Lab 3

John Munyi

11/09/2018

Table of Contents

1. Introduction

This lab focussed on 2 critical areas of matrices. In the first part, the aim was to implement a manual way to perform a matrix inverse using the step by step method which is a variation of the Gauss Jordan technique. The method takes a step by step approach operating on 2 matrices converting the target matrix to an identity matrix and changing the identity matrix to the inverse of the target matrix.

The second part of the lab focusses on using the application matrix inversion, transformation, and rotations. In this lab, the aim is to transfer a red square from a black background to another black background using rotations and transformations. This demonstrates the applications of matrix inversion in transformations and rotations.

2. Part 1: Matrix Inversion

The goal of this exercise is to understand the inner workings of matrix inversion using a variation of the Gauss Jordan technique. The method takes a step by step approach operating on 2 matrices converting the target matrix to an identity matrix and changing the identity matrix to the inverse of the target matrix.

2.1. Results

Below is a sample for the calculation of the inverse of a given matrix. The result also contains the MAE and MSE for A dot A-inverse. .

Sample result 1

The Matric to get Inverse of

[[0. 8. -3. 2.]

[4. -6. -3. 5.]

[7. 2. -9. -1.]

[8. 7. -4. 0.]]

---

inverse matrix is:

[[0.02984625 -0.09647272 -0.0437142   0.1483268]

[-0.03376545 0.06873681 -0.03135363 0.04431715]

[0.00060295 -0.07265601 -0.14229726 0.12420862]

[0.13596623 0.11606874 -0.08803135 0.00904432]]

[[4.94724148e-01   1.76846548e+00 -2.22044605e-16 -3.78956889e-01]

[4.10009044e-01 -4.35031655e-01 -9.99200722e-16   3.07506783e-01]

[-2.93035876e-01   1.02562557e+00   1.00000000e+00 -2.19776907e-01]

[-7.95899910e-02   2.78564968e-01 -5.55111512e-17   9.40307507e-01]]

The Mean Squared Error (MSE) is: 0.445121360721

The Maximum Absolute Error (MAE) is: 1.43503165511

Sample result 2

The Matric to get Inverse of

[[2. 3. 4.]

[5. 5. 6.]

[2. 4. 8.]]

---

inverse matrix is:

[[-1.33333333 0.66666667 0.16666667]

[2.33333333 -0.66666667 -0.66666667]

[-0.83333333 0.16666667 0.41666667]]

[[1.00000000e+00   7.77156117e-16   6.66133815e-16]

[-4.44089210e-16   1.00000000e+00   0.00000000e+00]

[1.11022302e-16   2.22044605e-16   1.00000000e+00]]

The Mean Squared Error (MSE) is: 2.82127337631e-31

The Maximum Absolute Error (MAE) is: 8.881784197e-16

Sample result 3

The Matric to get Inverse of

[[2. 8. -3. 2.]

[4. -6. -3. 5.]

[7. 2. -9. -1.]

[11. 4. -15. 6.]]

---

inverse matrix is:

[[0.5 0.5 0.5 -0.5]

[0.14939024 0.00914634 0.03353659 -0.05182927]

[0.41869919 0.38617886 0.30487805 -0.41056911]

[0.0304878 0.04268293 -0.17682927 0.09146341]]

[[1.00000000e+00 -1.77635684e-15 -8.88178420e-16 8.88178420e-16]

[1.11022302e-16 1.00000000e+00 -2.22044605e-16 1.11022302e-16]

[0.00000000e+00 -8.88178420e-16 1.00000000e+00 3.33066907e-16]

[0.00000000e+00 2.77555756e-17 2.22044605e-16 1.00000000e+00]]

The Mean Squared Error (MSE) is: 4.25293479969e-31

The Maximum Absolute Error (MAE) is: 1.7763568394e-15

2.2. Discussion

For most matrices of 2X2 and 3X3, the MSE is zero as the python script is able to perfectly calculate the inverse with 99.99% accuracy however when it comes to matrices have zeros in the diagonal column the MSE increase since the inverse matrix is not very perfect. Also for matrices where the determinant is equal to zero, it is impossible to get the inverse, this is taken care of by calculating the determinant before trying to do the inverse.

3.Linear Transformation on the plane

The goal of this section of the assignment is to demonstrate the power of rotations, scaling and transformation along the place of a 2D. The image is passed through different scalers and rotations to yield different positional transformations along the x and y-axis, this allows us to see how points are mapped from one scaler to the other.

3.1. Results

After the transformation have been executed on the original the following images are the resulting transformations.

The original version:

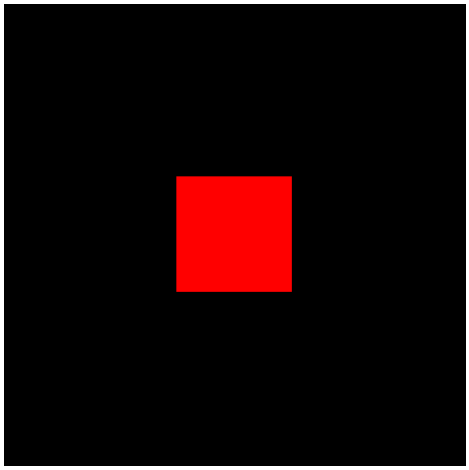

Image test 1:

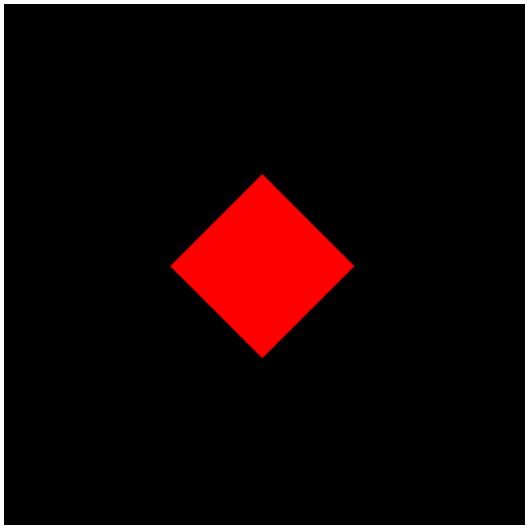

Image test 2:

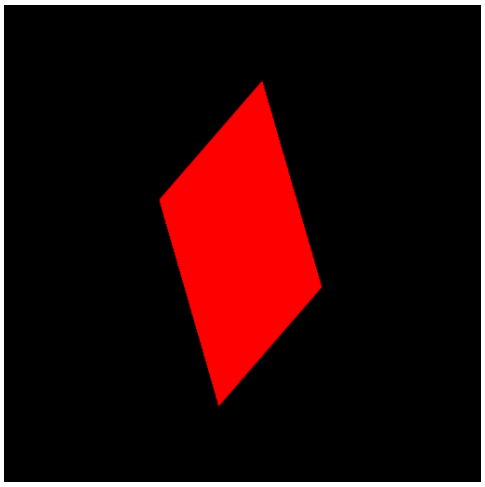

Image test 3:

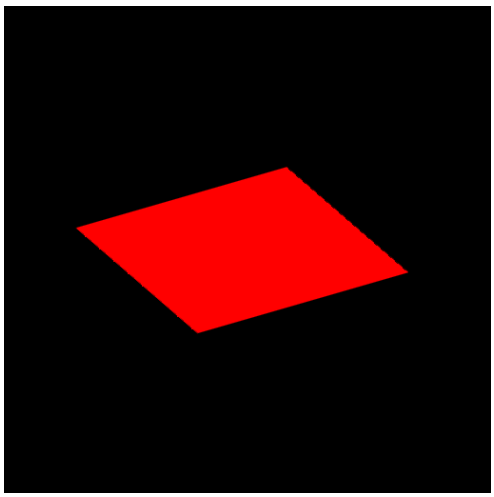

Image test 4:

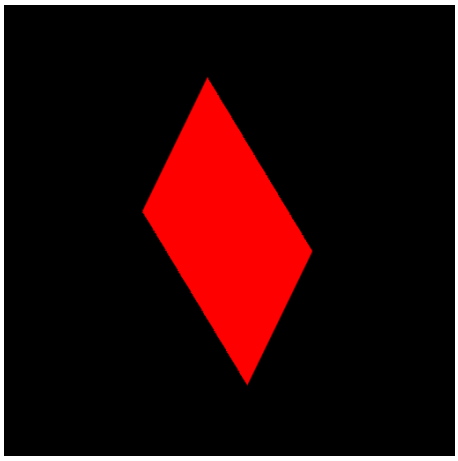

Image test 5:

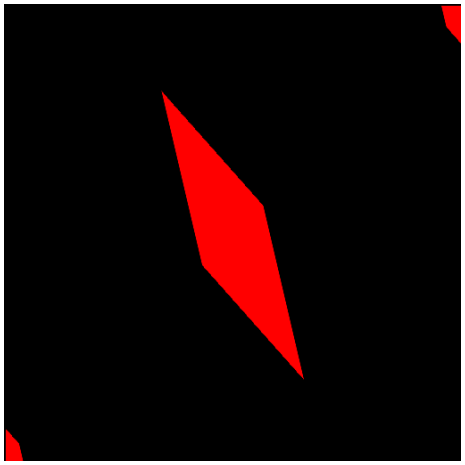

Image test 6:

3.2. Discussion

From the above results in its evident that different type of input scale or transform or rotate the image in different but specific ways. Lambda 1 and 2 are scalers while theta 1 and 2 are for rotations. The translation along x and y-axis is effected by the X and Y translations.

4. Conclusion

The labs demonstrate the application matrix transformations and rotation in mapping images in different vector spaces. The biggest take away here is behind the scenes effort that goes into mapping images onto different background in VR and AR.

5. Appendix:

5.1. Inverse.py

```
import numpy as np
import csv
from copy import deepcopy
from sklearn.metrics import mean_squared_error

def convert_inverse(file_name):

    # read matrix from csv file
    x = np.genfromtxt(file_name, delimiter=',')
    mat = np.array(x, np.float64)

    # make a copy of the matrix that we shall not modified to use to test
    # at the end to proof
    X = deepcopy(mat)

    rows = mat.shape[0]
    cols = mat.shape[1]
```

```
id_mat = np.identity(rows, np.float64)
Y = deepcopy(id_mat)

print("The Matric to get Inverse of \n", mat)

if np.linalg.det(mat) != 0:
    for j in range(cols):
        # take current row and multiply by 1/the that value that
        # makes it a 1
        if mat[j][j] != 1:
            if  mat[j][j] != 0:
                divisor = 1/mat[j][j]
            else:
                tmp = np.copy(mat[j])
                tmp2 = np.copy(mat[j+1])
                mat[j] = tmp2
                mat[j+1] = tmp
                divisor = 1/mat[j][j]

            id_mat[j] = np.multiply(id_mat[j], divisor)
            mat[j] = np.multiply(mat[j], divisor)

            # check that the current column doesnt have a 1 in the diagnol position
            # otherwise do the row operation
            for i in range(rows):
                if j != i:
                    # negate the number which is on the same column
                    neg_mat = np.multiply(-1, mat[i][j])

                    # multiply the -ve scalar by the specific number we want to change to a 0
                    add_mat_scalar = neg_mat * mat[j, :]
                    add_idmat_scalar = neg_mat * id_mat[j, :]

                    # add the result above the number we want to chnage to a 0
```

```
                mat[i, :] = np.add(add_mat_scalar, mat[i, :])
                id_mat[i, :] = np.add(add_idmat_scalar, id_mat[i, :])
    else:
        print("The matrix is not Invertible")

    print("\n")
    print("-------------------------------------------------------")
    print("\n")
    print("inverse matrix is: \n", id_mat)
    print("\n")
    # print("The proof: \n",)
    # print("-------------------------------------------------------")

    # To verify the inverse, I use the numpy multiplication
    result = np.matmul(id_mat, X)
    print(result)

    # Computing the Maximum Absolute Error  and Mean Squared Error
    print("\n")
    print("-------------------------------------------------------")
    print("The Mean Squared Error (MSE) is: ",
          mean_squared_error(result, Y))
    print("The Maximum Absolute Error (MAE) is: ", np.max(np.subtract(Y, result)))


convert_inverse("matrix.csv")
```

5.2.Transformation.py

```
import numpy as np
import bmp_io_c
from numpy.linalg import inv
import math
```

```
# define image size
rows = 512
cols = 512
planes = 3

# create a black image
black_image = np.zeros([3, rows, cols], np.uint8)
bmp_io_c.output_bmp_c("my_image.bmp", black_image)



# insert a red square
def insert_red_sq(image):
    for i in range(255 - 64, 255 + 64):
        for j in range(255 - 64, 255 + 64):
            image[0, i, j] = 255

    return image



image_with_red = insert_red_sq(black_image)
bmp_io_c.output_bmp_c("my_image_red.bmp", image_with_red)

black_image_original = np.zeros([3, rows, cols], np.uint8)

# convert from PICS to CICS and vice versa by + 255 and -255
def change_to_CISC_PICS(image, type):
    pic = np.zeros([3, 512, 512], np.uint8)
    if type == "to_cics":
        for i in range(rows):
            for j in range(cols):
                pic[0, i, j] = image[0, i - 255, j - 255]
    else:
        for i in range(rows):
            for j in range(cols):
```

```
                pic[0, i-255, j-255] = image[0, i, j]

    return pic


black_cics = change_to_CISC_PICS(image_with_red, "to_cics")
bmp_io_c.output_bmp_c("PICS_image.bmp", black_cics)


# bi-linear interpolation
def lininterp(x, y, black_cics):
    x1 = math.floor(x)
    x2 = math.ceil(x)
    y1 = math.floor(y)
    y2 = math.ceil(y)

    my_mat = np.array(
        [[1, x1, y1, (x1*y1)], [1, x1, y2, (x1*y2)],
         [1, x2, y1, (x2*y1)], [1, x2, y2, (x2*y2)]])

    multi_mat = np.array([[1, x, y, (x*y)]]).reshape(4, 1)

    invmymat = np.linalg.inv(my_mat).T
    final_coord = np.dot(invmymat, multi_mat)

    if x1 > 511:
        x1 = 511

    if x2 > 511:
        x2 = 511

    if y1 > 511:
        y1 = 511
```

```
    if y2 > 511:
        y2 = 511

    final_value = final_coord[0] * black_cics[0, x1, y2] + final_coord[1] * black_cics[0, x1,
                                                                y2] + final_coord[2] * black_cics[0, x2, y1] +
final_coord[3] * black_cics[0, x2, y2]

    return final_value



# fill in the formula and plug in the 6 values and generate the 6 images
def transformer_formula(black_cics, black_image_original, theta1, theta2, lambda1, lambda2, tx, ty):

    theta1 = np.deg2rad(theta1)
    theta2 = np.deg2rad(theta2)

    S = np.array([[lambda1, 0], [0, lambda2]])

    R1 = np.array([[np.cos(theta1), np.sin(-theta1)],
                   [np.sin(theta1), np.cos(theta1)]])

    R2 = np.array([[np.cos(theta2), np.sin(-theta2)],
                   [np.sin(theta2), np.cos(theta2)]])

    A = np.dot(np.dot(R2, S), R1)
    # print("this is A", A)

    ainv = np.linalg.inv(A)
    tytx = np.array([[tx],[ty]])

    for i in range(-255, 255):
        for j in range(-255, 255):
            imageArray = np.array([[i], [j]])
```

```
            final_points = np.dot(ainv, (imageArray - tytx))
            # print("image array", imageArray)
            # print("tytx", tytx)
            # print(final_points)

            if math.ceil(final_points[0]) != math.floor(final_points[0]) and \
                math.ceil(final_points[1]) != math.floor(final_points[1]):
               black_image_original[0, i, j] = lininterp(
                  final_points[0], final_points[1], black_cics)
            else:
               black_image_original[0, i, j] = black_cics[0,
                                       final_points[0].astype(int), final_points[1].astype(int)]

    return black_image_original


# transforming the image with given values
img1 = transformer_formula(black_cics, black_image_original, 45, 0, 1, 1, 0, 0)
bmp_io_c.output_bmp_c("image_test1.bmp", change_to_CISC_PICS(img1, "to_pics"))

img2 = transformer_formula(black_cics, black_image_original, 30, 0, 2, 1, 0, 0)
bmp_io_c.output_bmp_c("image_test2.bmp", change_to_CISC_PICS(img2, "to_pics"))

img3 = transformer_formula(black_cics, black_image_original,30,0,1,2,0,0)
bmp_io_c.output_bmp_c("image_test3.bmp", change_to_CISC_PICS(img3, "to_pics"))

img4 = transformer_formula(black_cics, black_image_original, 30,15,2,1, 0, 0)
bmp_io_c.output_bmp_c("image_test4.bmp", change_to_CISC_PICS(img4, "to_pics"))

img5 = transformer_formula(black_cics, black_image_original, 50, 25, 2, 0.5, 0, 0)
bmp_io_c.output_bmp_c("image_test5.bmp", change_to_CISC_PICS(img5, "to_pics"))

img6 = transformer_formula(black_cics, black_image_original, 50, 25, 2, 0.5, 80, -70)
bmp_io_c.output_bmp_c("image_test6.bmp", change_to_CISC_PICS(img5, "to_pics")
```