

**Disciplina:** Métodos Numéricos

**Semestre:** 2020.2

**Aluno:** Jardel Brandon de Araujo Regis

**Mátricula:** 201621250014

**2ª Unidade:** Álgebra Linear Computacional

# Projeto - 1

## Remoção de fundo com PCA robusto

### Base de dados

Será utilizado o conjunto de dados de vídeo real 003 do [BMC 2012 Background Models Challenge Dataset](#)

Outras fontes de conjuntos de dados:

- [Conjuntos de dados de vídeo de atividade humana](#)
- [Site de subtração de plano de fundo](#) (alguns links neste site estão corrompidos/desatualizados, mas muitos funcionam)

```
In [1]: import moviepy.editor as mpe
        from glob import glob
        from PIL import Image
```

```
In [2]: import sys, os
        import numpy as np
        import scipy.misc
        import scipy
```

```
In [3]: %matplotlib inline
        import matplotlib.pyplot as plt
```

```
In [4]: # MAX_ITERS = 10
        TOL = 1.0e-8
```

```
In [5]: video = mpe.VideoFileClip('data/video_003.avi') # https://github.com/momonala/image-processing-proc
```

```
In [6]: video.subclip(0,50).ipython_display(width=300)
```

```
t: 0%|
0<?, ?it/s, now=None]
```

| 0/350 [00:0

```
Moviepy - Building video __temp__.mp4.  
Moviepy - Writing video __temp__.mp4
```

```
Moviepy - Done !  
Moviepy - video ready __temp__.mp4
```

Out[6]:

0:00 / 0:50



```
In [7]: video.duration
```

Out[7]: 113.57

## Métodos auxiliares

```
In [8]: def create_data_matrix_from_video(clip, k=5, scale=50):  
        return np.vstack([rescale(rgb2gray(clip.get_frame(i/float(k))).astype(int),  
                                scale).flatten() for i in range(k * int(clip.duration))]).T
```

```
In [9]: def create_data_matrix_from_video(clip, k, scale):  
        frames = []  
        for i in range(k * int(clip.duration)):  
            frame = clip.get_frame(i / float(k))  
            frame = rgb2gray(frame).astype(int)  
            image = Image.fromarray(frame)  
            size = tuple((np.array(image.size) * scale).astype(int))  
            scaled_image = np.array(image.resize(size)).flatten()  
            frames.append(scaled_image)  
        return np.vstack(frames).T # stack images horizontally
```

```
In [10]: def create_data_matrix_from_video(clip, k, scale):  
        frames = []  
        for i in range(k * int(clip.duration)):  
            frame = clip.get_frame(i / float(k))  
            frame = rgb2gray(frame).astype(int)  
            image = Image.fromarray(frame)  
            size = tuple((np.array(image.size) * scale).astype(int))  
            scaled_image = np.array(image.resize(size)).flatten()  
            frames.append(scaled_image)  
        return np.vstack(frames).T # stack images horizontally
```

```
In [11]: def rgb2gray(rgb):  
        return np.dot(rgb[..., :3], [0.299, 0.587, 0.114])
```

```
In [12]: def plt_images(M, A, E, index_array, dims, filename=None):
    f = plt.figure(figsize=(15, 10))
    r = len(index_array)
    pics = r * 3
    for k, i in enumerate(index_array):
        for j, mat in enumerate([M, A, E]):
            sp = f.add_subplot(r, 3, 3*k + j + 1)
            sp.axis('Off')
            pixels = mat[:,i]
            if isinstance(pixels, scipy.sparse.csr_matrix):
                pixels = pixels.todense()
            plt.imshow(np.reshape(pixels, dims), cmap='gray')
    return f
```

```
In [13]: def plots(ims, dims, figsize=(15,20), rows=1, interp=False, titles=None):
    if type(ims[0]) is np.ndarray:
        ims = np.array(ims)
    f = plt.figure(figsize=figsize)
    for i in range(len(ims)):
        sp = f.add_subplot(rows, len(ims)//rows, i+1)
        sp.axis('Off')
        plt.imshow(np.reshape(ims[i], dims), cmap="gray")
```

## Carregamento e visualização os dados

Uma imagem de 1 momento no tempo tem 60 pixels por 80 pixels (quando dimensionada). Podemos *desenrolar* essa imagem em uma única coluna alta. Então, em vez de ter uma imagem 2D que é  $60 \times 80$ , temos uma coluna  $1 \times 4.800$

Isso não é muito legível, mas é útil porque permite empilhar as imagens de diferentes momentos umas sobre as outras, para colocar um vídeo todo em uma matriz. Se fosse pego a imagem do vídeo a cada décimo de segundo por 113 segundos (ou seja, 11.300 imagens diferentes, cada uma de um ponto no tempo diferente), teríamos uma matriz  $11300 \times 4800$ , representando o vídeo!

```
In [14]: fps = 100
scale = 25 # Adjust scale to change resolution of image
scale_percent = scale / 100

original_width = video.size[1]
original_height = video.size[0]

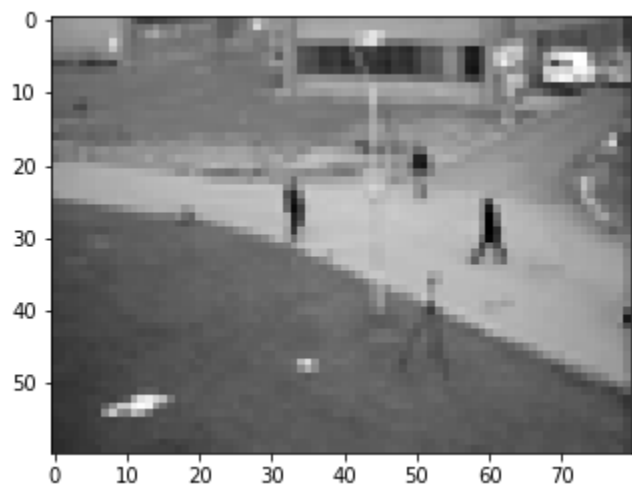
dims = (int(original_width * scale_percent), int(original_height * scale_percent))
```

```
In [15]: M = create_data_matrix_from_video(video, fps, scale_percent)
```

```
In [16]: print(dims, M.shape)
```

```
(60, 80) (4800, 11300)
```

```
In [17]: plt.imshow(np.reshape(M[:,140], dims), cmap='gray');
```



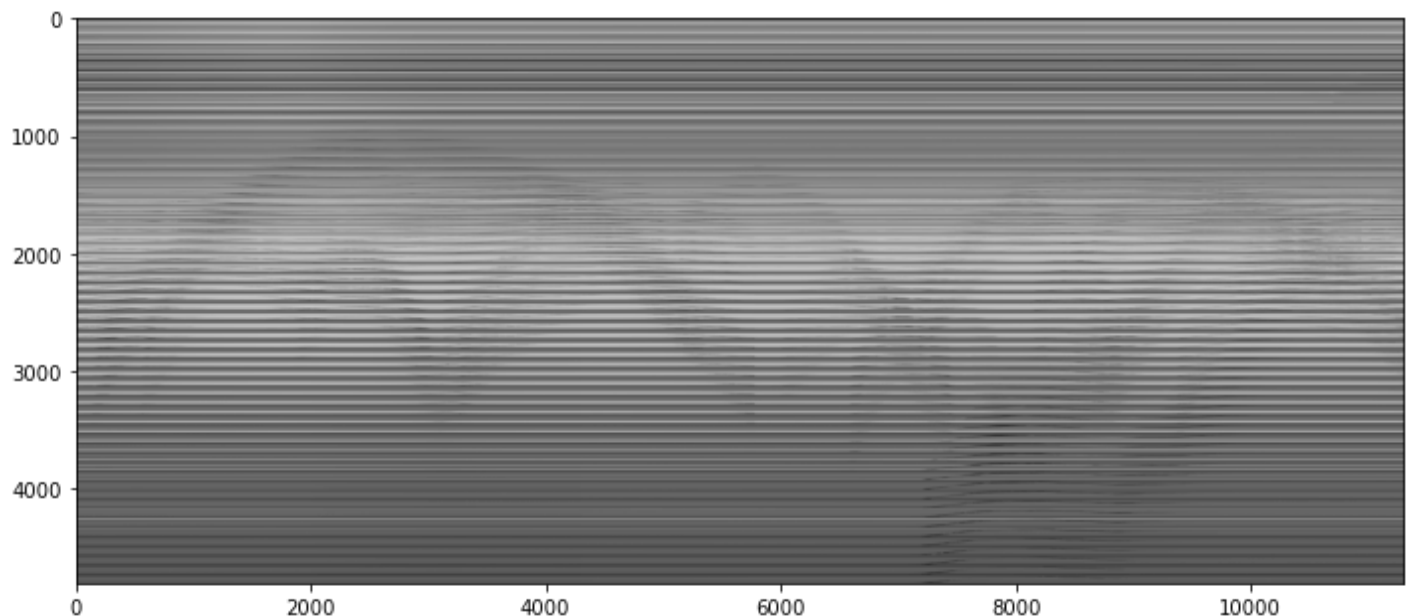
Como `create_data_from_matrix` é um tanto lento, vamos salvar a matriz. Em geral, sempre que houver etapas de pré-processamento lentas, é uma boa ideia prática salvar os resultados para uso futuro

```
In [18]: np.save("low_res_surveillance_matrix.npy", M)
```

Dessa forma é possível observar uma combinação linear (com pesos aleatórios) das colunas na matriz abaixo:

```
In [19]: plt.figure(figsize=(12, 12))
plt.imshow(M, cmap='gray')
```

```
Out[19]: <matplotlib.image.AxesImage at 0x23e594ec2c8>
```



Na imagem acima, as linhas planas horizontais são o mesmo pixel em todo o fluxo de vídeo. As curvas mostram mudanças nesse pixel, o movimento das pessoas. Linhas planas significam nenhum movimento e, matematicamente, baixa variação. O movimento pode ser visto como ruído em um sinal estável. Portanto, a filtragem de ruído deve nos permitir extrair as pessoas.

É como uma média ponderada aleatória. Se você pegar vários deles, acabará com colunas que são ou ortonormais entre si.

```
In [20]: plt.imsave(fname="image1.jpg", arr=np.reshape(M[:,140], dims), cmap='gray')
```

# Decomposição de valor singular (SVD)

Obviamente, esperaríamos que as palavras que aparecem com mais frequência em um tópico apareçam com menos frequência no outro - caso contrário, essa palavra não seria uma boa escolha para separar os dois tópicos. Portanto, esperamos que os tópicos sejam **ortogonais**. O algoritmo SVD fatoriza uma matriz em uma matriz com **colunas ortogonais** e outra com **linhas ortogonais** (junto com uma matriz diagonal, que contém a **importância relativa** de cada fator).

(fonte: [Facebook Research: Fast Randomized SVD] (<https://research.fb.com/fast-randomized-svd>))

SVD é uma **decomposição exata**, pois as matrizes que ele cria são grandes o suficiente para cobrir totalmente a matriz original. SVD é extremamente usado em álgebra linear e, especificamente, em ciência de dados, incluindo: - análise semântica - filtragem / recomendações colaborativas ([entrada vencedora do Prêmio Netflix](#)) - calcular o pseudoinverso Moore-Penrose - compressão de dados - análise de componentes principais (será abordada posteriormente no curso)

Pode-se fatorar a matriz do vídeo usando SVD para extrair os "conceitos" da matriz. Nesse caso, os conceitos principais devem ser artefatos de movimento. Na equação abaixo, A matriz de vídeo de entrada, U é uma matriz com dimensões  $[m, r]$  onde m é o único quadro do vídeo e r é o número de "conceitos",  $\Sigma$  é a matriz diagonal representando de tamanho  $[r, r]$  onde a magnitude de qualquer componente  $\Sigma_{ij}$  é a força desse componente, e  $V^T$  é a transposta de uma matriz de forma  $[r, n]$  onde n é o número de quadros do vídeo.

$$A = U\Sigma V^T$$

Será utilizado uma API de decomposição do sklearn para realizar o cálculo. `decomposition.randomized_svd` possui o argumento `n_components`, que é um hiperparâmetro que representa a variável  $r$  acima. É o número de "conceitos" que estamos tentando modelar abstratamente. Será definido arbitrariamente como dois, mas eventualmente precisará ser ajustado em um caso de uso realista.

## Uma primeira abordagem com SVD

```
In [21]: from sklearn import decomposition
```

```
In [22]: u, s, v = decomposition.randomized_svd(M, 2)
```

```
In [23]: u.shape, s.shape, v.shape
```

```
Out[23]: ((4800, 2), (2,), (2, 11300))
```

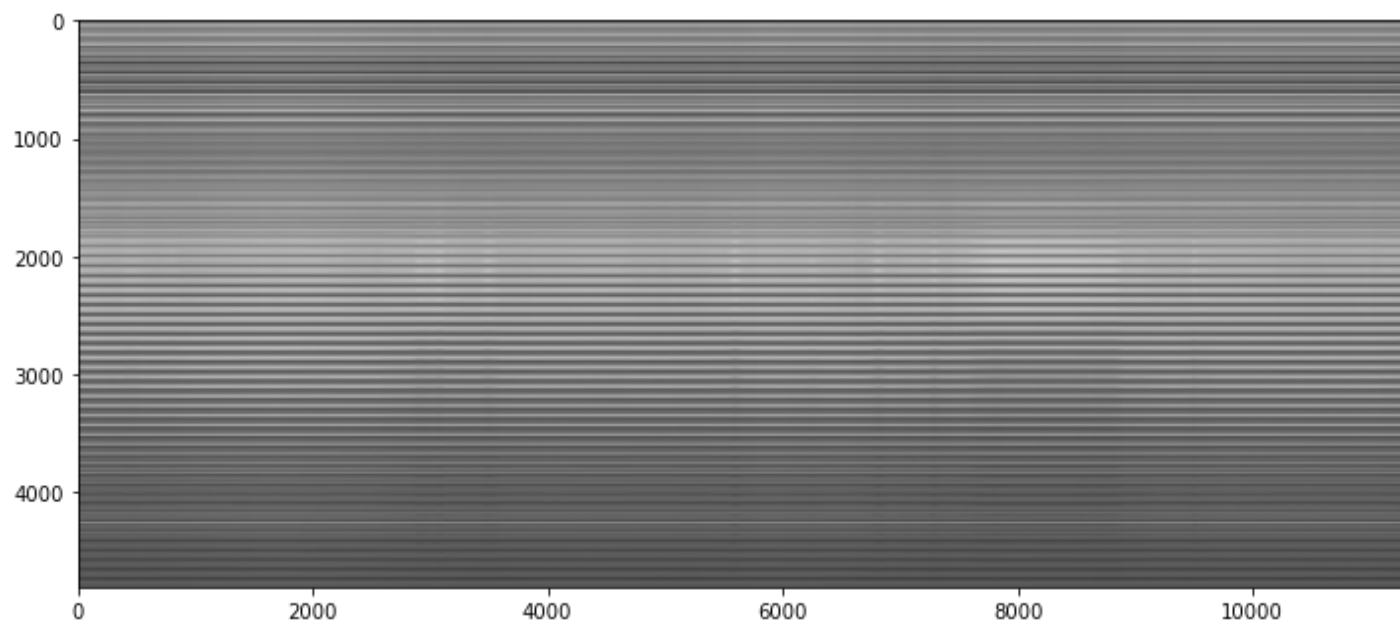
```
In [24]: low_rank = u @ np.diag(s) @ v
```

```
In [25]: low_rank.shape
```

```
Out[25]: (4800, 11300)
```

```
In [26]: plt.figure(figsize=(12, 12))
plt.imshow(low_rank, cmap='gray')
```

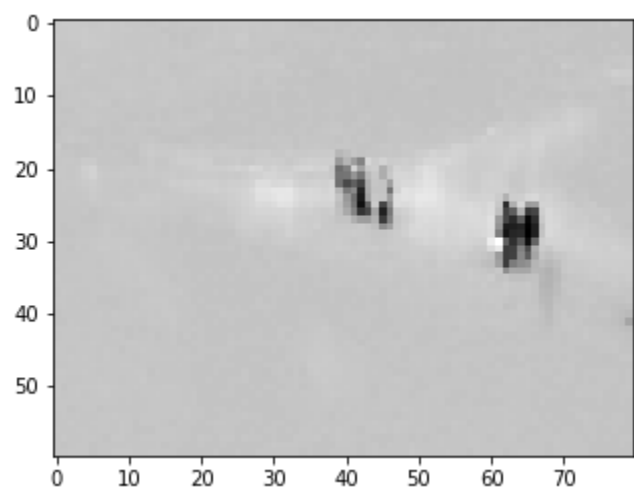
Out[26]: <matplotlib.image.AxesImage at 0x23e0e1fac48>



In [27]: `plt.imshow(np.reshape(low_rank[:,140], dims), cmap='gray');`



In [28]: `plt.imshow(np.reshape(M[:,550] - low_rank[:,550], dims), cmap='gray');`



Aproximação de classificação 1

```
In [29]: u, s, v = decomposition.randomized_svd(M, 1)
```

```
In [30]: u.shape, s.shape, v.shape
```

```
Out[30]: ((4800, 1), (1,), (1, 11300))
```

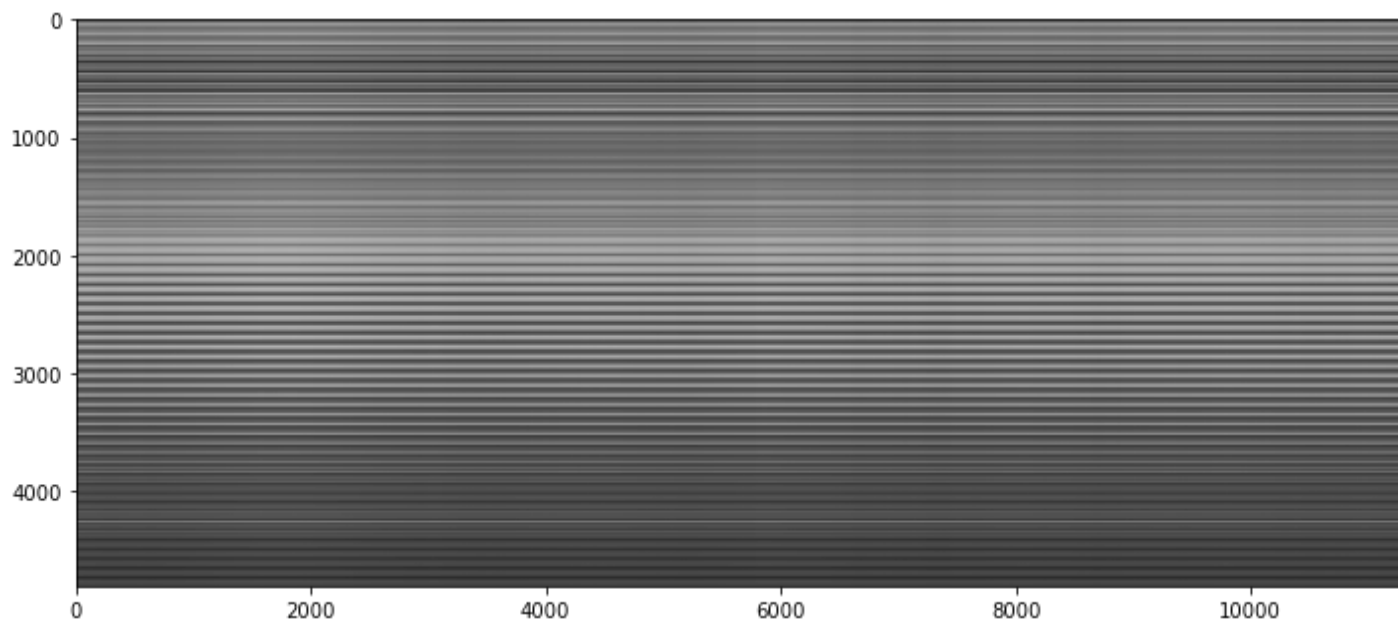
```
In [31]: low_rank = u @ np.diag(s) @ v
```

```
In [32]: low_rank.shape
```

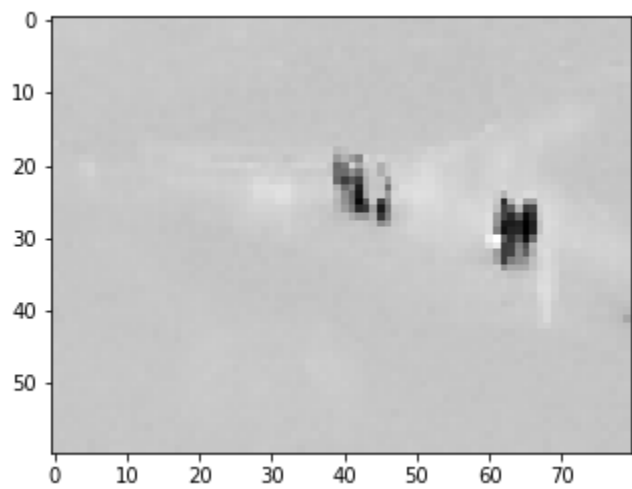
```
Out[32]: (4800, 11300)
```

```
In [33]: plt.figure(figsize=(12, 12))  
plt.imshow(low_rank, cmap='gray')
```

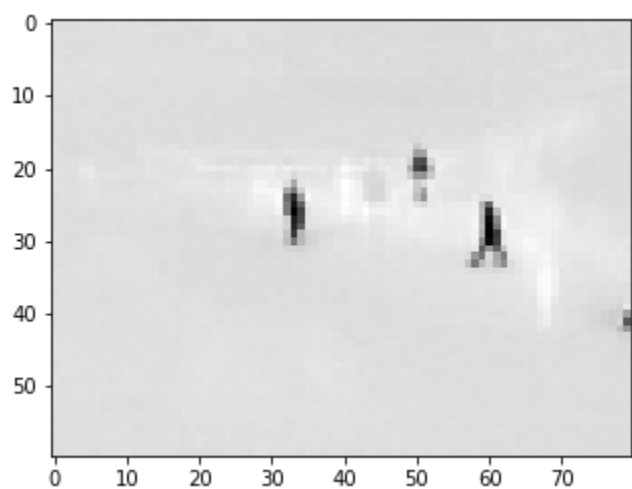
```
Out[33]: <matplotlib.image.AxesImage at 0x23e0e0f4048>
```



```
In [34]: plt.imshow(np.reshape(M[:,550] - low_rank[:,550], dims), cmap='gray');
```

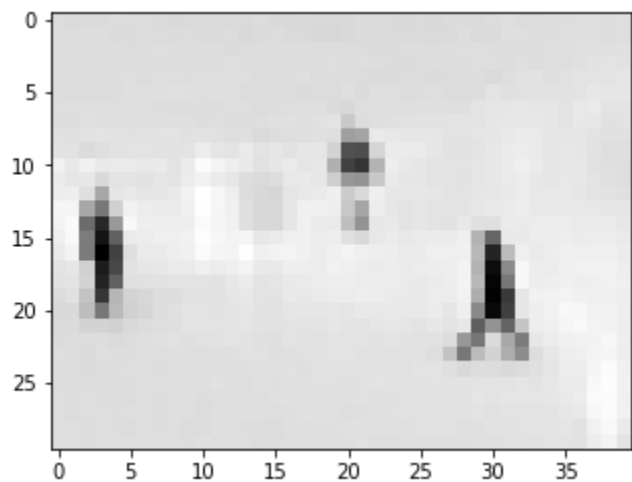


In [35]: `plt.imshow(np.reshape(M[:,140] - low_rank[:,140], dims), cmap='gray');`



Let's zoom in on the people:

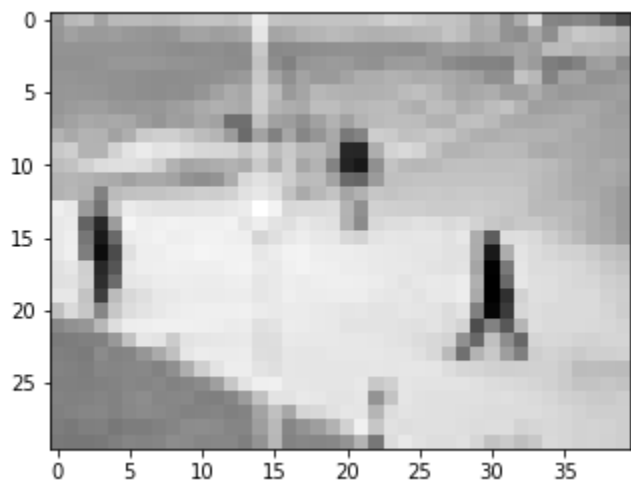
In [36]: `plt.imshow(np.reshape(M[:,140] - low_rank[:,140], dims)[10:40,30:70], cmap='gray');`



In [37]: `plt.imshow(np.reshape(M[:,140], dims)[10:40,30:70], cmap='gray')`

Out[37]: `<matplotlib.image.AxesImage at 0x23e0e1bd348>`







## Principal Component Analysis (PCA)

A Análise de Componentes Principais (ACP) ou Principal Component Analysis (PCA) é um procedimento matemático que utiliza uma transformação ortogonal (ortogonalização de vetores) para converter um conjunto de observações de variáveis possivelmente correlacionadas num conjunto de valores de variáveis linearmente não correlacionadas chamadas de componentes principais. O número de componentes principais é sempre menor ou igual ao número de variáveis originais. Os componentes principais são garantidamente independentes apenas se os dados forem normalmente distribuídos (conjuntamente). O PCA é sensível à escala relativa das variáveis originais. Dependendo da área de aplicação, o PCA é também conhecido como transformada de Karhunen-Loève (KLT) discreta, transformada de Hotelling ou decomposição ortogonal própria (POD).

Ao lidar com conjuntos de dados de alta dimensão, muitas vezes aproveitamos o fato de que os dados têm **baixa dimensionalidade intrínseca** para aliviar a maldição da dimensionalidade e da escala (talvez esteja em um subespaço de baixa dimensão ou em um variedade de baixa dimensão).

**Análise de componente principal** é útil para eliminar dimensões. A PCA clássica busca a melhor classificação -  $k$  estimativa  $L$  de  $M$  (minimizando  $|M - L|$  onde  $L$  tem classificação -  $k$ ). O SVD truncado faz esse cálculo! O PCA tradicional pode lidar com pequenos ruídos, mas é frágil em relação a observações grosseiramente corrompidas - até mesmo uma observação grosseiramente corrompida pode bagunçar significativamente a resposta. **PCA robusto** fatora uma matriz na soma de duas matrizes,  $M = L + S$ , onde  $M$  é a matriz original,  $L$  é **baixa classificação** e  $S$  é **escasso**. Isso é o que usaremos para o problema de remoção de fundo! **Baixa classificação** significa que a matriz tem muitas informações redundantes - neste caso, é o fundo, que é o mesmo em todas as cenas (fale sobre informações redundantes!). **Sparse** significa que a matriz tem quase zero entradas - neste caso, veja como a imagem do primeiro plano (as pessoas) está quase vazia. (No caso de dados corrompidos,  $S$  está capturando as entradas corrompidas).

## Aplicações de PCA robusto

- Fotos do **Reconhecimento facial**. O conjunto de dados aqui consiste em imagens de rostos de várias pessoas tiradas do mesmo ângulo, mas com diferentes iluminações.  PCA robusto  PCA robusto
- Indexação semântica latente:  $L$  captura palavras comuns usadas em todos os documentos, enquanto  $S$  captura as poucas palavras-chave que melhor distinguem cada documento dos outros
- Classificação e filtragem colaborativa: uma pequena parte das classificações disponíveis pode ser ruidosa e até mesmo adulterada (consulte [Netflix RAD - Outlier Detection on Big Data](#) no blog oficial netflix)

## A norma L1 que induz esparsidade

A esfera unitária  $\|x\|_1 = 1$  é um diamante na norma L1. Seus extremos são os cantos:



([Fonte](#))

Uma perspectiva semelhante é olhar para os *contornos* da função de perda:  L2 norm ([Fonte](#))

## Problema de otimização

O PCA robusto pode ser escrito:

$$\begin{aligned} & \text{minimize } \|L\|_* + \lambda \|S\|_1 \\ & \text{subject to } L + S = M \end{aligned}$$

Onde:

- $\|\cdot\|_1$  é a **norma L1**. Minimizar a [norma L1](#) resulta em valores esparsos. Para uma matriz, a norma L1 é igual à [norma de coluna máxima absoluta](#).
- $\|\cdot\|_*$  é a **norma nuclear**, que é a norma L1 dos valores singulares. Tentar minimizar isso resulta em valores singulares esparsos -> classificação baixa.

## Implementando um algoritmo de um artigo

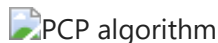
### Fontes

Será utilizado o **algoritmo de busca do componente primário** geral deste [artigo robusto do PCA](#) (Candes, Li, Ma, Wright), na forma específica de **Preenchimento da Matriz por meio do Método ALM Inexato** encontrado na [seção 3.1 deste documento](#) (Lin, Chen, Ma). Também menciono as implementações encontradas [aqui](#) e [aqui](#).

### The Good Parts

A seção 1 de Candes, Li, Ma, Wright é bem escrita, e a seção 5 Algoritmos é de principal interesse. **Não se faz necessário saber matemática ou compreender as provas para implementar um algoritmo de um papel.** Se faz necessário tentar coisas diferentes e vasculhar os recursos para obter informações úteis. Este campo tem mais pesquisadores teóricos e menos conselhos pragmáticos.

O algoritmo é mostrado na página 29:



Definições necessárias de  $\mathcal{S}$ , o operador de redução, e  $\mathcal{D}$ , o operador de limite de valor singular:



A seção 3.1 de [Chen, Lin, Ma] () contém uma variação mais rápida diso:



E a Seção 4 tem alguns detalhes de implementação muito úteis sobre quantos valores singulares calcular (bem como escolher os valores dos parâmetros):

## Mais fontes para aprender mais sobre a teoria:

- Otimização convexa por Stephen Boyd (Stanford Prof):
  - [Vídeos OpenEdX -Jupyter Notebooks](#)
- Método de direção alternada de multiplicadores (mais [Stephen Boyd](#))

## PCA robusto (via perseguição de componente primário)

### Métodos auxiliares

Será utilizada a biblioteca [Fast Randomized PCA do Facebook](#)..

```
In [38]: from scipy import sparse
from sklearn.utils.extmath import randomized_svd
import fbpc
```

```
In [39]: TOL=1e-9
MAX_ITERS=3
```

```
In [40]: def converged(Z, d_norm):
    err = np.linalg.norm(Z, 'fro') / d_norm
    print('error: ', err)
    return err < TOL
```

```
In [41]: def shrink(M, tau):
    S = np.abs(M) - tau
    return np.sign(M) * np.where(S>0, S, 0)
```

```
In [42]: def _svd(M, rank): return fbpc.pca(M, k=min(rank, np.min(M.shape)), raw=True)
```

```
In [43]: def norm_op(M): return _svd(M, 1)[1][0]
```

```
In [44]: def svd_reconstruct(M, rank, min_sv):
    u, s, v = _svd(M, rank)
    s -= min_sv
    nnz = (s > 0).sum()
    return u[:, :nnz] @ np.diag(s[:nnz]) @ v[:, :nnz], nnz
```

```
In [45]: def pcpx(X, maxiter=10, k=10): # refactored
    m, n = X.shape
    trans = m < n
    if trans: X = X.T; m, n = X.shape

    lamda = 1/np.sqrt(m)
    op_norm = norm_op(X)
    Y = np.copy(X) / max(op_norm, np.linalg.norm(X, np.inf) / lamda)
    mu = k*1.25/op_norm; mu_bar = mu * 1e7; rho = k * 1.5
```

```

d_norm = np.linalg.norm(X, 'fro')
L = np.zeros_like(X); sv = 1

examples = []

for i in range(maxiter):
    print("rank sv:", sv)
    X2 = X + Y/mu

    S = shrink(X2 - L, lamda/mu)

    L, svp = svd_reconstruct(X2 - S, sv, 1/mu)

    sv = svp + (1 if svp < sv else round(0.05*n))

    Z = X - L - S
    Y += mu*Z; mu *= rho

    examples.extend([S[140,:], L[140,:]])

    if m > mu_bar: m = mu_bar
    if converged(Z, d_norm): break

if trans: L=L.T; S=S.T
return L, S, examples

```

O algoritmo novamente (página 29 de [Candes, Li, Ma e Wright](#))

 PCP algorithm

## Results

In [46]:

```

m, n = M.shape
round(m * .05)

```

Out[46]: 240

In [47]:

```

L, S, examples = pcp(M, maxiter=5, k=10)

```

```

rank sv: 1
error: 0.13722384924500092
rank sv: 241
error: 0.047680855491970785
rank sv: 51
error: 0.005859082808973587
rank sv: 291
error: 0.0005685770322933045
rank sv: 531
error: 2.498632551722058e-05

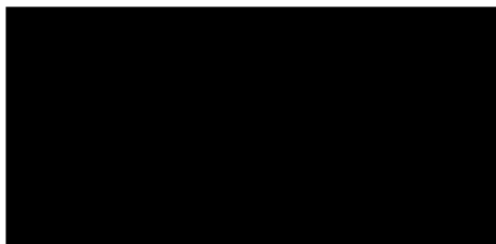
```

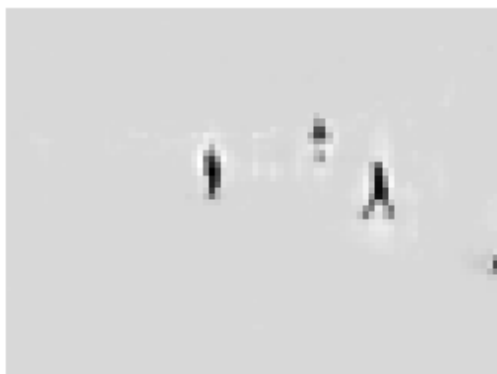
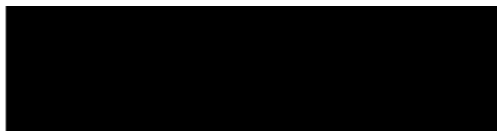
In [48]:

```

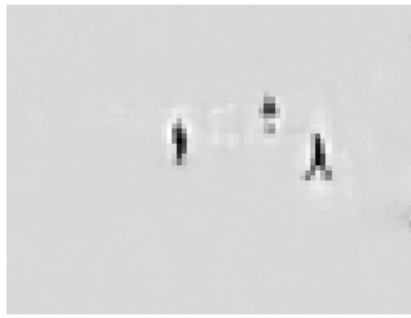
plots(examples, dims, rows=5)

```



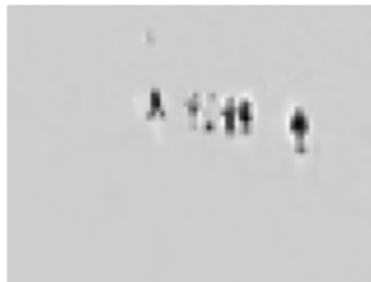


In [49]: `f = plt_images(M, S, L, [140], dims)`



```
In [50]: np.save("high_res_L.npy", L)
         np.save("high_res_S.npy", S)
```

```
In [51]: f = plt_images(M, S, L, [0, 100, 1000], dims)
```



Como pode ser observado, extrair um pouco do primeiro plano é mais fácil do que identificar o fundo. Para obter o plano de fundo com precisão, é preciso remover todo o primeiro plano, não apenas partes dele

## Fatoração LU

Ambos, `fbpca` e o próprio método escrito `randomized_range_finder` usaram fatoração LU, que fatora uma matriz no produto de uma matriz triangular inferior e uma matriz triangular superior.

## Eliminação gaussiana

Esta seção é baseada nas aulas 20-22 em Trefethen.

Caso, você não está familiarizado com a eliminação gaussiana ou precisa de uma atualização, segue uma fonte recomendada, [vídeo da Khan Academy](#).

Vamos usar a Eliminação Gaussiana manualmente para revisar:

A =

$$\begin{pmatrix} 1 & -2 & -2 & -3 \\ 3 & -9 & 0 & -9 \\ -1 & 2 & 4 & 7 \\ -3 & -6 & 26 & 2 \end{pmatrix}$$

Resposta:

$$LU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -3 & 4 & -2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -2 & -2 & -3 \\ 0 & -3 & 6 & 0 \\ 0 & 0 & 2 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

O exemplo acima é das palestras 20, 21 de Trefethen.

**Eliminação Gaussiana** transforma um sistema linear em um triangular superior aplicando transformações lineares à esquerda. É a *triangularização triangular*.

$$L_{m-1} \dots L_2 L_1 A = U$$

L é *unidade triangular inferior* : todas as entradas diagonais são 1

```
In [52]: def LU(A):
          U = np.copy(A)
          m, n = A.shape
          L = np.eye(n)
          for k in range(n-1):
              for j in range(k+1, n):
                  L[j, k] = U[j, k]/U[k, k]
                  U[j, k:n] -= L[j, k] * U[k, k:n]
          return L, U
```

```
In [53]: A = np.array([[2, 1, 1, 0], [4, 3, 3, 1], [8, 7, 9, 5], [6, 7, 9, 8]]).astype(np.float)
```

```
In [54]: L, U = LU(A)
```

```
In [55]: np.allclose(A, L @ U)
```

Out[55]: True

A fatora  o LU    til. Resolver  $Ax = b$  torna-se  $LUx = b$ :

1. encontre  $A = LU$
2. resolva  $Ly = b$
3. resolva  $Ux = y$

## Trabalho computacional

O Trabalho para a Eliminação Gaussiana é calculada como:  $2 \cdot \frac{1}{3}n^3$

### Memória

Acima, foram criadas duas novas matrizes,  $L$  e  $U$ . No entanto, podemos armazenar os valores de  $L$  e  $U$  em uma matriz  $A$  (sobrescrevendo a matriz original). Visto que a diagonal de  $L$  é toda 1 s, ela não precisa ser armazenada. Fazer fatorações ou cálculos **local** é uma técnica comum em álgebra linear numérica para economizar memória. Observação: você não gostaria de fazer isso se precisasse usar sua matriz original  $A$  novamente no futuro.

Consider the matrix

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$$

```
In [56]: A = np.array([[1e-20, 1], [1,1]])
```

À mão, use a Eliminação Gaussiana para calcular o que  $L$  e  $U$  são:

### Resposta

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{-20} \end{bmatrix}$$

```
In [57]: np.set_printoptions(suppress=True)
```

```
In [58]: #Exercise:
L1 = np.array([[1, 0], [1e20, 1]])
U1 = np.array([[10e-20, 1], [0, 1-10e20]])
```

```
In [59]: L2, U2 = LU(A)
```

```
In [60]: L2, U2
```

```
Out[60]: (array([[1.e+00, 0.e+00],
                [1.e+20, 1.e+00]]),
          array([[ 1.e-20,  1.e+00],
                [ 0.e+00, -1.e+20]]))
```

```
In [61]: np.allclose(L1, L2)
```

```
Out[61]: True
```

```
In [62]: np.allclose(U1, U2)
```



Out[62]: False

```
In [63]: np.allclose(A, L2 @ U2)
```

Out[63]: False

Esta é a motivação para  $LU$  fatoração **com pivotamento**.

Isso também ilustra que a fatoração LU é *estável*, mas não *estável para trás*. (Mesmo com pivotamento parcial, LU é "explosivamente instável" para certas matrizes, mas estável na prática)

## Estabilidade

Um algoritmo  $\hat{f}$  para um problema  $f$  é **estável** se para cada  $x$ ,

$$\frac{\|\hat{f}(x) - f(y)\|}{\|f(y)\|} = \mathcal{O}(\varepsilon_{máquina})$$

por cerca de  $y$  com

$$\frac{\|y - x\|}{\|x\|} = \mathcal{O}(\varepsilon_{máquina})$$

**Um algoritmo estável fornece quase a resposta certa para quase a pergunta certa**

Para traduzir isso:

- pergunta certa:  $x$
- a pergunta quase certa:  $y$
- resposta certa:  $f$
- resposta certa para a pergunta quase certa:  $f(y)$

## Estabilidade reversa

A estabilidade reversa é **mais forte** e **mais simples** do que a estabilidade. Um algoritmo  $\hat{f}$  para um problema  $f$  é **estável reversamente** se para cada  $x$ ,

$$\hat{f}(x) = f(y)$$

por cerca de  $y$  com

$$\frac{\|y - x\|}{\|x\|} = \mathcal{O}(\varepsilon_{máquina})$$

**Um algoritmo estável reversamente dá exatamente a resposta certa para quase a pergunta certa**

Resumindo:

- pergunta certa:  $x$
- a pergunta quase certa:  $y$
- resposta certa:  $f$
- resposta certa para a pergunta quase certa:  $f(y)$