9. hét / Epilógus

A mai órán a következő témaköröket fogjuk áttekinteni:

- A 2D Konvolúció elméleti bevezetése
- Élkeresés konvolúció segítségével (*Edge detection*)

Választható témák:

- Képszegmentáció K-Means Clustering algoritmus segítségével
- Mozgó objektum detektálása (*Motion detection*)
- Körök és tetszőleges geometriák detektálása (*Ridge detection - Hough transform*)

Az, hogy az órán melyik feladatokat fogjuk áttekinteni azt a hallgatók döntésére bízom. A fenti témákból szabadon felállíthatnak a hallgatók a jelenléti órán egy sorrendet, amely alapján megoldjuk a feladatokat.

9. hét / I. 2D Konvolúció

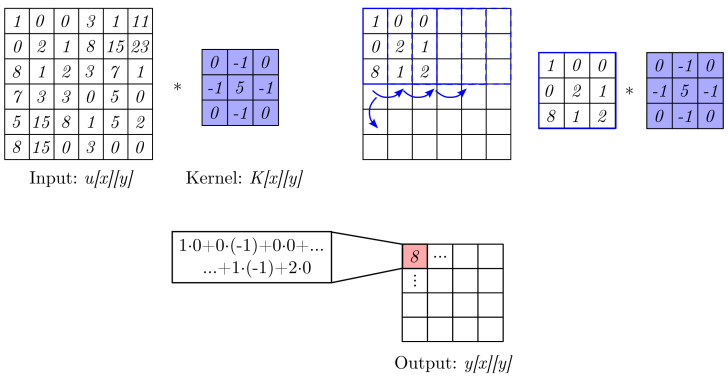
A **2D** (diszkrét) **konvolúció** egyike a legfontosabb műveleteknek, amellyel képfeldolgozás során találkozhatunk. Ezen a ponton már legtöbben találkozhattunk ennek *egydimenziós* változatával, amelyet a következő alakban szokás felírni:

$$y[k] = (u \star w)[k] := \sum_n u[k]w[n-k]$$

Röviden fejtsük ki, hogy valójában milyen lényegi műveletet végzünk itt el! Adott nekünk egy $u[k]$ tetszőleges *bemeneti függvény*ünk és egy $w[k]$ *súlyfüggvény*ünk, amelyek értékeit egy megadott sorrendben összeszorozzuk. (Ne ijesszen meg bennünket az $[n-k]$ jelölés, ez tulajdonképpen azt jelenti, hogy amíg az $u[k]$ függvénnyel az időben előre felé haladunk, addig a $w[n-k]$ esetén annak n. elemétől visszafelé jövünk.) Tehát összefoglalva a *konvolúció* segítségével az $y[k]$ függvényt előállítom a **bemenet** gondosan megválasztott **súlyozott átlag**aként ($||w||_1 = 1$), vagy **súlyozott összeg**eként ($||w||_1 \neq 1$).

Kétdimenziós esetben is ugyanezzel a szemlélettel kell élnünk, csak a **bemenetünk** egy egyváltozós helyett *kétváltozós függvény* lesz, ez diszkrét esetben egy **mátrix** (programozás során egy kép), ugyanígy a **súlyfüggvény**, melyet itt **kernel**nek nevezünk is egy (az esetek túlnyomó többségében páratlan) rangú **négyzetes mátrix**, (leggyakrabban 3x3 és 5x5). Ekkor a konvolúció felírható, mint

$$y[x][y] = (K \star u)[x][y] = \sum_n \sum_m K[n][m] \cdot u[x-n][y-m]$$



A *kernel*től függően *különböző hatásokat érhetünk el*. Alábbiakban rendre az *elhomályosítás*, *élesítés* és *éldetektálás* kerneleit láthatjuk:

$$\mathbf{K}_1 = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad \mathbf{K}_2 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \qquad \mathbf{K}_3 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

A **konvolúció**(s neurális háló) úgynevezett (lokális) **feature extractor**, azaz a *megfelelő súlyok megválasztásával* lehetséges olyan programot írni, amely képes megtanulni és felismerni az emberi arc jellemzőit, amely felett könnyen implementálhatunk egy *arcfelismerő*t. Ugyanígy a konvolúció alapja az *image classification*nek, amely segítségével eldönthetjük, hogy egy képen milyen számjegyek vannak, milyen ruhadarabok, vagy ételek.

1. példa - Élkeresés konvolúció segítségével

Keressük meg az éleket a `bme.jpg` képen konvolúciós eljárás segítségével! A konvolúciós kernelek legyenek az alábbiak:

$$\mathbf{K}_1 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad \mathbf{K}_2 = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad \mathbf{K}_3 = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

A $\mathbf{K}_{2,3}$ kernelek az úgynevezett *Prewitt operátor*ok.

In [46]:
```python
# Szükséges importok
import cv2              # Képfeldolgozáshoz
import numpy as np      # Kernel definiáláshoz

# Beolvassuk a képet
image = cv2.imread('bme.jpg')

# Ha a képet nem találjuk, akkor hibát jelzünk
if image is None:
    print('Could not read image')

# Definiáljunk a kerneleket
kernel1 = np.array([[-1, -1, -1],
                    [-1, 8, -1],
                    [-1, -1, -1]])
kernel2 = np.array([[-1, 0, 1],
                    [-1, 0, 1],
                    [-1, 0, 1]])
kernel3 = kernel2.transpose()

# Elkészítjük a maszkolt képet
masked = cv2.filter2D(image, ddepth=-1, kernel=kernel1)
masked2 = cv2.filter2D(image, ddepth=-1, kernel=kernel2)
masked3 = cv2.filter2D(image, ddepth=-1, kernel=kernel3)

# Megjelenítjük a képeket
cv2.imshow('BME', image)
cv2.imshow('K1', masked)
cv2.imshow('K2', masked2)
cv2.imshow('K3', masked3)

# Tetszőleges kép lenyomásakor bezárjuk az ablakokat
cv2.waitKey()
cv2.destroyAllWindows()
```

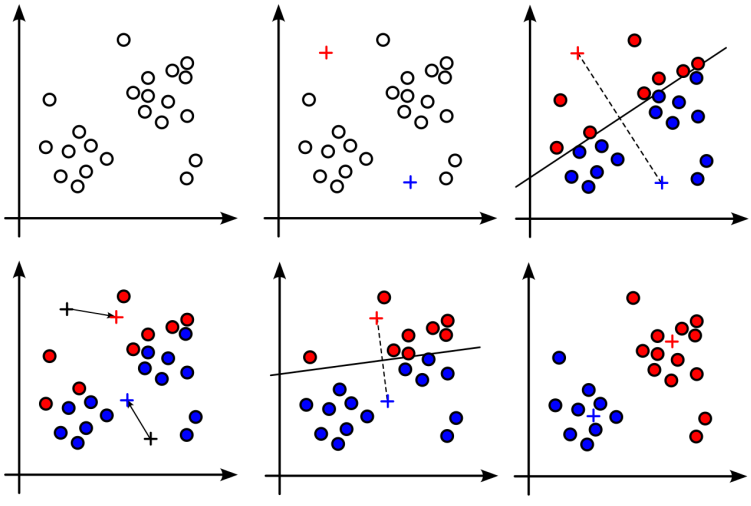
9. hét / II. Képszegmentáció K-means Clustering segítségével

A **képszegmentáció** lényege, hogy egy kép pixeleit úgynevezett **szegmensekbe csoportosítjuk** egy megadott szempont alapján. Például ha van egy utcaképünk, akkor a képen szeretnénk különválasztani a *gyalogos*okat az *úttest*től és az *épület*ektől. Ekkor a kép pixeleit három különböző (gyalogos, úttest, épület) szegmensbe soroljuk. Ezt jellemzően egy **komoly kihívást jelent**ő feladat, ugyanis gyakran a szegmenshatárok összefolynak és nehezen megkülönböztethetőek (Magdi nénin pont olyan mintájú nadrág van, mint a fali tapéta...), a képek gyakran zajjal terheltek, emellett az algoritmusok jelentős része összetett matematikai műveletekkel dolgozik, ezért nagyon számításigényes is a képszegmentáció. Leggyakrabban a következő eljárásokkal dolgozunk:

- **Thresholding** alapú szegmentáció: a pixelek *intenzitása* alapján létrehozzuk a szegmenseket, ahova besoroljuk az egyes pixeleket. GeeksforGeeks: Thresholding Segmentation
- **Edge-based** szegmentáció: megkeressük egy képen az éleket, és az általuk meghatározott zárt geometriák alkotják a szegmenseket. Elterjedt a *Canny edge detection*, *Sobel operator* és a *Roberts operator* alkalmazása ezen a területen. Wiki: Canny edge detector | OpenCV: Canny Edge Detection
- **Clustering-based** szegmentáció: klaszterekbe osztjuk a hasonló intenzitású/színű/textúrájú, vagy akármilyen általunk megválasztott tulajdonságú pixeleket. Elterjedt a *K-means clustering*, *fuzzy C-means clustering* és a *hierarchical clustering*. Wiki: k-Means clustering | OpenCV: cv2.kmeans() | OpenCV: Color Quantization
- Machine learning alapú algoritmusok.

A *k-Means clustering* algoritmus működési elve:

1. **Feladat**: Adott adathalmazt szeretnénk felosztani *k* darab szegmensre.
2. **Inicializálás**: Az adathalmazban felveszük *k* darab **center**t *véletlenszerűen*.
3. **Felosztás**: Ezt követően az adathalmaz összes elemét hozzárendeljünk a hozzá legközelebb eső *center*hez. Ezáltal kialakul *k* darab *cluster*.
4. **Centerek újraszámítása**: Az előző lépésben kialakult clusterekben **átlagoljuk az elemeket**, így megkapjuk az **új center**eket.
5. **Iteráció**: *Ismételjük a 3-4. lépéseket*. Az újonnan megkapott centerek segítségével *újra felosztjuk* az adathalmaz elemeit. A kialakult clustereket átlagoljuk és megkapjuk az újabb centereket.
6. **Kilépési feltétel**: Addig iterálunk, míg a klaszter határok stabilak maradnak (azaz ε hibán belül mozognak az iterálás során, vagy egy meghatározott `MAX_ITER` határt el nem érünk). Ekkor megkapjuk a klasztereket, amik szegmensekre osztják a képet.



2. példa / Képszegmentáció k-Means algoritmus segítségével

Olvassunk be egy képet, és szegmentáljuk *szín* szerint $K = 3, 4, 5$ részre!

In [43]:
```
# Szükséges importok
import cv2              # Képfeldolgozáshoz
import numpy as np      # Típuskonverzióhoz

# Beolvassuk a képet
image = cv2.imread('sunflower.jpg')

# Átrendezzük 3 oszlopba a kép tartalmát
Z = image.reshape((-1,3))
# Végzünk egy típuskonverziót, hogy alkalmazható legyen a k-Means
Z = np.float32(Z)

# Definiáljuk a clusterek számát
K = 3

# Definiáljuk a k-Means kritériumait: meddig fusson, hogyan induljon...
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret,label,center=cv2.kmeans(Z,K,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)

# Visszaalakítjuk a képet, az eredeti dimenziójába
center = np.uint8(center)
res = center[label.flatten()]
image_kmeans = res.reshape((image.shape))

# Megjelenítjük a képet
cv2.imshow('k-Means segmentation',image_kmeans)

# Billentyű lenyomása után bezárjuk az ablakokat
cv2.waitKey(0)
cv2.destroyAllWindows()
```

9. hét / III. Mozgó objektumok detektálása

A **motion detection** egyike a képfeldolgozás legfontosabb irányainak, a jelen és jövő technológiája gyakorlatilag elképzelhetetlenné válik enélkül. Gondoljunk csak a legegyszerűbb riasztórendszerre, amely csak akkor és azokon a helyeken rögzíti az utcaképet, ha azon lényeges mozgás megfigyelhető. Ugyanígy eszünkbe juthatnak a különböző közlekedési eszközök, amelyek fel vannak szerelve védelmi rendszerekkel, így képesek detektálni a gyalogosokat, akadályokat. Orvostechnikai alkalmazásokat is szép számmal említhetünk, különösképp a járásvizsgálatot, de akár ha az ipari automatizálásra gondolunk, akkor is kiderül, hogy mennyire fontos dologról van szó.

Alkalmazástól függően az alábbi eljárásokat használhatjuk mozgásdetektálásra:

- **Frame differencing**: videórögzítés során képezzük az aktuális és a közvetlen előző képkockák *különbség*ét, amelyből kirajzolódik a mozgás.
- **Optical Flow**: a videón található objektumok mozgását próbáljuk becsülni, közelíteni az *elmozdulásvektor*ok és/vagy *sebességmező* segítségével. Wiki: Optical Flow | OpenCV: Optical Flow
- Machine Learning alapú megoldások.

Mi alapvetően a *Frame differencing* eljárással fogunk megismerkedni, révén ez egy nagyon egyszerű és könnyen implementálható algoritmus. Érdemes megjegyezni, hogy a fentebb említett eljárások mind széles körben alkalmazottak, mindig a feladattól függ, hogy mit érdemes használni!

3. példa - Mozgásdetektálás

Implementáljuk a *frame differencing* algoritmust a videókameránk képére!

In [1]:
```
# Szükséges importok
import cv2          # Az alapvető képfeldolgozáshoz
import numpy as np  # A matematikai műveletekhez

# Elindítjuk a videókamerát
vid = cv2.VideoCapture(0)

# Segédváltozó
previous_frame = None

# Amíg be nem zárjuk az alkalmazásablakot...
while True:
  # Beolvassuk a videókamerából a képet:
  ret, frame = vid.read()

  # Áttérünk szürkeárnyalatos képre és alkalmazunk egy blurt
  prepared_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
  prepared_frame = cv2.GaussianBlur(prepared_frame, (5,5), 0)

  # Kiszámítjuk az előző képhez képesti különbségeket:
  if (previous_frame is None):
    # Az első képkocka előtt még nincs korábbi
    previous_frame = prepared_frame
    continue

  # Megnézzük (abszolút értékben) a két kép közötti különbséget
  diff_frame = cv2.absdiff(previous_frame,prepared_frame)

  # A mostani képkocka a következő ciklusban az előző képkocka lesz
  previous_frame = prepared_frame

  # Dilatáció segítségével kiemeljük a változást
  diff_frame = cv2.dilate(diff_frame, np.ones((5,5)), 1)

  # Csak a kellően nagy eltéréseket szeretnénk megjeleníteni, ezért thresholdolunk
  thresh_frame = cv2.threshold(diff_frame, thresh=20, maxval=255,
                               type=cv2.THRESH_BINARY)[1]

  # A különbség kontúrvonalai megmutatják a különbségeket
  contours, _ = cv2.findContours(thresh_frame, mode=cv2.RETR_EXTERNAL,
                                 method=cv2.CHAIN_APPROX_SIMPLE)
  cv2.drawContours(frame, contours=contours, contourIdx=-1,
                   color=(255, 255, 0), thickness=2, lineType=cv2.LINE_AA)

  # Megjelenítjük a kontúrvonalakkal ellátott képet
  cv2.imshow('frame', frame)
  cv2.imshow('diff_frame', diff_frame)
  cv2.imshow('prepared_frame', prepared_frame)

  # ESC gomb lenyomására kilépünk
  if (cv2.waitKey(30) == 27):
      break

# Bezárjuk az alkalmazásablakot
cv2.destroyAllWindows()

# Megszíktjuk a videókamera használatát
vid.release()
```

9. hét / IV. Körök és tetszőleges geometriák detektálása

A következőkben azt fogjuk áttekinteni, hogyan tudunk köröket és egyéb geometriákat detektálni a képen. Legyen szó arról, hogy az önvezető autónak kell egy sávtartó algoritmushoz az útburkolati jeleket (például a felező- és záróvonalat, de akár a zebrát vagy közlekedési táblákat) felismerni, vagy éppen egy röntgenfelvételen szeretnénk a megfelelő csontokat felismerni, az úgynevezett *Ridge detecion* kiemelt szerepet kap a képfeldolgozásban.

Ezen a területen alapvetően az alábbi két eljárást alkalmazzák:

- Az (általánosított) **Hough transzformáció**t (GHT), amelynek mi egy speciális esetével, a *Circle Hough Transform*mal fogunk megismerkedni Wiki: Circle Hough Transform,
- Illetve az (általánosított) **Struktúra tenzor**t. Wiki: Structure Tensor

Mivel a *struktúra tenzor*ok megértéséhez komolyabb matematikai apparátusra lenne szükség, ezért a kurzus keretein belül nem fogjuk tárgyalni. Amit érdemes róla megjegyezni, hogy az úgynevezett *gradiens*ek segítségével dolgozik, ergó többváltozós függvények heves deriválásából és integrálásából áll. Viszont mindkét eljárás nagyon elterjedt, ezért alább összefoglalom az egyes algoritmusok előnyeit és hátrányait.

A *Hough-transzformáció* **előnyei**:

- Sokkal **robosztus**abb, és a *struktúra tenzor*ral szemben **nem érzékeny** a **zajra** (mely zaj a deriválásnál jelent problémát);
- **Globálisan** képes megtalálni az optimális megoldásokat, azaz mindig figyelembe veszi a kép teljes egészét;
- Gyakorlatilag tetszőlegesen bonyolult görbéket képes megtalálni.

A *Struktúra tenzor* **előnyei**:

- Lényegesen **kisebb** a **számítási igénye** a *Hough-transzformáció*hoz képest. Sokkal gyorsabban és kisebb memórián is futtatható algoritmus;
- **Lokálisan** képes megtalálni a megoldásokat, azaz **pontosabb** és **részletesebb** információ nyerhető ki belőle;
- Mivel lokálisan akar jellemzőket detektálni, ezért **flexibilis**, tetszőleges méretű és alakú képre alkalmazható.

Így ha adott a kérdés, hogy mikor melyiket használjuk, a válasz: **attól függ!**

4. példa - Körök detektálása

Olvassuk be a videókamera képét és detektáljunk rajta köröket!

In [45]:
```
# Szükséges importok
import cv2                              # Képfeldolgozáshoz
import numpy as np                  # Alapvető numerikus műveletkhez

# Kérjük be a videókamera képét
vid = cv2.VideoCapture(0)

# Amíg be nem zárjuk az alkalmazást...
while(True):
```

```
        # Beolvassuk a kamerából a képkockát
        ret, frame = vid.read()

        # A képfeldolgozást megkönnyítendő, szürkeárnyalatos képet készítünk
        gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

        # A zajt csökkentendő elmosást alkalmazunk
        gray_b = cv2.blur(gray,(11,11))

        # Hough Transzformáció; visszatérési értékei: a kör középpontja (a,b) és sugara (r)
        detect_circles = cv2.HoughCircles(gray_b,cv2.HOUGH_GRADIENT,1,20,param1=50
                                  ,param2=30,minRadius=1,maxRadius=40)

        # Ha találtunk köröket...
        if detect_circles is not None:
                # Beolvassuk egyesével a köröket
                for pt in detect_circles[0,:]:
                        # Típuskonverziók
                        a,b,r = np.uint16(pt[0]),np.uint16(pt[1]),np.uint16(pt[2])

                        # A detektált kört pirossal jelöljük
                        cv2.circle(frame,(a,b),r,(0,0,255),2)

                        # A kör középpontját zölddel
                        cv2.circle(frame,(a,b),1,(0,255,0),3)

        # Megjelenítjük a körökkel ellátott képet
        cv2.imshow("frame",frame)

        # A 'q' gomb megnyomására
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

# Megszüntetjük a kamerakép használatát
vid.release()

# Bezárjuk az alkalmazásablakot
cv2.destroyAllWindows()
```

9. hét / Prológus

Kiegészítés az órai anyaghoz:

- OpenCV dokumentációja: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

Bármilyen kérdés, kérés vagy probléma esetén keressetek minket az alábbi elérhetőségeken:

- Monori Bence - m.bence02@outlook.hu
- Wenesz Dominik - weneszdominik@gmail.com

Illetve anonim üzenetküldésre is lehetőséget biztosítunk, ezt az alábbi linken tudjátok elérni: https://forms.gle/Qvj7okQqCMRc4cBu7