

Driverless \Perception Report

Andrea Boarini

April 2025

Contents

1	Introduzione	2
2	Task 1, Load and Display	2
3	Task 2, Cone Detection	2
3.1	Stabilire i range	2
3.2	Estrarre le maschere	3
3.3	Processare le maschere	3
3.4	Identificare i coni	4
4	Task 3, Object classification	5
4.1	Problema dei doppioni	7
5	Considerazioni Computer Vision tasks	8
6	Task 4, Extract Track Edges	9
7	Task 5, Odometry	10
7.1	Features detection e descriptors	10
7.2	Matching	11
7.3	Calcolo della matrice essenziale	11

1 Introduzione

L'intenzione del seguente elaborato è quella di evidenziare tutti i passaggi eseguiti e i metodi applicati al fine di risolvere la challenge proposta nella seguene repository Github: <https://github.com/eagletrt/driverless/perception>.

In aggiunta, verrà data particolare attenzione a quelle che sono state le difficoltà incontrate nella ricerca delle soluzioni ai problemi proposti e, più nello specifico, verranno presentate delle possibili ottimizzazioni.

Si elencano di seguito tutti i *tools* che hanno reso possibile la realizzazione del progetto:

- OpenCV library (v 4.11.0)
- C++ language
- CMake e CMakeLists.txt per la compilazione

2 Task 1, Load and Display

In questa prima task viene richiesto di caricare e stampre a video le immagini che verranno utilizzate per le diverse challenge di computer vision nel progetto.

Questo segmento non ha causato particolari difficoltà implementative: è stato sufficiente avvalersi delle funzioni già implementate nella librera OpenCV:

```
string imagePath1 = "driverless_perception/frame_1.png";
string imagePath2 = "driverless_perception/frame_2.png";

Mat read1 = imread(imagePath1, IMREAD_COLOR);
Mat read2 = imread(imagePath2, IMREAD_COLOR);

imshow("frame_1", read1);
imshow("frame_2", read2);
```

3 Task 2, Cone Detection

La fase di *Cone detection* è stata la parte più laboriosa dell'intero progetto. La prima difficoltà incontrata è stata trovare un range di valori valido che potesse rappresntare lo spazio colore dei coni da identificare. Ho scelto di lavorare nello spazio colore HSV (*Hue, Saturation, Value*) per poter avere maggiore precsione e flessibilità nel momento in cui si è presentata la necessità di invocare i moduli per il riconoscimento dei contorni.

3.1 Stabilire i range

Per iniziare ho stabilito **nello scope globale** i valori limite dei vari range di colore in modo tale da includere più coni possibili in fase di riconoscimento. Questo processo è estremamente delicato in quanto, alla minima variazione di condizione di luce, si presenta il rischio di fuoriuscire dal range stabilito e non identificare quindi l'oggetto.

```
Scalar lowBlue(100, 100, 50);
Scalar highBlue(130, 255, 255);
Scalar lowRed1(0, 160, 160);
Scalar highRed1(10, 255, 255);
Scalar lowRed2(160, 160, 160);
Scalar highRed2(179, 255, 255);
Scalar lowYellow(10, 150, 150);
Scalar highYellow(25, 255, 255);
```

Si noti la presenza di due intervalli per il colore rosso. Questo è dovuto al fatto che il campo H (*Hue*) nello spazio HSV è codificato su una circonferenza e, dunque, assume valori compresi tra 0 e 360 (a livello di codice si sfrutta la codifica a 8 bit dell'immagine e quindi i valori numerici variano leggermente), con le sfumature del rosso esattamente intersecate dall'asse zero.

3.2 Estrarre le maschere

Con gli intervalli trovati sono state estratte le maschere colore dall'immagine originale.

```
void extractColorMask(Mat& src, Scalar lower, Scalar higher, Mat& outputColor,
Scalar lower2 = Scalar(0,0,0), Scalar higher2 = Scalar(0,0,0))
{
    Mat hsv, primary, secondary;
    cvtColor(src, hsv, COLOR_BGR2HSV);
    inRange(hsv, lower, higher, primary);
    if (lower2 != Scalar(-1, -1, -1) && higher2 != Scalar(-1, -1, -1)) {
        inRange(hsv, lower2, higher2, secondary);
        bitwise_or(primary, secondary, outputColor);
    } else {
        outputColor = primary.clone();
    }
}
```

La matrice sorgente passata come parametro viene convertita nello spazio colore HSV e tramite la funzione `void cv::inRange(...)` sono stati salvati in una matrice di destinazione solo i pixel rientranti nel range specificato.

La funzione `extractColorMask` è stata implementata per operare sia con range unici (come con il giallo e il blu) sia con range multipli (il caso del rosso), che richiedono una fusione di due sottorange.

Il risultato dell'operazione è inserito nella matrice `outputColor` che verrà poi processata nei passaggi successivi.

3.3 Processare le maschere

Per processare le maschere sono state utilizzate tre principali *tools* della libreria OpenCV: `void cv::dilate(...)` e `void cv::erode(...)` per eliminare le bande catarinfrangenti dei coni ed eventuale rumore dell'immagine e `void cv::Canny(...)` per estrarre i contorni dei profili rilevati.

Un aspetto importante è la scelta dei parametri passati a queste funzioni base.

`void cv::dilate(...)` e `void cv::erode(...)` richiedono come parametro un *kernel* che è stato definito come:

```
kernel = getStructuringElement(MORPH_RECT, Size(kSizeA, kSizeB));
```

I parametri di questo oggetto, le sue dimensioni per esempio, sono **fondamentali** per un corretto riconoscimento dei contorni nella fase successiva.

Analogamente, le due funzioni succitate richiedono anche il passaggio del numero di iterazioni che esse devono compiere sull'immagine da processare. Un corretto valore fa la differenza. Nel caso di questo progetto i valori sono stati stabiliti ad hoc cercando di riconoscere più coni possibili (almeno quelli più vicini alla vettura) ma, come si vedrà, alla base degli errori di riconoscimento ci sarà proprio un'errata valutazione dei parametri passati alle funzioni di processamento della maschera.

Per comodità tutta la fase di processamento delle maschere è stata inserita in un'unica funzione:

```
void processColorMask(Mat& colorThreshold, Mat& kernel, int kSizeA, int kSizeB,
int dilateIterations, int erodeIterations, int cannyAperture,
Mat& outputCanny)
{
    kernel = getStructuringElement(MORPH_RECT, Size(kSizeA, kSizeB));
    dilate(colorThreshold, colorThreshold, kernel, Point(-1, -1),
           dilateIterations, MORPH_ELLIPSE);
    erode(colorThreshold, colorThreshold, kernel, Point(-1, -1),
           erodeIterations, MORPH_ELLIPSE);
    Canny(colorThreshold, outputCanny, 30, 100, cannyAperture);
}
```

A seguire si riportano le immagini delle maschere pre e post processamento (range colore del rosso):



Figure 1: maschera identificazione colore

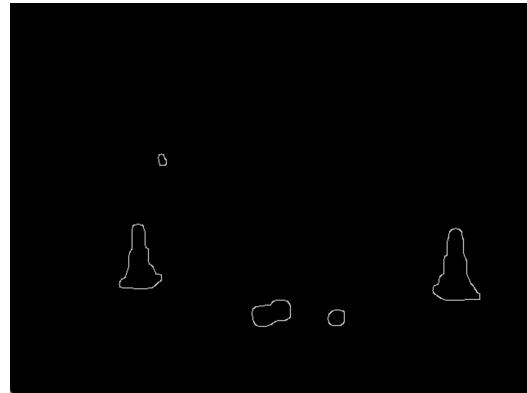


Figure 2: contorni post erosione/dilatazione

3.4 Identificare i coni

Una volta ottenuta la maschera evidenziante i contorni dei possibili coni, il tutto è stato passato alla seguente funzione:

```
void findBoundaries(Mat& cannyMat, Scalar colorRect, string labelName,
vector<vector<Point>>& contours, vector<Cone>& coneVector)
{
    findContours(cannyMat, contours, RETR_TREE, CHAIN_APPROX_SIMPLE);
    for(const auto& cnt : contours) {
        vector<Point> approx;
        Rect r = boundingRect(cnt);
        approxPolyDP(cnt, approx, 0.08*arcLength(cnt, true), true);
        if (approx.size() == 3) {
            coneVector.push_back(Cone(Rect(r.x, r.y, r.width, r.height), labelName,
colorRect));
        }
    }
    contours.clear();
}
```

`findBoundaries` si avvale di `void cv::findContours(...)` per salvare nel vettore `contours` (*vector* di *vector* di punti) tutti gli insiemi di punti formanti una curva chiusa a partire dall'analisi della matrice `cannyMat` passata per riferimento a `findBoundaries`.

Successivamente, tutti gli oggetti in `contours` vengono analizzati dalla funzione `void cv::approxPolyDP(...)` con il fine di isolare quegli insiemi di punti che assumono una forma triangolare. Anche in questo caso, i parametri sono estremamente importanti per la corretta identificazione dei coni. Particolare attenzione è posta sul terzo parametro, esso va a specificare l'**accuratezza dell'approssimazione**. La scelta di utilizzare come valore `0.08 * arcLength(...)` (con `void cv::arcLength(...)` funzione che ritorna il perimetro della curva analizzata) è stata puramente empirica in quanto valori più bassi tendevano a tenere in considerazione e a classificare come coni anche piccole zone di rumore. Ritengo dunque che tale valore possa essere un buon compromesso per il set di immagini fornito ma non necessariamente adatto per immagini nelle quali le condizioni di luce e ombra sono molto differenti.

Infine, tutti quegli insiemi che a seguito dell'approssimazione presentavano 3 vertici (trattandosi dunque di una qualsiasi forma triangolare), sono stati aggiunti ad un vettore di `Cone`, che tiene traccia dei coni rilevati, ossia i rettangoli che fungono da *box* per tali triangoli identificati.

La classe `Cone` è stata progettata nel seguente modo:

```
class Cone {
public:
    Scalar colorClass;
    string classifiedAs;
    Rect boundaries;
    Point center;
    bool isRight;
    Cone() {
        classifiedAs = "";
        boundaries = Rect();
        colorClass = Scalar(255, 255, 255);
        center = Point(0, 0);
        isRight = false;
    }
    Cone(Rect bound, string classAs, Scalar cc) {
        classifiedAs = classAs;
        boundaries = bound;
        colorClass = cc;
        center = Point(bound.x + bound.width/2, bound.y + bound.height/2);
        isRight = (classAs == "Yellow" || (classAs == "Red" && center.x > 0));
    }
    ~Cone() {}
};
```

4 Task 3, Object classification

La fase di *Object Classification* è sostanzialmente consistita nel popolare il vettore `conesFound` contenente tutte le corrispondenze dei coni identificati, da stampare a video sovrapponendole all'immagine fornita come input.

Per ottenere tale risultato è stata definita una classe `RangeColor` le quali istanze rappresentano i singoli range da tenere in considerazione. Successivamente, per ogni range

di colore da tenere in considerazione, nel main, è stato eseguito un ciclo che ad ogni iterazione invoca le funzioni di processamento della maschera secondo i valori del range preso in considerazione.

Struttura della classe RangeColor:

```

class RangeColor {
public:
    string label;
    Scalar valueLow; // hsv coded
    Scalar valueUp;
    Scalar valueLow2; // always =0 if the range doesn't involve red
    Scalar valueUp2;
    Scalar color; // BGR coded

RangeColor(){
    label = "";
    valueLow = Scalar(0, 0, 0);
    valueUp = Scalar(0, 0, 0);
    valueLow2 = Scalar(-1, -1, -1);
    valueUp2 = Scalar(-1, -1, -1);
    color = Scalar(0, 0, 0);
}
RangeColor(string l, Scalar vL, Scalar vU, Scalar clr) {
    label = l;
    valueLow = vL;
    valueUp = vU;
    valueLow2 = Scalar(-1, -1, -1);
    valueUp2 = Scalar(-1, -1, -1);
    color = clr;
}
RangeColor(string l, Scalar vL, Scalar vU, Scalar vL2, Scalar vU2,
Scalar clr)
{
    label = l;
    valueLow = vL;
    valueUp = vU;
    valueLow2 = vL2;
    valueUp2 = vU2;
    color = clr;
}
~RangeColor() {}
};


```

Ciclo nel main per ogni range definito:

```

RangeColor red("Red", lowRed1, highRed1, lowRed2, highRed2, redLabel);
RangeColor yellow("Yellow", lowYellow, highYellow, yellowLabel);
RangeColor blue("Blue", lowBlue, highBlue, blueLabel);
vector<RangeColor> rc = {red, yellow, blue};
vector<vector<Point>> contours;
vector<Cone> conesFound;

for(const auto& range : rc) {
    extractColorMask(read1, range.valueLow, range.valueUp, threshold,
    range.valueLow2, range.valueUp2);

```

```

    processColorMask(threshold, kernel, 3, 3, 11, 8, 7, smoothed);
    findBoundaries(smoothed, range.color, range.label, contours, conesFound);
}

```

4.1 Problema dei doppioni

Per quanto riguarda la classificazione dei coni, l'unico vero imprevisto è stata la presenza di elementi duplicati, all'interno di `conesFound`, che si riferiscono allo stesso oggetto nell'immagine.

Questo è dovuto al fatto che la funzione `void cv::findContours(...)`, specialmente in presenza di lieve rumore, può trovare diversi possibili contorni attorno ad un cono, seppur tutti molto simili. La funzione di approssimazione con grande probabilità considererà tutti questi contorni come delle forme triangolari, indipendentemente dalla lieve differenza.

Una soluzione a questo problema potrebbe essere quella di lavorare meglio sul processamento dell'immagine applicando un **filtro di blur Gaussiano**, filtro particolare che contribuisce a ridurre il rumore generale nell'immagine (specialmente nei punti di stacco cromatico).

Personalmente, per non inficiare nel riconoscimento delle soglie di colore, ho optato per eliminare i duplicati agendo direttamente sul vettore nel seguente modo:

Riordino il vettore per ascissa dei rettangoli che fungono da contorno dei coni:

```

sort(conesFound.begin(), conesFound.end(), [](const Cone& c1, const Cone& c2) {
    return (c1.boundaries.x < c2.boundaries.x);
});

```

Rimuovo tutti gli oggetti pseudo uguali specificando una soglia di tolleranza (pixel di scarto massimo tra due rettangoli affinché vengano comunque considerati uguali. Anche tale **soglia è arbitraria**):

```

int tolerance = 10;
auto toRemove = unique(conesFound.begin(), conesFound.end(), [tolerance]
    (const Cone& c1, const Cone& c2)
{
    return (abs(c1.boundaries.x - c2.boundaries.x) < tolerance &&
        abs(c1.boundaries.y - c2.boundaries.y) < tolerance &&
        abs(c1.boundaries.width - c2.boundaries.width) < tolerance &&
        abs(c1.boundaries.height - c2.boundaries.height) < tolerance);
});

conesFound.erase(toRemove, conesFound.end());

```

5 Considerazioni Computer Vision tasks

Dopo aver implementato l'algoritmo di stampa dei contorni dei coni rilevati, il risultato finale si presenta nel seguente modo:

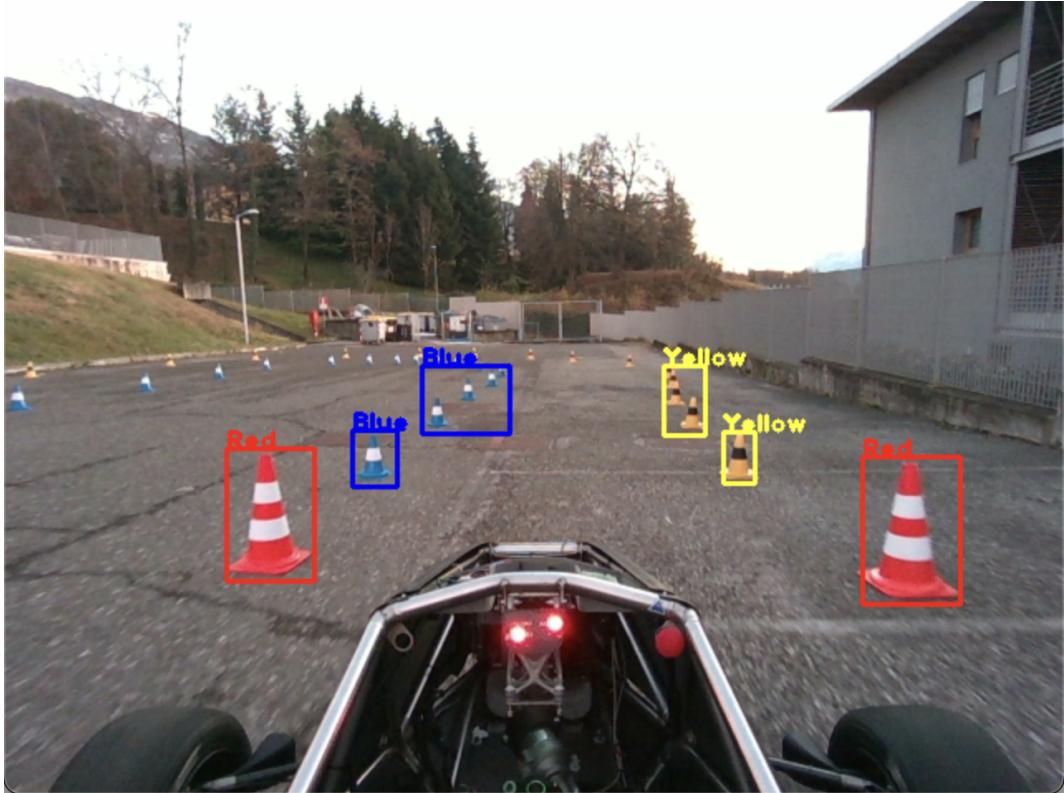


Figure 3: Risultato finale sezione di Computer Vision

Si nota chiaramente come alcuni coni non vengano riconosciuti mentre altri vengano raggruppati e considerati un'unica entità.

Questi due problemi sono accomunati dalla **stessa causa scatenante**: un approccio troppo aggressivo nella fase di processamento dell'immagine (all'interno di `processColorMask` per intenderci).

Nel **primo caso**, la maschera risultante dall'applicazione di un dato range di colore (lo si vede chiaramente nel caso del Blu), contiene troppe poche informazioni circa i coni più distanti (pochi pixel isolati) e quindi, in fase di dilatazione ed erosione, questi ultimi, vengono trasformati in rumore, **ignorato** successivamente dall'algoritmo di identificazione delle forme geometriche.

Nel **secondo caso** il rumore dei bordi dei coni nell'immagine (dovuto anche al fatto che il colore nei pressi dei bordi rientra nei range specificati) si dilata eccessivamente nel processamento con `void cv::dilate(...)` andando a fondersi con altri coni se disposti vicini a livello di prospettiva.

Al fine di poter risolvere entrambi questi problemi è fondamentale riuscire a trovare un buon *tradeoff* tra soglie dei range di colore, parametri delle funzioni di processamento e parametri dell'algoritmo di approssimazione delle forme geometriche.

Una soluzione ragionevole potrebbe essere ottenuta tramite **algoritmi di machine learning** che, tramite un training set di immagini caratterizzate da diverse esposizioni di luce e differenti zone d'ombra, possano stimare i parametri da passare a `processColorMask` e a `findBoundaries`.

Di seguito possiamo notare i problemi descritti nel caso dei coni blu.

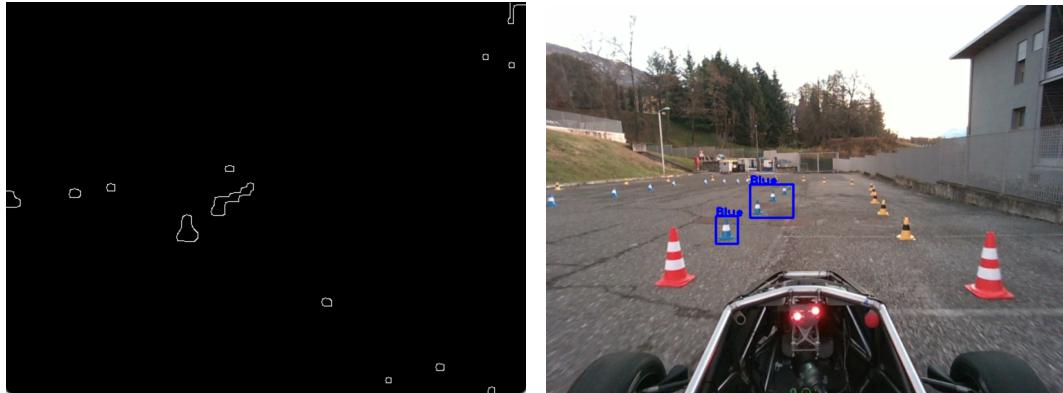


Figure 4: contorni post erosione/dilatazione

Figure 5: risultato finale

6 Task 4, Extract Track Edges

Per iniziare a proiettare i lati del tracciato è stata implementata una funzione che determina, per i coni rossi di partenza, quale fosse il cono di sinistra e quale quello di destra.

```
int assignRedConeSides(vector<Cone>& v, const Mat& input) {
    int imageWidth = 0;
    for (const auto& item : v) {
        if (item.center.x + item.boundaries.width > imageWidth) {
            imageWidth = item.center.x + item.boundaries.width;
        }
    }
    int centerX = imageWidth/2;
    for (auto& item : v) {
        if (item.classifiedAs == "Red") {
            item.isRight = (item.center.x >= centerX);
        }
    }
    return centerX;
}
```

`assignRedConeSides` calcola anche una stima del centro del tracciato relativa al frame d'immagine che si sta analizzando.

Successivamente tramite la funzione `drawTrackBoundaries` si vanno a tracciare le linee vere e proprie del tracciato, ovvero segmenti che partono da un cono all'altro, per ogni rispettivo lato.

All'interno di `drawTrackBoundaries` vengono creati due vettori popolati rispettivamente dai coni di destra e di sinistra. Tali vettori vengono poi sortati per stabilire l'ordine con cui tracciare i segmenti (partendo dal cono rosso fino ai coni successivi per ogni lato).

Si noti che l'algoritmo di sorting utilizza un criterio che viene stabilito all'interno di `drawTrackBoundaries` stessa. Tale criterio è il seguente:

```
vector<Cone> rightBoundaryCones;
vector<Cone> leftBoundaryCones;
...
auto sortByProximity = [](const Cone& a, const Cone& b) {
    return (a.center.y > b.center.y);
};
sort(rightBoundaryCones.begin(), rightBoundaryCones.end(), sortByProximity);
sort(leftBoundaryCones.begin(), leftBoundaryCones.end(), sortByProximity);
```

`sortByProximity` è piuttosto *debole* come criterio in quanto funziona in questo specifico caso di rettilineo ma **non tutela** nei casi di immagini di **curve molto strette**. Un'ottimizzazione da considerare potrebbe essere quella di sortare i vettori non solo per ordinata (a ordinata maggiore corrisponde maggiore vicinanza alla vettura) ma anche in combinazione con la distanza dall'asse centrale, dando pesi differenti a questi valori in base alle diverse situazioni che si presentano (anche in questo caso un algoritmo di *pattern recognition* semplificherebbe la vita di molto).

7 Task 5, Odometry

Nell'ultima task, relativa al calcolo della posizione della vettura a partire dalle due immagini passate, è stato utilizzato l'algoritmo ORB (*Oriented Fast and Rotated Brief*), uno specifico algoritmo di *features detection*.

Entrambi le parti di *features detection* e *matching* sono state implementate in un'unica grande funzione: `featuresMatcher`.

7.1 Features detection e descriptors

Per semplificare la gestione della memoria è stato fatto utilizzo di *smart pointers* per quanto riguarda l'istanziazione di oggetti della classe ORB. Inoltre, le matrici rappresentanti le immagini da analizzare sono state preventivamente convertite in scala di grigi tramite la funzione `void cv::cvtColor(...)`.

La ricerca dei *keypoints* presenti nelle due immagini è stata fatta tramite il metodo membro della classe ORB, `detectAndCompute`, facendo attenzione a non voler passare nessuna maschera come parametro ma analizzare l'immagine per intera (realizzato specificando `noArray()` come parametro).

```
void featuresMatcher(const Mat& first, const Mat& second, Mat& output) {
    Mat grayFirst, graySecond, descriptorFirst, descriptorSecond;
    cvtColor(first, grayFirst, COLOR_BGR2GRAY);
    cvtColor(second, graySecond, COLOR_BGR2GRAY);

    Ptr<ORB> orb = ORB::create(10);
    vector<KeyPoint> keyPointsFirst, keyPointsSecond;

    orb->detectAndCompute(grayFirst, noArray(), keyPointsFirst,
                           descriptorFirst);
    orb->detectAndCompute(graySecond, noArray(), keyPointsSecond,
                           descriptorSecond);
    ...
}
```

`detectAndCompute` si occupa anche di calcolare i *descrittori* dei keypoints identificati in modo tale che possa essere eseguita la fase di *matching*.

7.2 Matching

Per implementare la fase di *matching* è stato creato un *matcher Brute Force*, oggetto che confronta ogni descrittore del primo set di keypoints con ogni descrittore del secondo set di keypoints.

Successivamente, avvalendosi dell'algoritmo *K-Nearest Neighbours*, o KNN, si sono certati per ogni descrittore in `descriptors1` i 2 più simili descrittori in `descriptors2`. I risultati vengono poi salvati nel vettore `knn`.

```
void featuresMatcher(const Mat& first, const Mat& second, Mat& output) {
    ...
    BFMatcher match(NORM_HAMMING);
    vector<vector<DMatch>> knn;
    match.knnMatch(descriptorFirst, descriptorSecond, knn, 2);
    ...
}
```

Per aumentare la precisione e ridurre l'ambiguità dei match è stato applicato il **Lowe's ratio test** ossia un controllo tra i due match più simili al primo per scovare quale dei due ha distanza minore. Molto brevemente, se il match migliore è molto più vicino del secondo più simile allora è probabile che sia un vero match. Il tutto è riassunto nell'espressione condizionata: `if(knn[i][0].distance < 0.75f * knn[i][1].distance){...}`.

```
void featuresMatcher(const Mat& first, const Mat& second, Mat& output) {
    ...
    vector<DMatch> matchesFound;
    for (int i = 0; i < knn.size(); i++) {
        if (knn[i].size() > 1) {
            if (knn[i][0].distance < 0.75f * knn[i][1].distance) {
                matchesFound.push_back(knn[i][0]);
            }
        }
    }
    ...
}
```

Sono state poi estratte le coordinate 2D dei match e salvate nei vettori `points1` e `points2` rispettivamente.

7.3 Calcolo della matrice essenziale

Per concludere la task di odometria è stata calcolata la **matrice essenziale**, matrice fondamentale se si vuole stimare il movimento tra le due immagini, che richiede almeno 5 punti tra quelli salvati nei vettori `points1` e `points2`. Il calcolo vero e proprio è stato affidato alla funzione `cv::Mat cv::findEssentialMat(...)`.

```
void featuresMatcher(const Mat& first, const Mat& second, Mat& output) {
    ...
    vector<Point2f> pointsFirst, pointsSecond;
    for (int i = 0; i < matchesFound.size(); i++) {
        pointsFirst.push_back(keyPointsFirst[matchesFound[i].queryIdx].pt);
        pointsSecond.push_back(keyPointsSecond[matchesFound[i].trainIdx].pt);
```

```

        if(pointsFirst.size() >= 5) {
            essentialMat = findEssentialMat(pointsFirst, pointsSecond,
                intrinsicMatrix, RANSAC, 0.999, 1.0, mask);
            recoverPose(essentialMat, pointsFirst, pointsSecond,
                intrinsicMatrix, rotationMat, translationMat, mask);
        }
    }
    drawMatches(first, keyPointsFirst, second, keyPointsSecond,
        matchesFound, output);
}

```

Notiamo che uno dei parametri di `cv::Mat cv::findEssentialMat(...)` è proprio la matrice dei parametri interni della camera, fornita come dato a priori e definita **globablmente** come:

```

Mat intrinsicMatrix = (Mat_<double>(3, 3) <<
    387.3502807617188, 0, 317.7719116210938,
    0, 387.3502807617188, 242.4875946044922,
    0, 0, 1
);

```

Per concludere, prima della stampa dei punti trovati nella fase di matching, tramite `int cv::recoverPose(...)`, sono stati ricavati **rotazione** e **traslazione** tra le due pose della camera.

Nelle immagini a seguire si mostra il risultato ottenuto:

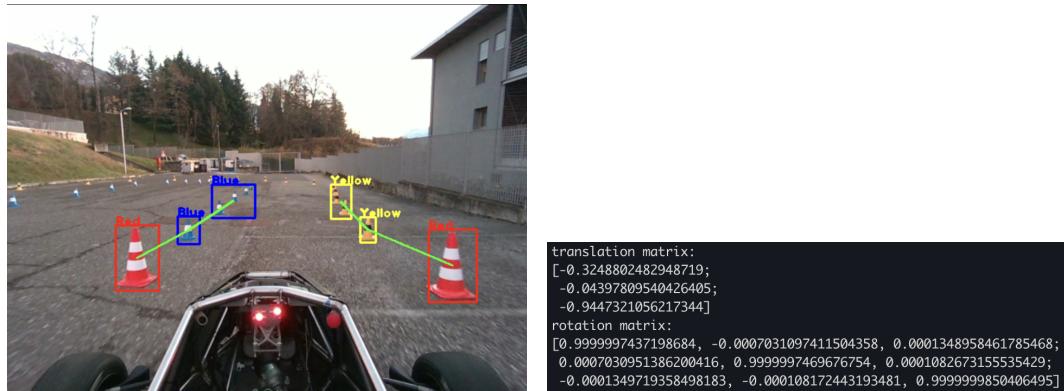


Figure 6: *track detection*

Figure 7: *position matrix*