COMP0114 Inverse Problems in Imaging Coursework 1

Student ID 18145399

1. Solving Underdetermined Problems

a) Define a function `phi(x2,p)` of two variables, x_2 and p, to compute the value of Φ as given below.

$$\Phi = \sum_i |x_i|^p$$

i.e. $\Phi = |x_1|^p + |x_2|^p$ where x_i satisfies $x_1 + 2x_2 = 5$. The reason I used x_2 to be an input instead of (x_1, x_2) is that optimization function I used in b) need parameter to be 1D-array.

```
def phi(x2,p):
    """
    Inputs:
        x2 is one element of a vector of length 2 (x1,x2)
        p is a scalar
    Returns:
        phi
    """
    # x1 = 5-2*x2
    Sum = (np.abs(5-2*x2))**p + (np.abs(x2))**p

    return Sum
```

b) Use library function `scipy.optimize.minimize(phi,x_start,args=p_i)` to solve the constrained optimization problem

$$\text{minimize} \quad \Phi = \sum_i |x_i|^p$$

for p in range of $(1, 1.5, 2, 2.5, 3, 3.5, 4)$.

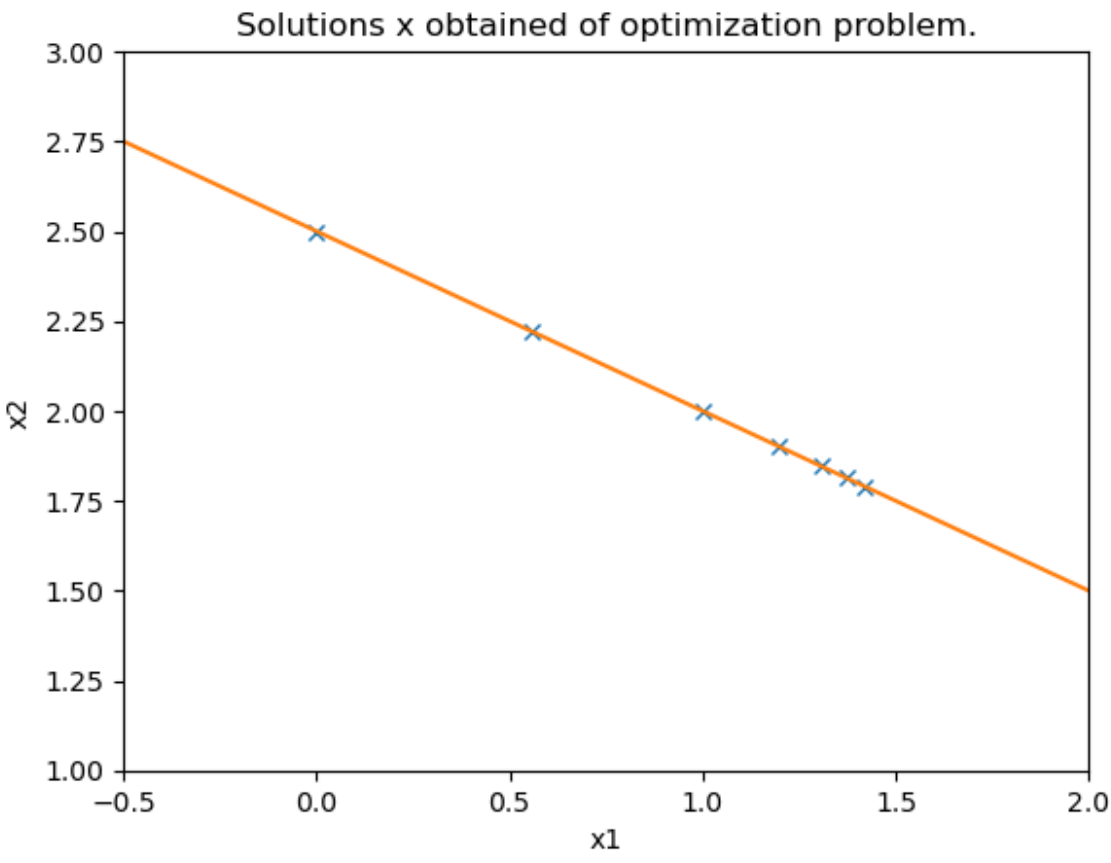
The input phi is the function need to be optimised, x_start is a start point of x_2 to find the solution, args is the constant parameter in the optimising function and the result of this function has the structure of:

```
fun: 2.500000011175871
hess_inv: array([[1.73444551]])
     jac: array([0.])
message: 'Optimization terminated successfully.'
    nfev: 104
    nit: 3
    njev: 52
status: 0
success: True
       x: array([2.49999999])
```

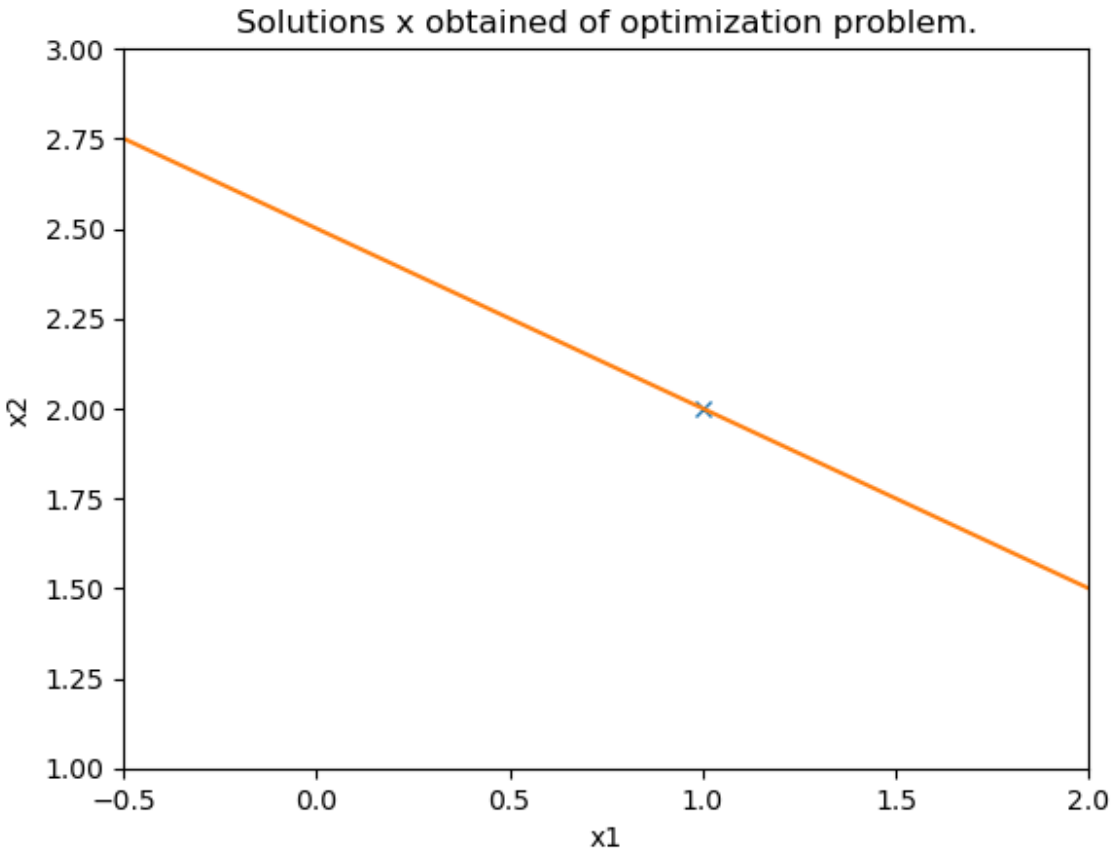
To plot the solution, we only need the `x = reuslt.x` and `y = result.fun` when `result.success == True`. And I got the solution point:

```
when p = 1.5:
x=([0.5555566042412448, 2.2222216978793776]), y =
3.726779962500272
when p = 2.0:
x=([0.9999995008103717, 2.000000249594814]), y = 5.000000000000312
when p = 2.5:
x=([1.1976615735037677, 1.9011692132481162]), y =
6.553467806802202
when p = 3.0:
x=([1.3060193930427793, 1.8469903034786104]), y =
8.528433037009236
when p = 3.5:
x=([1.3739978884128101, 1.813001055793595]), y =
11.064585668127403
when p = 4.0:
x=([1.4205182909810472, 1.7897408545094764]), y =
14.33212122809243
```

C) The plot of solution points on the constraint line $x_1 + 2x_2 = 5$ is shown below.



d) Another way to find the solution is using the Moore-Penrose generalised inverse $A^{\dagger} := A^{\intercal}(AA^{\intercal})^{-1}$ and applying it to get solution $x_{MP} = A^{\dagger}b$. As shown below, the solution I got by using this method is same as the solution obtaining with previous method when $p = 2$. The reason of similarity is that Moore–Penrose pseudoinverse is commonly used to compute a least squares solution to a system of linear equations that lacks a unique solution, which is asked the degree of equation to be 2.



2.Singular Value Decomposition

a) Firstly, define a function `grid(n)` to set up a equally spaced grid on the interval [−1,1] with library function `np.linspace(-1,1,n)` where n is the number of steps.

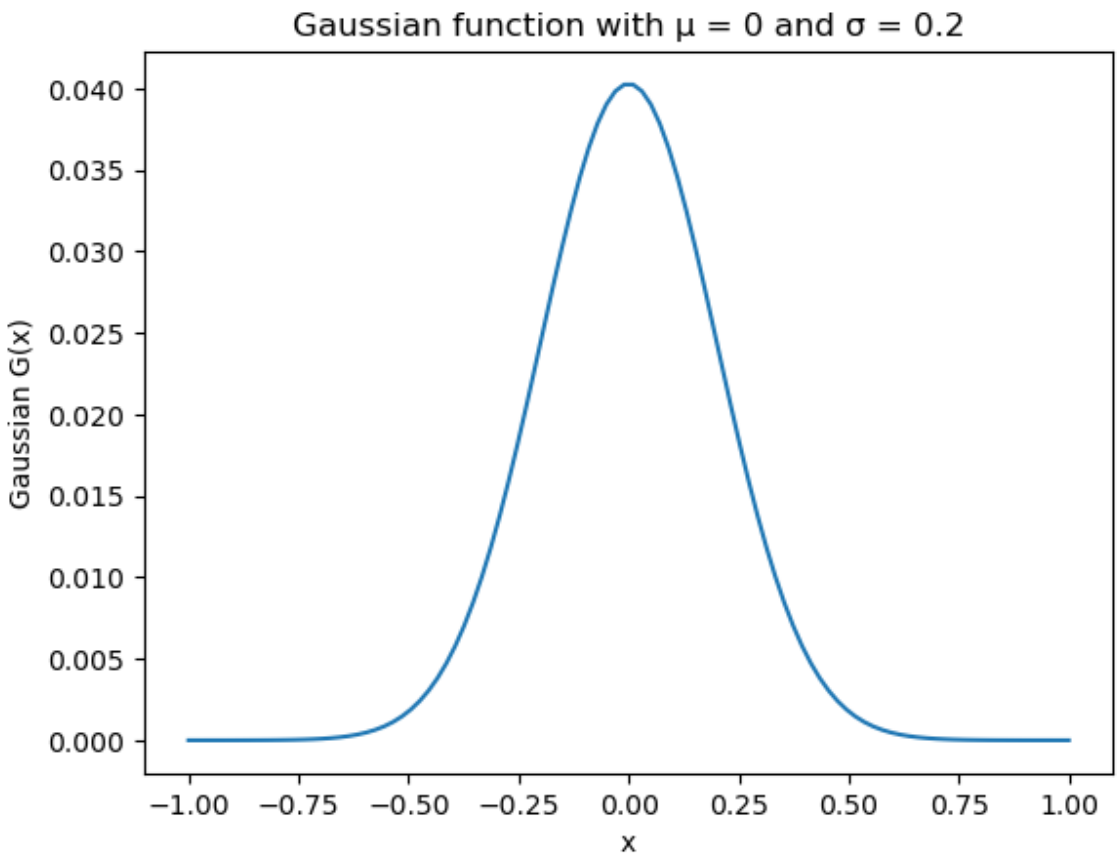
```
def grid(n):
    g = np.linspace(-1,1,n)
    return g
```

b) Then define a Gaussian function centred at $\mu = 0$ with $\sigma = 0.2$ by equation

$$G(x) = \frac{\delta n}{\sqrt{2\pi}\sigma} exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

```
def Gaussian(X,sigma,miu):
    G = []
    for x in X:
        dn = 2/(len(X)-1)
        G.append((dn/(np.sqrt(2*np.pi)*sigma)) * np.exp(-((x-
miu)**2)/(2*sigma**2)))
    return G
```

and evaluate it along the grid set in part a). The result of evaluation is shown below as a plot.



c) Then define a convolution matrix A based on Gaussian of size $n \times n$ with entires

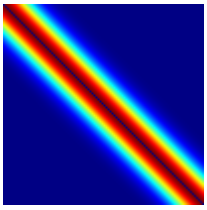
$$A_{i,j} = G(x_i - x_j) = \frac{\delta n}{\sqrt{2\pi}\sigma} exp(-\frac{(x_i - x_j)^2}{2\sigma^2})$$

```
def Convolution_Matrix(n,sigma):
    M = np.zeros((n,n))
    # step size dn
    dn = 2/(n-1)
    X = grid(n)
    for i in range(n):
        for j in range(n):
            M[i,j] = ((dn/(np.sqrt(2*np.pi)*sigma)) * np.exp(-((X[i]-
X[j])**2)/(2*sigma**2)))
    return M
```

d) The convolution matrix with size $n = 100$ can be visualised by rescaling the matrix and explicitly defining a colormap with the codes

```
import cv2
import numpy as np
A = Convolution_Matrix(100,sigma)
# rescaling matrix A
Atmp = np.array(np.ceil(A/np.max(A)*256), dtype = np.uint8)
# plot rescaled matrix A
Aimg = cv2.applyColorMap(Atmp, cv2.COLORMAP_JET)
cv2.imwrite("2d.png",Aimg)
```

This graph satisfies my expectation as matrix A is diagonally symmetric resulted from its calculation equation of entries.



e) Now, let's compute SVD of convolution matrix by library function `U,W,V_T = np.linalg.svd(A)` in which `V_T` is the transpose of matrix V and W is the entries'list of diagonal matrix containing the singular values with size n. And the results `U`, `W` and `V_T` is satisfied the equation $A = UWV^T$ within a tolerance: $2.434181540759949 \times 10^{-15}$, where the tolerance is the norm of two product calculated by `np.linalg.norm(A - (U * W) @ V_T)`.

f) Now, compute the pseudoinverse $A^\dagger$ of A.

i) Firstly, constructe $W^\dagger$ into a sparse matrix by using two scipy functions

```
Wdiag = scipy.sparse.spdiags(W,0,A.shape[0],A.shape[1])
W_inv = scipy.sparse.linalg.inv(Wdiag)
```

ii) Then For the case $n = 10$, check if $WW^\dagger = W^\dagger W = n \times n$ Identity matrix by

```
# set a n*n identity matrix
identity_n = np.identity(N)
# check if WW^† = W^†W
if (Wdiag.todense() * W_inv).all() == (W_inv.todense() * Wdiag).all():
    print('Two products about W are the same.')
else:
    print('Two products about W are not the same.')
#check if WW^† = n*n identity matrix
if ((Wdiag.todense() * W_inv).all() == (identity_n).all()):
    print('Two products about W are identity.')
```

with the outputs

```
Two products about W are the same.
Two products about W are identity.
```

iii) Then, compute the pseudoinverse $A^\dagger$ of A by using both the library function `np.linalg.pinv(A)` and the formula $A^\dagger = VW^\dagger U^T$.

```
# use library function
A_inv = np.linalg.pinv(A)
# use formula
A_pinv = V_T.T @ W_inv @ U.T
```

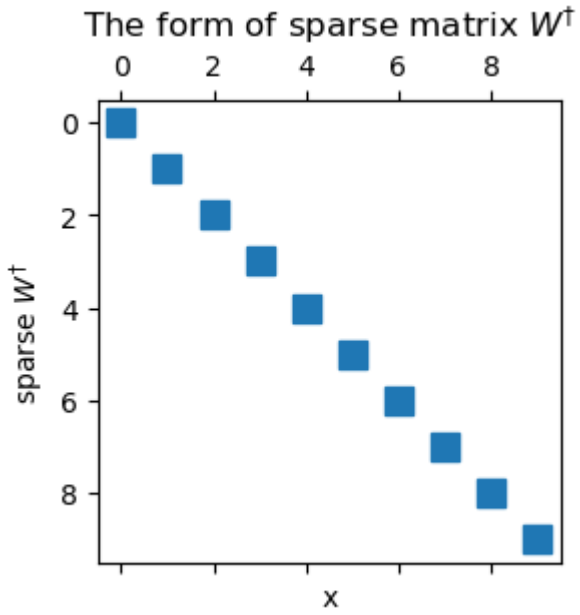
vi) Check also that $AA^\dagger = A^\dagger A = Id_n$ for $n = 10$ by codes:

```
# norm of difference between A†A and AA†
norm = (np.linalg.norm(A * A_inv - A_inv * A))
# norm of difference between pseudoinverse A† calculated by twio methods
norm_Apinv = np.linalg.norm(A_inv - A_pinv)
# check two A†
```

```
if np.allclose(A_inv, A_pinv):
    print('Two A† is element-wise equal within a tolerance:',norm_Apinv)
else:
    print('Two A† is not equal with a norm:',norm_Apinv)
# check two products
if np.allclose(A * A_inv, A_inv * A):
    print('Two products about A are element-wise equal within a
tolerance:',norm)
else:
    print('Two products about A are not equal with a norm:',norm)
# check if products are identity
product = (A * A_inv)
product[product < 2*1e-15] = 0
if (product.all() == identity_n.all()):
    print('Two products about A are identity.')
```

with the outputs

```
Two A† is element-wise equal within a tolerance:
2.8400347913994436e-15
Two products about A are element-wise equal within a tolerance:
0.0
Two products about A are identity.
```



Brief discussion

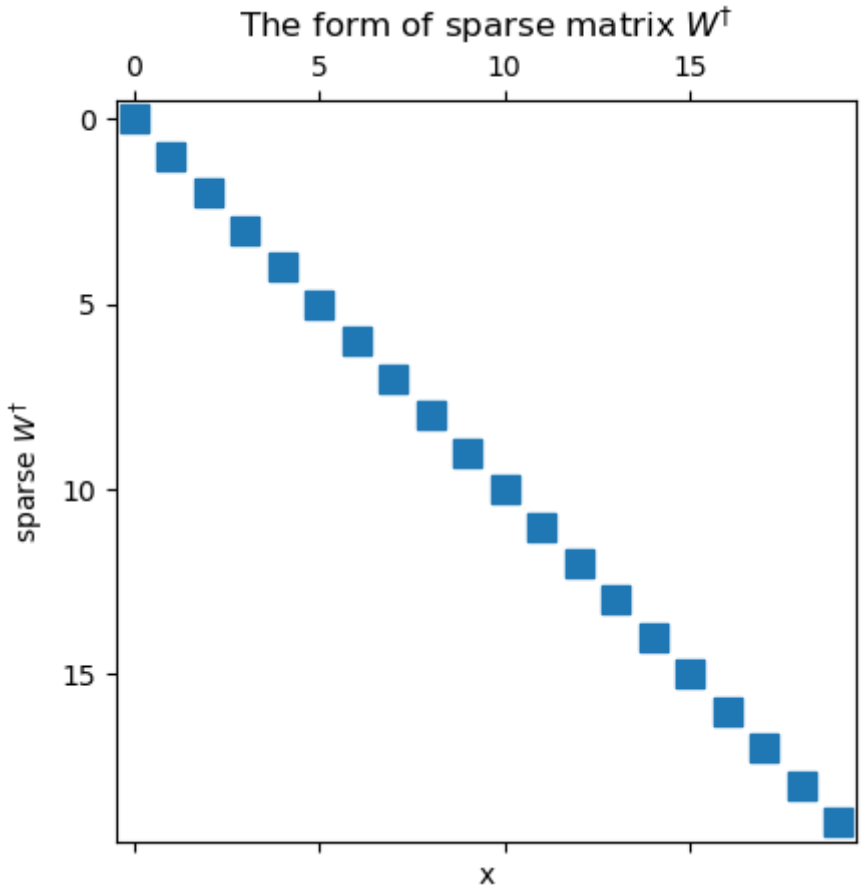
The overall result is:

```
When the size of marix is 10 :
Two products about W are the same.
Two products about W are identity.
Two A† is element-wise equal within a tolerance:
2.8400347913994436e-15
Two products about A are element-wise equal within a tolerance:
0.0
Two products about A are identity.
```

Among the five equivalence eqaution for $n = 10$, only `np.linalg.pinv(A) == V_T.T @ W_inv @ U.T` exists inequivalence which might be resulted from the computing memeory shortage as the size of matrix product.

g) Repeat e) and f) for n = 20, obtain:

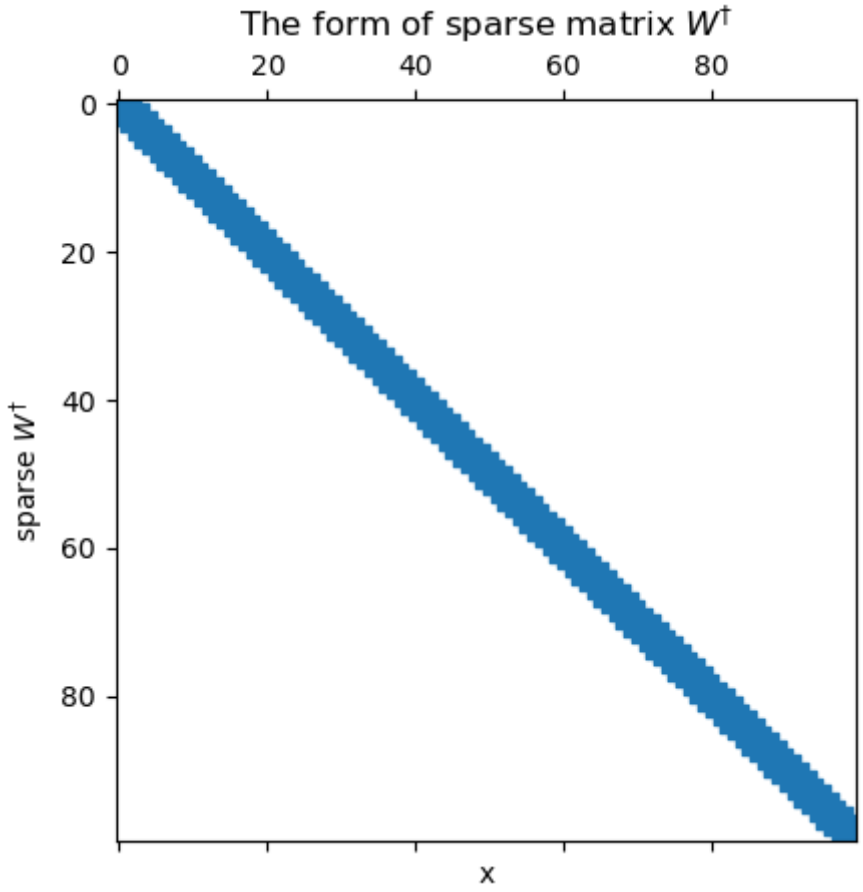
```
When the size of marix is 20 :
Two products about W are the same.
Two products about W are identity.
Two A† is element-wise equal within a tolerance:
3.7416236697031865e-10
Two products about A are element-wise equal within a tolerance:
0.0
Two products about A are identity.
```



Then, choose $n = 100$.

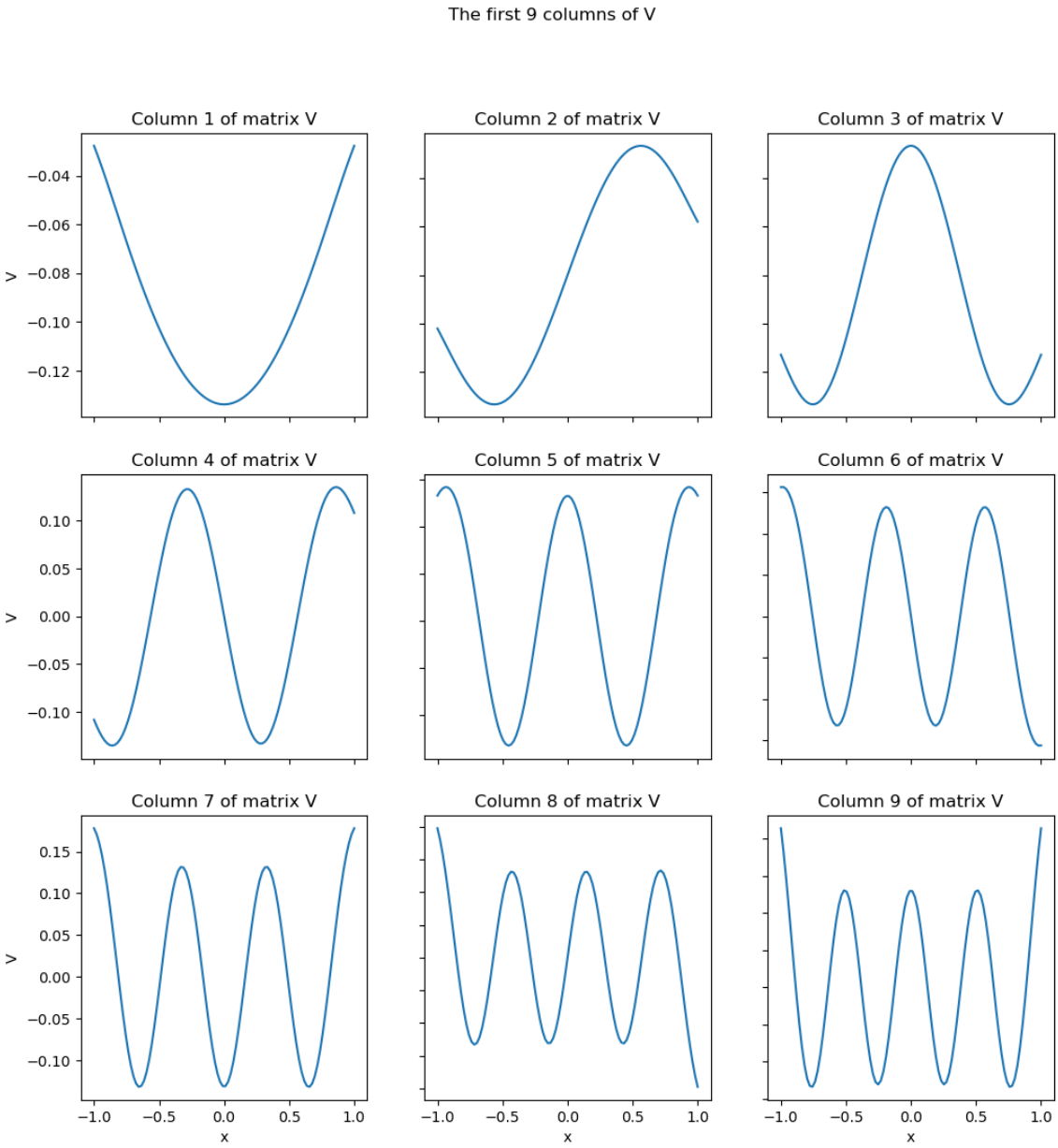
In this case, there is a difference against previous two:

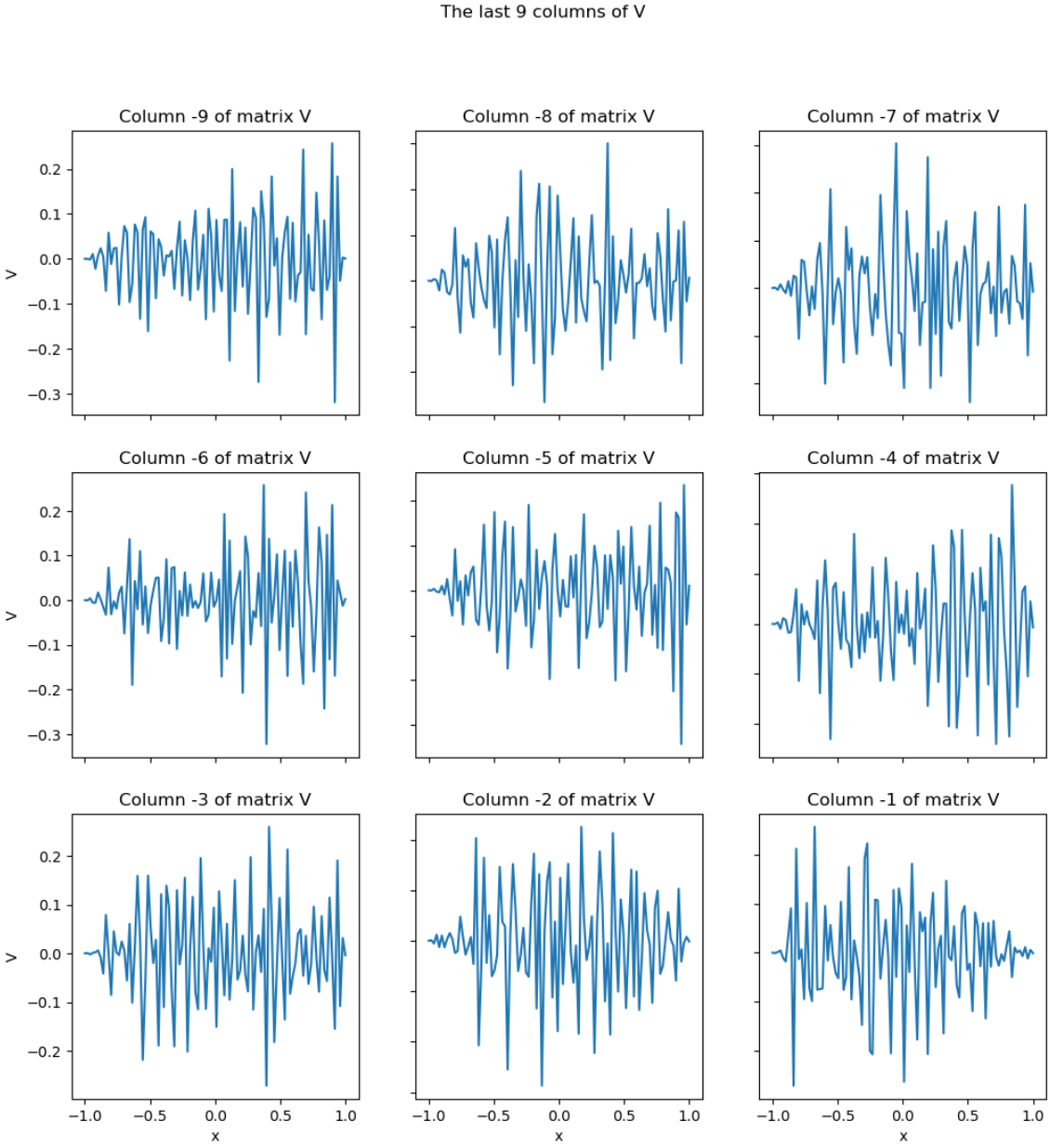
```
When the size of marix is 100 :
Two products about W are the same.
Two products about W are identity.
Two A† is not equal with a norm: 1.4308362856453842e+17
Two products about A are element-wise equal within a tolerance:
0.0
Two products about A are identity.
```

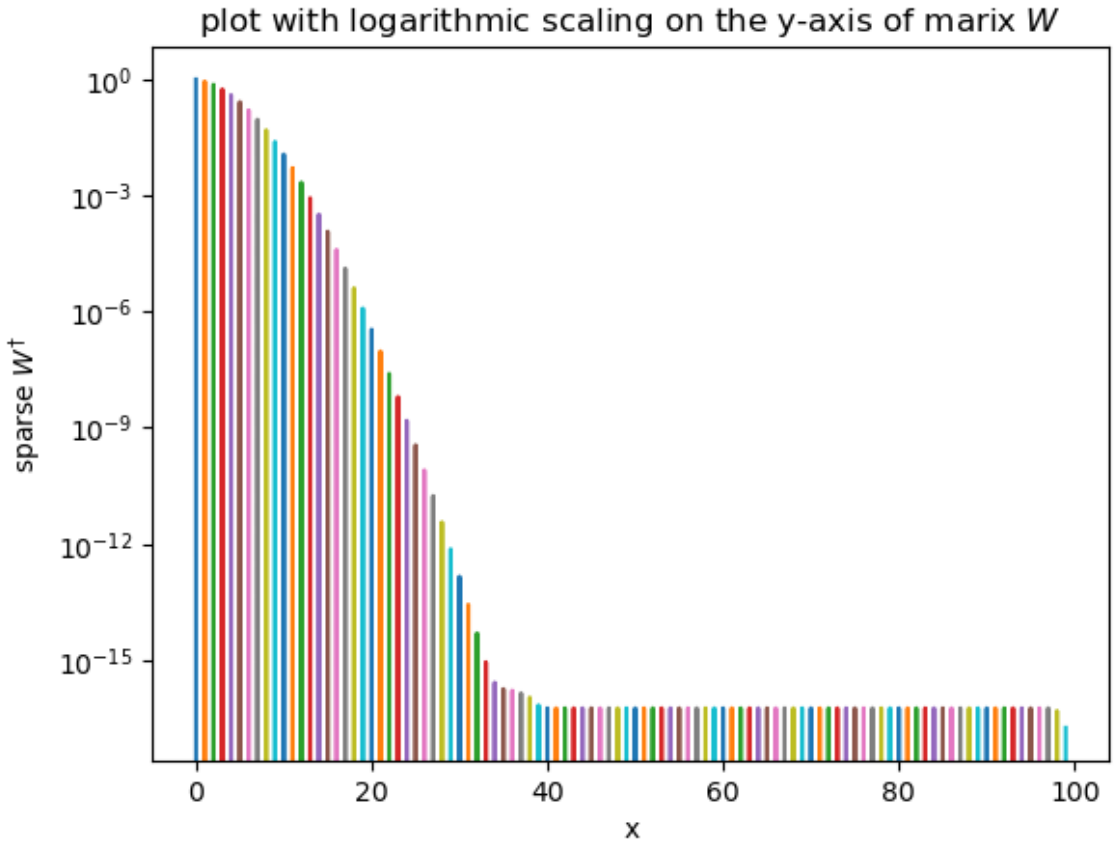


The difference between two A† is extermely large. Also, when n > 30, two A† is hardly equal with norms larger than 0.031135550123037235.

Now, plot the first 9 columns of V, the last 9 columns of V, and the singular diagonal W on a log-scale, i.e. log(diag(W))


The first 9 columns of V
Column 1 of matrix V
Column 2 of matrix V
Column 3 of matrix V
Column 4 of matrix V
Column 5 of matrix V
Column 6 of matrix V
Column 7 of matrix V
Column 8 of matrix V
Column 9 of matrix V
V
x
−0.04
−0.06
−0.08
−0.10
−0.12
0.10
0.05
0.00
−0.05
−0.10
0.15
−1.0
−0.5
0.0
0.5
1.0


The last 9 columns of V
Column -9 of matrix V
Column -8 of matrix V
Column -7 of matrix V
Column -6 of matrix V
Column -5 of matrix V
Column -4 of matrix V
Column -3 of matrix V
Column -2 of matrix V
Column -1 of matrix V
V
x
0.2
0.1
0.0
−0.1
−0.2
−0.3
−1.0
−0.5
0.0
0.5
1.0



Brief discussion

The 5 equivalence equations still performance similarly as when $n = 10$, but there is a larger difference betwen two $A^{\dagger}$, with a value that is still in the acceptable error range. However when $n = 100$, the difference betwen two $A^{\dagger}$ far exceeds the tolerance threshold with an extremly high value of $1.4308362856453842 \times 10^{17}$

3. Convolutions and Fourier transform

a) Create a step function $f(x)$ defined by

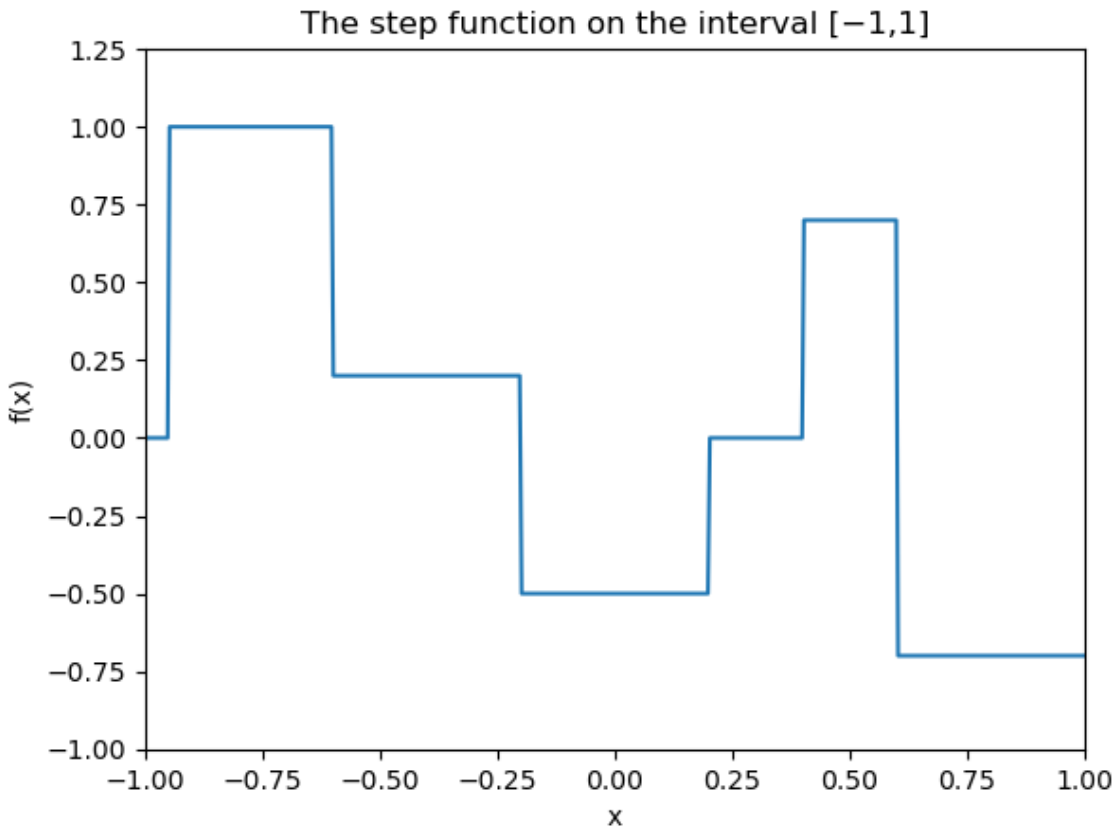
$$f(x) = \chi_{(-0.95,-0.6]}(x) + 0.2\chi_{(-0.6,-0.2]}(x) - 0.5\chi_{(-0.2,0.2]}(x) + 0.7\chi_{(0.4,0.6]}(x) - 0.7\chi_{(0.6,1]}($$

where $\chi(x)$ of an interval (a,b] is defined by

$$\chi(x) = \begin{cases} 1 & \text{for } a < x \leq b \\ 0 & \text{otherwise} \end{cases}.$$

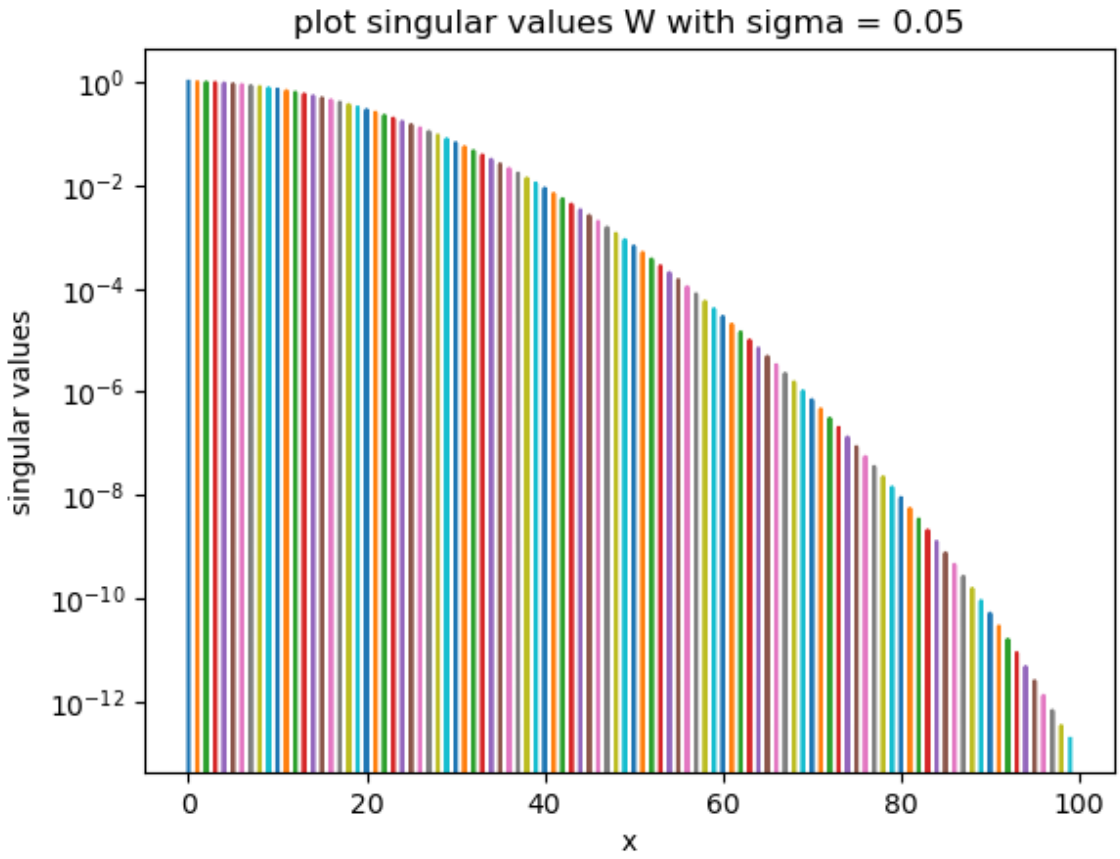
```
def f(x):
    chi1 = lambda x : 1 if -0.95 < x <= -0.6 else 0;
    chi2 = lambda x : 1 if -0.6 < x <= -0.2 else 0;
    chi3 = lambda x : 1 if -0.2 < x <= 0.2 else 0;
    chi4 = lambda x : 1 if 0.4 < x <= 0.6 else 0;
    chi5 = lambda x : 1 if 0.6 < x <= 1 else 0;
    result = chi1(x) + 0.2*chi2(x) - 0.5*chi3(x) + 0.7*chi4(x) -
0.7*chi5(x)
    return result
```

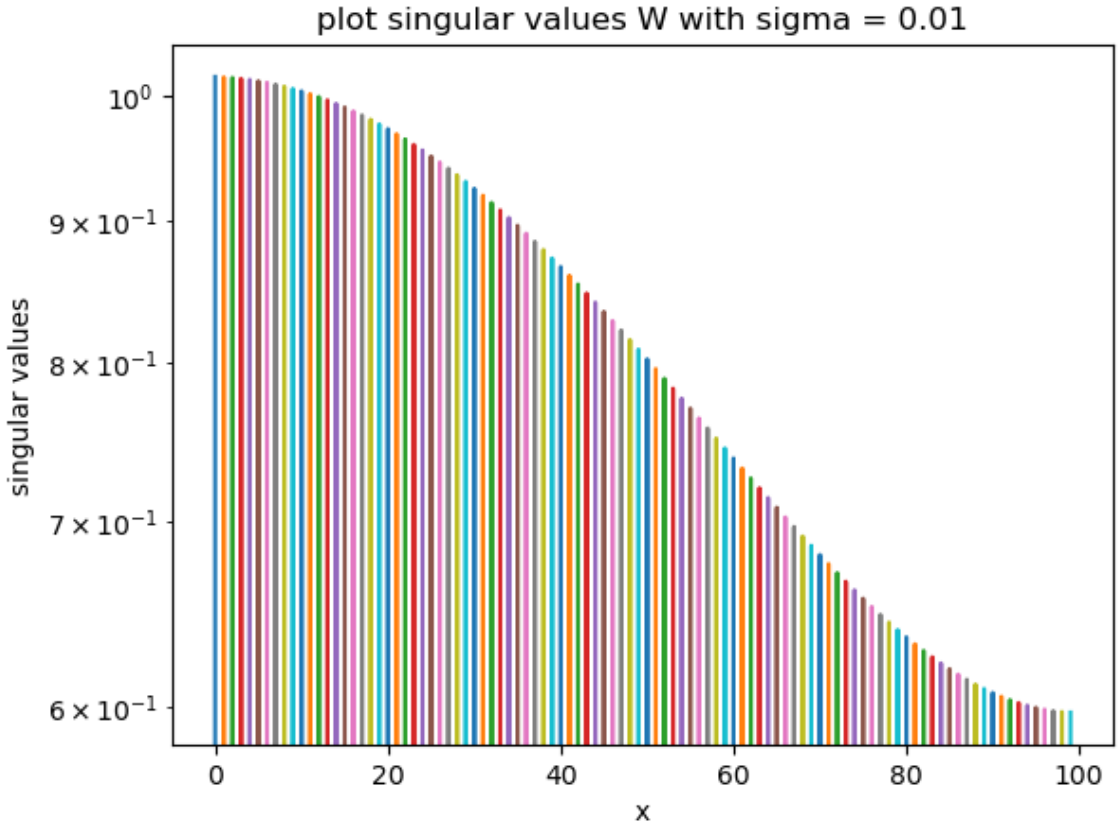
And plot it on a grid on the interval [−1,1] with a step number $n = 500$n = 200 $with step size$dn = 0.01$ is enough for resolve the jump).

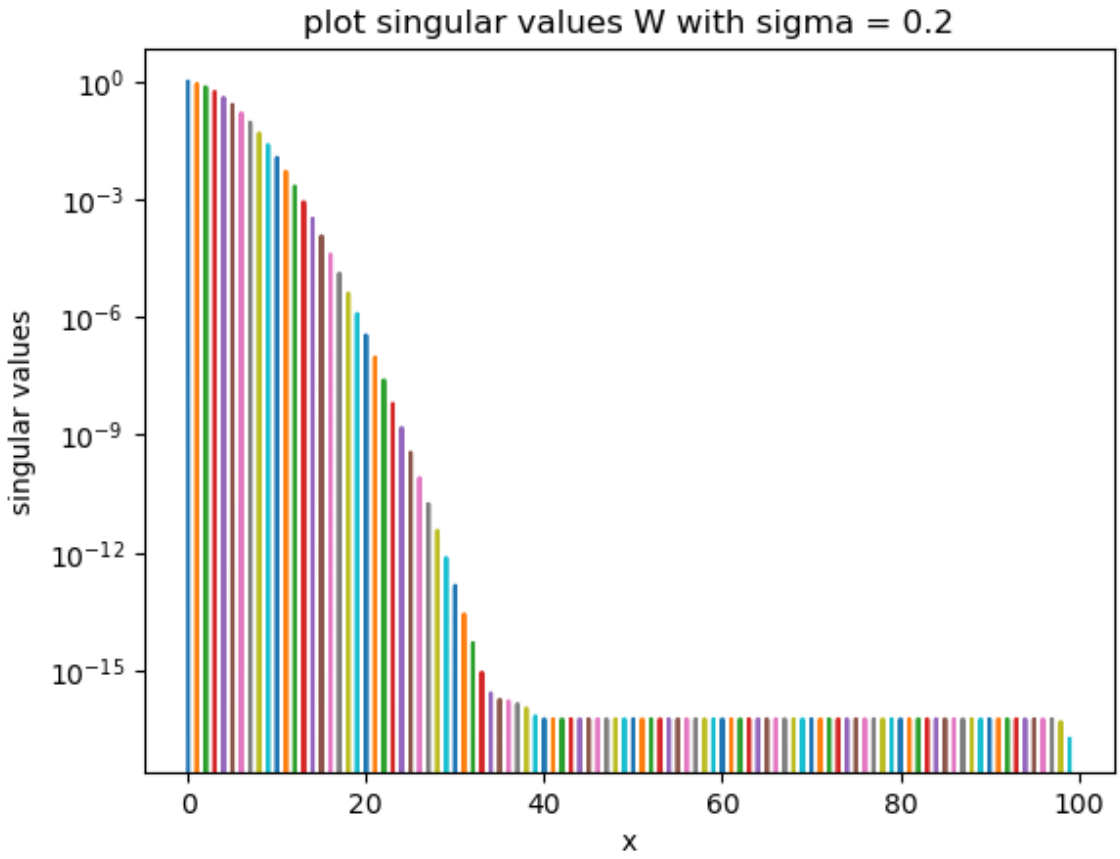


b) Then obtain matrix A with function `Convolution_Matrix(n,sigma)` for σ = 0.05, 0.1, 0.2 and plot the singular values W from

```
A = Convolution_Matrix(n,sigma)
U,W,V_T = np.linalg.svd(A)
```


plot singular values W with sigma = 0.05
singular values
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
0
20
40
60
80
100
x


plot singular values W with sigma = 0.01
singular values
10^{0}
9×10^{-1}
8×10^{-1}
7×10^{-1}
6×10^{-1}
0
20
40
60
80
100
x



c) Shown from above graphs, the singular values is following a Gaussian function. To determine the variance of this Gaussian, use `popt, pcov = scipy.optimize.curve_fit(half_Gaussian,x,W)` where

- `popt` is the best-fit optimising parameter,
- `half_Gaussian` is the function with input of `(x>0, sigma)` and returns the Gaussian value on the x>0 interval.
- `x` is the grid of singular values W.
- and `W` is the singular values shown on the graph which I want to fit the Gasussian in.

Firstly define the function by

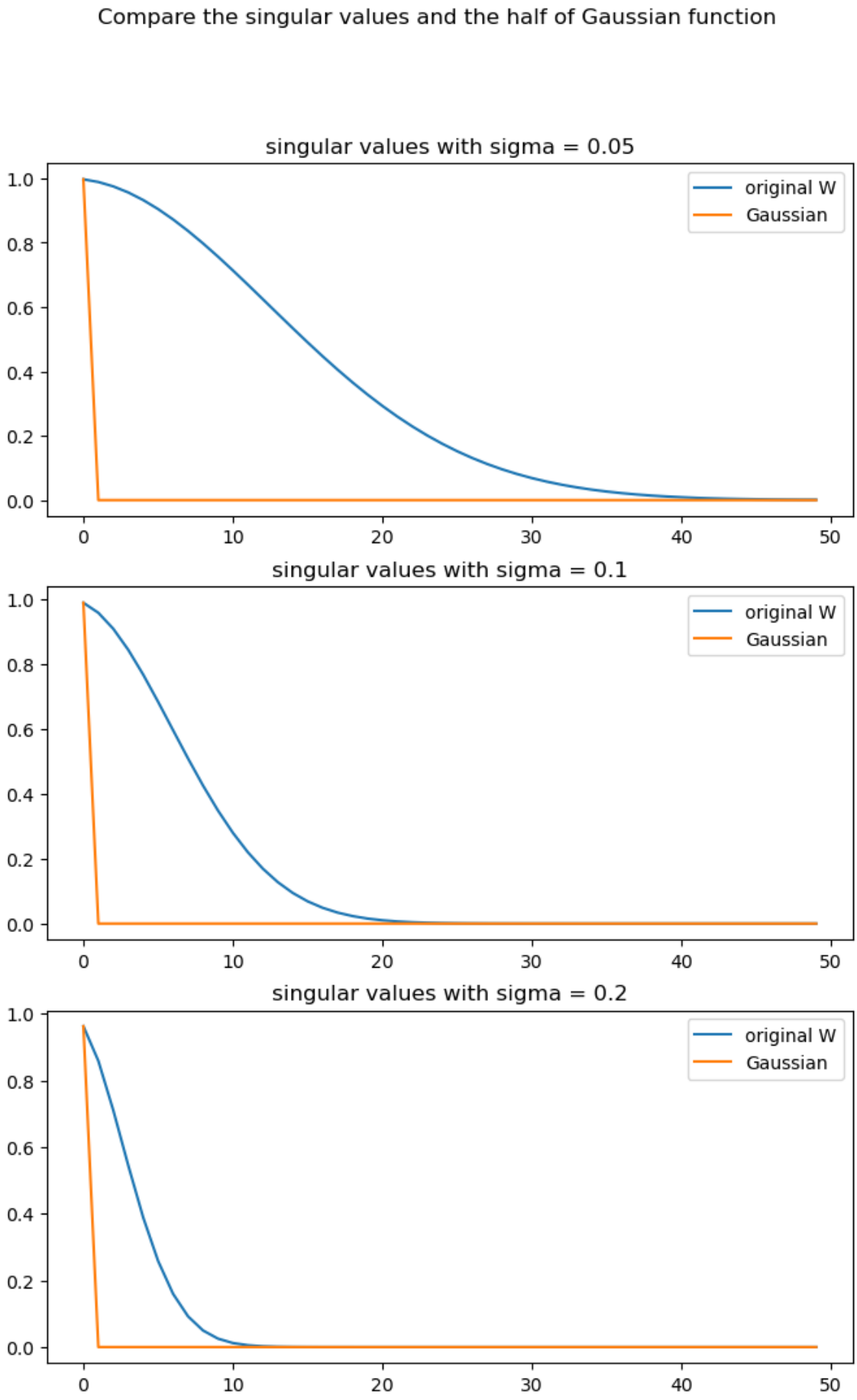
```
def half_Gaussian(x,sigma):
    x_all = np.append((-x[::-1]),x[1:])
    G = Gaussian(x_all,sigma,0)
    x_size = int((len(x_all)+1)/2)
    return G[-x_size:]
```

Then apply the `scipy.optimize.curve_fit` to find the best fit Gaussians, and get the varaince of them as:

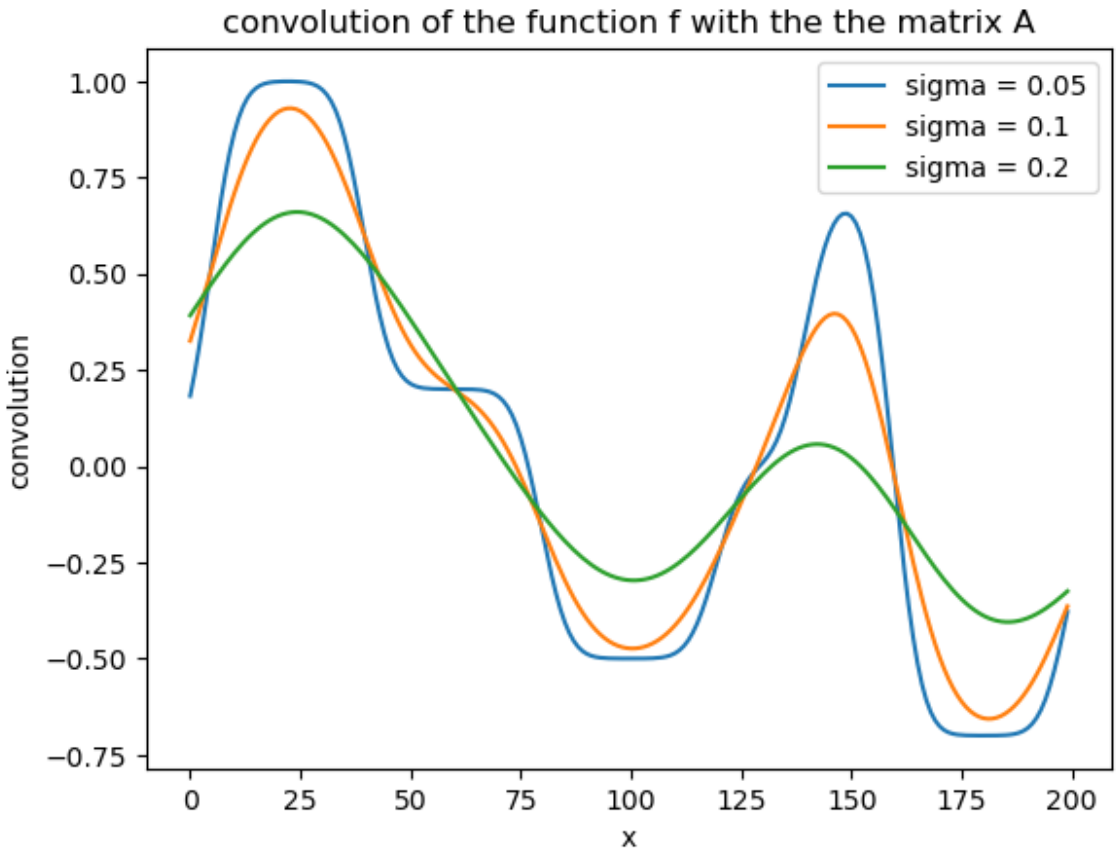
```
When sigma = 0.05, the variance of Gaussian is [6.66570836e-05].
When sigma = 0.1, the variance of Gaussian is [6.77037328e-05].
When sigma = 0.2, the variance of Gaussian is [7.15376509e-05].
```

Which is not as expected (there might be some wrong when using the `curve_fit` .)

And the graphs of the singular values and Gaussian values are shown below:



d) Using matrix multiplication obtain the convolution of the step functin f(x) for three choices of σ and plot them as below.


convolution of the function f with the the matrix A
sigma = 0.05
sigma = 0.1
sigma = 0.2
convolution
x
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
25
50
75
100
125
150
175
200

e)