CSC2529-hw3

1010171181 Xinran Zhang

October 2023

1 Task1

1.1 low-pass filter

In this part, I implemented low-pass filtering for three Gaussian convolution kernels with 3 different standard deviations: 0.1, 1 and 10. When we implementing low-pass filter, we set the kernel size 9 times larger than the standard deviation.

Also, I used spatial-domain and fourier-domain respectively. Here are the resulting images. In figure 1, the 3 pictures in the first line show the resulting pictures using spatial-domain and the 3 pictures in the second line show the resulting pictures using fourier-domain. And the other 3 pictures in the third line show the OFTs with different sigmas.

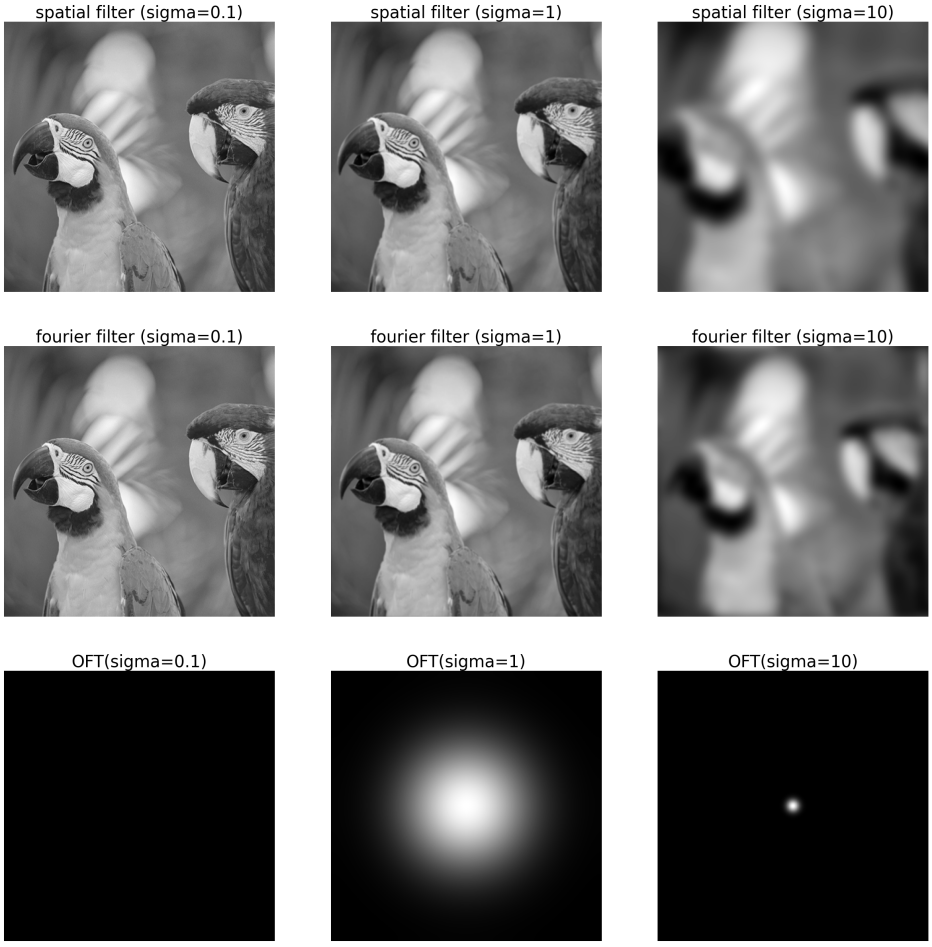

Figure 1: Images Processed with Different Filter Sizes

From the Figure 2, it is evident that in the first method, there is a significant difference in time when processed with different sigma values, with orders of magnitude difference. Whereas, in the second method Fourier-domain, the times for different sigma values are similar, with no significant gap and independent of kernel size.

I think that is because in the Fourier domain approach, both the input image and the convolution kernel undergo Fourier transforms, resulting in convolution becoming a straightforward multiplication operation with a consistent time complexity.

But when I applied spatial domain, spatial filtering is going to be a function of the kernel size, because the convolution would be a bunch of addition, and those additions is a function of the number of elements that you have in your kernel. So the time complexity varies when different sigma values (different kernel sizes) are employed. When the kernel size is very small, the kernel is relatively small and spatial-domain convolution may be very fast. But as the kernel size grows, the Gaussian kernel is much larger, requiring a greater number of computations and thereby affecting processing time.

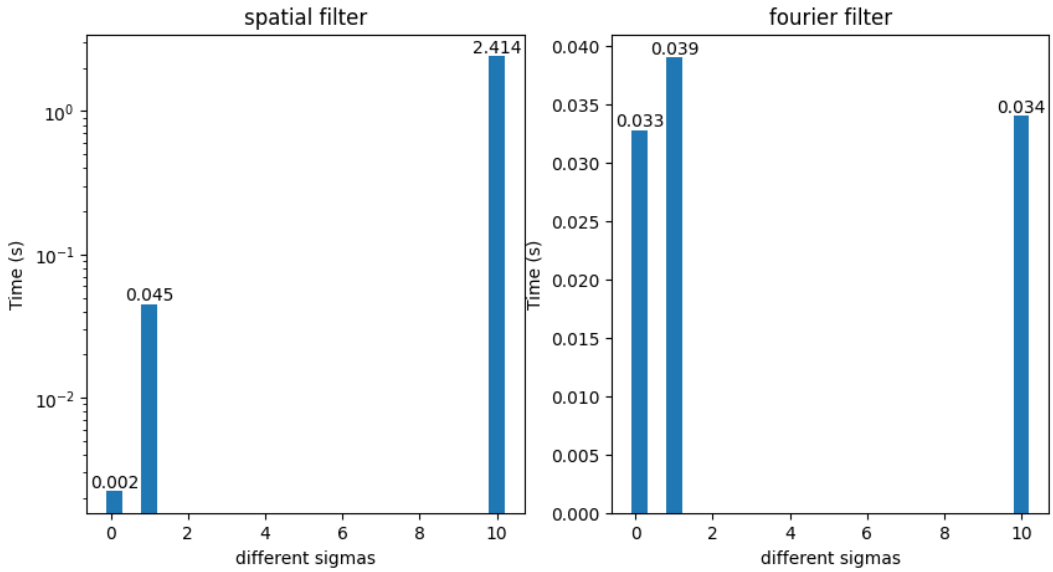

Figure 2: Images Processed with Different Filter Sizes

1.2 high-pass filter

Then, I changed to high-pass filter instead of low-pass filter. I implement high pass filtering using these formulas.

$$Spatial - domain : I = I - I * PSF_{LP} \tag{1}$$

$$Fourier - domain : \widetilde{I} = \widetilde{I} * (1 - OTP_{LP}) \tag{2}$$

I maintained the choice of sigma values as 0.1, 1, and 10, but used the same kernel size of 101 $\times$ 101 pixels for each of them. Here are the resulting images and timings in figure 3. Similar with figure1, the first row of Figure 3 showcases three output images processed using the spatial-domain method, each with a different sigma. The second row displays the resulting images obtained through the Fourier-domain approach, while the last row shows the OFTs.

The timings are shown in figure4, we can find that in this scenario, there is no significant difference in the time it takes to perform convolutions with different sigma values, whether using the spatial-domain or Fourier-domain method. I think that in the spatial-domain method, the reason there is no significant difference in processing time when using different sigma values is that we employed a fixed kernel size, which was set to 101.

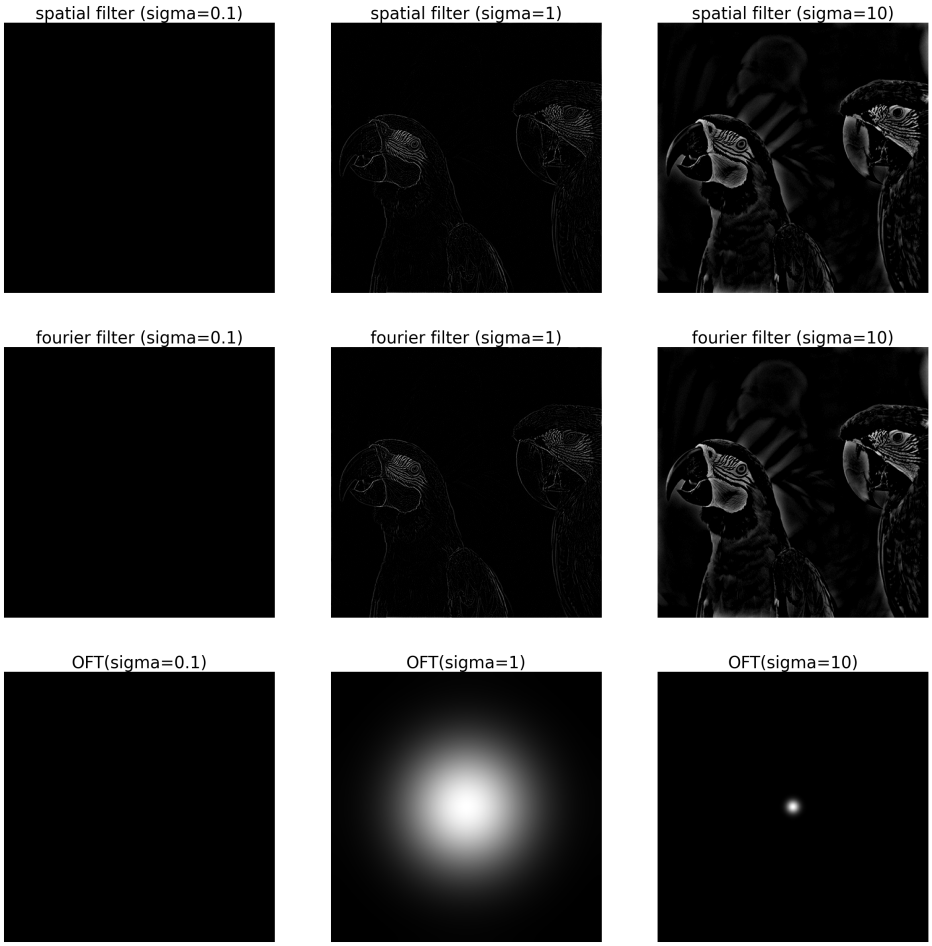

Figure 3: Images Processed with Different Filter Sizes

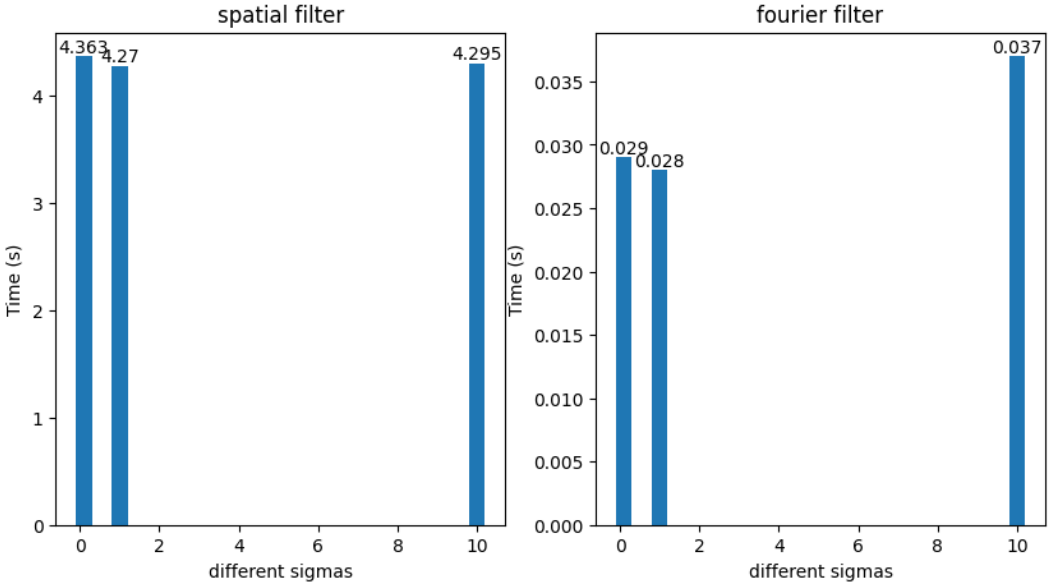

Figure 4: Images Processed with Different Filter Sizes

2 Task2

I blurred the image by using a Gaussian PSF with a standard deviation of 5.

```
c = fspecial_gaussian_2d((35, 35), 5.)
blur = convolve2d(img, c, mode='same', boundary='wrap')
```

And then I added random, additive Gaussian noise with the following standard deviations to the blurred image: 0, 0.001, 0.01 and 0.1. Then, I used two different methods to inversely filter the blurred image.

2.1 division in the Fourier domain

In this method, I simply divided the Fourier-transformed blurred image by cFT and then performed an inverse Fourier transform. The results are as shown in the figure5. The first row shows the blurred image, each with different random noise added to them. The second row shows the divided resulting images.

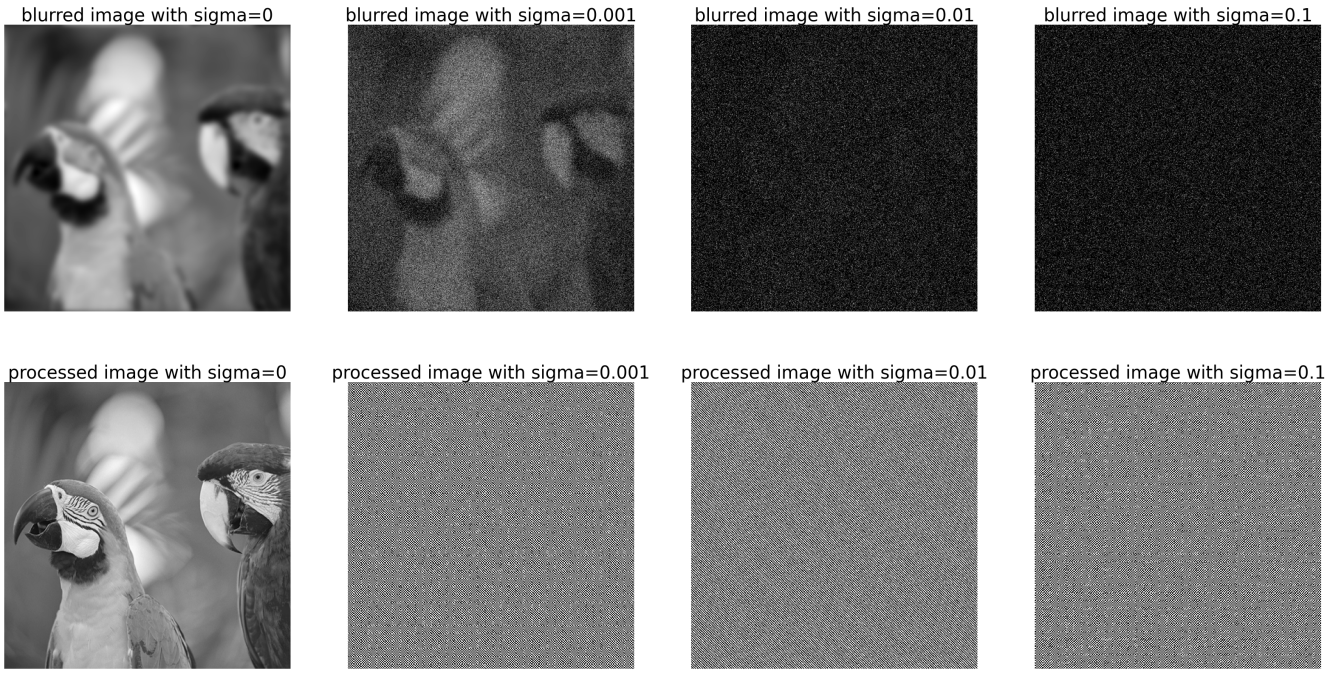

Figure 5: Images Processed with Different Filter Sizes

We can observe that, except when random noise is zero, which is equivalent to inversely transforming the blurred image, resulting in a clear original image. When random noise is not zero, the results of this method are very poor.

There are the peak signal-to-noise ratio (PSNR) for each resulting images. Firstly, I computed the PSNR by using the blurred image and the resulting images. Then, I computed the PSNR by using the original image and the resulting images. Here are the results.

We can find that, whether we compare with the original image or with the blurred image, only in the case of zero noise, PSNR is a positive value. In all other cases, regardless of how much noise is added, PSNR is negative, indicating poor processing quality.

```
when the sigma of noise is  0 , the psnr is  73.12423332688242
when the sigma of noise is  0.001 , the psnr is  -156.49377861656293
when the sigma of noise is  0.01 , the psnr is  -176.88245528338643
when the sigma of noise is  0.1 , the psnr is  -190.48925667770897
```

Figure 6: PSNR for each noise, comparing with blurred image; division in the Fourier domain

```
when the sigma of noise is  0 , the psnr is  139.30137606397096
when the sigma of noise is  0.001 , the psnr is  -162.51098423408408
when the sigma of noise is  0.01 , the psnr is  -177.34280979247316
when the sigma of noise is  0.1 , the psnr is  -188.93718600271526
```

Figure 7: PSNR for each noise, comparing with original image; division in the Fourier domain

2.2 Wiener de-convolution

Because of the poor processing quality, We change to wiener de-convolution. In this method, unlike the previous simple division, the calculation follows the formula as shown below.

It's worth noting that $\frac{|H|^2}{|H|^2+\frac{1}{SNR}}$ here is still a matrix. Furthermore, we should also consider a special case where, when the noise(σ_{noise}=0) is 0, SNR cannot be computed normally. In such instances, I set SNR to math.inf.

$$SNR = \frac{\overline{I}}{\sigma_{noise}} \tag{3}$$

$$H^{'} = \frac{1}{H} * \frac{|H|^2}{|H|^2 + \frac{1}{SNR}} \tag{4}$$

I multiplied the Fourier-transformed blurred image by $H^{'}$ and subsequently conducted an inverse Fourier transform. Below, you can see the resulting images in figure 8.

We can observe that, although the image deconvolution results are still not ideal when there is noise, there has been significant improvement compared to the previous method. When the noise is very small (e.g., 0.001), the restored image closely resembles the original.

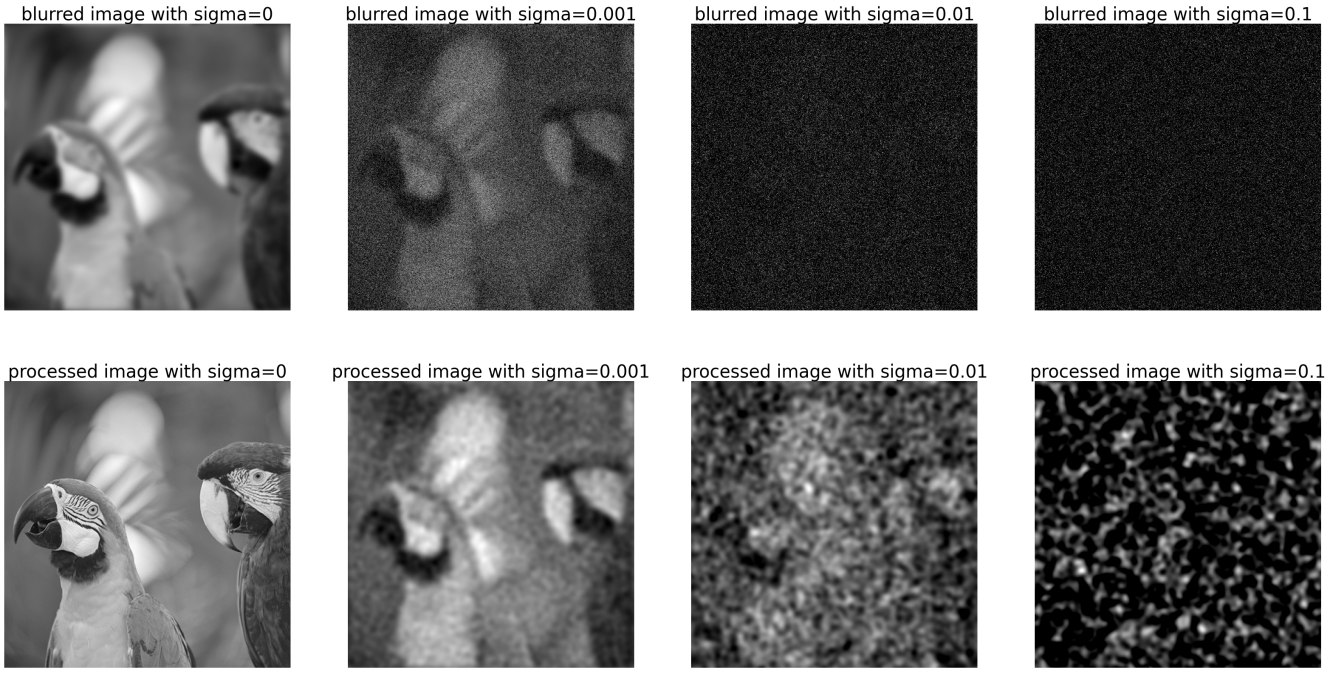

Figure 8: Images Processed with Different Filter Sizes

Similarly, I separately calculated the PSNR when compared to the original image and when compared to the blurred image. Here are the results.

We can observe a pattern similar to that of the previous method in the PSNR results; the smaller the noise, the higher the PSNR. However, the difference here is that for all four levels of noise, the obtained PSNR values are greater than 50 in this method, indicating better processing results compared to Method 1.

```
when the sigma of noise is  0 , the psnr is  73.12423332688242
when the sigma of noise is  0.001 , the psnr is  63.1710685405132 54
when the sigma of noise is  0.01 , the psnr is  55.347799216966884
when the sigma of noise is  0.1 , the psnr is  53.9873137573666
```

Figure 9: PSNR for each noise, comparing with blurred image; Wiener de-convolution

```
when the sigma of noise is  0 , the psnr is  139.30137606397096
when the sigma of noise is  0.001 , the psnr is  62.64916646321642
when the sigma of noise is  0.01 , the psnr is  55.25353327777805
when the sigma of noise is  0.1 , the psnr is  53.86802938970271
```

Figure 10: PSNR for each noise, comparing with original image; Wiener de-convolution

3 Task3

First of all, whether you want to implement GD or SGD, we need to calculate the gradient of this norm, as shown in the formula below. And move in the direction of the negative gradient.

$$\nabla_x \left[\frac{1}{2} \|Ax - b\|_2^2 \right] = A^T Ax - A^T b \tag{5}$$

$$x^{k+1} = x^k - \alpha * \nabla f(x^k) \tag{6}$$

The SGD method is very similar with GD. We just need to select a batch of rows randomly from the matrix A before doing each step of gradient descent. Initially, I executed the program on my personal laptop, and the timing curves exhibited not smooth at all. Subsequently, I switched to the remote server, resulting in smoother timing curves as depicted in Figure 11. Details are shown on my personal website: ssh to server and some git commands.

The left picture plots the residual vs optimization iteration for GD, SGD and SVD with batch sizes of 10, 100, and 1000. The right picture shows the residuals vs. execution time for GD and SGD with batch sizes of 10, 100 and 1000.

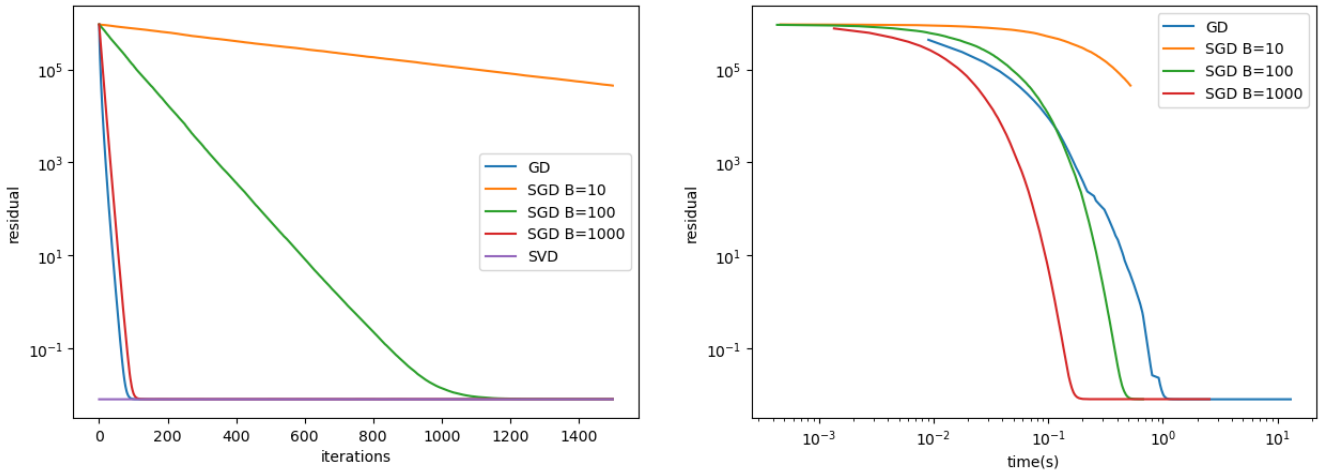

Figure 11: Images Processed with Different Filter Sizes

Compared to GD (Gradient Descent), we can observe that SGD (Stochastic Gradient Descent) requires a relatively higher number of iterations to converge. Excluding SVD, GD converges with the fewest iterations, and the convergence of SGD with a batch size of 1000 is similar to GD but demands slightly more iterations. However, with a batch size of 100, SGD requires significantly more iterations to converge, and for a batch size of 10, SGD is even slower, to the extent that it still notably fails to converge even after 1500 iterations.

In the right figure, we can observe the convergence times of different methods. From this, we can discern that SGD with a batch size of 1000 achieves the shortest convergence time, followed by SGD with a batch size of 100, which is shorter than GD's convergence time. SGD with a batch size of 10 exhibits the slowest convergence speed.

There is obvious a tradeoff with using different batch size in the optimization. In general, a larger batch size typically implies fewer iterations for convergence but longer execution time per iteration. Conversely, a smaller batch size indicates more iterations, but each iteration is executed more quickly.

When the size becomes too small, it can lead to the need for an excessive number of iterations for convergence, thus causing the convergence time to be slower. When the size is too large, although the convergence may require a small number of iterations, each iteration is computationally slow, resulting in an overall slow convergence time.

Hence, we need to find a batch size that strikes a balance between the time it takes for a single iteration and the number of iterations, resulting in the overall minimum convergence time.

4 Bonus

In this section, I derived $\nabla_x \left[\frac{1}{2} \|Ax - b\|_1\right]$ and implement a Python function that calculates this gradient given A, x and b. Here are the formulas.

$$\begin{aligned} A &= (\alpha_1^T, ..., \alpha_m^T) \\ x &= (x_1, ..., x_n) \\ b &= (b_1, ..., b_m) \end{aligned} \tag{7}$$

$$\frac{1}{2} \|Ax - b\|_1 = \frac{1}{2} * \sum_{i=1}^{m} |\alpha_i^T * x - b_i| \tag{8}$$

$$\frac{\partial |z|}{\partial x} = sign(z) * \frac{\partial z}{\partial x} \tag{9}$$

$$\begin{aligned} \nabla_x \left[\frac{1}{2} \|Ax - b\|_1\right] &= \frac{1}{2} * \nabla_x \sum_{i=1}^{m} |\alpha_i^T * x - b_i| \\ &= \frac{1}{2} * \sum_{i=1}^{m} \nabla_x |\alpha_i^T * x - b_i| \\ &= \frac{1}{2} * \sum_{i=1}^{m} sign(\alpha_i^T * x - b_i) * \frac{\partial(\alpha_i^T * x - b_i)}{\partial x} \end{aligned} \tag{10}$$

In python we can use matrix multiplication: 0.5 * A.T @ np.sign(A @ x - b) for equation 10 and use 0.5 * np.linalg.norm(A @ x - b, ord=1) to calculate the residuals.

When one of the entries of Ax-b is zero, the function sign$(\alpha_i^T * x - b_i)$ treat it as 0. The return of question 10 is a vector which will be used in the following gradient descent.

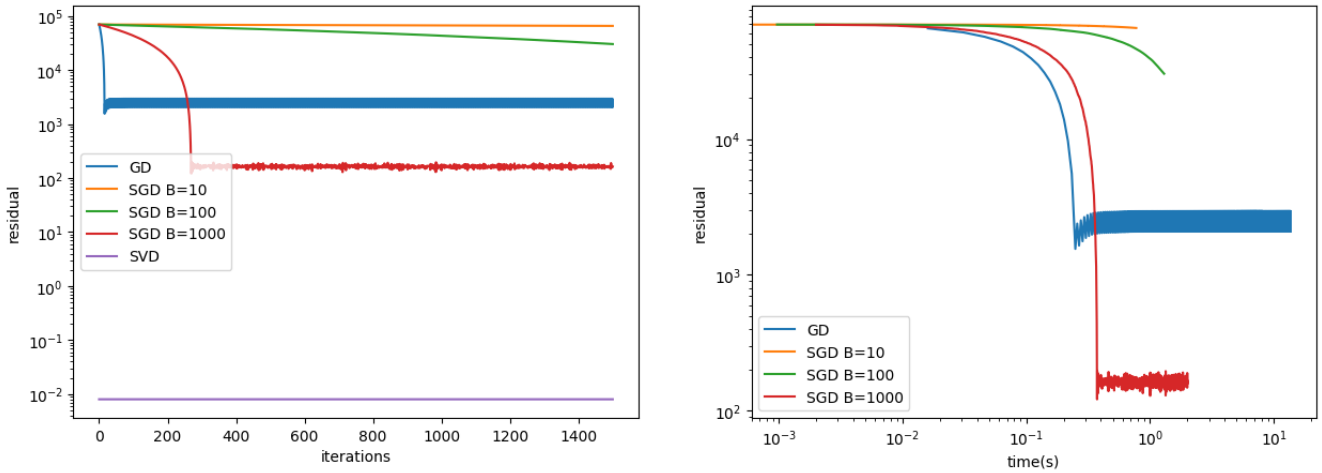

Figure 12: Images Processed with Different Filter Sizes

A code

A.1 Task1

A.1.1 low-pass filter

```
import numpy as np
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import skimage.io as io
from scipy.signal import convolve2d
from pypher.pypher import psf2otf
import matplotlib.pyplot as plt
import time
from fspecial import fspecial_gaussian_2d

img =io.imread('birds_gray.png', as_gray=True).astype(float) /255
spatial_times =[]
fourier_times =[]

fig, axs =plt.subplots(nrows=3, ncols=3, figsize=(30, 30))
sigmas =[0.1, 1, 10]
for k in range(len(sigmas)):
    sigma =sigmas[k]
    filtSize =np.ceil(9 *sigma).astype(int)
    lp =fspecial_gaussian_2d((filtSize, filtSize), sigma)

    ### Your code here ###
    spatial_start =time.time()
    spatial_result =convolve2d(img, lp, mode='valid', boundary='wrap')
    spatial_times.append(time.time() -spatial_start)
    axs[0, k].imshow(np.clip(spatial_result, a_min=0, a_max=1), cmap='gray')
    axs[0, k].set_title(f'spatial filter (sigma={sigma})', fontsize=30)
    axs[0, k].axis('off')

    fourier_start =time.time()
    img_ft =fft2(img)
    img_ftshift =fftshift(img_ft)
    ft =psf2otf(lp, img.shape)
    ftshift =fftshift(ft)
    ft_mid =img_ftshift *ftshift
    ishift =ifftshift(ft_mid)
    ft_result =ifft2(ishift).real

    fourier_times.append(time.time() -fourier_start)
    axs[1, k].imshow(np.clip(ft_result, a_min=0, a_max=1.), cmap='gray')
    axs[1, k].set_title(f'fourier filter (sigma={sigma})', fontsize=30)

    axs[1, k].axis('off')

    axs[2, k].imshow(ftshift, cmap='gray')
    axs[2, k].set_title(f'OFT(sigma={sigma})', fontsize=30)
    axs[2, k].axis('off')
fig.savefig('task1_a_filter.png', bbox_inches='tight')
plt.show()

fig1, axs =plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
axs[0].bar(sigmas, spatial_times, width=0.4, label='Spatial Convolution')
for i, j in zip(sigmas, spatial_times):
    axs[0].text(i, j, round(j, 3), ha='center', va='bottom')
axs[0].set_title(f'spatial filter')
axs[0].set_xlabel('different sigmas')
axs[0].set_ylabel('Time (s)')

axs[1].bar(sigmas, fourier_times, width=0.4, label='Fourier Convolution')
for i, j in zip(sigmas, fourier_times):
    axs[1].text(i, j, round(j, 3), ha='center', va='bottom')
axs[1].set_title(f'fourier filter')

axs[1].set_xlabel('different sigmas')
axs[1].set_ylabel('Time (s)')
```

```
axs[0].set_yscale('log')
fig1.savefig('task1_a_time.png', bbox_inches='tight')
plt.show()
```

A.1.2 high-pass filter

```
import numpy as np
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import skimage.io as io
from scipy.signal import convolve2d
from time import process_time
from pypher.pypher import psf2otf
import matplotlib.pyplot as plt
import time

from fspecial import fspecial_gaussian_2d

img =io.imread('birds_gray.png',as_gray=True).astype(float) /255

spatial_times =[]
fourier_times =[]

fig, axs =plt.subplots(nrows=3, ncols=3, figsize=(30, 30))
sigmas =[0.1, 1, 10]
for k in range(len(sigmas)):
    sigma =sigmas[k]
    filtSize =101
    lp =fspecial_gaussian_2d((filtSize, filtSize), sigma)

    ### Your code here ###
    spatial_start =time.time()
    spatial_result =img -convolve2d(img, lp, mode='same', boundary='wrap')
    spatial_times.append(time.time() -spatial_start)
    axs[0, k].imshow(np.clip(spatial_result, a_min=0, a_max=1), cmap='gray')
    axs[0, k].set_title(f'spatial filter (sigma={sigma})', fontsize=30)
    axs[0, k].axis('off')

    fourier_start =time.time()
    img_ft =fft2(img)
    img_ftshift =fftshift(img_ft)
    ft =psf2otf(lp, img.shape)
    ftshift =np.ones_like(img)-fftshift(ft)
    ft_mid =img_ftshift *ftshift
    ishift =ifftshift(ft_mid)
    ft_result =ifft2(ishift).real

    fourier_times.append(time.time() -fourier_start)
    axs[1, k].imshow(np.clip(ft_result, a_min=0, a_max=1), cmap='gray')
    axs[1, k].set_title(f'fourier filter (sigma={sigma})', fontsize=30)
    axs[1, k].axis('off')

    axs[2, k].imshow(fftshift(ft), cmap='gray')
    axs[2, k].set_title(f'OFT(sigma={sigma})', fontsize=30)
    axs[2, k].axis('off')
fig.savefig('task1_b_filter.png', bbox_inches='tight')
plt.show()

fig1, axs =plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
axs[0].bar(sigmas, spatial_times, width=0.4, label='Spatial Convolution')
for i, j in zip(sigmas, spatial_times):
    axs[0].text(i, j, round(j, 3), ha='center', va='bottom')
axs[0].set_title(f'spatial filter')
axs[0].set_xlabel('different sigmas')
axs[0].set_ylabel('Time (s)')

axs[1].bar(sigmas, fourier_times, width=0.4, label='Fourier Convolution')
for i, j in zip(sigmas, fourier_times):
    axs[1].text(i, j, round(j, 3), ha='center', va='bottom')
axs[1].set_title(f'fourier filter')
```

```
axs[1].set_xlabel('different sigmas')
axs[1].set_ylabel('Time (s)')
#axs[0].set_yscale('log')
fig1.savefig('task1_b_time.png', bbox_inches='tight')
plt.show()
```

A.2 Task2

A.2.1 division in the Fourier domain

```
import numpy as np
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import skimage.io as io
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
from skimage.metrics import peak_signal_noise_ratio as compute_psnr
from pypher.pypher import psf2otf
from skimage.metrics import peak_signal_noise_ratio as psnr
from pdb import set_trace

from fspecial import fspecial_gaussian_2d

img =io.imread('birds_gray.png', as_gray=True).astype(float) /255

# Task 2a - Inverse filtering

c =fspecial_gaussian_2d((35, 35), 5.)

cFT =psf2otf(c, img.shape)
cFT_shift =fftshift(cFT)

# Blur image with kernel
blur =convolve2d(img, c, mode='same', boundary='wrap')

fig, axs =plt.subplots(nrows=2, ncols=4, figsize=(40, 20))
sigmas =[0, 0.001, 0.01, 0.1]
for i in range(4):
    sigma =sigmas[i]
    # Add noise to blurred image
    unfilt =blur +sigma *np.random.randn(*blur.shape)
    axs[0, i].imshow(np.clip(unfilt, a_min=0, a_max=1), cmap='gray')
    axs[0, i].set_title(f'blurred image with sigma={sigma}')
    axs[0, i].axis('off')
    ### Inverse filter image here ###
    img_ft =fft2(unfilt)
    img_ftshift =fftshift(img_ft)
    ft_mid =img_ftshift *(1/ cFT_shift)
    ishift =ifftshift(ft_mid)
    ft_result =ifft2(ishift).real
    axs[1, i].imshow(np.clip(ft_result, a_min=0, a_max=1), cmap='gray')
    axs[1, i].set_title(f'processed image with sigma={sigma}')
    axs[1, i].axis('off')
    PSNR =psnr(img, ft_result)
    print('when the sigma of noise is ', sigma, ', the psnr is ', PSNR)
plt.savefig('task2_a.png', bbox_inches='tight')
plt.show()
```

A.2.2 Wiener de-convolution

```
import math

import numpy as np
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import skimage.io as io
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
```

```
from skimage.metrics import peak_signal_noise_ratio as compute_psnr
from pypher.pypher import psf2otf
from skimage.metrics import peak_signal_noise_ratio as psnr
from pdb import set_trace
from fspecial import fspecial_gaussian_2d

img =io.imread('birds_gray.png', as_gray=True).astype(float) /255

# Task 2b - Wiener filtering

c =fspecial_gaussian_2d((35, 35), 5.)
cFT =psf2otf(c, img.shape)
cFT_shift =fftshift(cFT)

# Blur image with kernel
blur =convolve2d(img, c, mode='same', boundary='wrap')

fig, axs =plt.subplots(nrows=2, ncols=4, figsize=(40, 20))
sigmas =[0, 0.001, 0.01, 0.1]
for i in range(4):
    sigma =sigmas[i]
    # Add noise to blurred image
    unfilt =blur +sigma *np.random.randn(*blur.shape)

    ### Your code here ###
    axs[0, i].imshow(np.clip(unfilt, a_min=0, a_max=1), cmap='gray')
    axs[0, i].set_title(f'blurred image with sigma={sigma}')
    axs[0, i].axis('off')
    if sigma ==0:
        SNR =math.inf
    else:
        SNR =np.mean(unfilt) /sigma

    img_ft =fft2(unfilt)
    img_ftshift =fftshift(img_ft)
    H =(1 /cFT_shift) *((abs(cFT_shift) **2) /(abs(cFT_shift) **2 +1 /SNR))
    ft_mid =img_ftshift *H
    ishift =ifftshift(ft_mid)
    ft_result =ifft2(ishift).real
    axs[1, i].imshow(np.clip(ft_result, a_min=0, a_max=1), cmap='gray')
    axs[1, i].set_title(f'processed image with sigma={sigma}')
    axs[1, i].axis('off')
    PSNR =psnr(img, ft_result)
    print('when the sigma of noise is ', sigma, ', the psnr is ', PSNR)
plt.savefig('task2_b.png', bbox_inches='tight')
plt.show()
```

A.3 Task3

```
import numpy as np
from tqdm import tqdm
from time import time
import matplotlib.pyplot as plt


def grad_l2(A, x, b):
    # TODO: return the gradient of 0.5 * ||Ax - b||_2^2
    return A.T @ A @ x -A.T @ b


def residual_l2(A, x, b):
    return 0.5 *np.linalg.norm(A @ x -b) **2


def run_gd(A, b, mode, step_size=1e-4, num_iters=1500, grad_fn=grad_l2, residual=residual_l2):
    ''' Run gradient descent to solve Ax = b

    Parameters
```

```
    ----------
    A : matrix of size (N_measurements, N_dim)
    b : observations of (N_measurements, 1)
    step_size : gradient descent step size
    num_iters : number of iterations of gradient descent
    grad_fn : function to compute the gradient
    residual : function to compute the residual

    Returns
    -------
    x
        output matrix of size (N_dim)

    residual
        list of calculated residuals at each iteration

    timing
        time to execute each iteration (should be cumulative to each iteration)

    '''

    # initialize x to zero
    x = np.zeros((A.shape[1], 1))

    # TODO: complete the gradient descent algorithm here
    # you can also complete and use the grad_l2 and residual_l2 functions

    # this function can return a list of residuals and timings at each iteration so
    # you can plot them and include them in your report

    # don't forget to also implement the stochastic gradient descent version of this function!

    ### gradiant decent
    if mode == 'GD':
        i = 0
        residuals = [residual(A, x, b)]
        times = []
        time_start = time()
        while i < num_iters:
            time1 = time()
            x -= step_size * grad_fn(A, x, b)
            residuals.append(residual(A, x, b))
            times.append((time() - time_start))
            i += 1
        return x, residuals, times
    elif mode == 'SGD':
        i = 0
        residuals = [residual(A, x, b)]
        times = []
        time_start = time()
        while i < num_iters:

            random_integers = np.random.randint(0, N_measurements, size=B)
            A1 = A[random_integers, :]
            b1 = b[random_integers]

            x -= step_size * grad_fn(A1, x, b1)

            residuals.append(residual(A, x, b))
            times.append((time() - time_start))
            i += 1
        return x, residuals, times


def run_lsq(A, b):
    ''' Numpy's implementation of least squares which uses SVD & matrix factorization from LAPACK '''
    x, resid, _, _ = np.linalg.lstsq(A, b, rcond=None)
    return x, 0.5 * resid


if __name__ == '__main__':
```

```
    # set up problem
    N_dim =128 # size of x
    N_measurements =16384 # number of measurements or rows of A

    # load matrix
    dat =np.load('task3.npy', allow_pickle=True)[()]

    # data matrix -- here the rows are measurements and columns are dimension N_dim
    A =dat['A']

    # corrupted measurements
    b =dat['b']

    # least squares solve using SVD + matrix factorization
    # you can compare your solution to this
    x_lsq, resid_lsq =run_lsq(A, b)
    iteration =1500

    fig, ax =plt.subplots(nrows=1, ncols=2, figsize=(15, 5))

    # TODO: implement and call your GD and SGD functions
    result1 =run_gd(A, b, 'GD', step_size=1e-4, num_iters=iteration, grad_fn=grad_l2, residual=residual_l2)
    ax[0].plot(range(iteration +1), result1[1])
    ax[1].plot(result1[2], result1[1][1:])
    for B in [10, 100, 1000]:
        result2 =run_gd(A, b, 'SGD', step_size=1e-4, num_iters=iteration, grad_fn=grad_l2, residual=residual_l2)
        ax[0].plot(range(iteration +1), result2[1])
        ax[1].plot(result2[2], result2[1][1:])
    ax[0].set_yscale('log')
    ax[1].set_xscale('log')
    ax[1].set_yscale('log')
    ax[0].plot(range(iteration +1), [resid_lsq] *(iteration +1))
    ax[0].legend(['GD', 'SGD B=10', 'SGD B=100', 'SGD B=1000', 'SVD'])
    ax[1].legend(['GD', 'SGD B=10', 'SGD B=100', 'SGD B=1000'])
    ax[0].set_xlabel('iterations')
    ax[0].set_ylabel('residual')
    ax[1].set_xlabel('time(s)')
    ax[1].set_ylabel('residual')
    plt.savefig('task3.png', bbox_inches='tight')
    plt.show()
```

A.4 bonus

```
import numpy as np
from tqdm import tqdm
from time import time
import matplotlib.pyplot as plt


def grad_l1(A, x, b):
    # TODO: return the gradient of 0.5 * ||Ax - b||_1
    K =A @ x -b
    return 0.5*A.T @ np.sign(K)


def residual_l1(A, x, b):
    return 0.5 *np.linalg.norm(A @ x -b, ord=1)


def run_gd(A, b, mode, step_size=1e-4, num_iters=1500, grad_fn=grad_l1, residual=residual_l1):
    ''' Run gradient descent to solve Ax = b

    Parameters
    ----------
    A : matrix of size (N_measurements, N_dim)
    b : observations of (N_measurements, 1)
    step_size : gradient descent step size
    num_iters : number of iterations of gradient descent
```

```
    grad_fn : function to compute the gradient
    residual : function to compute the residual

    Returns
    -------
    x
        output matrix of size (N_dim)

    residual
        list of calculated residuals at each iteration

    timing
        time to execute each iteration (should be cumulative to each iteration)

    '''

    # initialize x to zero
    x = np.zeros((A.shape[1], 1))

    # TODO: complete the gradient descent algorithm here
    # you can also complete and use the grad_l2 and residual_l2 functions

    # this function can return a list of residuals and timings at each iteration so
    # you can plot them and include them in your report

    # don't forget to also implement the stochastic gradient descent version of this function!

    ### gradiant decent
    if mode == 'GD':
        i = 0
        residuals = [residual(A, x, b)]
        times = []
        time_start = time()
        while i < num_iters:
            x -= step_size * grad_fn(A, x, b)
            residuals.append(residual(A, x, b))
            times.append((time() - time_start))
            i += 1
        return x, residuals, times
    elif mode == 'SGD':
        i = 0
        residuals = [residual(A, x, b)]
        times = []
        time_start = time()
        while i < num_iters:
            random_integers = np.random.randint(0, N_measurements, size=B)
            A1 = A[random_integers, :]
            b1 = b[random_integers]
            x -= step_size * grad_fn(A1, x, b1)
            residuals.append(residual(A, x, b))
            times.append((time() - time_start))
            i += 1
        return x, residuals, times


def run_lsq(A, b):
    ''' Numpy's implementation of least squares which uses SVD & matrix factorization from LAPACK '''
    x, resid, _, _ = np.linalg.lstsq(A, b, rcond=None)
    return x, 0.5 * resid


if __name__ == '__main__':
    # set up problem
    N_dim = 128 # size of x
    N_measurements = 16384 # number of measurements or rows of A

    # load matrix
    dat = np.load('task3.npy', allow_pickle=True)[()]

    # data matrix -- here the rows are measurements and columns are dimension N_dim
    A = dat['A']
```

```
# corrupted measurements
b =dat['b']
x =np.zeros((A.shape[1], 1))
grad_l1(A, x, b)
# least squares solve using SVD + matrix factorization
# you can compare your solution to this
x_lsq, resid_lsq =run_lsq(A, b)
iteration =1500

fig, ax =plt.subplots(nrows=1, ncols=2, figsize=(15, 5))
#
# # TODO: implement and call your GD and SGD functions
result1 =run_gd(A, b, 'GD', step_size=1e-4, num_iters=iteration, grad_fn=grad_l1, residual=residual_l1)
print(result1[2][:20])
ax[0].plot(range(iteration +1), result1[1])
ax[1].plot(result1[2], result1[1][1:])
for B in [10, 100, 1000]:
    result2 =run_gd(A, b, 'SGD', step_size=1e-4, num_iters=iteration, grad_fn=grad_l1, residual=residual_l1)
    print(result2[2][:20])
    ax[0].plot(range(iteration +1), result2[1])
    ax[1].plot(result2[2], result2[1][1:])
ax[0].set_yscale('log')
ax[1].set_xscale('log')
ax[1].set_yscale('log')
ax[0].plot(range(iteration +1), [resid_lsq] *(iteration +1))
ax[0].legend(['GD', 'SGD B=10', 'SGD B=100', 'SGD B=1000', 'SVD'])
ax[1].legend(['GD', 'SGD B=10', 'SGD B=100', 'SGD B=1000'])
ax[0].set_xlabel('iterations')
ax[0].set_ylabel('residual')
ax[1].set_xlabel('time(s)')
ax[1].set_ylabel('residual')
plt.savefig('bonus.png', bbox_inches='tight')
plt.show()
```