



# 디지털 영상 처리

2021년 2학기 강의 교재  
서경대학교 컴퓨터공학과 김진헌

# II 부

---

## 영상 처리 알고리즘

### 내용

#### 5장 공간 필터링과 가우시안 블러링

6장 에지 검출과 미분

7장 화질 개선 알고리즘

8장 칼라 영상 처리

9장 푸리에 변환

10장 DCT와 영상 압축

# 5

---

## 공간 필터링과 가우시안 블러링

### 내용

- 5.1 필터링 동작
- 5.2 평균화 커널을 이용한 필터링
- 5.3 가우시안 함수와 가우시안 블러링
- 5.4 검토사항

## 5.1 필터링 동작



필터링(filtering)은 커피 원두에서 필요한 커피 성분을 필터를 사용하여 추출해 내는 동작이다. 영상에서 필터링이라 함은 영상에서 필요한 성분을 커널(kernel)을 이용해 추출해 내는 동작을 의미한다.

그림 5.1.1은 입력 영상을 주어진 3개의 커널을 적용하여 얻어진 결과를 보인 것이다. 그림에서 어떤 커널을 사용하는가에 따라 필터링 결과가 달라지는 것을 알 수 있다.

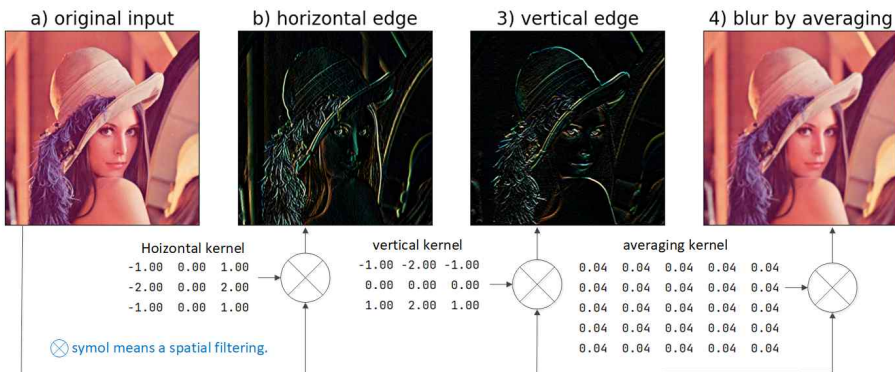


그림 5.1.1 커널을 영상에 적용하여 필터링한 사례

### ◇ 커널이란?

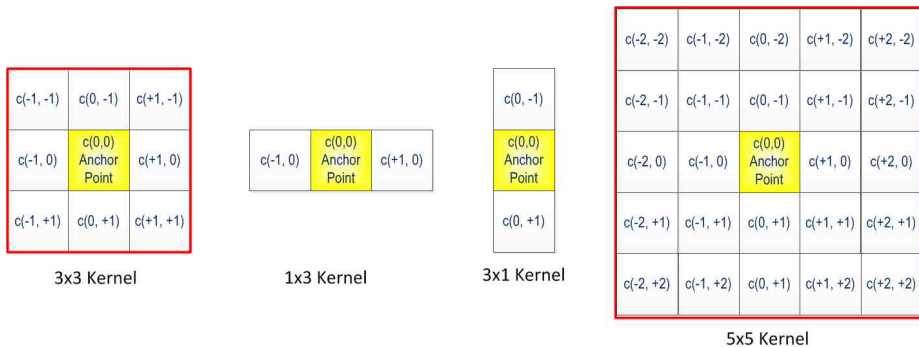


그림 5.1.2 커널의 사례

그림 5.1.2에 다양한 크기의 커널의 사례를 보였다. 커널은 NxM의 크기의 2차원 매트릭스로 되어 있는데 그림에서 보이듯 'NxN' 정방형 매트릭스 혹은 1xN 혹은 Nx1 매트릭스가 주로 사용된다<sup>1)</sup>.

그림에서  $c(x,y)$ 는 좌표 $(x,y)$ 에서의 계수(coefficient)를 나타내며  $(x,y)$ 는 각각 중심좌표(이른바 anchor point)에 대한 상대적 위치를 나타낸다. 필터링의 목적에 따라 용도에 맞게 이들 필터계수와 필터의 크기가 달라진다.

### ◇ 공간 필터링(spatial filtering)이란?

$c_1$ (s-1, t-1)	$c_2$ (s, t-1)	$c_3$ (s+1, t-1)
$c_4$ (s-1, t)	$c_5$ (s, t)	$c_6$ (s+1, t)
$c_7$ (s-1, t+1)	$c_8$ (s, t+1)	$c_9$ (s+1, t+1)

공간 필터링<sup>2)</sup>은 필터 커널과 영상과의 선형 필터링(linear filtering) 연산으로 이루어진다. 그림 5.1.3과 같은 3x3 커널에 대한 코릴레이션 연산 값,  $g(x,y)$ 은 식 5.1.1과 같이 이루어진다. 여기서  $c_1 \sim c_9$ 은 커널의 계수값을 의미한다.

그림 5.1.3 3x3 커널

$$g(x,y) = c_1 \cdot f(x-1,y-1) + c_2 \cdot f(x,y-1) + c_3 \cdot f(x+1,y-1) + c_4 \cdot f(x-1,y) + c_5 \cdot f(x,y) + c_6 \cdot f(x+1,y) + c_7 \cdot f(x-1,y+1) + c_8 \cdot f(x,y+1) + c_9 \cdot f(x+1,y+1) \quad (\text{식 5.1.1})$$

### 함수 설명

**$f(x,y)$ :** 입력영상을  $x$  좌표,  $y$  좌표에 따른 함수로 나타낸 것이다. 특정좌표가 정해지면  $f(x,y)$ 가 출력하는 함수값은 그 지점의 계조값(혹은 RGB 값)이다.

**$g(x,y)$ :** 입력 영상을 커널을 사용하여 코릴레이션한 결과 영상. 역시 입력영상과 같은  $(x,y)$  좌표로 표현한다. 이것은 입력의 특정 좌표에 대해 같은 위치의 출력값이 연산된다는 것을 뜻한다.

### ◇ 코릴레이션 연산

좌표  $(x,y)$ 의 화소 값이  $f(x,y)$ 인 영상에 대해  $M \times N$ 의 크기의 커널,  $c(s,t)$ 를 적용하여 코릴레이션 연산 값,  $g(x,y)$ 를 구하는 과정을 일반화하여 표현하면 식 5.1.2와 같다. 이때 커널은  $M=2a+1$ ,  $N=2b+1$ 의 크기를 갖는다.

1)  $N$ 은 양의 정수인 홀수만 사용된다. 즉,  $N=1, 3, 5, 7, 9, 11, \dots$

2) 영상처리에서 필터링은 공간 필터링(spatial domain filtering)과 주파수 공간에서 이루어지는 필터링(frequency domain filtering)으로 나뉜다.

$$g(x,y) = \sum_{s=-a}^a \sum_{t=-b}^b c(s,t) \cdot f(x+s,y+t) \quad (\text{식 5.1.2})$$

이러한 코릴레이션 연산은 영상의 모든 화소에 대해 순차적으로 적용한다. 그 사례를 보이기 위해 그림 5.1.4에는 (a) 영상의 사례와 (b) 3x3 커널을 사례를 들어 처리과정을 설명하기로 한다. 그림 (a)의 숫자는 화소의 계조 값을 표현한 것이며, (b)의 숫자는 커널의 계수를 보인 것이다.

94	121	138	71	53	65
74	26	84	103	126	145
191	236	87	189	92	125
148	152	172	93	101	251

1	2	3
4	5	6
7	8	9

(a) Array of Pixel Values(=Image)

(b) Convolution Kernel Coefficients

그림 5.1.4 임의의 영상과 커널 사례

그림 5.1.5에 코릴레이션 연산 절차와 그 결과를 보였다.

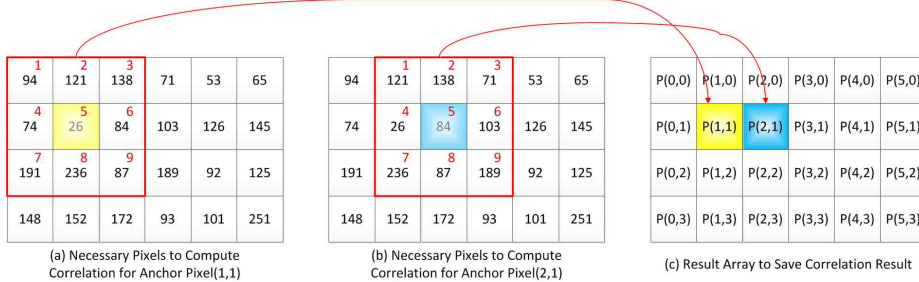


그림 5.1.5 코릴레이션 연산을 위한 소요 화소 및 결과 저장 방법

그림에서 (a), (b)는 중심화소를 중심으로 각각 3x3 주변의 화소가 연산에 활용되며 연산 결과는 (c)와 같은 어레이에 별도로 저장된다. 코릴레이션의 결과 값 P(1,1), P(2,1)은 다음과 같이 계산된다.

$$P(1,1) = 1 \cdot 94 + 2 \cdot 121 + 3 \cdot 138 + 4 \cdot 74 + 5 \cdot 26 + 6 \cdot 84 + 7 \cdot 191 + 8 \cdot 236 + 9 \cdot 87$$

$$P(2,1) = 1 \cdot 121 + 2 \cdot 138 + 3 \cdot 71 + 4 \cdot 26 + 5 \cdot 84 + 6 \cdot 103 + 7 \cdot 236 + 8 \cdot 87 + 9 \cdot 189$$

이와 같은 연산은 그림 5.1.6과 같이 영상의 전 영역에 대하여 행해진다.

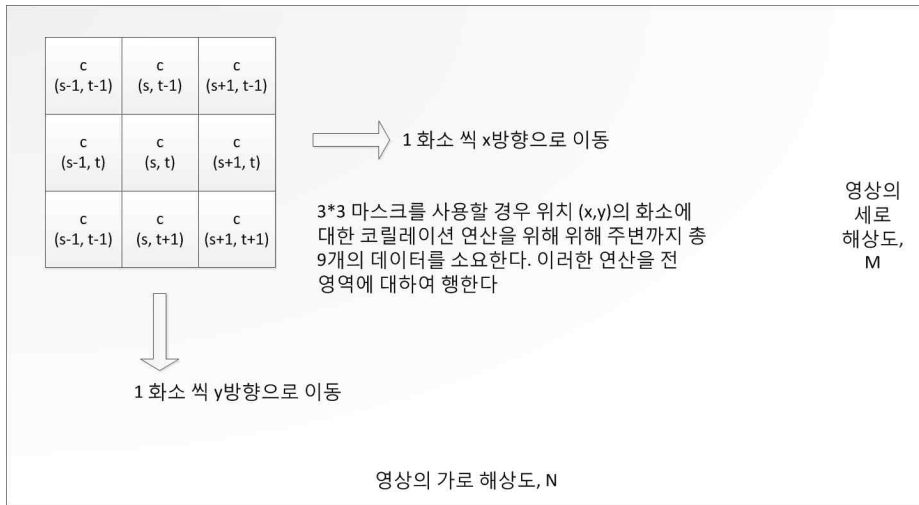


그림 5.1.6 코릴레이션 동작의 적용범위

## ◇ 컨벌루션(Convolution)

코릴레이션과 유사한 연산으로 컨벌루션 연산이 있다. 컨벌루션은 필터 함수를  $x, y$ 로 한 번씩 대칭하여 코릴레이션 한 것과 같다. 즉, 필터를 중심으로 대칭하여 코릴레이션한 것이라 생각할 수 있다<sup>3)</sup>.

1	2	3	9	8	7
4	5	6	6	5	4
7	8	9	3	2	1
Kernel A			Kernel B		

그림 5.1.7 코릴레이션과 컨벌루션 커널

그림 5.1.7의 경우 (a)의 커널을 사용하여 컨벌루션 동작을 시행하고자 한다면 (b)의 커널을 이용하여 코릴레이션하면 된다. 즉, 어떤 커널을 이용하여 컨벌루션한 결과를 얻고자 한다면 그 커널을 중심 대칭하여 만든 커널을 이용하여 코릴레이션 하면 된다는 것이다.

결과 1 = Kernel A로 코릴레이션 = Kernel B로 컨벌루션  
 결과 2 = Kernel B로 코릴레이션 = Kernel A로 컨벌루션

여기서 만약 중심점을 기준으로 서로 마주 보고 있는 계수값이 같

3) 코릴레이션(correlation)과 컨벌루션(convolution) 연산은 두 연산의 수학적 연산법은 다르지만 영상처리에서는 거의 대부분 같은 결과를 얻기 때문에 혼용해서 사용되는 경우도 적지 않다.

다면 위 2개의 결과는 완전히 같다.

$$\text{결과 1} = \text{결과 2}$$

만약 중심점을 기준으로 서로 마주 보고 있는 계수값이 각각 서로 부호만 반대이고 크기는 같다면 두 결과의 차이는 부호만 달라질 뿐이다.

$$\text{결과 1} = -\text{결과 2}$$

컨벌루션과 코릴레이션은 영상처리 세계에서 구분하지 않고 사용되는 경우가 많다. 이렇게 연산하는 방법이 다른데도 혼용되어 쓰이는 이유는 디지털 영상처리에서는 대부분의 커널이 중심점 대칭인 커널을 사용하기 때문에 어떤 연산을 하여도 사실상 같은 결과를 얻기 때문에 일어나는 현상으로 볼 수 있다. 컨벌루션은 추후 서술할 주파수 영역의 필터링에서 사용된다.



## 5.2 평균화 커널을 이용한 필터링

spatial filtering은 커널과 영상에 대한 코릴레이션으로 이루어지며 커널을 잘 정의하면 여러가지 다양한 결과 영상을 얻을 수 있다. 만약 계수의 값이 모두 같고 그 모든 계수들의 합이 1인 커널을 사용하면 어떤 결과를 얻을 수 있을까?

중심점 위치의 필터링 값을 얻기 위해 커널 내에 위치한 인접화소의 값을 동일한 비중으로 가중평균하는 동작이 수행될 것이다. 예상되는 결과는 흐릿한 영상이 얻어지는데 이 흐릿해지는 정도는 커널의 크기가 커질수록 심해질 것이다<sup>4)</sup>. 극단적으로 커널의 크기가 1이고, 계수도 1이라면 원본과 같은 영상이 얻어질 것이다.



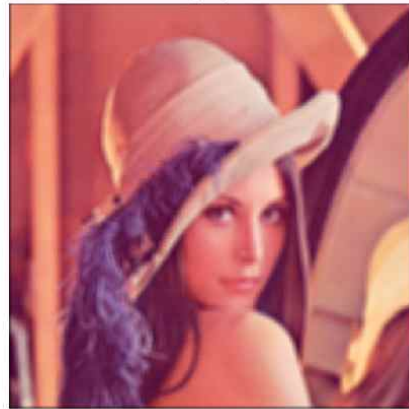
### program sf1\_averaging\_filter2D.py 실습 1

목표	코릴레이션 수행하기 - <b>filter2D()</b>
실습 1, 2	평균화 처리를 처리하는 5x5 혹은 9x9 커널을 만들어 이것으로 입력영상에 대해 코릴레이션 동작을 수행한 결과를 아래와 같이 출력하시오.

Original



Averaging: N=9



### □ 평균화 커널 만들기

`N=5; kernel = np.ones((N, N), np.float32)`

4) 이런 커널을 평균화 커널이라 할 수 있는데 OpenCV 용어로는 이런 커널을 box filter라고 한다.

```
kernel /= np.sum(kernel) # 커널의 합으로 나누어 커널의 합이 1이 되도록 정규화한다.
```

- numpy의 np.ones() 함수는 지정된 크기(5x5)의 매트릭스를 생성하면서 내부 원소의 값을 1로 선언한다.

- np.sum() 함수는 다차원 행렬에 대한 모든 원소의 값을 합산한다. 내부 원소가 1인 행렬에 대해 총합으로 나누는 정규화를 행하는 이유는 각 원소의 값의 합이 1이 되도록 하기 위한 것이다. 그래야 영상의 계조 값이 증가하지 않게 될 것이다.

```
np.set_printoptions(precision=2) # 소수 이하 2자리까지 출력. 이하 0이면 출력 안한다.
```

```
print('kernel=Wn', kernel) # 커널의 원소 값을 출력
```

```
print('np.sum(kernel)=', np.sum(kernel)) # 커널의 총합을 확인.
```

- 커널의 각 원소의 값을 소수 이하 2자리까지만 보인다.

- 원소의 총합을 출력하여 1이 되는 것을 확인한다.

```
kernel=
[[0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]]
np.sum(kernel)= 1.0
```

## □ 코릴레이션 행하기

```
img = cv.imread(FullName) # 원본 영상 읽기
```

```
dst = cv.filter2D(img, -1, kernel) # 코릴레이션 행하기
```

```
print('type(img)=', type(img), ' | img.shape=', img.shape, ' | img.dtype=', img.dtype )
```

```
print('type(dst)=', type(dst), ' | dst.shape=', dst.shape, ' | dst.dtype=', dst.dtype )
```

```
type(img)= <class 'numpy.ndarray'> | img.shape= (512, 512, 3) | img.dtype= uint8
type(dst)= <class 'numpy.ndarray'> | dst.shape= (512, 512, 3) | dst.dtype= uint8
```

- 영상과 커널에 대한 코릴레이션 연산은 화소 단위로 이동하면서 전 영역에 대해 이루어져야 한다. 이 같은 반복 연산을 시행하는 함수가 filter2D()이다. 이 함수는 OpenCV 패키지에 내장된 함수로서 다음과 같은 형식을 가진다. 여기서 입력 영상은 src, 반환 값은 dst이다.

```
dst = cv.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]])
```

<b>src</b>	입력 영상.
<b>dst</b>	필터링된 출력 영상. 소스 영상과 같은 크기와 같은 채널 수를 갖는다. -> dst.shape=src.shape.
<b>ddepth</b>	출력 영상의 희망하는 depth. ddepth=-1이면 입력 영상과 같은 depth를 갖는다. -> -1이면 dst.dtype=src.dtype. 아래 참조 : <a href="#">Depth Combinations</a>
<b>kernel</b>	convolution kernel (or rather a correlation kernel) <sup>5)</sup> . a single-channel floating point matrix-> 2차원 행렬.
<b>anchor</b>	커널의 상대적 중심점의 위치. default value (-1,-1) means that the anchor is at the kernel center.
<b>delta</b>	코릴레이션 연산을 수행하고 난 후 처리 결과, dst에 더해질 값.
<b>borderType</b>	pixel extrapolation method, see <a href="#">BorderTypes</a>

#### Depth combinations

Input depth ( <a href="#">src.depth()</a> )	Output depth (ddepth)
CV_8U	-1/CV_16S/CV_32F/CV_64F
CV_16U/CV_16S	-1/CV_32F/CV_64F
CV_32F	-1/CV_32F/CV_64F
CV_64F	-1/CV_64F

5) 매뉴얼에는 컨볼루션 커널이라 되어 있지만 사실상 코릴레이션이라고 적시되어 있음.

## § BorderTypes

enum cv::BorderTypes

Enumerator	
BORDER_CONSTANT Python: cv.BORDER_CONSTANT	iiiiii abcdefgh iiiiiii with some specified i
BORDER_REPLICATE Python: cv.BORDER_REPLICATE	aaaaaa abcdefgh hhhhhhh
BORDER_REFLECT Python: cv.BORDER_REFLECT	fedcba abcdefgh hgfedcb
BORDER_WRAP Python: cv.BORDER_WRAP	cdefgh abcdefgh abcdefg
BORDER_REFLECT_101 Python: cv.BORDER_REFLECT_101	gfedcb abcdefgh gfedcba
BORDER_TRANSPARENT Python: cv.BORDER_TRANSPARENT	uvwxyz abcdefgh ijklmno
BORDER_REFLECT101 Python: cv.BORDER_REFLECT101	same as BORDER_REFLECT_101
BORDER_DEFAULT Python: cv.BORDER_DEFAULT	same as BORDER_REFLECT_101
BORDER_ISOLATED Python: cv.BORDER_ISOLATED	do not look outside of ROI

### □ BorderType 옵션 바꾸어 보기

소스 영상의 모서리 부분에 대해서는 외곽에는 실제 화소가 없기 때문에 코릴레이션과 같은 필터링 처리가 불가능하다. 그림 5.2.1(a)와 같이 3x3 필터링 처리를 시행할 경우 커널의 중심점(노란색)의 좌측과 상단부의 값이 존재하지 않는다. BorderType 옵션은 이 같은 상황을 방지하기 위해 존재하지 않는 영역의 화소값을 가정하는 방법을 정의한다.

<sup>1</sup> 0	<sup>2</sup> 0	<sup>3</sup> 0	0	0	0	0
<sup>4</sup> 0	<sup>5</sup> 94	<sup>6</sup> 121	138	71	53	65
<sup>7</sup> 0	<sup>8</sup> 74	<sup>9</sup> 26	84	103	126	145
0	191	236	87	189	92	125
0	148	152	172	93	101	251

(a) Array of Pixel Values(zero padding option, default)

<sup>1</sup> 94	<sup>2</sup> 94	<sup>3</sup> 121	138	71	53	65
<sup>4</sup> 94	<sup>5</sup> 94	<sup>6</sup> 121	138	71	53	65
<sup>7</sup> 74	<sup>8</sup> 74	<sup>9</sup> 26	84	103	126	145
191	191	236	87	189	92	125
148	148	152	172	93	101	251

(b) Array of Pixel Values(replicate option)

그림 5.2.1 테두리 부분에서의 코릴레이션 동작

- BorderType=cv.BORDER\_REPLICATE

아래 그림과 같이 영상의 경계 부분에 있는 화소 값이 존재하지 않는 부분에 반복되어 있다고 가정한다. 이 옵션을 선택하는 방법은 다음과 같다.

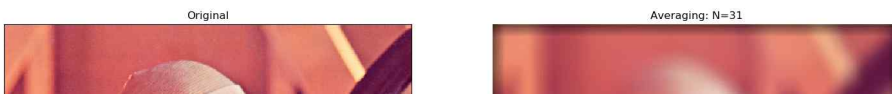
좌측가상화소: 실제영상화소 :우측가상화소  
 aaaaaa|abcdefgh|hhhhhh

`dst = cv.filter2D(img, -1, kernel, borderType = cv.BORDER_REPLICATE )`

- BorderType=cv.BORDER\_ISOLATED

다음과 같이 BORDER\_ISOLATED을 선택하면 경계외부의 값을 0으로 간주하여 처리하기 때문에 테두리 부분이 검게 되는 것을 관찰할 수 있다.

`dst = cv.filter2D(img, -1, kernel, borderType = cv.BORDER_ISOLATED )` # 경계처리 안함.




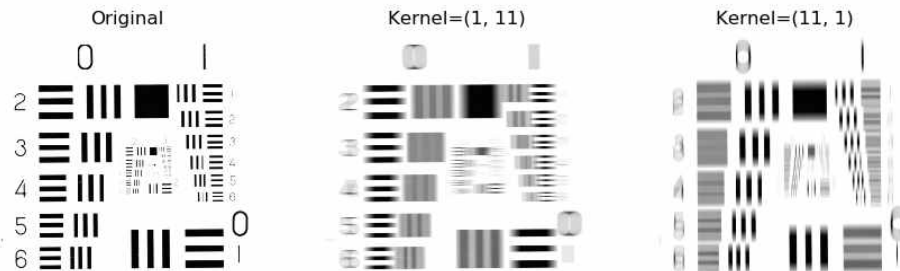
- BorderType=cv.BORDER\_DEFAULT

함수 호출시 지정하지 않을 때 사용되는 default 값이다. 이는 값은 사실상 아래와 같은 BORDER\_REFLECT\_101이다.

좌측가상화소: 실제영상화소 :우측가상화소  
 gfedcb|abcdefgh|gfedcba

BorderDefault는 거울과 같이 대칭된 화소들이 있는 것을 가정하고 있다.

 <b>program</b> sf1_averaging_filter2D.py 실습 3(Matplotlib)	
목표	코릴레이션 수행하기 - <b>filter2D()</b>
실습	실습 3 - 평균화 처리를 처리하는 1x11, 11x1 커널을 만들어 이것으로 입력영상에 대해 코릴레이션 동작을 수행한 결과를 아래와 같이 출력하시오.



## □ 2 종의 커널 만들기

`kernel_list = [ ]` # 다수의 커널로 list 자료로 기록한다.

`kernel = np.ones((1, N), np.float32) / (N)` # 1) 1xN 커널 정의

`kernel_list.append(kernel)`

`kernel = np.ones((N, 1), np.float32)` # 2) Nx1 커널 정의

`kernel /= np.sum(kernel)` # 커널의 합이 1이 되도록 정규화한다.

- 1)과 2)의 커널은 각각 1x11, 11x1 벡터로 만들어진 평균화 필터 커널이다. 1xN 커널은가로 방향의 평균화 작업을 하는 반면, Nx1 커널은 세로방향의 평균화 작업을 시행한다.

## □ pyplot 함수로 1x3창에서 화면에 출력하기

- 1x3의 서브창으로 나눈 후 원본(1번 창)과 수행 결과를 2, 3번 창에 보였다. (1x11) 커널과 (11x1) 커널은 각각 가로 방향 혹은 세로 방향으로 블러링이 발생했음을 알 수 있다.

```
i = 0 # kernel index
for knl in kernel_list:
    print(type(knl), knl.shape)
    dst = cv.filter2D(img, -1, knl, borderType=cv.BORDER_DEFAULT)
    plt.subplot(130 + (i+2)) # 1x3 창의 (i+2)번째 창 선택
    plt.imshow(dst) # filtering 결과 영상
    plt.axis('off')
    plt.title('Kernel=' + str(knl.shape))
    i += 1
```

## 5.3 가우시안 함수와 가우시안 블러링

평균화 커널로 블러링 효과를 얻을 수는 있으나 이는 사람의 시각과는 거리가 있어 실제로는 가우시안 함수를 사용하는 가우시안 블러링을 사용한다.

### ◇ 가우시안(Gaussian) 함수

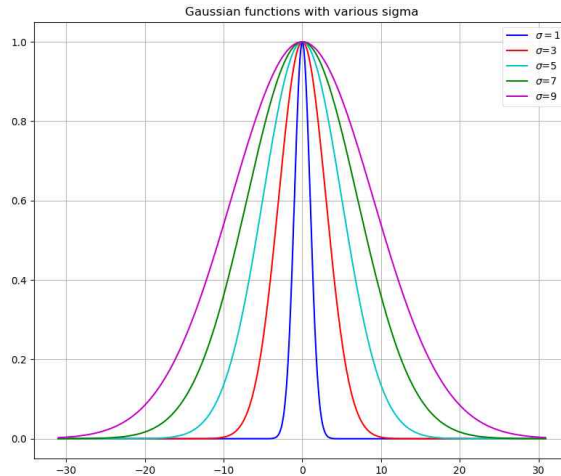


그림 5.3.1 다양한  $\sigma$ 에 따른 가우시안 함수의 곡선

가우시안(Gaussian) 함수( $G(r) = e^{-r^2/2\sigma^2}$ )는 그림 5.3.1처럼 평균값을 중심으로  $\sigma^2$ 의 분산(variance)을 갖고 있는 함수이다.  $\sigma$ (표준편차)의 값이 크면 클수록 중심 값에서 넓은 분포를 갖게 되고  $\sigma$ 가 작으면 그 함수 값에 중심 값의 근처에 집중되어 있는 특성을 가진다.

관측된 데이터, 변량을  $x(i)$ 라 하자. 여기서  $i = 1 \sim N$ .  $\sigma$ 의 정의는 다음 식 5.3.1과 같다.

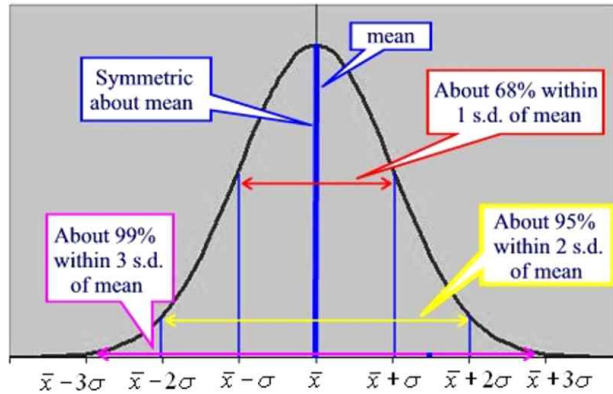
$$\text{평균(Average)} \quad \bar{x} = \frac{1}{N} \sum_{i=1}^N x(i)$$

$$\text{편차(Deviation)} \quad D(i) = x(i) - \bar{x}$$

$$\text{분산(Variance)} \quad \sigma^2 = \frac{1}{N} \sum_{i=1}^N (x(i) - \bar{x})^2$$

$$\text{표준편차(Standard Deviation)} \quad \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x(i) - \bar{x})^2} \quad (\text{식 5.3.1})$$

가우스 함수의 값은  $\pm 1\sigma$  범위에 68%,  $\pm 2\sigma$  범위에 95%,  $\pm 3\sigma$  범위에 99%가 분포한 것으로 알려져  $\sigma$  값이 분포도를 대표하는 파라미터로 활용하기에 유리한 것으로 알려져 있다.



가우시안 함수를 특성을 이해하고 잘 활용하기 위해 직접 이 함수를 그림으로 나타내는 코드를 작성해 보자.

	<b>program</b> sf2_01_gaussian1d_draw_on_2d.py
목표	$\sigma$ 값의 변화에 따른 가우스 함수의 특성을 이해한다.
실습 1	다양한 $\sigma$ 값에 따른 1차원 가우스 함수를 2차원 평면에 그린다.

그림 5.3.1에 보인 바와 같은 시그마의 변화에 따른 가우시안 함수를 도시하는 프로그램 소스는 아래와 같다.

```
def gaussian(r, sigma):
    return np.exp(-r**2/(2*sigma**2))
```

- 가우시안 함수를 정의하였다. x축은 r 변수로 sequence 나열형 변수를 사용한다. 반환 값은 r의 값에 따른 가우시안 함수 y축 값이다.

```
r = np.arange(-31,31,0.1,np.float32)
```

-  $\sigma$  값의 범위를 -31 ~ +31까지로 정한다. 가우시안 함수의 입력 파라미터로 쓰인다.

```
curve_list = [(1, 'b', r'$\sigma=1$'), (3, 'r', r'$\sigma=3$'), (5, 'c', r'$\sigma=5$'),
              (7, 'g', r'$\sigma=7$'), (9, 'm', r'$\sigma=9$')]
```

- curve\_list = [ ( $\sigma$  , 색상\_스트링, 레전드\_스트링), (...), ... ] => 곡선 그림으로 표현될  $\sigma$  값과 곡선의 색상 및 레전드에 사용된 스트링 문자열을 정의한다.



- 레전드(legend) 문자열은 그리스 문자가 사용되기 때문에 `r'$W~$'`로 둘러싸야 한다.


for sigma, color, lbl in curve\_list:

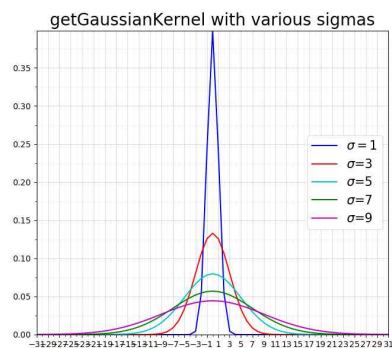
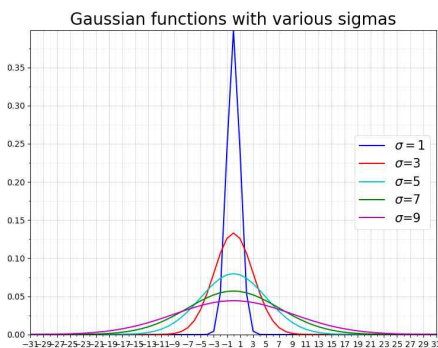
plt.plot(r, gaussian(r, sigma), color, label=lbl)

plt.legend()

- curve\_list에 있는 다양한 파라미터 변화에 따른 가우시안 함수를 그린다. 또한 legend() 함수로 각 선의 색상과 시스마 값을 우측 상단에 보였다.

### 예제: sf2\_gaussian1d\_draw\_on\_2d.py

	<b>program sf2_01_gaussian1d_draw_on_2d.py</b>
<b>목표</b>	가우시안 커널을 1) 직접 만들어 본다. 2) getGaussainKernel() 함수로 받아온다. 이 2개의 결과를 그림으로 그려 비교한다.
<b>실습 2</b>	1) 가우시안 커널을 만들기 위해 정규화된 1차원 가우시안 함수의 데이터를 반환하는 함수를 정의하고, 이를 이용하여 다양한 시그마(표준편차, sigma)에 대한 함수 곡선을 색상을 바꾸어가며 그린다. 2) getGaussainKernel() 함수로 정규화된 커널을 구해서 그림으로 출력하여 비교해 본다.



### ⇒ getGaussianKernel() 함수

getGaussianKernel()은 1차원 가우시안 커널을 반환하는 함수이다.

1차원 가우시안 필터를 반환하는 getGaussianBlur() 함수를 아래에 보였다.

```
retval = cv.getGaussianKernel(ksize, sigma[, ktype])
```

본 함수의 파라미터는 다음과 같다.

<b>ksize</b>	Aperture size. It should be odd ( ksize%2=1 ) and positive.
<b>sigma</b>	Gaussian standard deviation. If it is non-positive, it is computed from ksize as $\sigma = 0.3 * ((ksize - 1) * 0.5 - 1) + 0.8$ .
<b>ktype</b>	Type of filter coefficients. It can be CV_32F or CV_64F .

```
k=11; s=2; t = cv.CV_32F;
```


```
k_1D = cv.getGaussianKernel(ksize=k, sigma=s, ktype=t)
```

```
np.set_printoptions(precision=2) # 소수 이하 2자리까지 출력. 이하 0이면 출력 안한다.
```

```
print('type(k_1D)=', type(k_1D), '| k_1D.shape=', k_1D.shape, 'Wnk_1D.T=Wn', k_1D.T)
```

`filter2D()` 함수는 2차원 커널을 입력으로 받기 때문에 1차원 커널은 2차원으로 변환하여야 한다. 다음 예제는 2차원 가우시안 커널을 3차원으로 표기하는 예제이다. 3차원으로 표시한다는 것은 값이 크면 Z축의 높이가 올라간다는 것이다.

예제: `sf2_02_gaussian2d_draw_on_3d.py`

 <b>program</b> sf2_02_gaussian2d_draw_on_3d.py	
목표	2차원 Gaussian 함수의 3D 표현
실습	2차원 가우시안 함수의 데이터를 반환하는 함수를 정의하고 이를 이용하여 다양한 시그마=2인 2차원 가우시안 함수를 3D 평면에 표시한다. 높이가 그 계수의 크기가 됨.

2차원 가우시안 커널을 수학적식에서 직접 구하는 방법으로 2가지를 보였다.

1) 정규화를 하지 않은 경우

```
def gaussian2D_no(x, y, sigma):
```

```
    return(np.exp(-(x**2 + y**2)/(2*sigma**2)))
```

- 2차원 가우시안 함수를 정의를 보인 것이다. 이는 ( $G(r) = e^{-r^2/2\sigma^2}$ )에서  $r^2 = x^2 + y^2$ 을 대입한 것이다<sup>6)</sup>.

6) 이는 원의 방정식에서 (x, y)의 변화에 따른 원의 반지름 r의 관계식에서 나온 것이다. 중심에서 거리가 얼마나 떨어져 있는가를 (x, y)의 좌표값과 관계지은 것이다.

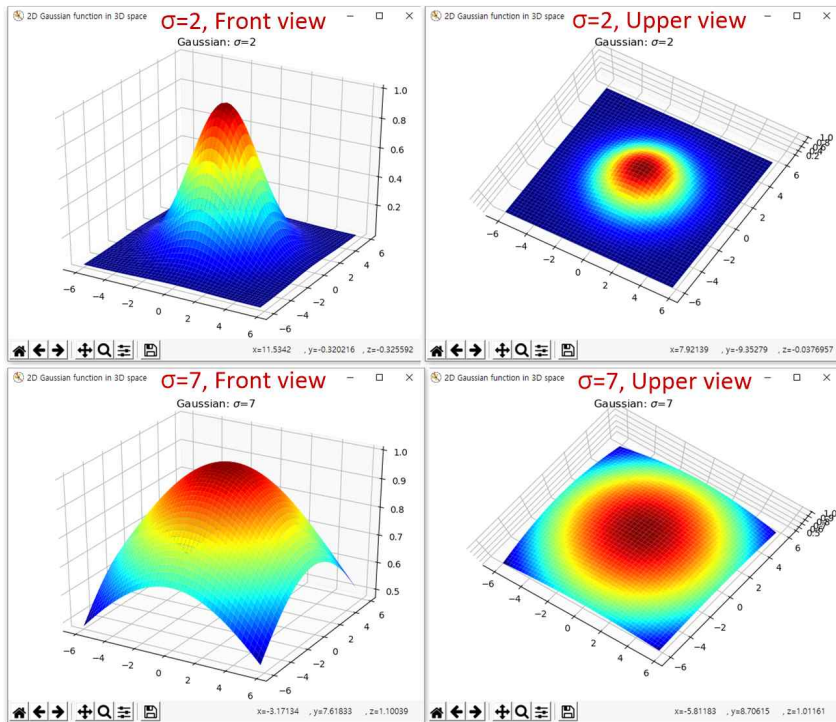


그림 5.3.2 다양한  $\sigma$ 에 따른 2D 가우시안 함수

## 2) 정규화를 한 경우


일단 2차 함수 계수를 구한 다음 모든 계수의 합으로 나누어 합이 1이 되도록 하였다.

```
def gaussian2D_yes(x ,y, sigma):
    tmp = np.exp(-(x**2 + y**2)/(2*sigma**2))
    tmp = tmp/np.sum(tmp)
    return(tmp)
```

filter2D() 함수를 사용한 경우에는 다음과 같이 행렬곱셈으로 2차원 커널을 구할 수 있다.

```
knl = cv.getGaussianKernel()    # 1차원 가우시안 커널
knl2d = knl @ knl.T           # @는 행렬곱 연산을 의미한다. .T는 전치행렬이다.
```

예제: sf3\_getGaussianKernel\_filter2D.py

 <b>program</b> sf3_getGaussianKernel_filter2D.py	
목표	블러링 영상을 코릴리에션 처리로 얻는다.
실습	getGaussainKernel()로 1차원 커널을 만들고, 이를 2차원으로 확장한 후 filter2D() 함수로 필터링하여 Gaussian Blurring을 행한다.

### ◇ 가우시안 블러링

가우시안 함수는 관측된 값(변량,  $r$ )이 평균값에 근처로 분포하는 모습을  $\sigma$ 라는 지표를 이용하여 모델링한다. 가우시안 블러링은 **중심에 가까운 화소는 가중치를 높이고 중심에서 멀리 떨어진 함수의 가중치는 낮추는 방법으로 가중 평균을 행한다.**

가우시안 블러링 처리는 사람이 피사체를 멀리서 바라볼 때 일어나는 현상을 수학적으로 모델링한 것으로 간주할 수 있으며 영상처리에서 중요시되는 알고리즘이다<sup>7)</sup>.

예제: sf3\_getGaussianKernel\_filter2D.py

그림 5.3.3에  $\sigma$ 가 1, 3, 5, 7, 9로 변함에 따라 영상이 블러링되는 모습을 보였다. 가우시안 함수를 사용한 블러링은 대상을 멀리서 바라볼 때 영상이 흐릿해지는 현상과 유사하여 가우시안 함수는 영상을 블러링하는 함수로 많이 활용된다<sup>8)</sup>.

7) 블러링은 영상의 축소 과정 혹은 영상의 잡음 제거에서도 사용되지만 심지어 영상을 선명하게 만들기 위한 방법으로 활용되기도 한다.


8) 블러링은 영상의 축소 과정 혹은 영상의 잡음 제거에서도 사용되지만 디지털 영상처리의 전 분야에 걸쳐 널리 사용되기 때문에 매우 중요한 처리 과정이다. 심지어 영상을 선명하게 만들기 위한 과정으로 블러링이 사용되기도 한다.



그림 5.3.3  $\sigma(1, 3, 5, 7, 9)$ 의 변화에 따른 가우시안 블러링의 사례

가우시안 블러링할 때마다 번거로운 2차원 커널을 연산하는 번거로움을 덜기 위해 openCV에서는 가우시안 블러링 전용 함수를 제공한다.

예제: [sf4\\_GaussianBlur.py](#)

 <b>program</b> sf4_GaussianBlur.py	
목표	GaussianBlur() 함수로 블러링을 행한다. 이 함수의 용법을 익힌다. 이 함수도 양방향, x축 혹은 y축 방향으로 블러링을 행할 수 있다.
실습 1	GaussianBlurring() 함수로 영상에 블러링을 행한다.

### ➤ GaussianBlur() 함수

이 함수는 입력 영상을 주어진 크기의 커널과 x, y 방향의 시그마 값으로 블러링한 결과를 반환하는 함수이다.

```
dst = cv.GaussianBlur( src, ksize, sigmaX[, dst[, sigmaY[, borderType]] ] )
```

본 함수의 파라미터는 다음과 같다.

<b>src</b>	input image: the image can have any number of channels, which are processed independently, but the depth should be CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
<b>dst</b>	output image of the same size and type as src.
<b>ksize</b>	Gaussian kernel size. ksize.width and ksize.height can differ but they both must be positive and odd. Or, they can be zero's and then they are computed from sigma.
<b>sigmaX</b>	Gaussian kernel standard deviation in X direction.
<b>sigmaY</b>	Gaussian kernel standard deviation in Y direction; if sigmaY is zero, it is set to be equal to sigmaX, if both sigmas are zeros, they are computed from ksize.width and ksize.height, respectively (see <a href="#">getGaussianKernel</a> for details); to fully control the result regardless of possible future modifications of all this semantics, it is recommended to specify all of ksize, sigmaX, and sigmaY.
<b>borderType</b>	pixel extrapolation method, see <a href="#">BorderTypes</a>

#### □ 활용 1) 시그마를 제시하고 적절한 커널 크기를 자동 산정. 양방향 블러링. 추천방식

1-1 `blur = cv.GaussianBlur(img, (0, 0), 9)`

- 시그마=9의 값으로 양방향으로 블러링.

2-2 `blur = cv.GaussianBlur(src=img, ksize=(0, 0), sigmaY=sigma, sigmaX=sigma)`

- 키워드를 제시하는 방식. (0, 0)으로 지정하고 두개의 시그마를 모두 지정한다.

3-3 `blur = cv.GaussianBlur(src=img, ksize=(0, 0), sigmaX=9)`

- 키워드를 제시하는 방식. (0, 0)으로 지정하고 sigmaX만 지정하면 양방향으로 블러링.

1-4 `ks = (sigma*6+1, sigma*6+1)`

`blur = cv.GaussianBlur(src=img, ksize=ks, sigmaX=sigma)`

- 적절한 사이즈를 직접계산해서 지정한다.

#### 1-활용 2) 한쪽 방향으로만 블러링하는 사례 한쪽 방향 블러링시 유용.

sigmaX 혹은 sigmaY 중 하나만 지정하고 그에 따라 커널의 크기도 적절히 지정해 준다.

2-1 `N = 6*sigma + 1`

`blur = cv.GaussianBlur(src=img, ksize=(N, 1), sigmaX=sigma)`

- (N, 1) =(가로, 세로)를 지정한다. 가로 방향의 블러링.

2-2 `blur = cv.GaussianBlur(src=img, ksize=(1, N), sigmaX=0, sigmaY=sigma)`

- (1, N) =(가로, 세로)를 지정한다. 세로 방향의 블러링.

□ 활용 3) 커널 사이즈를 제시하고 그에 따라 자체적으로 적당한 시그마를 계산.

3-1 `blur = cv.GaussianBlur(img, (41, 41), 0)`

- 시그마를 0으로 정하면 ksize로 부터 시그마를 계산.

3-2 `blur = cv.GaussianBlur(img, (41, 1), 0)`

한쪽 방향 블러링.

- ksize로 시그마를 계산. 가로 방향의 블러링

3-3 `blur = cv.GaussianBlur(img, (1, 41), 0)`

한쪽 방향 블러링.

- ksize로 시그마를 계산. 세로 방향의 블러링

□ 활용 4) 주의 사례: 오류는 아니지만 바람직하지 않은 파라미터

아래 2개의 사례는 블러링 호출한 결과가 원본과 다를 바 없다.

4-1 `blur = cv.GaussianBlur(src=img, ksize=(51, 51), sigmaX=1)`

- 시그마에 비해 커널의 크기가 크다. => 연산낭비. 커널의 외곽부분은 0으로 둘러 쌓여있다. 이 값들은 커널 계수와 화소 값의 곱셈 연산에서 0의 값을 만들어내므로 쓸모없는 연산시간 낭비를 초래할 수 있다<sup>9)</sup>.

4-2 `blur = cv.GaussianBlur(img, (3, 3), 15)`


- 시그마 값에 비해 지나치게 작은 크기의 커널을 사용한다. 이는 연산 결과가 부정확할 수 있는 여지가 있다. 커널이 작으면 가우시안의 종(bell) 모양이 제대로 반영되지 않는다. 지나치게 큰 시그마에 작은 커널은 사실과 평균 필터와 유사한 결과를 만들어 낼 수 있고 심하여 계수의 합이 1보다 작아서 어두운 영상을 만들어 낼 수도 있다.

---

9) 그러나 보다 엄밀히 말하면 커널이 크면 주파수 공간에서 컨벌루션이 곱셈으로 수행되므로 이를 이용하는 함수라면 연산 소모가 거의 발생하지는 않는다.

2차원 필터를 사용하지 않고 x방향 1차원 필터로 가로 방향으로 처리한 결과에 같은 필터를 전치하여 세로 방향으로 만든 필터를 사용하여 다시 필터링하여 결과가 2차원 필터를 쓴 것과 같으면 그런 2차원 필터를 분리 가능한 필터이라고 한다.

예제: `sf5_filter2D_sepFilter2D.py`

 <b>program</b> <code>sf5_filter2D_sepFilter2D.py</code>	
목표	<code>filter2D()</code> 함수와 <code>sepFilter2D()</code> 의 연산 내부 동작 원리를 파악한다.
실습	<code>filter2D()</code> 함수와 <code>sepFilter2D()</code> 의 연산 시간을 비교한다.

### ⇒ `sepFilter2D` 함수

2차원 필터링한 결과가 가로 방향으로 1차원 필터링한 결과 영상을 다시 세로 방향으로 1차원 필터링한 결과와 같으면 이 2차원 필터링 연산을 **선형분리가능(linearly separable)**하다고 한다. **결론적으로 평균 필터와 가우시안 필터는 선형분리가능한 필터이다<sup>10)</sup>**.

#### □ `sepFilter2D()` 함수를 사용한 필터링 처리

1차원 필터링을 1회 혹은 1회 시행하는 함수를 아래에 보였다.

```
dst = cv.sepFilter2D(src, ddepth, kernelX, kernelY[, dst[, anchor[, delta[, borderType]]]])
```

본 함수의 파라미터는 다음과 같다.

<b>src</b>	Source image.
<b>dst</b>	Destination image of the same size and the same number of channels as src .
<b>ddepth</b>	Destination image depth, see <a href="#">combinations</a>
<b>kernelX</b>	Coefficients for filtering each row.
<b>kernelY</b>	Coefficients for filtering each column.
<b>anchor</b>	Anchor position within the kernel. The default value $(-1, -1)$ means that the

10) 선형 분리 필터의 개념은 연산량 절감을 위해 제시되었다. 그러나 최근의 필터링 연산은 11x11 이상의 커널을 사용하면 푸리에 변환을 향려 연산을 고속화한다. 이 때문에 선형분리 필터링을 행하는 `sepFilter2D()` 함수가 오히려 더 연산이 소요되는 것을 발견하였다. 예제: `sf5_filter2d_sepFilter2D.py` 실험 결과를 참조바랍니다.



	anchor is at the kernel center.
<b>delta</b>	Value added to the filtered results before storing them.
<b>borderType</b>	Pixel extrapolation method, see <a href="#">BorderTypes</a>

`dst = cv.sepFilter2D(src=img, ddepth=-1, kernelX=k_1D, kernelY=k_1D.T)`

- `getGaussianKernel()` 함수로 생성한 1차원 커널, `k_1D`를 x 방향과 y방향에 적용하여 블러링된 영상을 취득한다.

각 함수의 연산 회수는 `sepFilter2D()`가 2\*K번이라면 `filter2D()`는 KxK번의 연산이 필요하다. 연산회수 관점에서는 `sepFilter2D`가 매우 유리할 것으로 보인다. 그러나, 실제로 실험을 해보면 예상외로 `filter2D()` 함수가 `sepFilter2D()` 보다 수행 속도가 오히려 더 빠르다.

## 이유

OPenCV 매뉴얼에 의하면 `filter2D()` 함수는 11x11 이상이면 DFT 변환에 의해 연산을 수행한다고 기술되어 있다. DFT 변환과 역변환에 시간이 소요되지만 일단 변환이 되면 코릴레이션 연산은 두 어레이의 요소간 곱셈으로 단순화되기 때문에 일정 수준이상으로는 커널이 커져도 시간이 별로 증가하지 않는 것이다.

큰 필터를 사용할수록 DFT 변환을 사용하는 `filter2D()` 함수가 유리해질 것으로 예상된다. `sepFilter2D()`가 유리하다는 발상은 과거 SIMD 구조, GPU 등의 가속기가 지원되지 않는 환경에서 CPU의 연산력에 의존하는 4중 loop 방식의 한계를 극복하기 위해 창안된 기법이라고 추정된다. 아래 실제 측정사례를 보았다.

커널의 크기	<code>filter2D()</code> 경과시간(초)	<code>sepFilter2D()</code> 경과시간(초)
(21, 21)	0.08178114891052246	0.0718073844909668
(51, 51)	0.09973454475402832	0.17752528190612793
(91, 91)	0.11568975448608398	0.3231358528137207

## 5.4 검토사항

### ◇ 참고: 2차원 가우시안 필터 설계

2차원 필터를 만들기 위해서는 1차원 가우시안 함수를  $r^2 = x^2 + y^2$  의 관계를 이용하면 식 5.3.2와 같이 2차원화할 수 있다.

$$G(x, y) = e^{-(x^2 + y^2)/2\sigma^2} \quad (\text{식 5.3.2})$$

여기서 (x,y)는 중점을 (0,0)으로 한다. 커널의 크기가 (2a+1) x (2a+1)이라면 x와 y는 각각 -a ~ +a의 값을 가진다. 즉, 3x3 커널이라면 x와 y에 각각 [-1 0 +1]의 값을 대입하고, 5x5 크기의 커널이라면 [-2 -1 0 +1 +2]의 값을 대입한다. 그림 5.3.3에 5x5 커널의 경우 2차원 가우시안 함수(식 5.3.2)에 대입해야 할 값을 (x,y) 좌표별로 보였다.

$g(-2, -2)$	$g(-1, -2)$	$g(0, -2)$	$g(1, -2)$	$g(2, -2)$
$g(-2, -1)$	$g(-1, -1)$	$g(0, -1)$	$g(1, -1)$	$g(2, -1)$
$g(-2, 0)$	$g(-1, 0)$	$g(0, 0)$	$g(1, 0)$	$g(2, 0)$
$g(-2, 1)$	$g(-1, 1)$	$g(0, 1)$	$g(1, 1)$	$g(2, 1)$
$g(-2, 2)$	$g(-1, 2)$	$g(0, 2)$	$g(1, 2)$	$g(2, 2)$

그림 5.3.3 (x,y) 좌표에 따른  
가우시안 함수의 값

**미션:** 파이썬으로 주어진 시그마(sigma)와 커널의 크기(kize)에 따라 해당 가우시안 커널을 반환하는 함수를 설계해 보자.

```
반환하는 커널의 타입은 ndarray, 데이터 형은 ktype으로 지정된다.  
kernel_1D = getMyGaussiankernel(sigma, ksize, ktype)  
# ksize가 정수일 때: 반환받는 커널의 크기는 1xN. N=3, 5, 7, 홀수...  
# ksize = (가로, 세로)일 때: 반환받는 커널의 크기=(N, M). N, M 모두 홀수.
```

## ◇ 참고: 시그마와 필터 크기의 관계

필터 커널이 커지면 연산량이 많아진다. 시그마가 적을 경우에는 중심 외곽의 계수의 값은 0에 가까운 값이 되기 때문에 이 경우에는 필터의 크기를 줄이는 것이 연산량 절감에 도움이 된다.

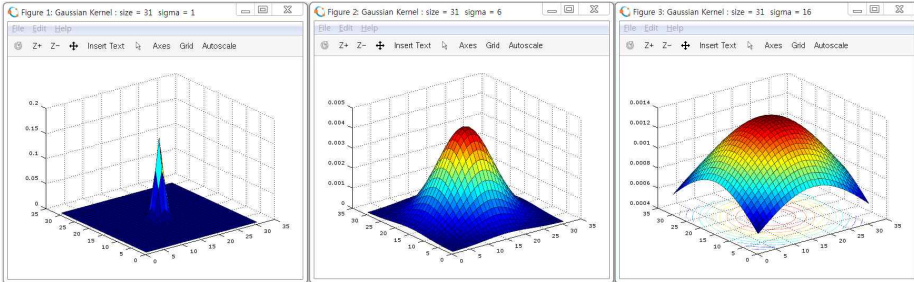


그림 5.4.1 31x31 커널의 모양(각각  $\sigma=1, 6, 16$ )

그림 5.4.1은 시그마가 1, 6, 16으로 변할 때 31x31 가우시안 커널의 모양을 보인 것이다.

좌측의 그림( $\sigma=1$ )을 보면 가우시안 커널의 계수 값은 중앙부에만 일부 0이 아닌 값이 존재하고 나머지는 모두 0으로 채워져 있음을 알 수 있다. 이 경우 31x31 크기의 커널을 사용하면 계수가 0인 부분에 대해 불필요한 연산을 유발하기 때문에 커널 크기가 적당하다고 할 수 없다<sup>11)</sup>.

반면 우측의 시그마가 16일 때는 커널의 테두리에도 0이 아닌 값이 적지 않게 분포함으로 이 경우에는 올바른 연산이 이루어지지 않기 때문에 필터의 크기를 좀 더 확장할 필요가 있다.

결론적으로 시그마 값이 비해 지나치게 적은 커널을 사용하면 정확한 블러링을 기대할 수 없고, 필요 이상의 큰 커널은 연산 낭비를 초래한다. 일반적으로 활용되는 지침으로 말하면 다음과 같이 시그마의 6배+1 정도를 사용한다.

$$\text{커널 사이즈} = 6 \cdot \sigma + 1$$

11) 코릴레이션 연산이 커널의 계수와 해당 위치의 화소 값과의 곱셈한 결과를 모두 합산하는 연산으로 이루어져 있음을 기억하자. 계수가 0이면 해당 위치와의 곱셈 연산은 연산 노력만 투여될 뿐 의미가 없다.