기하학 처리
Geometric Image Processing

담당교수: 김민기

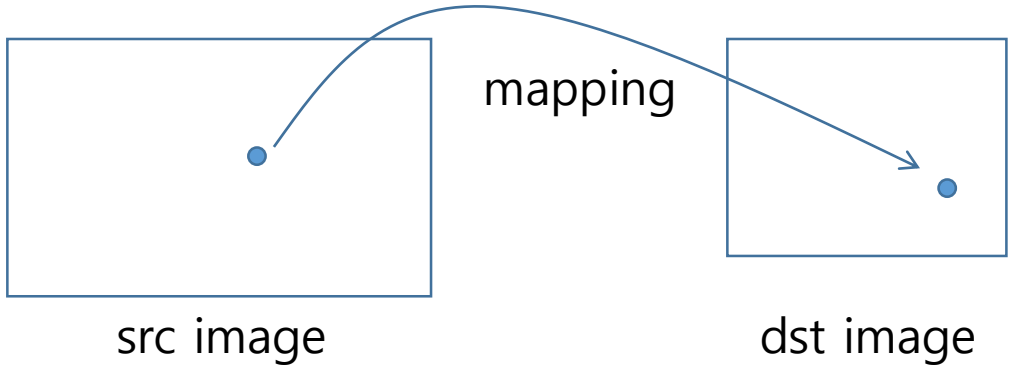
Contents

- ❖사상(mapping)
- ❖크기 변경(확대/축소)과 보간(scaling and interpolation)
- ❖평행이동(translation)
- ❖회전(rotation)
- ❖행렬 연산을 통한 기하학 변환
  - 어파인(affine) 변환
  - 원근(perspective) 변환
- ❖기타 변환

사상(mapping)

❖기하학적 처리의 기본

- 화소들의 공간적 배치 변경



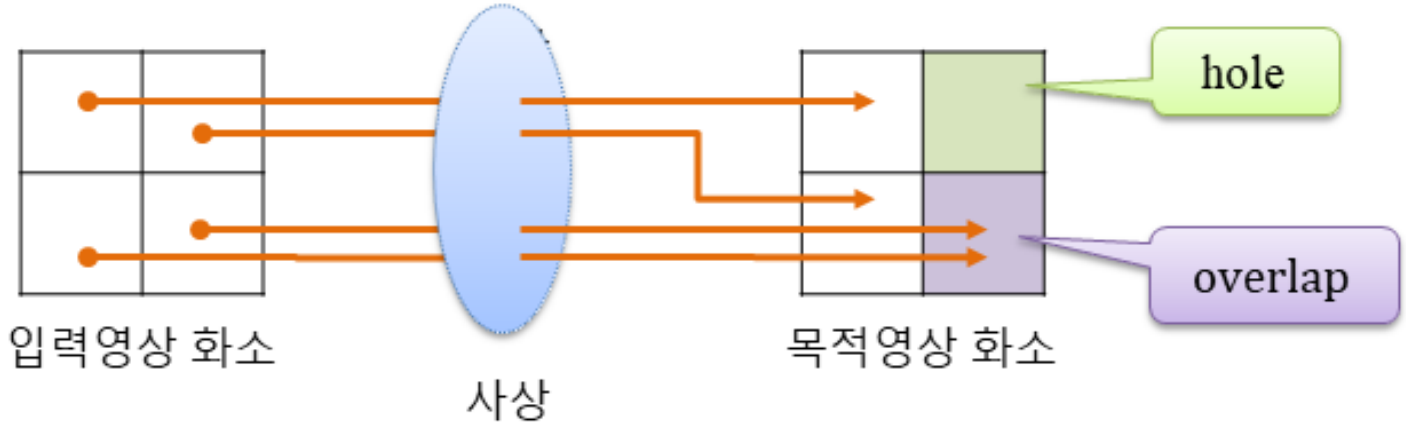
❖사상(mapping)

- 화소들의 배치를 변경할 때, 입력 영상의 좌표에 해당하는 목적 영상의 좌표를 찾아서 화소 값을 옮기는 과정

사상(mapping)

❖순방향 사상

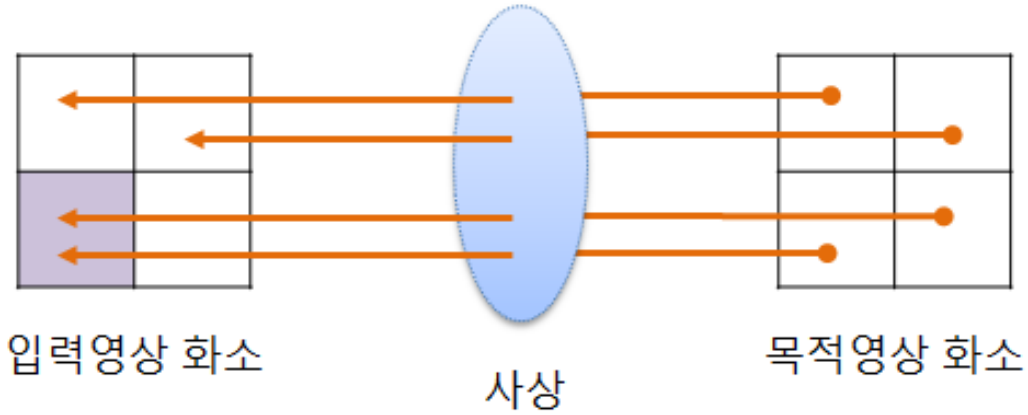
- 원본영상의 좌표를 중심으로 목적영상의 좌표를 계산하여 화소의 위치를 변환하는 방식
- 홀(hole)이나 오버랩(overlap)의 문제가 발생



사상(mapping)

❖ 역방향 사상

- 목적영상의 좌표를 중심으로 역변환을 계산하여 해당하는 입력 영상의 좌표를 찾아서 화소값을 가져오는 방식
- 홀이나 오버랩은 발생하지 않음
- 입력영상의 한 화소를 목적영상의 여러 화소에서 사용하게 되면 결과 영상의 품질 저하



확대할 때: 순방향 사상 vs. 역방향 사상

	x=0,	x=1,	x=2,	x=3
y=0,	0	40	80	120
y=1,	20	60	100	140
y=2,	40	80	120	160
y=3	60	100	140	180

$f(x) = 2x$

$f(y) = y$

x=0,	x=1,	x=2,	x=3
0		80	

$f'(x) = x/2$

$f'(y) = y$

x=0,	x=1,	x=2,	x=3
0	0	40	40

축소할 때: 순방향 사상 vs. 역방향 사상

	x=0,	x=1,	x=2,	x=3
y=0,	0	40	80	120
y=1,	20	60	100	140
y=2,	40	80	120	160
y=3	60	100	140	180

$f(x) = x/2$

$f(y) = y$

x=0,	x=1,	x=2,	x=3
40	120		

$f'(x) = 2x$

$f'(y) = y$

x=0,	x=1,	x=2,	x=3
0	80		

크기 변경 (확대/축소)

❖크기 변경(scaling)

- 입력영상의 가로와 세로로 크기를 변경해서 목적영상을 만드는 방법
- 입력영상보다 변경하고자 하는 영상의 크기가 커지면 확대, 작아지면 축소

❖크기 변경 수식

$$ratio\ X = \frac{dst_{width}}{org_{width}}, \quad ratio\ Y = \frac{dst_{height}}{org_{height}}$$

$$x' = x \cdot ratio\ X$$
$$y' = y \cdot ratio\ Y$$

```python
def time_check(func, image, size, title):                    # 수행시간 체크 함수
    start_time = time.perf_counter()
    ret_img = func(image, size)
    elapsed = (time.perf_counter() - start_time) * 1000
    print(title, "수행시간 = %0.2f ms" % elapsed)
    return ret_img

image = cv2.imread("images/scaling.jpg', cv2.IMREAD_GRAYSCALE)
if image is None: raise Exception("영상파일 읽기 에러")

dst1 = scaling(image, (150, 200))                            # 크기 변경– 축소
dst2 = scaling2(image, (150, 200))                           # 크기 변경– 축소
dst3 = time_check(scaling, image, (300,400), "[방법1]: 정방행렬 방식>")      # 확대
dst4 = time_check(scaling2, image, (300,400), "[방법2]: 반복문 방식>")      # 확대

cv2.imshow("image", image)
```

예제 8.2.1 영상 크기 변경 – 01.scaling.cpp

```
01  import numpy as np, cv2
02
03  def scaling(img, size):                                  # 크기 변경 함수1
04      dst = np.zeros(size[::-1], img.dtype)                # size와 shape는 원소 역순
05      ratioY, ratioX = np.divide(size[::-1], img.shape[:2])          # 비율 계산
06      y = np.arange(0, img.shape[0], 1)                    # 입력 영상 세로(y) 좌표 생성
07      x = np.arange(0, img.shape[1], 1)                    # 입력 영상 가로(x) 좌표 생성
08      y, x = np.meshgrid(y, x)                             # i, j 좌표에 대한 정방행렬 생성
09      i, j = np.int32(y * ratioY), np.int32(x * ratioX)    # 목적 영상 좌표
10      dst[i, j] = img[y, x]                                # 정방향 사상→목적 영상 좌표 계산
11      return dst
12
13  def scaling2(img, size):                                 # 크기 변경 함수2
14      dst = np.zeros(size[::-1], img.dtype)
15      ratioY, ratioX = np.divide(size[::-1], img.shape[:2])
16      for y in range(img.shape[0]):                        # 입력 영상 순회– 순방향 사상
17          for x in range(img.shape[1]):
18              i, j = int(y * ratioY), int(x * ratioX)      # 목적 영상의 y, x 좌표
19              dst[i, j] = img[y, x]
20      return dst
21
```

OpenCV : resize() 함수

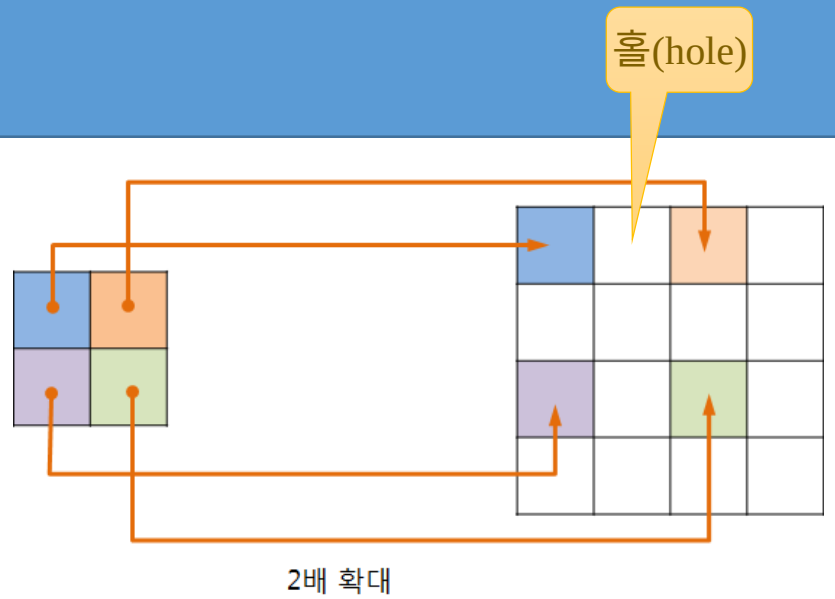
```
Resize(
    InputArray src,        // input image
    Size   dsize,
    double fx=0, double fy=0,    // dsize=0일 때, scale factor
    int interpolation=INTER_LINEAR)
```

〈표 8.3.1〉 보간 방법에 대한 flag 옵션

옵션 상수	값	설명
INTER_NEAREST	0	최근접 이웃 보간
INTER_LINEAR	1	양선형 보간 (기본값) 2x2 이웃화소 이용
INTER_CUBIC	2	바이큐빅 보간 – 4x4 이웃 화소 이용
INTER_AREA	3	픽셀 영역의 관계로 리샘플링
INTER_LANCZOS4	4	Lanczos 보간 – 8x8 이웃 화소 이용

보간(interpolation)

❖순방향 사상으로 영상 확대
→ 홀이 많이 발생



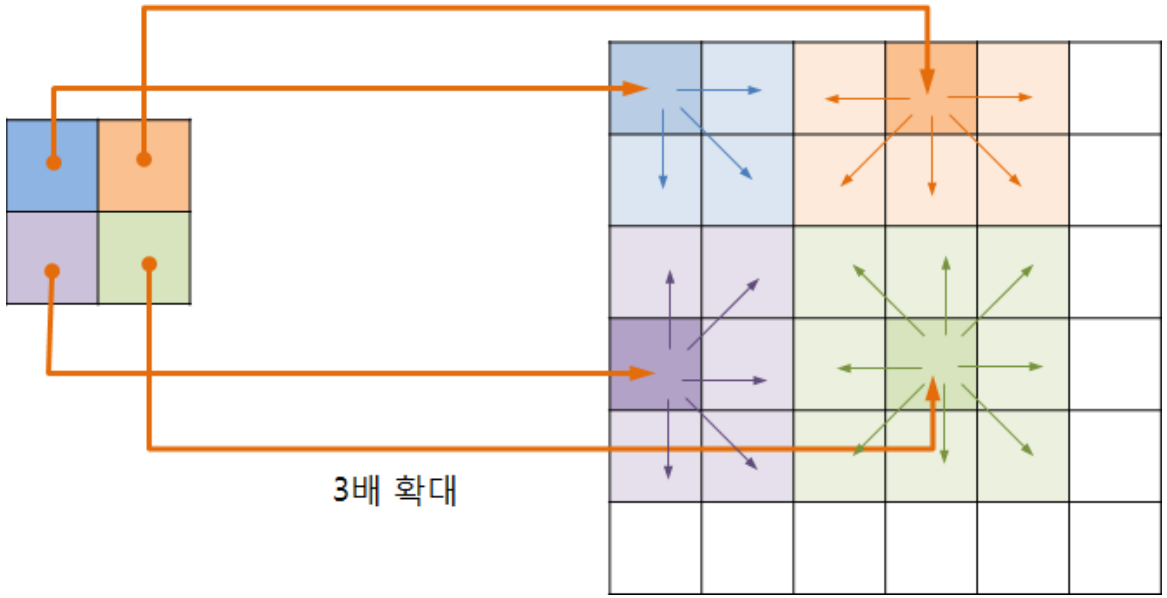
❖역방향 사상을 통해서 홀의 화소들을 입력영상에서 찾음
- 영상을 축소할 때에는 오버랩의 문제가 발생

❖보간법이 필요
- 목적영상에서 홀의 화소들을 채우고, 오버랩이 되지 않게 화소들을 배치하여 목적영상을 만드는 기법

최근접 이웃 보간법

❖목적영상을 만드는 과정에서 홀이 발생할 때,

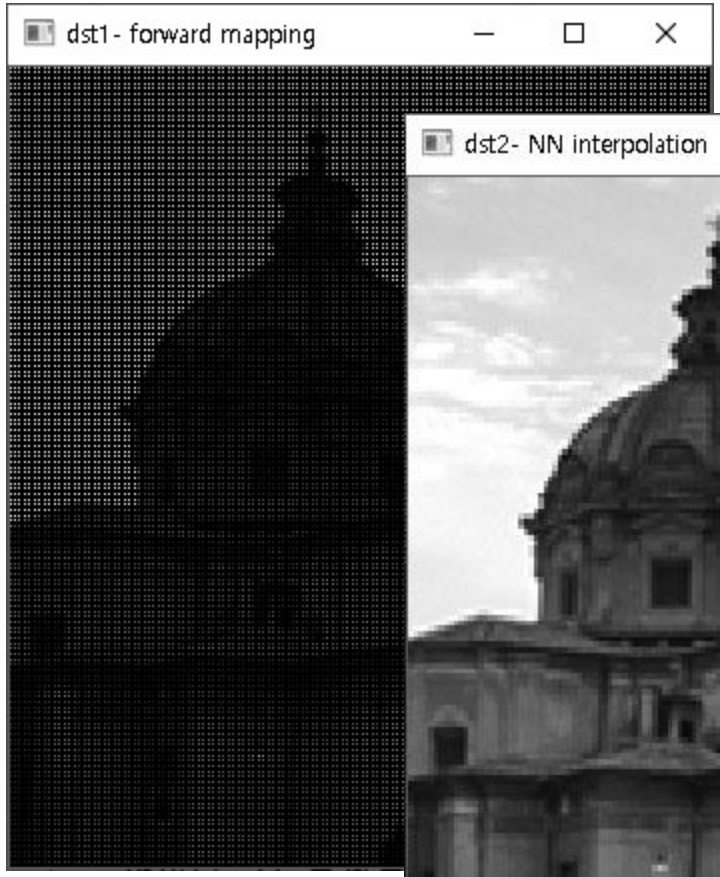
- 목적영상의 화소에 가장 가깝게 이웃한 입력영상의 화소값을 가져오는 방법

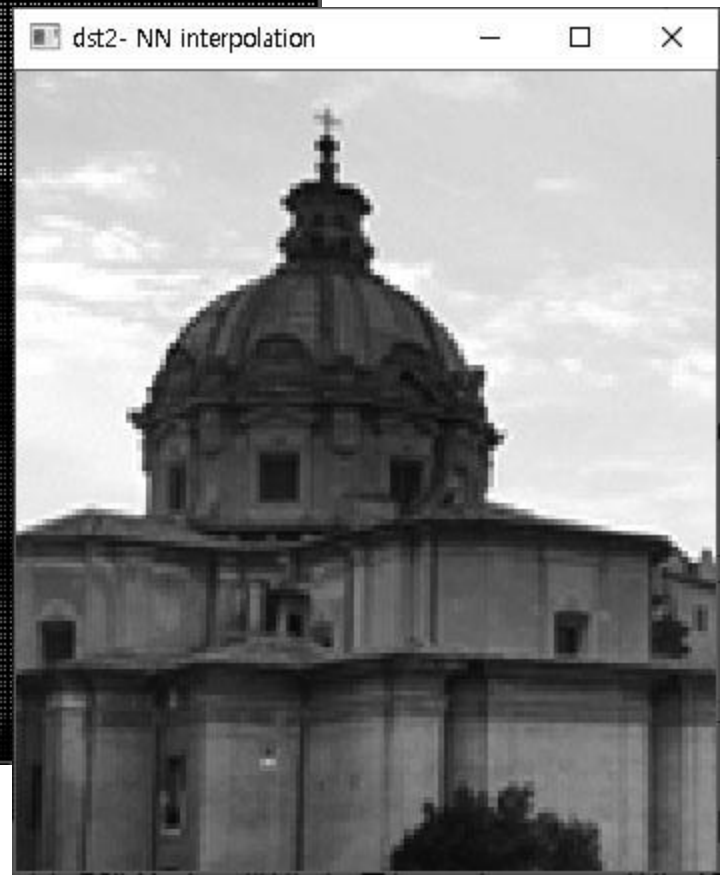


$$y' = y \cdot ratio\ Y \quad x' = x \cdot ratio\ X \quad \Rightarrow \quad y = \frac{y'}{ratio\ Y},\ x = \frac{x'}{ratio\ X}$$

예제 8.3.1 크기변경 & 최근접 이웃 보간- 02.scaling_nearset.cpp

```
01  import numpy as np, cv2
02  from Common.interpolation import scaling      # interpolation 모듈의 scaling() 함수 임포트
03
04  def scaling_nearest(img, size):                          # 크기 변경 함수3
05      dst = np.zeros(size[::-1], img.dtype)                # 행렬과 크기는 원소가 역순
06      ratioY, ratioX = np.divide(size[::-1], img.shape[:2]) # 변경 크기 비율
07      i = np.arange(0, size[1], 1)                         # 목적 영상 세로(i) 좌표 생성
08      j = np.arange(0, size[0], 1)                         # 목적 영상 가로(j) 좌표 생성
09      i, j = np.meshgrid(i, j)
10      y, x = np.int32(i / ratioY), np.int32(j / ratioX)    # 입력 영상 좌표
11      dst[i, j] = img[y, x]                                # 역방향 사상 → 입력 영상 좌표 계산
12
13      return dst
14
15  image = cv2.imread("images/interpolation.jpg", cv2.IMREAD_GRAYSCALE)
16  if image is None: raise Exception("영상파일 읽기 에러")
17
18  dst1 = scaling(image, (350, 400))                        # 크기 변경- 기본
19  dst2 = scaling_nearest(image, (350, 400))                # 크기 변경- 최근접 이웃 보간
20
21  cv2.imshow("image", image)
22  cv2.imshow("dst1- forward mapping", dst1)                # 순방향 사상
23  cv2.imshow("dst2- NN interpolation", dst2)               # 최근접 이웃 보간
24  cv2.waitKey(0)
```





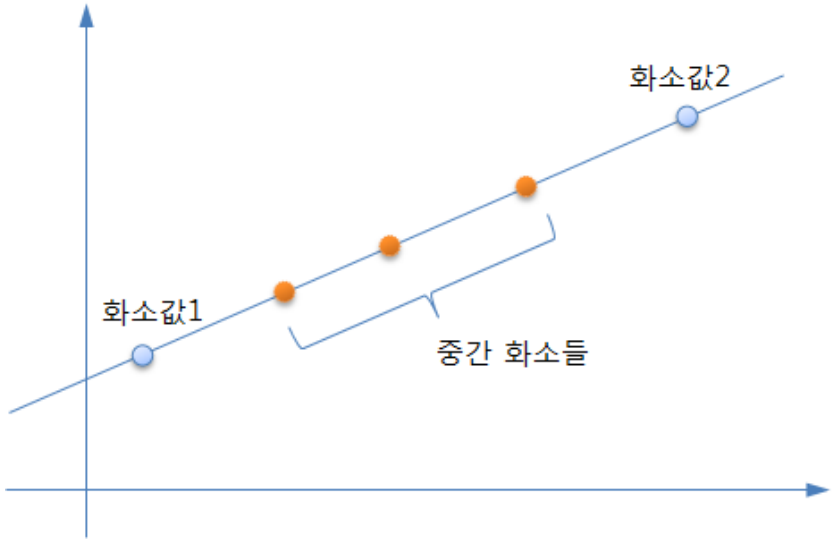
양선형 보간법

❖최근접 이웃 보간법

- 확대비율이 커지면, 모자이크 현상 혹은 경계부분에서 계단 현상 발생

❖선형 보간으로 해결

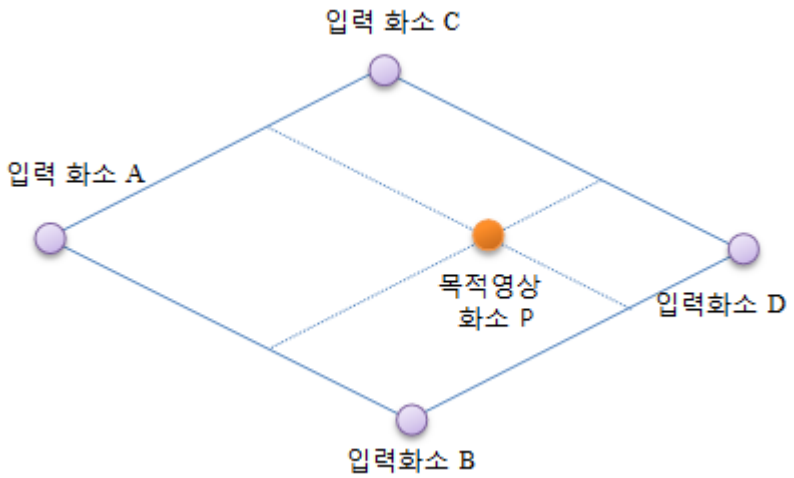
- 직선의 선상에 위치한 중간 화소들의 값은 직선의 수식을 이용해서 쉽게 계산

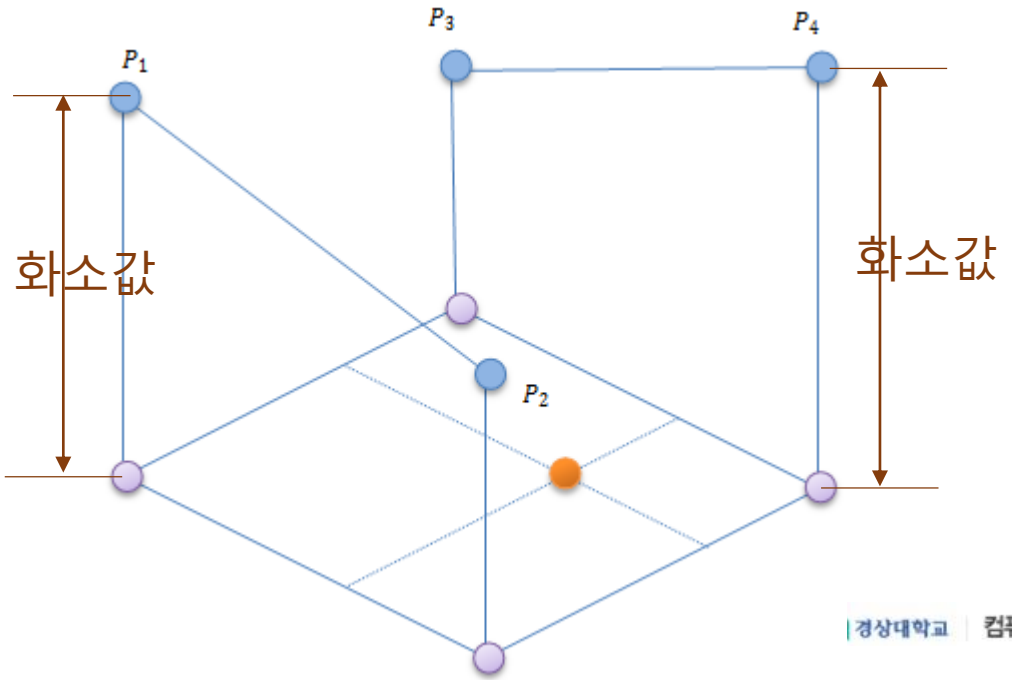


양선형 보간법

❖양선형 보간법

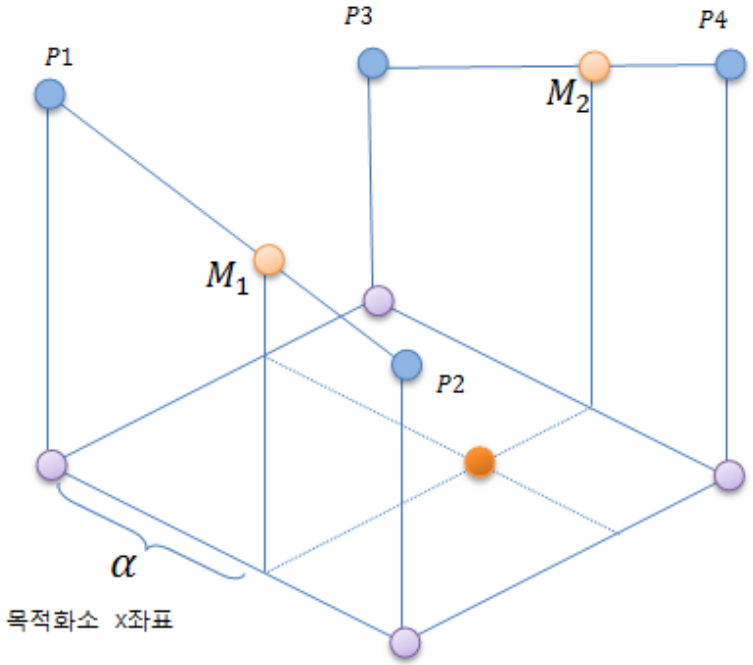
- 선형 보간을 두 단계에 걸쳐서 수행하기에 붙여진 이름
- 목적영상 화소(P)를 역변환으로 계산하여 가장 가까운 위치에 있는 입력영상의 4개 화소(A, B, C, D) 값 가져옴
- 4개 화소를 두 개씩(AB, CD) 묶어서 두 화소를 화소값(P_1, P_2, P_3, P_4)으로 잇는 직선 구성

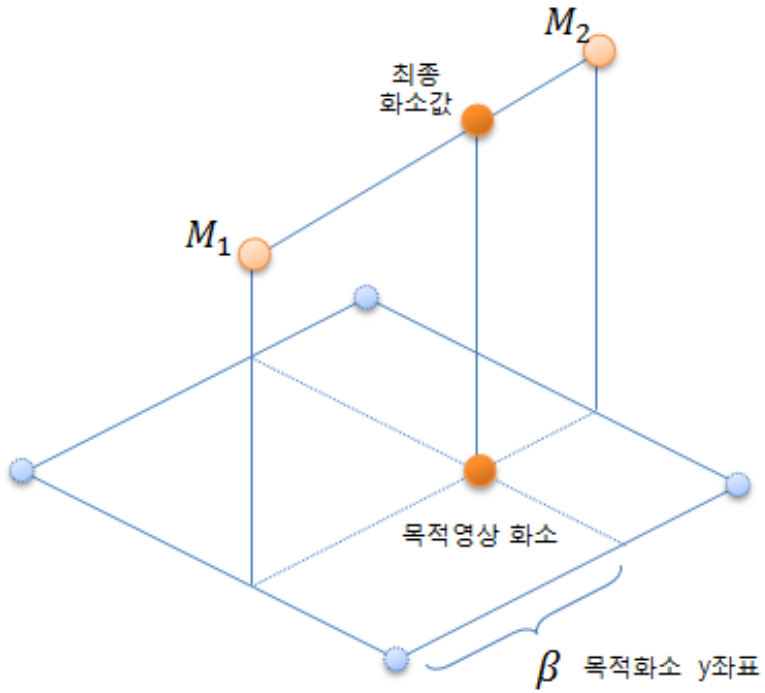




양선형 보간법

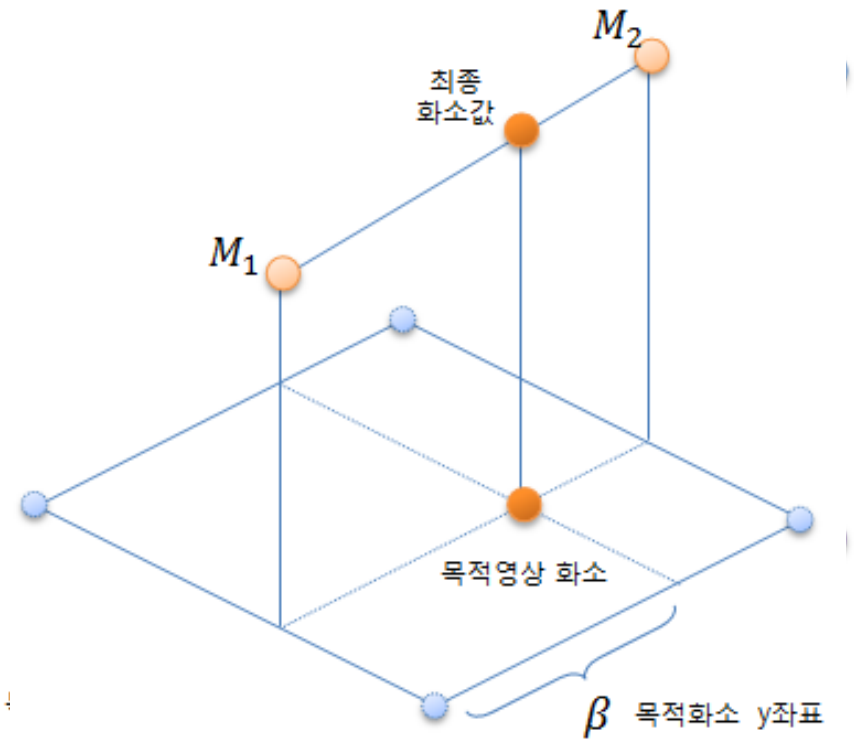
- 직선상에서 목적영상 화소의 좌표로 중간 위치 찾음
  - 거리 비율(α , 1- α)로 화소값(M_1 , M_2) 계산
- 두 개의 중간 화소값(M_1 , M_2)을 잇는 직선 다시 구성
  - 거리 비율(β , 1- β)로 최종 화소값 계산





양선형 보간법

❖수식으로 정리



$$M_1 = \alpha P_2 + (1-\alpha)P_1 = P_1 + \alpha(P_2 - P_1)$$

$$M_2 = \alpha P_4 + (1-\alpha)P_3 = P_3 + \alpha(P_4 - P_3)$$

$$P = \beta M_2 + (1-\beta)M_1 = M_1 + \beta(M_2 - M_1)$$

OpenCV 보간 옵션

❖OpenCV 보간 옵션

〈표 8.3.1〉 보간 방법에 대한 flag 옵션

옵션 상수	값	설명
INTER_NEAREST	0	최근접 이웃 보간
INTER_LINEAR	1	양선형 보간 (기본값)
INTER_CUBIC	2	바이큐빅 보간 – 4x4 이웃 화소 이용
INTER_AREA	3	픽셀 영역의 관계로 리샘플링
INTER_LANCZOS4	4	Lanczos 보간 – 8x8 이웃 화소 이용

OpenCV 보간 옵션

INTER_NEAREST

INTER_LINEAR

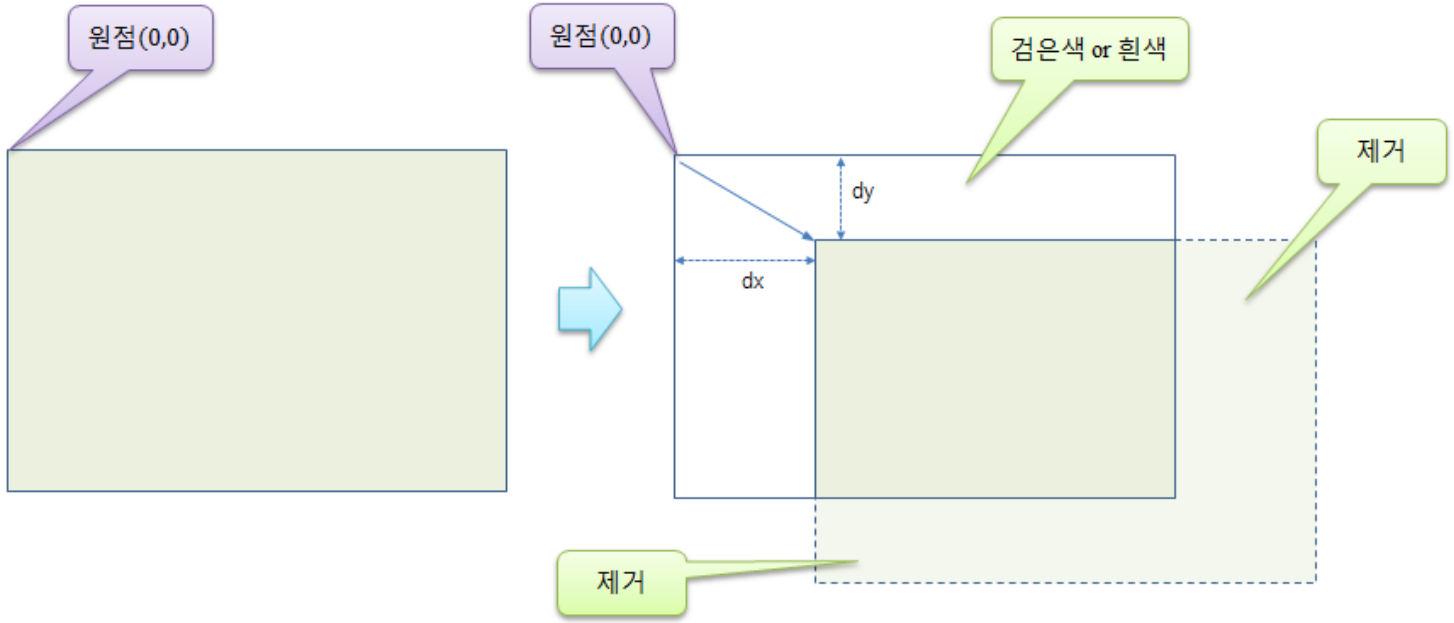

```
dst3 = cv2.resize(image, size, 0, 0, cv2.INTER_LINEAR)  # OpenCV 함수 – 양선형
dst4 = cv2.resize(image, size, 0, 0, cv2.INTER_NEAREST) # OpenCV 함수 – 최근접
```

평행 이동 (translation)

❖평행 이동

- 영상의 원점을 기준으로 모든 화소를 동일하게 가로 방향과 세로 방향으로 옮기는 것

Ex) 가로 방향으로 *dx*만큼, 세로 방향으로 *dy*만큼 전체 영상의 모든 화소 이동한 예



순방향 사상

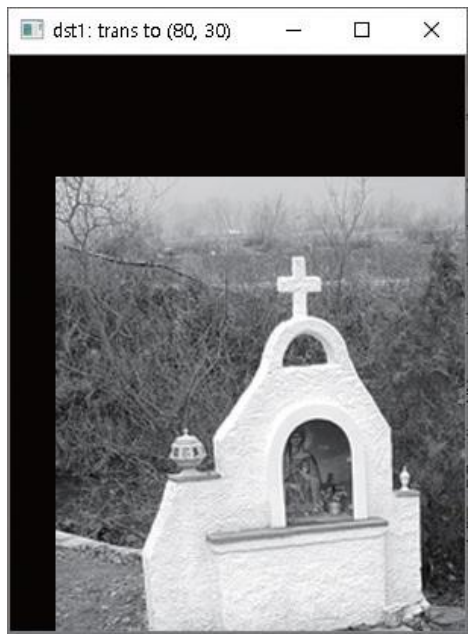
$$x' = x + dx$$
$$y' = y + dy$$

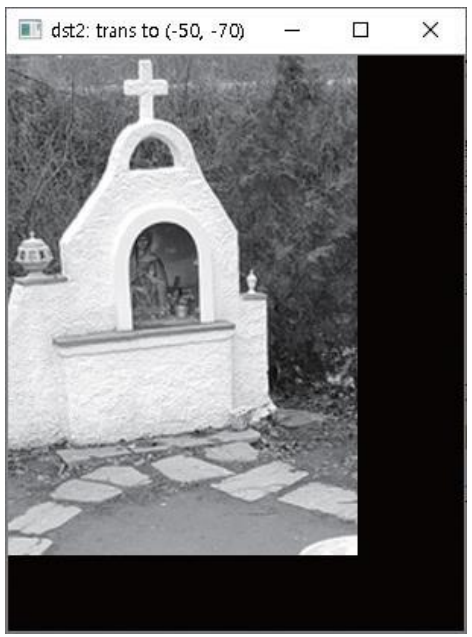
역방향사상

$$x = x' - dx$$
$$y = y' - dy$$

예제 8.4.1 영상 평행이동 – 04.translation.cpp

```
01  import numpy as np, cv2
02
03  def contain(p, shape):                          # 좌표(y,x)가 범위내 인지 검사
04      return 0<= p[0] < shape[0] and 0<= p[1] < shape[1]
05
06  def translate(img, pt):
07      dst = np.zeros(img.shape, img.dtype)        # 목적 영상 생성
08      for i in range(img.shape[0]):               # 목적 영상 순회– 역방향 사상
09          for j in range(img.shape[1]):
10              x, y = np.subtract((j, i) , pt)     # 좌표는 가로, 세로 순서
11              if contain((y, x), img.shape):      # 영상 범위 확인
12                  dst[i, j] = img[y, x]           # 행렬은 행, 열 순서
13      return dst
14
15  image = cv2.imread("images/translate.jpg", cv2.IMREAD_GRAYSCALE)
16  if image is None: raise Exception("영상파일 읽기 에러")
17
18  dst1 = translate(image, (30, 80))               # x=30, y=80 으로 평행이동
19  dst2 = translate(image, (-70, -50))
20
21  cv2.imshow("image", image)
22  cv2.imshow("dst1: transted to (30, 80)", dst1);
23  cv2.imshow("dst2: transted to (-70, -50)", dst2);
24  cv2.waitKey(0)
```

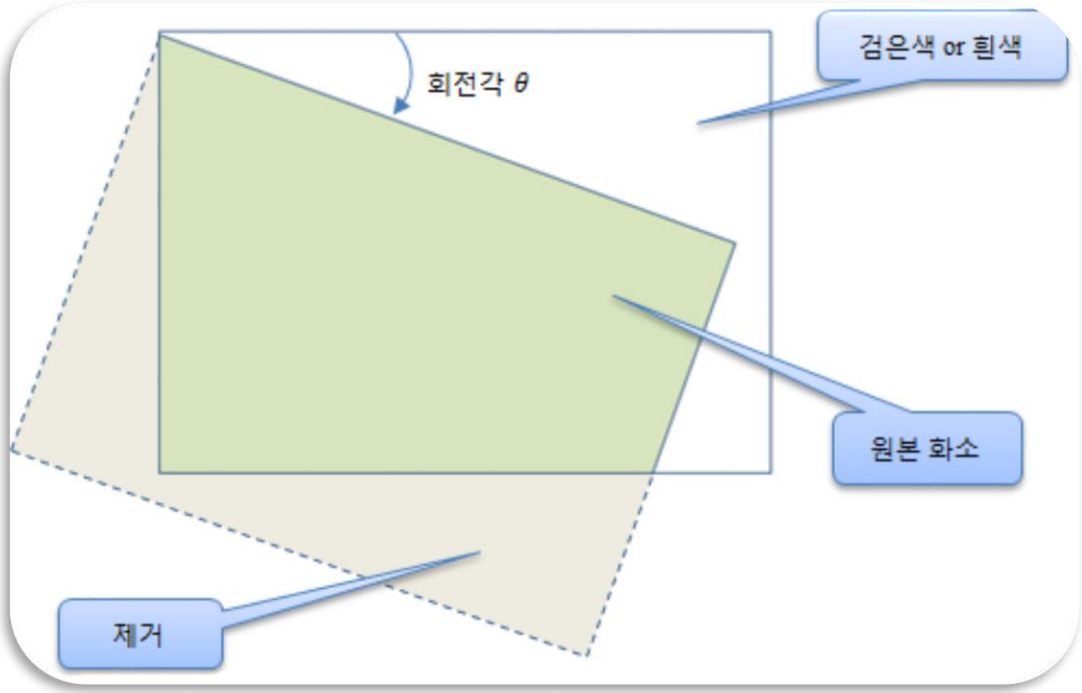




회전 (rotation)

❖영상의 원점을 기준으로 주어진 각도만큼 회전시킴
- 각도가 양수이면 반시계방향으로 회전

❖원점(0, 0)으로부터 θ만큼 회전



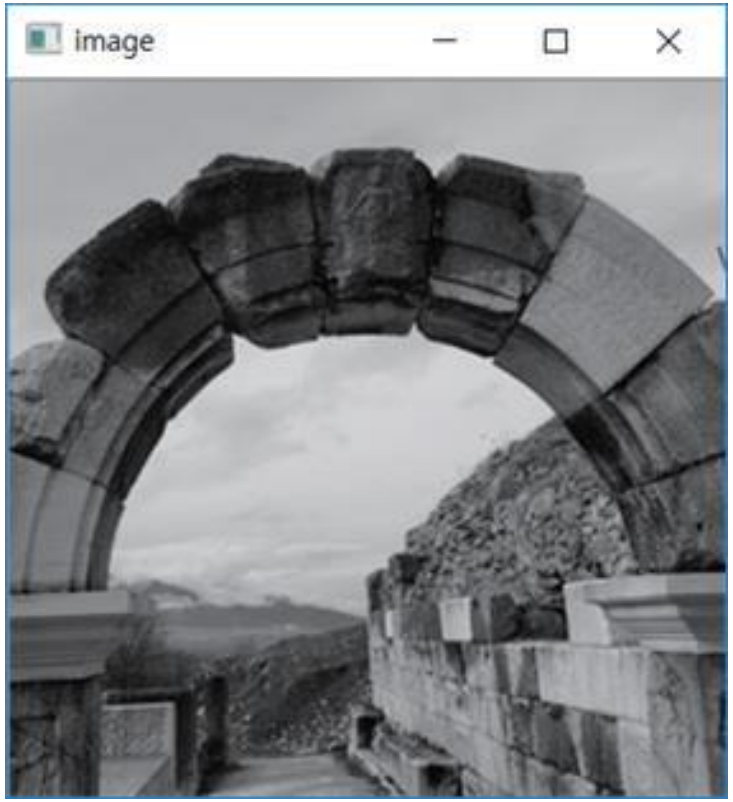
순방향 사상

$$x' = x \cdot \cos\theta - y \cdot \sin\theta$$
$$y' = x \cdot \sin\theta + y \cdot \cos\theta$$

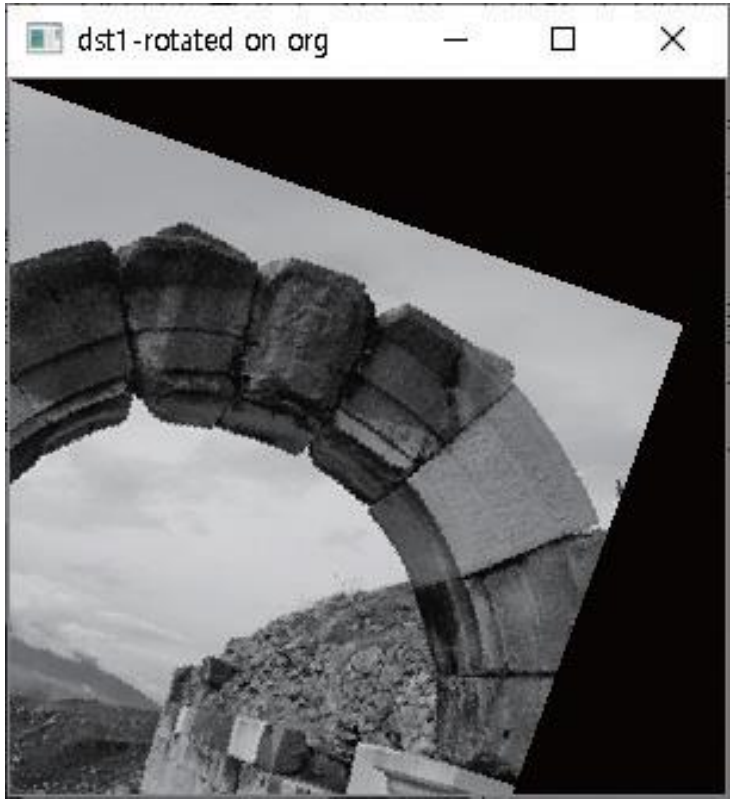
역방향사상

$$x = x' \cdot \cos\theta + y' \cdot \sin\theta$$
$$y = -x' \cdot \sin\theta + y' \cdot \cos\theta$$

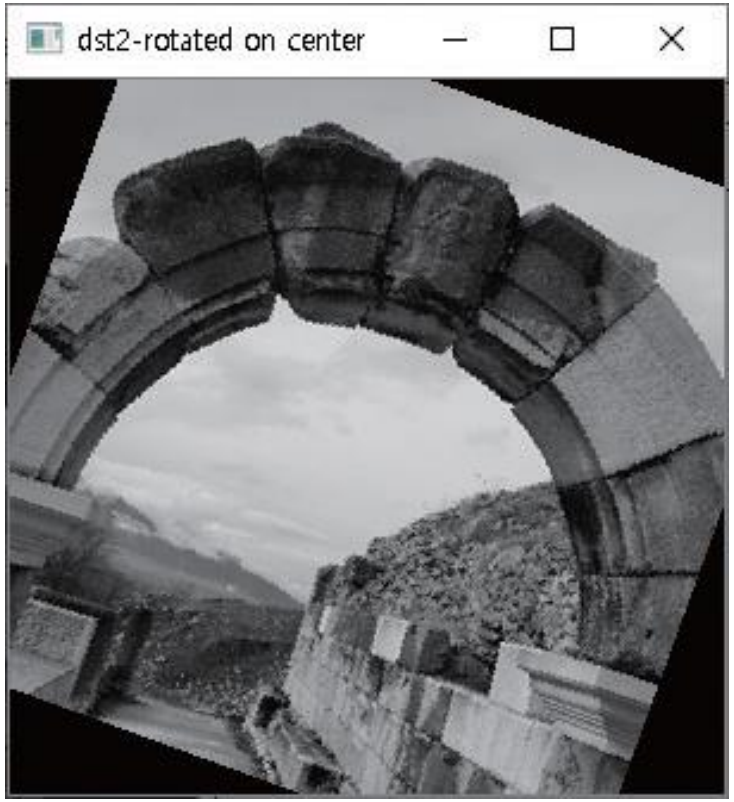
회전 (rotation)



입력 영상



원점 기준 회전



영상의 중심점 기준 회전

회전 (rotation)

❖특정 좌표(centerX, centerY)를 기준으로 θ만큼 회전

[step1] 회전의 기준점을 원점으로 이동시킨 후,

[step2] 회전 수행

[step3] 다시 본래 위치로 이동

역방향사상

$$x = x' \cdot \cos\theta + y' \cdot \sin\theta$$
$$y = -x' \cdot \sin\theta + y' \cdot \cos\theta$$

$$x = (x' - center\ X)\cos\theta + (y' - center\ Y)\sin\theta + center\ X$$
$$y = -(x' - center\ X)\sin\theta + (y' - center\ Y)\cos\theta + center\ Y$$

$$x = (x' - center\ X)\cos\theta + (y' - center\ Y)\sin\theta + center\ X$$
$$y = -(x' - center\ X)\sin\theta + (y' - center\ Y)\cos\theta + center\ Y$$

역방향사상

$$x = x' \cdot \cos\theta + y' \cdot \sin\theta$$
$$y = -x' \cdot \sin\theta + y' \cdot \cos\theta$$

```
18  def rotate_pt(img, degree, pt):                    # pt 기준 회전 변환 함수
19      dst = np.zeros(img.shape[:2], img.dtype)       # 목적 영상 생성
20      radian = (degree/180) * np.pi                  # 회전 각도- 라디언
21      sin, cos = math.sin(radian), math.cos(radian)  # 사인, 코사인 값 미리 계산
22
23      for i in range(img.shape[0]):                  # 목적 영상 순회- 역방향 사상
24          for j in range(img.shape[1]):
25              jj, ii = np.subtract((j, i), pt)       # 중심 좌표로 평행이동
26              y = -jj * sin + ii * cos               # 회선 변환 수식
27              x = jj * cos + ii * sin
28              x, y = np.add((x, y), pt)              # 중심 좌표로 평행이동
29              if contain((y, x), img.shape):         # 입력 영상의 범위 확인
30                  dst[i, j] = bilinear_value(img, (x, y))    # 화소값 양선형 보간
31      return dst
```

행렬 연산을 통한 기하학 변환: 어파인 변환

❖기하학 변환 수식을 행렬의 곱으로 표현 가능

✓ 크기 변환

$$x' = \alpha x$$
$$y' = \beta x$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

✓ 이동 변환

$$x' = x + t_x$$
$$y' = y + t_y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

어파인 변환

❖기하학 변환 수식을 행렬의 곱으로 표현 가능

✓ 회전 변환

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

어파인 변환

❖기하학 변환 수식을 행렬의 곱으로 표현 가능

✓ 어파인 변환

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$x' = \alpha_{11}x + \alpha_{12}y + \alpha_{13}$$

$$y' = \alpha_{21}x + \alpha_{22}y + \alpha_{23}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

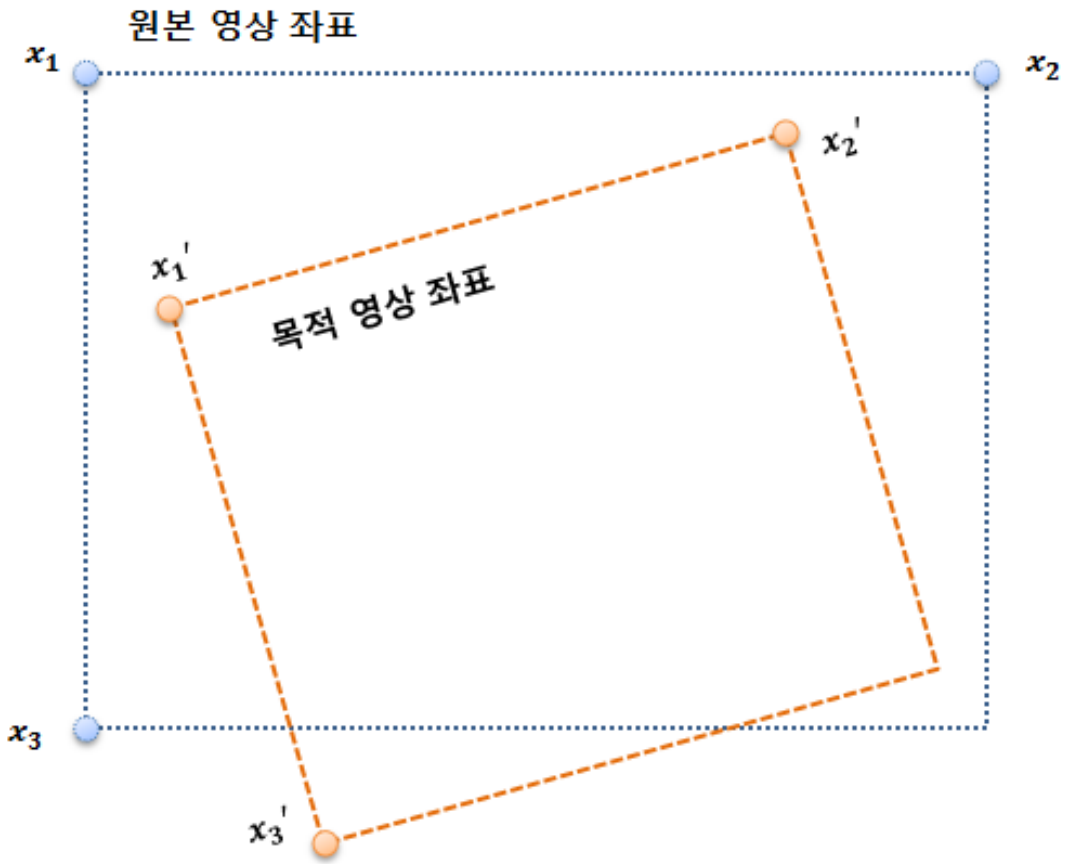
어파인 변환

❖각 변환 행렬을 행렬 곱으로 구성 가능

$$\text{어파인 변환행렬} = \begin{bmatrix} \cos\theta & -sin'\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

어파인 변환

❖입력 영상의 좌표 3개(x_1, x_2, x_3)와 변환이 완료된 목적영상에서 상응하는 좌표 3개(x'_1, x'_2, x'_3) 를 알면 → 어파인 행렬 구성 가능



OpenCV : getAffineTransform()

❖ 3쌍의 좌표 점으로 부터 2행 3열의 어파인 변환 행렬을 반환

```
getAffineTransform(
    InputArray src,        // src영상의 세 점의 좌표 (행렬)
    OutputArray dst,       // dst영상의 대응되는 세 점의 좌표 (행렬)
)
```

```
getAffineTransform(
    const Point2f  src[],     // src영상의 세 점의 좌표 (배열)
    const Point2f  dst[],     // dst영상의 대응되는 세 점의 좌표 (배열)
)
```

OpenCV : getRotationMatrix2D()

❖ 2행 3열의 회전 및 크기 변환 행렬을 반환

```
getRotationMatrix2D(
    Point2f  center,        // 회전 기준점
    double  angle,          // 회전 각도 (양수가 반시계 방향 회전을 의미)
    double  scale           // 크기 변화
)
```

OpenCV : warpAffine()

❖ Src 영상에 어파인 변환을 적용하여 dst 영상 생성

```
warpAffine(
    InputArray    src,    // 입력 영상
    InputArray    M,      // 어파인 변환 행렬
    Size   dsize,         // 출력 영상의 크기
    int   flags=INTER_LINEAR,    // 보간방법
    int   borderMode=BORDER_CONSTANT,   // 경계지정 방법
    const Scalar &borderValue=Scalar()   // 경계 값 지정(default=0)
)
```

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$dst(x, y) = src(m_{11}x + m_{12}y + m_{13}, m_{21}x + m_{22}y + m_{23})$$

```python
image = cv2.imread("images/affine.jpg", cv2.IMREAD_GRAYSCALE)
if image is None: raise Exception("영상파일 읽기 에러")

center = (200, 200)                                  # 회전 변환 기준 좌표
angle, scale = 30, 1                                 # 30도 회전, 크기 변경 안 함
size = image.shape[::-1]                             # 크기는 행태의 역순

pt1 = np.array([( 30, 70),(20, 240), (300, 110)], np.float32)
pt2 = np.array([(120, 20),(10, 180), (280, 260)], np.float32)
aff_mat = cv2.getAffineTransform(pt1, pt2)           # 3개 좌표쌍 어파인 행렬 생성
rot_mat = cv2.getRotationMatrix2D(center, angle, scale)      # 어파인 행렬

dst1 = affine_transform(image, aff_mat)              # 어파인 변환 수행
dst2 = affine_transform(image, rot_mat)              # 회전 변환 수행
dst3 = cv2.warpAffine(image, aff_mat, size, cv2.INTER_LINEAR)
dst4 = cv2.warpAffine(image, rot_mat, size, cv2.INTER_LINEAR)

image = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
dst1 = cv2.cvtColor(dst1, cv2.COLOR_GRAY2BGR )
dst3 = cv2.cvtColor(dst3, cv2.COLOR_GRAY2BGR )

for i in range(len(pt1)):                            # 3개 좌표 표시
    cv2.circle(image, tuple(pt1[i].astype(int)), 3, (0, 0, 255), 2)
    cv2.circle(dst1 , tuple(pt2[i].astype(int)), 3, (0, 0, 255), 2)
    cv2.circle(dst3 , tuple(pt2[i].astype(int)), 3, (0, 0, 255), 2)
```

심화예제 8.6.2 어파인 변환의 연결 – 07.affine_combination.py

```
01  import numpy as np, math, cv2
02  from Common.interpolation import affine_transform     # 저자 구현 어파인변환 함수 임포트
03
04  def getAffineMat(center, degree, fx=1, fy=1, translate=(0,0)):    # 변환 행렬 합성 함수
05      scale_mat = np.eye(3, dtype=np.float32)             # 크기 변경 행렬
06      cen_trans = np.eye(3, dtype=np.float32)             # 중점 평행 이동
07      org_trans = np.eye(3, dtype=np.float32)             # 원점 평행 이동
08      trans_mat = np.eye(3, dtype=np.float32)             # 좌표 평행 이동
09      rot_mat   = np.eye(3, dtype=np.float32)             # 회전 변환 행렬
10
11      radian = degree / 180 * np.pi                       # 회전 각도– 라디언 계산
12      rot_mat[0] = [ np.cos(radian), np.sin(radian), 0]   # 회전행렬 0행
13      rot_mat[1] = [-np.sin(radian), np.cos(radian), 0]   # 회전행렬 1행
14
15      cen_trans[:2, 2] = center                           # 중심 좌표 이동
16      org_trans[:2, 2] = -center[0], -center[1]           # 원점으로 이동
17      trans_mat[:2, 2] = translate                        # 평행 이동 행렬의 원소 지정
18      scale_mat[0, 0], scale_mat[1, 1] = fx, fy           # 크기 변경 행렬의 원소 지정
19
20      ret_mat = cen_trans.dot(rot_mat.dot(trans_mat.dot(scale_mat.dot(org_trans))))
21      # ret_mat = cen_trans.dot(rot_mat.dot(scale_mat.dot(trans_mat.dot(org_trans))))
22      return np.delete(ret_mat, 2, axis=0)                # 마지막행 제거 ret_mat[0:2,:]
23
```

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

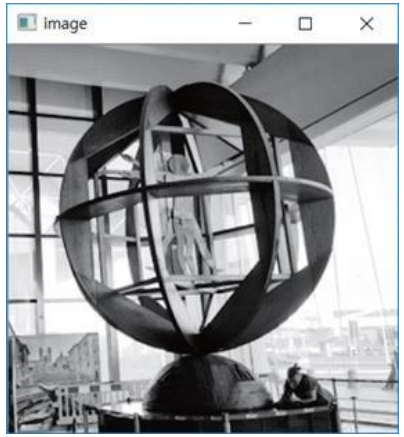
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

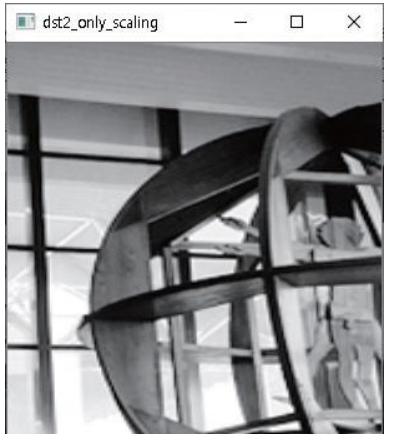
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

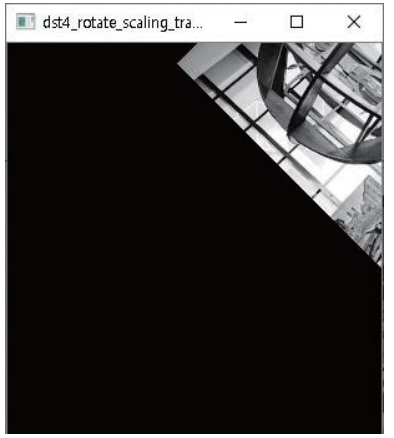
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```python
image = cv2.imread("images/affine2.jpg", cv2.IMREAD_GRAYSCALE)
if image is None: raise Exception("영상파일 읽기 에러")

size = image.shape[::-1]
center = np.divmod(size, 2)[0]                    # 회전 중심 좌표
angle, tr = 45, (200, 0)                          # 각도와 평행이동 값 지정

aff_mat1 = getAffineMat(center, angle)            # 중심 좌표 기준 회전
aff_mat2 = getAffineMat((0,0), 0, 2.0, 1.5)       # 크기 변경– 확대
aff_mat3 = getAffineMat(center, angle, 0.7, 0.7)  # 회전 및 축소
aff_mat4 = getAffineMat(center, angle, 0.7, 0.7, tr)  # 복합 변환

dst1 = cv2.warpAffine(image, aff_mat1, size)      # OpenCV 함수
dst2 = cv2.warpAffine(image, aff_mat2, size)
dst3 = affine_transform(image, aff_mat3)      # 사용자 정의 함수
dst4 = affine_transform(image, aff_mat4)

cv2.imshow("image", image)
cv2.imshow("dst1_only_rotate", dst1)
cv2.imshow("dst2_only_scaling", dst2)
cv2.imshow("dst3_rotate_scaling", dst3)
cv2.imshow("dst4_rotate_scaling_translate", dst4)
cv2.waitKey(0)
```







원근 투시(투영) 변환

❖아테네 학당

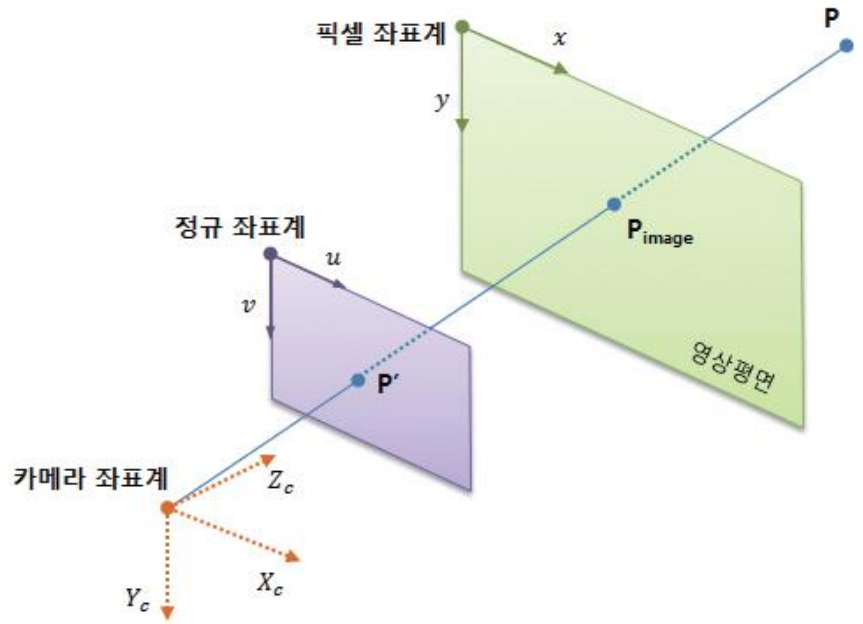
- 벽면에 그려진 그림에서 이렇게 입체감을 느끼는 이유 → 원근법

- 원근법
  - 눈에 보이는 3차원의 세계를 2차원의 그림(평면)으로 옮길 때에 관찰자가 보는 것 그대로 사물과의 거리를 반영하여 그리는 방법

원근 투시(투영) 변환

❖원근 투시 변환(perspective projection transformation)

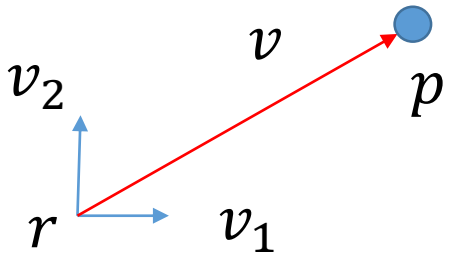
- 원근법을 영상 좌표계에서 표현하는 것
- 3차원의 실세계 좌표를 투영 스크린상의 2차원 좌표로 표현할 수 있도록 변환해 주는것



원근 투시(투영) 변환

❖동차 좌표계(homogeneous coordinates)

- 벡터와 점(point)을 동일한 차원으로 표현하기 위하여 제시된 좌표계
- n차원의 투영 공간을 n+1개의 좌표로 나타냄.
  - 직교 좌표인 (x, y)를 (x, y, 1)로 표현하는 것
  - 일반화해서 0이 아닌 상수 w에 대해 (x, y)를 (wx, wy, w)로 표현
  - 상수 w가 무한히 많기 때문에 (x, y)에 대한 동차 좌표 표현은 무한히 많이 존재



$$v = 4v_1 + 3v_2$$

$$p = r + 4v_1 + 3v_2$$

$$(0, 4, 3) \longrightarrow (4, 3, 0)$$

$$v = 0 \cdot r + 4v_1 + 3v_2$$

$$p = 1 \cdot r + 4v_1 + 3v_2$$

$$(1, 4, 3) \longrightarrow (4, 3, 1)$$

원근 투시(투영) 변환

❖동차 좌표계(homogeneous coordinates)

- 동차 좌표계에서 한 점(wx, wy, w)을 직교 좌표로 나타내면
  - 각 원소를 w로 나누어서 (x/w, y/w)가 됨

Ex) 동차 좌표계에서 한 점(5, 7, 5) → 직교 좌표에서(5/5, 7/5) 즉, (1 , 1.4)

2차원 point $(x, y) \longrightarrow (x, y, 1)$

3차원 point $(x, y, z) \longrightarrow (x, y, z, 1)$

원근 투시(투영) 변환

❖원근 변환을 수행하는 행렬

$$w \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

❖OpenCV 함수

- cv::getPerspectiveTransform() 함수
  - 4개의 좌표쌍으로부터 원근 변환 행렬을 계산
- cv::warpPerspective() 함수
  - 원근변환 행렬에 따라서 원근 변환 수행

OpenCV : getPerspectiveTransform()

❖ 4쌍의 좌표 점으로 부터 3행 3열의 원근 변환 행렬을 반환

```
getPerspectiveTransform(
    InputArray   src,       // 입력 영상의 4개의 좌표
    InputArray   dst        // 출력 영상의 4개의 좌표
)
```

OpenCV : warpPerspective()

❖ Src 영상에 원근 변환을 적용하여 dst 영상 생성

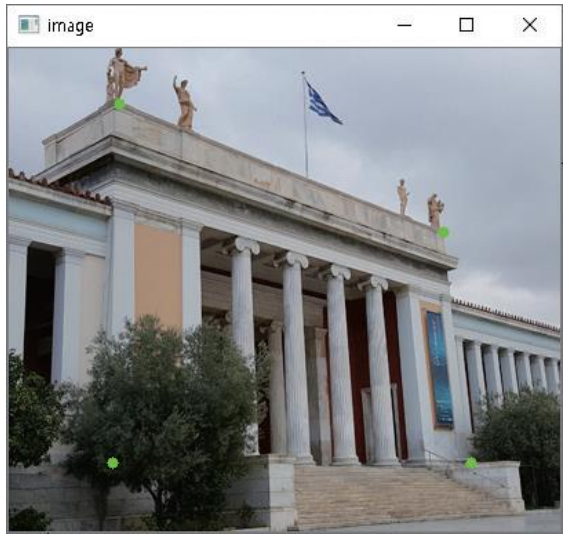
```
warpPerspective(
    InputArray    src,    // 입력 영상
    OutputArray   dst,    // 출력 영상
    InputArray    M,      // 원근 변환 행렬
    Size   dsize,         // 출력 영상의 크기
    int    flags=INTER_LINEAR,    // 보간방법
    int   borderMode=BORDER_CONSTANT,   // 경계지정 방법
    const Scalar &borderValue=Scalar()   // 경계 값 지정(default=0)
)
```

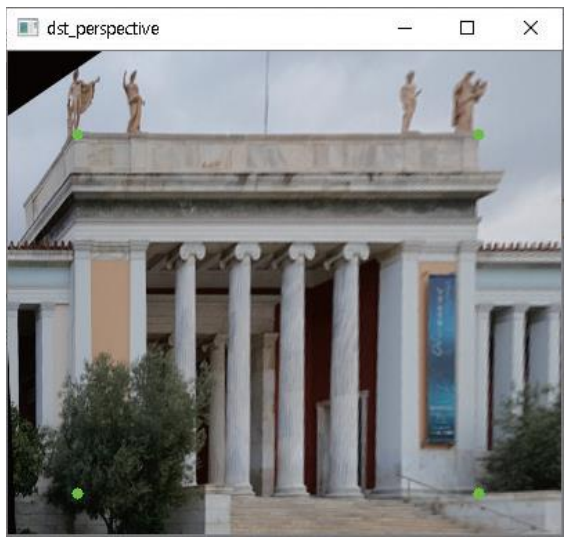
$$\omega \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$dst(x,y) = src\left(\frac{m_{11}x + m_{12}y + m_{13}}{m_{31}x + m_{32}y + m_{33}}, \frac{m_{21}x + m_{22}y + m_{23}}{m_{31}x + m_{32}y + m_{33}}\right)$$

심화예제 8.7.1 원근 왜곡 보정 – 10.perspective_transform.py

```
01  import numpy as np, cv2
02
03  image = cv2.imread("images/perspective.jpg", cv2.IMREAD_COLOR)
04  if image is None: raise Exception("영상파일 읽기 에러")
05
06  pts1 = np.float32([(80, 40), (315, 133), (75, 300), (335, 300)])    # 입력 영상 4개 좌표
07  pts2 = np.float32([(50, 60), (340, 60), (50, 320), (340, 320)])     # 목적 영상 4개 좌표
08
09  perspect_mat = cv2.getPerspectiveTransform(pts1, pts2)              # 원근 변환 행렬
10  dst = cv2.warpPerspective(image, perspect_mat, image.shape[1::-1], cv2.INTER_CUBIC)
11  print("[perspect_mat] = \n%s\n" % perspect_mat )
12
```





OpenCV에서 제공하는 기타 변환

❖flip()

- 상하, 좌우, 원점 대칭 변환

```
flip(
    InputArray    src,    // 입력 영상
    int   flipcode        // 0이면 상하, 양수면 좌우, 음수면 원점으로 대칭 변환
)
```

$$dst_{ij} = \begin{cases} src_{src.rows-i-1,j} & \text{, if filpcode} = 0 \\ src_{i,src.cols-j-1} & \text{, if flipcode} > 0 \\ src_{src.rows-i-1,src.cols-j-1} & \text{, if flipcode} < 0 \end{cases}$$

Vertical flipping

Horizontal flipping

OpenCV에서 제공하는 기타 변환

❖remap()

- 모든 변환을 가능하게 하는 포괄적 변환
- 제시된 사상 행렬 map1, map2를 이용하여 변환

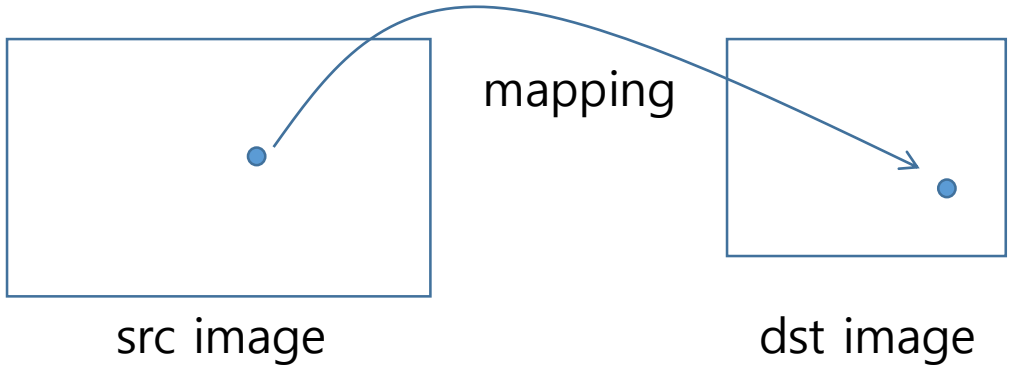
```
remap(
    InputArray    src,    // 입력 영상
    InputArray    map1,      // x 좌표
    InputArray    map2,      // y 좌표
    int   interpolation,
    int   borderMode=BORDER_CONSTANT,    // 경계 값 지정 방법
    const Scalar &borderValue=Scalar()          // 경계 값 지정(default=0)
)
```

$$dst(x, y) = src(map_x(x\ y), map_y(x\ y))$$

(유의) This function cannot operate in-place.

Summary

❖기하학 처리



- 회전 변환(Rotation), 크기 변환(Scaling), 이동 변환(Translation)
- 아파인 변환 (Affine transform)
  → 3쌍의 대응점 필요

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- 원근 변환 (Perspective transform)
  → 4쌍의 대응점 필용

$$\omega \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$