MATRIX EXAMPLES REPORT

School or Department: Wu Han University

Grade and Specialty: Excellent engineer 2016

Name: Guo Yang, Kong Chuishun, Tang Rui

Advisor: Fangling Pu

May 2 2018

【Abstract】In this lab, we describe some special matrices that occur often in applications. And we use a python program to familiar it. In the application, we detect edge of a camera car in a rgb picture by convolution.

【**Key words**】 Matrices ;Python; convolution

Table of Contents

Chapter 1 Matrix examples

1.1 Geometric transformations

Suppose the 2-vector (or 3-vector) x represents a position in 2-D (or 3-D) space. Several important geometric transformations or mappings from points to points can be expressed as matrix-vector products $\boldsymbol{y} = A\boldsymbol{x}$, with A a 2 × 2 (or 3 × 3) matrix. In the examples below, we consider the mapping from x to y, and focus on the 2-D case (for which some of the matrices are simpler to describe).

The properties include Scaling, Dilation, Rotation, Reflection and so on. We don't repeat them.

1.2 Selectors

An m×n selector matrix A is one in which each row is a unit vector (transposed):

$$A = \begin{bmatrix} e_{k_1}^T \\ \vdots \\ e_{k_m}^T \end{bmatrix}$$

where k1,...,km are integers in the range 1,...,n. When it multiplies a vector, it simply copies the k_ith entry of x into the ith entry of y = Ax:

$$y = (x_{k1}, x_{k2}, \ldots, x_{km}).$$

In words, each entry of Ax is a selection of an entry of x. The identity matrix, and the reverser matrix

$$A = \begin{bmatrix} e_n^T \\ \vdots \\ e_1^T \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & \iddots & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

are special cases of selector matrices. (The reverser matrix reverses the order of the entries of a vector: Ax = (xn, xn−1, ... , x2,x1).) Another one is the r:s slicing matrix, which can be described as the block matrix

$$A = [\, 0_{m\times(r-1)} \quad I_{m\times m} \quad 0_{m\times(n-s)} \,]$$

where m = s−r +1. (We show the dimensions of the blocks for clarity.) We have Ax = $x_{r:s}$, i.e., multiplying by A gives the r:s slice of a vector.

Down-sampling. Another example is the (n/2)×n matrix (with n even)

If y = Ax, we have y = (x_1,x_3,x_5,...,x_{n-3},x_{n-1}). When x is a time series, y is called the 2× down-sampled version of x. If x is a quantity sampled every hour, then y is the same quantity, sampled every 2 hours.

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \end{bmatrix}$$

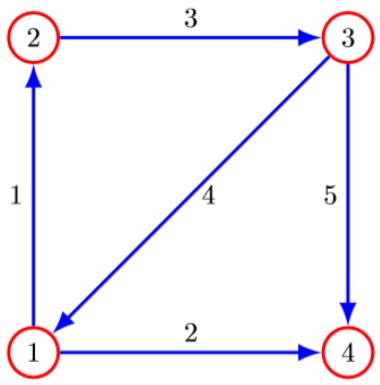

Figure 1.1 Directed graph with four vertices and five edges.

1.3 Incidence matrix

Directed graph. A directed graph consists of a set of vertices (or nodes), labeled 1,...,n, and a set of directed edges (or branches), labeled 1,...,m. Each edge is connected from one of the nodes and into another one, in which case we say the two nodes are connected or adjacent. Directed graphs are often drawn with the vertices as circles or dots, and the edges as arrows, as in figure 1.1 . A directed graph can be described by its n×m incidence matrix, defined as

$$A_{ij} = \begin{cases} 1 & \text{edge } j \text{ points to node } i \\ -1 & \text{edge } j \text{ points from node } i \\ 0 & \text{otherwise.} \end{cases}$$

1.4 Networks

In many applications a graph is used to represent a network, through which some commodity or quantity such as electricity, water, heat, or vehicular traffic flows. The edges of the graph represent the paths or links over which the quantity can move or flow, in either direction. If x is an m-vector representing a flow in the network, we interpret xj as the flow (rate) along the edge j, with a positive value meaning the flow is in the direction of edge j, and negative meaning the flow is in the opposite direction of edge j. In a network, the direction of the edge or link does not specify the direction of flow; it only specifies which direction of flow we consider to be positive.

Sources. In many applications it is useful to include additional flows called source flows or exogenous flows, that enter or leave the network at the nodes, but not along the edges, as shown in figure 1.2 . We denote these flows with an n-vector s. We can

think of si as a flow that enters the network at node i from outside, i.e., not from any edge. When si > 0 the exogenous flow is called a source, since it is injecting the quantity into the network at the node. When si < 0 the exogenous flow is called a sink, since it is removing the quantity from the network at the node.

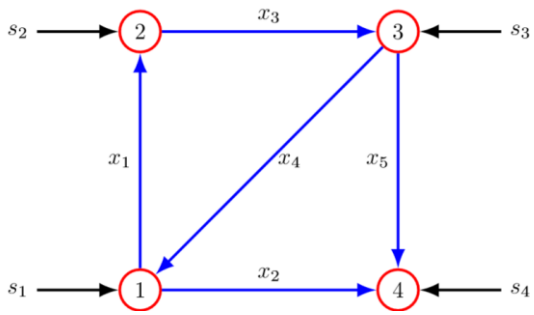

Figure 1.2 Network with four nodes and five edges, with source flows shown.

Dirichlet energy. When the m-vector ATv is small, it means that the potential differences across the edges are small. Another way to say this is that the potentials of connected vertices are near each other. A quantitative measure of this is the function of v given by

$$\mathcal{D}(v) = \|A^T v\|^2.$$

This function arises in many applications, and is called the Dirichlet energy (or Laplacian quadratic form) associated with the graph. It can be expressed as

$$\mathcal{D}(v) = \sum_{\text{edges } (k,l)} (v_l - v_k)^2,$$

which is the sum of the squares of the potential differences of v across all edges in the graph. The Dirichlet energy is small when the potential differences across the edges of the graph are small, i.e., nodes that are connected by edges have similar potential values.

1.5 Convolution

The convolution of an n-vector a and an m-vector b is the (n + m − 1)-vector denoted c = a∗b, with entries

$$c_k = \sum_{i+j=k+1} a_i b_j, \quad k = 1, \ldots, n+m-1,$$

where the subscript in the sum means that we should sum over all values of i and j in their index ranges 1,...,n and 1,...,m, for which the sum i + j is k + 1. For example, with n = 4, m = 3, we have

$$\begin{aligned} c_1 &= a_1 b_1 \\ c_2 &= a_1 b_2 + a_2 b_1 \\ c_3 &= a_1 b_3 + a_2 b_2 + a_3 b_1 \\ c_4 &= a_2 b_3 + a_3 b_2 + a_4 b_1 \\ c_5 &= a_3 b_3 + a_4 b_2 \\ c_6 &= a_4 b_3 . \end{aligned}$$

Convolution reduces to ordinary multiplication of numbers when n = m = 1, and to scalar-vector multiplication when either n = 1 or m = 1. Convolution arises in many applications and contexts.

Polynomial multiplication. If a and b represent the coefficients of two polynomials

$$p(x) = a_1 + a_2 x + \cdots + a_n x^{n-1},$$

$$q(x) = b_1 + b_2 x + \cdots + b_m x^{m-1},$$

then the coefficients of the product polynomial p(x)q(x) are represented by c = a∗b:

$$p(x)q(x) = c_1 + c_2 x + \cdots + c_{n+m-1} x^{n+m-2}$$

To see this we will show that ck is the coefficient of x^{k-1} in p(x)q(x). We expand the product polynomial into mn terms, and collect those terms associated with xk−1. These terms have the form aibjxi+j−2, for i and j that satisfy i + j −2 = k −1, i.e., i + j = k − 1. It follows that ck = $\sum_{i+j=k+1}$ aibj.

Properties of convolution. Convolution is symmetric: We have a∗b = b∗a. It is also associative: We have (a∗b)∗c = a∗(b∗c), so we can write both as a∗b∗c. Another property is that a ∗ b = 0 implies that either a = 0 or b = 0. These properties follow from the polynomial coefficient property above, and can also be directly shown. As an example, let us show that a∗b = b∗a. Suppose p is the polynomial with coefficients a, and q is the polynomial with coefficients b. The two polynomials p(x)q(x) and q(x)p(x) are the same (since multiplication of numbers is commutative), so they have the same

coefficients. The coefficients of p(x)q(x) are a∗b and the coefficients of q(x)p(x) are b∗a. These must be the same. A basic property is that for fixed a, the convolution a∗b is a linear functionof b; and for fixed b, it is a linear function of a. This means we can express a∗b as a matrix-vector product:

$$a*b = T(b)a = T(a)b,$$

where T(b) is the (n + m−1)×n matrix with entries

$$T(b)_{ij} = \begin{cases} b_{i-j+1} & 1 \le i-j+1 \le m \\ 0 & \text{otherwise} \end{cases}$$

and similarly for T(a).

2-D convolution Convolution has a natural extension to multiple dimensions. Suppose that A is an m×n matrix and B is a p×q matrix. Their convolution is the (m+p−1)×(n+q−1) matrix

$$C_{rs} = \sum_{i+k=r+1,\ j+l=s+1} A_{ij}B_{kl}, \quad r = 1,\ldots,m+p-1, \quad s = 1,\ldots,n+q-1,$$

where the indices are restricted to their ranges (or alternatively, we assume that A_{ij} and B_{kl} are zero, when the indices are out of range). This is not denoted C = A ∗ B, however, in standard mathematical notation. So we will use the notation C = A * B.

Image blurring. If the m×n matrix X represents an image, Y = X*B represents the effect of blurring the image by the point spread function (PSF) given by the entries of the matrix B. If we represent X and Y as vectors, we have y = T(B)x, for some (m + p−1)(n + q−1)×mn-matrix T(B).

Chapter 2 Solutions

The rgb picture experiences the following process:

1 rgb -> grey -> binary -> show edge

2 rgb -> grey -> convolution -> invert -> show edge

2.1 import libraries

```
from PIL import Image
import numpy as np
import os
from skimage import io
```

2.2 Show the original picture：

```
# load a rgb picture and show it
im_rgb = Image.open('./Camaro.jpg')
im_rgb.show()
im_rgb_array = np.array(im_rgb)

# show some details of the rgb picture
print (im_rgb_array)
print (im_rgb_array[0][0])
print (len(im_rgb_array))
print (len(im_rgb_array[0]))
print (len(im_rgb_array[0][0]))
```

2.3 Convolution

We covert the original picture to a grey level picture， and convert it to binary picture by the table. This method is named Laplace operator. It is used to detect the edge.

```
def conv (im_array, conv_x):
    im_copy = im_array.copy()
    height, width = im_copy.shape
    for i in range(0, height-2):
        for j in range(0, width-2):
            tmp = (im_array[(i):(i+3),(j):(j+3)]*conv_x).sum()
            if tmp > 255:
                tmp = 255
            elif tmp < 0:
                tmp = 0
            im_copy[i][j] = tmp
    return im_copy
```

2.4 Distinction

To make the result more obvious, deal with it further. And we import output image after convolution to make it obvious

```
# invert defined function
def invert (im_array):
    im_copy = im_array.copy()
    height, width = im_copy.shape
    for i in range(0, height-2):
        for j in range(0, width-2):
            if im_copy[i][j] > 100:
                im_copy[i][j] = 0
            else:
                im_copy[i][j] = 255
    return im_copy
```

Chapter 3 Results and Conclusion

The original picture is shown as following:

Figure 3.1

After the Grey Processing:

Figure 3.2

After the binaryzaton:

Figure 3.3

After the convolution:

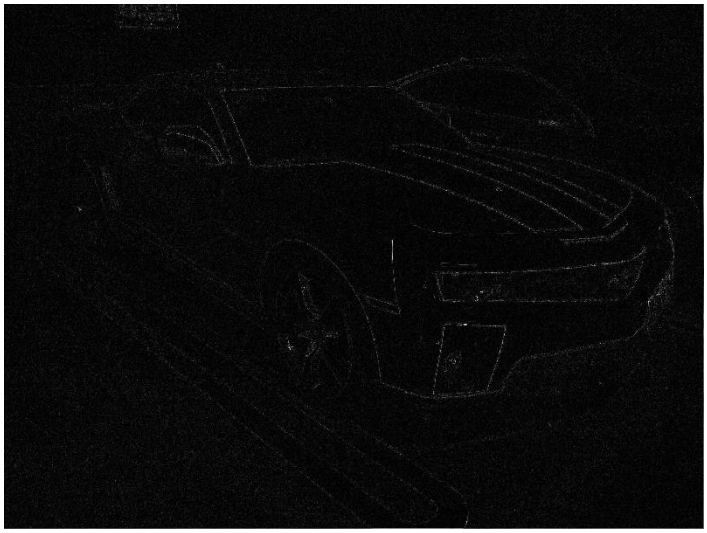

Figure 3.4

To make the result more obvious, we invert the figure 3.4 as shown in the figure 3.5

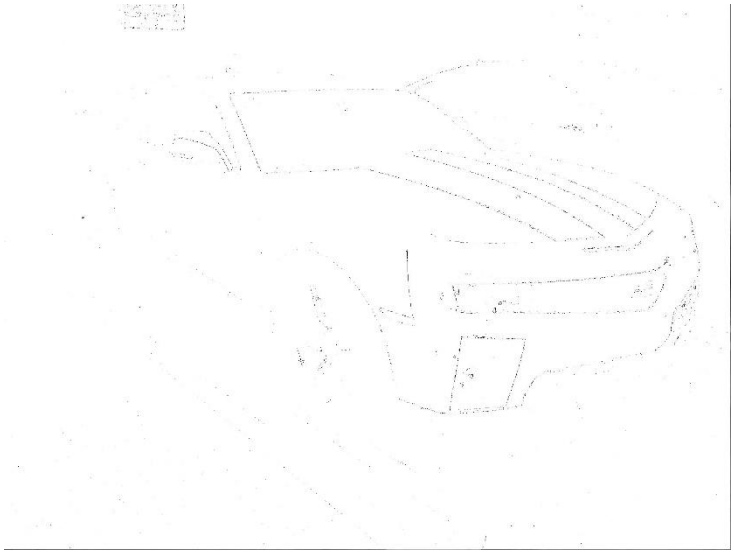

Figure3.5

And we can find that the picture behave differently and present the properties

Bibliography

[1] https://en.wikipedia.org/wiki/Kernel_(image_processing)

[2] Introduction to Applied Linear Algebra Vectors, Matrices, and Least Squares Stephen Boyd Department of Electrical Engineering Stanford University Lieven Vandenberghe Department of Electrical and Computer Engineering University of California, Los Angeles

Appendix

```
# May 1, 2018
'''
  These codes are to detect edge of a camaro car in a
  rgb picture by convolution.
  The rgb picture experiences the following process:
      1 rgb -> grey -> binary -> show edge
      2 rgb -> grey -> convolution -> invert -> show edge
  You can find different visual effects of two methods.
'''
```

```
# coding:utf-8


# import libraries
from PIL import Image
import numpy as np
import os
from skimage import io


# load a rgb picture and show it
im_rgb = Image.open('./Camaro.jpg')
im_rgb.show()
im_rgb_array = np.array(im_rgb)

# show some details of the rgb picture
print (im_rgb_array)
print (im_rgb_array[0][0])
print (len(im_rgb_array))
print (len(im_rgb_array[0]))
print (len(im_rgb_array[0][0]))

# covert to grey level picture
im_grey = Image.open('./Camaro.jpg')
im_grey = im_grey.convert('L')
try:
        im_grey.save("Camaro_Grey.jpg")
except IOError:
        print ("Cannot convert")
im_grey.show()
```

```
im_grey_array = np.array(im_grey)

# setup a converting table with constant threshold
threshold = 45
table = []
for i in range( 256 ):
        if i < threshold:
            table.append(0)
        else :
            table.append(1)

# convert to binary picture by the table
im_binary = im_grey.point(table,"1")
im_binary.save( "Camaro_Binary.jpg" )
im_binary.show()
im_binary_array = np.array(im_binary)


# convolution kernel array
# This array is named Laplace operator.
# It is used to detect the edge.
conv_input = np.array([[ 1,    1,    1],
                                    [ 1, -8,    1],
                                    [ 1,    1,    1]])


# convolution defined function
# import image array and convolution kernel array
def conv (im_array, conv_x):
      im_copy = im_array.copy()
```

```
    height, width = im_copy.shape
    for i in range(0, height-2):
        for j in range(0, width-2):
            tmp = (im_array[(i):(i+3),(j):(j+3)]*conv_x).sum()
            if tmp > 255:
                tmp = 255
            elif tmp < 0:
                tmp = 0
            im_copy[i][j] = tmp
    return im_copy

# import grey image array to function
# output edge detection of grey image.
im_conv = conv(im_grey_array, conv_input)

# show result
new_im = Image.fromarray(im_conv)
new_im.show()
new_im.save("Camaro_Conv.jpg")

# To make the result more obvious, deal with it further
new_im_arrray = np.array(new_im)

# invert defined function
def invert (im_array):
    im_copy = im_array.copy()
    height, width = im_copy.shape
    for i in range(0, height-2):
        for j in range(0, width-2):
            if im_copy[i][j] > 100:
```

```
                    im_copy[i][j] = 0
                else:
                    im_copy[i][j] = 255
    return im_copy

# import output image after convolution to make it obvious
im_invert = invert(new_im_arrray)
new_im_invert = Image.fromarray(im_invert)

# show result again
new_im_invert.show()
new_im_invert.save("Camaro_Conv_Invert.jpg")
```