Universidad de Salamanca | Grado en Ingeniería Informática

Visión Artificial

Algoritmo de visión con OpenCV y simulación

López Sánchez, Javier
Mateos Pedraza, Alejandro

VNiVERSiDAD D SALAMANCA

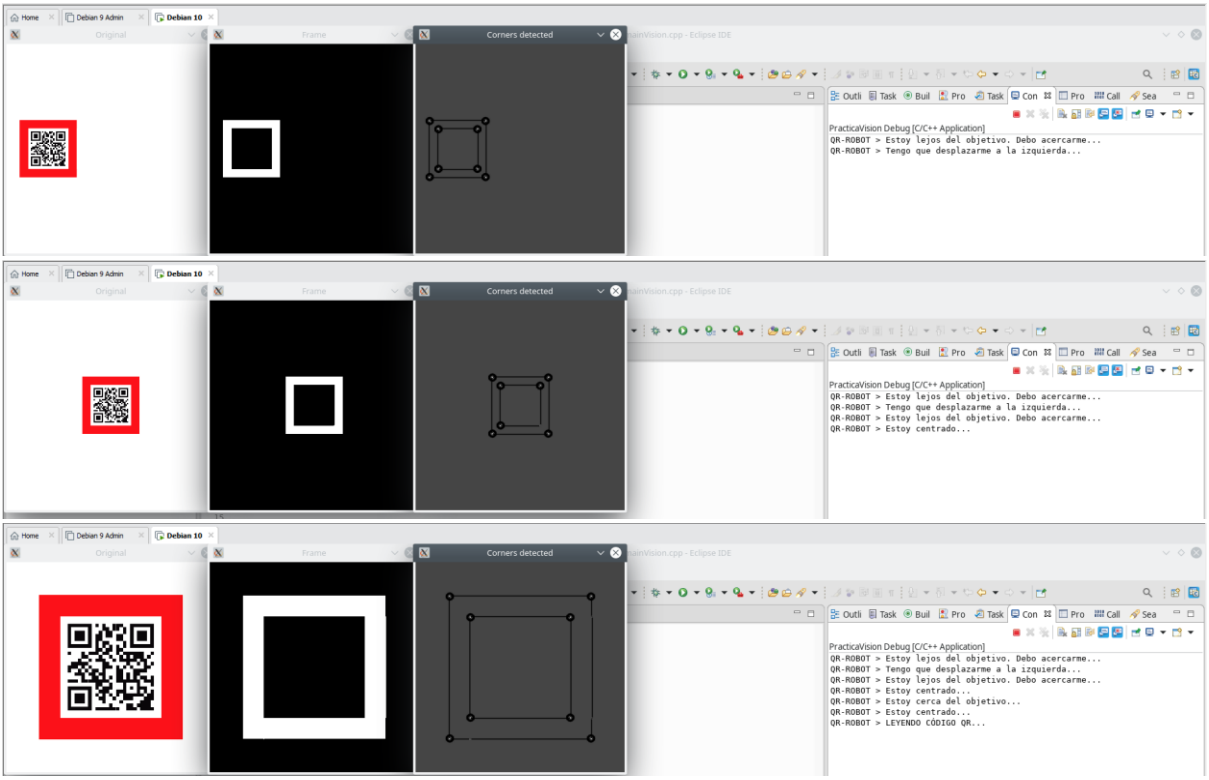
CAMPUS DE EXCELENCIA INTERNACIONAL

Abstract: The purpose of this work is to introduce the reader to the world of artificial vision in mobile robotics, and to make a first approach to the programming of its techniques with OpenCV.

Keywords: Artificial Vision, OpenCV, Harris Corners Algorithm

1. Explicación del algoritmo

1.1. Nota preliminar: planteamiento del problema



Para abordar la solución, se ha estructurado el problema en una serie de etapas secuenciales que se repiten mientras que la cámara del robot -hablando en abstracto- se sitúe lo suficientemente cerca del objetivo y centrada. Para simular todo este proceso, el primer paso ha sido crear una sencilla batería de imágenes que contemplan los casos solicitados, tanto de imagen cercana como lejana, y así también tanto centrada como no.

```
//Fijamos la posición inicial de robot de forma aleatoria
srand(time(NULL));
int photoIndex = rand()%5;

switch(photoIndex){
    case 0:
        if (cargaImagen("RobotQR_Izquierda.png") == -1){
            return -1;
        }
    break;

    case 1:
        if (cargaImagen("RobotQR_Derecha.png") == -1){
            return -1;
        }
    break;

    case 2:
        if (cargaImagen("RobotQR_Lejos.png") == -1){
            return -1;
        }
    break;

    case 3:
        if (cargaImagen("RobotQR_IzquierdaLejos.png") == -1){
            return -1;
        }
    break;

    case 4:
        if (cargaImagen("RobotQR_DerechaLejos.png") == -1){
            return -1;
        }
    break;
}
```

Antes de iniciar todo el procedimiento, lo primero que hacemos es cargar una imagen completamente aleatoria de todo nuestro set - excepto la imagen centrada final, que carece de sentido mostrarla como imagen de partida. Esto se consigue con el bloque de código anexo.

Una vez cargada la imagen inicial, tenemos los recursos necesarios para comenzar con el procesamiento de la imagen para que proporcione la información deseada. La secuencia de pasos a ejecutar será, por este orden de etapas:

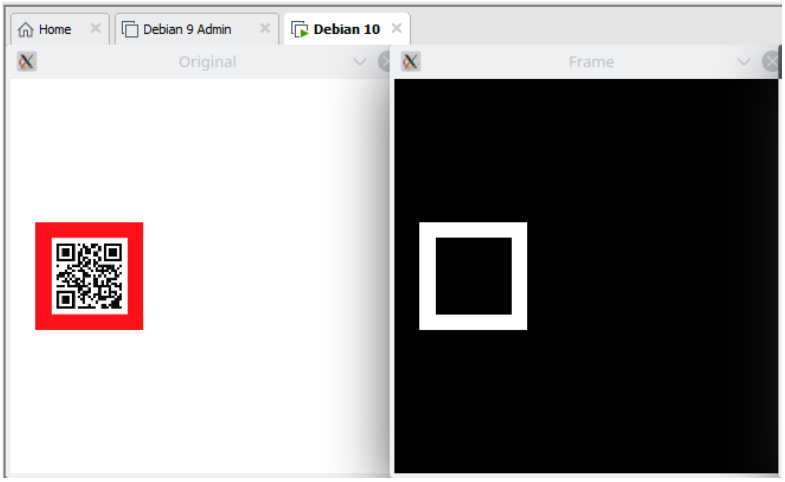
- Etapa 1: filtrado por color del marco rojo
- Etapa 2: cálculo de esquinas: método de Harris
- Etapa 3: obtención de las esquinas críticas
- Etapa 4: algoritmo de decisión

En función de la decisión final tomada por el robot, simulamos su posible movimiento mostrando la imagen que correspondería a la nueva situación de la cámara, y el bucle se repite hasta obtener la imagen final centrada. El proceso se encapsula de la siguiente manera:

```
    //Bucle mientras no estemos centrados y cerca del objetivo
    while(flag != 22){

        //ETAPA 1 | Filtrado: nos quedamos con el marco rojo
        cvtColor(src, src_color, COLOR_BGR2HSV);
        inRange(src_color, Scalar(170,100,100), Scalar(179,255,255), marco);
        imshow("Frame", marco);

        //ETAPA 2 | Algoritmo de esquinas de Harris
        cornerHarris(0, 0);

        //ETAPA 3 | Calculamos las esquinas críticas
        esquinasCriticas(cornerMenorX, cornerMayorX, cornerMenorY);

        //ETAPA 4 | El robot decide su siguiente movimiento
        flag = decide(cornerMenorX, width-cornerMayorX, cornerMenorY);

        waitKey();

        //Mostramos la siguiente imagen en función de la decisión
        switch(flag){
            case 10: //Lejos y debo desplazarme a derecha para centrarme
            case 11: //Lejos y debo desplazarme a izquierda para centrarme
                if (cargaImagen("RobotQR_Lejos.png") == -1){
                    return -1;
                }
            break;

            case 12: //Lejos y centrado
                if (cargaImagen("RobotQR.png") == -1){
                    return -1;
                }
            break;

            case 20: //Cerca y debo desplazarme a derecha para centrarme
            case 21: //Cerca y debo desplazarme a izquierda para centrarme
                if (cargaImagen("RobotQR.png") == -1){
                    return -1;
                }
            break;
        }
    }

    return 0;
}
```

1.2. Proceso de filtrado por color del marco rojo

```
    //ETAPA 1 | Filtrado: nos quedamos con el marco rojo
    cvtColor(src, src_color, COLOR_BGR2HSV);
    inRange(src_color, Scalar(170,100,100), Scalar(179,255,255), marco);
    imshow("Frame", marco);
```



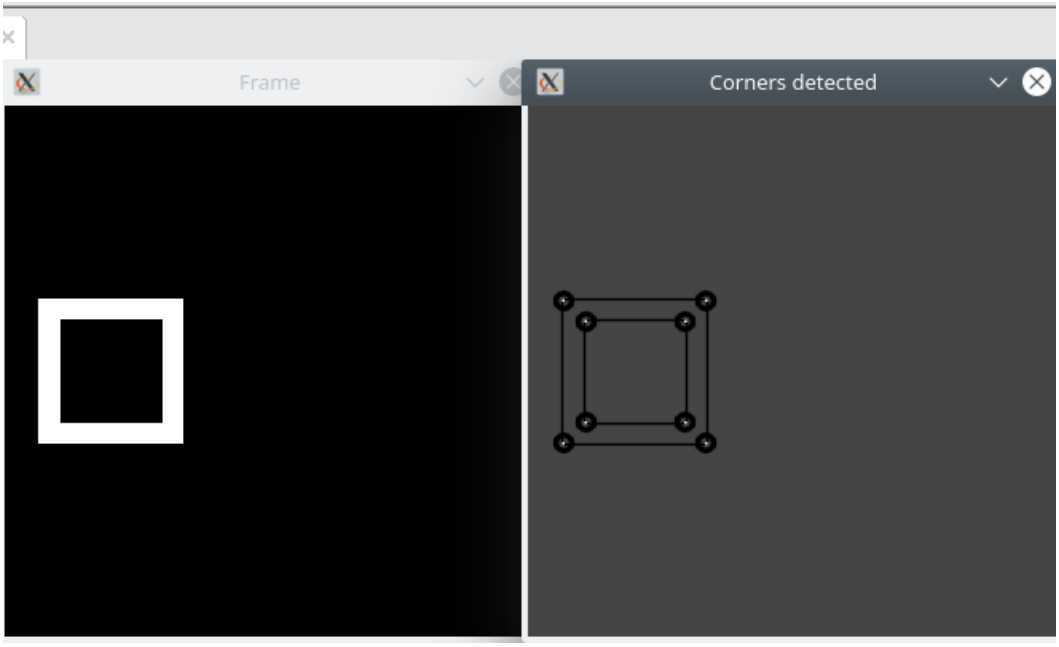
El objetivo de esta primera fase de procesamiento es pasar de la imagen original a una imagen con un marco perfectamente definido y único. Esto lo conseguimos de manera muy sencilla con un proceso de filtrado (Thresholding). Lo primero es definir el modo de color: se ha decidido utilizar HSV (Hue, Saturation, Value) en lugar de RGB. Este modo de color será utilizado por la función inRange para filtrar los colores que se encuentren en el rango (170,100,100) y (179,255,255) -que es rango de los rojos- y enviar la imagen de salida al objeto "marco", tal y como hemos definido. Todo lo que no sea rojo desaparecerá de la imagen tras el filtro, mientras que el marco quedará perfectamente visible y diferenciable en color blanco, como se puede apreciar.

1.3. Detección de esquinas: algoritmo de Harris

```
//Fuente: https://docs.opencv.org/3.4/d4/d7d/tutorial_harris_detector.html
void cornerHarris(int, void*){
    numCorners=0;
    int blockSize = 2;
    int apertureSize = 3;
    double k = 0.04;
    Mat dst = Mat::zeros( src.size(), CV_32FC1 );

    cornerHarris(marco, dst, blockSize, apertureSize, k);
    Mat dst_norm, dst_norm_scaled;

    normalize( dst, dst_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat() );
    convertScaleAbs( dst_norm, dst_norm_scaled );

    for(int i = 0; i < dst_norm.rows ; i++){
        for(int j = 0; j < dst_norm.cols; j++){
            if((int) dst_norm.at<float>(i,j) > thresh){
                circle(dst_norm_scaled, Point(j,i), 5,  Scalar(0,0,255), 2, 9, 0);

                //Almacenamos las esquinas
                cornersX[numCorners]=j; //j: valores en el eje x de las esquinas
                cornersY[numCorners]=i; //i: valores en el eje y de las esquinas
                numCorners++;
            }
        }
    }

    namedWindow(corners_window);
    imshow(corners_window, dst_norm_scaled);
}
```

Una vez hemos definido el marco, lo tenemos más sencillo para detectar las esquinas del mismo. Para hacerlo, recurrimos a la documentación de OpenCV, donde dimos con el algoritmo de Harris, que aquí adjuntamos. Véase como en la parte inferior aprovechamos a medida que el algoritmo va detectando las esquinas para guardar su posición espacial (x,y), que será útil posteriormente para conocer la distancia del marco a los bordes de la imagen procesada.

La ejecución del algoritmo de Harris nos ofrece un resultado como el siguiente, donde observamos claramente la detección de esquinas: cuatro exteriores y otras tantas interiores.



1.4. Cálculo de distancias a partir de las esquinas críticas

```
void esquinasCriticas(int &cornerMenorX, int &cornerMayorX, int &cornerMenorY){

    //Calculamos el mayor y menor valor en x de todas las esquinas
    //Estos valores nos servirán para saber si el objetivo está centrado
    cornerMenorX = cornersX[0];
    cornerMayorX = cornersX[0];

    for (int i=1; i<numCorners; i++){
        if (cornersX[i] > cornerMayorX) cornerMayorX = cornersX[i];
        if (cornersX[i] < cornerMenorX) cornerMenorX = cornersX[i];
    }

    cornerMenorY = cornersY[0];

    //Calculamos el menor valor en y de todas las esquinas
    //Este valor nos servirá para saber si estamos cerca del objetivo
    for (int i=1; i<numCorners; i++){
            if (cornersY[i] < cornerMenorY) cornerMenorY = cornersY[i];
        }
}
```

El algoritmo de esquinas de Harris nos devuelve un total de ocho esquinas nuestro marco: las cuatro exteriores y otras cuatro interiores. Para conocer la distancia del marco a los bordes, debemos obtener las que hemos llamado “esquinas críticas”, esto es, las esquinas exteriores. Éstas son las esquinas que utilizaremos para medir la distancia del marco a los bordes de la imagen. Con el algoritmo anterior, basado en el cálculo de los máximos y mínimos de las posiciones ‘x’ e ‘y’ de las respectivas esquinas que habíamos ido guardando en la etapa anterior, obtenemos lo siguiente:

- Valor en x de la esquina del marco situada más a la izquierda: habrá, obviamente, dos esquinas “más a la izquierda”, por ser un marco perfectamente cuadrado; recogemos el valor que comparten, que es lo único que necesitaremos.
- Valor en x de la esquina del marco situada más a la derecha: mismo caso que el anterior, pero en este caso recogemos el valor máximo en este eje.
- Valor en y de la esquina más próxima al borde superior de la imagen.

Como se comenta en el propio código, si comparamos estos valores con las dimensiones de la imagen, el procesamiento en el eje x nos permitirá saber si el marco está centrado; en cuanto al valor obtenido en el eje y, podremos utilizarlo, como se verá, para saber si el marco está lejos o cerca de la supuesta cámara.

1.5. Método de decisión

```
int decide(int distanciaIzquierda, int distanciaDerecha, int distanciaVertical){
    if (distanciaVertical > factorCercania){
            cout << "QR-ROBOT > Estoy lejos del objetivo. Debo acercarme..." << endl;

            if (distanciaIzquierda > distanciaDerecha){
                cout << "QR-ROBOT > Tengo que desplazarme a la derecha..." << endl;
                return 10;
            }

            else if (distanciaIzquierda < distanciaDerecha){
                cout << "QR-ROBOT > Tengo que desplazarme a la izquierda..." << endl;
                return 11;
            }

            else{
                cout << "QR-ROBOT > Estoy centrado..." << endl;
                return 12;
            }
    }

    else{
        cout << "QR-ROBOT > Estoy cerca del objetivo..." << endl;

        if (distanciaIzquierda > distanciaDerecha){
            cout << "QR-ROBOT > Tengo que desplazarme a la derecha..." << endl;
            return 20;
        }

        else if (distanciaIzquierda < distanciaDerecha){
            cout << "QR-ROBOT > Tengo que desplazarme a la izquierda..." << endl;
            return 21;
        }

        else{
            cout << "QR-ROBOT > Estoy centrado..." << endl;
            cout << "QR-ROBOT > LEYENDO CÓDIGO QR..." << endl;
            return 22;
        }
    }
}
```

Ahora que tenemos el marco ubicado espacialmente, el último paso para el robot es decidir. En primer lugar, comparamos el valor obtenido en el eje 'y' con un factor de cercanía predefinido que nos indica la frontera sobre si el objetivo está lejos o cerca. Si 'y' es mayor que este factor, significa que la distancia del marco con respecto al borde superior de la foto es más amplia, y por lo tanto estamos lejos, por lo que ordenamos al robot acercarse, que en este caso de simulación significará mostrar la misma imagen que teníamos en esta etapa, pero ampliada.

Una vez resuelto el problema de la cercanía (la distancia del marco al borde superior de la foto será menor que el factor de cercanía), queda centrarse, si el robot todavía no lo está. Para ello, se comparan los valores obtenidos en x de las esquinas más próximas a los extremos laterales de la imagen. La x menor ya nos devuelve la distancia de nuestro marco con respecto al borde izquierdo de la foto. Para conocer la distancia del marco con respecto al derecho, simplemente el parámetro de distancia que pasaremos será igual al ancho de la ventana menos ese valor x máximo calculado (véase la llamada al método que se incluye en el apartado 1.1 como nota preliminar). La comparación de estas distancias nos ofrece las siguientes posibilidades de decisión:

- La distancia a la izquierda es mayor que a la derecha: eso quiere decir que el objetivo está desplazado a la derecha, y por tanto que el robot debe moverse en esa dirección.
- La distancia a la derecha es mayor que a la izquierda: el objetivo está desplazado a la izquierda, por lo que el robot debe moverse a la izquierda.
- Si no se verifica ninguno de los casos anteriores, el robot está finalmente centrado y hemos alcanzado nuestro *'goal'*.