2.3.2 연결요소

- 범람 채움
  - 아주 큰 연결요소를 만나면 스택 오버플로우라는 메모리 문제 발생 가능

0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	1	0	1	1	0
0	1	0	1	0	1	1	0	1	0
0	1	0	1	0	1	0	0	1	0
0	1	0	1	0	1	0	0	1	0
0	1	1	0	0	1	0	0	1	0
0	0	0	0	0	0	0	0	0	0

(a) 입력 이진 영상

알고리즘 2-5 **범람 채움(4-연결성 버전)**

입력 : 이진 영상 $b(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

출력 : 번호를 매긴 영상 $l(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

```
b를 l로 복사한다. 이때 0은 0, 1은 -1로 복사한다. // -1은 아직 번호를 안 붙였음을 표시
l의 경계, 즉 j=0, j=M-1, i=0, i=N-1인 화소를 0으로 설정한다. // 영상 바깥으로 나가는 것을 방지
label=1;
for(j=1 to M-2)
  for(i=1 to N-2) {
    if(l(j, i)=-1) {
      flood_fill4(l, j, i, label);
      label++;
    }
  }

// 4-연결성 범람 채움 함수
function flood_fill4(l, j, i, label) {
  if(l(j, i)=-1) { // 아직 번호를 안 붙인 화소이면
    l(j, i)=label;
    flood_fill4(l, j, i+1, label); // east
    flood_fill4(l, j-1, i, label); // north
    flood_fill4(l, j, i-1, label); // west
    flood_fill4(l, j+1, i, label); // south
  }
}
```

2.3.2 연결요소

- 범람 채움
  - 아주 큰 연결요소를 만나면 스택 오버플로우라는 메모리 문제 발생 가능
  - 메모리와 계산 시간 측면에서 더 효율적

알고리즘 2-6 범람 채움(메모리를 적게 사용하는 버전)

입력 : 이진 영상 $b(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

출력 : 번호를 매긴 영상 $l(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

```
b를 l로 복사한다. 이때 0은 0, 1은 -1로 복사한다.  // -1은 아직 번호를 안 붙였음을 표시
l의 경계, 즉 j=0, j=M-1, i=0, i=N-1인 화소를 0으로 설정한다.  // 영상 바깥으로 나가는 것 방지
label=1;
for(j=1 to M-2)
  for(i=1 to N-2) {
    if(l(j, i)=-1) {
      efficient_floodfill4(l, j, i, label);
      label++;
    }
  }
```

2.3.2 연결요소

- 범람 채움
  - 메모리와 계산 시간 측면에서 더 효율적

알고리즘 2-6 범람 채움(메모리를 적게 사용하는 버전)

입력 : 이진 영상 $b(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

출력 : 번호를 매긴 영상 $l(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

```
b를 l로 복사한다. 이때 0은 0, 1은 -1로 복사한다.  // -1 은 아직 번호를 안 붙였음을 표시
l의 경계, 즉 j=0, j=M-1, i=0, i=N-1인 화소를 0으로 설정한다.  // 영상 바깥으로 나가는 것 방지
label=1;
for(j=1 to M-2)
  for(i=1 to N-2) {
    if(l(j, i)=-1 ) {
      efficient_floodfill4(l, j, i, label);
      label++;
    }
  }

```

알고리즘 2-5 범람 채움(4-연결성 버전)

입력 : 이진 영상 $b(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

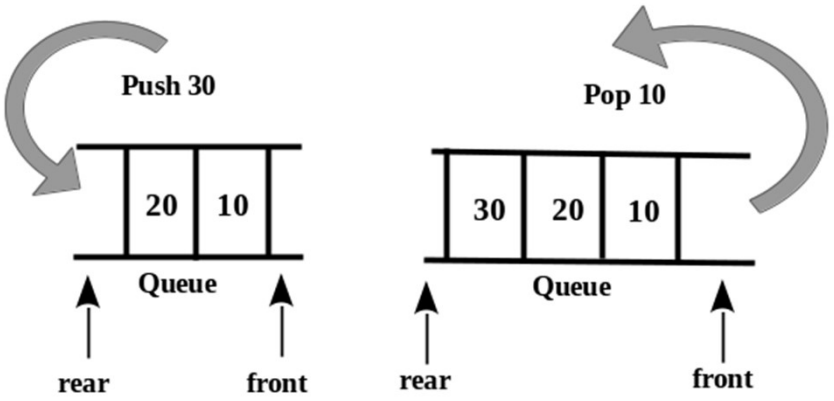
출력 : 번호를 매긴 영상 $l(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

```
b를 l로 복사한다. 이때 0은 0, 1은 -1로 복사한다. // -1은 아직 번호를 안 붙였음을 표시
l의 경계, 즉 j=0, j=M-1, i=0, i=N-1 인 화소를 0으로 설정한다. // 영상 바깥으로 나가는 것을 방지
label=1;
for(j=1 to M-2)
  for(i=1 to N-2) {
    if(l(j, i)=-1) {
      flood_fill4(l, j, i, label);
      label++;
    }
  }

// 4-연결성 범람 채움 함수
function flood_fill4(l, j, i, label) {
  if(l(j, i)=-1 ) { // 아직 번호를 안 붙인 화소이면
    l(j, i)=label;
    flood_fill4(l, j, i+1, label); // east
    flood_fill4(l, j-1, i, label); // north
    flood_fill4(l, j, i-1, label); // west
    flood_fill4(l, j+1, i, label); // south
  }
}
```

2.3.2 연결요소

- Queue
  - First-in First-out




```
// 메모리를 적게 사용하는 효율적인 4-연결성 범람 채움 함수
function efficient_floodfill4(I, j, i, label) {
  Q=∅; // 빈 큐 Q를 생성한다.
  push(Q,(j, i));
  while(Q≠∅) {
    (y, x)=pop(Q); // Q에서 원소를 하나 꺼낸다.
    if(I(y, x)=-1) {
      left=right=x;
      while(I(y, left-1)=-1) left--;  // 아직 미처리 상태인 열을 찾는다.
      while(I(y, right+1)=-1) right++;
      for(c=left to right) {
        I(y, c)=label;
        if(I(y-1, c)=-1 and (c=left or I(y-1, c-1)≠-1)) push(Q, (y-1, c));
        if(I(y+1, c)=-1 and (c=left or I(y+1, c-1)≠-1)) push(Q, (y+1, c));
      }
    }
  }
}
```

2.3.2 연결요소

- 20~21행
  : (y, x)를 중심으로 좌우로 확장
  : -1을 갖는 연속된 화소 열을 찾음
  : 찾은 화소 열을 label 값으로 대체

- 24~25행
  : 위아래를 확인하여 추가로 처리할 화소를 큐에 삽입

```
12  // 메모리를 적게 사용하는 효율적인 4-연결성 범람 채움 함수
13  function efficient_floodfill4(I, j, i, label) {
14    Q=∅; // 빈 큐 Q를 생성한다.
15    push(Q,(j, i));
16    while(Q≠∅) {
17      (y, x)=pop(Q); // Q에서 원소를 하나 꺼낸다.
18      if(I(y, x)=-1) {
19        left=right=x;
20        while(I(y, left-1)=-1) left--;  // 아직 미처리 상태인 열을 찾는다.
21        while(I(y, right+1)=-1) right++;
22        for(c=left to right) {
23          I(y, c)=label;
24          if(I(y-1, c)=-1 and (c=left or I(y-1, c-1)≠-1)) push(Q, (y-1, c));
25          if(I(y+1, c)=-1 and (c=left or I(y+1, c-1)≠-1)) push(Q, (y+1, c));
26        }
27      }
28    }
29  }
```

2.3.2 연결요소

- 20~21행
  : (y, x)를 중심으로 좌우로 확장
  : -1을 갖는 연속된 화소 열을 찾음
  : 찾은 화소 열을 label 값으로 대체

알고리즘 2-6 **범람 채움(메모리를 적게 사용하는 버전)**

입력 : 이진 영상 $b(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

출력 : 번호를 매긴 영상 $l(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

```
1   b를 l로 복사한다. 이때 0은 0, 1은 -1로 복사한다.  // -1은 아직 번호를 안 붙였음을 표시
2   l의 경계, 즉 j=0, j=M-1, i=0, i=N-1인 화소를 0으로 설정한다.  // 영상 바깥으로 나가는 것 방지
3   label=1;
4   for(j=1 to M-2)
5     for(i=1 to N-2) {
6       if(l(j, i)=-1) {
7         efficient_floodfill4(l, j, i, label);
8         label++;
9       }
10    }
11
12  // 메모리를 적게 사용하는 효율적인 4-연결성 범람 채움 함수
13  function efficient_floodfill4(l, j, i, label) {
14    Q=∅;  // 빈 큐 Q를 생성한다.
15    push(Q,(j, i));
16    while(Q≠∅) {
17      (y, x)=pop(Q);  // Q에서 원소를 하나 꺼낸다.
18      if(l(y, x)=-1) {
19        left=right=x;
20        while(l(y, left-1)=-1) left--;   // 아직 미처리 상태인 열을 찾는다.
21        while(l(y, right+1)=-1) right++;
22        for(c=left to right) {
23          l(y, c)=label;
24          if(l(y-1, c)=-1 and (c=left or l(y-1, c-1)≠-1)) push(Q, (y-1, c));
25          if(l(y+1, c)=-1 and (c=left or l(y+1, c-1)≠-1)) push(Q, (y+1, c));
26        }
27      }
28    }
29  }
```

0	0	0	0	0	0	0	0	0	0
0	0	0	0	**1**	**1**	0	0	0	0
0	0	0	0	0	**1**	0	0	0	0
0	0	0	0	0	**1**	0	0	0	0
0	**1**	**1**	0	0	**1**	0	**1**	**1**	0
0	**1**	0	**1**	0	**1**	**1**	0	**1**	0
0	**1**	0	**1**	0	**1**	0	0	**1**	0
0	**1**	0	**1**	0	**1**	0	0	**1**	0
0	**1**	**1**	0	0	**1**	0	0	**1**	0
0	0	0	0	0	0	0	0	0	0

(a) 입력 이진 영상

0	0	0	0	0	0	0	0	0	0
0	0	0	0	**1**	**1**	0	0	0	0
0	0	0	0	0	**1**	0	0	0	0
0	0	0	0	0	**1**	0	0	0	0
0	**2**	**2**	0	0	**1**	0	**3**	**3**	0
0	**2**	0	**4**	0	**1**	**1**	0	**3**	0
0	**2**	0	**4**	0	**1**	0	0	**3**	0
0	**2**	0	**4**	0	**1**	0	0	**3**	0
0	**2**	**2**	0	0	**1**	0	0	**3**	0
0	0	0	0	0	0	0	0	0	0

(b) 번호 붙이기(4-연결성)

그림 2-17 **연결요소 번호 붙이기**

```
// 메모리를 적게 사용하는 효율적인 4-연결성 범람 채움 함수
function efficient_floodfill4(I, j, i, label) {
  Q=∅; // 빈 큐 Q를 생성한다.
  push(Q,(j, i));
  while(Q≠∅) {
    (y, x)=pop(Q); // Q에서 원소를 하나 꺼낸다.
    if(I(y, x)=-1) {
      left=right=x;
      while(I(y, left-1)=-1) left--;  // 아직 미처리 상태인 열을 찾는다.
      while(I(y, right+1)=-1) right++;
      for(c=left to right) {
        I(y, c)=label;
        if(I(y-1, c)=-1 and (c=left or I(y-1, c-1)≠-1)) push(Q, (y-1, c));
        if(I(y+1, c)=-1 and (c=left or I(y+1, c-1)≠-1)) push(Q, (y+1, c));
      }
    }
  }
}
```

2.4 세 가지 기본 연산

2.4 영상 처리의 세 가지 기본 연산

- 2.4.1 점 연산 (Point operation)
  - 오직 자신의 명암 값에 따라 새로운 값을 결정

- 2.4.2 영역 연산 (Area operation)
  - 이웃에 있는 몇 개 화소들의 명암 값에 따라 새로운 값 결정

- 2.4.3 기하 연산 (Geometric operation)
  - 일정한 기하학적 규칙에 따라 다른 곳에 있는 값을 새로운 값 결정

2.4.1 점 연산

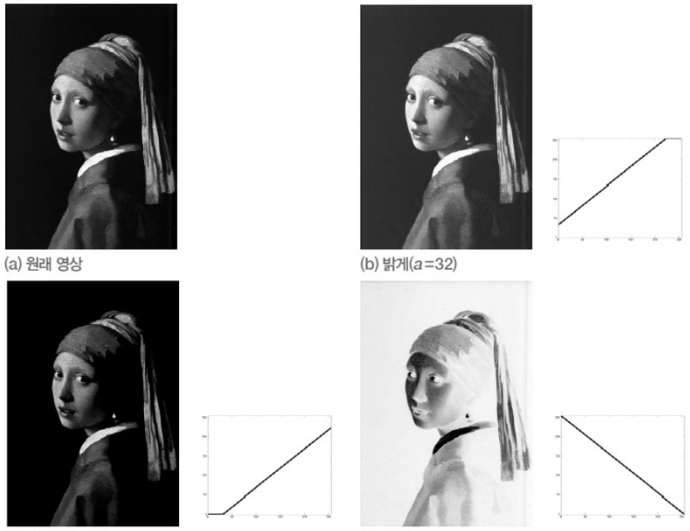
- 출력 영상 f_{out}에서 픽셀 (j, i) 값은 k개의 입력 영상에서 같은 위치에 존재하는 픽셀 값에 따라 정해짐

$$f_{out}(j,i) = t(f_1(j,i),\ f_2(j,i), \cdots\ f_k(j,i))$$

- 대부분의 경우: $k = 1$ (즉, 한 장의 영상을 입력으로)

- 점 연산 예시

$$f_{out}(j,i) = t(f(j,i)) = \begin{cases} \min(f(j,i) + a, L-1), & \text{(밝게)} \\ \max(f(j,i) - a, 0), & \text{(어둡게)} \\ (L-1) - f(j,i), & \text{(반전)} \end{cases}$$



(a) 원래 영상 (b) 밝게(a=32) (c) 어둡게(a=32) (d) 반전

그림 2-18 여러 가지 선형 점 연산

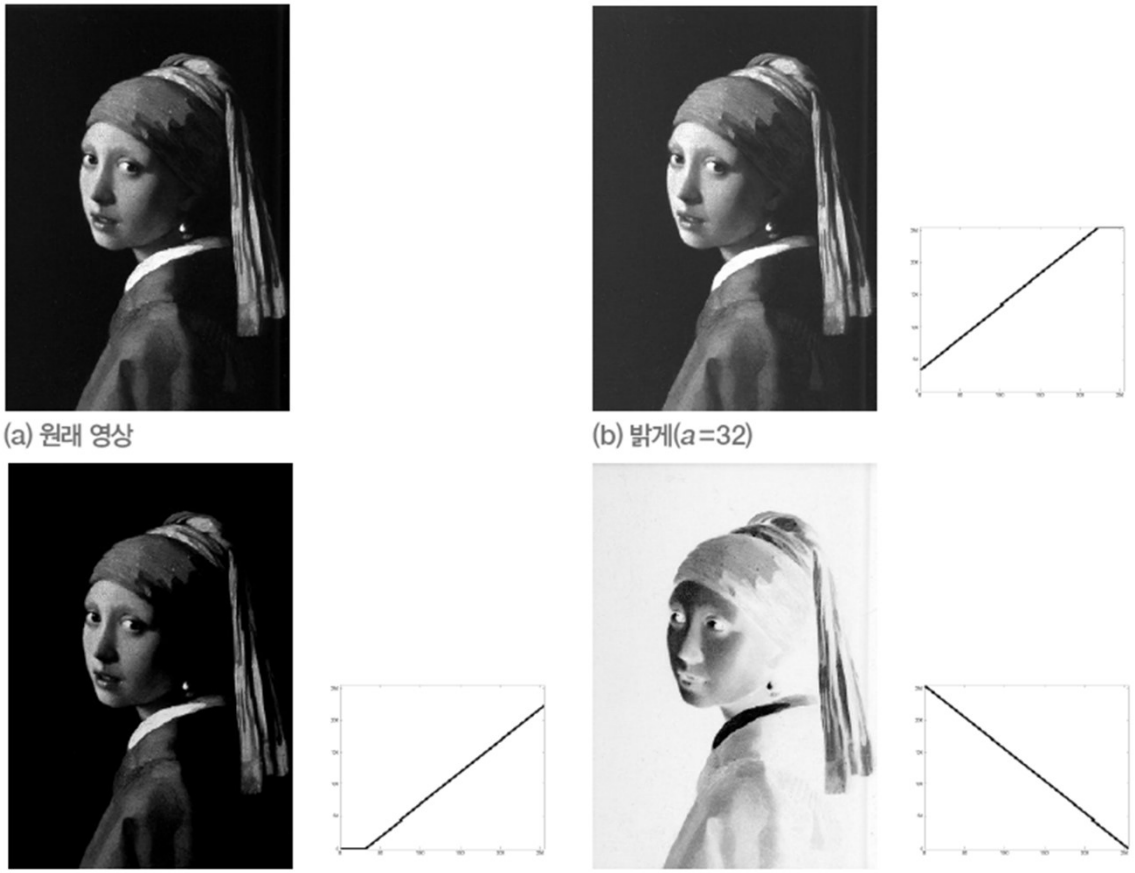
2.4.1 점 연산

- 점 연산 예시

$$f_{out}(j,i) = t(f(j,i))$$

$$= \begin{cases} \min(f(j,i)+a, L-1), & (\text{밝게}) \\ \max(f(j,i)-a, 0), & (\text{어둡게}) \\ (L-1)-f(j,i), & (\text{반전}) \end{cases}$$

  - a: 양수
  - 선형 연산



(a) 원래 영상 (b) 밝게(a=32) (c) 어둡게(a=32) (d) 반전

그림 2-18 여러 가지 선형 점 연산

2.4.1 점 연산

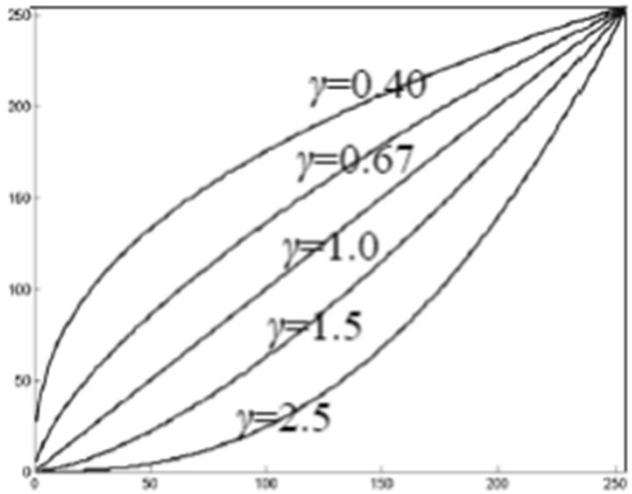
- 점 연산 예시
  - 감마 수정 (Gamma correction)
    - 비선형 연산

$$f_{out}(j,i) = (L-1) \times (\hat{f}(j,i))^{\gamma}$$

이때 $\hat{f}(j,i) = \dfrac{f(j,i)}{(L-1)}$

    - γ가 1보다 작으면 밝아지고
    - 1보다 크면 어두워지는 효과

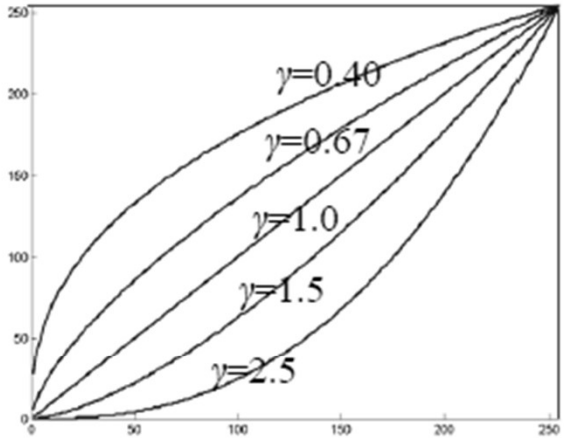
    - 모니터나 프린터 색상 조절에 사용



(a) 감마값에 따른 변환 함수의 모양

2.4.1 점 연산

- 점 연산 예시
  - 감마 수정 (Gamma correction)



(a) 감마값에 따른 변환 함수의 모양

(b) $\gamma = 0.40$

(c) $\gamma = 0.67$

(d) $\gamma = 1.0$(원본 영상)

(e) $\gamma = 1.5$

(f) $\gamma = 2.5$

그림 2-19 **감마 수정**

2.4.1 점 연산

- 점 연산 예시
  - 장면 디졸브 (scene dissolve)
    - $k = 2$인 경우
    - α를 1에서 0.2씩 줄인 결과

$$f_{out}(j, i) = \alpha f_1(j, i) + (1 - \alpha) f_2(j, i)$$

그림 2-21 디졸브 효과

2.4.2 영역 연산

- 상관(correlation)과 컨볼루션(convolution) 학습

- 1차원 예제
- f: 입력 영상
- u: 윈도우 (검출하려는 물체)
- g: 출력 영상

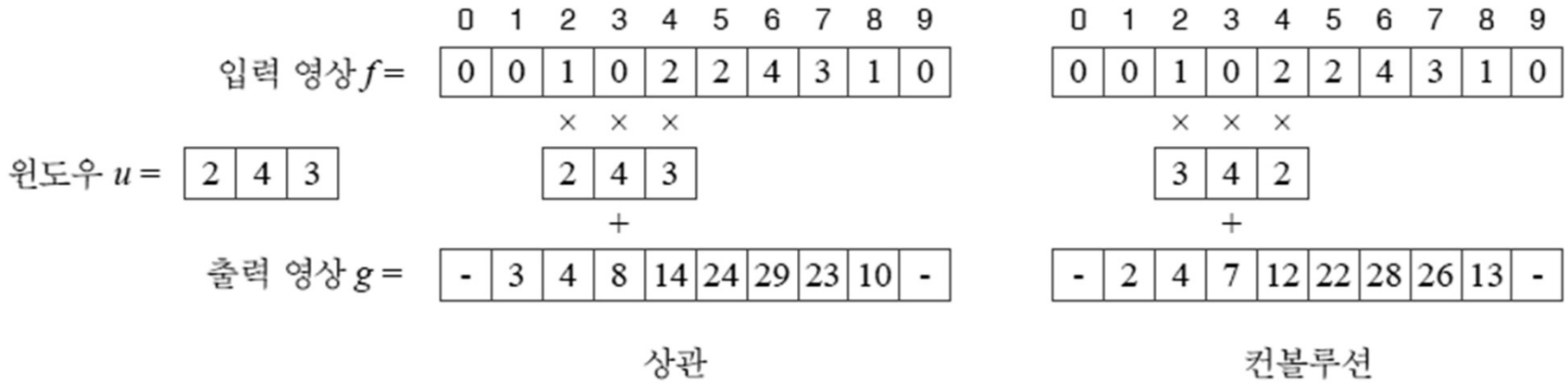


그림 2-22 **상관과 컨볼루션의 원리**

2.4.2 영역 연산

- 상관(correlation)
  - 윈도우를 각각의 위치에 위치시키고 곱의 합을 연산
  - 위치시킬 때, 윈도우의 중간을 기준으로 사용

  - 위치 6에서 최대값을 가지고 해당 위치에 찾고자 하는 물체가 있음
  - 윈도우와 입력 영상이 얼마나 비슷한 지를 측정

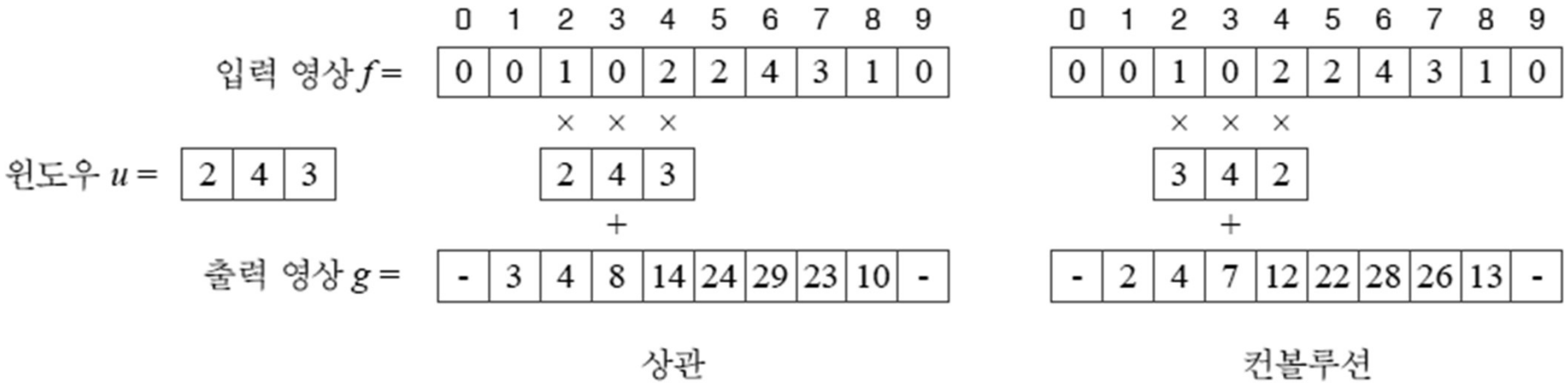


그림 2-22 **상관과 컨볼루션의 원리**

2.4.2 영역 연산

- 상관(correlation)
- 컨볼루션(convolution)
  - 윈도우를 뒤집은 후 상관 적용

- 가장자리 처리: 값을 구하지 않음, 0 또는 가장자리 화소 값으로 가정하기도

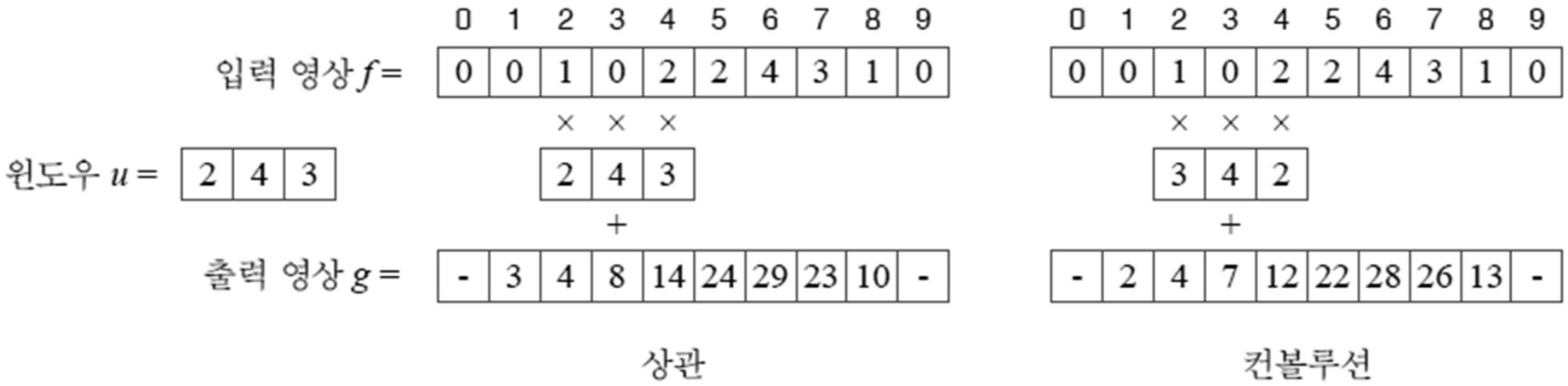


그림 2-22 **상관과 컨볼루션의 원리**

2.4.2 영역 연산

- 2차원으로 확장
  - 상관: 윈도우가 뒤집어진 모양이 결과 영상에
  - 컨볼루션: 윈도우가 그대로 결과 영상에

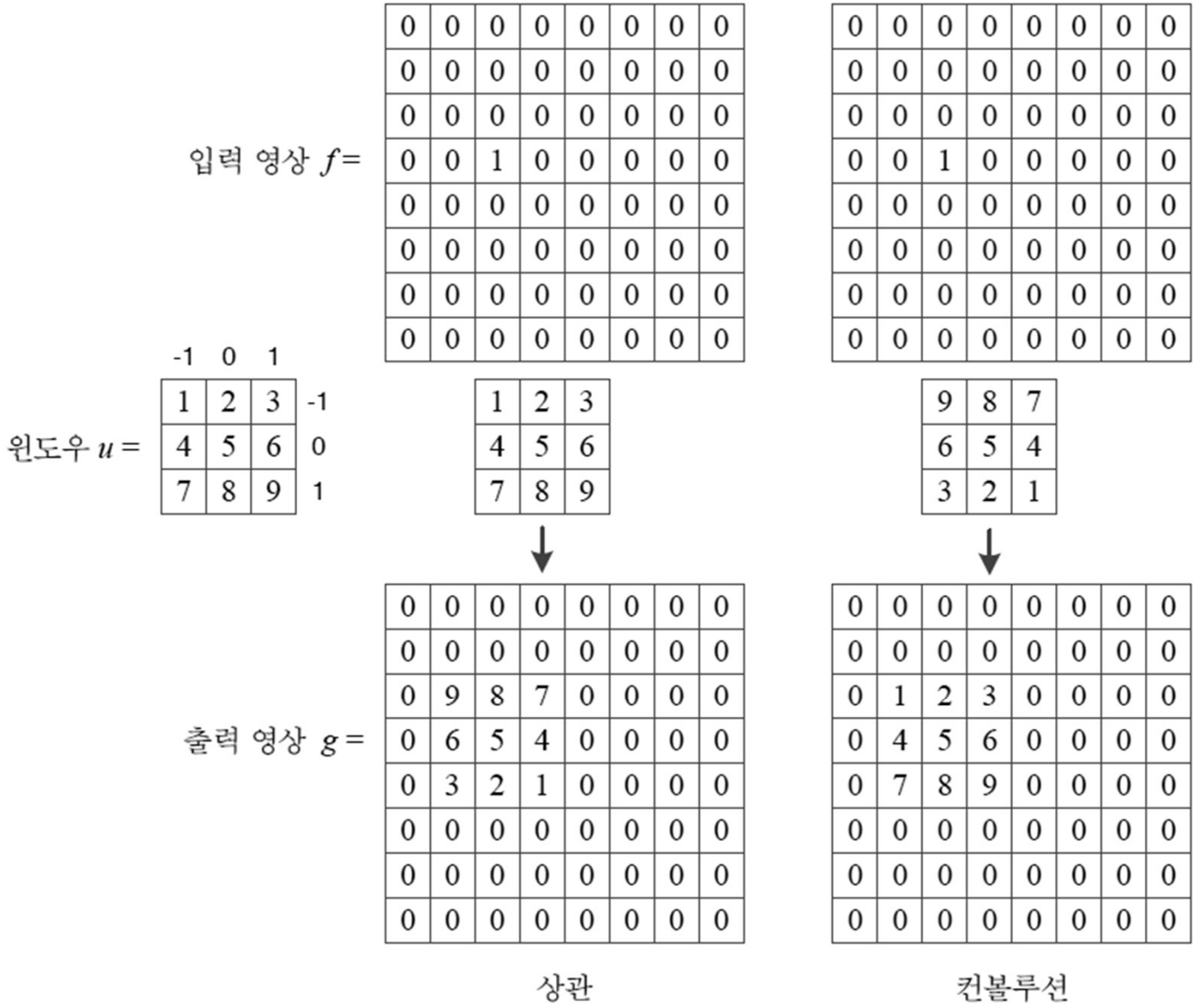


그림 2-23 2차원 상관과 컨볼루션

2.4.2 영역 연산

- 수식으로 쓰면,

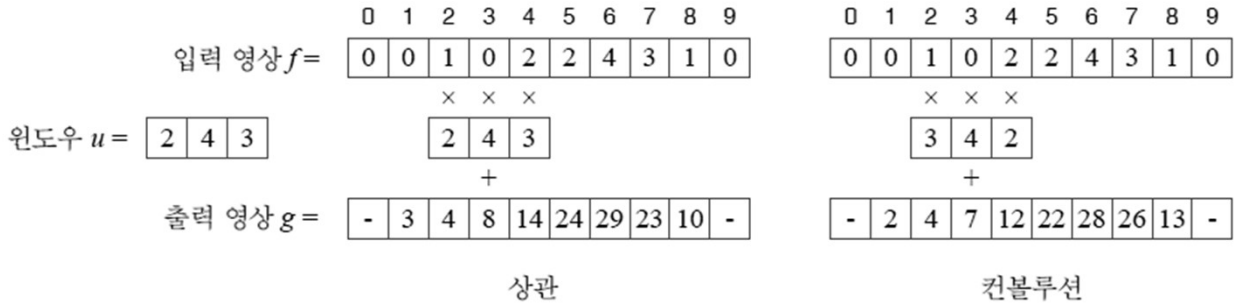


그림 2-22 **상관과 컨볼루션의 원리**

$$\left.\begin{aligned} \text{상관 } g(i) = u \otimes f &= \sum_{x=-(w-1)/2}^{(w-1)/2} u(x) f(i+x) \\ \text{컨볼루션 } g(i) = u \circledast f &= \sum_{j=-(w-1)/2}^{(w-1)/2} u(x) f(i-x) \end{aligned}\right] \text{1차원}$$

$$\left.\begin{aligned} \text{상관 } g(j,i) = u \otimes f &= \sum_{y=-(h-1)/2}^{(h-1)/2} \sum_{x=-(w-1)/2}^{(w-1)/2} u(y,x) f(j+y, i+x) \\ \text{컨볼루션 } g(j,i) = u \circledast f &= \sum_{y=-(h-1)/2}^{(h-1)/2} \sum_{x=-(w-1)/2}^{(w-1)/2} u(y,x) f(j-y, i-x) \end{aligned}\right] \text{2차원}$$

2.4.2 영역 연산

- 상관과 컨볼루션은 윈도우를 뒤집는 것 외에는 동일
- 윈도우가 대칭이면 결과는 동일

- 본 강의에서는 컨볼루션 용어 사용 & 상관 수식 사용
- 윈도우는 마스크(mask), 커널(kernel), 템플릿(template), 필터(filter)라고도 함

$$\left.\begin{aligned} \text{상관 } g(i) = u \otimes f &= \sum_{x=-(w-1)/2}^{(w-1)/2} u(x) f(i+x) \\ \text{컨볼루션 } g(i) = u \circledast f &= \sum_{j=-(w-1)/2}^{(w-1)/2} u(x) f(i-x) \end{aligned}\right] \text{1차원}$$

$$\left.\begin{aligned} \text{상관 } g(j,i) = u \otimes f &= \sum_{y=-(h-1)/2}^{(h-1)/2} \sum_{x=-(w-1)/2}^{(w-1)/2} u(y,x) f(j+y, i+x) \\ \text{컨볼루션 } g(j,i) = u \circledast f &= \sum_{y=-(h-1)/2}^{(h-1)/2} \sum_{x=-(w-1)/2}^{(w-1)/2} u(y,x) f(j-y, i-x) \end{aligned}\right] \text{2차원}$$

2.4.2 영역 연산

- 컨볼루션 예시

- 박스(box) 마스크
  - 화소의 평균
  - 스무딩 효과
    (노이즈 줄여 영상 향상)
- 가우시안 마스크
  - 스무딩 효과
    (노이즈 줄여 영상 향상)
  - 표준편차 0.5인 경우

(a) 원래 영상과 여러 가지 마스크들

박스

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

가우시안

.0000	.0000	.0002	.0000	.0000
.0000	.0113	.0837	.0113	.0000
.0002	.0837	.6187	.0837	.0002
.0000	.0113	.0837	.0113	.0000
.0000	.0000	.0002	.0000	.0000

샤프닝

0	-1	0
-1	5	-1
0	-1	0

수평 에지

1	1	1
0	0	0
-1	-1	-1

수직 에지

1	0	-1
1	0	-1
1	0	-1

모션

.0304	.0501	0	0	0
.0501	.1771	.0519	0	0
0	.0519	.1771	.0519	0
0	0	.0519	.1771	.0501
0	0	0	.0501	.0304

> 박스

> 가우시안

> 샤프닝

> 수평 에지

> 수직 에지

> 모션

(b) 다양한 마스크로 컨볼루션한 영상들

그림 2-24 **다양한 마스크와 컨볼루션 효과**

2.4.2 영역 연산

- 샤프닝 마스크
  - 명암 대비 강조 (에지 강조)
- 수평/수직 에지 마스크
  - y-/x-방향의 미분 값 측정
- 모션 마스크
  - 51×51 중 5×5만 시각화
  - 모션 효과 생성

- 컨볼루션은 선형 연산

박스

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

가우시안

.0000	.0000	.0002	.0000	.0000
.0000	.0113	.0837	.0113	.0000
.0002	.0837	.6187	.0837	.0002
.0000	.0113	.0837	.0113	.0000
.0000	.0000	.0002	.0000	.0000

샤프닝

0	-1	0
-1	5	-1
0	-1	0

수평 에지

1	1	1
0	0	0
-1	-1	-1

수직 에지

1	0	-1
1	0	-1
1	0	-1

모션

.0304	.0501	0	0	0
.0501	.1771	.0519	0	0
0	.0519	.1771	.0519	0
0	0	.0519	.1771	.0501
0	0	0	.0501	.0304

(a) 원래 영상과 여러 가지 마스크들

> 박스

> 가우시안

> 샤프닝

> 수평 에지

> 수직 에지

> 모션

(b) 다양한 마스크로 컨볼루션한 영상들

그림 2-24 **다양한 마스크와 컨볼루션 효과**

2.4.2 영역 연산

- 비선형 연산 예시

- 메디안(median) 필터
  - 중간 값
  - 스무딩 효과
  - 솔트페퍼 잡음(salt-and-pepper noise)에 효과적임
  - 메디안은 가우시안에 비해 에지 보존 효과 뛰어남

(a) 원래 영상 (b) 솔트페퍼 잡음

(c) 가우시안 필터 (d) 메디안 필터

그림 2-25 **가우시안과 메디안 필터의 비교**

2.4.3 기하 연산

- 점 연산: 자기 자신
- 영역 연산: 자기 이웃
- 영상을 회전하거나 특정 영역을 크게 하는 등의 작업이 필요한 경우에는 멀리 있는 화소 값도 고려해야 함
- 동차 좌표와 동차 행렬
  - 동차 좌표 (homogeneous coordinate)
    - 2차원 상의 점 $\boldsymbol{x} = (y, x)$를 3차원 벡터로 표현
    - 차원을 하나 늘리고 값을 1로 설정
    - 세 요소에 같은 값을 곱하면 같은 좌표
    - (3,5) → (3,5,1), (6,10,2), (0.3,0.5,0.1), …

$$\dot{\mathbf{x}} = (y \; x \; 1)$$

2.4.3 기하 연산

- 동차 좌표와 동차 행렬
  - 동차 행렬 (homogeneous matrix)

표 2-1 기하 변환을 위한 동차 행렬

변환	동차 행렬 $\dot{\mathbf{H}}$	설명
이동	$T(t_y, t_x) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_y & t_x & 1 \end{pmatrix}$	y방향으로 t_y, x방향으로 t_x만큼 이동
회전	$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$	원점을 중심으로 시계방향으로 θ만큼 회전
크기	$S(s_y, s_x) = \begin{pmatrix} s_y & 0 & 0 \\ 0 & s_x & 0 \\ 0 & 0 & 1 \end{pmatrix}$	y방향으로 s_y, x방향으로 s_x만큼 확대

2.4.3 기하 연산

- 동차 행렬을 이용한 기하 변환

예를 들어, 어떤 점을 y방향으로 3, x방향으로 2만큼 이동시키는 동차 행렬 $\dot{\mathbf{H}}$는 다음과 같다. 식 (2.16)은 동차 좌표 $\dot{\mathbf{x}}$와 동차 행렬 $\dot{\mathbf{H}}$를 이용한 기하 변환이다.

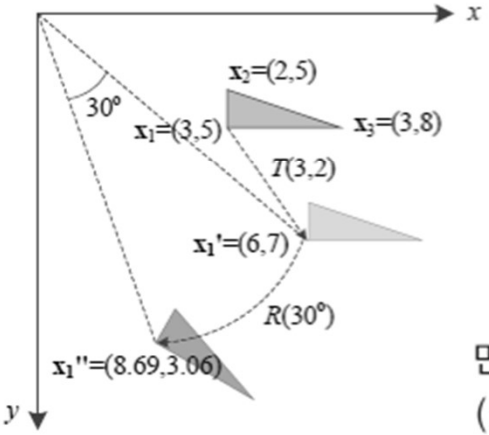
$$\dot{\mathbf{H}} = T(3,2) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix}$$

$$\dot{\mathbf{x}}' = (y'x'1) = \dot{\mathbf{x}}\dot{\mathbf{H}} = (y\ x\ 1)\begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{pmatrix} \tag{2.16}$$

풀어쓰면, $y' = a_{11}y + a_{21}x + a_{31}$, $x' = a_{12}y + a_{22}x + a_{32}$

예제 2-3 **동차 행렬을 이용한 기하 변환**

[그림 2-26]의 삼각형을 y방향으로 3, x방향으로 2만큼 이동시킨 후 30° 회전 시켜보자.



먼저, 이동 변환을 구하려면 $T(3,2)$가 필요하다. 꼭지점 $\mathbf{x}_1=(3,5)$를 동차 좌표로 확장하여 $\dot{\mathbf{x}}_1=(3,5,1)$을 만들고 식 (2.16)의 연산을 적용한다.

$$T(3,2)=\begin{pmatrix}1&0&0\\0&1&0\\3&2&1\end{pmatrix},\quad \dot{\mathbf{x}}_1'=(3\quad 5\quad 1)\begin{pmatrix}1&0&0\\0&1&0\\3&2&1\end{pmatrix}=(6\quad 7\quad 1)$$

연산의 결과로 $\dot{\mathbf{x}}_1'=(6\ 7\ 1)$을 얻었는데, 마지막 요소를 제거하여 2차원 좌표로 바꾸면 $\mathbf{x}_1'=(6,7)$이 된다. 나머지 두 점 $\mathbf{x}_2$와 $\mathbf{x}_3$도 같은 과정으로 변환한 후 이동한 삼각형을 그려보면 가운데 삼각형과 같다. 이제 이동한 삼각형을 30° 회전 시켜보자. 회전을 계산하는 데 필요한 행렬 $R(30°)$를 꼭지점 $\mathbf{x}_1'$에 적용하면, 다음과 같이 $\mathbf{x}_1''=(8.6962, 3.0622)$를 얻는다. 나머지 두 점을 계산하고 결과를 그려보면 맨 아래 삼각형과 같다.

$$(6\quad 7\quad 1)\begin{pmatrix}\cos 30^\circ & -\sin 30^\circ & 0\\ \sin 30^\circ & \cos 30^\circ & 0\\ 0 & 0 & 1\end{pmatrix}=(6\quad 7\quad 1)\begin{pmatrix}0.866 & -0.500 & 0\\ 0.500 & 0.866 & 0\\ 0 & 0 & 1\end{pmatrix}=(8.6962\quad 3.0622\quad 1)$$

2.4.3 기하 연산

- 왜 동차 좌표를 사용하나?
  - 복합 변환을 이용한 계산 효율
  - 행렬의 곱셈 결합법칙 $A(BC) = (AB)C$

  - 수 만 개의 점에 대하여 이동과 회전을 적용하면
  - 각 점에 대해서 3x3 행렬 곱셈을 두 번씩 수행해야 함.

  - 복합 변환을 활용하면 미리 이동과 회전을 결합한 행렬을 구해두고
  - 이를 활용하여 각 점에 대하여 3x3 행렬 곱셈을 한번만 수행하면 됨.

표 2-1 기하 변환을 위한 동차 행렬

변환	동차 행렬 $\dot{\mathbf{H}}$
이동	$T(t_y, t_x) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_y & t_x & 1 \end{pmatrix}$
회전	$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$
크기	$S(s_y, s_x) = \begin{pmatrix} s_y & 0 & 0 \\ 0 & s_x & 0 \\ 0 & 0 & 1 \end{pmatrix}$

2.4.3 기하 연산

이동 $$T(t_y, t_x) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_y & t_x & 1 \end{pmatrix}$$

회전 $$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

예제 2-4 **복합 기하 변환**

[예제 2-3]의 두 단계 변환을 효율적으로 해 보자. 변환에 필요한 행렬 $T(3, 2)$와 $R(30°)$를 곱하면 다음과 같다.

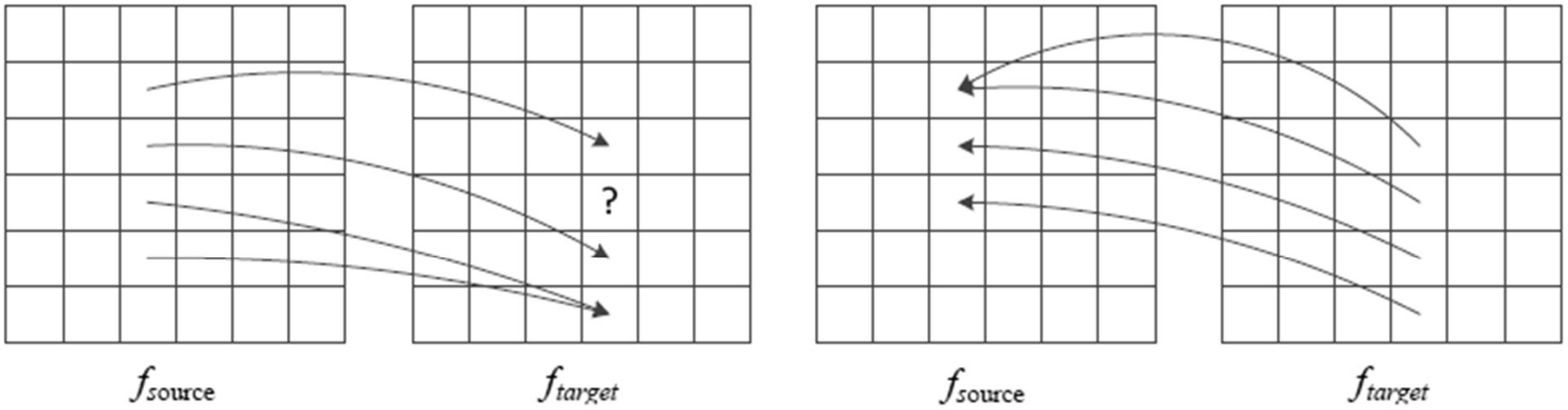
$$T(3,2)R(30^\circ) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} 0.866 & -0.500 & 0 \\ 0.500 & 0.866 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0.866 & -0.500 & 0 \\ 0.500 & 0.866 & 0 \\ 3.5981 & 0.2321 & 1 \end{pmatrix}$$

이제 이 행렬을 원래 삼각형의 세 개의 꼭지점 각각에 대해 적용하면, 최종 변환된 삼각형을 얻을 수 있다. 예를 들어, 꼭지점 $\mathbf{x}_1 = (3, 5)$에 적용한 결과는 다음과 같다. [예제 2-3]에서 행렬 곱을 두 번 한 것과 결과가 동일하다.

$$(3 \quad 5 \quad 1) \begin{pmatrix} 0.866 & -0.500 & 0 \\ 0.500 & 0.866 & 0 \\ 3.5981 & 0.2321 & 1 \end{pmatrix} = (8.6962 \quad 3.0622 \quad 1)$$

2.4.3 기하 연산

- 영상에 적용
  - 디지털 영상의 좌표는 항상 정수
  - 전방 변환
    - 입력 영상에 있는 각 화소에 대해 목표 영상의 변환 위치를 계산하고 복사
    - 여러 곳에서 값을 받는 화소도 있고 아예 못 받는 화소도 있음 -> 심각한 문제
    - 인공적으로 발생한 왜곡: 에일리어싱(aliasing)이라고 함

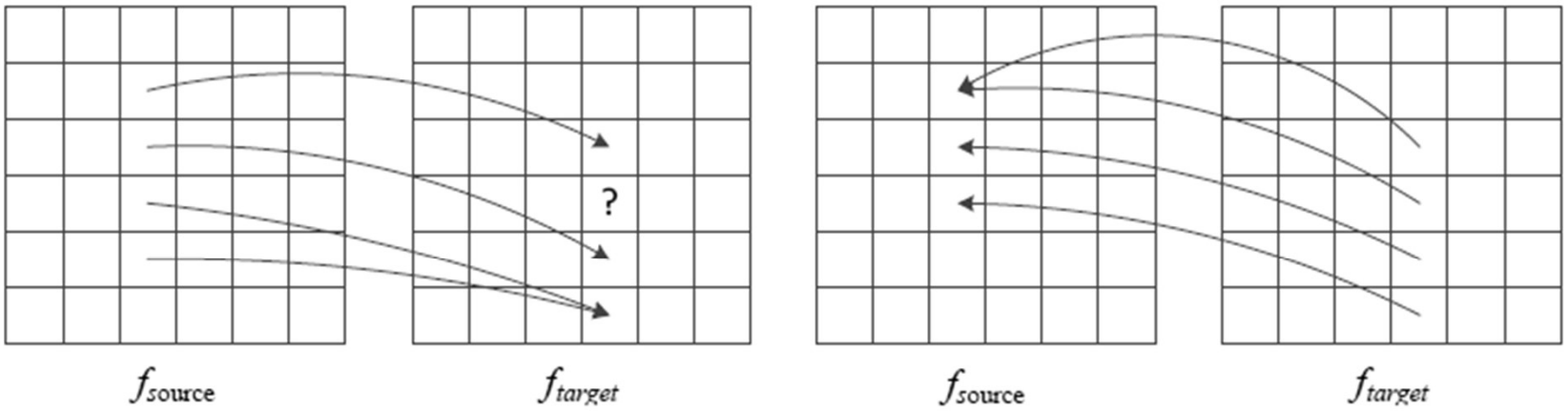


(a) 전방 변환 (b) 후방 변환

그림 2-27 후방 기하 변환의 안티 에일리어싱 효과

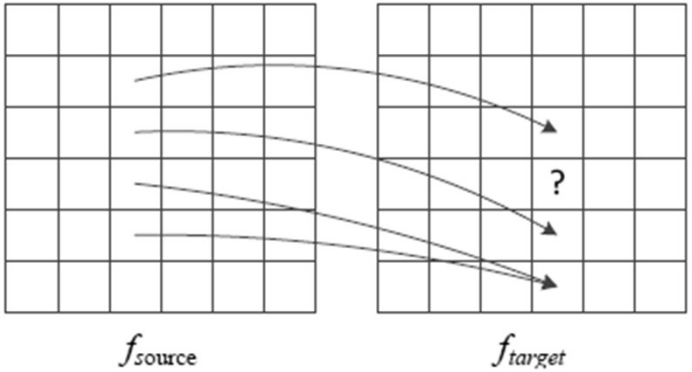
2.4.3 기하 연산

- 전방 변환
  - 인공적으로 발생한 왜곡: 에일리어싱(aliasing)이라고 함
- 후방 변환
  - 을 이용한 안티 에일리어싱(anti-aliasing)
  - 목표 영상의 화소가 자신의 값을 찾아 나서는 것
  - 변환 행렬 $\dot{H}$의 역행렬 $\dot{H}^{-1}$을 활용



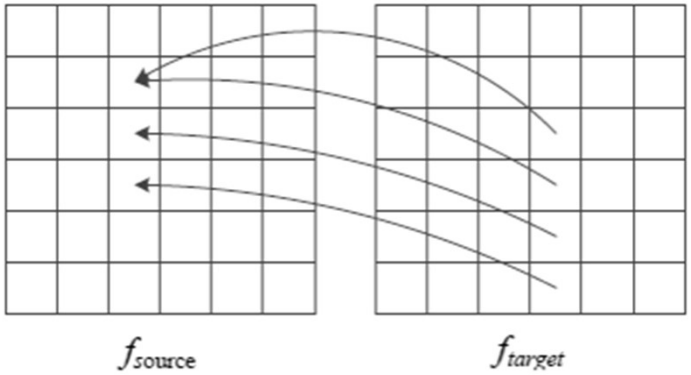
(a) 전방 변환 (b) 후방 변환

그림 2-27 후방 기하 변환의 안티 에일리어싱 효과



(a) 전방 변환

그림 2-27 **후방 기하 변환의 안티 에일리어싱 효과**



(b) 후방 변환

알고리즘 2-7 **전방 기하 변환**

입력 : 영상 $f_{source}(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$, 변환 행렬 $\dot{\mathbf{H}}$

출력 : 기하 변환된 영상 $f_{target}(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

```
for (j=0 to M-1)
  for (i=0 to N-1) {
    (j, i)에 Ḣ를 적용하여 변환된 점 (j', i')를 구한다. 실수는 반올림하여 정수로 만든다.
    f_target(j', i') = f_source(j, i);   // 영상 공간을 벗어난 점은 무시한다.
}
```

알고리즘 2-8 **후방 기하 변환**

입력 : 영상 $f_{source}(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$, 변환 행렬 $\dot{\mathbf{H}}$

출력 : 기하 변환된 영상 $f_{target}(j, i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

```
for (j=0 to M-1)
  for (i=0 to N-1) {
    (j, i)에 Ḣ^-1을 적용하여 변환된 점 (j', i')를 구한다. 실수는 반올림하여 정수로 만든다.
    f_target(j, i) = f_source(j', i');   // 영상 공간을 벗어난 점은 무시한다.
}
```

2.4.3 기하 연산

- 보간(interpolation)에 의한 안티 에일리어싱
  - 앞에서는 실수 좌표를 반올림하여 정수로 변환
    : 에일리어싱 발생
  - 주위 화소 값을 이용한 보간으로 안티 에일리어싱

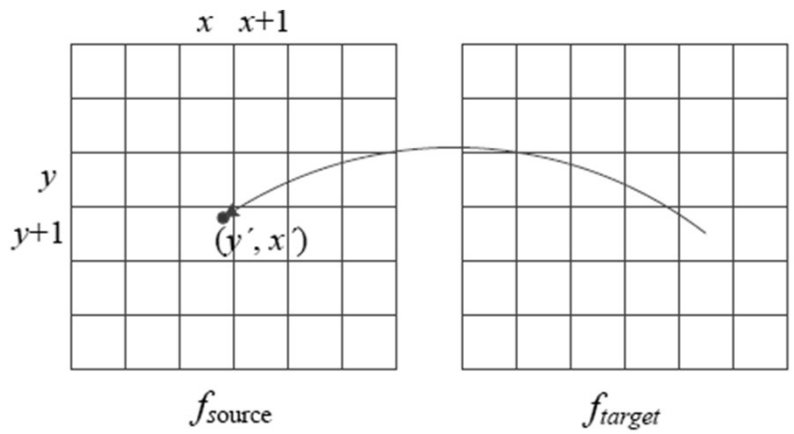


그림 2-28 기하 변환한 점의 좌표

2.4.3 기하 연산

- 보간(interpolation)에 의한 안티 에일리어싱
- 양선형 보간
  - 1차원으로 생각
    - (좌측) 해당 영역 확대
    - (우측) 명암 값을 높이로 간주

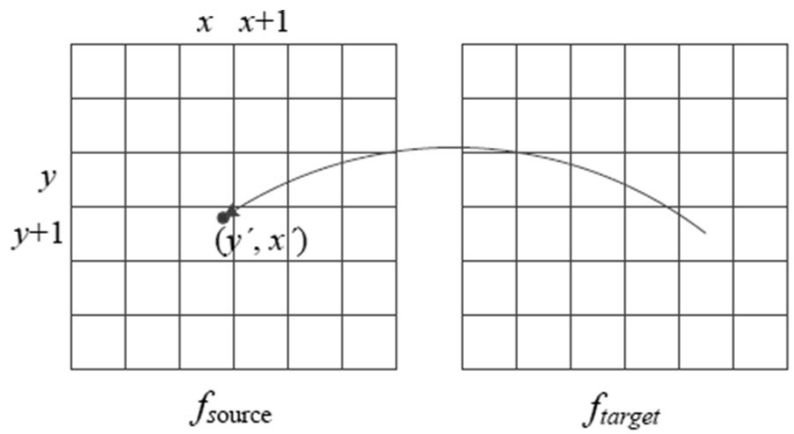


그림 2–28 기하 변환한 점의 좌표

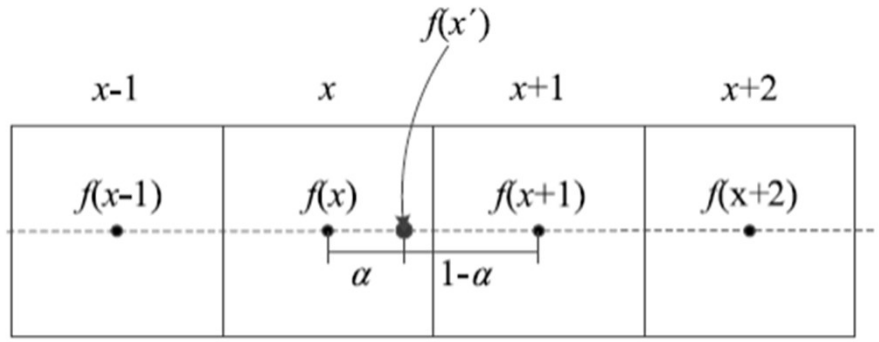
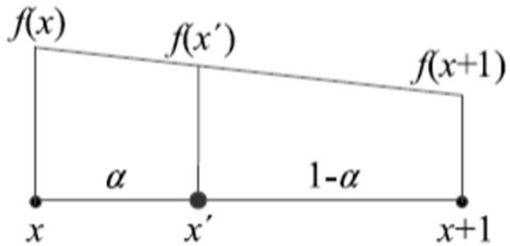


그림 2–29 1차원에서 보간식 유도



$$f(x') = (1 - \alpha)f(x) + \alpha f(x + 1)$$

2.4.3 기하 연산

$$f(y,x') = (1-\alpha)f(y,x) + \alpha f(y,x+1)$$
$$f(y+1,x') = (1-\alpha)f(y+1,x) + \alpha f(y+1,x+1)$$
$$f(y',x') = (1-\beta)f(y,x') + \beta f(y+1,x')$$

- 양선형 보간
  - 2차원
  - 행 y와 $y+1$에서 x방향으로 interpolation
  - x'에서 y방향으로 interpolation

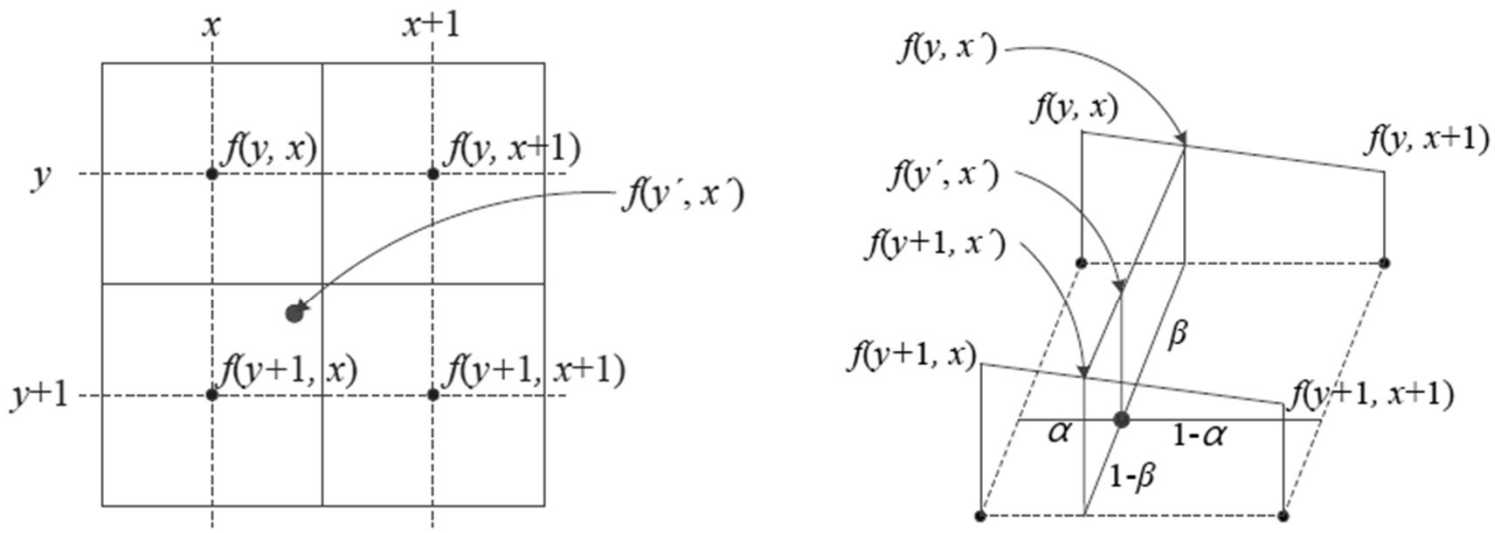


그림 2-30 2차원에서 보간식 유도

2.4.3 기하 연산

- 최근접 이웃 (nearest neighbor)
  - 가장 가까운 점의 명암 값을 채택
  - 반올림하여 처리

- 양선형 보간 (bilinear interpolation)
  - 가장 가까운 네 개의 점을 활용
  - 시간을 더 오래 걸리지만 품질을 더 나음
  - 선형 보간(linear interpolation)을 두 방향으로 함
  - 양방향 선형 보간

- 양 3차 보간 (bicubic interpolation)
  - 비선형 보간 방법
  - 16개의 이웃 화소와 3차 함수를 사용

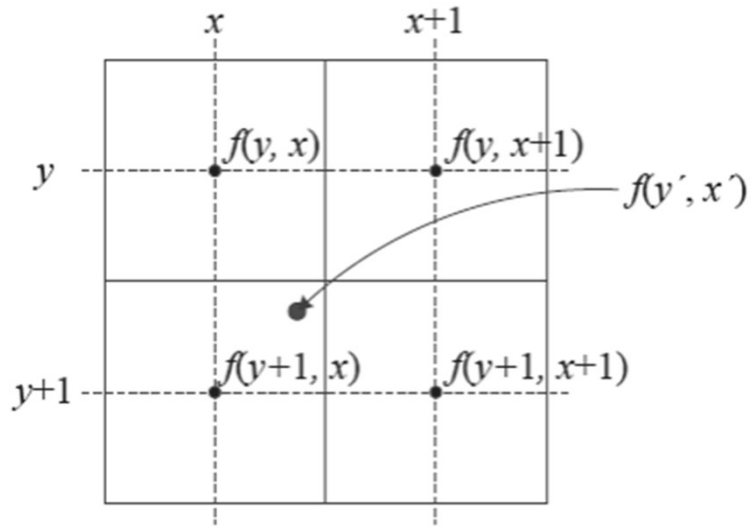


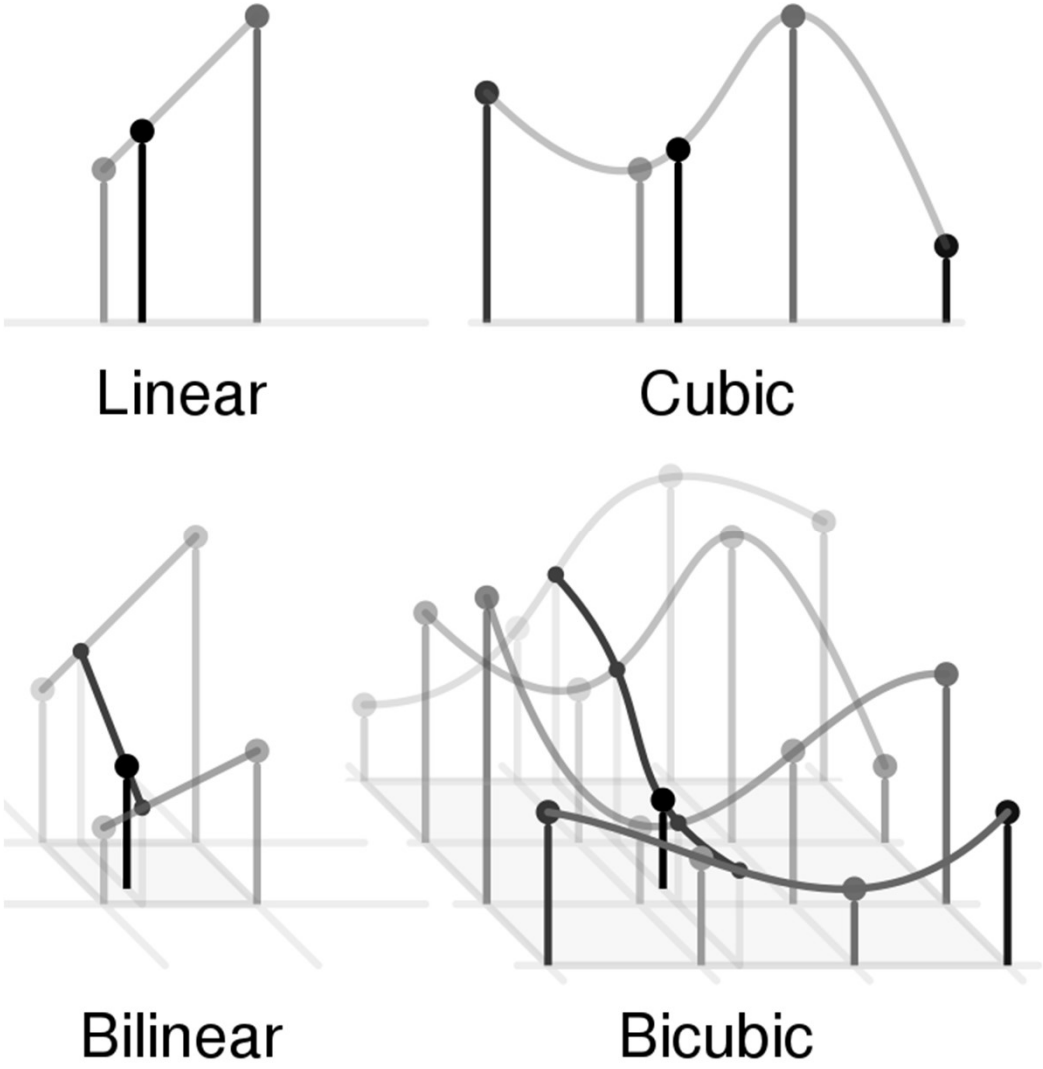
그림 2-30 2차원에서 보간식 유도

2.4.3 기하 연산

- 최근접 이웃 (nearest neighbor)
  - 가장 가까운 점의 명암 값을 채택
  - 반올림하여 처리

- 양선형 보간 (bilinear interpolation)
  - 가장 가까운 네 개의 점을 활용
  - 시간을 더 오래 걸리지만 품질을 더 나음
  - 선형 보간(linear interpolation)을 두 방향으로 함
  - 양방향 선형 보간

- 양 3차 보간 (bicubic interpolation)
  - 비선형 보간 방법
  - 16개의 이웃 화소와 3차 함수를 사용



2.4.3 기하 연산

- 최근접 이웃
  (nearest neighbor)
- 양선형 보간
  (bilinear interpolation)
- 양 3차 보간
  (bicubic interpolation)

(a) 원래 영상과 조각 영상

> 최근접 이웃 > 양선형 보간 > 양 3차 보간

(b) 10° 회전한 영상

그림 2-31 **영상 보간**