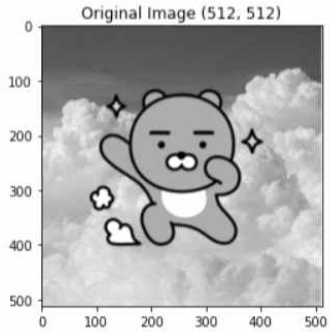
Homework #3

영상신호처리
2020451142 정수영

1. Get a grey level image which size is N*N($N=2^n$). and partition to 8*8 sub images.
- 이미지 cloudRyan.png를 512*512의 흑백 이미지로 변환한다.

```
[4] N = 512
    image = Image.open("/content/drive/MyDrive/Colab Notebooks/영상처리/datas/cloudRyan.png").resize((N, N)).convert('L')
    original_f = np.array(image)
    plt.imshow(original_f, cmap="gray", vmin=0, vmax=255 )
    plt.title("Original Image " + str(original_f.shape))
    plt.show()  # 이미지 출력
```



2. Apply DCT to these sub images, and get the transformed image D with DCT coefficients for elements.
- Cosine Fourier 공식을 이용하여 DCT 함수를 정의한다.

```
[7] # Cosine Fourier Formular
    def CosineFourier(N, m, n):
      if m == 0: c = 1/np.sqrt(2)
      else: c = 1
      return np.sqrt(2/N) * c * np.cos((2*n+1) * m * np.pi/(2*N))

    # DCT (Discrete Cosine Transform)
    def DCT(N):
      U = []
      for i in range(N):
        row = []
        for j in range(N):
          row.append(CosineFourier(N, j, i))
        U.append(row)
      return np.array(U)
```

- 8*8 서브이미지들을 처리할 D=DCT(8)를 만들고 적용하여 Frequency Domain의 이미지를 생성한다.

```
[8] # Ds : The 8*8 image D list (Discrete Cosine Fourier Transformed matrix)
    Ds = []
    D8 = DCT(8)
    for f in f_s:
      Ds.append(np.dot(D8, np.dot(f, D8)))  # D8 = D8.T
    Ds = np.array(Ds)
```

3. From D, keep the coefficient values for only upper left triangular region and set zeros for lower right region to approximate the image. (That is, only half of data is used.)

- Upper Left Triangular 부분을 구한다.

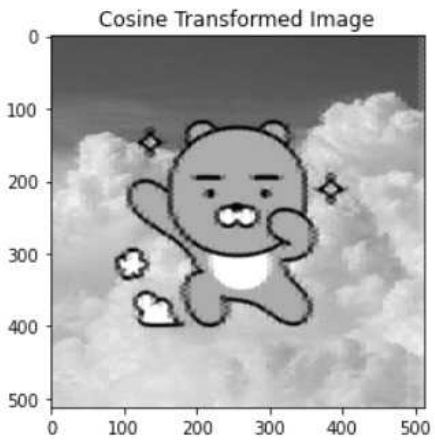
```
[9]  # Upper Left Triangular region
     def UpperLeft(matrix):
       N = len(matrix)
       ret = np.zeros(shape=(N,N), dtype=np.float128)
       for i in range(N):
         for j in range(N):
           if i >= j:
             ret[i][j] = matrix[i][j]
       return ret
```

```
[17] # UpperLeft를 구한다.
     LDs = []
     for D in Ds:
       LDs.append(UpperLeft(D))
     #  LDs.append(D)
```

4. Take Inverse DCT to get the approximated image.

- DCT의 Inverse를 이용해 Spacial Domain 영역의 이미지로 변환한 후 출력한다.

```
[18] # Spatial Domain으로 변환한다.
     ILDs = []
     for LD in LDs:
       ILDs.append(np.dot(np.linalg.inv(D8), np.dot(LD, np.linalg.inv(D8))))
     ILDs = np.array(ILDs)

     # 그림을 그려준다.
     plt.imshow(GetNxN(ILDs), cmap='gray', vmin=0, vmax=255 )
     plt.title("Cosine Transformed Image")
     plt.show()
```



2'. Get the covariance matrix (64*64) of sub images.

- Autocorrelation을 이용하여 Autocovariance Matrix를 만들 함수를 정의한다.

```
[13] # Autocorrelation
     def R(g):
       N, _ = g.shape
       R = []
       for term in range(N):  # term = 0 to 15
         tempR = 0
         for i in range(N):
           tempR += (g[i][0] * g[(i+term)%N][0])
         R.append(tempR/N)
       return np.array(R)

       # Autocovariance Matrix
     def C(g):
       ret = []
       autocov = (R(g) - np.mean(g)**2)
       N = autocov.shape[0]
       for crit in range(N):
         ret.append(np.concatenate((autocov[-crit:], autocov[:-crit]), axis=None))
       return np.array(ret)
```

3'. Calculate the corresponding 64 eigenvectors and eigenvalues.

- 앞서 구한 Autocovariance Matrix C(g)에 SVD를 적용하여 eigenvector와 eigenvalue를 구해 적용하는 KLT 함수를 정의한다.

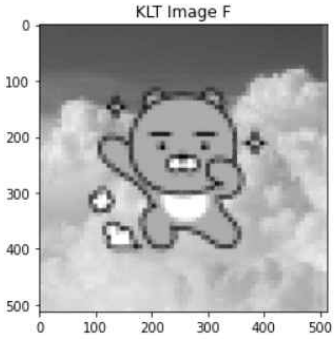
```
def KLT(gs, n):
  ret = []
  for g in gs:
    U, s, V = np.linalg.svd(C(g))
    for i in range(s.shape[0]):
      if i >= n:
        U[i] *= 0
    temp = np.dot(U.T, g-np.mean(g)) + np.mean(g)
    temp[temp<0] = 0
    ret.append(temp)
  return np.array(ret)
```

4'. Approximate the image by taking the 4 most dominant eigenvectors among 64 eigenvectors. (That is, only four eigenimages are used.)

- KLT 함수에 매개변수 값 n=4를 넣어 주면 4개의 가장 큰 eigen 값들만 적용해 이미지를 출력한다.

```
[14] # Ks : THe 8*8 images KLT list (KLT Transformed matrix)
     Ks = Shorten(KLT(Flatten(f_s), 4))

     # Get KLT image directly
     plt.imshow(GetNxN(Ks), cmap='gray', vmin=0, vmax=255)
     plt.title("KLT Image F")
     plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in ubyte_scalars
```

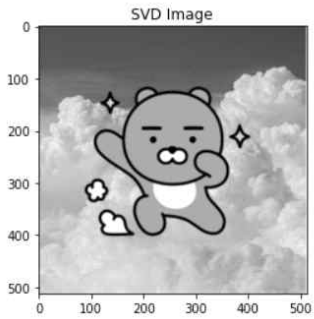


2". Get the matrices gg^T and g^Tg, and corresponding 8 eigenvectors u and v.
- SVD 함수를 정의한다. 함수 내에 eigenvector U와 V를 구하는 것은 numpy의 내장함수를 사용한다.

```
[15] # SVD 함수를 만들어 준다.
     def SVD(g, n=4):
       A = np.array(g).astype(np.float64)
       U, s, V = np.linalg.svd(A)
       S = np.zeros(A.shape)
       S_sorted = np.sort(s)[::-1]
       for i in range(len(S)):
     #    if s[i] >= S_sorted[n-1]:  # eigen value 중 n번째 이상의 값만 필터링
         if i < n:
           S[i][i] = s[i]
       return np.dot(U, np.dot(S, V))
```

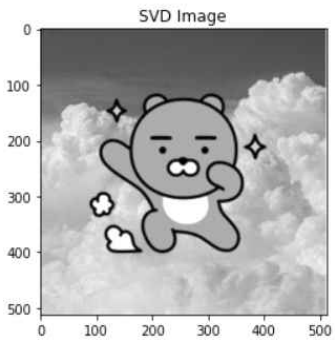
3". Represent the original image with Singular Value Decomposition.
- 8 개의 eigenvalue를 모두 사용한 SVD 이미지는 다음과 같다.

```
[21] # Ss : The 8*8 image SVD list (SVD Transformed matrix)
     Ss = []
     for f in f_s:
       Ss.append(SVD(f, 8))
     Ss = np.array(Ss)

     plt.imshow(GetNxN(Ss), cmap='gray', vmin=0, vmax=255)
     plt.title("SVD Image")
     plt.show()
```



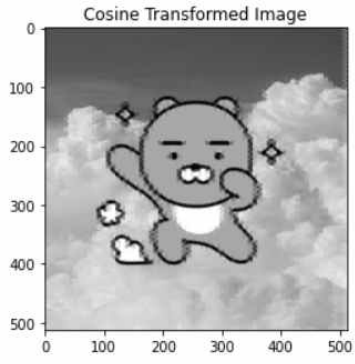
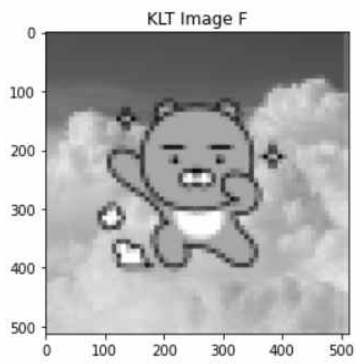
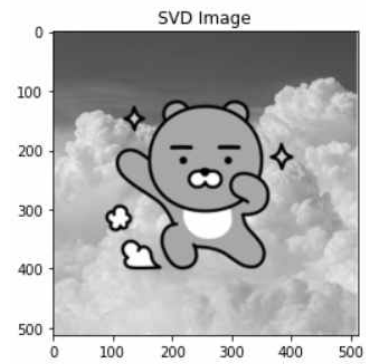
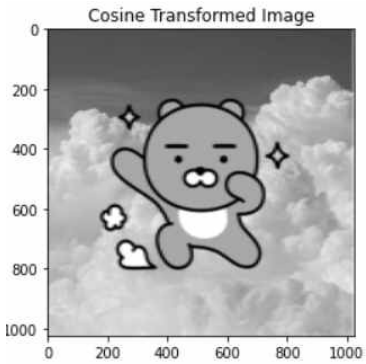
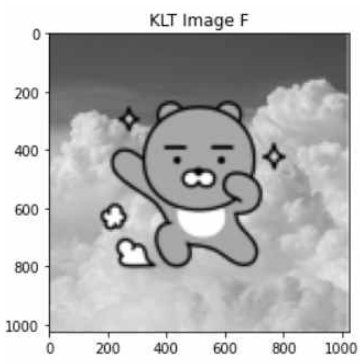
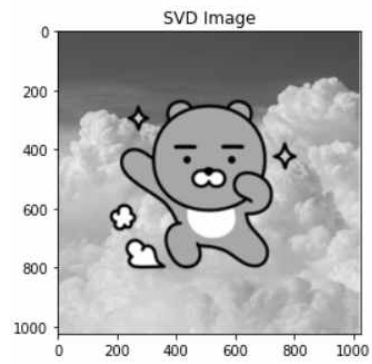
4". Approximate the image by taking off the 4 smallest eigenvalues among 8 eigenvectors. (That is, only half of information is used.)
- 상위 4개의 eigenvalue 값만 사용한 SVD 이미지는 다음과 같다.

```
[16] # Ss : The 8*8 image SVD list (SVD Transformed matrix)
     Ss = []
     for f in f_s:
       Ss.append(SVD(f, 4))
     Ss = np.array(Ss)

     plt.imshow(GetNxN(Ss), cmap='gray', vmin=0, vmax=255)
     plt.title("SVD Image")
     plt.show()
```



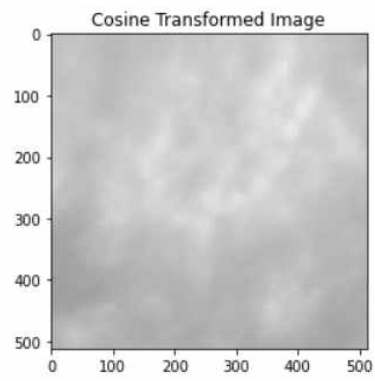
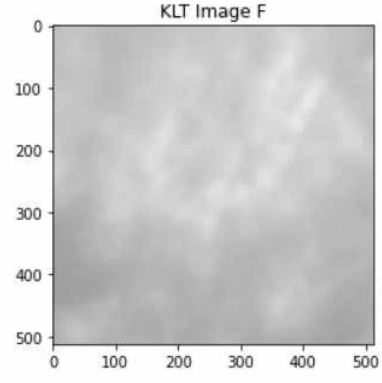
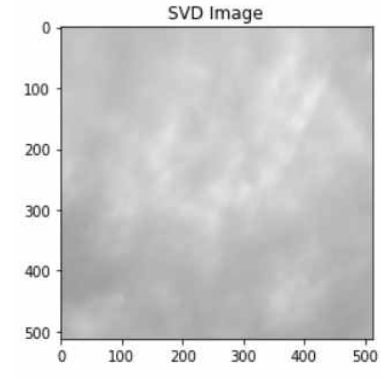
5. Compare the images from 4 , 4' and 4".

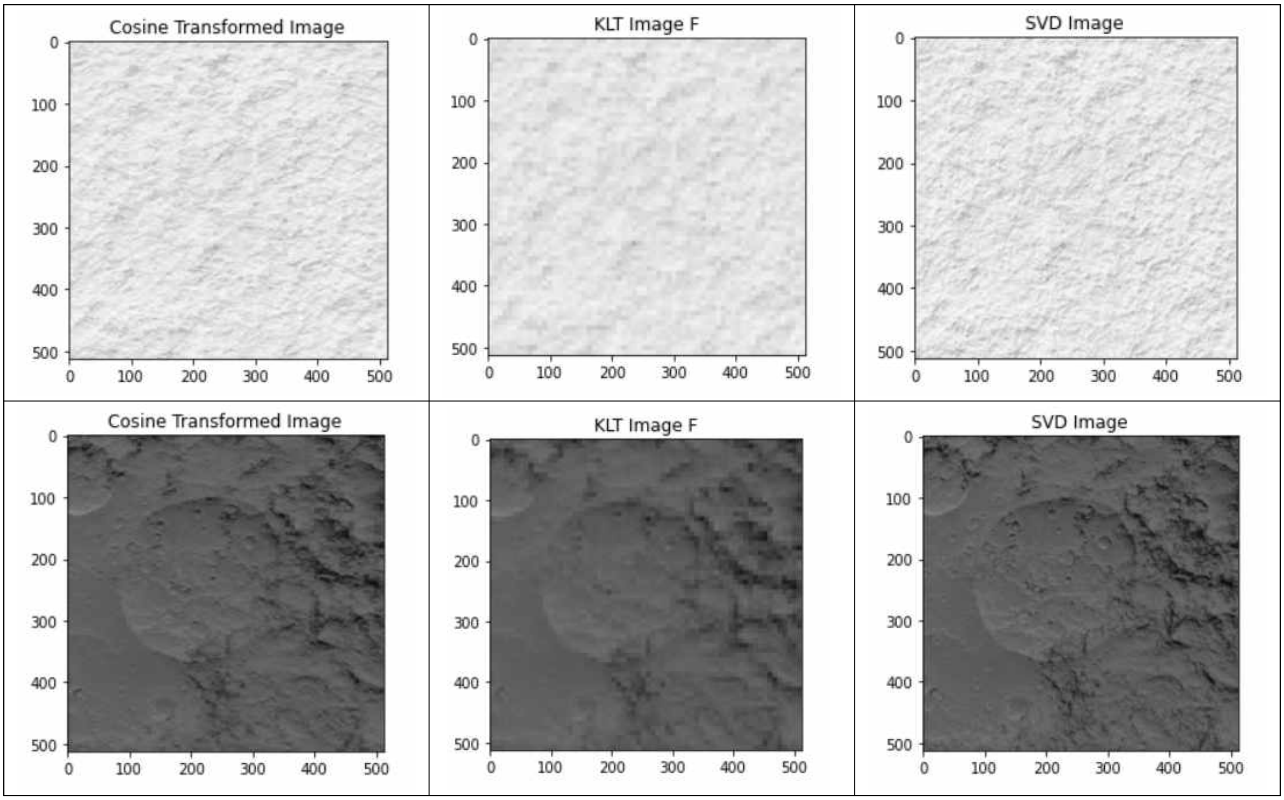
- 선명한 순으로 비교하면 SVD > DCT > KLT 이다.
- 라이언 캐릭터 같이 경계가 분병한 부분은 차이를 확실히 느낄 수 있지만 구름 같은 텍스쳐는 화질 저하가 얼핏 보아서는 눈에 띄지 않는다.
- 화질이 높아질수록 그 차이는 많이 없어진다. 특히 KLT는 전반적으로 부드러운 느낌으로 이미지가 변환되었을 확인할 수 있다.

N	4. DCT	4'. KLT	4". SVD
512			
1024			

6. Repeat same process to different images (if possible, random texture images.) and discuss the results.

- 모두 512*512로 크기를 고정하여 cloud_texture.jpg, wall_texture.jpg, mars_surface.jpg 이미지를 변환해보았다.

4. DCT	4'. KLT	4". SVD
		



- Texture 부분은 DCT와 비교가 힘들 정도로 성능이 괜찮은 것을 확인할 수 있다. 특히 cloud_texture 부분에서는 다른 이미지들과 거의 차이가 없음을 확인할 수 있다.
- 하지만 선명한 윤곽이 있으면 SVD에 비해서는 성능이 떨어진다. mars_surface에서도 표면 윤곽이 선명한 부분은 차이가 나지만 표면 윤곽이 흐지부지한 부분들은 부드럽게 복원된 것을 알 수 있다.
- KLT는 다른 변환들에 비하여 모호하고 랜덤한 텍스쳐들에 대한 변환 능력이 뛰어남을 확인할 수 있었다.
- 흐릿한 이미지만 잘 캐치한다는 것은 어쩌면 선명한 윤곽선을 더 잘 인식한다고도 볼 수 있다. 그래서 인터넷에 보면 KLT를 이용한 인체 동작 인식이나 윤곽 인식 툴이 많은 것을 확인할 수 있었다.

※ 사용된 이미지들과 코드는 datas/ 폴더에 넣어두겠습니다. 코드는 다음 Colab에서 결과와 함께 확인하실 수 있습니다.
https://colab.research.google.com/drive/1OdCljPMsnDMiIkR0_qRfrwDJ8gdicvKZ

감사합니다.