Streamed Principal Component Analysis

Análise de Componentes Principais (PCA) é uma das ferramentas básicas em machine learning e estatística não-supervisionada. O objetivo de PCA é reduzir a dimensão dos dados mantendo uma boa parte de sua "informação estatística". Seja $X \in \mathbb{R}^{n\times d}$ a matriz de dados cuja $i$ézima linha $x_i^\top := X_{i,\bullet}$ é uma amostra d-dimensional. Seja $\hat{\Sigma} := \frac{1}{n}X^\top X$ a matriz de covariância empírica. Por exemplo, a primeira *componente principal* é obtida resolvendo o problema de encontrar o "máximo" auto-vetor da matriz de covariância:

$$\begin{aligned} \underset{w\in\mathbb{R}^{d\times 1}}{\text{maximize}} \quad & w^\top \hat{\Sigma} w \\ \text{sujeito à} \quad & w^\top w = 1. \end{aligned}$$

De fato, podemos resolver um problema parecido para obter as k primeiras componentes principais:

$$\begin{aligned} \underset{W\in\mathbb{R}^{d\times k}}{\text{maximize}} \quad & \mathrm{tr}\left(W^\top \hat{\Sigma} W\right) \\ \text{sujeito à} \quad & W^\top W = I_k. \end{aligned}$$

O problema àcima é mais precisamente chamado *offline* PCA já que necessita ler toda a matriz de dados X para ser resolvido. Conjunto de dados modernos possuem vetores de dados com dimensão d tão grande que por na memória toda matriz de dados pode ser computacionalmente demandante. Além disso, dados modernos podem ser obtidos via "streaming", de modo que toda a matriz de dados não é observada até o momento. Uma alternativa é *streaming* ou *online* PCA. Veja IEEE para uma aplicação de online PCA para a previsão de voltagem em sistemas de rede elétrica distribuídos.

O *método de Oja* consiste num método iterativo que tenta resolver PCA lendo apenas um vetor de dados em cada iteração: inicializando-se com uma matriz W_0 e escolhendo passos η_t, iteramos:

1. $Z_{t+1} := (I_{d\times k} + \eta_k x_t x_t^\top)W_t$, onde x_t é o t-ézimo vetor de dados.
2. Calcule a decomposição QR de $Z_{t+1} = Q_{t+1}R_{t+1}$.
3. $W_{t+1} := Q_{t+1}$.

Para entender a heurística deste método note que o gradiente da função $f(W) = \mathrm{tr}(W^\top \hat{\Sigma} W)$ é

$$\nabla f(W) = 2\hat{\Sigma}W = 2\left(\frac{1}{n}\sum_{i=1}^n x_i x_i^\top\right)W.$$

Portanto, ao implementar o SGD correspondente, $2x_t x_t^\top W$ é uma amostra do gradiente na t-ézima iteração (conforme linha 1). As linhas 2-3 correspondem a projetar ortogonalmente Z_{t+1} no conjunto viável $C = \{W \in \mathbb{R}^{d\times k} : W^\top W = I_k\}$. Em conclusão, streaming PCA é o método gradiente (ascendente) estocástico projetado para resolver o problema de PCA populacional.

Começamos carregando alguns módulos necessários:

In [1]:
```python
#Chamando módulos necessários:
import numpy as np
import numpy.linalg as la
import pandas as pd
import seaborn as sns
import matplotlib
from matplotlib import pyplot as plt
from scipy import sparse as sp
import scipy.sparse.linalg as spla
from time import time
```

Exercício 1: Dados sintéticos

À seguir, iremos gerar vetores de dados x_t de uma distribuição normal multivariada $N(0, \Sigma)$ cuja matriz de covariância $\Sigma \in \mathbb{R}^{d \times d}$ tem a forma

$$\Sigma = A_0 Diag(w)^2 A_0^\top + \sigma^2 I_d,$$

onde $A_0 \in \mathbb{R}^{d \times k}$ é uma matriz cujas colunas são vetores ortonormais, $w \in \mathbb{R}^k$ é um vetor com coordenadas $w_k = \frac{u_k}{u_1}$ onde $u_1 \geq \cdots \geq u_k$ são gerados independentemente de uma distribuição uniforme $U(0,1)$ e $\sigma > 0$. Àcima $Diag(w)$ denota a matriz diagonal cuja diagonal é preenchida com o vetor w.

Construa uma função `spiked_covariance(n, d, k, sigma)` com entradas n (o tamanho da amostra), d (dimensão), k e σ que retorna Σ (`=cov`) da forma àcima, w, A_0 e matriz de dados $X \in \mathbb{R}^{n \times d}$ cujos vetores linha são gerados de $N(0, \Sigma)$. *Dica*: para gerar uma matriz $n \times k$ com colunas ortonormais você pode aplicar `la.qr()` a uma matriz $n \times k$ cujas entradas são iid $U(0,1)$. Veja as funções `np.random.rand()`, `np.sort()` e `np.random.multivariate_normal()`.

In [2]:
```python
#Escreva código aqui
def spiked_covariance(n:int, d:int, k:int, sigma:float) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    A0 = la.qr(np.random.rand(d, k))[0]

    u = np.sort(np.random.rand(k))[::-1]
    w = u / u[0]

    Sigma = A0 @ np.diag(w**2) @ A0.T + sigma**2 * np.eye(d)

    X = np.random.multivariate_normal(np.zeros(d), Sigma, n)

    return Sigma, w, A0, X
```

In [ ]:
```python
#Exemplo:
n = 500
d = 5
k = 2
```

```
sigma = 10*1e-2
cov, w, A0, X = spiked_covariance(n,d,k,sigma)
cov[:6,:6], w[:6], A0[:6,:6], X[:6,:6]
```

Out[ ]:
```
(array([[0.30346724, 0.30332387, 0.23835301, 0.18441119, 0.11053343],
        [0.30332387, 0.35085478, 0.1944905 , 0.19678209, 0.01883206],
        [0.23835301, 0.1944905 , 0.30197921, 0.13806056, 0.27076777],
        [0.18441119, 0.19678209, 0.13806056, 0.12727719, 0.04790287],
        [0.11053343, 0.01883206, 0.27076777, 0.04790287, 0.38457761]]),
 array([1.        , 0.64664984]),
 array([[-0.52184755, -0.22485767],
        [-0.49499123, -0.47874113],
        [-0.50803489,  0.28464331],
        [-0.31789586, -0.19694655],
        [-0.35142794,  0.77487819]]),
 array([[ 0.72414355,  0.64940176,  0.28037007,  0.301704  , -0.33350207],
        [-0.3305817 , -0.36223631,  0.12527922, -0.09995934,  0.64302417],
        [-0.23571263, -0.30629497,  0.2309853 , -0.20124834,  0.19240161],
        [-1.01307993, -1.13952975, -0.95772684, -0.5638142 , -0.66110348],
        [ 1.02320777,  1.2783429 ,  0.23883778,  0.69338582, -1.02569667],
        [-0.51367324, -0.59847424, -0.30308659, -0.26843794,  0.19131149]]))
```

Exercício 2: Método de Oja

Construa uma função `Oja(X,cov, k,c,lr_type,random)` que toma como entradas a matriz de dados `X`, a matriz covariância `cov`, o número de componentes principais `k`, número positivo `c`, Boolean `lr_type` onde o passo é $\eta_t = \frac{c}{t}$ se `lr_type=False` e $\eta_t = \frac{c}{\sqrt{t}}$ se `lr_type=True` e Boolean `random` onde o ponto inicial W_0 é a origem se `random=False` e randomizado se `random=True`. Tal função deve aplicar o método de Oja e retornar a sequência $\|W_t - U_{\bullet,k}\|_F^2$ e o último iterado W_n. Aqui, $U_{\bullet,k}$ é a matriz $d \times k$ cujas colunas são os k primeiros auto-vetores de Σ.

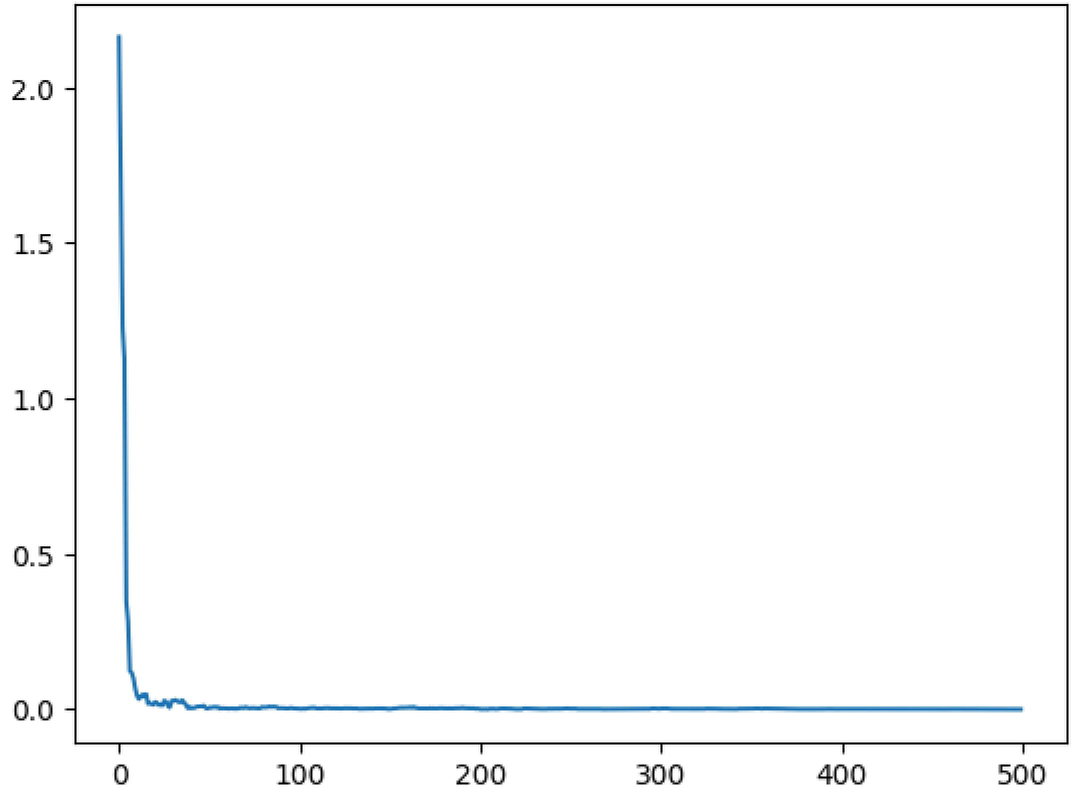
NOTA: se `random=True`, inicialize o método de Oja tomando W_0 como o fator Q da fatorizacão QR de uma matriz $d \times k$ cujas entradas são iid $N(0,1)$.

In [4]:
```
#Escreva código aqui
def Oja(X:np.ndarray, cov:np.ndarray, k:int, c:float, lr_type:bool, random:bool) -> tuple[list[float], np.ndarray]:
    n, d = X.shape
    errors = list()

    eigvals, eigvecs = la.eigh(cov)
    U_k = eigvecs[:, -k:]

    if random:
        W = la.qr(np.random.randn(d, k))[0]
    else:
        W = np.zeros((d, k))
```

```python
    for t in range(1, n + 1):
        x_t = X[t - 1].reshape(-1, 1)

        if lr_type:
            eta_t = c / np.sqrt(t)
        else:
            eta_t = c / t

        Z = W + eta_t * (x_t @ x_t.T @ W)

        W = la.qr(Z)[0]

        error = la.norm(W @ W.T - U_k @ U_k.T, 'fro') ** 2
        errors.append(float(error))

    return errors, W
```

In [5]:

```python
#Exemplo:
# Para F muito grande oscila, para F muito pequeno fica constante e maior do que deveria
F = 10*1e+1
f1 = Oja(X, cov, k, c=F*sigma, lr_type=False, random=False)

error_Oja = f1[0]

plt.plot(error_Oja)
plt.show()
```



Pergunta:

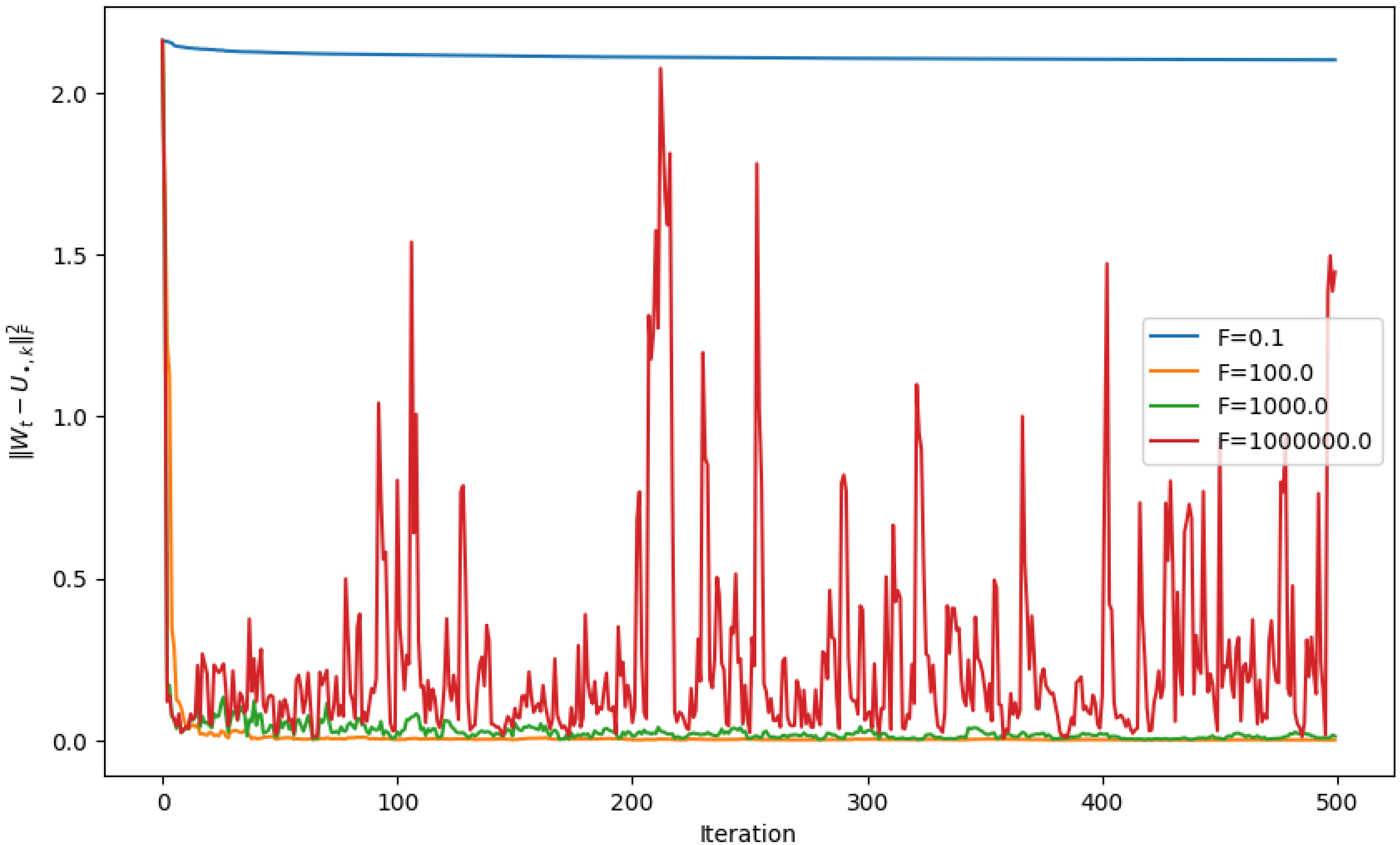
Tomando:

`n = 500  d = 5  k = 2  sigma = 10*1e-2  lr_type=False  random=False`

plote $\|W_t - U_{\bullet,k}\|_F^2$ em função do número de iterações t para `F=10*1e-2`, `F=10*1e+1`, `F=10*1e+2`, `F=10*1e+5`. Qual a sua interpretação?

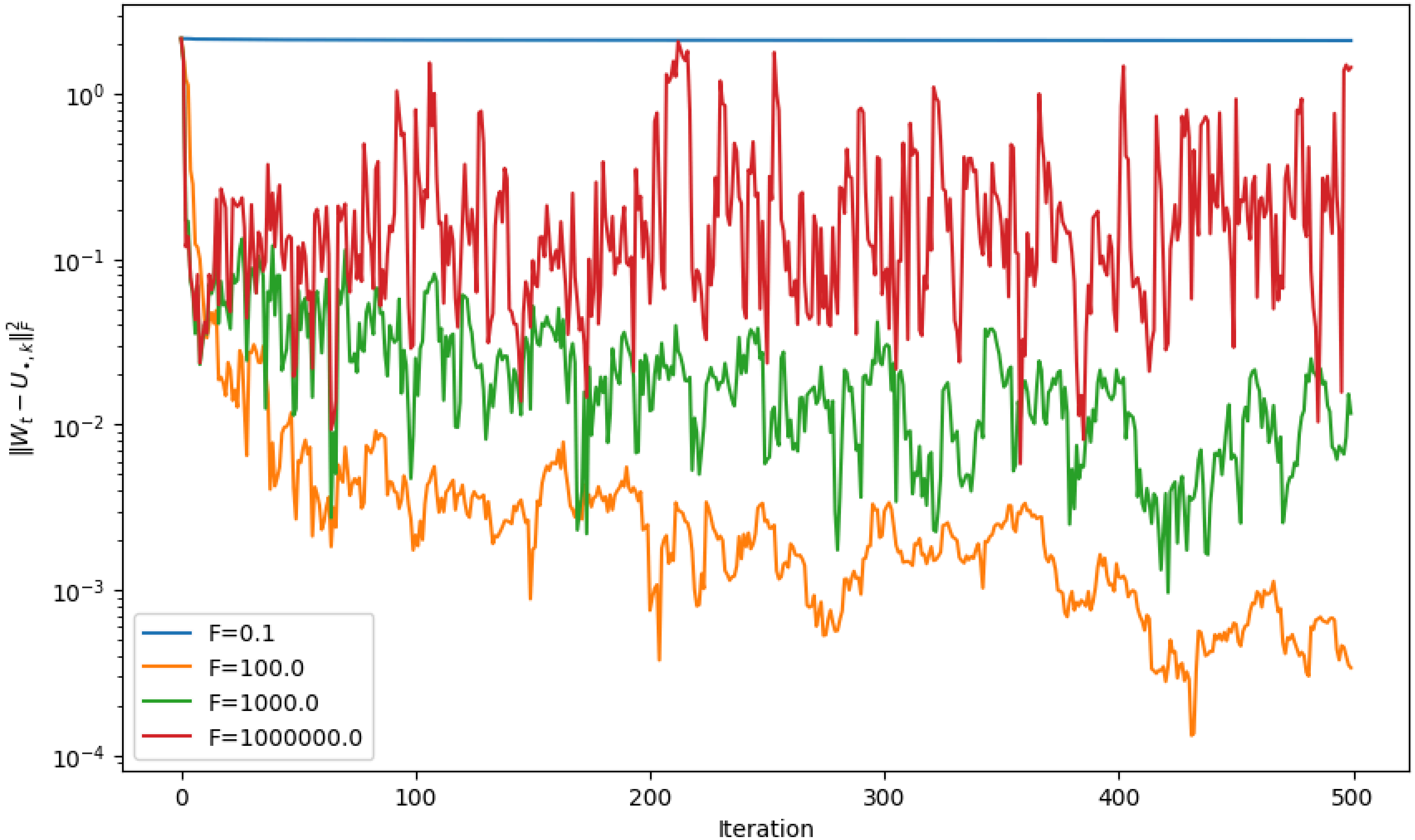
In [6]:
```
n = 500
d = 5
k = 2
sigma = 10*1e-2
lr_type = False
random = False

F_values = [10*1e-2, 10*1e+1, 10*1e+2, 10*1e+5]

plt.figure(figsize=(10, 6))
F_erros = list()
```

```python
for F in F_values:
    errors, _ = Oja(X, cov, k, c=F*sigma, lr_type=lr_type, random=random)
    F_erros.append(errors)
    plt.plot(errors, label=f'F={F}')

plt.xlabel('Iteration')
plt.ylabel(r'$\Vert W_t - U_{\bullet,k}\Vert^2_F$')
plt.legend()
plt.show()
```



In [7]:
```python
plt.figure(figsize=(10, 6))

for i in range(len(F_erros)):
    plt.plot(F_erros[i], label=f'F={F_values[i]}')

plt.xlabel('Iteration')
```

```
plt.ylabel(r'$\Vert W_t - U_{\bullet,k}\Vert^2_F$')
plt.yscale('log')
plt.legend()
plt.show()
```



Após realizar algumas execuções, foi perceptível como um F pequeno, como 0.1 atrapalha a convergência do problema, fazendo com que a mudança entre uma iteração apra outra seja pequena, já com valores de F entre 100 e 1000, a estabilidade da convergência aparenta ser bem maior, cheganod a ótimos resultados em 500 iterações, por fim, um valor de F muito alto como 1000000 deixa uma convergência instável, divergindo muito entre valores baixos e altos, isto indica que achar um F ideal pode mudar todo o problema.

Exercício 3: AdaOja-Norm

O método de Oja é bastante sensível à calibração do passo η_t. Iremos implementar uma versão adaptativa do método de Oja baseado no algoritmo Adagrad-Norm. Inicializando-se com uma matriz W_0 e escolhendo números positivos $\eta > 0$ e $b_0 > 0$, iteramos:

1. Na t-ézima iteração,

$$Z_{t+1} := \left(I_{d\times k} + \frac{\eta}{\sqrt{b_0^2 + \sum_{j=1}^t \|x_j x_j^\top W_j\|_F^2}} x_t x_t^\top \right) W_t,$$

2. Calcule a decomposição QR de $Z_{t+1} = Q_{t+1}R_{t+1}$.
3. $W_{t+1} := Q_{t+1}$.

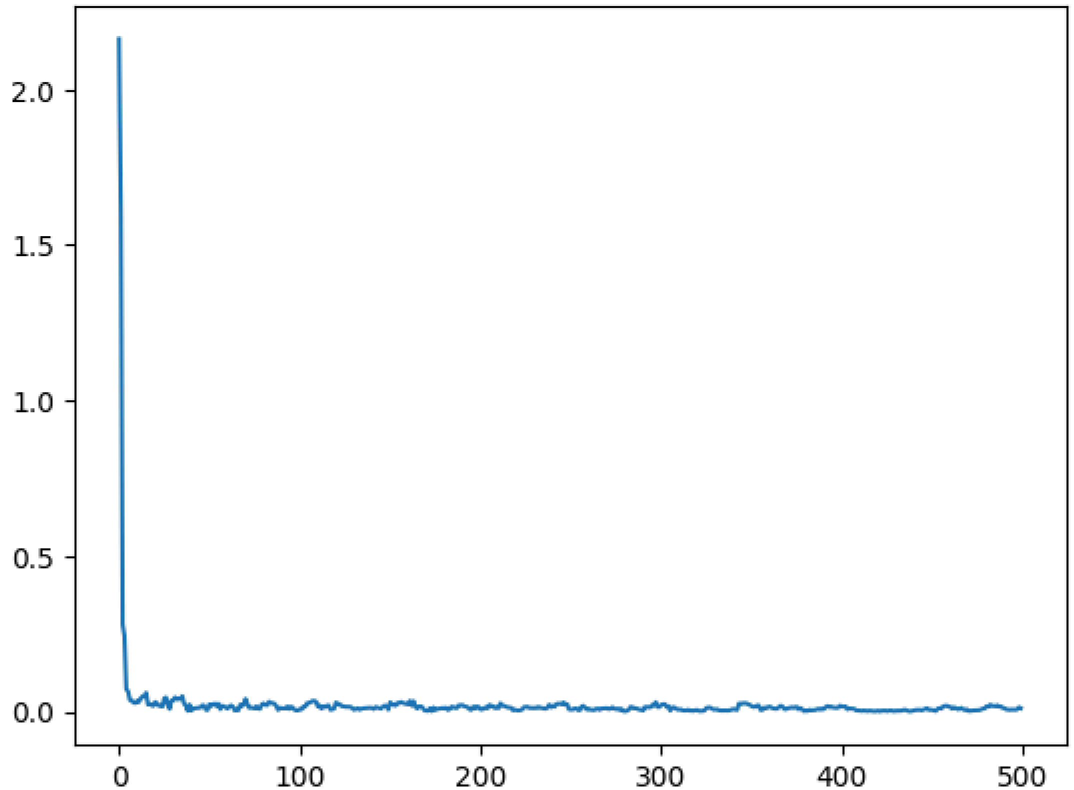
Construa uma função `AdaOja(X, cov, k, eta, b0, random)` que toma como entradas a matriz de dados `X`, a matriz covariância `cov`, o número de componentes principais `k`, número positivos `eta` e `b0` e Boolean `random` onde o ponto inicial W_0 é a origem se `random=False` e randomizado se `random=True`. Tal função deve aplicar o método AdaOja-Norm àcima e retornar a sequência $\|W_t - U_{\bullet,k}\|_F^2$ e o último iterado W_n. Aqui, $U_{\bullet,k}$ é a matriz $d \times k$ cujas colunas são os k primeiros auto-vetores de Σ.

NOTA: se `random=True`, inicialize o método AdaOja tomando W_0 como o fator Q da fatorizacão QR de uma matriz $d \times k$ cujas entradas são iid $N(0,1)$.

In [8]:
```
#Escreva código aqui
def AdaOja(X:np.ndarray, cov:np.ndarray, k:int, eta:float, b0:float, random:bool) -> tuple[list[float], np.ndarray]:
    n, d = X.shape
    errors = list()

    eigvals, eigvecs = la.eigh(cov)
    U_k = eigvecs[:, -k:]

    if random:
        W = la.qr(np.random.randn(d, k))[0]
    else:
        W = np.zeros((d, k))

    sum_grad_norms = 0

    for t in range(1, n + 1):
        x_t = X[t - 1].reshape(-1, 1)
        grad = x_t @ x_t.T @ W
        sum_grad_norms += la.norm(grad, 'fro') ** 2

        eta_t = eta / np.sqrt(b0 ** 2 + sum_grad_norms)
        Z = W + eta_t * grad

        W = la.qr(Z)[0]

        error = la.norm(W @ W.T - U_k @ U_k.T, 'fro') ** 2
```

```
        errors.append(float(error))

    return errors, W
```

In [9]:
```
#Exemplo:
F = 10*1e+1
f2 = AdaOja(X, cov, k, eta=F*sigma, b0=1e-5, random=False)

error_AdaOja = f2[0]

plt.plot(error_AdaOja)
plt.show()
```



Exercício 4: AdaOja-Coordinate

Alternativamente, podemos implementar a versão baseada no Adagrad-Coordinate, a iteração é implementada coordenada a coordenada (semelhante ao método Adam):

1. Na t-ézima iteração,

$$G_t := x_t x_t^\top W_t,$$
$$b_t[j] := \sqrt{b_0^2 + \sum_{j=1}^{t} \|G_t[:,j]\|_2^2},$$
$$Z_{t+1}[:,j] := Z_t[:,j] + \frac{\eta}{b_t[j]} G_t[:,j].$$

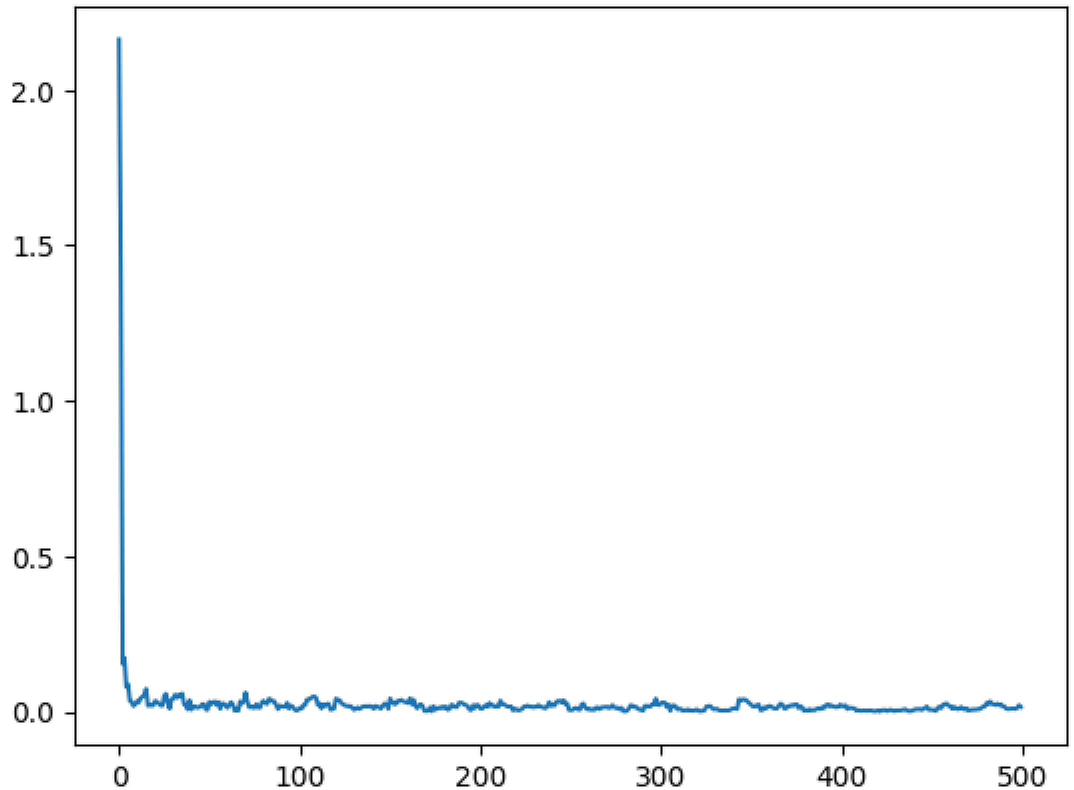
2. Calcule a decomposição QR de $Z_{t+1} = Q_{t+1}R_{t+1}$.
3. $W_{t+1} := Q_{t+1}$.

Construa uma função `AdaOja_Coord(X, cov, k, eta, b0, random)` que toma como entradas a matriz de dados `X`, a matriz covariância `cov`, o número de componentes principais `k`, número positivos `eta` e `b0` e Boolean `random` onde o ponto inicial W_0 é a origem se `random=False` e randomizado se `random=True`. Tal função deve aplicar o método AdaOja-Coordindate àcima e retornar a sequência $\|W_t - U_{\bullet,k}\|_F^2$ e o último iterado W_n. Aqui, $U_{\bullet,k}$ é a matriz $d \times k$ cujas colunas são os k primeiros auto-vetores de Σ.

NOTA: se `random=True`, inicialize o método AdaOja-Coord tomando W_0 como o fator Q da fatorizacão QR de uma matriz $d \times k$ cujas entradas são iid $N(0,1)$.

In [10]:
```
#Escreva código aqui
def AdaOja_Coord(X:np.ndarray, cov:np.ndarray, k:int, eta:float, b0:float, random:bool) -> tuple[list[float], np.ndarray]:
    n, d = X.shape
    errors = list()

    eigvals, eigvecs = la.eigh(cov)
    U_k = eigvecs[:, -k:]

    if random:
        W = la.qr(np.random.randn(d, k))[0]
    else:
        W = np.zeros((d, k))

    b_t = (b0 * np.ones(k)).reshape(-1, 1)

    for t in range(1, n + 1):
        x_t = X[t - 1].reshape(-1, 1)
        G_t = x_t @ x_t.T @ W

        for j in range(k):
            b_t[j] = np.sqrt(b_t[j] ** 2 + la.norm(G_t[:, j]) ** 2)
            W[:, j] += (eta / b_t[j]) * G_t[:, j]

        W = la.qr(W)[0]

        error = la.norm(W @ W.T - U_k @ U_k.T, 'fro') ** 2
        errors.append(float(error))

    return errors, W
```

In [11]:
```
#Exemplo:
F = 10*1e+1
f2 = AdaOja_Coord(X, cov, k, eta=F*sigma, b0=1e-5, random=False)

error_AdaOja_Coord = f2[0]

plt.plot(error_AdaOja_Coord)
plt.show()
```



Dados CIFAR-10

Iremos utilizar os dados CIFAR-10. Ele consiste de 60mil imagens 32x32 com 10 classes. Não usaremos os labels e usaremos apenas os 5 primeiros batches de 50mil imagens com 10mil imagens cada um. Estes correspondem ao dados de treinamento ("training data") com arquivos:

1. data_batch_1
2. data_batch_2
3. data_batch_3
4. data_batch_4

5. data_batch_5

Estes estão na pasta onde este notebook está salvo. Maiores informações sobre como ler os dados CIFAR-10 veja vídeo e aplicações de redução de dimensão via PCA no CIFAR-10 em https://www.kaggle.com/code/adtysregita/pca-application-using-cifar10-dataset.

À seguir iremos ler os dados e salvá-los como uma matriz de dados `X`. As imagens estão decodificadas em arquivo texto, portanto temos que ler os dados e transformá-los em imagens. Note que cada linha corresponderá a uma imagem 32x32 *vetorizada*. Mais precisamente, cada linha tem dimensão 3072 correspondente a 3 imagens 32x32, cada uma com os tons de vermelho, azul e verde. Este formato é suficiente para aplicarmos PCA. Ao usarmos todos 5 batches, leva-se um tempo demorado para o método de Oja processar todas as 50000 iterações. Portanto, iremos dar a opção de escolher os `m` primeiros batches para colocar da matriz de dados `X`.

In [12]:
```
# Criando uma função para ler os dados e transformá-los em imagens, salvas em um dicionário python:
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
```

In [13]:
```
#Exemplo:
unpickle('./datasets/data_batch_2')[b'data'].shape, unpickle('./datasets/data_batch_2')[b'data']
```

Out[13]:
```
((10000, 3072),
 array([[ 35,  27,  25, ..., 169, 168, 168],
        [ 20,  20,  18, ..., 111,  97,  51],
        [116, 115, 155, ...,  18,  84, 124],
        ...,
        [127, 139, 155, ..., 197, 192, 191],
        [190, 200, 208, ..., 163, 182, 192],
        [177, 174, 182, ..., 119, 127, 136]], dtype=uint8))
```

In [14]:
```
# Colocando m batches em uma única matriz de dados X e escolhendo k:
db = []
m = 5
k = 2

for i in range(1, m+1):
    db.append(unpickle('./datasets/data_batch_' + str(i))[b'data'])
CIFAR = np.vstack(db)

# Centralizando os dados:
X = CIFAR - CIFAR.mean(axis=0)
```

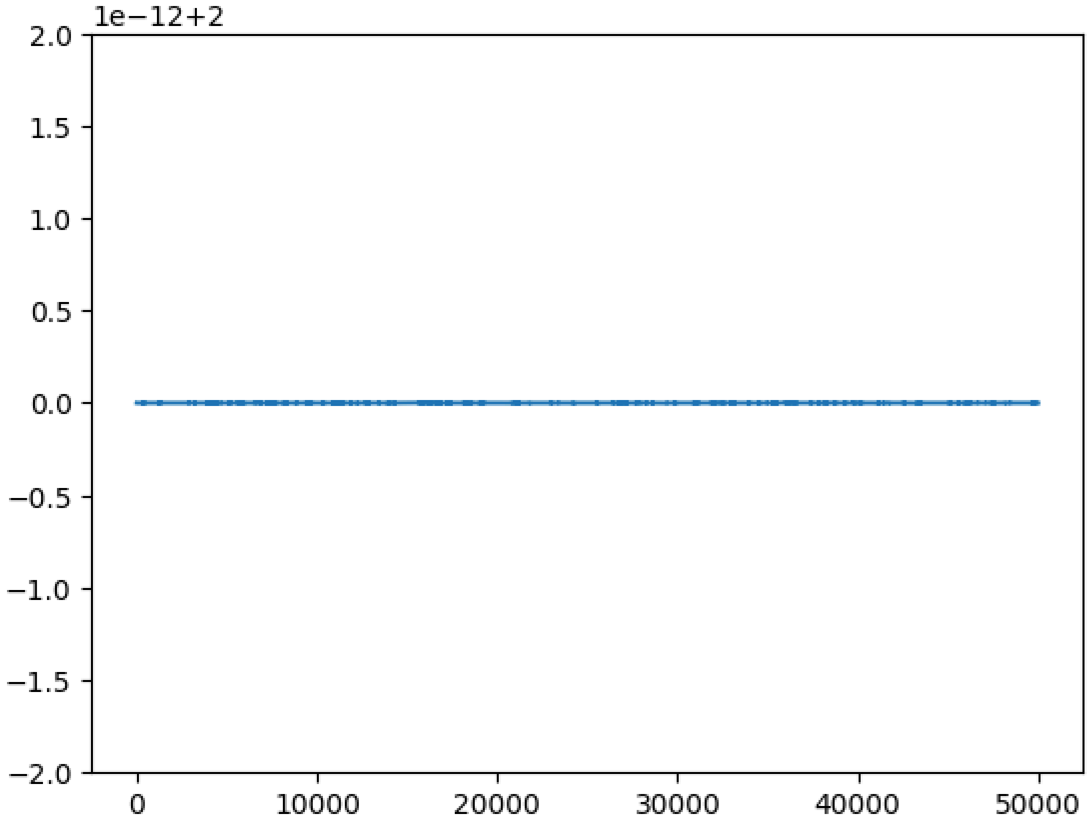
Exercício 6: Método de Oja em CIFAR-10

Construa uma função `Oja_real(X,k,c,lr_type,random)` que toma como entradas a matriz de dados `X`, o número de componentes principais `k`, número positivo `c`, Boolean `lr_type` onde o passo é $\eta_t = \frac{c}{t}$ se `lr_type=False` e $\eta_t = \frac{c}{\sqrt{t}}$ se `lr_type=True` e Boolean `random` onde o ponto inicial W_0 é a origem se `random=False` e randomizado se `random=True`. Tal função deve aplicar o método de Oja e retornar a sequência $\|W_t\|_F^2$ e o último iterado W_n.

NOTA: se `random=True`, inicialize o método de Oja tomando W_0 como o fator Q da fatorizacão QR de uma matriz $d \times k$ cujas entradas são iid $N(0,1)$.

In [15]:
```
#Escreva código aqui
def Oja_real(X:np.ndarray, k:int, c:float, lr_type:bool, random:bool) -> tuple[list[float], np.ndarray]:
    n, d = X.shape
    errors = list()

    if random:
        W = la.qr(np.random.randn(d, k))[0]
    else:
        W = np.zeros((d, k))

    for t in range(1, n + 1):

        x_t = X[t - 1].reshape(-1, 1)

        if lr_type:
            eta_t = c / np.sqrt(t)
        else:
            eta_t = c / t

        Z = W + eta_t * (x_t @ x_t.T @ W)

        W = la.qr(Z)[0]

        error = la.norm(W, 'fro') ** 2
        errors.append(float(error))

    return errors, W
```

In [16]:
```
#Exemplo:
ans1 = Oja_real(X, k, c=10*1e-1, lr_type=False, random=False)
error_Oja_real = ans1[0]

plt.plot(error_Oja_real)
plt.show()
```



Exercício 7: AdaOja-Norm em CIFAR-10

Construa uma função `AdaOja_real(X, k, eta, b0, random)` que toma como entradas a matriz de dados `X`, o número de componentes principais `k`, número positivos `eta` e `b0` e Boolean `random` onde o ponto inicial W_0 é a origem se `random=False` e randomizado se `random=True`. Tal função deve aplicar o método AdaOja-Norm àcima e retornar a sequência $\|W_t\|_F^2$ e o último iterado W_n.

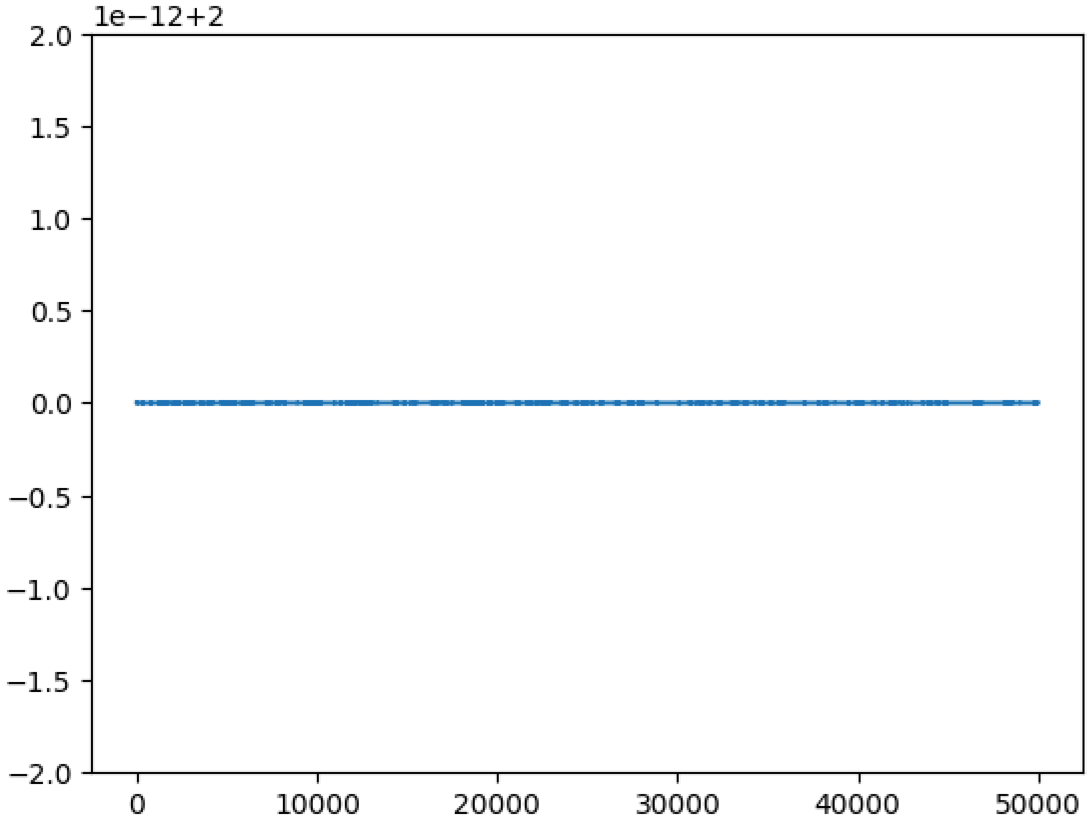
NOTA: se `random=True`, inicialize o método AdaOja tomando W_0 como o fator Q da fatorizacão QR de uma matriz $d \times k$ cujas entradas são iid $N(0, 1)$.

In [17]:
```
#Escreva código aqui
def AdaOja_real(X:np.ndarray, k:int, eta:float, b0:float, random:bool) -> tuple[list[float], np.ndarray]:
    n, d = X.shape
    errors = list()

    if random:
        W = la.qr(np.random.randn(d, k))[0]
    else:
        W = np.zeros((d, k))
```

```python
    sum_grad_norms = 0

    for t in range(1, n + 1):
        x_t = X[t - 1].reshape(-1, 1)
        grad = x_t @ x_t.T @ W
        sum_grad_norms += la.norm(grad, 'fro') ** 2

        eta_t = eta / np.sqrt(b0 ** 2 + sum_grad_norms)
        Z = W + eta_t * grad

        W = la.qr(Z)[0]

        error = la.norm(W, 'fro') ** 2
        errors.append(float(error))

    return errors, W
```

In [18]:

```python
#Exemplo:
ans2 = AdaOja_real(X, k, eta=10*1e-1, b0=1e-5, random=False)

error_AdaOja_real = ans2[0]

plt.plot(error_AdaOja_real)
plt.show()
```



Exercício 8: AdaOja-Coord em CIFAR-10

Construa uma função `AdaOja_Coord_real(X, k, eta, b0, random)` que toma como entradas a matriz de dados `X`, o número de componentes principais `k`, número positivos `eta` e `b0` e Boolean `random` onde o ponto inicial W_0 é a origem se `random=False` e randomizado se `random=True`. Tal função deve aplicar o método AdaOja-Coordindate àcima e retornar a sequência $\|W_t\|_F^2$ e o último iterado W_n.

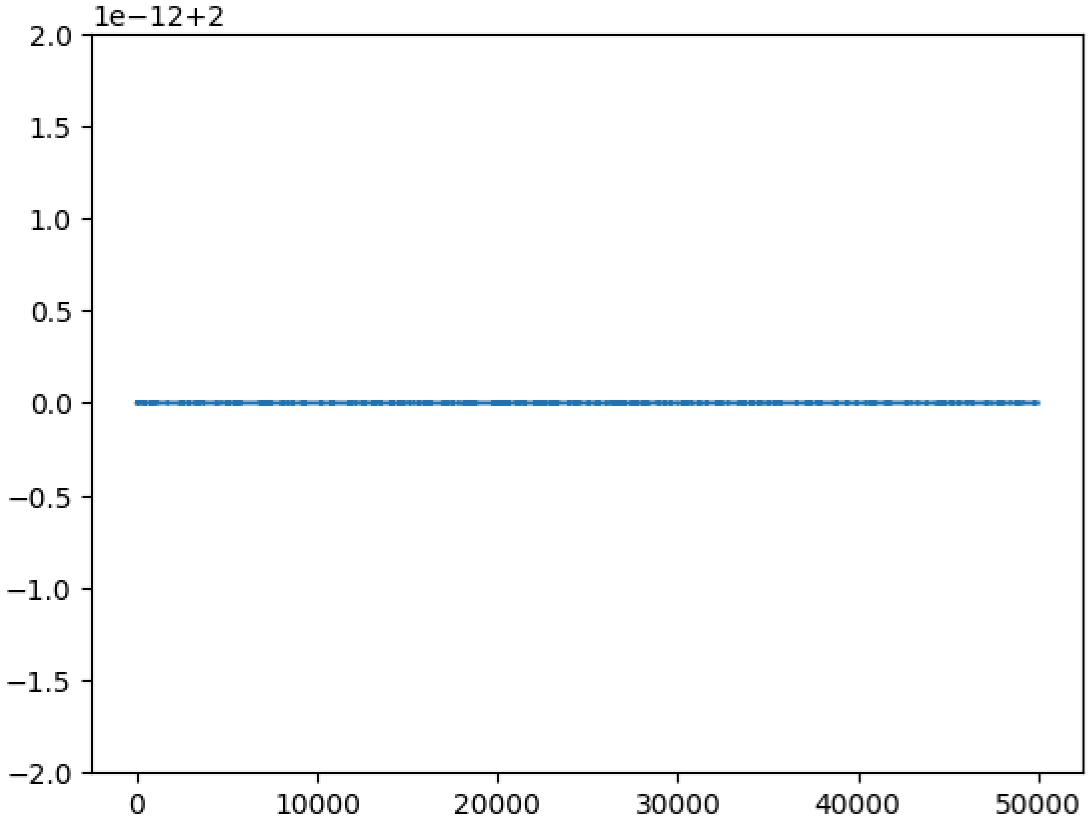
NOTA: se `random=True`, inicialize o método AdaOja-Coord tomando W_0 como o fator Q da fatorizacão QR de uma matriz $d \times k$ cujas entradas são iid $N(0, 1)$.

In [19]:
```
#Escreva código aqui
def AdaOja_Coord_real(X:np.ndarray, k:int, eta:float, b0:float, random:bool) -> tuple[list[float], np.ndarray]:
    n, d = X.shape
    errors = list()

    if random:
        W = la.qr(np.random.randn(d, k))[0]
    else:
        W = np.zeros((d, k))
```

```python
    b_t = (b0 * np.ones(k)).reshape(-1, 1)

    for t in range(1, n + 1):
        x_t = X[t - 1].reshape(-1, 1)
        G_t = x_t @ x_t.T @ W

        for j in range(k):
            b_t[j] = np.sqrt(b_t[j] ** 2 + la.norm(G_t[:, j]) ** 2)
            W[:, j] += (eta / b_t[j]) * G_t[:, j]

        W = la.qr(W)[0]

        error = la.norm(W, 'fro') ** 2
        errors.append(float(error))

    return errors, W
```

In [20]:

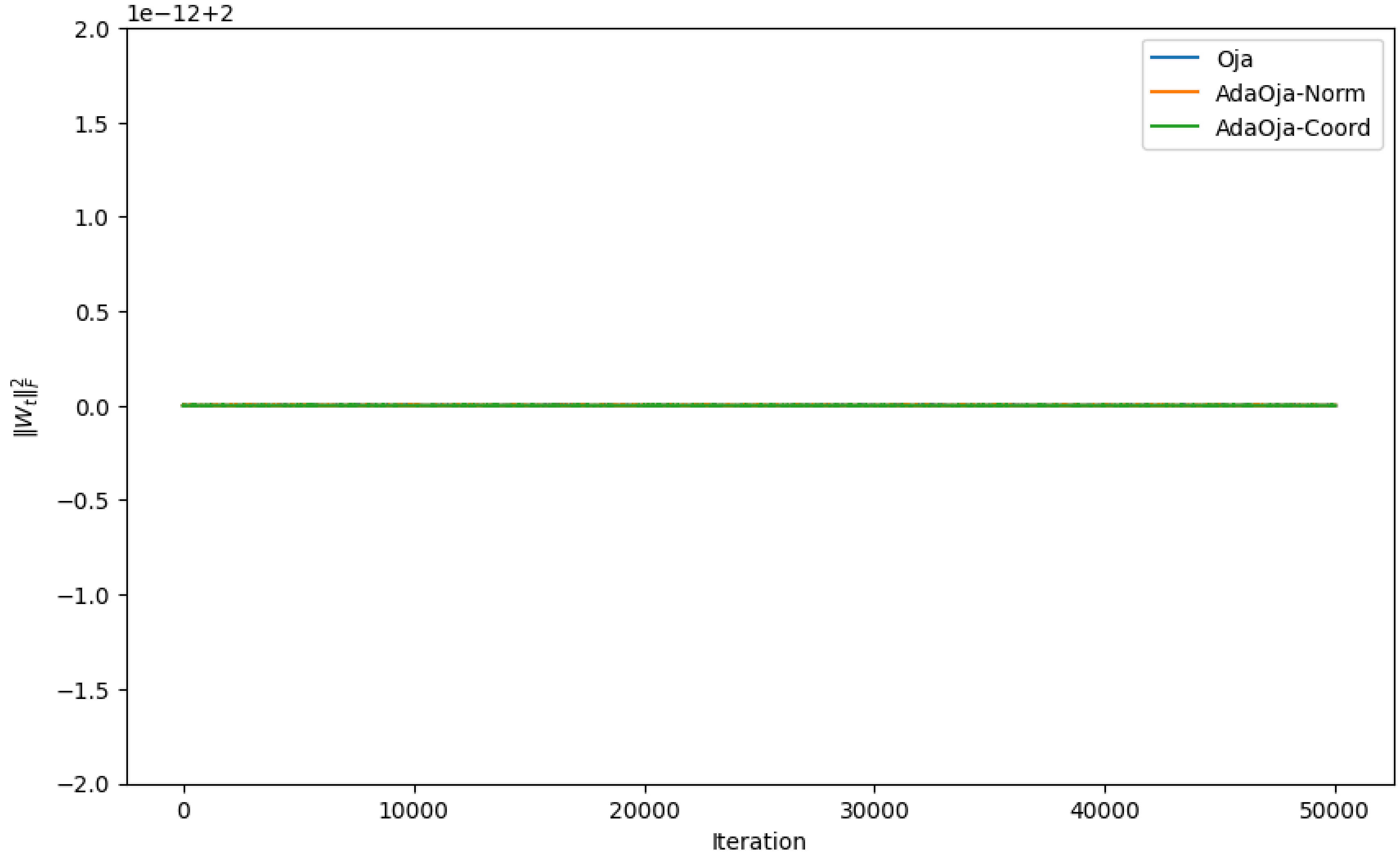
```python
#Exemplo:
ans3 = AdaOja_Coord_real(X, k, eta=10*1e-1, b0=1e-5, random=False)

error_AdaOja_Coord_real = ans3[0]

plt.plot(error_AdaOja_Coord_real)
plt.show()
```



Exercício 9:

Implemente num mesmo gráfico os erros $\|W_k\|_F^2$ de cada método em CIFAR-10 em função no número de iterações.

In [21]:
```
#Escreva código aqui

plt.figure(figsize=(10, 6))
plt.plot(error_Oja_real, label='Oja')
plt.plot(error_AdaOja_real, label='AdaOja-Norm')
plt.plot(error_AdaOja_Coord_real, label='AdaOja-Coord')
plt.xlabel('Iteration')
plt.ylabel(r'$\Vert W_t \Vert^2_F$')
plt.legend()
plt.show()
```


1e−12+2
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
$\|W_t\|_F^2$
0
10000
20000
30000
40000
50000
Iteration
Oja
AdaOja-Norm
AdaOja-Coord