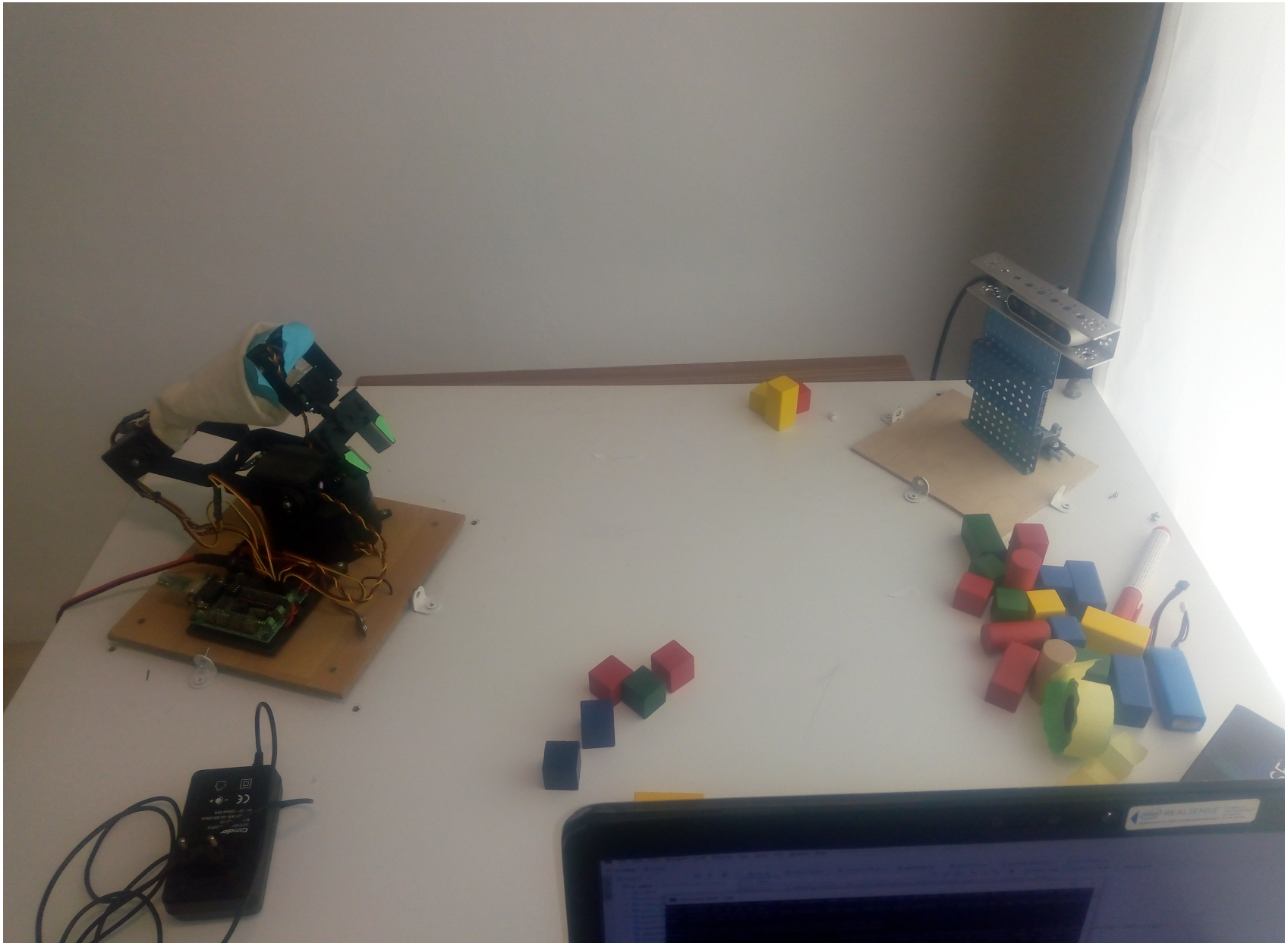
Detekcia objektov pomocou algoritmu RANSAC

Viktor Kocur
viktor.kocur@fmph.uniba.sk

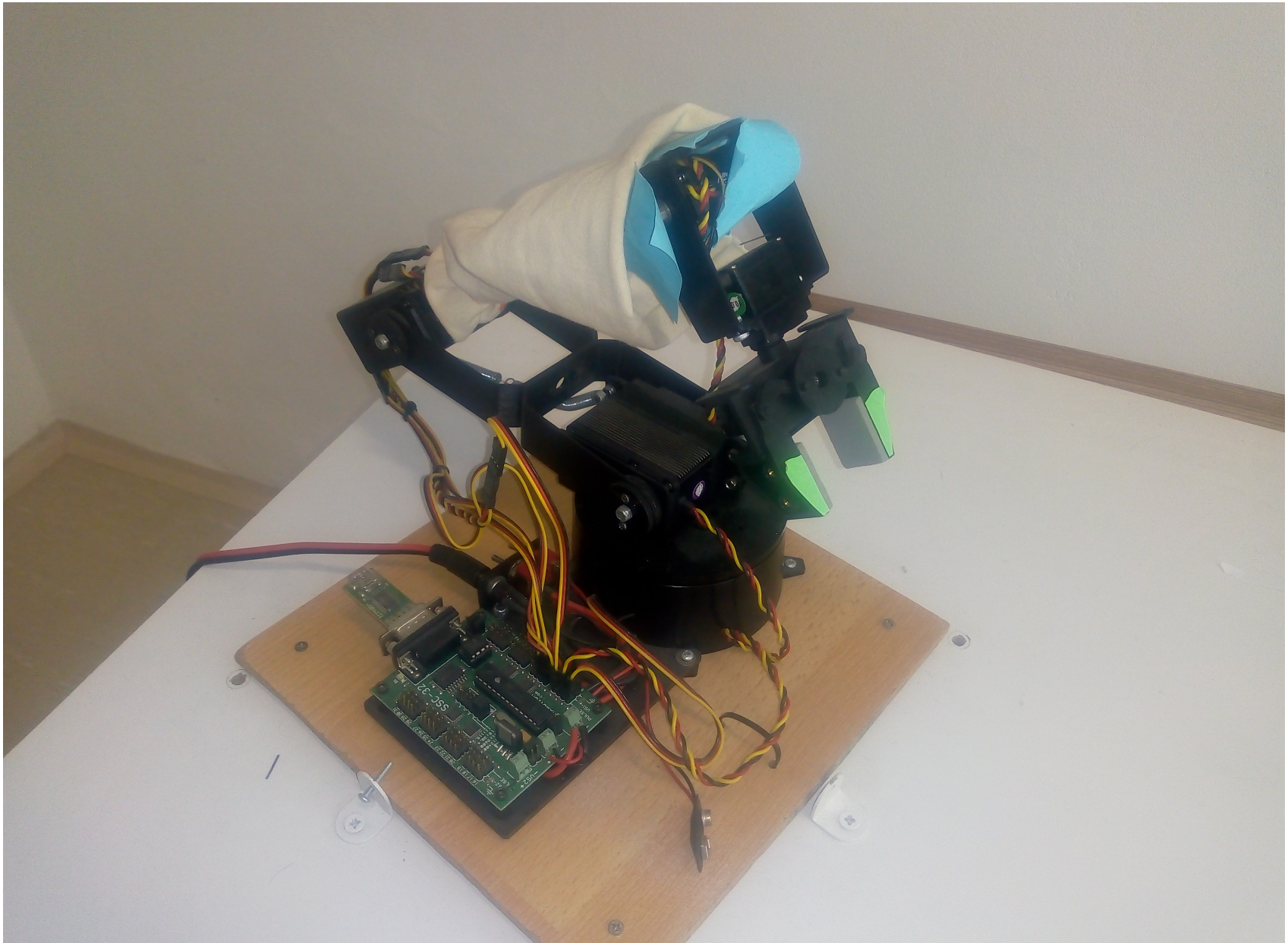
DAI FMFI UK

4.3.2020

Setup

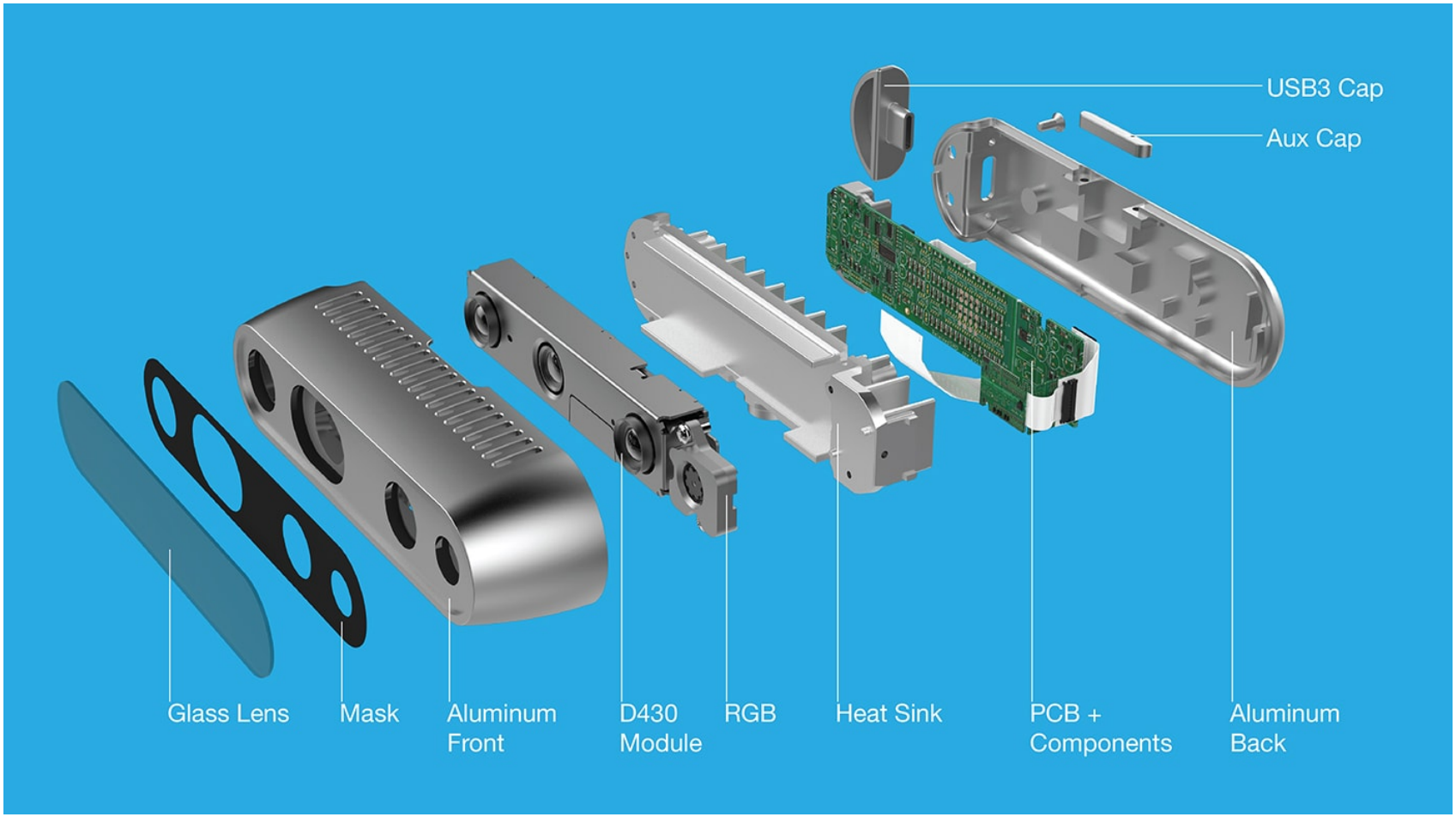


Robotické rameno

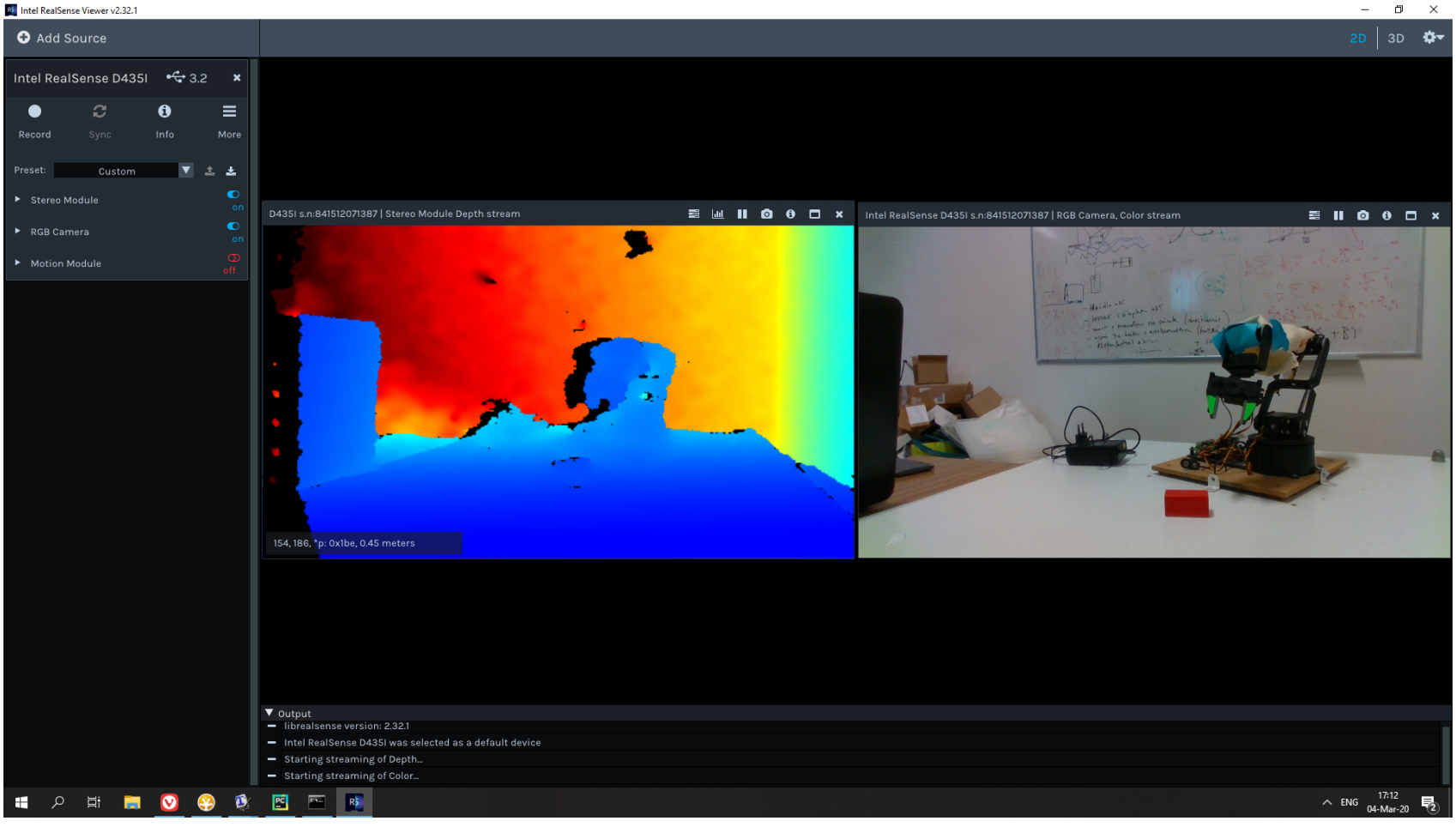


Detegované objekty

Intel RealSense D435i



Intel RealSense Viewer



Využité knižnice

Kód je implementovaný v Pythone 3.6 s využitím knižníc:

- pyrealsense
- NumPy
- OpenCV
- PPTK viewer
- scikit-learn

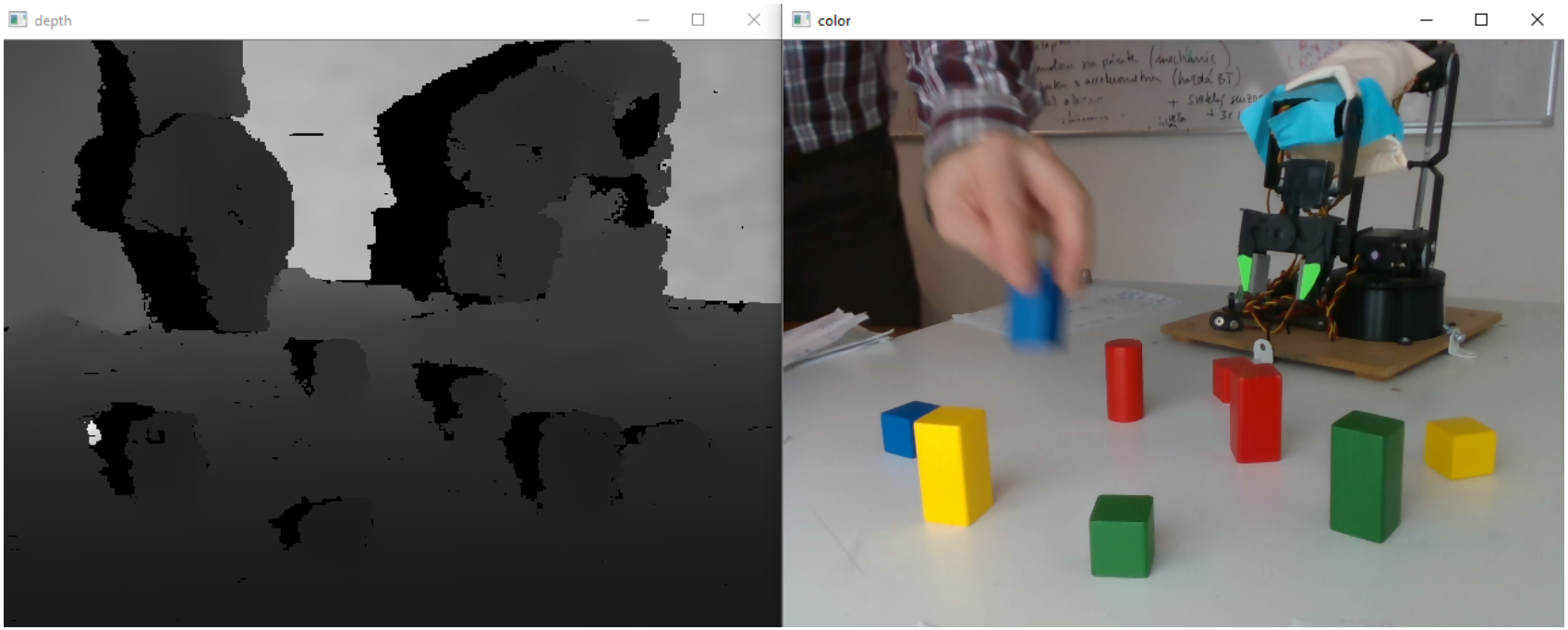
Čitanie dát z kamery - pipeline a align

```
import pyrealsense2 as rs

def retrieve_aligned_pipeline(width=640, height=480, fps=15
    cfg = rs.config()
    cfg.enable_stream(rs.stream.depth, width, height,
                      rs.format.z16, fps)
    cfg.enable_stream(rs.stream.color, width, height,
                      rs.format.rgb8, fps)

    pipeline = rs.pipeline()
    p_cfg = pipeline.start(cfg)

    align_to = rs.stream.color
    align = rs.align(align_to)

    return pipeline, align
```

Čitanie dát z kamery - základný cyklus

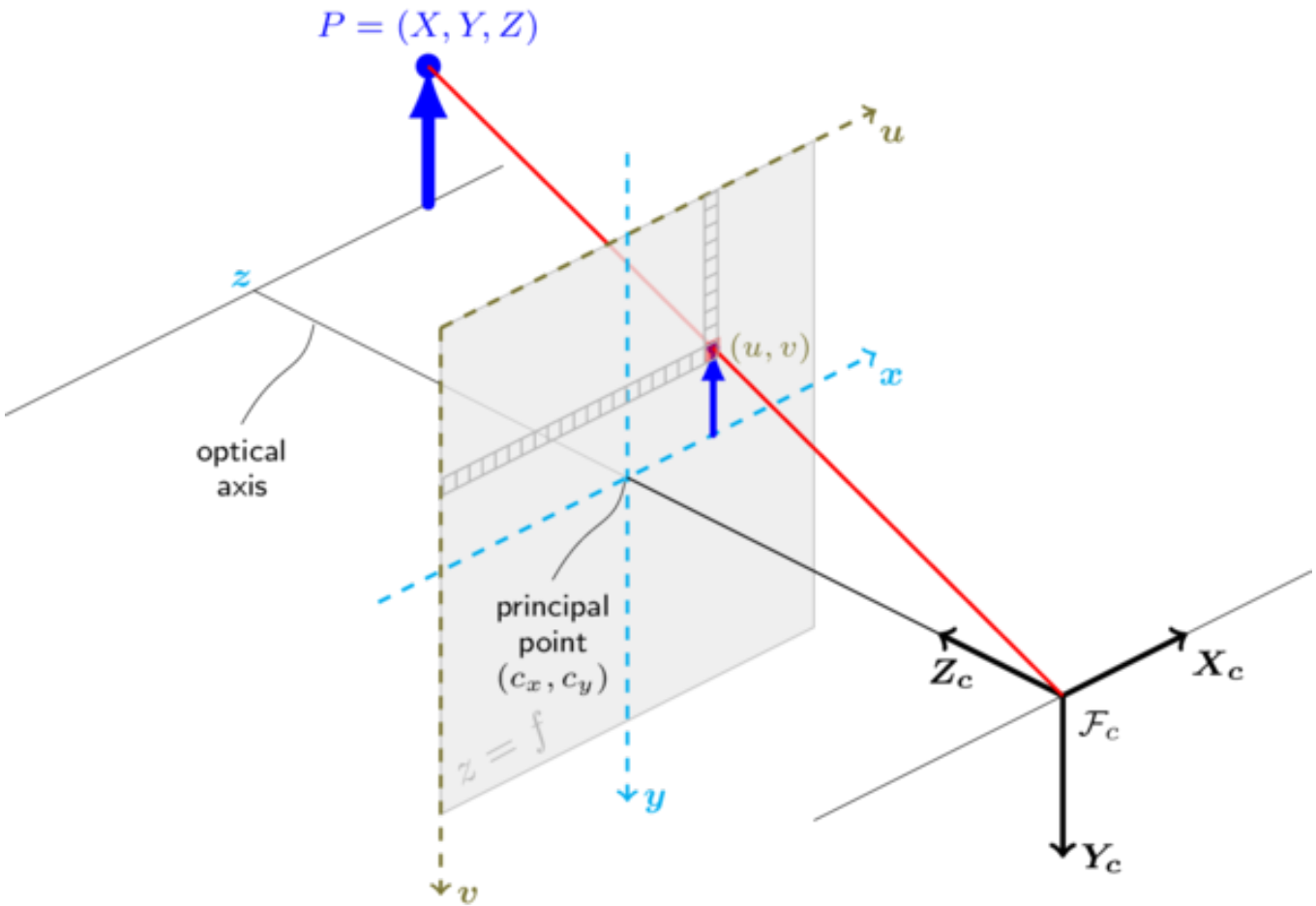
```
import numpy as np

set_dev_preset("presets/ShortRangePreset.json")
pipeline, align = retrieve_aligned_pipeline()
try:
    while (True):
        frames = pipeline.wait_for_frames(timeout)
        aligned_frames = align.process(frames)

        depth_frame = aligned_frames.get_depth_frame()
        color_frame = aligned_frames.get_color_frame()

        depth_image = np.asanyarray(depth_frame.get_data())
        color_image = np.asanyarray(color_frame.get_data())

        #do something

finally:
    pipeline.stop()
```

Čitanie dát z kamery - farebný obraz a hĺbka




```
import cv2
cv2.imshow(depth_image / np.max(depth_image))
cv2.imshow(color_image[:,:,::-1])
cv2.waitKey(0)
```

Výpočet pointcloudu



$$Z = z,\ X = \frac{z}{f_x}(u - c_x),\ Y = \frac{z}{f_y}(v - c_y)$$

Výpočet pointcloudu - kód

```
def create_xyz(depth):
    depth = depth / 1000
    height = intrinsics['height']
    width = intrinsics['width']
    xyz = np.zeros([height, width, 3], dtype=np.float32)
    xyz[:, :, 2] = depth
    mg = np.mgrid[0: height, 0: width]
    xyz[:, :, 0] = (mg[1, :, :] - intrinsics['ppx'])
                   * depth / intrinsics['fx']
    xyz[:, :, 1] = (mg[0, :, :] - intrinsics['ppy'])
                   * depth / intrinsics['fy']

    return xyz
```

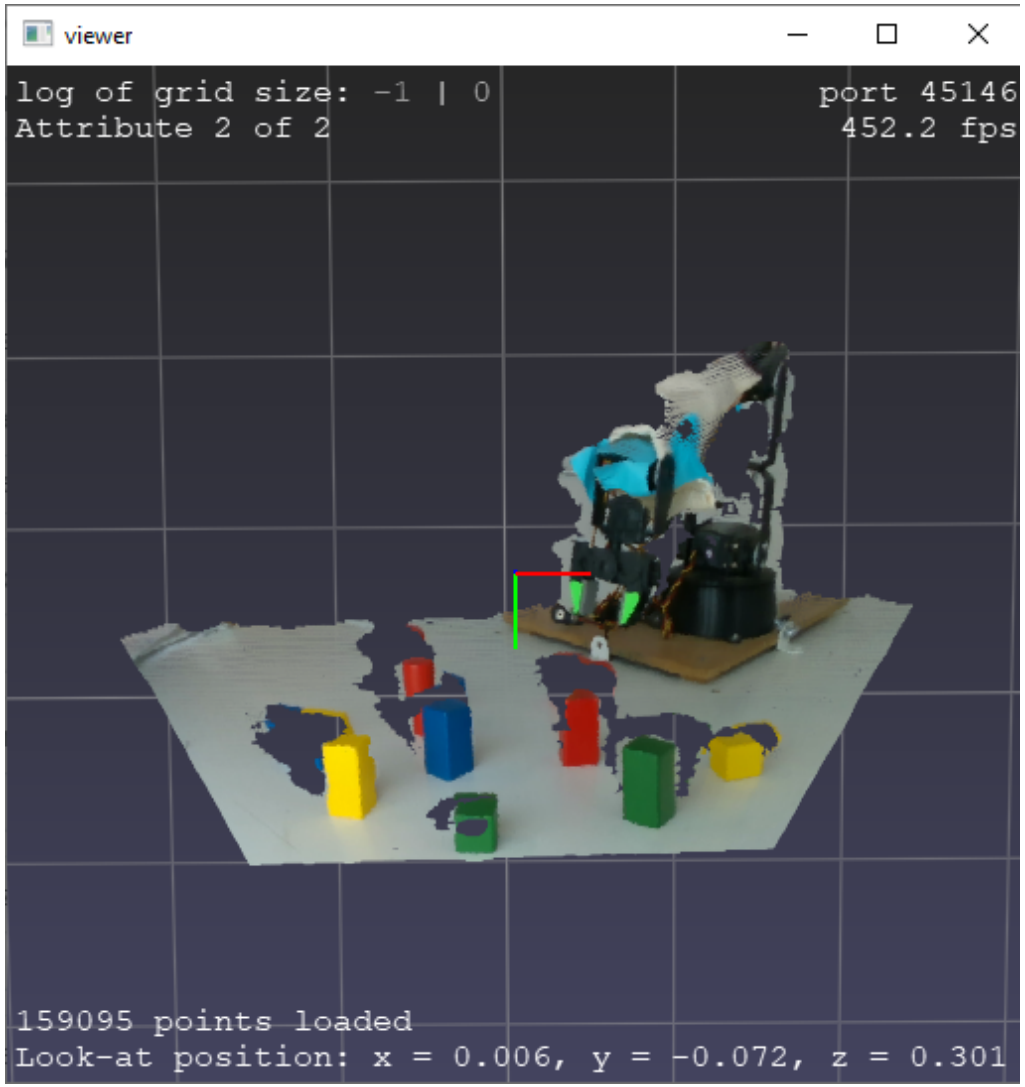
Výpočet pointcloudu - redukcia

```
def reduce_pts(color_image, xyz):
    c = np.logical_and.reduce([~np.isnan(xyz[:, :, 0]),
                               xyz[:, :, 2] > min_limit,
                               xyz[:, :, 2] < max_limit,
                               xyz[:, :, 2] != 0])
    l = np.where(c)
    return color_image[l], xyz[l]
```

Zobrazenie pointcloudu

```
import pptk

viewer = pptk.viewer(xyz)
viewer.attributes(color_image / 255)
viewer.wait()
viewer.close()
```

Pointcloud viewer



Farebné priestory

RGB priestor

Farby reprezentujeme RGB trojicou, buď ako uint8 (0-255), alebo ako float (0-1.0). Tieto hodnoty tvoria priestor na ktorom môžeme zaviesť metriku. A merať tak vzdialenosti medzi farbami.

Iné priestory

RGB priestor však nieje vhodný pre rozlišovanie farieb. Existujú rôzne farebné priestory ktoré vedia reprezentovať farby napr. HSV, CMYK, YUV, CIE Lab atď.

CIE Lab

CIE Lab

Budeme využívať priestor CIE Lab. V ňom máme namiesto RGB trojice trojicu Lab, kde L reprezentuje svetlosť - luminance (0-100), a reprezentuje pozíciu na červeno-zelenej ose a b reprezentuje pozíciu na modro žltej ose. Tento priestor je vhodný na segmentáciu, pretože lepšie korešponduje s ľudským vnímaním priestoru.

Segmentácia pomocou CIE Lab

Farby s ktorými porovnávame

Najprv si musíme zistiť farby s ktorými porovnávame. To spravíme tak, že si manuálne vyberieme z obrázkov časti na ktorý sa nachádzajú farby. Potom z týchto častí spočítame pra každú zložku Lab spočítame priemernú hodnotu a štandardnú odchylku.

Výpočet vzdialenosti od farby

Pre každý pixel p v obrázku počítame hodnotu d_f pre farbu f s priemermi $\mu_f^L, \mu_f^a, \mu_f^b$ a štandardnými odchylkami $\sigma_f^L, \sigma_f^a, \sigma_f^b$ nasledovne:

$$d_f = \sqrt{\sum_{i \in [L,a,b]} \left(\frac{p^i - \mu_f^i}{\sigma_f^i} \right)^2}$$

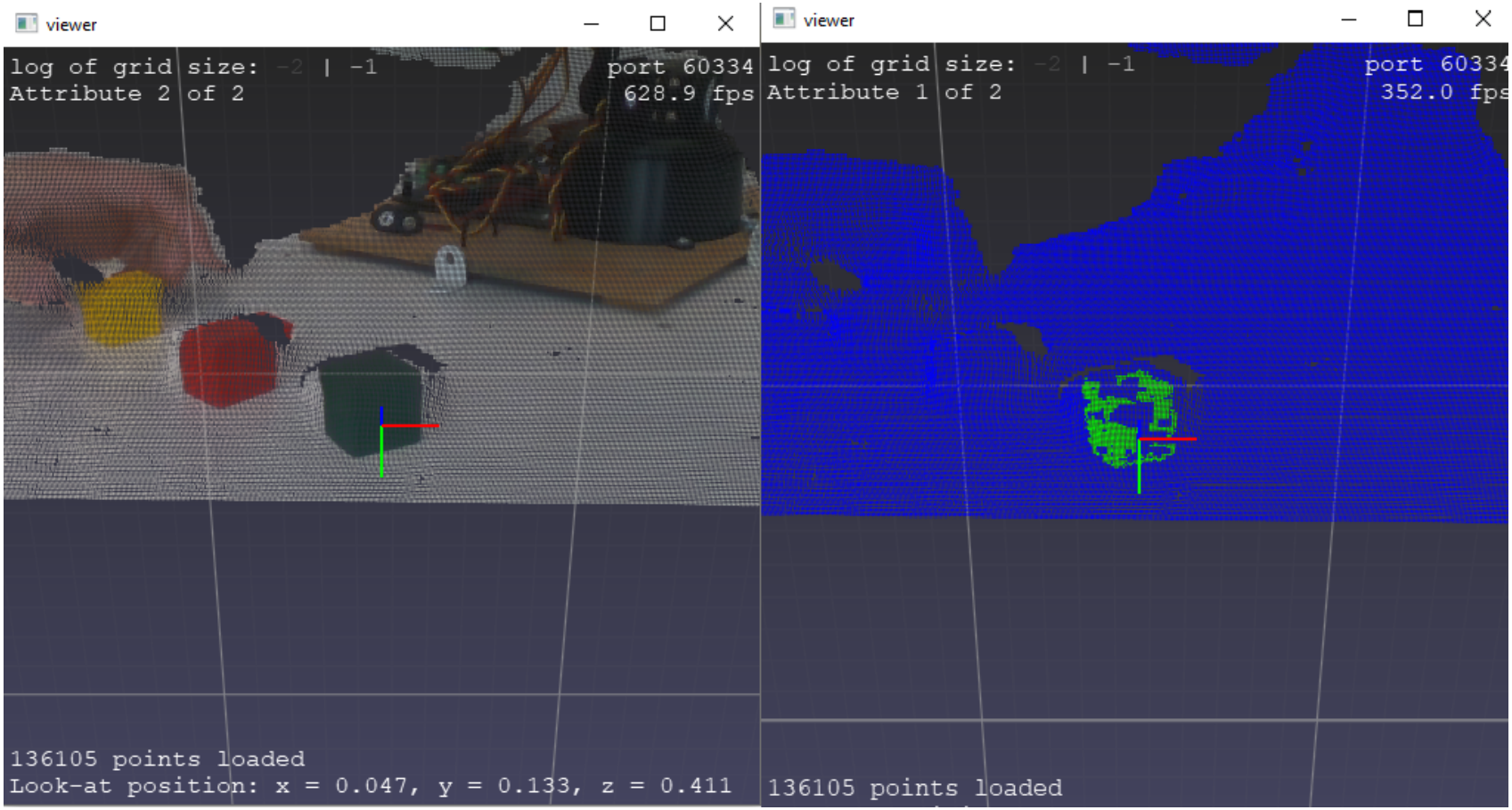
Nájdenie bodov v objektoch

Prah

Pri prvom porovnávaní budeme používať relatívne prísny prah, aby sme našli len zopár pixelov o ktorých sa ale s veľkou pravdepodobnosťou dá povedať, že patria želaným objektom.

```
lab = cv2.cvtColor(color_image[:, np.newaxis, :],
                   cv2.COLOR_RGB2LAB)[:, 0, :]
stddev = np.divide(1, stddev)
col_dist = np.sum(np.square((lab - color) * stddev),
                  axis=-1)

l_col_match = np.asarray(col_dist < t1).nonzero()[0]
```

Nájdenie bodov v objektoch



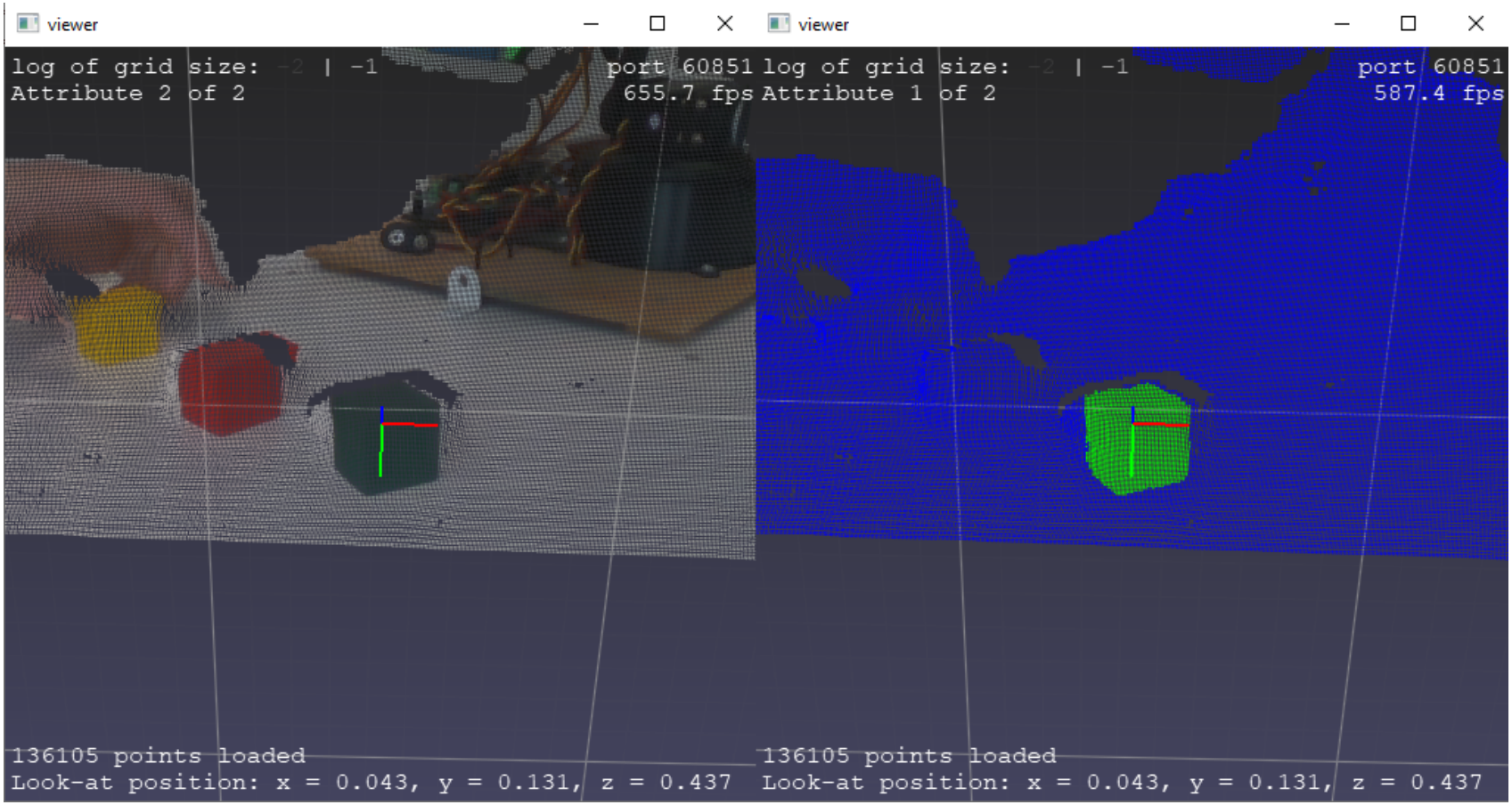
Nájdenie bodov v objektoch

Druhé prahovanie

Pri druhom porovnávaní využijeme znalosť priemernej pozície pixelov z prvého prahovania. Pri využití vyšieho prahu nám tak nebude vadiť napr. pixely z pozadia.

```
col_center = np.average(xyz[l_col_match], axis=0)
spatial_dist = np.sum((xyz - col_center) ** 2, axis=-1)

c_final = np.logical_and.reduce([col_dist < t2,
                                 spatial_dist < d_t])
l_final = np.asarray(c_final).nonzero()[0]
```

Nájdenie bodov v objektoch



Viacero objektov jednej farby

Viacero objektov

Tento prístup nám však zlyhá pri 2 a viac objektoch rovnakej farby.

MeanShift

Potrebujeme teda body, ktoré získame z prvého prahovania klustrovať. Na to môžeme použiť algoritmus MeanShift.

MeanShift

Iterácia

Každý bod budeme posúvať do novej pozície až do konvergencie:

$$x := \frac{\sum_{i \in N(x)} K(x_i - x) x_i}{\sum_{i \in N(x)} K(x_i - x)},$$

kde $N(x)$ udáva tie body pre ktoré $K(x_i - x) \neq 0$.

Kernelové funkcie

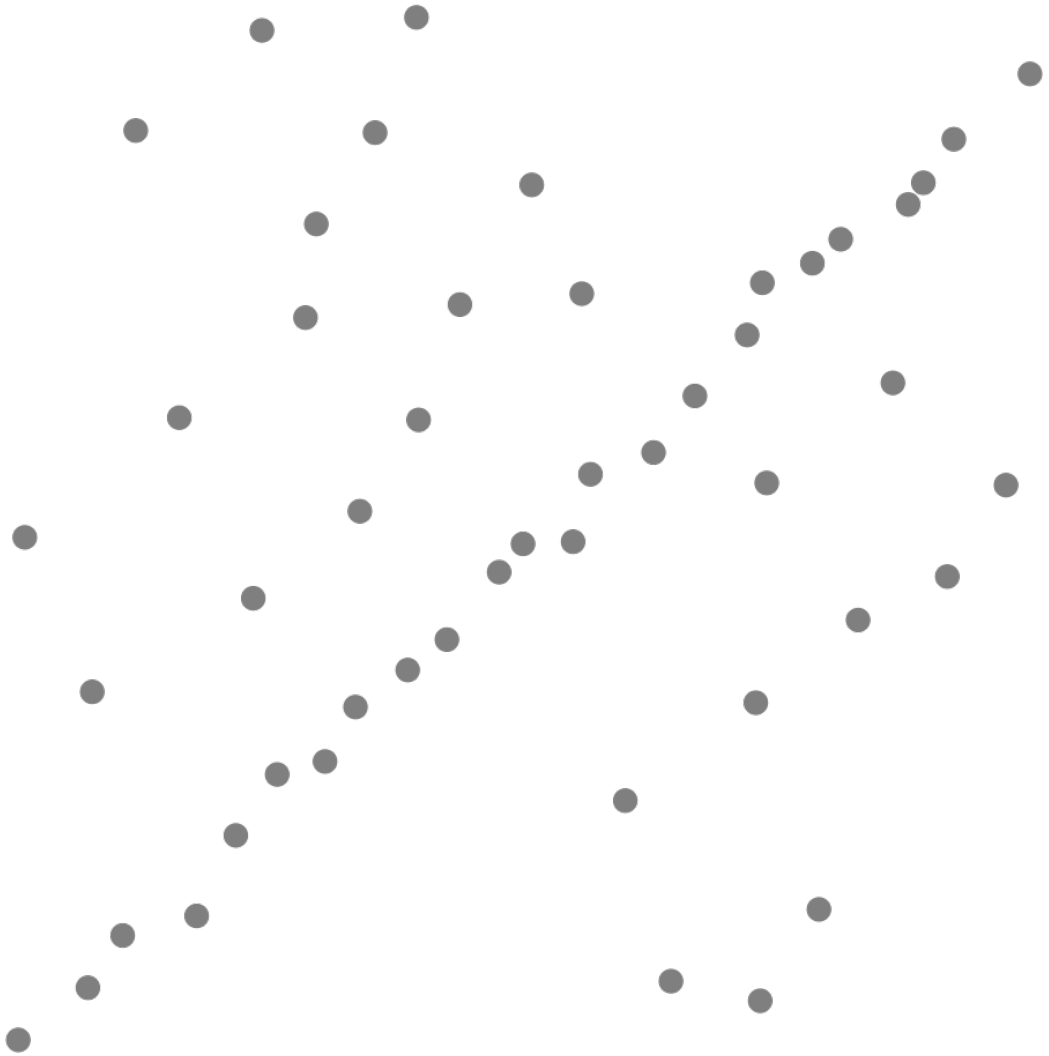
Kernelové funkcie sú rôzne, ale v našej implementácii použijeme funkciu:

$$K(x) = \begin{cases} 1 & \text{if } \|x\| \leq h \\ 0 & \text{if } \|x\| > h \end{cases}$$

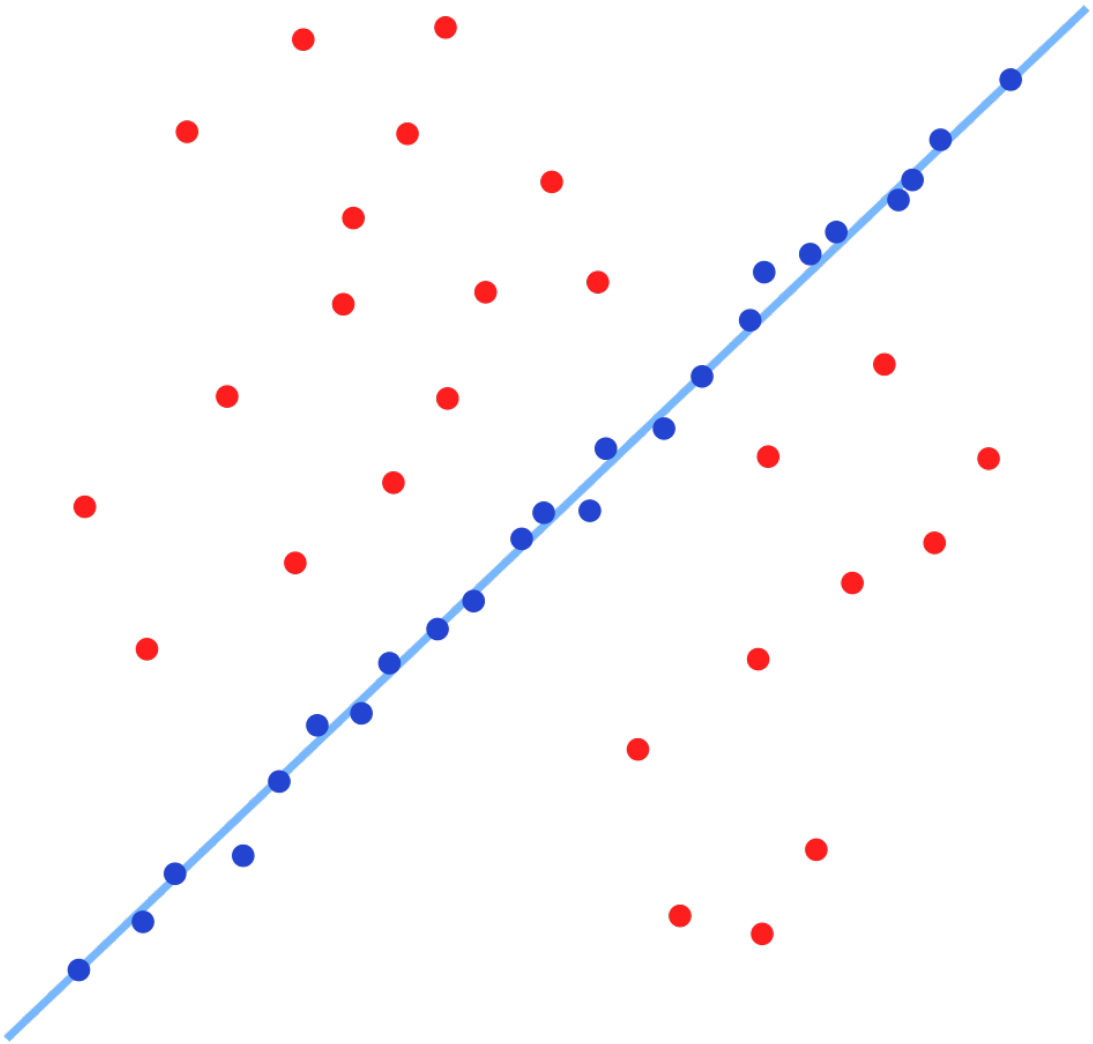
Mean Shift po prvom prahovaní

```
from sklearn.cluster import MeanShift
ms = MeanShift(bandwidth=2 * object_bandwidth,
               bin_seeding=True)
ms.fit(xyz[l_col_match])
ms_labels = ms.labels_
labels_unique = np.unique(ms_labels)

pts_list = []
for label in labels_unique:
    l_cluster = np.asarray(ms_labels == label).nonzero()[0]
    col_center = np.average(xyz[l_col_match[l_cluster]],
                            axis=0)
    spatial_dist = np.sum((xyz - col_center) ** 2, axis=-1)
    c_final = np.logical_and.reduce([col_dist < t2,
                                     spatial_dist < d_t])
    l_final = np.asarray(c_final).nonzero()[0]
    if len(l_final) > 100:
        pts_list.append(xyz[l_final])
```

Idea RANSACu



Idea RANSACu



RANdom SAmple Consensus Algorithm

Cieľ je nájsť parametre modelu, tak aby najlepšie sedeli na dáta.
Algoritmus:

1. Vyberieme náhodnú podmnožinu dát
2. Nafitujeme model na túto podmnožinu
3. Otestujeme model voči všetkým dátam
4. Iterujeme pokiaľ nesplníme nejaké kritérium

Pointcloud a RANSAC

V našom prípade sú dáta vektory z $\mathbb{R}^3$.

Modelujeme najmä jednoduché objekty:

- Priamka - 2 body
- Rovina - 3 body
- Guľová plocha - 4 body
- Valec - 4 body

Kritéria ukončenia

Maximálny počet iterácii

Najjednoduchším kritériom je určenie si maximálneho počtu iterácií a vybrať najlepší zo zatiaľ testovaných modelov.

Vyhodnotenie modelu

Model vyhodnocujeme väčšinou počtom dát, ktoré súhlasia s modelom. Pri jednoduchých geometrických modeloch si určíme napr. vzdialenosť od objektu pri ktorej si povieme, že dátový bod súhlasí s modelom. Toto môžeme nahradiť aj nejakým iným vyhodnotením.

Geometria

Rovina

Rovina je množina vektorov $\vec{x} \in \mathbb{R}^3$ pre ktorú platí: $ax_1 + bx_2 + cx_3 + d = 0$. Vektor $\vec{n} = (a, b, c)$ je normálový vektor roviny. Normálový vektor spolu s parametrom d nieje pre rovinu jednoznačný, ale to nám nevadí.

Rovina z 3 bodov

Majme tri body $\vec{p}_1, \vec{p}_2, \vec{p}_3$. Potom parametre roviny v ktorej ležia môžeme určite nasledovne:

$$\vec{n} = (\vec{p}_2 - \vec{p}_1) \times (\vec{p}_3 - \vec{p}_1)$$

$$d = -\langle \vec{n} | \vec{p}_3 \rangle$$

Geometria

Vzdialenosť bodu od roviny

Majme rovinu definovanú vektorom $\vec{P} = (a, b, c, d)$. Body $\vec{x} \in \mathbb{R}^3$ budeme reprezentovať v tzv. homogénnych súradniciach ako $\vec{X} = (x_1, x_2, x_3, 1) \in \mathbb{R}^4$. Potom vzdialenosť bodu od roviny počítame pomocou:

$$dist = \frac{\left\langle \vec{P} | \vec{X} \right\rangle}{||\vec{n}||}$$

Rovina z 3 bodov

```
def get_plane_3pts(p1, p2, p3):
    v1 = p3 - p1
    v2 = p2 - p1
    cp = np.cross(v1, v2)
    d = -np.sum(cp * p3)
    return np.concatenate([cp, d], axis=-1)
```

RANSAC pre rovinu

```
def find_first_plane(xyz_h):
    n = xyz_h.shape[0]
    bestP = None
    bestC = 0
    for i in range(64):
        idxs = np.random.choice(n, 3, replace=False)
        P = get_plane_3pts(xyz_h[idxs[0]],
                           xyz_h[idxs[1]],
                           xyz_h[idxs[2]])
        sp = np.matmul(xyz_h, P.T)
        C = np.sum(sp < 1e-3 * np.linalg.norm(P[:3]))
        if C > bestC:
            bestC = C
            bestP = P
    return bestP
```

Vektorizácia

Vieme predchádzajúci kód zefektívniť?

V predchádzajúcom kóde máme for cyklus. Naskytá sa teda otázka, či nieje možné nahradiť ho vektorizovanými (v našom prípade skôr vektorizovanejšími) operáciami

Riešenie

Potrebujeme teda najprv vygenerovať namiesto 64 vektorov P jednu maticu s rozmermi 64×4, kde každý riadok bude predstavovať jednu rovinu.

Rovina z 3 bodov

```
def get_plane_3pts(p):
    """ Returns a plane vector diven three points

    :param p: array of shape 3 x n x 3, where n is number
              of planes, vectors are in the last dim
    :return: n x 4 array with plane spec such in each
             row coresponds to a, b, c, d in
             a*x + b*y + c*z + d = 0
    """
    v1 = p[2] - p[0]
    v2 = p[1] - p[0]
    cp = np.cross(v1, v2)
    d = -np.expand_dims(np.sum(cp * p[2], axis=-1), axis=-1
    return np.concatenate([cp, d], axis=-1)
```

Vektorizácia

Maticové násobenie

Nech $\mathbb{A} \in \mathbb{R}^{p \times q}, \mathbb{B} \in \mathbb{R}^{q \times r}, \mathbb{C} \in \mathbb{R}^{p \times r}$, potom:

$$\mathbb{C} = \mathbb{A}\mathbb{B} \iff C_{ij} = (\forall i \in \hat{p})\,(\forall j \in \hat{r}) \sum_{k=1}^{q} A_{ik} B_{kj}$$

Maticové násobenie v RASNSACu

Maticové násobenie môžeme využiť pri výpočte vzdialenosti bodov od rovín bez použitia for cyklu.

RANSAC pre rovinu - vektorizovaná verzia

```
row_i = np.random.choice(n, [3, 64])
P = get_plane_3pts(xyz_h[row_i, :3])
d = np.abs(np.matmul(xyz_h, P.T))
n = np.sqrt(np.sum(P[:, :3] ** 2, axis=-1))
l = d < 1e-3 * n
good = np.sum(l, axis=0)
best_idx = np.argmax(good)
bestP = P[best_idx]
```

Body patriace rovine

```
l = np.abs(np.matmul(xyz_h, bestP)) <
            2e-3 * np.sqrt(np.sum(bestP[:3] ** 2))
pts = xyz_h[(l).nonzero()[0], :]
r_xyz_h = xyz_h[(~l).nonzero()[0], :]
```

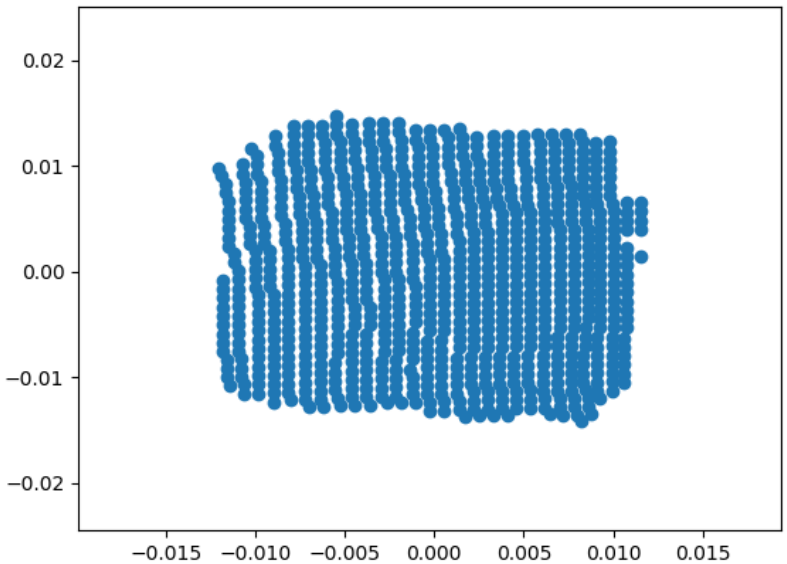
Súradnice roviny

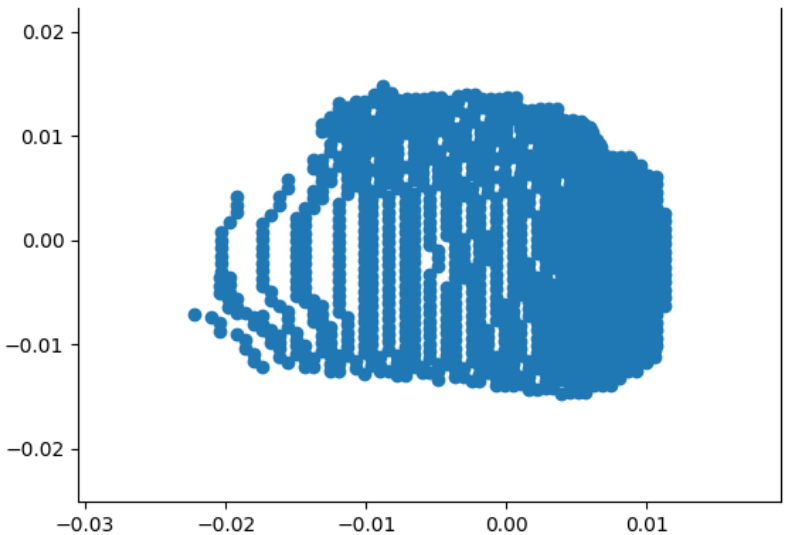
Ak nájdeme $\vec{p}_1, \vec{p}_2$, ktoré su kolmé navzájom a na normálu roviny. Navyše ak $||\vec{p}_1|| = ||\vec{p}_2|| = 1$, tak pre každý bod x nájdeme jeho priemet nového súradného systému ako $x_r = \langle \vec{x} - \vec{c} | \vec{p}_1 \rangle, y_r = \langle \vec{x} - \vec{c} | \vec{p}_2 \rangle$, kde vektor $\vec{c}$ sa premietne na bod (0, 0) v novom systéme.

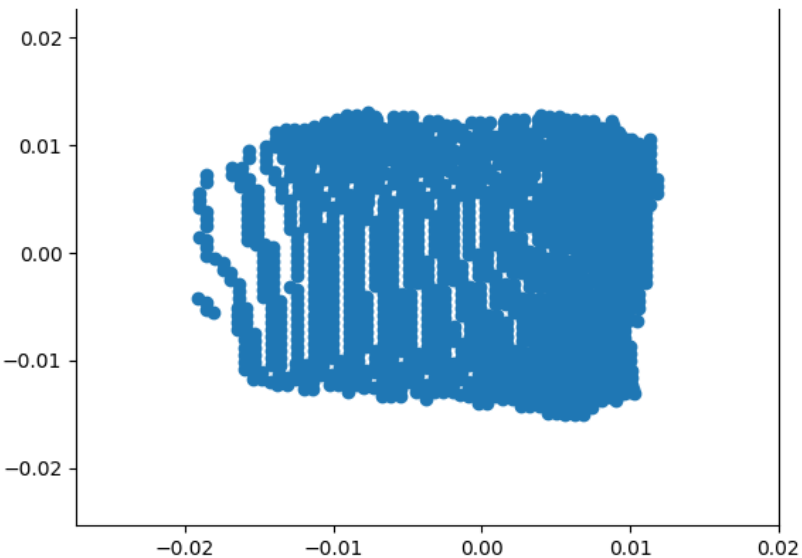
Spracovanie bodov v rovine

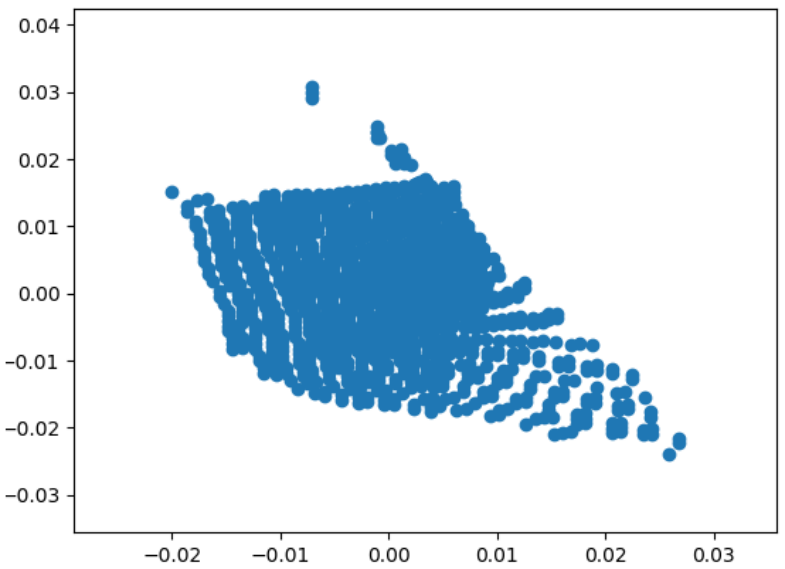
```
P = np.expand_dims(bestP, axis=-1)
xyz_c = np.mean(pts, axis=0)
dist = np.matmul(xyz_c, P)
Pn = np.divide(P[:3], np.linalg.norm(P[:3]))
xyz_c = xyz_c[:3] - dist * Pn.T

p1 = np.array([P[2, 0], 0, -P[0, 0]], dtype=np.float32)
p1 = p1 / np.linalg.norm(p1)
p2 = np.cross(p1, P[:3, 0].T)
p2 = p2 / np.linalg.norm(p2)

x = np.sum(p1 * (pts[:, :3] - xyz_c), axis=1)
y = np.sum(p2 * (pts[:, :3] - xyz_c), axis=1)
```

Body v nových súradniciach









Určenie orientácie

Orientácia obdĺžnika

Našim cieľom v tomto kroku bude zistiť orientáciu obdĺžnika projektovaného na rovinu získanú RANSACom.

Konvexný obal

Budú nám stačiť body na obvode útvaru. Preto použijeme konvexný obal. Z minulého slidu vieme, že je možné mať body, ktoré sú falošné a budú kaziť detekciu. Preto budeme orientáciu určovať z konvexného obalu pre náhodnú podmnožinu bodov a nie pre celú množinu bodov.

Určenie orientácie

Orientácia

Body konvexného obalu sú usporiadané proti smeru hodinových ručičiek. Použiejeme preto vektor rozdielu po sebe idúcich bodov na určenie ich smeru. Tieto smery potom spriemerujeme aby sme dostali želaný priemer.

Cirkulárny priemer

Pre cirkulárne hodnoty nemá zmysel počítať aritmetický priemer. Napríklad priemer uhlu 359° a 1°, by sme čakali ako 0°, ale s aritmetickým priemerom to bude 180°. Definujme preto cirkulárny priemer pre hodnoty s periódou 2π:

$$\bar{\alpha} = \text{atan2}\left(\frac{1}{n}\sum_{j=1}^{n}\sin\alpha_j, \frac{1}{n}\sum_{j=1}^{n}\cos\alpha_j\right)$$

Určenie orientácie

Orientácia obdĺžnika

Keďže obdĺžnik je symetrický pre rotácie o 90°. Perióda pre cirkulárny priemer by teda mala byť $\frac{\pi}{2}$. To docielime tak, že každý uhol α upravíme na β nasledovným spôsobom:

$$\beta = 4\left(\alpha \bmod \frac{\pi}{2}\right)$$

Potom spočítame cirkulárny priemer $\bar{\beta}$. Z neho dostaneme náš žiadaný priemer nasledovne:

$$\bar{\alpha} = \frac{\bar{\beta}}{4}$$

Kód

```
xy = np.column_stack([x, y])
s, c, ns = 0, 0, 0

for _ in range(num_repeats):
    hull_pts = cv2.convexHull(xy[np.random.choice(
                                  xy.shape[0], xy.shape[0]
                                  // num_repeats)]
                                  .astype(np.float32))
    ns += hull_pts.shape[0]
    lines = hull_pts[:, 0, :] - np.roll(hull_pts[:, 0, :],
                                        -1, axis=0)
    alphas = np.arctan2(lines[:, 0], lines[:, 1])
    alphas = np.mod(alphas, np.pi/4)
    alphas *= 4
    s += np.sum(np.sin(alphas))
    c += np.sum(np.cos(alphas))

alpha_final = np.arctan2(c / ns, s / ns) / 4
direction = np.array([np.cos(alpha_final),
                      np.sin(alpha_final)])
```

Súradnice zarované s obdĺžnikom

Nový systém súradníc

Smer z posledného slidu nám umožní vytvoriť nový suradnicový systém, ktorý je zarovananý s našim predpokladaným obdĺžnikom.

```
pp1 = np.matmul(np.column_stack([p1, p2]), direction)
pp1 = pp1 / np.linalg.norm(pp1)
pp2 = np.cross(pp1, P[:3, 0].T)
pp2 = pp2 / np.linalg.norm(pp2)

x = np.sum(pp1 * (pts[:, :3] - xyz_c), axis=1)
y = np.sum(pp2 * (pts[:, :3] - xyz_c), axis=1)
```

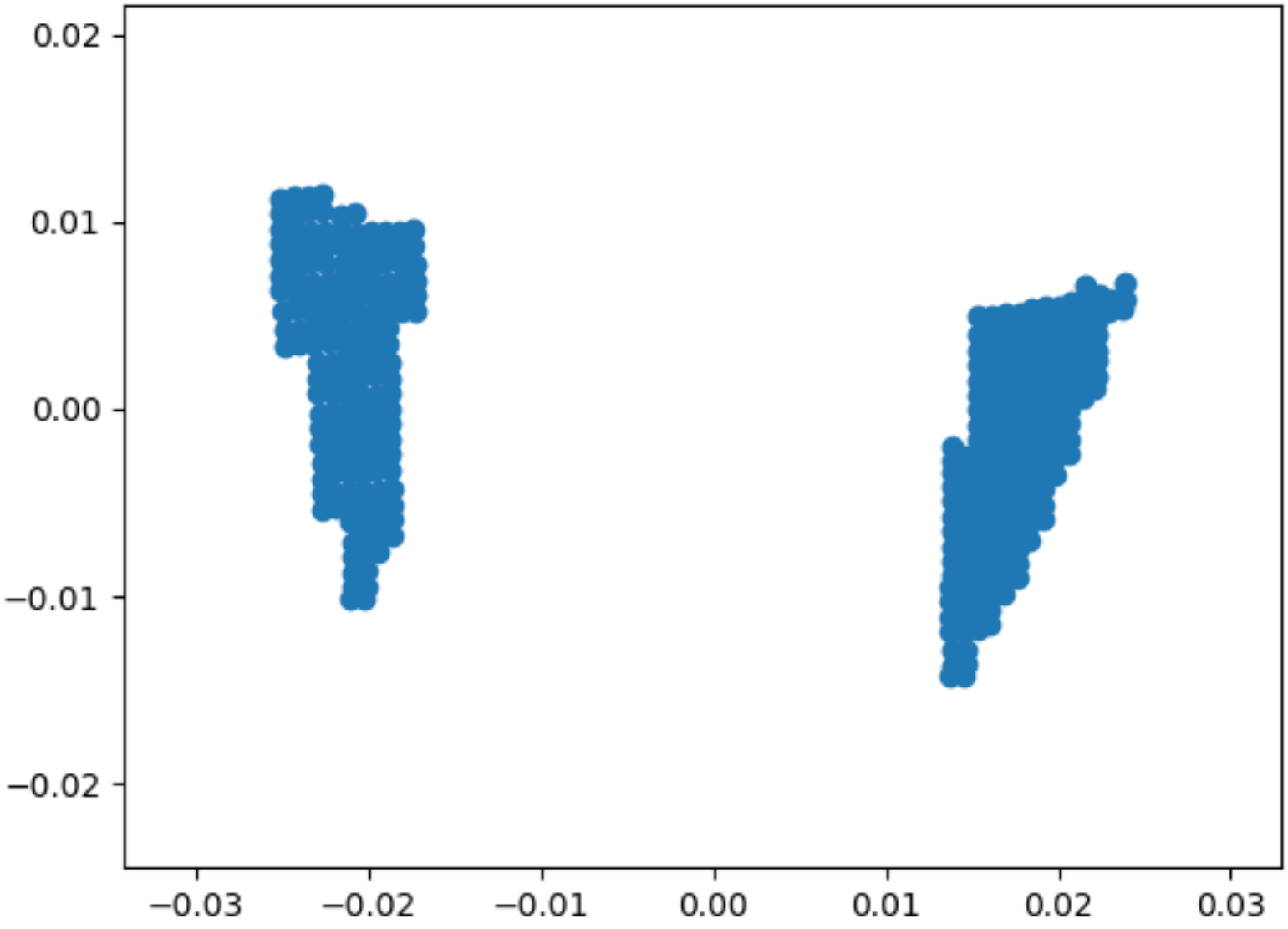
Rozmery objektu

Rozmery objektu

V novom súradnom systéme si spočítame rozmery objektu jednoducho podľa maxímálnych a minimálnych hodnôt x-ových a y-ových súradníc. Potom môžeme nájsť rohy kvádru.

```
x_m, y_m, x_dim, y_dim, z_dim = get_cuboid_dims(x,y)
xy_corners = np.array([[x_m - x_dim / 2, y_m - y_dim / 2],
                       [x_m - x_dim / 2, y_m + y_dim / 2],
                       [x_m + x_dim / 2, y_m + y_dim / 2],
                       [x_m + x_dim / 2, y_m - y_dim / 2]],
                       dtype=np.float32)

corners = xyz_c + np.matmul(np.column_stack([pp1, pp2]),
                            xy_corners.T).T
nP = P[:3, 0].T / np.linalg.norm(P[:3, 0].T)
if nP[2] < 0:
    nP = -nP
corners = np.row_stack([corners, corners + z_dim * nP])
```

Robotická ruka v novvých súradniciach



Určenie orientácie

Orientácia graspera

Rovinu graspera sme našli obdobne ako pre kvádre. Budeme potrebovať opäť zarovnať súradnicový systém. To spravíme tak, že najprv rozdelíme body do dvoch klustrov vďaka k-means clusteringu. Potom aplikujeme algoritmus LDA, tak aby sme našli smer v ktorom sú tieto dva klustre najlepšie separovateľné.

LDA

LDA (Linear Discriminant Analysis) je metóda hľadá lineárnu kombináciu príznakov (v našom prípade x, y), ktorá je najlepšia pre klasifikáciu dát. Dostaneme teda smer, ktorý najlepšie rozdeľuje dve triedy.

Orientácia graspera

```
labels = KMeans(n_clusters=2).fit_predict(xy)
lda = LDA()
lda.fit(xy, labels)
w = lda.coef_

pp1 = w[0, 0] * p1 + w[0, 1] * p2
pp1 = pp1 / np.linalg.norm(pp1)
pp2 = np.cross(pp1, P[:3, 0].T)
pp2 = pp2 / np.linalg.norm(pp2)

x = np.sum(pp1 * (pts[:, :3] - xyz_c), axis=1)
y = np.sum(pp2 * (pts[:, :3] - xyz_c), axis=1)
```

Relevantné body

```
if np.median(y) < 0:
    pp2 = np.cross(-pp1, P[:3, 0].T)
    pp2 = pp2 / np.linalg.norm(pp2)
    y = np.sum(pp2 * (pts[:, :3] - xyz_c), axis=1)

x_0_min = np.percentile(x[labels == 0], 5)
x_1_min = np.percentile(x[labels == 1], 5)

if x_0_min < x_1_min:
    label_left, label_right = 0, 1
else:
    label_left, label_right = 1, 0

    y_bottom = np.min(y)
    y_top = y_bottom + 0.010
    x_left = np.percentile(x[labels == label_left], 95)
    x_right = np.percentile(x[labels == label_right], 5)
```

Detegované body