1. LASSO vs Ridge

Homework 8

(a) $\|Xw-y\|_2^2 + \lambda\|w\|_1$

$\langle Xw-y, Xw-y\rangle + \lambda\sum|w_i|$

$= \langle Xw, Xw\rangle - 2\langle Xw, y\rangle + \langle y, y\rangle + \lambda\sum|w_i|$

$= w^TX^TXw - 2y^TXw + y^Ty + \lambda\sum|w_i|$

$= nw^Tw \quad \ldots$

$= n\sum w_i^2 - 2y^T\sum\vec{x}_iw_i + \sum y_i^2 + \lambda\sum|w_i|$

$= n\sum w_i^2 - 2\sum y^T\vec{x}_iw_i + \sum y_i^2 + \lambda\sum|w_i|$

$\Rightarrow \boxed{\min_{w_i}\left(nw_i^2 - 2(y^T\vec{x}_i)w_i + y_i^2 + \lambda|w_i|\right)}$

(b) if $w_i^* > 0$

derivative of objective function

$2nw_i - 2y^Tx_i + \lambda = 0$ set equal to zero

$\boxed{w_i^* = \frac{2y^Tx_i - \lambda}{2n}}$

$\boxed{y^Tx_i > \frac{\lambda}{2}}$

and $y^Tx_i > 0$

$\Rightarrow |w_i| = w_i$

(by definition of abs. value $|x| = \begin{cases} x & x \geq 0 \\ -x & x < 0 \end{cases}$)

Scanned with CamScanner

* (c) $w_i^* < 0$

by definition of $|\cdot|$

derivative

$$2n w_i - 2y^T x_i - \lambda = 0 \qquad |w_i| = -w_i$$

$$\boxed{w_i^* = \frac{2y^T x_i + \lambda}{2n}}$$

$$\boxed{|y^T x_i| > \frac{\lambda}{2}}$$

and $y^T x_i < 0$

* (d) if $|y^T x_i| \leq \frac{\lambda}{2}$

if $y^T x_i > 0$

$\Rightarrow \quad y^T x_i \leq \frac{\lambda}{2}$ and

if $y^T x_i < 0$

$-y^T x_i \leq \frac{\lambda}{2}$

$y^T x_i \geq -\frac{\lambda}{2}$ and $\boxed{w_i^* < 0}$

(e) Ridge Regression

$$w^* = \arg\min_{w \in \mathbb{R}^d} \underbrace{\|Xw - y\|_2^2 + \underline{\lambda\|w\|_2^2}}$$

objective function

$$w^T X^T X w - 2y^T X w + y^T y + \lambda w^T w$$

$$n w^T w - 2 y^T X w + y^T y + \lambda w^T w$$

component-wise

$$(n+\lambda)\, w_i^2 - 2(y^T x_i)\, w_i + y_i^2$$

$\therefore$ the problem is equivalent to

$$\boxed{\min_{w_i} (n+\lambda) w_i^2 - 2(y^T x_i) w_i + y_i^2}$$

$$\Rightarrow \quad w_i^* = \frac{2y^T x_i}{2(n+\lambda)} = \frac{y^T x_i}{n+\lambda}$$

which removes restrictions on the previous condition on $y^T x_i$, which was dependent on the choice of λ. This seems to be a more more robust form of regression

2. Image Compression

(a) full SVD

$$\underset{m\times n}{A} = U \Sigma V^T$$

$$= \underbrace{\begin{bmatrix} u_1 & \cdots & u_r & u_{r+1} & \cdots & u_m \end{bmatrix}}_{m\times m} \underbrace{\left[\begin{array}{ccc|c} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_r & \\ \hline & & & \ddots \end{array}\right]}_{m\times n} \underbrace{\begin{bmatrix} - & v_1^T & - \\ & \vdots & \\ - & v_r^T & - \\ - & v_{r+1}^T & - \\ & \vdots & \\ - & v_n^T & - \end{bmatrix}}_{n\times n}$$

$\Rightarrow$ rank k approximation

$$\boxed{\underbrace{\begin{bmatrix} u_1 & \cdots & u_k \end{bmatrix}}_{m\times k} \underbrace{\begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{bmatrix}}_{k\times k} \underbrace{\begin{bmatrix} - & v_1^T & - \\ & \vdots & \\ - & v_k^T & - \end{bmatrix}}_{k\times n} = A_{m\times n}}$$

(b)

$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|$$

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} = \sqrt{\operatorname{trace}(A^T A)} = \sqrt{\sum_{i=1}^{r} \sigma_i^2}$$

$$\boxed{\min_{\hat{y}} \|A - \hat{y}\|_F + \lambda \|\hat{y}\|_1}$$

c. Jupyter

3. Image Restoration

(a) $F = \underbrace{\begin{pmatrix} f(1,1) & \cdots & f(1,W) \\ & \vdots & \\ f(H,1) & \cdots & f(H,W) \end{pmatrix}}_{\text{Width}} \Big\} \text{Height}$

since $f(i,j)$ represent the grayscale value of the image at discrete (pixel) coordinate (i,j)

we can represent the image as a matrix as shown above.

So it makes sense $F(i,j) = f(i,j)$

(b) $F(i,j) = f(i,j)$

$$\Rightarrow \nabla F(i,j) = \nabla f(i,j) = \begin{pmatrix} \dfrac{f(i+h,j) - f(i,j)}{h} \\ \dfrac{f(i,j+h) - f(i,j)}{h} \end{pmatrix}$$

for $h=1$

$$\nabla F(i,j) = G(i,j) = \begin{pmatrix} f(i+1,j) - f(i,j) \\ f(i,j+1) - f(i,j) \end{pmatrix}$$

(c) $\min_{\substack{\hat{f} \\ \forall (x,y) \notin A}} \iint_{\Omega} \|\nabla \hat{f}(x,y)\|_2 \, dx\,dy$

expressed as discrete Reimann sum over pixel values

$$\boxed{\min_{\substack{\hat{F} \\ \forall (i,j) \notin A}} \sum_{i=1}^{H} \sum_{j=1}^{W} \|\nabla \hat{F}(i,j)\|_2 \underbrace{\Delta i}_{=1} \underbrace{\Delta j}_{=1}}$$

OR

$$= \min_{\substack{\hat{F} \\ \forall (i,j) \notin A}} \sum_{i=1}^{H} \sum_{j=1}^{W} \sqrt{\left(f(i+1,j) - f(i,j)\right)^2 + \left(f(i,j+1) - f(i,j)\right)^2}$$

(d) $\min_{\substack{\hat{F},\, y \\ \forall i,j \notin A}} \sum_{i=1}^{H} \sum_{j=1}^{W} y_j$

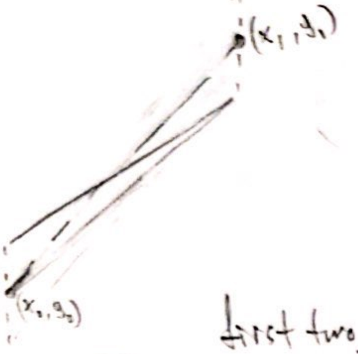
s.t $\|\nabla \hat{F}(i,j)\|_2 \leq y_j \quad j = 1, \ldots, W$

(e) In notebook.

It worked

4. Slalom problem

(a)



first two points

$$\min \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

$$\min_{y} \sum_{i=0}^{n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

$$\text{s.t.} \quad y_i \geq y_i - c_i/2$$

$$y_i \leq y_i + c_i/2$$

as SOCP

$$\boxed{\begin{aligned} \min_{y,t} \; & \sum_{i=0}^{n-1} t_i \\ \text{s.t.} \; & \sqrt{(x_{i+1}-x_i)^2 + (y_{i+1}-y_i)^2} \leq t_i \\ & y_i \geq y_i - c_i/2 \\ & y_i \leq y_i + c_i/2 \end{aligned}}$$

(b) notebook

5. Sphere enclosure

$$\min_{x,R} B(x,R)$$

we want to find the minimum radius R of ball centered at x

subject to

$$\|x - x_i\|_2 + \rho_i \leq R$$

distance between center of B to center of each enclosed Ball B_i

→ this distance plus the radius ρ_i should be less than minimum radius R for each B_i

cast as SOCP

$$\boxed{\begin{aligned} &\min_{x,R} R \\ &\text{s.t. } B(x,r) \leq R \\ &\|x - x_i\|_2 + \rho_i \leq R \end{aligned}}$$

Scanned with CamScanner

image_compression

November 7, 2019

[23]:
```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
import pywt
```

[24]:
```
%matplotlib inline
```

1 Image Compression

Image compression is used to efficently store and/or transmit images. Practically every image you have ever seen has been compressed in some form. You probably use some form of image compression daily, whether it's buffering videos on Youtube or scrolling through your Instagram feed. With the concepts you have learned in this class, you can get a first hand experience into how some of these images are actually compressed.

Firstly, there are two types of compression: lossless and lossy. Lossless compression is when you compress your data without losing any information. This form of compression is found in the Lempel-Ziv based zip functionality built into your computer, or in the Huffman Encoding you may have seen in your other classes (EE126, CS170). These methods are common where loss of information can have catastrophic effects - missing bytes in a file could prevent you from opening that file.

Then there's lossy compression, when you compress data by **intelligently** getting rid of some information. Lossy compression is common in applications where speed and size are more important than quality. Would you rather wait for hours and rack up a large internet bill trying to watch Netflix in perfect quality? Or would you like fast streaming with minimal bandwidth usage that results in slightly blocky images? Most would choose the latter.

This question will focus on three different methods of lossy compression: SVD compression, Fourier Compression, and Wavelet compression.

The question when it comes to lossy compression is how do we choose what information to throw away? The best answer(s) to this question come from when we can find an alternative representation of the image that has some form of sparsity. Why? Because if there is a sparse representation of the image, we can set the near zero coeffecients to zero (so we have less data representing the image) without sacrificing the quality of the image by too much.

1.1 Image Loading

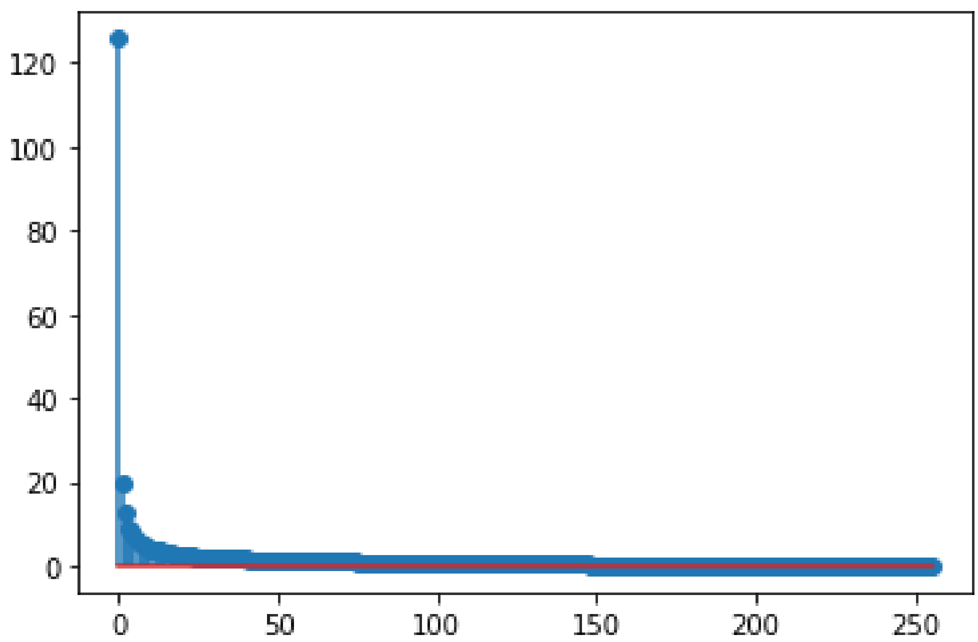
The cell below is necessary for reading in the image we will use for the rest of this notebook. We have provided you two images (the same ones from last week) to run this notebook with. You can submit the notebook with the cells run for either image. (Optional) We highly recommend using your own images to see how the different compression methods might work on different images.

[25]:
```
im_name = "lichtenstein.png"
# im_name = "lenna.png"

im = np.mean(plt.imread(im_name),axis=2).astype("float64")
im = (im/im.max())[::2,::2]

plt.figure(figsize=(10,10))
plt.imshow(im, cmap="gray")
plt.show()
assert im.shape == (256,256)
```

1.2 SVD Image Compression

Because we interpret our image as a matrix, our image has a certain rank. The idea behind image compression with SVD is to represent our compressed image by using a low rank approximation of our original image. But how is the SVD sparse? How do we know whether its a good method for compression? Let's plot the singular values of our image below.

[26]:
```
_, s, _ = np.linalg.svd(im)
plt.figure()
plt.stem(s, use_line_collection=True)
plt.show()
```

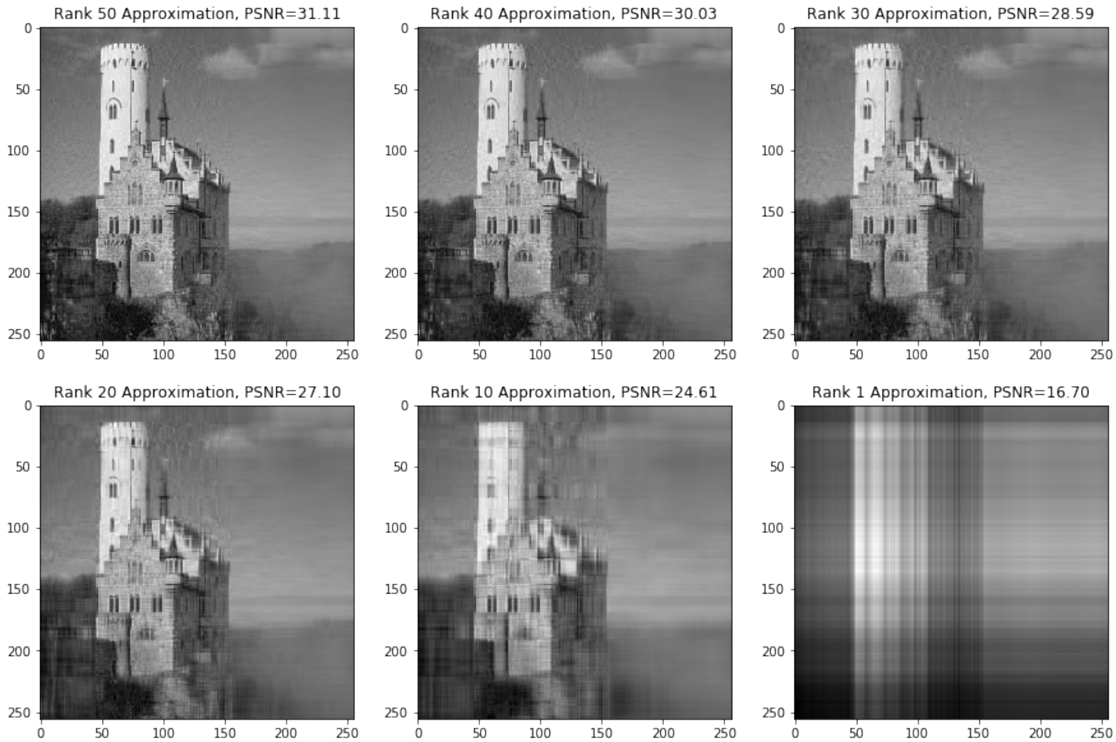


As you can see, the sparsity of the SVD refers to the sparsity of the singular values. That means we can use a low rank approximation of the image to represent our original image with fairly high accuracy. Implement the low rank approximation function labeled `svd_comp` below.

Quality Measurements To measure the quality of each compression technique, we will use two quantitative measurements: mean squared error (MSE) and peak signal to noise ratio (PSNR). These metrics are common ones in the imaging field and have been implemented for you. If you would like to learn more about these measurements, you can check this link.

[27]:
```
def mse(gt, im):
    return np.mean(np.square(np.abs(gt-im)))

def psnr(gt, im):
    return 20*np.log10(im.max())-10*np.log10(mse(gt,im))

def svd_comp(im, k):
    assert k <= np.linalg.matrix_rank(im)

    u, s, v = np.linalg.svd(im)
    comp_im = u[:,:k]@np.diag(s[0:k])@v[:k,:] # rank k approximation

    assert k == np.linalg.matrix_rank(comp_im)
    return comp_im
```

[28]:
```
k_vals = np.linspace(1, 50, 6).astype(int)[::-1]
plt.figure(figsize=(15,10))
for i,k in enumerate(k_vals):
    plt.subplot(2, 3, i+1)
    comp_im = svd_comp(im,k)
    plt.imshow(comp_im, cmap='gray')
    plt.title("Rank {} Approximation, PSNR={:.2f}".format(k, psnr(im,comp_im)))
```



Q: Describe how the image changes (with respect to the PSNR and overall quality) as the rank decreases

A:

We get more noise in the signal as rank decreases.

1.3 Fourier Transform Image Compression

As you discovered in last week's homework, the Fourier transform for natural images tends to have large low frequency coeffecients and small high frequency coeffecients. We can take advantage of this sparsity for image compression, but how do we decide what coeffecients to zero? We can pose the problem as a LASSO problem and solve.

Implement the three functions `loss_fn`, `regularizer`, and `objective_fn` in the cell below, and in the cell below the next, setup the problem as you described it the written part (b) of this question.

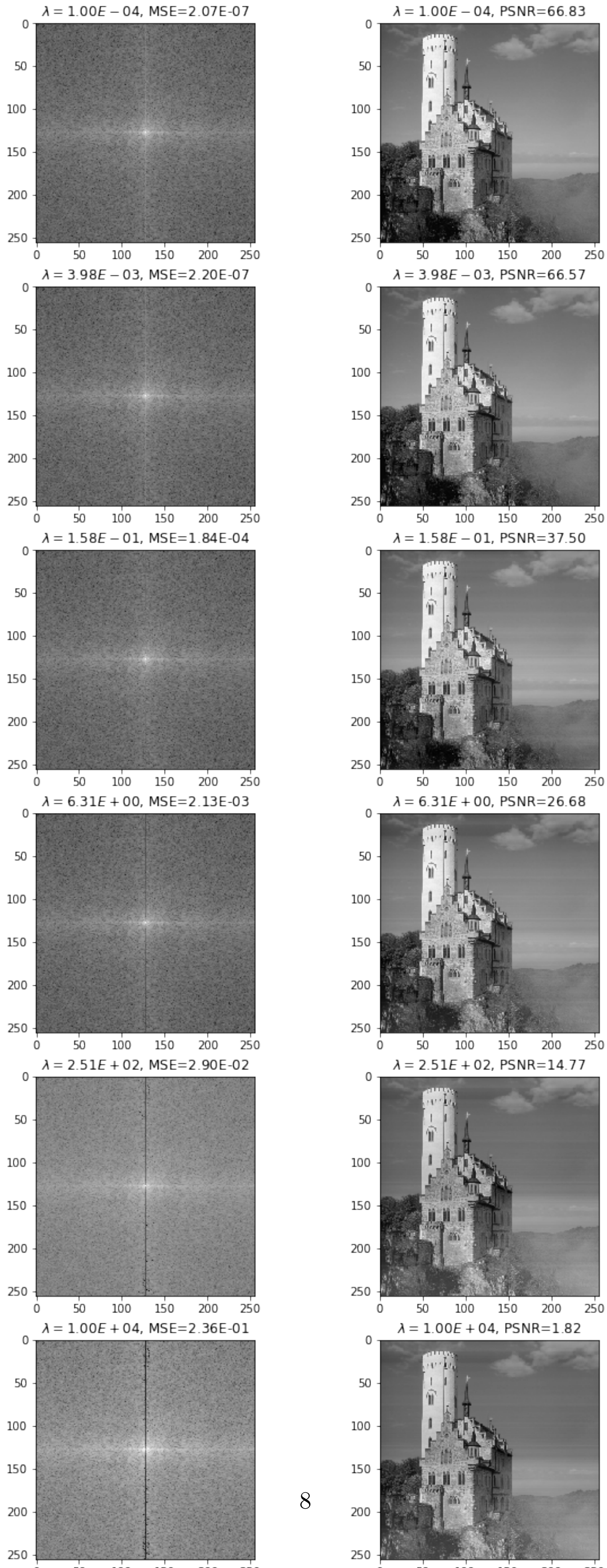
[61]:
```python
#TODO: Fill in the 3 functions below

def loss_fn(Y_hat, Y):
#     frobenius of Y - Y_hat
#     return cp.sum_squares(Y-Y_hat)
    return cp.norm(Y-Y_hat, 'fro')**2

def regularizer(Y_hat):
#     one norm of Y_hat
    return cp.norm(Y_hat, 1)
#     return np.linalg.norm(Y_hat, 1)

def objective_fn(Y_hat, Y, lambd):
    return loss_fn(Y_hat, Y) + lambd*regularizer(Y_hat)

def fft2c(im):
    return np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(im), norm="ortho"))

def ifft2c(ksp):
    return np.fft.fftshift(np.fft.ifft2(np.fft.ifftshift(ksp), norm="ortho"))
```

[66]:
```python
Y_hat = cp.Variable(shape=im.shape, complex=True)
lambd = cp.Parameter(nonneg=True)

obj = cp.Minimize(objective_fn(Y_hat, fft2c(im), lambd))
problem = cp.Problem(obj) #TODO: Create the problem using the functions your␣
↪wrote above

lambd_values = np.logspace(-4, 4, 6)
ft_compressed_imgs = []

for v in lambd_values:
    print("Solving with lambda = {:.2E}".format(v))
    lambd.value = v
    problem.solve(verbose=False, solver=cp.SCS, max_iters=100)
    ft_compressed_imgs.append(ifft2c(Y_hat.value).real)
print("Done!")
```

```
Solving with lambda = 1.00E-04
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 3.98E-03
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 1.58E-01
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 6.31E+00
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 2.51E+02
```

```
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 1.00E+04
WARN: aa_init returned NULL, no acceleration applied.
Done!
```

[67]:
```
plt.figure(figsize=(10,25))
for i in range(len(lambd_values)):
    plt.subplot(6, 2, 2*i+1)
    ft_im = fft2c(ft_compressed_imgs[i])
    plt.imshow(np.log(np.abs(ft_im)), cmap='gray')
    plt.title("$\lambda={:.2E}$, MSE={:.2E}".format(lambd_values[i],␣
 ↪mse(fft2c(im),ft_im)))
    plt.subplot(6, 2, 2*i+2)
    plt.imshow(ft_compressed_imgs[i], cmap='gray')
    plt.title("$\lambda={:.2E}$, PSNR={:.2f}".format(lambd_values[i],␣
 ↪psnr(im,ft_compressed_imgs[i])))
```


λ = 1.00E − 04, MSE=2.07E-07
λ = 1.00E − 04, PSNR=66.83
λ = 3.98E − 03, MSE=2.20E-07
λ = 3.98E − 03, PSNR=66.57
λ = 1.58E − 01, MSE=1.84E-04
λ = 1.58E − 01, PSNR=37.50
λ = 6.31E + 00, MSE=2.13E-03
λ = 6.31E + 00, PSNR=26.68
λ = 2.51E + 02, MSE=2.90E-02
λ = 2.51E + 02, PSNR=14.77
λ = 1.00E + 04, MSE=2.36E-01
λ = 1.00E + 04, PSNR=1.82

Q: What happens to the MSE, PSNR, and image quality as we increase λ? Why?

A:

We get more noise in the signal as we increase lambda.

Q: Qualitatively, how does this method compare to the SVD compression above?

A:

We still get a recognizable signal as we experience more noise.

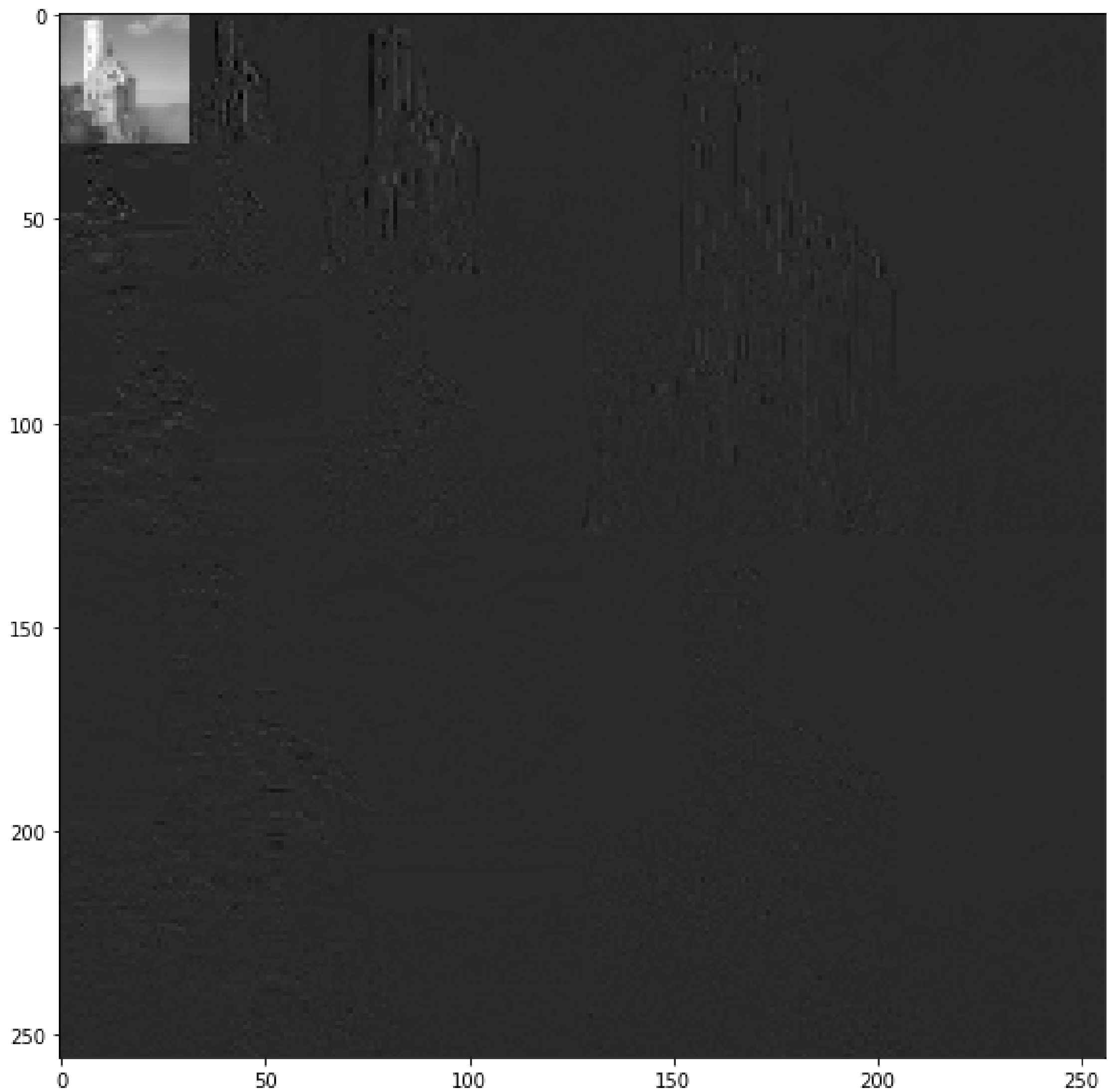
1.4 Wavelet Transform Image Compression

Wavelet transform image compression is very similar to the Fourier transform image compression. In fact, the LASSO problem you used above is the same setup for Wavelet image compression. The only difference now is we are comparing to the wavelet transform of the image instead of the Fourier transform of the image.

Even though the transform itself is out of scope for the class, we wanted to expose you to this form of compression because it is widely used, specifically in in JPEG-2000.

Fill in the `problem` variable two cells below using the same functions you used above.

[68]:
```
coeffs = pywt.wavedec2(im, "haar", level=3)
wave_im, slices = pywt.coeffs_to_array(coeffs)
plt.figure(figsize=(10,10))
plt.imshow(wave_im, cmap="gray")
plt.show()
```

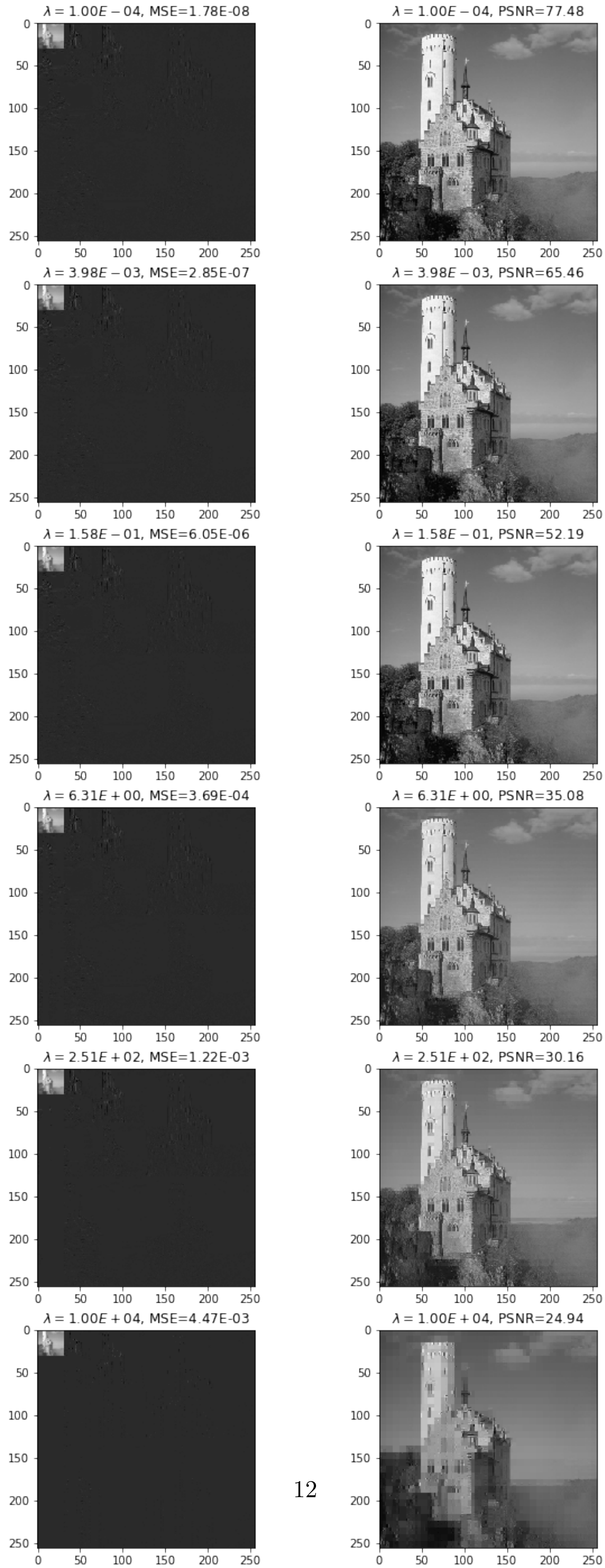


[69]:
```
Y_hat = cp.Variable(shape=im.shape)
lambd = cp.Parameter(nonneg=True)

obj = cp.Minimize(objective_fn(Y_hat, wave_im, lambd))
problem = cp.Problem(obj) #TODO: Create the problem using the functions your␣
↪wrote above

lambd_values = np.logspace(-4, 4, 6)
wave_compressed_imgs = []
wave_imgs = []

for v in lambd_values:
    print("Solving with lambda = {:.2E}".format(v))
```

```
    lambd.value = v
    problem.solve(verbose=False, solver=cp.SCS, max_iters=100)
    Y_hat.value[:32,:32] = wave_im[:32,:32]
    wave_imgs.append(Y_hat.value)
    wave_compressed_imgs.append(pywt.waverec2(pywt.array_to_coeffs(Y_hat.value,␣
 ↪slices, output_format="wavedec2"),"haar"))
print("Done!")
```

```
Solving with lambda = 1.00E-04
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 3.98E-03
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 1.58E-01
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 6.31E+00
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 2.51E+02
WARN: aa_init returned NULL, no acceleration applied.
Solving with lambda = 1.00E+04
WARN: aa_init returned NULL, no acceleration applied.
Done!
```

[70]:
```
plt.figure(figsize=(10,25))
for i in range(len(lambd_values)):
    plt.subplot(6, 2, 2*i+1)
    plt.imshow(wave_imgs[i], cmap='gray')
    plt.title("$\lambda={0:.2E}$, MSE={1:0.2E}".
 ↪format(lambd_values[i],mse(wave_im, wave_imgs[i])))
    plt.subplot(6, 2, 2*i+2)
    plt.imshow(wave_compressed_imgs[i], cmap='gray')
    plt.title("$\lambda={0:.2E}$, PSNR={1:0.2f}".
 ↪format(lambd_values[i],psnr(im, wave_compressed_imgs[i])))
```



$\lambda = 1.00E-04$, MSE=1.78E-08
$\lambda = 1.00E-04$, PSNR=77.48
$\lambda = 3.98E-03$, MSE=2.85E-07
$\lambda = 3.98E-03$, PSNR=65.46
$\lambda = 1.58E-01$, MSE=6.05E-06
$\lambda = 1.58E-01$, PSNR=52.19
$\lambda = 6.31E+00$, MSE=3.69E-04
$\lambda = 6.31E+00$, PSNR=35.08
$\lambda = 2.51E+02$, MSE=1.22E-03
$\lambda = 2.51E+02$, PSNR=30.16
$\lambda = 1.00E+04$, MSE=4.47E-03
$\lambda = 1.00E+04$, PSNR=24.94

Q: For the same λ values, how does the PSNR of the wavelet image compression compare the the fourier image compression?

A:

PSNR is higher for wavelet than fourier image compression for the same values of lambda.

Q: Qualitatively, which form of image compression do you think looks best? Is your answer a function of λ?

A:

It appears fourier image compression looks best. But note I increased the logspace range from between -4 to 1 to -4 to 4, so I could see a more pronounced difference in quality as lambda increased.

[ ]:

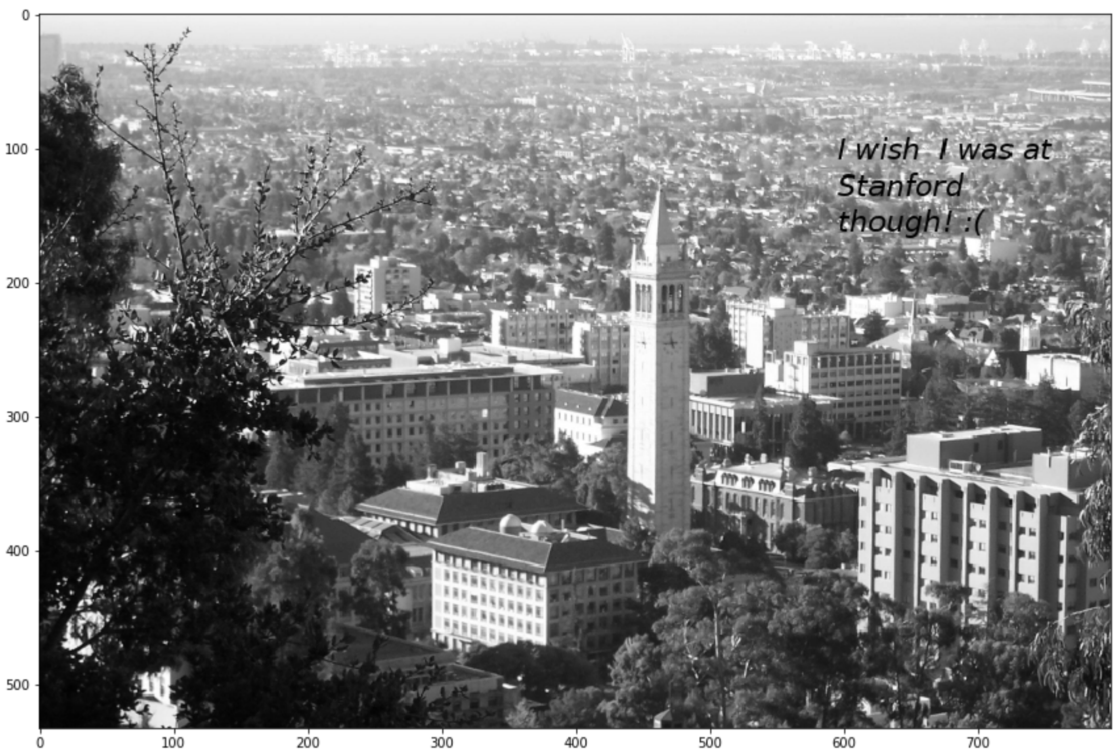
```

```

Total_variation_image_restoration

November 7, 2019

[3]:
```
import cv2
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt
```

[4]:
```
# get image and constraints matrix
corrupted_image_filename = '../data/campanile_img_corrupted.jpg'
constraints_matrix_filename = '../data/constraint_matrix.txt'

u_corr = cv2.imread(corrupted_image_filename, 0)
F = u_corr

A = np.loadtxt(constraints_matrix_filename, delimiter=",")

# visualize image
fig = plt.figure(figsize=(30,10),facecolor='w')
ax = fig.add_subplot(111)
ax.imshow(u_corr, cmap='gray', vmin=0, vmax=255)
plt.show()
```



[9]:
```
rows, cols = F.shape
F_hat = cp.Variable(shape=(rows, cols))

# 1.5
# You complete this ---------------------------------------------------->
# obj = ?? (write objective function)
obj = cp.Minimize(cp.tv(F_hat))
# constraints = ?? (write constriants)
constraints = [cp.multiply(A, F) == cp.multiply(A, F_hat)]
prob = cp.Problem(obj, constraints)

# Use SCS to solve the problem.
# Could take about 10 mins to solve
prob.solve(verbose=True, solver=cp.SCS)
print("optimal objective value: {cv}".format(obj.value))
```

```
----------------------------------------------------------------------------
        SCS v2.1.1 - Splitting Conic Solver
        (c) Brendan O'Donoghue, Stanford University, 2012
----------------------------------------------------------------------------
Lin-sys: sparse-direct, nnz in A = 2549641
eps = 1.00e-04, alpha = 1.50, max_iters = 5000, normalize = 1, scale = 1.00
acceleration_lookback = 0, rho_x = 1.00e-03
```

```
Variables n = 851468, constraints m = 1701604
Cones:  primal zero / dual free vars: 426400
        soc vars: 1275204, soc blks: 425068
WARN: aa_init returned NULL, no acceleration applied.
Setup time: 5.81e+00s
----------------------------------------------------------------------------
 Iter | pri res | dua res | rel gap | pri obj | dua obj | kap/tau | time (s)
----------------------------------------------------------------------------
     0| 1.12e+22  1.35e+22  1.00e+00 -3.88e+29  2.22e+28  1.02e+29  3.48e-01
   100| 2.09e-03  1.19e-03  4.64e-04  8.30e+06  8.31e+06  3.71e-13  1.59e+01
   200| 7.77e-04  2.46e-04  1.34e-04  8.33e+06  8.33e+06  2.65e-09  3.13e+01
   300| 3.86e-04  1.12e-04  5.31e-05  8.34e+06  8.34e+06  2.67e-09  4.67e+01
   400| 1.92e-04  2.86e-05  2.18e-05  8.34e+06  8.34e+06  2.68e-09  6.21e+01
   500| 1.18e-04  5.40e-06  1.19e-05  8.34e+06  8.34e+06  2.68e-09  7.75e+01
   540| 9.92e-05  4.30e-06  9.56e-06  8.34e+06  8.34e+06  5.36e-09  8.37e+01
----------------------------------------------------------------------------
Status: Solved
Timing: Solve time: 8.37e+01s
        Lin-sys: nnz in L factor: 22215955, avg solve time: 8.54e-02s
        Cones: avg projection time: 4.59e-03s
        Acceleration: avg step time: 1.95e-07s
----------------------------------------------------------------------------
Error metrics:
dist(s, K) = 5.6843e-14, dist(y, K*) = 3.3307e-16, s'y/|s||y| = -1.0357e-18
primal res: |Ax + s - b|_2 / (1 + |b|_2) = 9.9215e-05
dual res:   |A'y + c|_2 / (1 + |c|_2) = 4.2952e-06
rel gap:    |c'x + b'y| / (1 + |c'x| + |b'y|) = 9.5562e-06
----------------------------------------------------------------------------
c'x = 8338290.4356, -b'y = 8338449.8023
============================================================================
```

```
        ␣
 ↪---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call␣
 ↪last)

        <ipython-input-9-be3d44398493> in <module>
         13 # Could take about 10 mins to solve
         14 prob.solve(verbose=True, solver=cp.SCS)
    ---> 15 print("optimal objective value: {cv}".format(obj.value))


        KeyError: 'cv'
```

[10]:
```
# visualize result
fig = plt.figure(figsize=(30,10),facecolor='w')
ax = fig.add_subplot(111)
ax.imshow(F_hat.value, cmap='gray', vmin=0, vmax=255)
plt.show()
```

[ ]:
```
```

slalom_std

November 7, 2019

0.0.1 Slalom Problem

[21]:
```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
import pdb
```

Here, we give you the inputs for the problem : the x, y, and c coordinates as given in the problem statement (refer to the table)

[7]:
```
x0, y0 = 0, 4
x1, y1, c1 = 4, 5, 3
x2, y2, c2 = 8, 4, 2
x3, y3, c3 = 12, 6, 2
x4, y4, c4 = 16, 5, 1
x5, y5, c5 = 20, 7, 2
x6, y6 = 24, 4

xs = [x0,x1,x2,x3,x4,x5,x6]
ys = [y0,y1,y2,y3,y4,y5,y6]
cs = [0,c1,c2,c3,c4,c5,0]
```

Initialize the variables we are optimizing over here! You should be using cvx.Variable() to create the variables you are optimizing over.

[42]:
```
# Initialize any and all cvxpy variables that you will use
y = cp.Variable(7)
```

[43]:
```
# Now, we put in our constraints: the format should be as follows.

constraints = [y[i] >= ys[i]-cs[i]/2 for i in range(7)] + [y[i] <= ys[i]+cs[i]/
 ↪2 for i in range(7)]
# constraints = [
#     # constraint 1,
#     y[0] >= y0-c0/2
#     # constraint 2,
#     # etc.
```

```
# ]
```

[44]:
```
# Here, input your objective function. It should be of the form:
def objective_fn(y):
    return sum((xs[i+1]-xs[i])**2 + (y[i+1]-y[i])**2 for i in range(6))

# cp.norm(cp.vstack([x[1]-x[0],y[1]-y[0]]),2)

# obj = cp.Minimize( YOUR OBJECTIVE FUNCTION HERE )
obj = cp.Minimize(objective_fn(y))
```

[45]:
```
# creating the optimization problem here, putting together the objective and␣
↪the constraints
prob = cp.Problem(obj, constraints)

optimal_path_length = prob.solve() # this will output your optimal path length
```

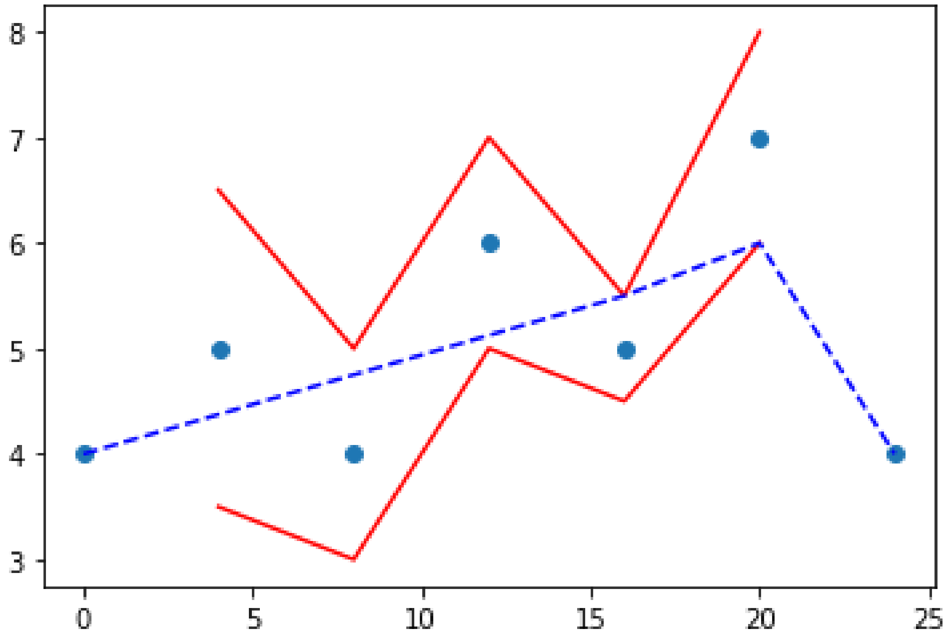
Just check that your optimization variables respect the constraints here (OPTIONAL, but good for debugging)

0.0.2 Print out the coordinates of the path (this should be an array with 7 tuples denoting the (x,y) position of where the skiier should cross

[47]:
```
# path = [(x0,y0), ..., (x6,y6)]
path = [(xs[i],y[i].value) for i in range(7)]
```

[48]:
```
x = np.array([x0, x1, x2, x3, x4, x5, x6])
y = np.array([y0, y1, y2, y3, y4, y5, y6])
c = np.array([c1, c2, c3, c4, c5])
plt.figure()
plt.scatter(x,y)
plt.plot(x[1:-1], y[1:-1]+c/2, c="r")
plt.plot(x[1:-1], y[1:-1]-c/2, c="r")
plt.plot(*zip(*path), "b--")
```

[48]:
```
[<matplotlib.lines.Line2D at 0x7f30d9475908>]
```



[49]:
```
print("{0:<3} {1:<3}".format("x", "y"))
for p in path:
    print("{0:<3} {1:.3f}".format(p[0], p[1]))
```

```
x   y
0   4.000
4   4.375
8   4.750
12  5.125
16  5.500
20  6.000
24  4.000
```

[ ]: