], label='Nesterov Accelerated Gradient')
plt.plot(f3[0], label='Nesterov Accelerated Gradient (Strongly Convex)')
```

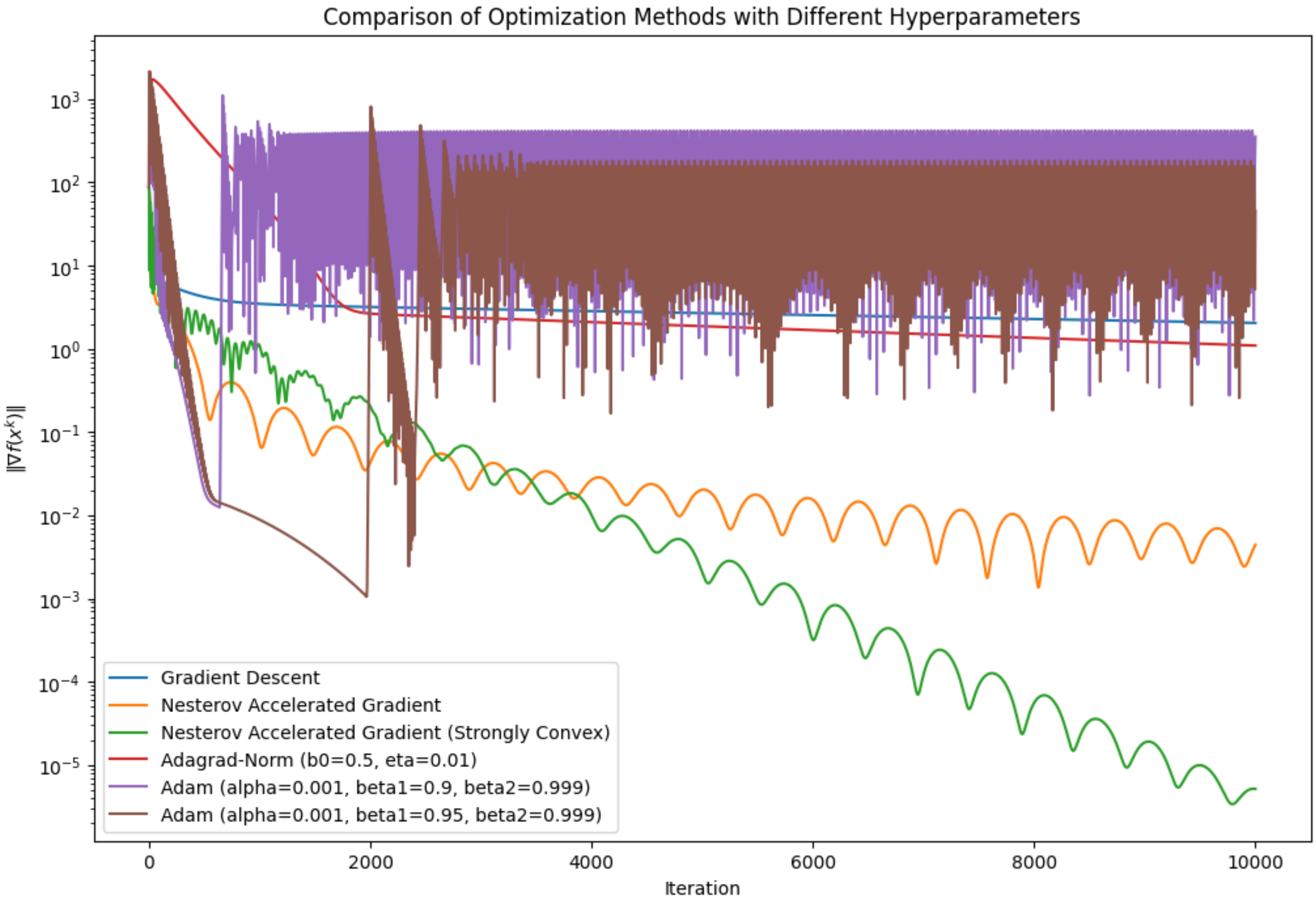
```
plt.plot(f4[0], label='Adagrad-Norm (b0=0.5, eta=0.01)')
plt.plot(f5[0], label='Adam (alpha=0.001, beta1=0.9, beta2=0.999)')
plt.plot(f5_1[0], label='Adam (alpha=0.001, beta1=0.9, beta2=0.999)')
plt.xlabel('Iteration')
plt.yscale('log')
plt.ylabel(r'$\Vert\nabla f(x^k)\Vert$')
plt.legend()
plt.title('Comparison of Optimization Methods with Different Hyperparameters')
plt.show()
```



plt.figure(figsize=(12, 8)) plt.plot(f1[0], label='Gradient Descent') plt.plot(f2[0], label='Nesterov Accelerated Gradient') plt.plot(f3[0], label='Nesterov Accelerated Gradient (Strongly Convex)') plt.plot(f4[0], label='Adagrad-Norm (b0=0.5, eta=0.01)') plt.plot(f5[0], label='Adam (alpha=0.001, beta1=0.9, beta2=0.999)') plt.plot(f5_2[0], label='Adam (alpha=0.01, beta1=0.9, beta2=0.999)') plt.xlabel('Iteration') plt.yscale('log') plt.ylabel(r'$\|\nabla f(x^k)\|$') plt.legend()

plt.title('Comparison of Optimization Methods with Different Hyperparameters') plt.show()

In [26]:
```
plt.figure(figsize=(12, 8))
plt.plot(f1[0], label='Gradient Descent')
plt.plot(f2[0], label='Nesterov Accelerated Gradient')
plt.plot(f3[0], label='Nesterov Accelerated Gradient (Strongly Convex)')
plt.plot(f4[0], label='Adagrad-Norm (b0=0.5, eta=0.01)')
plt.plot(f5[0], label='Adam (alpha=0.001, beta1=0.9, beta2=0.999)')
plt.plot(f5_3[0], label='Adam (alpha=0.001, beta1=0.95, beta2=0.999)')
plt.xlabel('Iteration')
plt.yscale('log')
plt.ylabel(r'$\Vert\nabla f(x^k)\Vert$')
plt.legend()
plt.title('Comparison of Optimization Methods with Different Hyperparameters')
plt.show()
```



Como foi possível observar, tanto o Adam quanto o Adagrad-Norm, conseguem se aproximar do GD, entretanto perdem se comparados com o Nesterov, mostrando sua possível superioridade em otimização em relação aos outros algoritmos, mesmo mudando os hyperparâmetros do Adam e do Adagrad-Norm