% macros

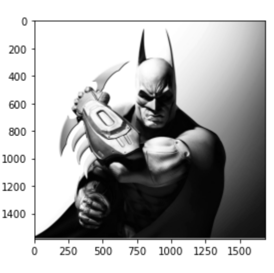
In [46]:
```python
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from math import sqrt, inf
from numpy.linalg import norm
```

In [47]:
```python
# image input and conversion into matrix
img = Image.open('batman.jpg')
og_size = float(os.path.getsize('batman.jpg'))/1024
image = np.matrix(img.getdata(band = 0), float)/255
image.shape = (img.size[1], img.size[0])

print("Image dimensions", image.shape, "Image size %.2f Kb"%og_size)
plt.imshow(image, cmap = 'gray')
```

```
Image dimensions (1584, 1688) Image size 341.64 Kb
```

Out[47]:
```
<matplotlib.image.AxesImage at 0x1bb91131280>
```



In [48]:
```python
# Preliminaries
def make_vector(V):
    return [np.squeeze(np.asarray(v)) for v in V]

def outer_product(u, v):
    return u*(v.T)

# return e_j, the j-th vector in the standard basis
def e(j, n):
    v = [0]*n; v[j - 1] = 1
    return np.matrix(v).T
```

Power method to calculate the maximum eigen value.

The algorithm is implemented in the following function _maximum_eigen_.

- Start with a random vector u, the algorithm uses e_1.
- Calculate Au and subsequently the unit vector in its direction v.
- If v is close to u then v is an eigen vector with eigen value $||Au||$.
- If v is close to $-u$ then v is an eigen vector with eigen value $-||Au||$.
- Otherwise set $u = v$ and repeat from step 2.

Why does power method work?

We would be calculating the maximum eigen value of $A^tA = P$, a positive semidefinite matrix, which is diagonalisable with a basis of its eigen vectors $v_1, v_2, \cdots, v_n$ and eigen values $s_1 \geq s_2 \geq \cdots \geq s_n$.
Let the original vector $u = c_1v_1 + c_2v_2 + \cdots + c_nv_n$.

Now let $u_k = A^k(u)$

Then, $\frac{u_k}{||u_k||} = \frac{1}{||u_k||}\sum_{j=0}^{n} c_jA^k(v_j) = \sum_{j=0}^{n}\frac{{s_j}^k c_j}{||u_k||}v_j \to v_1$, as $k \to \infty$.

Intuitively, the limit works with $s_1 > s_2 \geq \cdots \geq s_n$, and hence $\frac{||u_k||}{{s_j}^k c_j} \to \begin{cases} 1, j = 1 \\ \infty, j \neq 1 \end{cases}$.

With equality we fetch an unit vector in the (at least) 2-dimensional eigen space of s_1.

In [49]:
```python
def maximum_eigen(A, epsilon = 0.0001):
    n = np.shape(A)[0]; u = e(1, n)

    while True:
        z = A*u; eig = norm(z); v = z/eig

        if norm(v - u) < epsilon:
            return (eig, v)
        if norm(v + u) < epsilon:
            return (-eig, v)
        u = v
```

The SVD function

Let $A = USV^t$ be its singular value decomposition. Then it immediately follows that the columns of V, the right singular vectors are eigen vectors of $P = A^tA$ with the square of the diagonal entries of S, viz $s_1^2, \cdots, s_n^2$ as its eigen values in decreasing order.
The right singular vectors

The first singular value s_1 and singular vector v_1 can be found by applying the _maximum_eigen_ function to P as mentioned in the previous section. Now let $P' = P - s_1^2 v_1 {v_1}^t$. Observe,

$$P'v_j = Pv_j - s_1^2 v_1({v_1}^t v_j) = s_j^2 v_j - s_1^2 v_1 \delta_{1j} = \begin{cases} 0, j = 1 \\ s_j^2 v_j, j \neq 1 \end{cases}$$

Clearly, the eigen values of P' are $0, s_2^2, \cdots, s_n^2$, and _maximum_eigen_ can be reused on P' to fetch s_2. This process can be iteratively continued to find all eigen vectors and values of P.

The left singular vectors

Once the $i-th$ left singular vector v_i is found, observe,

$$Av_i = USV^t \cdot v_i = \sum_{j=0}^{n} s_j u_j(v_j^t v_i) = s_i u_i$$

So the corresponding right singular vector u_i can be immediately found by calculating Av_i/s_i.

In [50]:
```python
# The best k-rank approximator
def SVD(A, k = inf):
    n = A.shape[1]
    k = min(k, n)
    S = [0]; V = [e(1, n)]; U = []

    P = A.T*A
    for j in range(k):
        eig = S[-1]; v = V[-1]
        P = P - eig*outer_product(v, v)

        # The right singular vectors, values
        new_eig, new_v = maximum_eigen(P)
        S.append(new_eig); V.append(new_v)

        # The left singular vectors
        if new_eig != 0:
            U.append(A*new_v/sqrt(new_eig))

        else: break

    del S[0]; del V[0]
    S = [sqrt(x) for x in S]
    return (make_vector(U), S, make_vector(V))
```

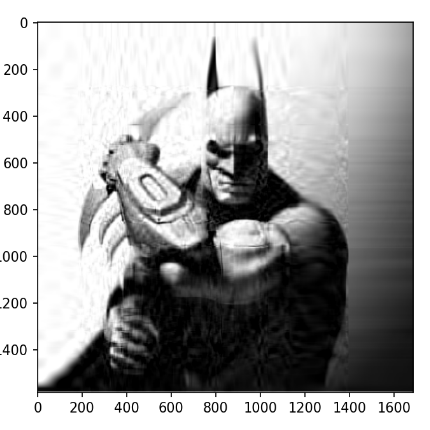
In [51]:
```python
U, S, V = SVD(image, 300)
```

In [73]:
```python
# Reconstructing the image matrix with k singular vectors
def image_constructor(k):
    m = len(V)
    k = min(m, k)
    U1 = np.matrix(U[:k]).T
    V1 = np.matrix(V[:k])
    S1 = np.diag(S[:k])
    im = Image.fromarray(U1 * S1 * V1 * 255)
    return im
```

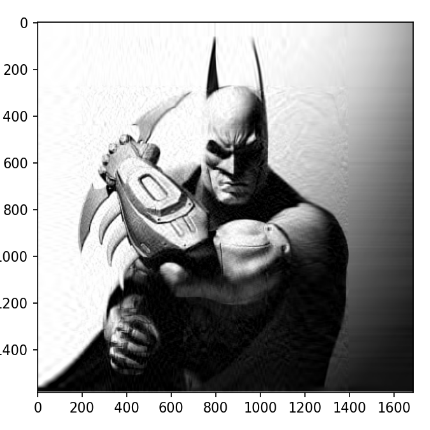
In [74]:
```python
%matplotlib notebook
sizes = []
for j in range(1, 11):
    im = image_constructor(j*30)
    print("Image with %d singular vectors."%(j*30))
    plt.figure()
    plt.imshow(im, cmap = 'gray')
    im_name = 'im' + str(j) + '.jpg'
    plt.imsave(im_name, im)

    size = float(os.path.getsize(im_name))/1024
    sizes.append((j*30, size))
```

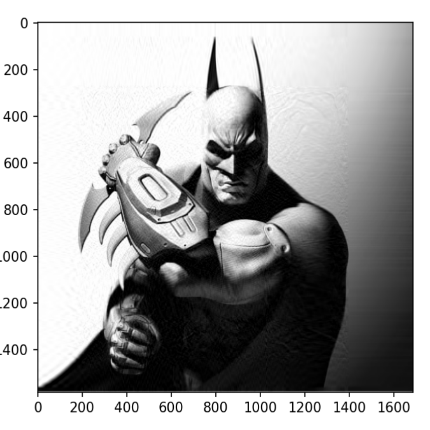
```
Image with 30 singular vectors.
```



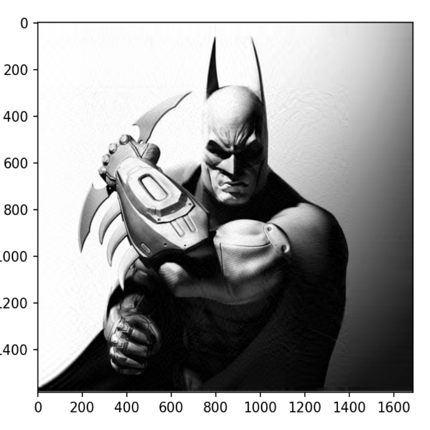
```
Image with 60 singular vectors.
```



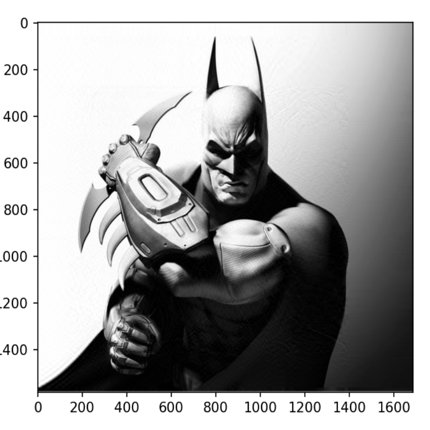
```
Image with 90 singular vectors.
```



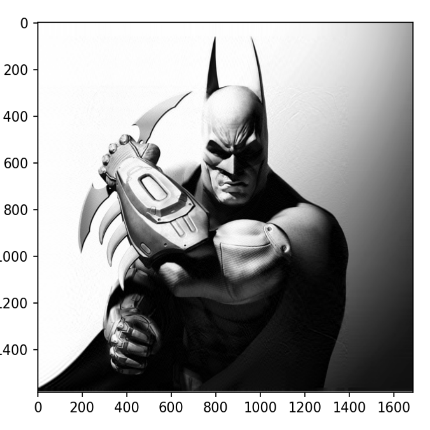
```
Image with 120 singular vectors.
```



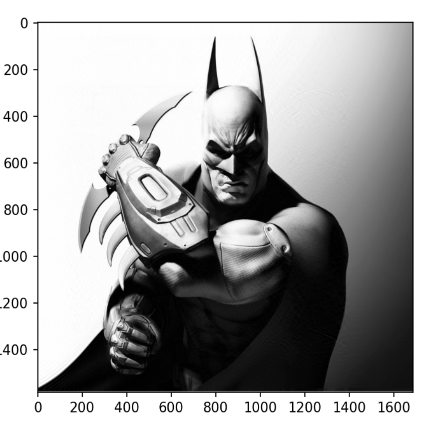
```
Image with 150 singular vectors.
```



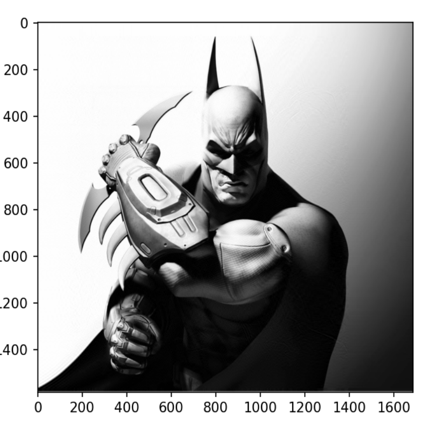
```
Image with 180 singular vectors.
```



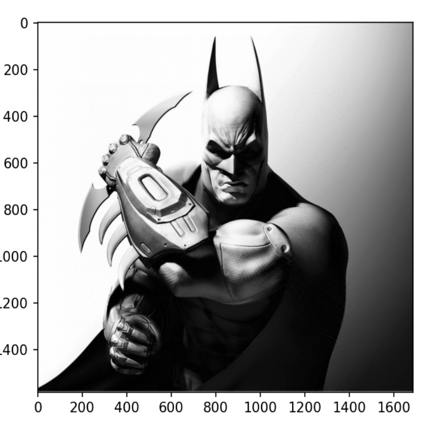
```
Image with 210 singular vectors.
```



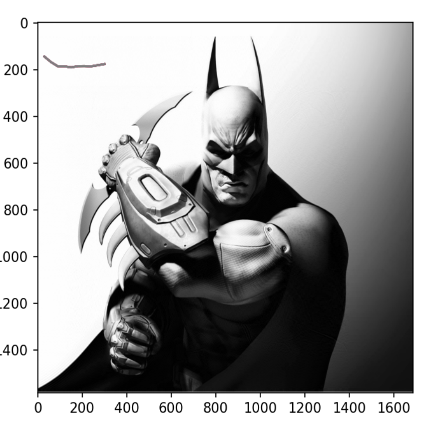
```
Image with 240 singular vectors.
```



```
Image with 270 singular vectors.
```



```
Image with 300 singular vectors.
```



In [83]:
```python
print("Orgininal size %.2f Kb \n\n No. of singular vectors    Compressed Image Size (kb)"%og_size)
for size in sizes:
```

```
Orgininal size 341.64 Kb

 No. of singular vectors    Compressed Image Size (kb)
                      30                        144.64
                      60                        168.36
                      90                        186.15
                     120                        186.26
                     150                        191.14
                     180                        185.84
                     210                        185.54
                     240                        186.63
                     270                        181.75
                     300                        176.82
```

In [ ]: