3. Background Removal with Robust PCA

Our goal today:

Getting Started

Let's use the real video 003 dataset from BMC 2012 Background Models Challenge Dataset (http://bmc.iut-auvergne.com/?page_id=24)

Other sources of datasets:

- Human Activity Video Datasets (https://www.cs.utexas.edu/~chaoyeh/web_action_data/dataset_list.html)
- Background Subtraction Website (https://sites.google.com/site/backgroundsubtraction/test-sequences) (a few links on this site are broken/outdated, but many work)

Import needed libraries:

In [2]:
```
import moviepy.editor as mpe
# from IPython.display import display
from glob import glob
```

In [3]:
```
import sys, os
import numpy as np
import scipy
```

In [4]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [5]:
```
# MAX_ITERS = 10
TOL = 1.0e-8
```

In [7]:
```
import requests

c = 0
def download_file(url,local_filename):
    r = requests.get(url, stream=True)
    with open(local_filename, 'wb') as f:
        for chunk in r.iter_content(chunk_size=1024):
            if chunk: # filter out keep-alive new chunks
                f.write(chunk)
                #f.flush() commented by recommendation from J.F.Sebast
ian
    return local_filename
```

In [11]:
```
download_file('http://bmc.iut-auvergne.com/wp-content/ressources/video
s/evaluation/Video_003.zip','Video_003.zip')
```

Out[11]:
```
'Video_003.zip'
```

In [15]:
```
ls
```

```
0. Course Logistics.ipynb
1. Why are we here?.ipynb
2. Topic Modeling with NMF and SVD.ipynb
3. Background Removal with Robust PCA.ipynb
4. Compressed Sensing of CT Scans with Robust Regression.ipynb
5. Health Outcomes with Linear Regression.ipynb*
6. How to Implement Linear Regression.ipynb
7. PageRank with Eigen Decompositions.ipynb
8. Implementing QR Factorization.ipynb*
convolution-intro.ipynb
graddesc.xlsm
gradient-descent-intro.ipynb
Homework 1.ipynb
Homework 2.ipynb
Homework 3.ipynb
images/
Project_ideas.txt
Video_003.zip
```

In [14]:
```
!rm Video_003.avi
```

```
rm: cannot remove 'Video_003.avi': No such file or directory
```

In [16]:
```
! unzip Video_003.zip
```

```
Archive:  Video_003.zip
  inflating: Video_003.avi
warning:  Video_003.zip appears to use backslashes as path separator
s
  inflating: private_truth/1.bmp
  inflating: private_truth/10.bmp
  inflating: private_truth/11.bmp
  inflating: private_truth/12.bmp
  inflating: private_truth/13.bmp
  inflating: private_truth/14.bmp
  inflating: private_truth/15.bmp
  inflating: private_truth/2.bmp
  inflating: private_truth/3.bmp
  inflating: private_truth/4.bmp
  inflating: private_truth/5.bmp
  inflating: private_truth/6.bmp
  inflating: private_truth/7.bmp
  inflating: private_truth/8.bmp
  inflating: private_truth/9.bmp
  inflating: private_truth/color/color1.bmp
  inflating: private_truth/color/color10.bmp
  inflating: private_truth/color/color11.bmp
  inflating: private_truth/color/color12.bmp
  inflating: private_truth/color/color13.bmp
  inflating: private_truth/color/color14.bmp
  inflating: private_truth/color/color15.bmp
  inflating: private_truth/color/color2.bmp
  inflating: private_truth/color/color3.bmp
  inflating: private_truth/color/color4.bmp
  inflating: private_truth/color/color5.bmp
  inflating: private_truth/color/color6.bmp
  inflating: private_truth/color/color7.bmp
  inflating: private_truth/color/color8.bmp
  inflating: private_truth/color/color9.bmp
  inflating: private_truth/sequence.dat
```

In [17]:
```
video = mpe.VideoFileClip("Video_003.avi")
```

In [18]:
```
video.subclip(0,50).ipython_display(width=300)
```

```
100%|██████████| 350/351 [00:00<00:00, 1549.11it/s]
```

Out[18]:

In [19]:
```
video.duration
```

Out[19]:
```
113.57
```

Helper Methods

In [21]:
```
def create_data_matrix_from_video(clip, k=5, scale=50):
    return np.vstack([scipy.misc.imresize(rgb2gray(clip.get_frame(i/fl
oat(k))).astype(int),
                      scale).flatten() for i in range(k * int(clip.dur
ation))]).T
```

In [20]:
```
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
```

In [22]:
```
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 [23]:
```
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")
```

Load and view the data

An image from 1 moment in time is 60 pixels by 80 pixels (when scaled). We can *unroll* that picture into a single tall column. So instead of having a 2D picture that is 60×80, we have a $1 \times 4,800$ column

This isn't very human-readable, but it's handy because it lets us stack the images from different times on top of one another, to put a video all into 1 matrix. If we took the video image every tenth of a second for 113 seconds (so 11,300 different images, each from a different point in time), we'd have a 11300×4800 matrix, representing the video!

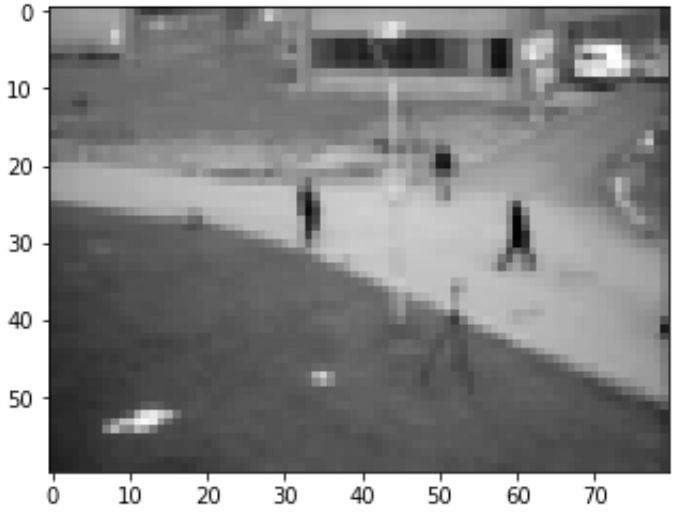
In [24]:
```
scale = 25   # Adjust scale to change resolution of image
dims = (int(240 * (scale/100)), int(320 * (scale/100)))
```

In [25]:
```
M = create_data_matrix_from_video(video, 100, scale)
# M = np.load("high_res_surveillance_matrix.npy")
```

In [26]:
```
print(dims, M.shape)
```

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

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



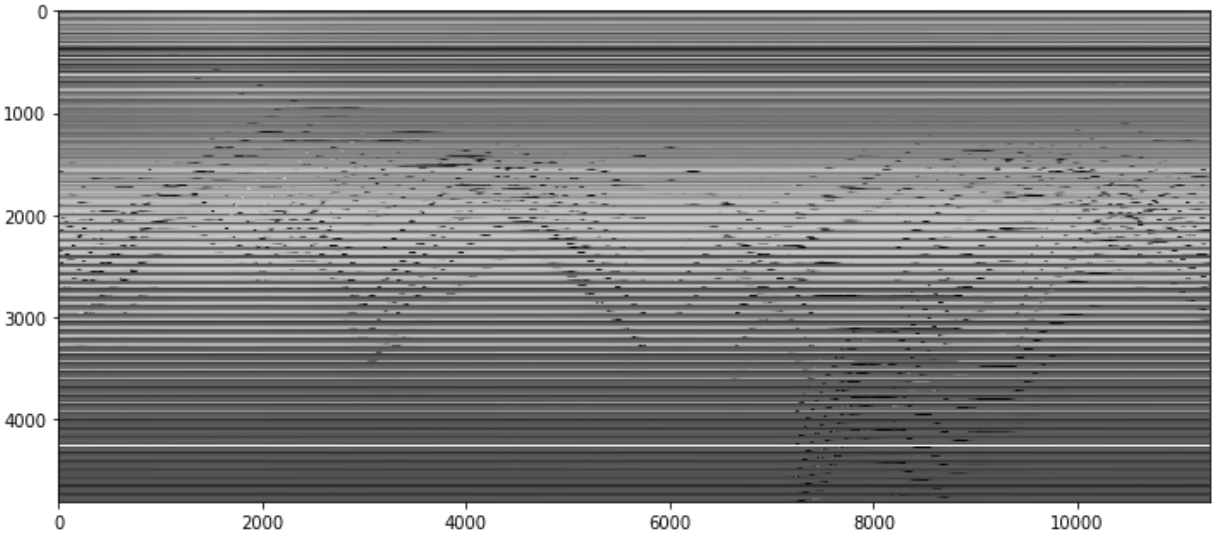
Since `create_data_from_matrix` is somewhat slow, we will save our matrix. In general, whenever you have slow pre-processing steps, it's a good idea to save the results for future use.

In [28]:
```
np.save("low_res_surveillance_matrix.npy", M)
```

Note: High-res M is too big to plot, so only run the below with the low-res version

In [29]:
```
plt.figure(figsize=(12, 12))
plt.imshow(M, cmap='gray')
```

Out[29]:
```
<matplotlib.image.AxesImage at 0x7fa0e2ed9898>
```



Questions: What are those wavy black lines? What are the horizontal lines?

In [30]:
```
plt.imsave(fname="image1.jpg", arr=np.reshape(M[:,140], dims), cmap='gray')
```

SVD

Review from Lesson 2:

- What kind of matrices are returned by SVD?
- What is a way to speed up truncated SVD?

A first attempt with SVD

In [31]:
```
from sklearn import decomposition
```

In [32]:
```
u, s, v = decomposition.randomized_svd(M, 2)
```

In [33]:
```
u.shape, s.shape, v.shape
```

Out[33]:
```
((4800, 2), (2,), (2, 11300))
```

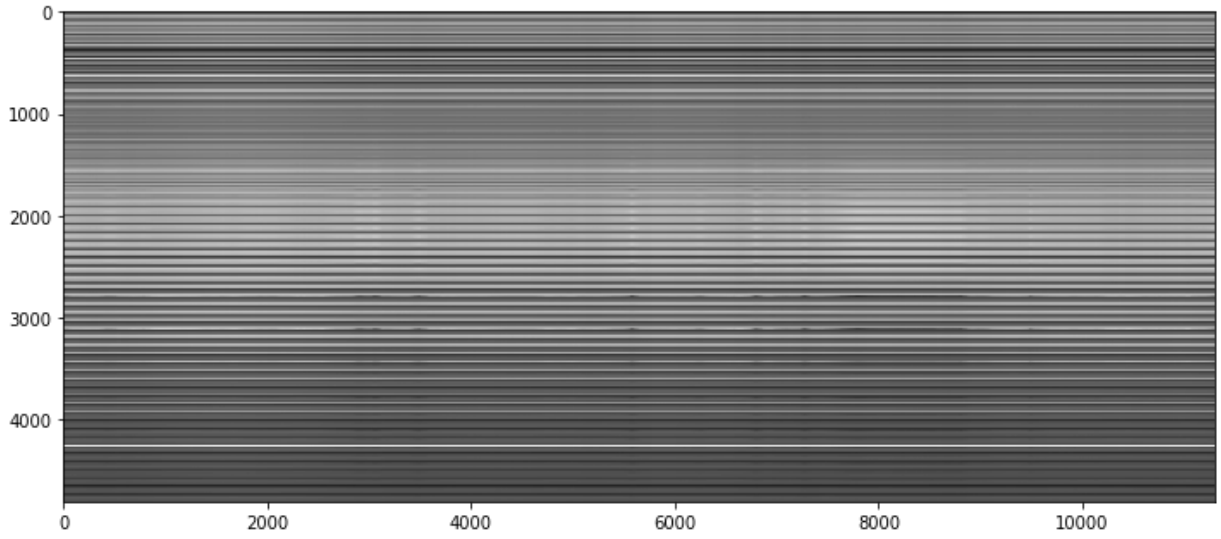
In [34]:
```
low_rank = u @ np.diag(s) @ v
```

In [35]:
```
low_rank.shape
```

Out[35]:
```
(4800, 11300)
```

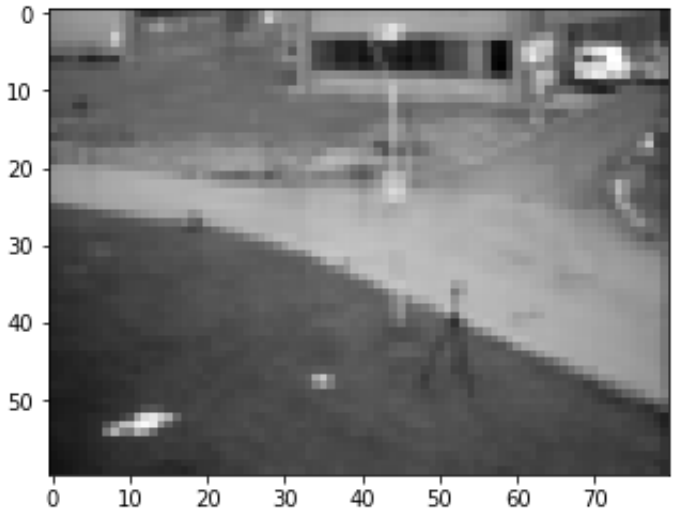
In [36]:
```
plt.figure(figsize=(12, 12))
plt.imshow(low_rank, cmap='gray')
```

Out[36]: <matplotlib.image.AxesImage at 0x7fa0af87d6d8>

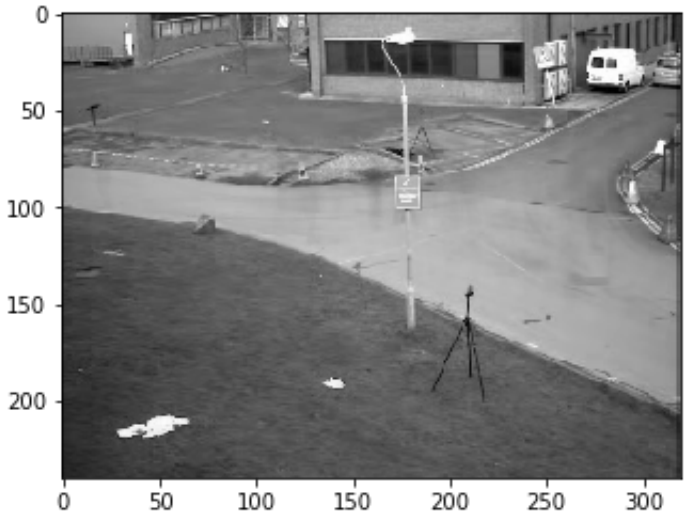


The below images were created with high-res data. Very slow to process:

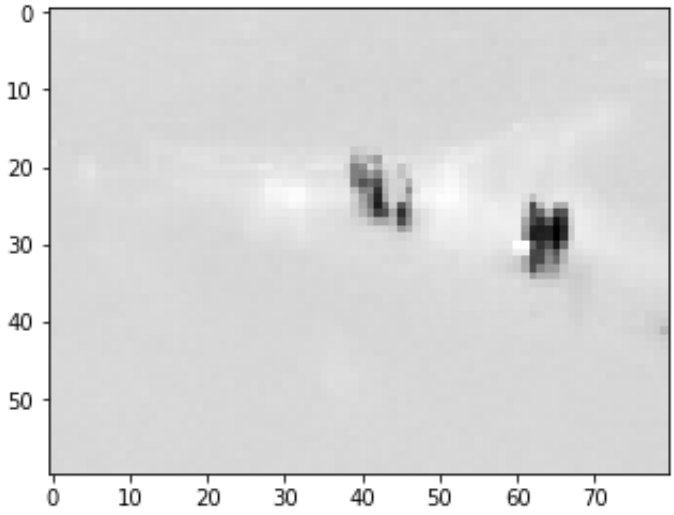
In [37]:
```
plt.imshow(np.reshape(low_rank[:,140], dims), cmap='gray');
```



In [18]:
```
plt.imshow(np.reshape(low_rank[:,140], dims), cmap='gray');
```



In [21]:
```
plt.imshow(np.reshape(M[:,550] - low_rank[:,550], dims), cmap='gray');
```



Rank 1 Approximation

In [39]:
```
u, s, v = decomposition.randomized_svd(M, 1)
```

In [40]:
```
u.shape, s.shape, v.shape
```

Out[40]: ((4800, 1), (1,), (1, 11300))

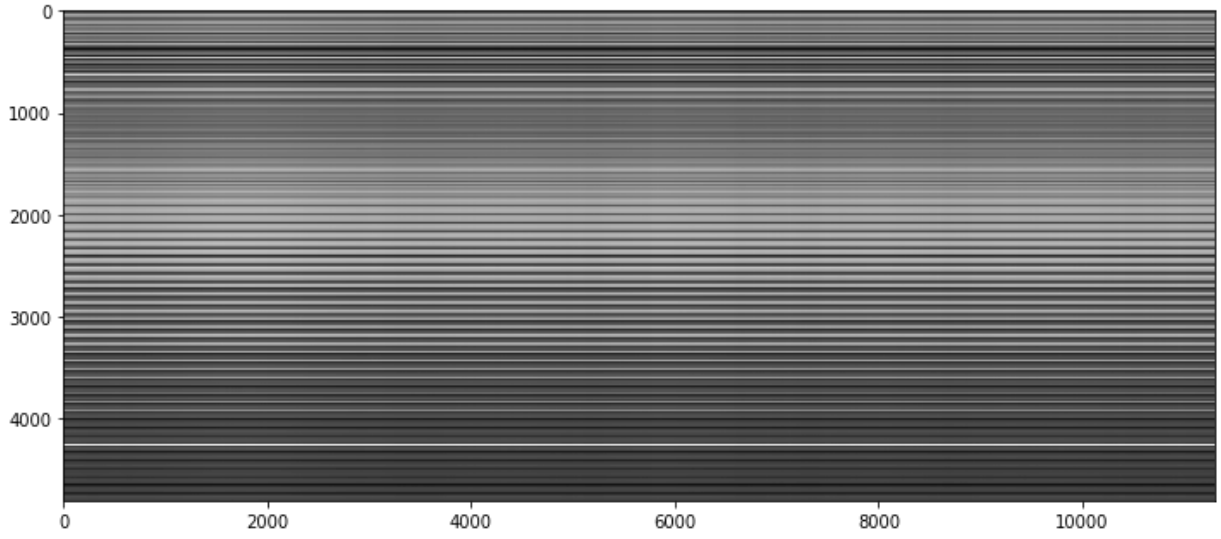
In [41]:
```
low_rank = u @ np.diag(s) @ v
```

In [42]:
```
low_rank.shape
```

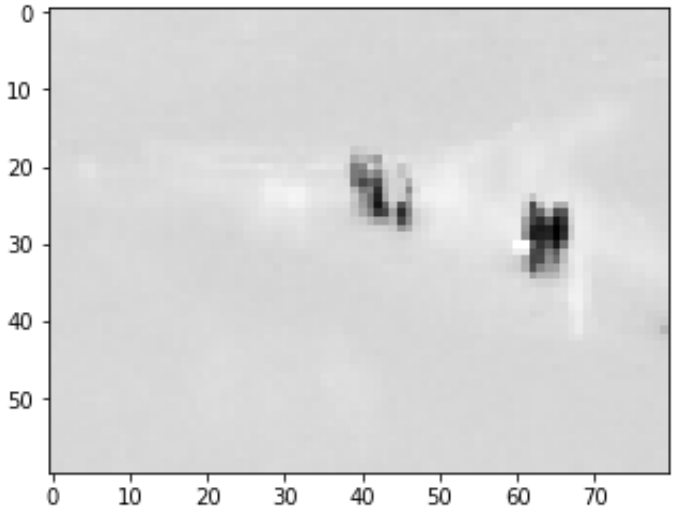
Out[42]: (4800, 11300)

In [43]:
```
plt.figure(figsize=(12, 12))
plt.imshow(low_rank, cmap='gray')
```

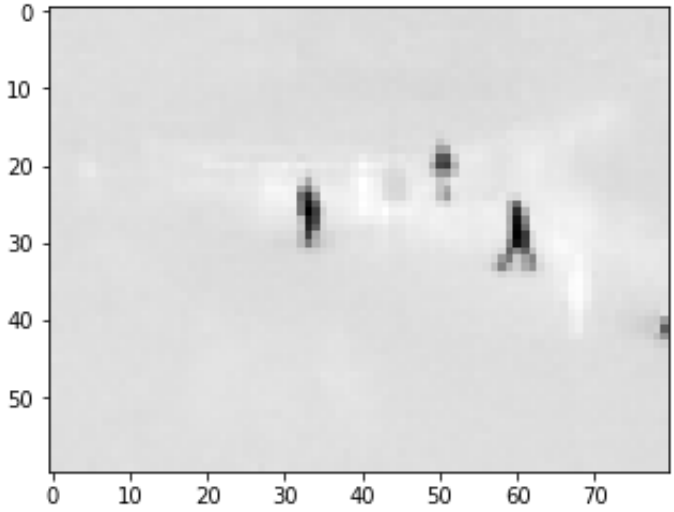
Out[43]: <matplotlib.image.AxesImage at 0x7fa0c961eda0>



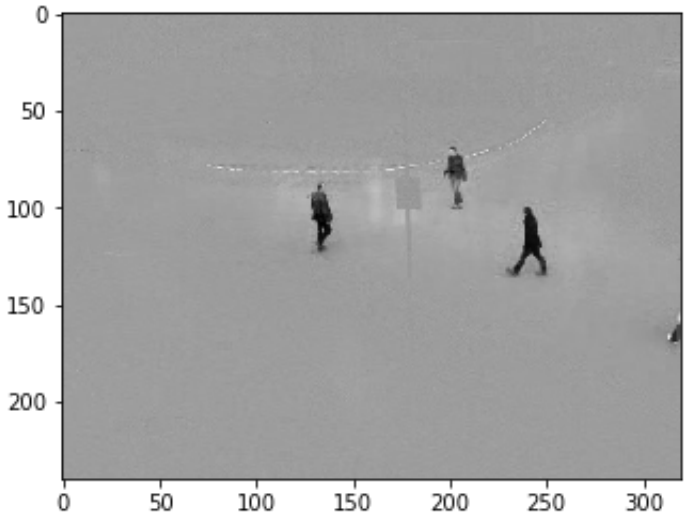
In [44]:
```
plt.imshow(np.reshape(M[:,550] - low_rank[:,550], dims), cmap='gray');
```



In [48]:
```
plt.imshow(np.reshape(M[:,140] - low_rank[:,140], dims), cmap='gray');
```

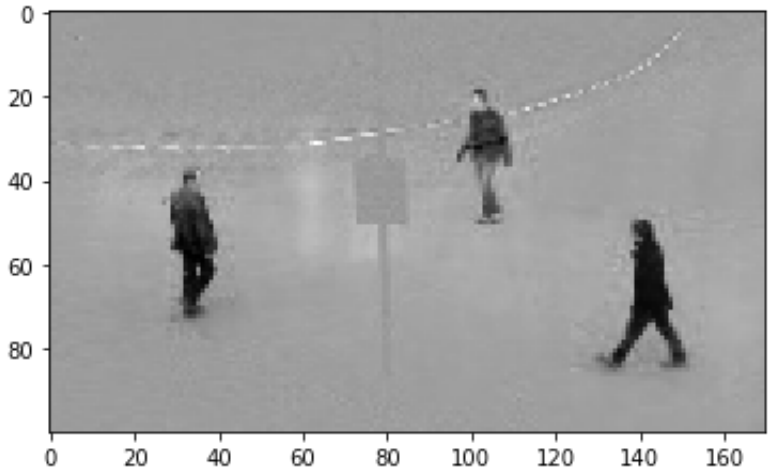


In [35]:
```
plt.imshow(np.reshape(M[:,140] - low_rank[:,140], dims), cmap='gray');
```



Let's zoom in on the people:

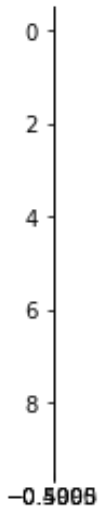
In [36]:
```
plt.imshow(np.reshape(M[:,140] - low_rank[:,140], dims)[50:150,100:270
], cmap='gray');
```



This is amazing for such a simple approach! We get somewhat better results through a more complicated algorithm below.

In [46]:
```
plt.imshow(np.reshape(M[:,140], dims)[50:150,100:270], cmap='gray')
```

Out[46]:
```
<matplotlib.image.AxesImage at 0x7fa0c9584d30>
```



Principal Component Analysis (PCA)

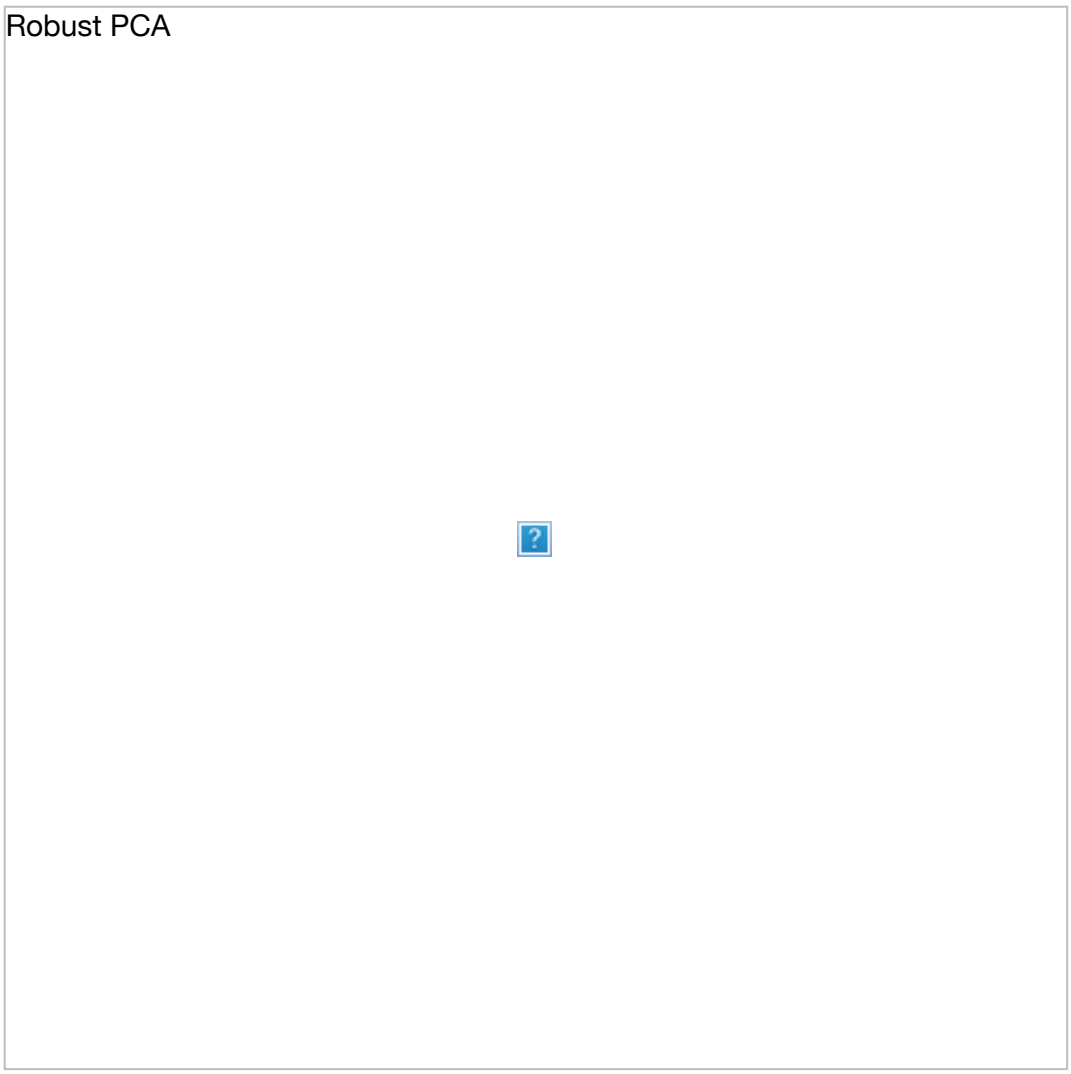
When dealing with high-dimensional data sets, we often leverage on the fact that the data has **low intrinsic dimensionality** in order to alleviate the curse of dimensionality and scale (perhaps it lies in a low-dimensional subspace or lies on a low-dimensional manifold). Principal component analysis (http://setosa.io/ev/principal-component-analysis/) is handy for eliminating dimensions. Classical PCA seeks the best rank-k estimate L of M (minimizing $||M - L||$ where L has rank-k). Truncated SVD makes this calculation!

Traditional PCA can handle small noise, but is brittle with respect to grossly corrupted observations-- even one grossly corrupt observation can significantly mess up answer.

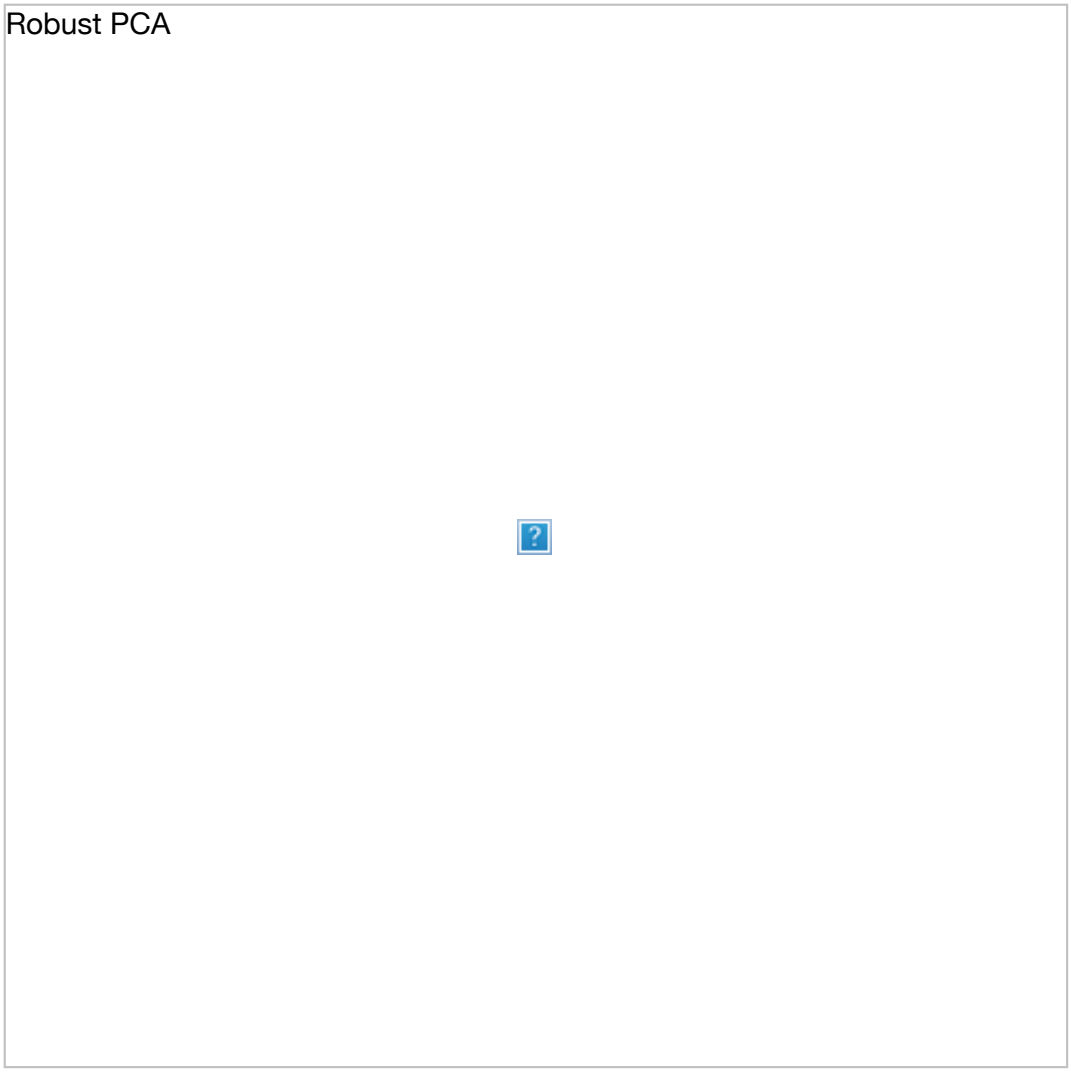
Robust PCA factors a matrix into the sum of two matrices, $M = L + S$, where M is the original matrix, L is **low-rank**, and S is **sparse**. This is what we'll be using for the background removal problem! **Low-rank** means that the matrix has a lot of redundant information-- in this case, it's the background, which is the same in every scene (talk about redundant info!). **Sparse** means that the matrix has mostly zero entries-- in this case, see how the picture of the foreground (the people) is mostly empty. (In the case of corrupted data, S is capturing the corrupted entries).

Applications of Robust PCA

- **Video Surveillance**
- **Face Recognition** photos from this excellent tutorial (https://jeankossaifi.github.io/tensorly/rpca.html). The dataset here consists of images of the faces of several people taken from the same angle but with different illumniations.



(Source: Jean Kossaifi (https://jeankossaifi.github.io/tensorly/rpca.html))

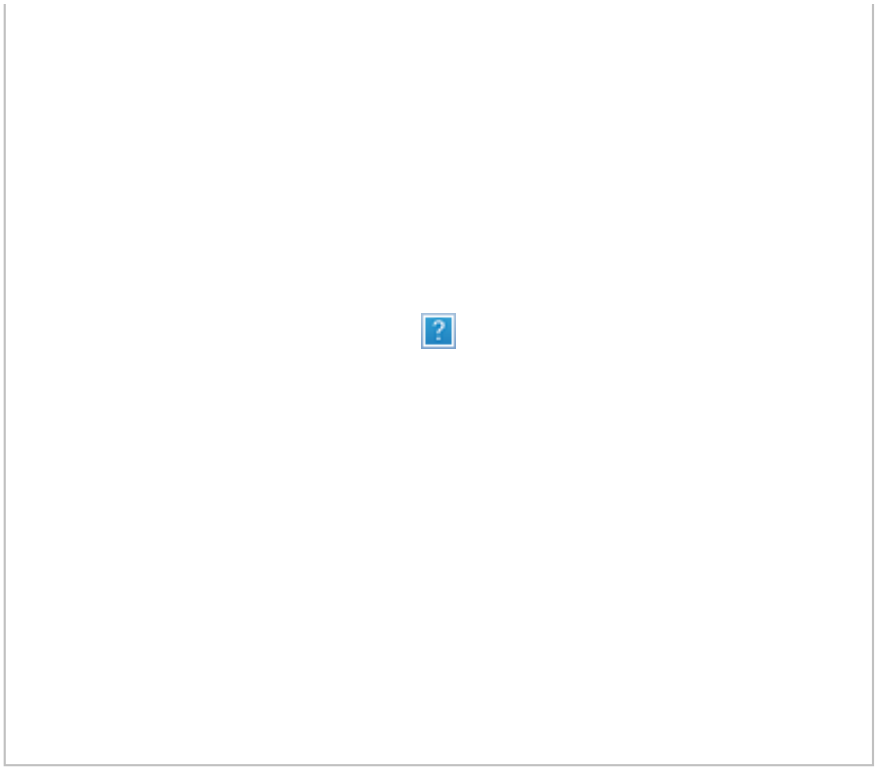


(Source: Jean Kossaifi (https://jeankossaifi.github.io/tensorly/rpca.html))

- Latent Semantic Indexing: L captures common words used in all documents while S captures the few key words that best distinguish each document from others
- Ranking and Collaborative Filtering: a small portion of the available rankings could be noisy and even tampered with (see Netflix RAD - Outlier Detection on Big Data (http://techblog.netflix.com/2015/02/rad-outlier-detection-on-big-data.html) on the official netflix blog)

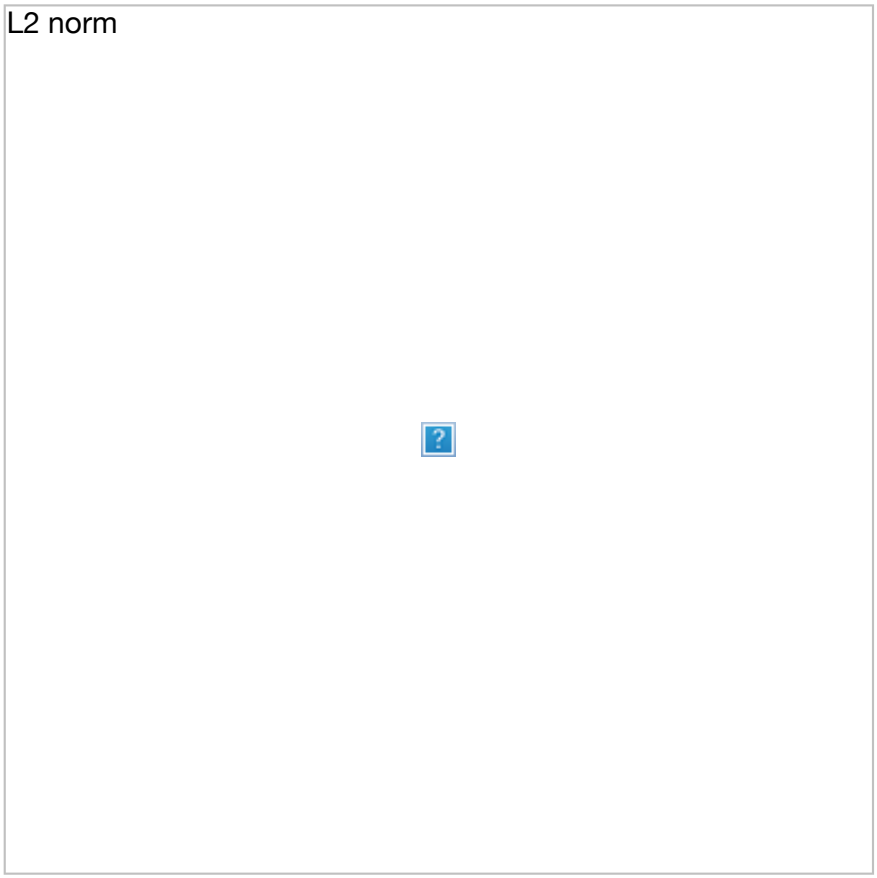
L1 norm induces sparsity

The unit ball $||x||_1 = 1$ is a diamond in the L1 norm. It's extrema are the corners:



(Source (https://www.quora.com/Why-is-L1-regularization-supposed-to-lead-to-sparsity-than-L2))

A similar perspective is to look at the *contours* of the loss function:



(Source (https://www.quora.com/Why-is-L1-regularization-better-than-L2-regularization-provided-that-all-Norms-are-equivalent))

Optimization Problem

Robust PCA can be written:

$$minimize\ \|L\|_* + \lambda\|S\|_1$$
$$subject\ to\ L + S = M$$

where:

- $\|\cdot\|_1$ is the **L1 norm**. Minimizing the L1 norm (https://medium.com/@shiyan/l1-norm-regularization-and-sparsity-explained-for-dummies-5b0e4be3938a) results in sparse values. For a matrix, the L1 norm is equal to the maximum absolute column norm (https://math.stackexchange.com/questions/519279/why-is-the-matrix-norm-a-1-maximum-absolute-column-sum-of-the-matrix).
- $\|\cdot\|_*$ is the **nuclear norm**, which is the L1 norm of the singular values. Trying to minimize this results in sparse singular values --> low rank.

Implementing an algorithm from a paper

Source

We will use the general **primary component pursuit algorithm** from this Robust PCA paper (https://arxiv.org/pdf/0912.3599.pdf) (Candes, Li, Ma, Wright), in the specific form of **Matrix Completion via the Inexact ALM Method** found in section 3.1 of this paper (https://arxiv.org/pdf/1009.5055.pdf) (Lin, Chen, Ma). I also referenced the implemenations found here (https://github.com/shriphani/robust_pcp/blob/master/robust_pcp.py) and here (https://github.com/dfm/pcp/blob/master/pcp.py).

The Good Parts

Section 1 of Candes, Li, Ma, Wright is nicely written, and section 5 Algorithms is our key interest. **You don't need to know the math or understand the proofs to implement an algorithm from a paper.** You will need to try different things and comb through resources for useful tidbits. This field has more theoretical researchers and less pragmatic advice. It took months to find what I needed and get this working.

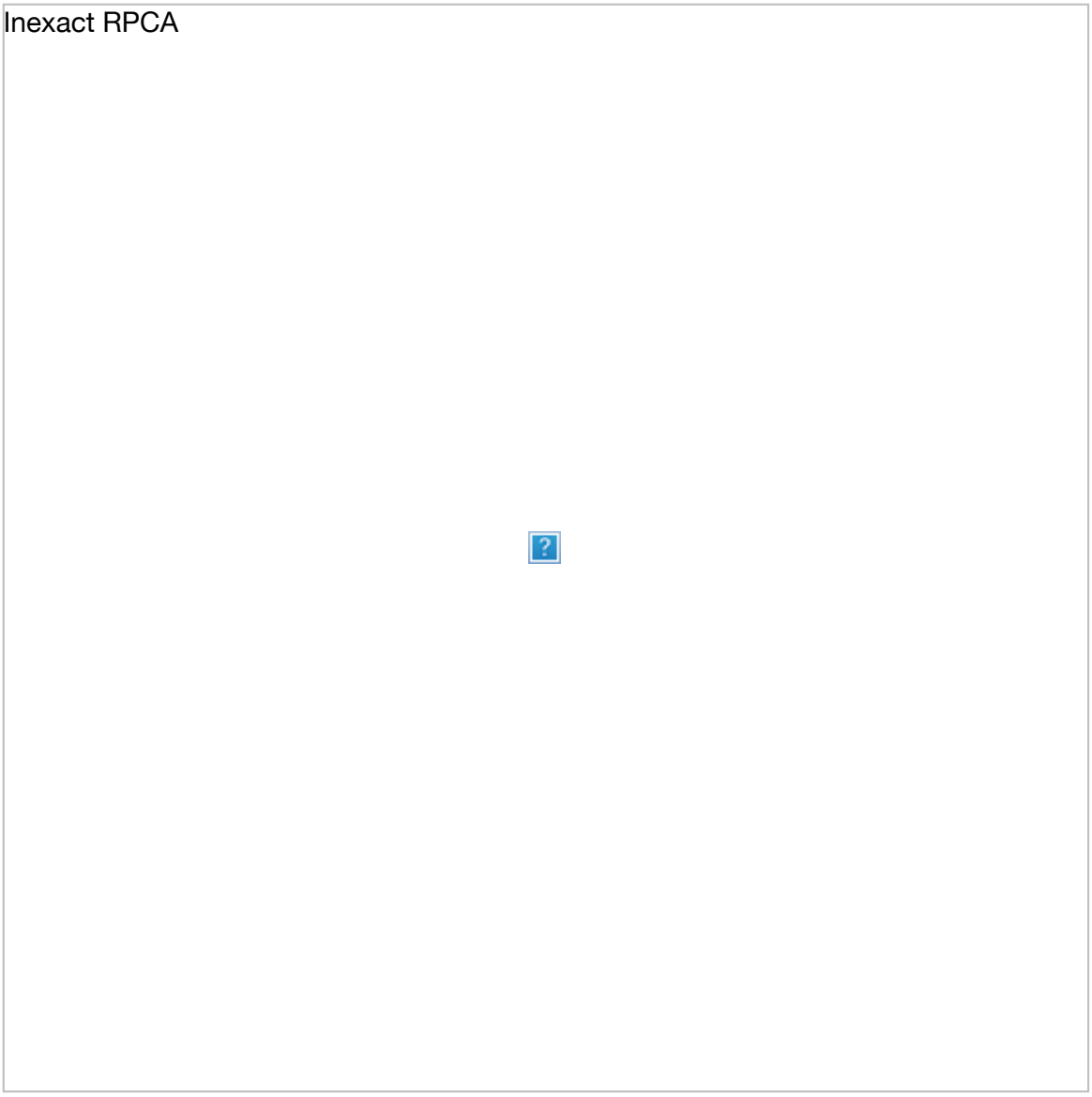
The algorithm shows up on page 29:

PCP algorithm

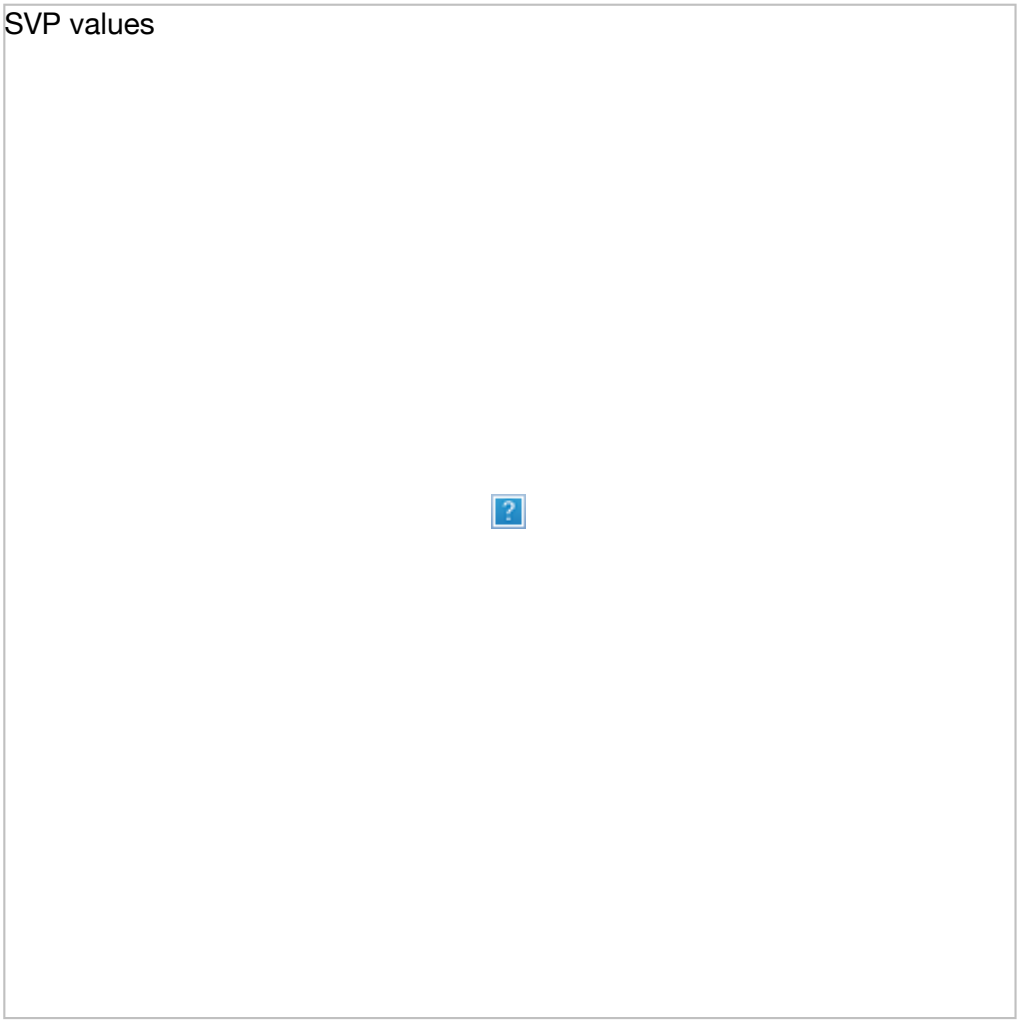
needed definitions of $\mathcal{S}$, the Shrinkage operator, and $\mathcal{D}$, the singular value thresholding operator:

PCP algorithm

Section 3.1 of Chen, Lin, Ma () contains a faster vartiation of this:



And Section 4 has some very helpful implementation details on how many singular values to calculate (as well as how to choose the parameter values):



If you want to learn more of the theory:

- Convex Optimization by Stephen Boyd (Stanford Prof):
  - OpenEdX Videos (https://www.youtube.com/playlist?list=PLbBM_dvjud8oFj09MqqYnGSrT6zek42Q0)
  - Jupyter Notebooks (http://web.stanford.edu/~boyd/papers/cvx_short_course.html)
- Alternating Direction Method of Multipliers (more Stephen Boyd (http://stanford.edu/~boyd/admm.html))

Robust PCA (via Primary Component Pursuit)

Methods

We will use Facebook's Fast Randomized PCA (https://github.com/facebook/fbpca) library.

In [48]:
```
from scipy import sparse
from sklearn.utils.extmath import randomized_svd
import fbpca
```

In [49]:
```
TOL=1e-9
MAX_ITERS=3
```

In [50]:
```
def converged(Z, d_norm):
    err = np.linalg.norm(Z, 'fro') / d_norm
    print('error: ', err)
    return err < TOL
```

In [51]:
```
def shrink(M, tau):
    S = np.abs(M) - tau
    return np.sign(M) * np.where(S>0, S, 0)
```

In [52]:
```
def _svd(M, rank): return fbpca.pca(M, k=min(rank, np.min(M.shape)), r
aw=True)
```

In [53]:
```
def norm_op(M): return _svd(M, 1)[1][0]
```

In [54]:
```
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 [97]:
```
def pcp(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

        # update estimate of Sparse Matrix by "shrinking/truncating":
original - low-rank
        S = shrink(X2 - L, lamda/mu)

        # update estimate of Low-rank Matrix by doing truncated SVD of
rank sv & reconstructing.
        # count of singular values > 1/mu is returned as svp
        L, svp = svd_reconstruct(X2 - S, sv, 1/mu)

        # If svp < sv, you are already calculating enough singular val
ues.
        # If not, add 20% (in this case 240) to sv
        sv = svp + (1 if svp < sv else round(0.05*n))

        # residual
        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
```

the algorithm again (page 29 of Candes, Li, Ma, and Wright (https://arxiv.org/pdf/0912.3599.pdf))

PCP algorithm

Results

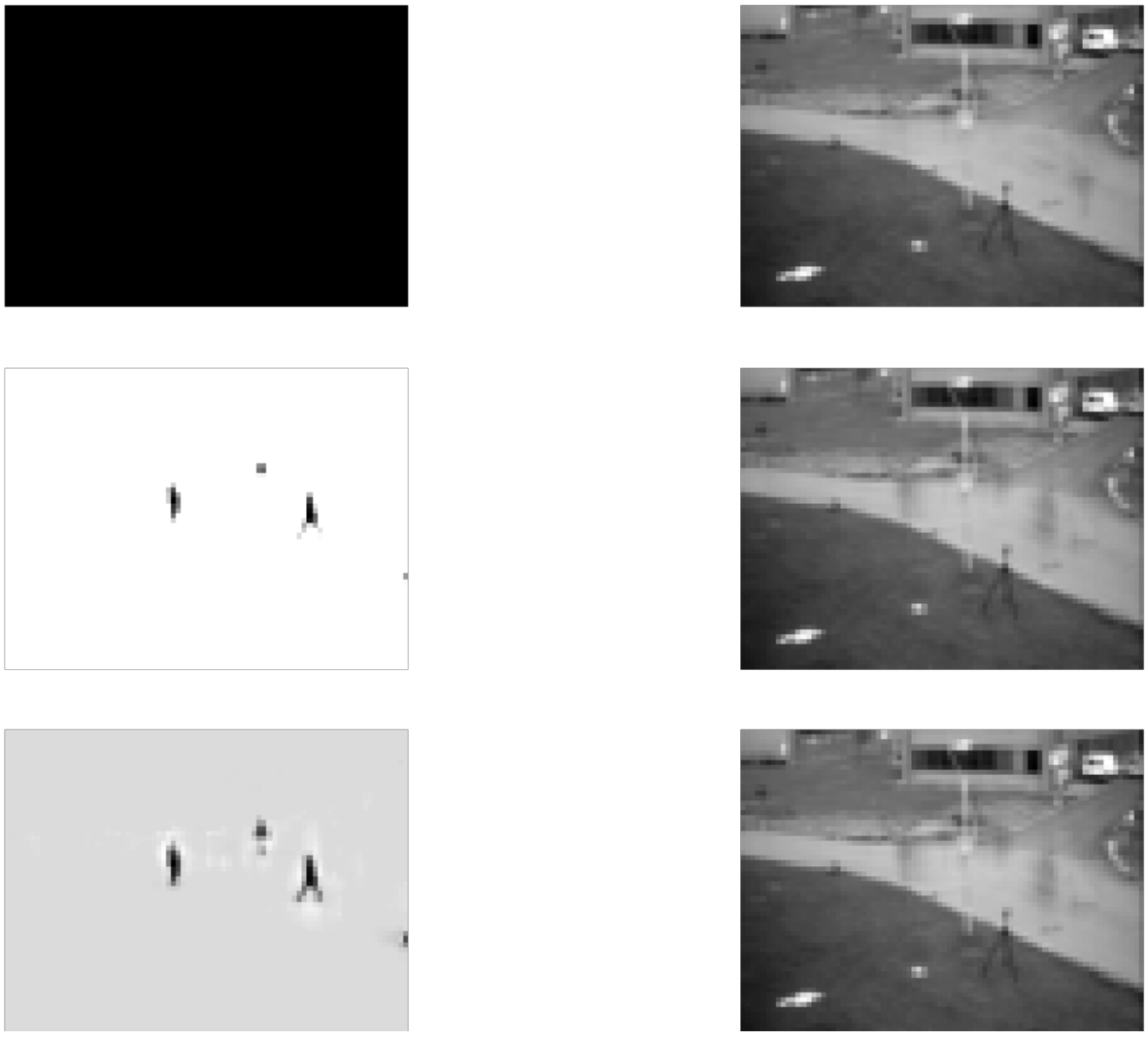
In [71]:
```
m, n = M.shape
round(m * .05)
```

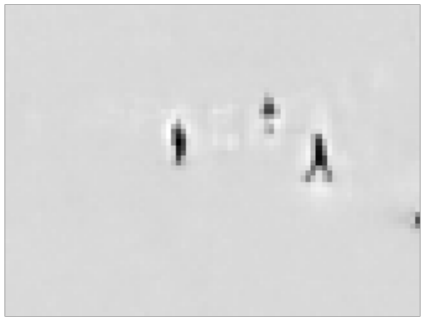
Out[71]: 240

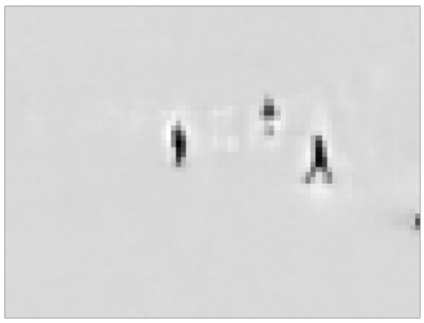
In [78]:
```
L, S, examples =  pcp(M, maxiter=5, k=10)
```

```
rank sv: 1
error:  0.131637937114
rank sv: 241
error:  0.0458515689278
rank sv: 49
error:  0.00591314217762
rank sv: 289
error:  0.000567221885441
rank sv: 529
error:  2.4633786172e-05
```

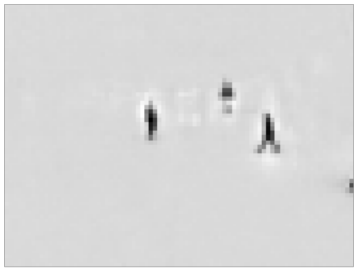
In [96]:
```
plots(examples, dims, rows=5)
```








In [92]:
```
f = plt_images(M, S, L, [140], dims)
```


In [37]:
```
np.save("high_res_L.npy", L)
np.save("high_res_S.npy", S)
```

In [38]:
```
f = plt_images(M, S, L, [0, 100, 1000], dims)
```

Extracting a bit of the foreground is easier than identifying the background. To accurately get the background, you need to remove all the foreground, not just parts of it

LU Factorization

Both `fbpca` and our own `randomized_range_finder` methods used LU factorization, which factors a matrix into the product of a lower triangular matrix and an upper triangular matrix.

Gaussian Elimination

This section is based on lectures 20-22 in Trefethen.

If you are unfamiliar with Gaussian elimination or need a refresher, watch this Khan Academy video (https://www.khanacademy.org/math/precalculus/precalc-matrices/row-echelon-and-gaussian-elimination/v/matrices-reduced-row-echelon-form-2).

Let's use Gaussian Elimination by hand to review:

A =

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

Answer:

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

Above example is from Lectures 20, 21 of Trefethen.

Gaussian Elimination transform a linear system into an upper triangular one by applying linear transformations on the left. It is *triangular triangularization*.

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

L is *unit lower-triangular*: all diagonal entries are 1

In [206]:
```
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 [207]:
```
A = np.array([[2,1,1,0],[4,3,3,1],[8,7,9,5],[6,7,9,8]]).astype(np.floa
t)
```

In [208]:
```
L, U = LU(A)
```

In [44]:
```
np.allclose(A, L @ U)
```

Out[44]: True

The LU factorization is useful!

Solving Ax = b becomes LUx = b:

1. find A = LU
2. solve Ly = b
3. solve Ux = y

Work

Work for Gaussian Elimination: $2 \cdot \frac{1}{3}n^3$

Memory

Above, we created two new matrices, L and U. However, we can store the values of L and U in our matrix A (overwriting the original matrix). Since the diagonal of L is all 1s, it doesn't need to be stored. Doing factorizations or computations **in-place** is a common technique in numerical linear algebra to save memory. Note: you wouldn't want to do this if you needed to use your original matrix A again in the future. One of the homework questions is to rewrite the LU method to operate in place.

Consider the matrix

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

In [128]:
```
A = np.array([[1e-20, 1], [1,1]])
```

By hand, use Gaussian Elimination to calculate what L and U are:

Answer

In [ ]:
```
#Exercise:
```

In [123]:
```
np.set_printoptions(suppress=True)
```

In [127]:
```
#Exercise:
```

In [129]:
```
L2, U2 = LU(A)
```

```
[[  1.00000000e-20   1.00000000e+00]
 [  0.00000000e+00  -1.00000000e+20]]
```

In [130]:
```
L2, U2
```

Out[130]:
```
(array([[  1.00000000e+00,   0.00000000e+00],
        [  1.00000000e+20,   1.00000000e+00]]),
 array([[  1.00000000e-20,   1.00000000e+00],
        [  0.00000000e+00,  -1.00000000e+20]]))
```

In [84]:
```
np.allclose(L1, L2)
```

Out[84]:
```
True
```

In [85]:
```
np.allclose(U1, U2)
```

Out[85]:
```
True
```

In [86]:
```
np.allclose(A, L2 @ U2)
```

Out[86]:
```
False
```

This is the motivation for LU factorization **with pivoting**.

This also illustrates that LU factorization is *stable*, but not *backward stable*. (spoiler alert: even with partial pivoting, LU is "explosively unstable" for certain matrices, yet stable in practice)

Stability

An algorithm $\hat{f}$ for a problem f is **stable** if for each x,

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

for some y with

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

A stable algorithm gives nearly the right answer to nearly the right question (Trefethen, pg 104)

To translate that:

- right question: x
- nearly the right question: y
- right answer: f
- right answer to nearly the right question: $f(y)$

Backwards Stability

Backwards stability is both **stronger** and **simpler** than stability.

An algorithm $\hat{f}$ for a problem f is **backwards stable** if for each x,

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

for some y with

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

A backwards stable algorithm gives exactly the right answer to nearly the right question (Trefethen, pg 104)

Translation:

- right question: x
- nearly the right question: y
- right answer: f
- right answer to nearly the right question: $f(y)$

LU factorization with Partial Pivoting

Let's now look at the matrix

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

```
In [89]: A = np.array([[1,1], [1e-20, 1]])
```

By hand, use Gaussian Elimination to calculate what L and U are:

Answer

```
In [ ]: #Exercise:
```

```
In [90]: L, U = LU(A)
```

```
In [93]: np.allclose(A, L @ U)
Out[93]: True
```

Idea: We can switch the order of the rows around to get more stable answers! This is equivalent to multiplying by a permutation matrix P. For instance,

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

$$PA = \hat{A}$$

Apply Gaussian elimination to PA.

At each step, choose the largest value in column k, and move that row to be row k.

Homework

In [100]:
```
def swap(a,b):
    temp = np.copy(a)
    a[:] = b
    b[:] = temp

a=np.array([1,2,3])
b=np.array([3,2,1])
swap(a,b)
a,b
```

In [102]:
```
#Exercise: re-write the LU factorization above to use pivoting
```

Example

In [104]:
```
A = np.array([[2,1,1,0],[4,3,3,1],[8,7,9,5],[6,7,9,8]]).astype(np.floa
t)
```

In [105]:
```
L, U, P = LU_pivot(A)
```

Can compare below to answers in Trefethen, page 159:

In [106]:
```
A
```

Out[106]:
```
array([[ 2.,  1.,  1.,  0.],
       [ 4.,  3.,  3.,  1.],
       [ 8.,  7.,  9.,  5.],
       [ 6.,  7.,  9.,  8.]])
```

In [107]:
```
U
```

Out[107]:
```
array([[ 8.        ,  7.        ,  9.        ,  5.        ],
       [ 0.        ,  1.75      ,  2.25      ,  4.25      ],
       [ 0.        ,  0.        , -0.28571429,  0.57142857],
       [ 0.        ,  0.        ,  0.        , -2.        ]])
```

In [114]:
```
P
```

Out[114]:
```
array([[ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.],
       [ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.]])
```

Partial pivoting permutes the rows. It is such a universal practice, that this is usually what is meant by *LU factorization*.

Complete pivoting permutes the rows and columns. Complete pivoting is significantly time-consuming and rarely used in practice.

Example

Consider the system of equations:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$

In [99]:
```
def make_matrix(n):
    A = np.eye(n)
    for i in range(n):
        A[i,-1] = 1
        for j in range(i):
            A[i,j] = -1
    return A
```

In [117]:
```
def make_vector(n):
    b = np.ones(n)
    b[-2] = 2
    return b
```

In [101]:
```
make_vector(7)
```

Out[101]:
```
array([ 1.,  1.,  1.,  1.,  1.,  2.,  1.])
```

Exercise

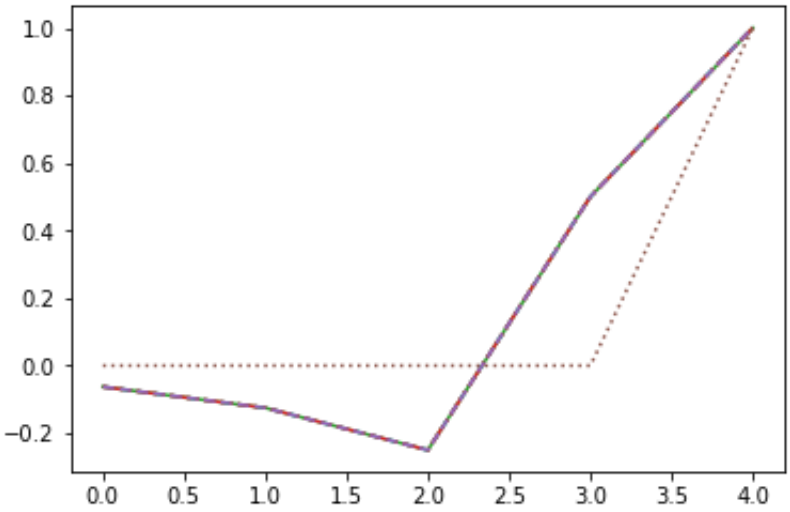
Exercise: Let's use Gaussian Elimination on the 5×5 system.

Scipy has this funtionality as well. Let's look at the solution for the last 5 equations with $n = 10, 20, 30, 40, 50, 60$.

In [131]:
```
?scipy.linalg.solve
```

In [112]:
```
for n, ls in zip(range(10, 70, 10), ['--', ':', '-', '-.', '--', ':'])
:
    soln = scipy.linalg.lu_solve(scipy.linalg.lu_factor(make_matrix(n)
), make_vector(n))
    plt.plot(soln[-5:], ls)
    print(soln)
```

```
[-0.00195312 -0.00390625 -0.0078125  -0.015625   -0.03125    -0.0625
-0.125
 -0.25        0.5         1.00195312]
[ -1.90734863e-06  -3.81469727e-06  -7.62939453e-06  -1.52587891e-05
  -3.05175781e-05  -6.10351562e-05  -1.22070312e-04  -2.44140625e-04
  -4.88281250e-04  -9.76562500e-04  -1.95312500e-03  -3.90625000e-03
  -7.81250000e-03  -1.56250000e-02  -3.12500000e-02  -6.25000000e-02
  -1.25000000e-01  -2.50000000e-01   5.00000000e-01   1.00000191e+00
]
[ -1.86264515e-09  -3.72529030e-09  -7.45058060e-09  -1.49011612e-08
  -2.98023224e-08  -5.96046448e-08  -1.19209290e-07  -2.38418579e-07
  -4.76837158e-07  -9.53674316e-07  -1.90734863e-06  -3.81469727e-06
  -7.62939453e-06  -1.52587891e-05  -3.05175781e-05  -6.10351562e-05
  -1.22070312e-04  -2.44140625e-04  -4.88281250e-04  -9.76562500e-04
  -1.95312500e-03  -3.90625000e-03  -7.81250000e-03  -1.56250000e-02
  -3.12500000e-02  -6.25000000e-02  -1.25000000e-01  -2.50000000e-01
   5.00000000e-01   1.00000000e+00]
[ -1.81898940e-12  -3.63797881e-12  -7.27595761e-12  -1.45519152e-11
  -2.91038305e-11  -5.82076609e-11  -1.16415322e-10  -2.32830644e-10
  -4.65661287e-10  -9.31322575e-10  -1.86264515e-09  -3.72529030e-09
  -7.45058060e-09  -1.49011612e-08  -2.98023224e-08  -5.96046448e-08
  -1.19209290e-07  -2.38418579e-07  -4.76837158e-07  -9.53674316e-07
  -1.90734863e-06  -3.81469727e-06  -7.62939453e-06  -1.52587891e-05
  -3.05175781e-05  -6.10351562e-05  -1.22070312e-04  -2.44140625e-04
  -4.88281250e-04  -9.76562500e-04  -1.95312500e-03  -3.90625000e-03
  -7.81250000e-03  -1.56250000e-02  -3.12500000e-02  -6.25000000e-02
  -1.25000000e-01  -2.50000000e-01   5.00000000e-01   1.00000000e+00
]
[ -1.77635684e-15  -3.55271368e-15  -7.10542736e-15  -1.42108547e-14
  -2.84217094e-14  -5.68434189e-14  -1.13686838e-13  -2.27373675e-13
  -4.54747351e-13  -9.09494702e-13  -1.81898940e-12  -3.63797881e-12
  -7.27595761e-12  -1.45519152e-11  -2.91038305e-11  -5.82076609e-11
  -1.16415322e-10  -2.32830644e-10  -4.65661287e-10  -9.31322575e-10
  -1.86264515e-09  -3.72529030e-09  -7.45058060e-09  -1.49011612e-08
  -2.98023224e-08  -5.96046448e-08  -1.19209290e-07  -2.38418579e-07
  -4.76837158e-07  -9.53674316e-07  -1.90734863e-06  -3.81469727e-06
  -7.62939453e-06  -1.52587891e-05  -3.05175781e-05  -6.10351562e-05
  -1.22070312e-04  -2.44140625e-04  -4.88281250e-04  -9.76562500e-04
  -1.95312500e-03  -3.90625000e-03  -7.81250000e-03  -1.56250000e-02
  -3.12500000e-02  -6.25000000e-02  -1.25000000e-01  -2.50000000e-01
   5.00000000e-01   1.00000000e+00]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.
  0.  0.  0.  0.  0.  1.]
```



What is going on when $n = 60$?

Theorem: Let the factorization $PA = LU$ of a matrix A be computed by Gaussian Elimination with partial pivoting. The *computed* (by a computer with Floating Point Arithmetic) matrices $\hat{P}$, $\hat{L}$, and $\hat{U}$ satisfy

$$\hat{L}\hat{U} = \hat{P}\hat{A} + \delta A, \quad \frac{\delta A}{A} = \mathcal{O}(\rho \varepsilon_{machine})$$

where ρ is the *growth factor*,

$$\rho = \frac{max_{i,j}|u_{ij}|}{max_{i,j}|a_{ij}|}$$

For our matrix above, $\rho = 2^{m-1}$

Unstable in theory, stable in practice

Stability of most algorithms (such as QR) is straightforward. Not the case for Gaussian Elimination with partial pivoting. Instability in Gaussian elimination (with or without pivoting) arises only if L and/or U is large relative to the size of A.

Trefethen: "Despite examples like (22.4), Gaussian elimination with partial pivoting is utterly stable in practice... In fifty years of computing, no matrix problems that excite an explosive instability are known to have arisen under natural circumstances." [although can easily be constructed as contrived examples]

Although some matrices cause instability, but extraordinarily small proportion of all matrices so "never" arise in practice for statistical reasons. "If you pick a billion matrices at random, you will almost certainly not find one for which Gaussian elimination is unstable."

Further Reading

- Gaussian Elimination/LU factorization-- Trefethn Lecture 20
- Pivoting -- Trefethn Lecture 21
- Stability of Gaussian Elimination -- Trefethn Lecture 22

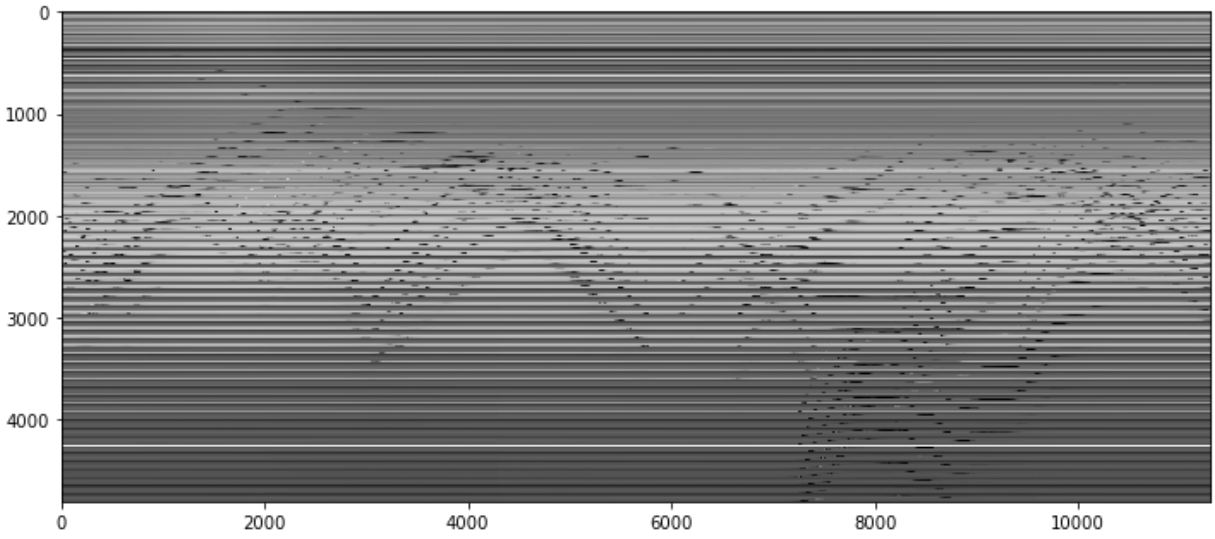
Follow up from last class

What is going on with Randomized Projections?

We are taking a linear combination (with random weights) of the columns in the matrix below:

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

Out[33]: <matplotlib.image.AxesImage at 0x7f601f315fd0>



It's like a random weighted average. If you take several of these, you end up with columns that are orthonormal to each other.

Johnson-Lindenstrauss Lemma: (from wikipedia (https://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma)) a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved.

It is desirable to be able to reduce dimensionality of data in a way that preserves relevant structure. The Johnson–Lindenstrauss lemma is a classic result of this type.

History of Gaussian Elimination

Fascinating history of Gaussian Elimination (https://jvns.ca/blog/2017/04/16/making-small-culture-changes/). Some highlights:

- First written record of Gaussian elimination from ~200 BC in the Chinese book *Nine Chapters on Arithmetic*
- The ancient Chinese used colored bamboo rods placed in columns on a "counting board"
- Japanese mathematician Seki Kowa (16432-1708) carried forward the Chinese elimintion methods and invented the determinant before 1683. Around the same time, Leibniz made similar discoveries independently, but neither Kowa nor Leibniz go credit for their discoveries.
- Gauss referred to the elimination method as being "commonly known" and never claimed to have invented it, although he may have invented the Cholesky decomposition

More history here (http://www.sciencedirect.com/science/article/pii/S0315086010000376)

Speeding Up Gaussian Elimination

Parallelized LU Decomposition (https://courses.engr.illinois.edu/cs554/fa2013/notes/06_lu_8up.pdf) LU decomposition can be fully parallelized

Randomized LU Decomposition (http://www.sciencedirect.com/science/article/pii/S1063520316300069) (2016 article): The randomized LU is fully implemented to run on a standard GPU card without any GPU–CPU data transfer.

Scipy.linalg solve vs lu_solve

In [142]:
```
n = 100
A = make_matrix(n)
b = make_vector(n)
```

This problem has a large *growth factor* $= 2^{59}$. We get the wrong answer using scipy.linalg.lu_solve, but the right answer with scipy.linalg.solve. What is scipy.linalg.solve doing?

In [143]:
```
print(scipy.linalg.lu_solve(scipy.linalg.lu_factor(A), b)[-5:])
print(scipy.linalg.solve(A, b)[-5:])
```

```
[ 0.  0.  0.  0.  1.]
[-0.0625 -0.125  -0.25    0.5     1.    ]
```

In [136]:
```
%%timeit
soln = scipy.linalg.lu_solve(scipy.linalg.lu_factor(A), b)
soln[-5:]
```

```
91.2 µs ± 192 ns per loop (mean ± std. dev. of 7 runs, 10000 loops ea
ch)
```

In [137]:
```
%%timeit
soln = scipy.linalg.solve(A, b)
soln[-5:]
```

```
153 µs ± 5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Looking at the source code for scipy (https://github.com/scipy/scipy/blob/v0.19.0/scipy/linalg/basic.py#L25-L224), we see that it is calling the LAPACK routine `gesvx`. Here is the Fortran source code for sgesvx (http://www.netlib.org/lapack/explore-html/d0/db8/group__real_g_esolve_ga982d53a8a62d66af9bcaa50642c95ea4.html#ga982d53a8a62d66af9bca (`s` refers to single, there is also `dgesvx` for doubles and `cgesvx` for complex numbers). In the comments, we see that it is computing a *reciprocal pivot growth factor*, so it is taking into account this growth factor and doing something more complex than plain partial pivot LU factorization.

Block Matrices

This is a follow-up to a question about block matrices asked in a previous class. But first,

Ordinary Matrix Multiplication

Question: What is the computational complexity (big $\mathcal{O}$) of matrix multiplication for multiplying two $n \times n$ matrices $A \times B = C$?

You can learn (or refresh) about big $\mathcal{O}$ on Codecademy (https://www.codecademy.com/courses/big-o/0/1)

What this looks like:

```
for i=1 to n
    {read row i of A into fast memory}
    for j=1 to n
        {read col j of B into fast memory}
        for k=1 to n
            C[i,j] = C[i,j] + A[i,k] x B[k,j]
        {write C[i,j] back to slow memory}
```

Question: How many reads and writes are made?

Block Matrix Multiplication

Divide A, B, C into $N \times N$ blocks of size $\frac{n}{N} \times \frac{n}{N}$

(Source (http://avishek.net/blog/?p=804))

What this looks like:

```
for i=1 to N
    for j=1 to N
        for k=1 to N
            {read block (i,k) of A}
            {read block (k,j) of B}
            block (i,j) of C += block of A times block of B
        {write block (i,j) of C back to slow memory}
```

Question 1: What is the big-$\mathcal{O}$ of this?

Question 2: How many reads and writes are made?

End