Assignment 1

Callum Lau, 19102521

Inverse Problems

February 3, 2020

1 Solving Underdetermined Problems

1.1 (a)

The function Φ can be computed succinctly as `phi = lambda x, p: np.sum(abs(x)**p)`

1.2 (b)

We use the library function `scipy.optimize.minimize` to compute the solutions of the optimisation problem for $p = 1, 1.5, 2, 2.5, 3, 3.5, 4$. In order to solve this minimisation problem we must first specify the constraint `cons`: $x_1 + 2x_2 = 5$, as well as an initial starting point `x0`. For a given `p` value we then call:

```
sol = minimize(phi, x0, args=(p), constraints=cons)
```

to get the minimum norm solution.

1.3 (c)

A plot of the solutions is given below:

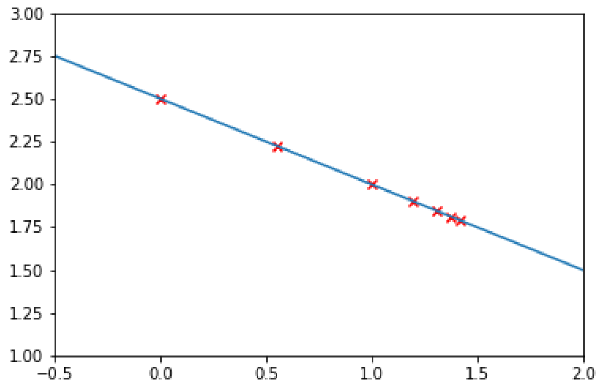


Figure 1: Minimum norm solutions

1.4 (d)

We provide the same plot as before, but with the Moore-Penrose solution x_{MP} in orange. This solution corresponds to $p = 2$, which follows the presentation (with proof) given in the notes that the moore-penrose solution corresponds to the least-squares, minimum norm solution.

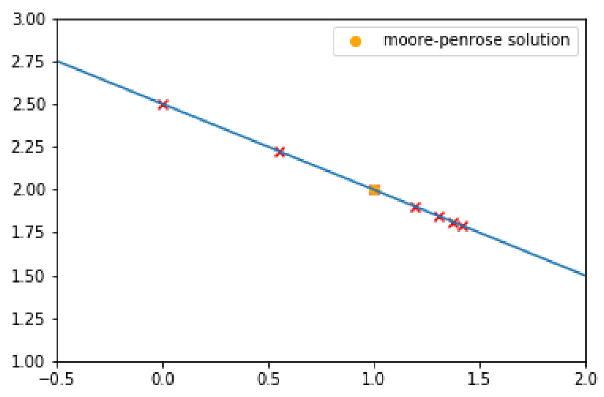


Figure 2: Moore-Penrose solution

2 Singular Value Decomposition

2.1 (a)

A grid with `n` regularly spaced intervals was set up using `x = np.linspace(-1,1, n)`.

2.2 (b)

We set `n=100` (so that $\delta n =$ `dn = 2/(n-1)`) and plotted the function

```
def g(x, dn, mu = 0, sigma = 0.2):
    return dn/(np.sqrt(2*np.pi)*sigma)*np.exp(-(x-mu)**2/(2*sigma**2))
```

given below:

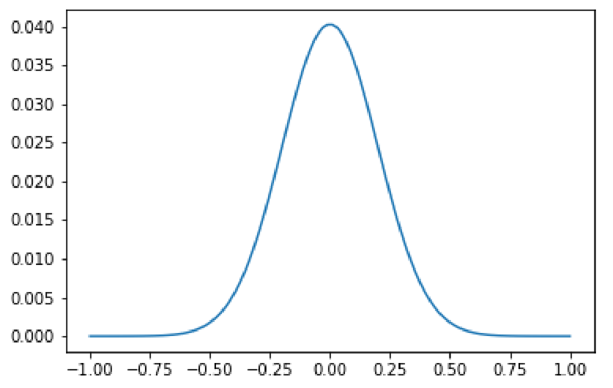


Figure 3: Gaussian function plot

2.3 (c)

The convolution matrix `A` was created via:

```
A = np.zeros(shape=(n,n))
for i in range(n):
    for j in range(n):
        A[i][j] = g(x[i] - x[j], dn)
```

2.4 (d)

It was then plotted in the case `n = 100` using the `matplotlib.pyplot` package:

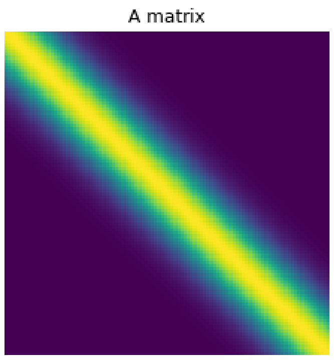


Figure 4: Convolution matrix A

2.5 (e)

The singular value decomposition of `A` was computed as `U,W,Vt = np.linalg.svd(A)`. In order to verify that the equation $A = UWV^\top$ is satisfied we computed the norm between `A` and the `A` reconstructed from the SVD: `diff = np.linalg.norm(A - U@np.diag(W)@Vt)`. When we performed this calculation, we found `diff = 1.252118594146205e-13`, i.e. an extremely small error between the true `A` and reconstructed `A`. This implies $A = UWV^\top$ is satisfied, since the difference can be attributed to computational innaccuracy.

2.6 (f)

We compute the pseudo-inverse $A^\dagger = VW^\dagger U^\top$ by the following method:

- perform the SVD on A: `U,W,Vt = np.linalg.svd(A)`
- form the sparse representation of $W^\dagger$: `Wdag = scipy.sparse.spdiags(np.divide(1,W), 0, n, n)`
- compute $A^\dagger = VW^\dagger U^\top$: `Adag = Vt.T @ Wdag @ U.T`

It is easy to check that $W^\dagger W = WW^\dagger = Id_n$ by computing `Wdag@np.diag(W)` and `np.diag(W)@Wdag`. We provide a plot of the form of $W^\dagger$ using the command `plt.spy(Wdag)` showing that it is indeed diagonal (a square indicates a non-zero value in the matrix):

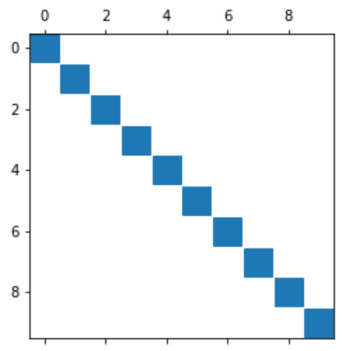


Figure 5: sparsity plot of $W^\dagger$

Furthermore we verify that our computation of $A^\dagger = VW^\dagger U^\top$ is correct by calculating:

```
np.linalg.norm(Adag - np.linalg.pinv(A))
```

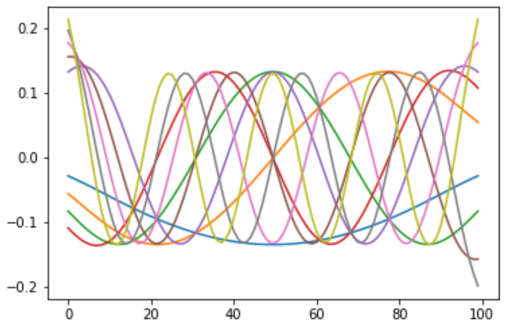
which in the case `n = 10` gave a difference of `2.8400347913994436e-15`, showing that our approach is almost identical.

2.7 (g)

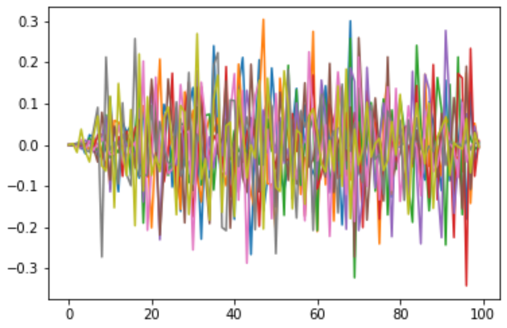
In the case `n = 20` we see that the numerical inaccuracies in the SVD and pseudo-inverse methods are amplified. Firstly we verify $A = UWV^\top$ is satisfied by computing as we did in **(e)** `diff = np.linalg.norm(A - U@np.diag(W)@Vt)`. This time `diff = 2.7568209020024993e-15` which is greater than the value `1.3713938405369824e-15` obtained in the case `n=10`.

Similarly we verify $A^\dagger = VW^\dagger U^\top$ as we did in **(f)** by calculating `np.linalg.norm(Adag -np.linalg.pinv(A))`. Again, for `n=20` we get a larger inaccuracy of `3.7416236697031865e-10` compared to `2.8400347913994436e-15`.

Therefore we see that in both sets of numerical computations, for a larger value of `n`, the numerical inaccuracies of the method become larger. In the case `n=100`, we also provide plots of the first and last 9 columns of V, and the singular values on a logarithmic scale:



(a) first 9



(b) last 9

Figure 6: Plot of columns of V

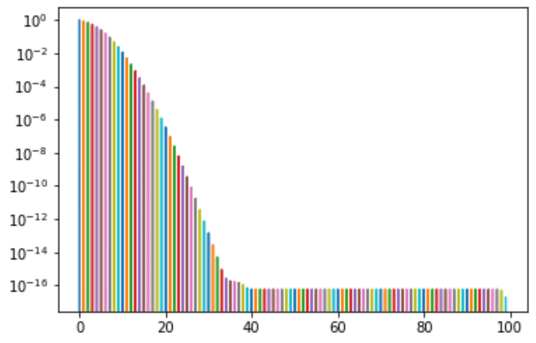


Figure 7: Singular Values of A

3 Convolutions and Fourier transform

3.1 (a)

We created the function $f(x)$ as follows:

```
def step(a,b,x):
  if a < x <= b:
      return 1
  else:
      return 0

def f(x):
      val = step(-0.95, -0.6, x) + 0.2*step(-0.6,0.2,x) -0.5*step(-0.2,0.2,x) \
            +0.7*step(0.4,0.6,x)-0.7*step(0.6,1,x)
  return val
```

then using a large grid ($n = 1000$), the plot of $f(x)$ is given below:

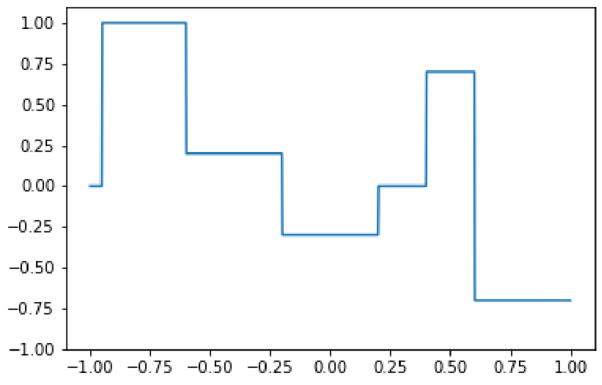


Figure 8: Plot of function $f(x)$

3.2 (b)

The singular value plots of A for $\sigma = 0.05, 0.1, 0.2$ are given (on a logarithmic scale)

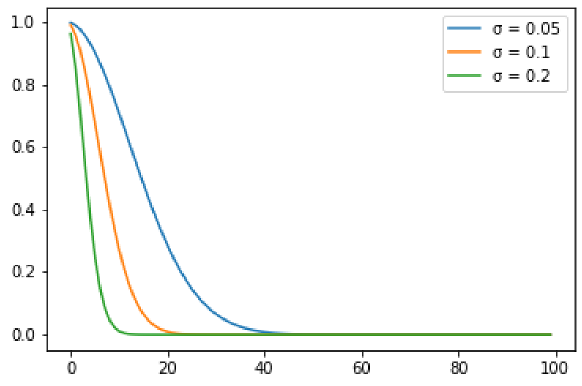
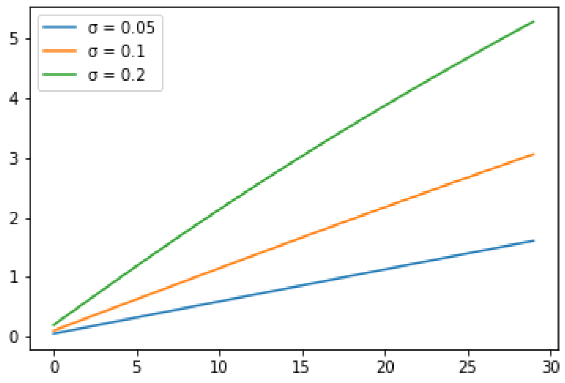


Figure 9: Singular values for varying σ

3.3 (c)

The singular values w_i follow a Gaussian function if we have that $w_i \propto e^{-\frac{i^2}{2\epsilon^2}}$, in which case we require that $i \propto \sqrt{2\epsilon^2 \log \frac{1}{w_i}}$ for $i \in \{1, \ldots, n\}$. Since the i are evenly spaced integer points it follows then that if w_i follow a gaussian function then a plot of $\sqrt{\log \frac{1}{w_i}}$ must be a straight line. We provide plots of this equation for $\sigma = 0.05, 0.1, 0.2$ and $i = 1, \ldots, 30$ below:



which verifies that the w_i do indeed follow a (half) Gaussian function. Furthermore we can estimate the variance ϵ via the slope of these lines. Note that the coefficent $\sqrt{2\epsilon^2}$ determines the slope and therefore we can find the variance as

$$\epsilon^2 \approx \frac{1}{2 * \text{slope}^2}$$

which when calculated numerically gave:

σ	0.05	0.1	0.2
slope	0.05	0.09	0.17
ϵ^2	194.71	52.47	16.36

3.4 (d)

The convolution of the function f with the matrix A is given by `conv = A @ f[:,np.newaxis]` This was performed with A formed with the three different values of σ, and the result plotted below:

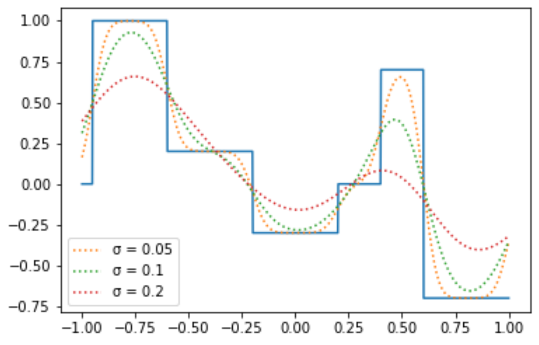


Figure 10: Convolution for varying σ

3.5 (e)

Note that the mean centred convolution kernel $K(x-y)$ is given by the $n/2$th row of the convolution matrix A. In practice we have that:

```
K = A[n//2]
```

Given our function $f(x) =$ `f`, and kernel `K`, we perform the convolution via multiplication in Fourier space and then apply the inverse Fourier transform to get the end result. This is achieved via the code:

```
Ff = np.fft.fftshift(np.fft.fft(np.fft.fftshift(f)))
FK = np.fft.fftshift(np.fft.fft(np.fft.fftshift(K)))
Fb = Fa*Ff
b = np.real(np.fft.fftshift(np.fft.ifft(np.fft.fftshift(Fb))))
plt.plot(xrange, b)
```

producing the plot

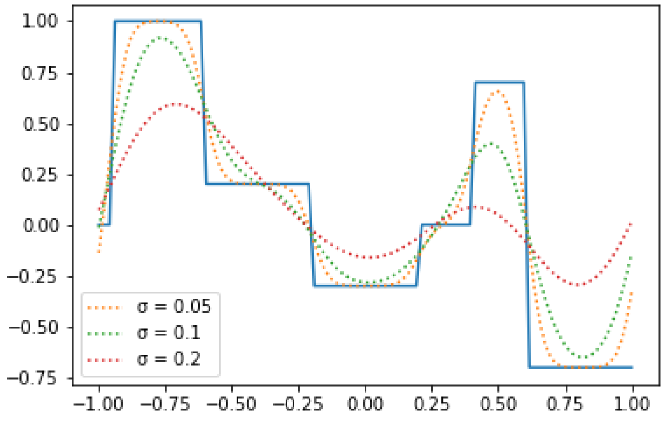


Figure 11: Convolution for varying σ

The difference we observe is that the result of the convolutions in fourier space now give solutions that are *periodic*. This is because the mean centred convolution kernel `K` is itself periodic. Note then when taking the normed difference between the solutions produced by the convolution methods, they were all significantly non-zero.

3.6 (f)

We form the convolution matrix A using periodic boundary conditions as follows. Firstly, we take the mean centred convolution kernel `K = A[n//2]` given by the $n/2$th row of A (as before in part **(e)**). This is plotted below:

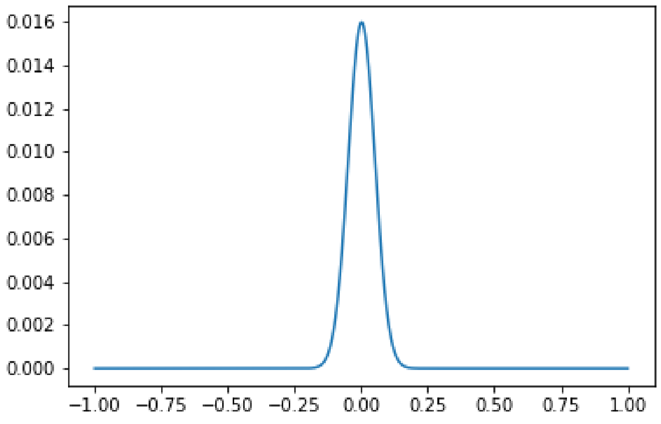


Figure 12: Mean centred convolution kernel

Secondly, we write K as an n-length sequence $[k_{n/2}, \ldots, k_1, k_0, k_1, \ldots, k_{n/2-1}]$ (for n even), noting the symmetry in k values around the mean. We can therefore form the periodic $A$ matrix as a circulant matrix below

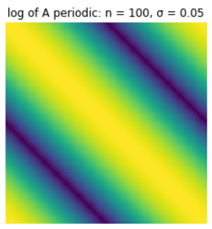
$$A = \begin{bmatrix} k_0 & k_1 & \ldots & k_{n/2-1} & k_{n/2} & k_{n/2-1} & \ldots & k_2 & k_1 \\ k_1 & k_0 & k_1 & \ldots & k_{n/2-1} & k_{n/2} & k_{n/2-1} & \ldots & k_2 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ k_{n/2-1} & \ldots & k_1 & k_0 & k_1 & \ldots & k_{n/2-4} & k_{n/2-3} & k_{n/2-2} \\ k_{n/2} & k_{n/2-1} & \ldots & k_1 & k_0 & k_1 & \ldots & k_{n/2-2} & k_{n/2-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ k_1 & k_2 & \ldots & k_{n/2-1} & k_{n/2} & k_{n/2-1} & \ldots & k_1 & k_0 \end{bmatrix} \tag{1}$$

where each row is obtained from the previous row by a circulant right shift. Note that the periodic boundary conditions are satisfied since, for any given row, the index of the last element in the row directly preceedes the index of the first element. In python we can form this matrix using the code:

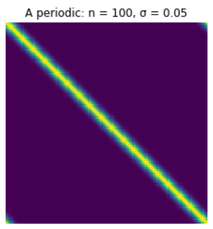
```
bd = n//2
k = A[bd,:] # k is the n/2 'th row of the convolution matrix $A$ (figure 12)
A_periodic = np.zeros(shape=(n,n)) # form our periodic matrix
for i in range(bd):
    A_periodic[i,:bd+i] = k[bd-i:]
    A_periodic[i,bd+i:] = k[:bd-i]

# operation to form bottom half of periodic matrix
x = np.flip(np.flip(A_periodic[:bd,:], axis=0), axis=1)
A_periodic[bd:,] = x
```

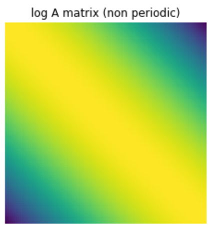
The resulting matrix is then visualised below for $n = 100$, $\sigma = 0.05$ with the left matrix produced by taking the log of the matrix values, and the right matrix produced by taking the log of the original (non-periodic) $A$ matrix



(a) log of `A_periodic`



(b) `A_periodic`



(c) log of `A` (non-periodic)

We then perform the convolution with the function $f(x)$ via matrix multiplication as in part **(d)**:

```
conv = A_periodic @ f[:,np.newaxis]
```

which produces the exact same result as fourier multiplication:

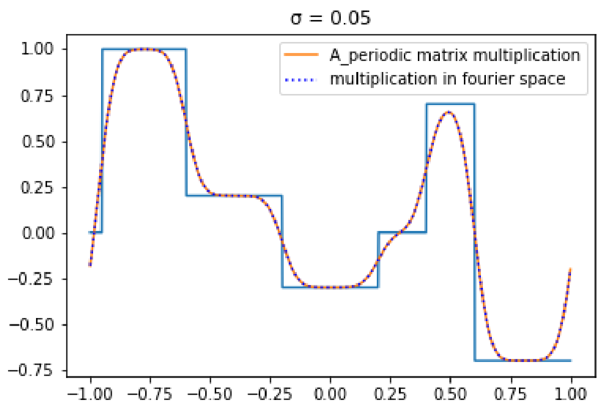


Figure 14: convolution with matrix `A_periodic`

since we see that the result of the convolutions are perfectly aligned. Furthermore we can easily check that multiplication in fourier space (producing result `b`) and the convolution with the periodic matrix (producing result `conv`) give the exact same result by performing the calculation:

```
diff = np.linalg.norm(conv.flatten() - b)
```

which gave `diff = 2.0429279379258072e-15`, showing the equivalency of the two methods.