Beihang University
School of General Engineering

Beijing, Dec. 20th, 2019
Sebastian Wandelt, Junyu Zhao, Pengfei Yi

Computer Science and Programming
Lab Class 14

Task 1. ***Numpy - Solving Linear Algebra Problems*** *(60 minutes)*

In this task, you will learn about the powerful scientific computing package `numpy`, particularly in its applications in Linear Algebra. In specific, the following contents cover Matrix construction, multiplication, finding eigenvalues, and more. After learning this task, one should have a basic understanding of the capacity of the package `numpy` and make good use of its capacity when needed.

1. Import `numpy` with the following code. Notice that the package `numpy` is abbreviated as `np`, which is more convenient and also widely used.

```
import numpy as np
```

2. Matrix Construction: Create `numpy` Matrix with the with the `np.matrix` function. One could create a matrix with the function using either a `list` or a `string`, as shown in the code below:

```
A1=np.matrix('1 2;3 4;5 6')
A2=np.matrix([[1,2,3],[4,5,6]])
A1
>>>[[1 2]
 [3 4]
 [5 6]]
A2
>>>[[1 2 3]
 [4 5 6]]
```

3. Matrix Multiplication: Use the `dot` function in `numpy` for matrix multiplication, as shown in the example below:

```
A1=np.matrix('1 2;3 4;5 6')
A2=np.matrix([[1,2,3],[4,5,6]])
B=A1.dot(A2)
C=A2.dot(A1)

B
>>>[[ 9 12 15]
 [19 26 33]
 [29 40 51]]

C
>>>[[22 28]
 [49 64]]
```

4. Finding the inverse matrix: Use the `np.linalg.inv` function to find the inverse matrix of a given matrix, as shown in the code below. Notice that `linalg` stands for Linear Algebra and there are also many other useful functions in this sub-package of `numpy`.

```
A=np.matrix('1 2;3 4')
A_inverse=np.linalg.inv(A)
A_inverse
>>> [[-2.   1. ]
 [ 1.5 -0.5]]
A.dot(A_inverse)
>>> [[1.00000000e+00 1.11022302e-16]
 [0.00000000e+00 1.00000000e+00]]
```

Note that the result of `A.dot(A_inverse)` is an identity matrix but the float system of `numpy` makes it a little bit strange. One alternative to avoid this is to use the `np.round` function to neglect the bothering decimals, as shown below:

```
np.round(A.dot(A_inverse),decimals=10)
>>> [[1. 0.]
 [0. 1.]]
```

5. Orthogonality test: Checking orthogonality needs checking whether the dot product of two matrices is a zero matrix, and one way of implementing the function is shown in the code below. Notice that in this case, the order of the two input is important because with the opposite order there is no dot product. Also, in the code the `np.zeros_like` function, which creates a zero matrix with the same dimension with the input matrix.

```
def check_orthogonal(A,B):
    Product=np.round(A.dot(B),decimals=10)
    return np.array_equal(Product,np.zeros_like(Product))
A=np.array([1,0,0])
B=np.matrix([[0,1,0],[0,0,1]])
check_orthogonal(B,A)
>>> True
```

6. Gram-Schmidt Algorithm: One could implement the famous Gram-Schmidt Algorithm using the structure of `numpy`, as shown in an example function below:

```
def Gram_Schmidt(A):
    vectors_A=[v for v in A]
    orthogonal_v=[]
    for v in vectors_A:
        v_copy = np.copy(v)
        for i in range(0,len(orthogonal_v)):
            norm_product=np.linalg.norm(orthogonal_v[i])*
            np.linalg.norm(orthogonal_v[i])
            v_copy -= float(np.dot(orthogonal_v[i],v.T))/
            norm_product*orthogonal_v[i]
        e = v_copy / np.linalg.norm(v_copy)
        orthogonal_v.append(np.round(e,decimals=5))
    new_v=[list(list(v)[0]) for v in orthogonal_v]
    Q=np.matrix(new_v)
    return (Q)

A=np.matrix('1. 2. -1.;-1. 3. 1.;4. -1. 0.')
Gram_Schmidt(A)
```

```
>>> [[ 0.40825   0.8165   -0.40825]
 [-0.57735   0.57735   0.57735]
 [ 0.70711   0.        0.70711]]
```

7. Least-squares approximation: Least-squares solves linear matrix equation $Ax = B$ by computing a vector x that minimizes the squared Euclidean 2-norm $||B - Ax||_2^2$. The equation may be under-, well-, or over-determined. If a is square and of full rank, then x is the "exact" solution of the equation.
   One can use the `np.linalg.lstsq` function to get the least-squares solution of the equation $Ax = B$. It takes matrix A and B as inputs, and returns four results:
   (a) The least-squares solution of x;
   (b) Sums of residuals;
   (c) Rank of matrix a;
   (d) Singular values of a.
   The code for solving the equation:
$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} x = \begin{bmatrix} -1 \\ 0.2 \\ 0.9 \\ 2.1 \end{bmatrix}$$
   is shown below.

```
A=np.matrix('0 1;1 1;2 1;3 1')
B=np.matrix('-1;0.2;0.9;2.1')
np.linalg.lstsq(A, B)
>>> (matrix([[ 1.   ], [-0.95]]),
    matrix([[0.05]]),
    2,
    array([4.10003045, 1.09075677]))
```

   The solution is $[1, -0.95]^T$.

8. Determinant: `np.linalg.det` is used to compute the determinant of a given matrix. It takes a $N \times N$ matrix as input, and returns the determinant of the matrix. Example is as shown in the code below.

```
A = np.matrix([[1, 2], [3, 4]])
np.round(np.linalg.det(A), decimals=10)
>>> -2.0
```

9. Eigenvalues: Use `np.linalg.eig` to compute the eigenvalues and right eigenvectors of a square square. It takes a $N \times N$ matrix as input, and returns:
   (a) w ($1 \times N$ matrix). The eigenvalues, each repeated according to its multiplicity. The eigenvalues are not necessarily ordered.
   (b) v ($N \times N$ matrix) The normalized (unit length) eigenvectors, such that the column `v[:,i]` is the eigenvector corresponding to the eigenvalue `w[i]`.
   The following code shows computing the eigenvalues, together with eigenvectors of the matrix A. It can be seen that the eigenvalues are $4, -2, -2$. And the corresponding eigenvectors are $[-0.41, -0.41, -0.82]^T$, $[-0.82, -0.32, 0.50]^T$, and $[0.20, -0.60, -0.78]^T$.

```
A = np.matrix([[1, -3, 3], [3, -5, 3], [6, -6, 4]])
np.linalg.eig(A)
>>>(array([ 4., -2., -2.]),
    matrix([[-0.40824829, -0.81034214,  0.1932607 ],
            [-0.40824829, -0.31851537, -0.59038328],
            [-0.81649658,  0.49182677, -0.78364398]]))
```

10. Matrix Exponentiation: Use `np.linalg.matrix_power` to raise a square matrix to the (integer) power n. It takes the matrix A and the power n as inputs, and returns A^n. Example is as shown in the code below.

```
A = np.matrix([[1, 2], [2, 3]])
>>>matrix([[ 5,  8],
           [ 8, 13]])
```

11. FFT for signal reduction: FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. Model `numpy.fft` provides functions of a variety of FFTs. See `https://docs.scipy.org/doc/numpy-1.14.1/reference/routines.fft.html` for details. The code below shows the FFT for a real input

$$e^{j\frac{2\pi}{8}k}, k = 0, 1, ..., 7$$

The output is a Hermitian matrix, i.e., symmetric in the real part and anti-symmetric in the imaginary part.

```
np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
>>>array([ -3.44505240e-16 +1.14383329e-17j,
            8.00000000e+00 -5.71092652e-15j,
            2.33482938e-16 +1.22460635e-16j,
            1.64863782e-15 +1.77635684e-15j,
            9.95839695e-17 +2.33482938e-16j,
            0.00000000e+00 +1.66837030e-15j,
            1.14383329e-17 +1.22460635e-16j,
           -1.64863782e-15 +1.77635684e-15j])
```

An application of FFT is signal processing. The following example shows manipulating FFTs for signal noise reduction in python by applying the `fft`, `fftfreq`, `ifft` functions from the `np.fft` package.
The input signal is a sine wave signal with random noise:

$$s(t) = sin(\frac{2\pi}{5}t) + n(t)$$

(a) The following code shows importing related packages and generating the original signal. The signal is shown in figure 1

```
import numpy as np
from matplotlib import pyplot as plt
# generate the signal
np.random.seed(1234)
time_step = 0.02
```

```
period = 5.
time_vec = np.arange(0, 20, time_step)
sig = (np.sin(2 * np.pi / period * time_vec)
      + 0.5 * np.random.randn(time_vec.size))
# plot the signal
plt.figure(figsize=(6, 5))
plt.plot(time_vec, sig, label='Original signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
```

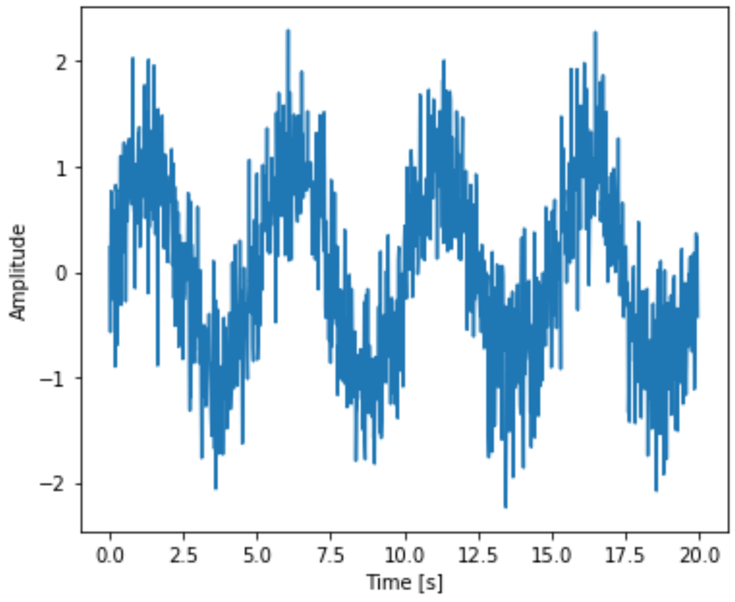

Figure 1: Original signal.

(b) By applying FFT, one can observe signal in frequency domain. The following code shows applying FFT for the signal, getting the power, and find the peak frequency. The power of the signal is shown in figure 2, with a peak frequency $\frac{1}{5} = 0.2$ Hz.

```
# The FFT of the signal (a complex dtype)
sig_fft = np.fft.fft(sig)
# The power
power = np.abs(sig_fft)
# The corresponding frequencies
sample_freq = np.fft.fftfreq(sig.size, d=time_step)
# Find the peak frequency
# Focus on only the positive frequencies
pos_mask = np.where(sample_freq > 0)
freqs = sample_freq[pos_mask]
peak_freq = freqs[power[pos_mask].argmax()]
# Plot the FFT power
plt.figure(figsize=(6, 5))
plt.plot(sample_freq, power)
# An inner plot to show the peak frequency
```

```
axes = plt.axes([0.55, 0.3, 0.3, 0.5])
plt.title('Peak frequency')
plt.plot(freqs[:8], power[:8])
plt.setp(axes, yticks=[])
plt.xlabel('Frequency [Hz]')
plt.ylabel('plower')
```

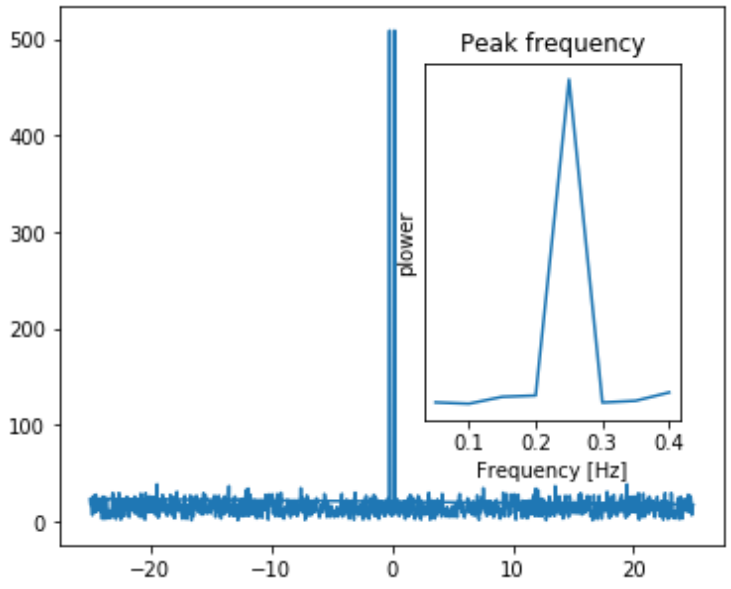

Figure 2: Signal power.

(c) Since the signal power is mainly concentrated at 0.2Hz (is actually the sine wave), removing all the high frequencies (actually caused by random noise) can play a noise reduction effect. This operation is actually a "filter". The code blow shows removing all the high frequencies and transform the signal back. The filtered signal is shown in figure 3.

```
#  The noise reduction process with FFT
high_freq_fft = sig_fft.copy()
high_freq_fft[np.abs(sample_freq) > peak_freq] = 0
filtered_sig = fftpack.ifft(high_freq_fft)
#  Plot the filtered signal
plt.figure(figsize=(6, 5))
plt.plot(time_vec, filtered_sig, linewidth=3)
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
```

Task 2. ***Kruskal's Algorithm*** *(20 minutes)*

Implement the famous Kruskal's algorithm in Python with the input graph constructed in `networkx`. Test the algorithm with the graph example shown in Figure 4. The graph shown is constructed with networkx as shown in the code below. One could simply copy the code and start implementing the algorithm with the functions of networkx library.

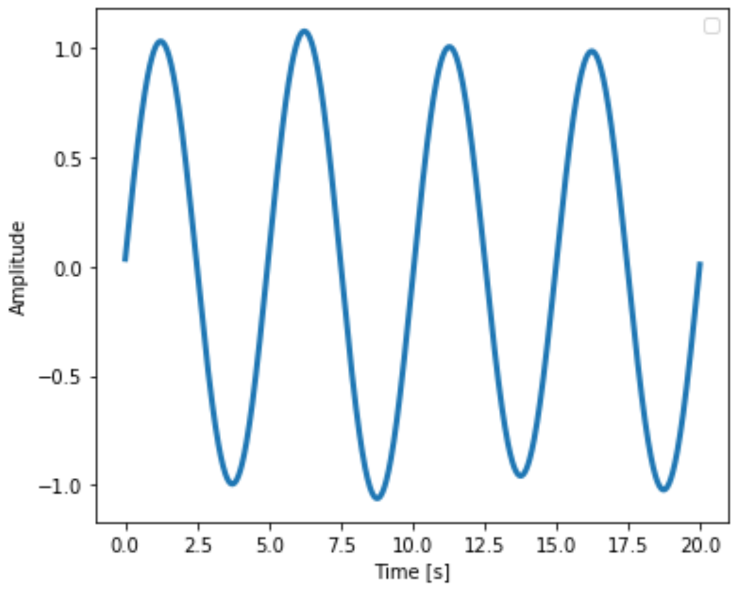

Figure 3: Filtered signal.

```
import networkx as nx
G=nx.Graph()
G.add_edge('A','B',weight=6)
G.add_edge('A','C',weight=2)
G.add_edge('C','D',weight=1)
G.add_edge('C','E',weight=7)
G.add_edge('C','F',weight=9)
G.add_edge('A','D',weight=3)
G.add_edge('G','E',weight=7)
G.add_edge('G','C',weight=4)
G.add_edge('E','D',weight=5)
```

Q = [{[illegible]}, {B}, [illegible] -- {[illegible]}]

{A, C, D}

G

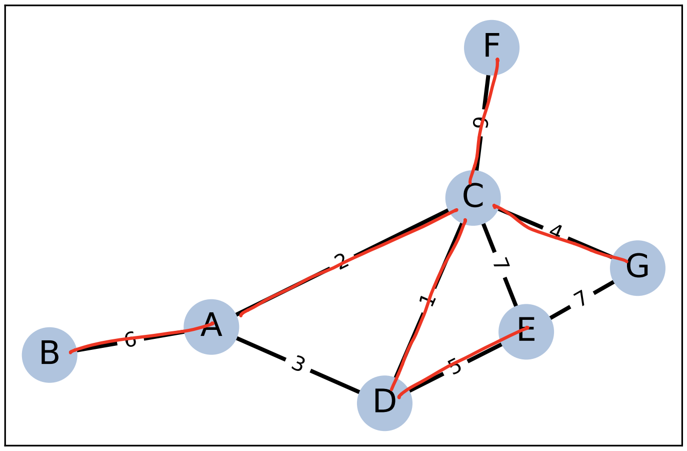

Figure 4: The graph example for Task 1.