Πανεπιστημιο

Σχολη

Γραμμική Άλγεβρα

Επιβλέπων:
Ονοματεπώνυμο

Συγγραφέας:
Ονοματεπώνυμο

Περίληψη

Σε αυτή την εργασία μελετάμε βασικά θέματα Γραμμικής Άλγεβρας μέσω της βιβλιοθήκης της Python, numpy. Στην αρχή, αναφέρουμε τα βασικά αντικείμενα(πίνακες, διανύσματα) και τις κύριες πράξεις(εσωτερικό γινόμενο, πολλαπλασιασμός πινάκων κλπ) στη γραμμική άλγεβρα. Στη συνέχεια, συνδέουμε τους πίνακες με την επίλυση γραμμικού συστήματος, και αναφέρουμε βασικούς αριθμητικούς αλγόριθμους όπως η απαλοιφή Gauss και η μέθοδος Jacobi. Παρακάτω, αναλύουμε τις διαφορετικές παραγοντοποιήσεις και την εύρεσή τους μέσω έτοιμων συναρτήσεων. Τέλος, συζητάμε για τη γραμμική μέθοδο ελαχίστων τετραγώνων ενώ έχουμε δώσει ένα απλό παράδειγμα για τις μεθόδους συμπίεσης εικόνας μέσω της αποσύνθεσης σε ιδιάζουσες τιμές.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Η numpy είναι η βασική βιβλιοθήκη της Python για εφαρμογές επιστημονικών υπολογισμών. Περιλαμβάνει τις περισσότερες μεθόδους γραμμικής άλγεβρας και αριθμητικής ανάλυσης. Έχουν γίνει αναλυτικές εργασίες σχετικά με την χρήση της numpy για γραμμικούς υπολογισμούς. Η βιβλιογραφία που θα χρησιμοποιηθεί ([1] [4] [2] [3]) βρίσκεται στο τέλος της παρούσας εργασίας. Έχουν επίσης χρησιμοποιηθεί documentation της βιβλιοθήκης καθώς και άλλες βιβλιοθήκες του οικοσυστήματος της Python όπως η scipy η οποία είναι γραμμένη με χρήση της numpy.

Κεφάλαιο 2

Διανύσματα, Πίνακες και Βασικές Πράξεις

Η δομή την οποία μελετάμε στην γραμμική άλγεβρα είναι ο πίνακας. Ο πίνακας, είναι μία ορθογώνια διάταξη από αριθμούς, σύμβολα ή εκφράσεις. Συμβολίζουμε έναν πίνακα m γραμμών και n στηλών ως:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = (a_{ij}) \in \mathbb{R}^{m \times n} \tag{2.1}$$

με $i = 0, 1 \cdots m, \quad j = 0, 1 \cdots n.$ Στην numpy, το αντικείμενο που χρησιμοποιούμε για την δήλωση πίνακα είναι το array():

```
#####create the array#####
A = np.array([[1,0,1],[2,1,0],[3,1,0],[0,4,5]])
#####display the array#####
print(A)
```

```
[[1 0 1]
 [2 1 0]
 [3 1 0]
 [0 4 5]]
```

Μπορούμε να χρησιμοποιήσουμε τα κατάλληλα attributes για να δούμε τον τύπο των δεδομένων στον πίνακα αλλά και το σχήμα της διάταξης:

```
######type of data#######
print(A.dtype)
########shape#########
print(A.shape)
```

```
int32
(4, 3)
```

Ένα ακόμη χρήσιμο attribute είναι ο υπολογισμός του ανάστροφου:

```
######transpose#######
print(A.T)
#######shape########
print(A.T.shape)
```

```
[[1 2 3 0]
 [0 1 1 4]
 [1 0 0 5]]
(3, 4)
```

ο οποίος ορίζεται ως:

$$\mathbf{A}^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix} = (a_{ji}) \in \mathbb{R}^{m \times n} \tag{2.2}$$

με $i = 0, 1 \cdots m, \quad j = 0, 1 \cdots n.$ Δηλαδή μετατρέπονται οι γραμμές σε στήλες και αντιστρόφως.

Στο πλαίσιο της γραμμικής άλγεβρας το διάνυσμα δεν είναι τίποτα άλλο παρά ένας πίνακας με διάταξη $1 \times n$ ή $m \times 1$. Στην πρώτη περίπτωση καλείται και γραμμοδιάνυσμα ενώ στην δεύτερη περίπτωση στυλοδιάνυσμα. Προφανώς ο ανάστροφος ενός στυλοδιανύσματος μας δίνει το αντίστοιχο γραμμοδιάνυσμα και αντιστρόφως.

Η numpy υποστηρίζει σχεδόν κάθε πιθανή πράξη μεταξύ πινάκων. Οι απλοί τελεστές τις πψτηον: $+, -, *, /, **, \%, //$ εκτελούν τις πράξεις όπως λέμε εν ελεμεντωισε, δηλαδή ανά στοιχείο των πινάκων με ακριβής αντιστοίχιση. Έστω ότι οι παραπάνω τελεστές

συμβολίζονται με @, τότε η χρήση των παραπάνω συμβόλων μεταξύ δύο np.array δίνουν:

$$\mathbf{A@B} = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix} @ \begin{pmatrix} b_{11} & b_{21} & \cdots & b_{m1} \\ b_{12} & b_{22} & \cdots & b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1n} & b_{2n} & \cdots & b_{mn} \end{pmatrix} \tag{2.3}$$

$$= \begin{pmatrix} a_{11}@b_{11} & \cdots & a_{m1}@a_{m1} \\ \vdots & \ddots & \vdots \\ a_{1n}@b_{1n} & \cdots & a_{mn}@b_{mn} \end{pmatrix} \tag{2.4}$$

Για παράδειγμα με χρήση του modulo %:

```
#######define A and B#######
A=np.array([[4,2,2],[3,5,6],[2,2,3],[8,10,9]])
B=np.array([[2,2,2],[1,3,1],[2,1,10],[4,5,5]])
#####display elementwise modulo########
print(A%B)
```

```
[[0 0 0]
 [0 2 0]
 [0 0 3]
 [0 0 4]]
```

Οι βασικές πράξεις μεταξύ πινάκων είναι η πρόσθεση και ο πολλαπλασιασμός πινάκων. Η πρόσθεση πινάκων είναι όμοια με την πρόσθεση ανά στοιχείο συνεπώς δεν χρειάζεται κάποια ειδική συνάρτηση για τον υπολογισμό της. Αντιθέτως, ο πολλαπλασιασμός πινάκων είναι ιδιαίτερη πράξη αποτελούμενη από στοιχεία c_{ij} ίσα με το εσωτερικό γινόμενο του γραμμοδιανύσματος i του πρώτου πίνακα, και του στηλοδιανύσματος j του δεύτερου πίνακα. Πιο συγκεκριμένα, το γινόμενο δύο πινάκων A(με $m \times n$) και B(με $n \times p$) το οποίο συμβολίζεται με AB(ένας πίνακας $m \times p$) ορίζεται ως:

$$\mathbf{AB} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix} \tag{2.5}$$

όπου αν προσέξουμε κάθε στοιχείο είναι εσωτερικό γινόμενο διανυσμάτων $a_i \cdot b_j$ (i για τις γραμμές, j για τις στήλες):

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj} \tag{2.6}$$

Ο τρόπος με τον οποίο πολλαπλασιάζουμε πίνακες στην np.numpy είναι με την χρήση της εντολής np.matmul:

```
#######define A and B#######
A=np.array([[4,2,2],[3,5,6],[2,2,3],[8,10,9]])
B=np.array([[2,2,2],[1,3,1],[2,1,10],[4,5,5]]).T
###calculate matrix product###
print(np.matmul(A,B))
```

```
[[ 16  12  30  36]
 [ 28  24  71  67]
 [ 14  11  36  33]
 [ 54  47 116 127]]
```

Σε περίπτωση που οι διαστάσεις των πινάκων δεν είναι κατάλληλες για τον πολλαπλασιασμό τους, η συγκεκριμένη εντολή θα επιστρέψει ValueError.

Αν έχω δύο διανύσματα και θέλω να υπολογίσω το εσωτερικό τους γινόμενο χωρίς να περιπλέξουμε την διαδικασία με το αν είναι σε μορφή στήλης, ή σε μορφή γραμμής, μπορούμε να χρησιμοποιήσουμε την εντολή dot ως εξής:

```
######define two vectors##########
x = np.array([1,2,1])
y = np.array([3,0,1])
####do we get the same result?####
print(np.dot(y,x)==np.dot(x,y))
######what is that result?########
print(np.dot(x,y))
```

```
True
4
```

Σε αυτή την εντολή θα μπορούσαμε να έχουμε ορίσει το διάνυσμα y ως np.array([[3],[0],[1]]). Σε αυτή την περίπτωση η εντολή np.dot(x,y) επιστρέφει την 1επί1 np.array([4]), ενώ η εντολή np.dot(x,y) δεν αποδέχεται την μη συμβατότητα των διαστάσεων και πετάει error. Φυσικά, το ίδιο συμβαίνει αν και τα δύο διανύσματα έχουν την μορφή 1επί3.

Κεφάλαιο 3

Αριθμητική Επίλυση Γραμμικού Συστήματος

Το πιο σημαντικό πρόβλημα που καλούμαστε να λύσουμε μέσω της γραμμικής άλγεβρας είναι η επίλυση γραμμικών συστημάτων. Σχεδόν, οποιαδήποτε μεθοδολογία διδάσκετε έχει έμμεση ή άμεση σχέση με την επίλυση γραμμικού συστήματος. Γενικά, ένα γραμμικό σύστημα είναι μία εξίσωση της μορφής:

$$A\mathbf{x} = \mathbf{b} \tag{3.1}$$

όπου A είναι ένας $m \times n$ πίνακας, $\mathbf{x}$ είναι ένα στηλοδιάνυσμα με n βαθμωτές μεταβλητές , και το $\mathbf{b}$ είναι πάλι στηλοδιάνυσμα αλλά με m στοιχεία.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{3.2}$$

Ουσιαστικά, έχουμε το σύνολο των εξισώσεων:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \tag{3.3}$$

με αγνώστους τα $x_1, x_2 \cdots x_n$, όπου για να έχουμε ένα καλώς ορισμένο πρόβλημα θα πρέπει τα αθροίσματα των γραμμών του πίνακα A να μην είναι 0.

Προκειμένου λοιπόν να επιλύσουμε το σύστημα αριθμητικά θα πρέπει να εκμεταλλευ-

τούμε την γνώση μας για τα στοιχεία του πίνακα A και του b. Οι αλγοριθμικοί τρόποι επίλυσης που χρησιμοποιούν αυτά τα δύο αριθμητικά αντικείμενα είναι εύκολα προγραμματίσιμη με την χρήση της numpy. Υπάρχουν δύο βασικές κατηγορίες αριθμητικών μεθόδων για την επίλυση γραμμικών συστημάτων, οι ακριβείς και οι επαναληπτικές. Πριν ξεκινήσουμε να δίνουμε παραδείγματα μεθόδων δείχνουμε με ένα μικρό script πόσο εύκολα μπορεί η numpy να λύσει ένα γραμμικό σύστημα:

```
from numpy.linalg import solve
import numpy as np
######define the system#######
A = np.array([[2,1],[3,1]])
b = np.array([1,1])
####print the variable vector#####
print(solve(A,b))
```

```
[0. 1.]
```

3.1 Αντίστροφος πίνακας και ορίζουσα

Όταν μιλάμε για επίλυση συστήματος γραμμικών εξισώσεων ουσιαστικά το πρόβλημα που προσπαθούμε να λύσουμε είναι η εύρεση του αντίστροφου του πίνακα A. Πολλές φορές μάλιστα βλέπουμε στη βιβλιογραφία, τις μεθόδους επίλυσης συστήματος (πχ απαλοιφή Gauss) να αναφέρονται ως μέθοδοι υπολογισμού αντίστροφου πίνακα. Ο αντίστροφος πίνακας B ενός αντιστρέψιμου $n \times n$ πίνακα A είναι ο πίνακας για τον οποίο ισχύει:

$$AB = BA = I_n \tag{3.4}$$

και συμβολίζεται με A^{-1}. Το σύμβολο I_n αναπαριστά τον μοναδιαίο πίνακα με την ιδιότητα $I_n A = A$ και την μορφή:

$$I_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{3.5}$$

Ένας πίνακας A είναι αντιστρέψιμος αν και μόνο αν η ορίζουσα $detA$ του πίνακα είναι διαφορετική του μηδενός. Η ορίζουσα είναι ουσιαστικά μία συνάρτηση από των χώρο του πίνακα $\mathbb{R}^{n \times n}$ στον άξονα των πραγματικών αριθμών $\mathbb{R}$. Ο τύπος υπολογισμού για

αυθαίρετη διάσταση n είναι το ανάπτυγμα Laplace στο οποίο προστίθενται/αφαιρούνται όλες οι ορίζουσες των υποπινάκων του A πολλαπλασιασμένες με το κατάλληλο στοιχείο του πίνακα. Αντί να σχολιάσουμε σε βάθος τον αναλυτικό τύπο θα εξηγήσουμε την περίπτωση των δύο και των τριών διαστάσεων. Η λογική είναι προφανώς μεταβιβάσιμη και γενικεύεται για οποιονδήποτε αριθμό διαστάσεων.

Στις δύο διαστάσεις η ορίζουσα ορίζεται ως:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc \tag{3.6}$$

ενώ στις 3 διαστάσεις:

$$\begin{aligned} det(A) = |A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} &= a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= aei + bfg + cdh - ceg - bdi - afh \end{aligned} \tag{3.7}$$

Η απλοποιημένη λογική πίσω από τον υπολογισμό της ορίζουσας τριών διαστάσεων(και κατ΄ επέκταση όλων των διαστάσεων) έχει ως εξής:

- Επιλέγουμε την πρώτη γραμμή του πίνακα.
- Στη συνέχεια διαδοχικά επιλέγουμε τις στήλες του πίνακα μία-μία από αριστερά προς τα δεξιά
- Για κάθε διαδοχική επιλογή στήλης, στο αποτέλεσμα της ορίζουσας προσθέτουμε ή αφαιρούμε εναλλάξ, το γινόμενο του αντίστοιχου αριθμού της πρώτης γραμμής με την ορίζουσα του υποπίνακα που δεν είναι επιλεγμένος.
- Επαναλαμβάνουμε την διαδικασία μέχρι να καταλήξουμε σε πραγματικό αριθμό.

Φυσικά στην περίπτωση της numpy μπορούμε εύκολα να υπολογίσουμε την ορίζουσα ενός πίνακα ως:

```
from numpy.linalg import det
A = np.array([[2,1],[3,1]])
print(det(A))
```

```
-1.0000000000000002
```

Στην περίπτωση που ο πίνακας που εισάγουμε στην εντολή det δεν είναι τετραγωνικός, θα δούμε στο output την ένδειξή LinAlgError.

Ο τρόπος με τον οποίο συνδέεται το πρόβλημα υπολογισμού του αντίστροφου με την επίλυση γραμμικού συστήματος έρχεται αν πολλαπλασιάσει κανείς την εξίσωση (3.8) από αριστερά με τον πίνακα A^{-1}:

$$\mathbf{x} = A^{-1}\mathbf{b} \tag{3.8}$$

Έτσι κάθε επίλυση συστήματος με ίδιο αριθμό μεταβλητών και εξισώσεων (δηλαδή τετραγωνικό πίνακα A) μπορεί να αναχθεί σε πρόβλημα υπολογισμού αντιστρόφου. Ας λύσουμε με αυτόν τον τρόπο το προηγούμενο σύστημα εξισώσεων χρησιμοποιώντας numpy:

```
from numpy.linalg import inv
from numpy import matmul
######define the system#######
A = [[2,1],[3,1]]
b = [1,1]
####calculate solution vector x#####
x = matmul(inv(A),b)
print(x)
```

```
[1.11022302e-16 1.00000000e+00]
```

Παρατηρούμε μία ιδιαιτερότητα της numpy. Επειδή κάποιες εντολές και υπολογισμοί, περιέχουν αριθμητικές επαναληπτικές μεθόδους, σε συνδυασμό με την δυνατότητα της βιβλιοθήκης να διατηρεί πολύ μεγάλη ακρίβεια στους υπολογισμούς της, υπάρχει περίπτωση μια λύση να τείνει προσεγγιστικά στην πραγματική τιμή. Αν παρατηρήσουμε το πρώτο στοιχείο του διανύσματος x, βλέπουμε ότι είναι ένας αριθμός της τάξης του 10^{-16} δηλαδή πολύ κοντά στο 0. Αυτό συμβαίνει λόγω του αλγόριθμου υπολογισμού. Αν θέλουμε να στρογγυλοποιήσουμε για περισσότερη ευκρίνεια:

```
####round to fifth decimal#####
print(x.round(5))
```

```
[0. 1.]
```

Παρατηρούμε ότι στην περίπτωση που ο πίνακας δεν είναι αντιστρέψιμος δηλαδή το σύστημά μας είναι είτε αόριστο είτε αδύνατο, η πψτηον θα τυπώσει LinAlgError: Singular matrix.

Κεφάλαιο 4

Απαλοιφή Gauss

Μία από τις ακριβείς αριθμητικές μεθόδους επίλυσης γραμμικών συστημάτων είναι η Απαλοιφή Gauss. Η απαλοιφή Gauss βασίζεται στην μετατροπή του επαυξημένου πίνακα σε κλιμακωτή ή ανηγμένη κλιμακωτή μορφή. Πριν ορίσουμε όλα τα παραπάνω, ας μιλήσουμε για τις πράξεις που μπορούμε να κάνουμε μέσα σε ένα πίνακα ενός συστήματος χωρίς να επηρεάζει το αποτέλεσμα.

- Μπορούμε να προσθαφαιρέσουμε γραμμές μεταξύ τους.
- Μπορούμε να πολλαπλασιάσουμε μία γραμμή με έναν αριθμό.
- Μπορούμε να αλλάξουμε θέση δύο γραμμές ή δύο στήλες (εκτός της τελευταίας του επαυξημένου).

Αν εκτελεστούν αυτές τις πράξεις σε ένα σύστημα και στη συνέχεια στον πίνακα του συστήματος, είναι εύκολο να δει κανείς γιατί δεν επηρεάζουν την λύση του συστήματος. Είναι σαν να εκτελούμε γραμμικές πράξεις μεταξύ εξισώσεων.

Ο επαυξημένος πίνακας ενός συστήματος είναι ο πίνακας της μορφής:

$$(\mathbf{A} \mid \mathbf{b}) = \left[\begin{array}{ccc|c} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & b_n \end{array}\right] \tag{4.1}$$

Όπου προφανώς έχουμε τετραγωνικό πίνακα αριστερά (λόγω μεταβλητών και εξισώσεων) και το διάνυσμα b από δεξιά. Μπορούμε λοιπόν να λύσουμε το σύστημα με τρεις τρόπους, ανάγοντας με πράξεις γραμμών και στηλών τον επαυξημένο πίνακα σε τρεις διαφορετικούς πίνακες:

- Κλιμακωτή μορφή.

- Ανηγμένη κλιμακωτή μορφή.
- Μοναδιαίος πίνακας.

Η κλιμακωτή μορφή είναι αυτή που θα κατασκευάσουμε με τον αλγόριθμο της απαλοιφής Gauss. Στην πράξη δεν χρειαζόμαστε κάποιον αλγόριθμο απαραίτητα γιατί μπορούμε να σκεφτούμε τις κατάλληλες γραμμοπράξεις για να έχουμε το γρήγορο επιθυμητό αποτέλεσμα. Κλιμακωτός πίνακας είναι ο επαυξημένος πίνακας όταν αριστερά της γραμμής διαχωρισμού (δηλαδή στην μεριά του πίνακα A) υπάρχει ένας άνω τριγωνικός πίνακας. Ο ανηγμένος κλιμακωτός αναφέρεται όταν ο πίνακας αυτός είναι άνω τριγωνικός αλλά έχει και τα διαγώνια στοιχεία του ίσα με 1. Είναι προφανές ότι αν σχηματίσουμε αριστερά της διαχωριστικής γραμμής τον μοναδιαίο πίνακα ουσιαστικά έχουμε λύσει το σύστημα.

Ο αλγόριθμος θα πρέπει να εφαρμοσθεί σε πίνακα με μη μηδενικά διαγώνια στοιχεία. Για να φέρουμε τον πίνακα σε ανηγμένη κλιμακωτή μορφή:

```
Gauss Elimination on Matrix A:

For i = 1 to n-1
    For j = i+1 to n
         Ratio = Aj,i/Ai,i
         For k = 1 to n+1
              Aj,k = Aj,k - Ratio * Ai,k
         Next k
    Next j
Next i
```

Για να βρούμε την λύση του συστήματος αντικαθιστούμε από κάτω προς τα πάνω τις μεταβλητές μας.

```
Obtaining Solution by Back Substitution:

Xn = An,n+1/An,n
For i = n-1 to 1 (Step: -1)
     Xi = Ai,n+1
     For j = i+1 to n
         Xi = Xi - Ai,j * Xj
     Next j
     Xi = Xi/Ai,i
Next i
```

Ας χρησιμοποιήσουμε τον παραπάνω αλγόριθμο χειροκίνητα για να λύσουμε το πα-

ρακάτω σύστημα βήμα-βήμα με χρήση της numpy:

$$\begin{cases} 1x_1 + 2x_2 - 3x_3 = 7 \\ 3x_1 - x_2 + 4x_3 = 6 \\ -2x_1 + x_2 + x_3 = 1 \end{cases} \tag{4.2}$$

Κατασκευάζουμε τον επαυξημένο πίνακα του συστήματος. Η εντολή concatenate μπορεί να ενώσει δύο np.array, είτε βάζοντας το δεύτερο np.array από κάτω (axis = 0), είτε βάζοντας το δεύτερο np.array από δεξιά (axis = 1)

```
import numpy as np

######define the system######
A = np.array([[1,2,-3],[3,-1,4],[-2,1,1]])
b = np.array([[7,6,1]])
######define the augmented matrix#######
augmat = np.concatenate((A,b.T),axis=1)
print(augmat)
```

```
[[ 1  2 -3  7]
 [ 3 -1  4  6]
 [-2  1  1  1]]
```

Σχεδιάζουμε μια συνάρτηση που επιλέγει μία γραμμή από τον πίνακά μας και επιστρέφει αυτή τη γραμμή συμπληρωμένη με μηδενικά στις διαστάσεις του πίνακα.

```
#####pickUpfunction######
def pick_up(the_array,get_row,give_row):
    the_row = the_array[get_row]
    to_return = np.zeros(the_array.shape)
    for x in range(len(the_row)):
        to_return[give_row,x] = the_row[x]
    return to_return
```

Πάμε τώρα να εφαρμόσουμε τον αλγόριθμο Gauss. Σημειώνουμε εδώ ότι προφανώς ξεκινάμε από το 0 την αρίθμηση των στοιχείων του A για λόγους συμβατότητας με την numpy και την python.

```
#####start the algorithm manually#####
i=0
j=1
ratio = augmat[j,i]/augmat[i,i]
augmat = augmat - ratio*pick_up(augmat,i,j)
print(augmat)
```

```
i=0
j=2
ratio = augmat[j,i]/augmat[i,i]
augmat = augmat - ratio*pick_up(augmat,i,j)
print(augmat)

i=1
j=2
ratio = augmat[j,i]/augmat[i,i]
augmat = augmat - ratio*pick_up(augmat,i,j)
print(augmat)
```

```
[[  1.   2.  -3.   7.]
 [  0.  -7.  13. -15.]
 [ -2.   1.   1.   1.]]
[[  1.   2.  -3.   7.]
 [  0.  -7.  13. -15.]
 [  0.   5.  -5.  15.]]
[[  1.          2.         -3.          7.        ]
 [  0.         -7.         13.        -15.        ]
 [  0.          0.          4.28571429   4.28571429]]
```

Τώρα πλέον έχουμε την ανηγμένη κλιμακωτή μορφή. Συνεπώς:

```
#####now substitute accordingly#######
x_3 = augmat[2,2]/augmat[2,3]

i = 1
x_2 = augmat[i,3]
j = 2
x_2 = x_2 - augmat[i,j]*x_3
x_2 = x_2/augmat[i,i]

i = 0
x_1 = augmat[i,3]
j = 1
x_1 = x_1 - augmat[i,j]*x_2
j = 2
x_1 = x_1 - augmat[i,j]*x_3
x_1 = x_1/augmat[i,i]

print(x_1,x_2,x_3)
```

```
1.9999999999999996 4.000000000000001 1.0000000000000004
```

Ελέγχουμε το αποτέλεσμά μας:

```
print(np.linalg.solve(A,b[0]))
```

```
[2. 4. 1.]
```

Κεφάλαιο 5

Μέθοδος Jacobi

Η πιο απλή επαναληπτική μέθοδος για την αριθμητική επίλυση γραμμικού συστήματος είναι η μέθοδος του Jacobi. Όπως και στην περίπτωση της απαλοιφής Gauss υποθέτουμε ότι το σύστημα έχει μοναδική λύση και ότι δεν υπάρχουν μηδενικά στη διαγώνιο του πίνακα A.

Η επαναληπτική μέθοδο αποτελείται από τον επαναλαμβανόμενο υπολογισμό του σετ $\{x_n\}$ με τις εξισώσεις:

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}\left(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n\right) \\ x_2 &= \frac{1}{a_{22}}\left(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n\right) \\ &\vdots \\ x_n &= \frac{1}{a_{nn}}\left(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1}\right) \end{aligned} \tag{5.1}$$

Συνήθως επιλέγουμε ως αρχικές τιμές των μεταβλητών μας το 0. Το διάνυσμα $\mathbf{x}$ θα προσεγγίζει την πραγματική λύση μετά από κάποιον αριθμό επαναλήψεων. Πάμε να εκτελέσουμε χειροκίνητα τη μέθοδο για μερικές επαναλήψεις. Πρώτα για ένα σύστημα που αποκλίνει και στη συνέχει για ένα που συγκλίνει.

```
import numpy as np
#######divergent example#########
######define the system######
A = np.array([[1,2,-3],[3,-1,4],[-2,1,1]])
b = np.array([7,6,1])

######we start fromm the zeroth vector######
x_1 = np.array([b[0]/A[0,0],b[1]/A[1,1],b[2]/A[2,2]])
print(f"First iteration: {x_1}")
```

```
x_2 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_1[1]-A[0,2]*x_1[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_1[0]-A[1,2]*x_1[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_1[0]-A[2,
                                    1]*x_1[1])])
print(f"Second iteration: {x_2}")

x_3 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_2[1]-A[0,2]*x_2[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_2[0]-A[1,2]*x_2[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_2[0]-A[2,
                                    1]*x_2[1])])
print(f"Second iteration: {x_3}")

x_4 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_3[1]-A[0,2]*x_3[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_3[0]-A[1,2]*x_3[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_3[0]-A[2,
                                    1]*x_3[1])])
print(f"Fourth iteration: {x_4}")

x_5 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_4[1]-A[0,2]*x_4[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_4[0]-A[1,2]*x_4[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_4[0]-A[2,
                                    1]*x_4[1])])
print(f"Fifth iteration: {x_5}")

x_6 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_5[1]-A[0,2]*x_5[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_5[0]-A[1,2]*x_5[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_5[0]-A[2,
                                    1]*x_5[1])])
print(f"Sixth iteration: {x_6}")

x_7 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_6[1]-A[0,2]*x_6[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_6[0]-A[1,2]*x_6[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_6[0]-A[2,
                                    1]*x_6[1])])
print(f"Sixth iteration: {x_7}")
```

```
First iteration: [ 7. -6.  1.]
Second iteration: [22. 19. 21.]
Second iteration: [ 32. 144.  26.]
Fourth iteration: [-203.  194.  -79.]
Fifth iteration: [-618. -931. -599.]
Sixth iteration: [   72. -4256.  -304.]
Sixth iteration: [ 7607. -1006.  4401.]
```

```
#######convergent example#########
######define the system######
A = np.array([[5,-2,3],[-3,9,1],[2,-1,-7]])
b = np.array([-1,2,3])

######we start fromm the zeroth vector######
x_1 = np.array([b[0]/A[0,0],b[1]/A[1,1],b[2]/A[2,2]])
print(f"First iteration: {x_1}")

x_2 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_1[1]-A[0,2]*x_1[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_1[0]-A[1,2]*x_1[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_1[0]-A[2,
                                    1]*x_1[1])])
print(f"Second iteration: {x_2}")

x_3 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_2[1]-A[0,2]*x_2[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_2[0]-A[1,2]*x_2[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_2[0]-A[2,
                                    1]*x_2[1])])
print(f"Second iteration: {x_3}")

x_4 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_3[1]-A[0,2]*x_3[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_3[0]-A[1,2]*x_3[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_3[0]-A[2,
                                    1]*x_3[1])])
print(f"Fourth iteration: {x_4}")

x_5 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_4[1]-A[0,2]*x_4[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_4[0]-A[1,2]*x_4[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_4[0]-A[2,
                                    1]*x_4[1])])
print(f"Fifth iteration: {x_5}")

x_6 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_5[1]-A[0,2]*x_5[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_5[0]-A[1,2]*x_5[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_5[0]-A[2,
                                    1]*x_5[1])])
print(f"Sixth iteration: {x_6}")

x_7 = np.array([(1/A[0,0])*(b[0]-A[0,1]*x_6[1]-A[0,2]*x_6[2]),(1/A[1,1])
                                    *(b[1]-A[1,0]*x_6[0]-A[1,2]*x_6[2])
                                    ,(1/A[2,2])*(b[2]-A[2,0]*x_6[0]-A[2,
```

```
                                        1]*x_6[1])])
print(f"Sixth iteration: {x_7}")
```

```
First iteration: [-0.2         0.22222222 -0.42857143]
Second iteration: [ 0.14603175  0.2031746  -0.51746032]
Second iteration: [ 0.19174603  0.32839506 -0.41587302]
Fourth iteration: [ 0.18088183  0.33234568 -0.42070043]
Fifth iteration: [ 0.18535853  0.32926066 -0.42436886]
Sixth iteration: [ 0.18632558  0.33116049 -0.42264909]
Sixth iteration: [ 0.18605365  0.33129176 -0.42264419]
```

Αξίζει να σημειωθεί ότι η συνθήκη σύγκλισης είναι:

$$\rho\left(D^{-1}(L+U)\right) < 1 \tag{5.2}$$

όπου $\rho(A) = \max\{|\lambda_1|, \ldots, |\lambda_n|\}$, D ο διαγώνιος πίνακας, L, U οι πίνακες από την παραγοντοποίηση LU.

Κεφάλαιο 6

Ιδιοτιμές, ιδιοδιανύσματα και διαγωνιοποίηση

Τα ιδιοδιανύσματα είναι διανύσματα για τα οποία ισχύει η παρακάτω εξίσωση:

$$A\mathbf{u} = \lambda\mathbf{u} \tag{6.1}$$

όπου Α είναι ένας τετραγωνικός πίνακας. Για να βρούμε τις ιδιοτιμές λ λύνουμε την εξίσωση:

$$|A - \lambda I| = (\lambda_1 - \lambda)(\lambda_2 - \lambda)\cdots(\lambda_n - \lambda) = 0 \tag{6.2}$$

Στη συνέχεια μελετάμε την τη αλγεβρική πολλαπλότητα, που αναφέρεται στην περίπτωση όπου μία εκ των ιδιοτιμών είναι πολλαπλή λύση στο παραπάνω πολυώνυμο. Η γεωμετρική πολλαπλότητα έχει να κάνει με τον διανυσματικό χώρο που σχηματίζουν τα ιδιοδιανύσματα των ιδιοτιμών. Στη συνέχεια υπολογίζουμε τα ιδιοδιανύσματα με την χρήση της (6.1) και βρίσκουμε την γεωμετρική πολλαπλότητα της κάθε ιδιοτιμής. Αν βρούμε n γραμμικώς ανεξάρτητα διανύσματα τότε ο πίνακάς μας είναι και διαγωνιοποιήσιμος. Αυτό σημαίνει ότι υπάρχει πίνακας P για τον οποίο:

$$P^{-1}AP = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \tag{6.3}$$

και αυτός θα είναι ο πίνακας των ιδιοδιανυσμάτων τοποθετημένα κατακόρυφα από αριστερά προς τα δεξιά. Ουσιαστικά παραπάνω έχουμε την διαγωνιοποιημένη μορφή του A. Αν πολλαπλασιάσουμε από αριστερά με τον πίνακα P και από δεξιά με τον πίνακα P παίρνουμε την ιδιοπαραγοντοποίηση του συστήματος. Πάμε να δούμε πως τα παραπάνω

υλοποιούνται με την numpy.

```
import numpy as np
import numpy.linalg as lin

# define example matrix
A = np.array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])

# factorize
###calculate eigenvalues and the corresponding eigenvectors###
values, vectors = lin.eig(A)

######we create the
P = vectors.T
D = np.diag(values)

print(f'The eigenvalues per column:{values}')
print(f'The corresponding eigenvectors:\n{P}')
```

```
The eigenvalues:[ 1.61168440e+01 -1.11684397e+00 -3.38433605e-16]
The corresponding eigenvectors:
[[-0.23197069 -0.52532209 -0.8186735 ]
 [-0.78583024 -0.08675134  0.61232756]
 [ 0.40824829 -0.81649658  0.40824829]]
```

Εδώ παρατηρούμε κατά τα γνωστά ότι η numpy χρησιμοποιεί αριθμητικές προσεγγιστικές μεθόδους για τον υπολογισμό των ιδιοτιμών και των ιδιοδιανυσμάτων. Στη συνέχει ελέγχουμε αν όντως ισχύει ο η εξίσωση (6.3) με χρήση του τελεστή @ που εκτελεί πολλαπλασιασμό πινάκων όμοιο με την np.matmul:

```
print(lin.inv(P)@D@P)
```

```
[[1. 4. 7.]
 [2. 5. 8.]
 [3. 6. 9.]]
```

Καταφέραμε λοιπόν να ιδιοπαραγοντοποιήσουμε τον πίνακα A. Οι παραγοντοποιήσεις είναι γενικά πολύ χρήσιμες στην γραμμική άλγεβρα. Θα δούμε παρακάτω επιπλέον τρόπους να παραγοντοποιούμε πίνακες μαζί με κάποιες εφαρμογές τους.

Κεφάλαιο 7

Παραγοντοποιήσεις

Έχουμε είδη δει την ιδιοπαραγοντοποίηση πίνακα. Θα δούμε σε αυτό το κεφάλαιο άλλες τρεις παραγοντοποιήσεις.

7.1 LU

Η παραγοντοποίηση LU αφορά την παραγοντοποίηση πίνακα σε ένα γινόμενο ενός κάτω τριγωνικού L και ενός άνω τριγωνικού U:

$$\mathbf{A} = \mathbf{LU} = \begin{pmatrix} \ell_{11} & 0 & 0 \\ \vdots & \ddots & 0 \\ \ell_{n1} & \cdots & \ell_{nn} \end{pmatrix} \begin{pmatrix} u_{11} & \cdots & u_{nn} \\ 0 & \ddots & \vdots \\ 0 & 0 & u_{nn} \end{pmatrix} \tag{7.1}$$

Υπάρχουν αλγόριθμοι να υπολογίσουν αναλυτικά την παραγοντοποίηση LU. Για να την υπολογίσουμε μέσω της Python θα χρησιμοποιήσουμε την scipy βιβλιοθήκη γραμμένη με χρήση της numpy:

```
from numpy import array
from scipy.linalg import lu

# define a square matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])

# factorize
P, L, U = lu(A)
print(P)
print(L)
```

```
print(U)

# reconstruct
print(P@L@U)
```

```
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
[[1.         0.         0.        ]
 [0.14285714 1.         0.        ]
 [0.57142857 0.5        1.        ]]
[[ 7.00000000e+00  8.00000000e+00  9.00000000e+00]
 [ 0.00000000e+00  8.57142857e-01  1.71428571e+00]
 [ 0.00000000e+00  0.00000000e+00 -1.58603289e-16]]
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

Παρατηρούμε ότι η αριθμητική παραγοντοποίηση χρησιμοποιεί και έναν πίνακα P. Αυτός ο πίνακας όπως φαίνεται και από το output χρησιμοποιείται για να περιστρέψει κάποιους υπόχωρους των τελικών πινάκων και δεν επηρεάζει τις αριθμητικές τιμές των L και U. Ο αλγόριθμος υπολογισμού PLU είναι πιο σταθερός και γι' αυτό επιλέχθηκε από τους developers της numpy.

7.2 SVD

Η ανάλυση πίνακα σε ιδιάζουσες τιμές είναι ουσιαστικά η γενίκευση της ιδιοπαραγοντοποίησης για οποιονδήποτε $m \times n$ πίνακα και όχι μόνο για τους τετραγωνικούς.

$$\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{7.2}$$

Όπου αν ο $\mathbf{M}$ είναι πραγματικός σημαίνει ότι $\mathbf{V}^* \equiv \mathbf{V}^T$:

- $\mathbf{U}$ είναι $m \times m$, μοναδιαίος (για πραγματικό $\mathbf{M}$ ορθογώνιος) και αποτελείται από τα αριστερά ιδιοδιανύσματα του $\mathbf{M}$
- $\mathbf{V}$ είναι $n \times n$, μοναδιαίος (για πραγματικό $\mathbf{M}$ ορθογώνιος) και αποτελείται από τα δεξιά ορθογώνια ιδιοδιανύσματα του $\mathbf{M}$
- Σ είναι $m \times n$, είναι διαγώνιος με μη αρνητικά στοιχεία

Πάμε να υπολογίσουμε την παραγοντοποίηση με χρήση της της numpy:

```python
import numpy as np
import numpy.linalg as lin

# define matrix
M = np.array([[3,2,2],[2,3,-2]])
m , n = M.shape

# use automated svd
U, s, V = lin.svd(M)

# create sigma matrix
Sigma = np.zeros((m,n))
for i in range(np.min([m,n])):
    Sigma[i,i] = s[i]

# print components - V is already transposed
print(U)
print(Sigma)
print(V)

# check result
print(U@Sigma@V)
```

```
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
[[5. 0. 0.]
 [0. 3. 0.]]
[[ 7.07106781e-01  7.07106781e-01  3.67439059e-16]
 [-2.35702260e-01  2.35702260e-01 -9.42809042e-01]
 [-6.66666667e-01  6.66666667e-01  3.33333333e-01]]
[[ 3.  2.  2.]
 [ 2.  3. -2.]]
```

Στον αναλυτικό υπολογισμό θα έπρεπε να βρούμε την ιδιοπαραγοντοποίηση των $\mathbf{M}\mathbf{M}^T$ και $\mathbf{M}^T\mathbf{M}$, ώστε να βρούμε τον $\mathbf{U}$ και τον $\mathbf{V}$ αντίστοιχα.

7.2.1 Εφαρμογές - Συμπίεση εικόνας

Μπορούμε να χρησιμοποιήσουμε την παραγοντοποίηση σε ιδιάζουσες τιμές για να συμπιέσουμε εικόνες. Η τεχνική ονομάζεται ελάττωση διάστασης και αφορά το ανάπτυγμα ενός πίνακα σε άθροισμα πινάκων τάξης 1. Η τάξη ενός πίνακα είναι ο μέγιστος αριθμός των γραμμικά ανεξάρτητων διανυσμάτων στήλης ή γραμμής. Σε ένα πίνακα όπου μία

γραμμή είναι ο πολλαπλασιασμός μιας άλλης με κάποιον αριθμό μιλάμε για εξαρτημένα διανύσματα. Όταν λοιπόν ανάγουμε ένα πίνακα(μία εικόνα) στην ιδιάζουσα παραγοντοποίηση, οι πίνακες $\mathbf{U}$ και $\mathbf{V}$ είναι ορθογώνιοι, επομένως, τα διανύσματα που τους αποτελούν είναι κάθετα μεταξύ τους. Συνεπώς η τάξη είναι όση και η διάστασή τους. Αναλύουμε λοιπόν τον πίνακα Σ:

$$\mathbf{M} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^* \tag{7.3}$$

όπου r είναι η τάξη του $\mathbf{M}$. Αν κρατήσουμε μόνο τις μεγαλύτερες ιδιοτιμές σ_i σημαίνει ότι έχουμε μία προσέγγιση του $\mathbf{M}$ σε άθροισμα πινάκων τάξης 1. Αυτή η προσέγγιση είναι που ονομάζουμε συμπίεση, αν ο πίνακας $\mathbf{M}$ έχει τιμές χρωμάτων. Ας κάνουμε ένα απλό παράδειγμα:

```python
import numpy as np
import numpy.linalg as lin

# define matrix
M = np.array([[1.8,1.2],[4.4,4.6]])
m,n = M.shape

# use automated svd
U, s, V = lin.svd(M)

# create sigma matrix
Sigma = np.zeros((m,n))
for i in range(np.min([m,n])):
    Sigma[i,i] = s[i]

# print components - V is already transposed
print(f"U = {U}\n")
print(f"Sigma = {Sigma}\n")
print(f"V = {V}")
```

```
U = [[-0.31622777 -0.9486833 ]
 [-0.9486833   0.31622777]]

Sigma = [[6.70820393 0.        ]
 [0.         0.4472136 ]]

V = [[-0.31622777 -0.9486833 ]
 [-0.9486833   0.31622777]]
```

```
# choose just the first approximation
# to do that we just : sigma[1,1]=0
Sigma[1,1]=0

M_approximate = U@Sigma@V
# M_approximate has rank one thus we don't have to store all the
                                        elements of M
# we just need to store [[1],[3]] and [1.5 1.5] then matmulthem
print(M)
```

```
[[1.8 1.2]
 [4.4 4.6]]
```

7.3 Cholesky

Η παραγοντοποίηση Cholesky είναι η παραγοντοποίηση ερμιτιανού($\mathbf{A} = \mathbf{A}^H$) ημιθετικού(μη αρνητικές ιδιοτιμές) πίνακα, σε έναν κάτω τριγωνικό πίνακα επί τον ερμιτιανό συζυγή του:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^* \tag{7.4}$$

όπου στην περίπτωση που ο $\mathbf{A}$ είναι πραγματικός:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T \tag{7.5}$$

Υπάρχουν αναλυτικοί έτοιμοι τύποι για τον υπολογισμό του $\mathbf{L}$. Στη numpy:

```
# cholesky
import numpy.linalg as lin

# hermitian positive semi definite matrix
M = np.array([[3,2,2],[2,3,2],[2,2,2]])
L = lin.cholesky(M)
print(L)

# check
print(L@L.T)
```

```
[[1.73205081 0.         0.        ]
 [1.15470054 1.29099445 0.        ]
 [1.15470054 0.51639778 0.63245553]]
[[3. 2. 2.]
 [2. 3. 2.]
 [2. 2. 2.]]
```

Κεφάλαιο 8

Μέθοδος Ελαχίστων Τετραγώνων

Η μέθοδος ελαχίστων τετραγώνων για ευθεία γραμμή είναι μέθοδος εύρεσης την ευθείας με την ελάχιστη τετραγωνική απόσταση από τα δεδομένα. Αν έχουμε ένα σύνολο δεδομένων $(x_i, y_i), i - 0, 1, ...n$ όπου τα x_i θεωρούνται η ανεξάρτητη μεταβλητή. Ο σκοπός μας είναι να υπολογίσουμε τις παραμέτρους της ευθείας $y = mx + c$ έτσι ώστε να το άθροισμα των τετραγωνικών αποστάσεων των δεδομένων από την ευθεία να είναι ελαχιστοποιημένο. Η απόσταση ενός σημείου από την ευθεία είναι:

$$r_i = y_i - mx_i - c \tag{8.1}$$

Αυτή η συνάρτηση δίνει την απόσταση της τιμής των δεδομένων από την ευθεία πρόβλεψης. Συνεπώς η ελαχιστοποίηση πρέπει να γίνει στην:

$$S = \sum_{i=1}^{n} r_i^2 \tag{8.2}$$

Πως χρησιμοποιούμε την numpy για την ευθεία ελαχίστων τετραγώνων:

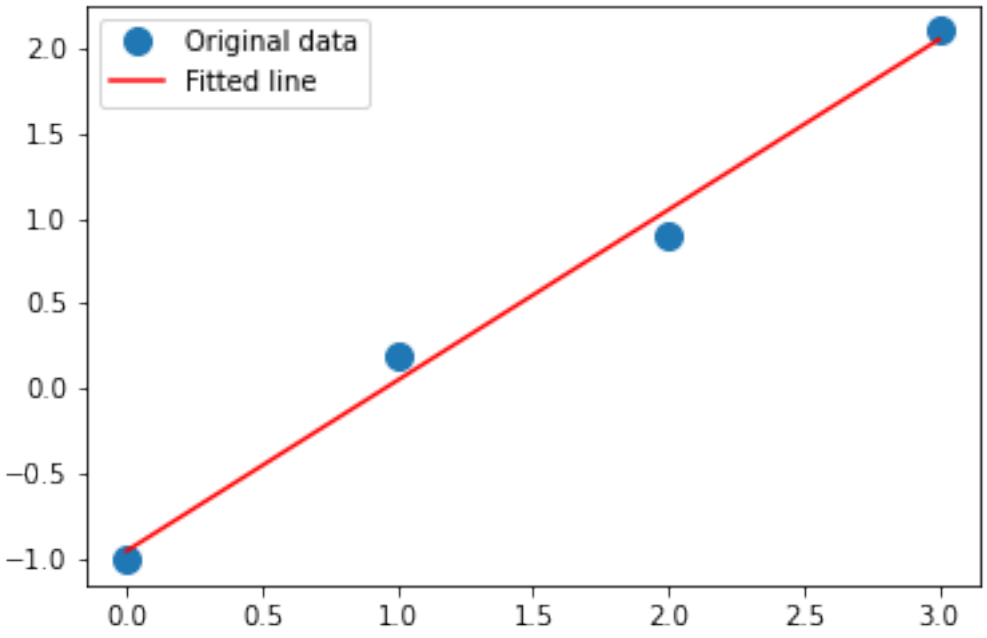
```
import numpy.linalg as lin
import numpy as np

# data
x = np.array([0, 1, 2, 3])
y = np.array([-1, 0.2, 0.9, 2.1])

# we rewrite the equation as y=Ap where A=[[x 1]],p=[[m],[c]]
A = np.vstack([x, np.ones(len(x))]).T

# we put it in the function
m, c = lin.lstsq(A, y, rcond=None)[0]
m = round(m,3)
```

```
c = round(c,3)
m,c
```

```
(1.0, -0.95)
```

Ας σχεδιάσουμε για να πάρουμε μία ιδέα για την ευθεία του παραδείγματος:

```
import matplotlib.pyplot as plt
_ = plt.plot(x, y, 'o', label='Original data', markersize=10)
_ = plt.plot(x, m*x + c, 'r', label='Fitted line')
_ = plt.legend()
plt.show()
```



Bibliography

[1] J. Brownlee. *Statistical methods for machine learning: Discover how to transform data into knowledge with Python*. Machine Learning Mastery, 2018.

[2] M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[3] P. Klein. *Coding the Matrix: Linear Algebra Through Applications to Computer Science*. Newtonian Press, 2013.

[4] H. Son. *Linear Algebra Coding with Python*. Hyun-Seok Son, 2020.