



INTELIGENCIA ARTIFICIAL

Trabajo Práctico No. 4

Transformación de Hough

Alumno: Alcover, N. Ariel.

Legajo: VINF012465

Titular Experto: Pablo Alejandro Virgolini.

Titular Disciplinar: Maria Paula Gonzalez.

Tutoría: CUATRIMESTRALES 2/24

Contenido

1. Revisar la transformación de Hough para rectas y circunferencias. Resumir su formulación y características, brindando ejemplos.	3
2. Prototipo para detección de rectas	6
3. Prototipo para detección de círculos.....	8
4. Balance de los tres enfoques tratados.....	9
Bibliografía	11

1. Revisar la transformación de Hough para rectas y circunferencias. Resumir su formulación y características, brindando ejemplos.

La transformada de Hough es una técnica utilizada en procesamiento de imágenes y visión por computadora que se utiliza para detectar formas geométricas en imágenes, siendo las más comunes las rectas y las circunferencias.

Se define como una técnica de votación, donde en lugar de probar todas las posibles líneas, se realiza el proceso inverso. Se deja que cada pixel (o característica) vote por ese pixel. Lo que define al algoritmo como un modelo de votación. Finalmente buscamos la línea con mayor número de votos.

→ Transformada de Hough para Rectas

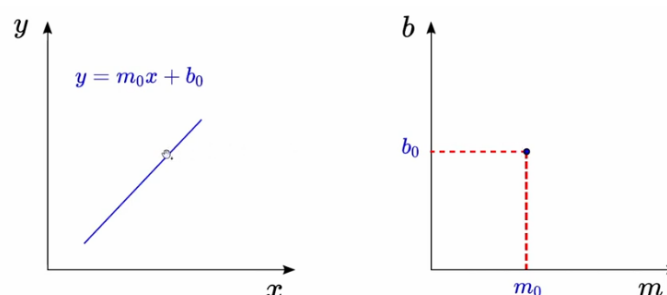
El modelo paramétrico inicia con una recta. La representación estándar de una recta en el plano cartesiano es:

$$y = mx + b \quad y = f(x)$$

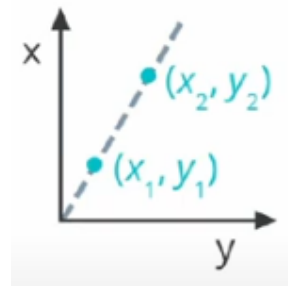
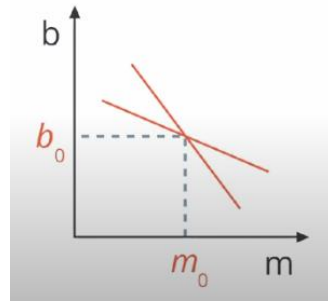
Cuando trabajamos con la transformada de Hough, hacemos una conversión

$$b = y - mx \quad b = g(m)$$

Ahora, nuestros parámetros serían “x” e “y”. En este modelo, partiendo de una recta, llegamos a un punto. Todos los puntos de una misma recta definida en el espacio de la imagen, llegan a un mismo punto en el espacio de Hough.

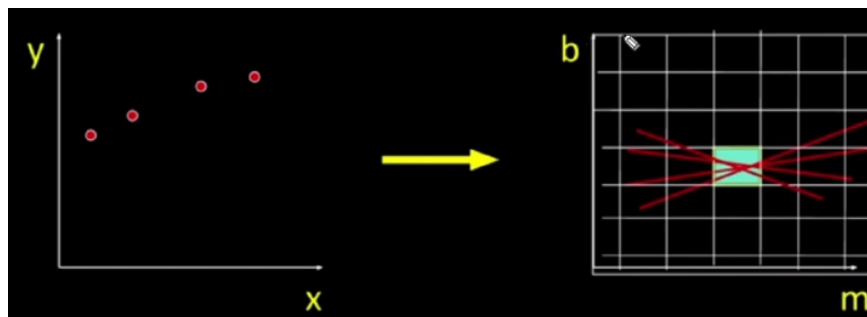


De esta manera, podemos decir que un punto en el espacio de Hough corresponde a una recta en el espacio de la imagen. Por lo tanto, queda definido el punto de intersección de dos (o más) rectas en el espacio de Hough, como la recta que une estos puntos en el espacio de la imagen.



$$y = m_0x + b_0$$

Pasando al algoritmo de Hough, vamos a tener como entradas una serie de puntos, en un espacio discreto, donde cada recta va a votar por los pixeles que ocupa, de esta manera, el espacio con más votaciones va a ser la recta que mejor se ajusta a los puntos de entrada.



Cómo vemos, los puntos no están alineados, pero esto no impide encontrar la recta que mejor ajusta. Sin embargo, esta forma puede ser problemáticamente indefinida para rectas verticales. Por este motivo, se utiliza una representación alternativa basada en coordenadas polares, con forma paramétrica:

$$r = x \cos(\theta) + y \sin(\theta)$$

donde:

- r : es la distancia desde el origen a la línea más corta perpendicular a la línea.
- θ : es el ángulo de inclinación de la línea con respecto al eje x .

En la implementación polar se utiliza el mismo procedimiento, aunque en este caso, vamos a ver curvas sinusoidales en el espacio de Hough.

→ Transformada de Hough para Circunferencias

El procedimiento y algoritmo es similar, aunque tiene más dimensiones. Para detectar circunferencias, la ecuación de una circunferencia en el plano se puede expresar como:

$$(x - a)^2 + (y - b)^2 = r^2$$

donde:

- (a, b) : son las coordenadas del centro de la circunferencia.

- r : es el radio.

En el contexto de la transformada de Hough, esta ecuación se convierte en un espacio de parámetros tridimensional

$$(a, b, r)$$

Conclusión

La transformada de Hough es una herramienta poderosa y versátil para la detección de formas geométricas, como rectas y circunferencias. Su robustez frente al ruido y la oclusión la hace adecuada para aplicaciones en diversas áreas, como la robótica, la visión artificial y el procesamiento de imágenes. Sin embargo, su complejidad computacional y el manejo del espacio de parámetros son desafíos que deben considerarse en su implementación.

Dicho esto, si bien en los trabajos anteriores navegué por las matemáticas de los modelos, la complejidad hace que crea inabarcable desarrollar todo el código de cero para el trabajo práctico. Por este motivo, investigando sobre el caso encontré que existen librerías muy poderosas que explotan adecuadamente las posibilidades del modelo, no solo para en la aplicación del trabajo práctico, sino además, en usos más sofisticados hasta detección de rostros en tiempo real mediante cámara.

Por lo que pude averiguar, Python es el lenguaje más utilizado y que mejor se adapta a estos usos, aunque no es un lenguaje que yo domine, por lo que opté por continuar con Java ya que la librería **OpenCV** se encuentra disponible para este lenguaje.

2. Prototipo para detección de rectas

La librería es muy robusta por lo que la implementación no lleva más de algunos renglones. Considero las centrales las siguientes:

→ bloque de código para cargar imagen y prepararla para:

```
// Cargar la imagen
String imagePath = "C:/Users/ariel/OneDrive/Documents/NetBeansProjects/"
    + "JavaApplication1/src/imagen/rectas_Mesa.jpg";
Mat src = Imgcodecs.imread(imagePath);
Mat gray = new Mat();
Mat blurred = new Mat();
Mat edges = new Mat();

// Convertir a escala de grises
Imgproc.cvtColor(src, gray, Imgproc.COLOR_BGR2GRAY);

// Aplicar desenfoque
Imgproc.GaussianBlur(gray, blurred, new Size(5, 5), 1, 1);

// Detectar bordes usando Canny
Imgproc.Canny(blurred, edges, 50, 150);
```

→ Bloque de código para rectas

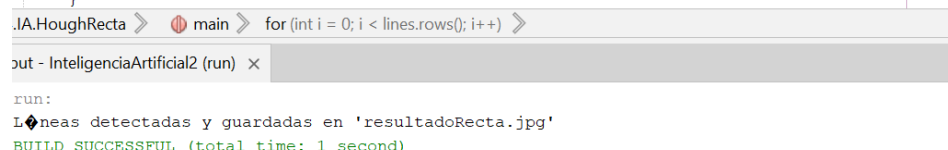
```
// Detectar líneas usando la Transformada de Hough
Mat lines = new Mat();
Imgproc.HoughLinesP(edges, lines, 1, Math.PI / 180, 50, 50, 1);

// Dibujar las líneas detectadas
for (int i = 0; i < lines.rows(); i++) {
    double[] line = lines.get(i, 0);
    int x1 = (int) line[0];
    int y1 = (int) line[1];
    int x2 = (int) line[2];
    int y2 = (int) line[3];

    // Dibujar la línea en la imagen original
    Imgproc.line(src, new Point(x1, y1), new Point(x2, y2), new Scalar
        (0, 255, 0), 3);
}
```

→ Bloque de código para imprimirlas en la imagen y salida confirmando la acción

```
// Guardar la imagen resultante
Imgcodecs.imwrite("resultadoRecta.jpg", src);
System.out.println("Líneas detectadas y guardadas en 'resultadoRecta.jpg'");
}
```



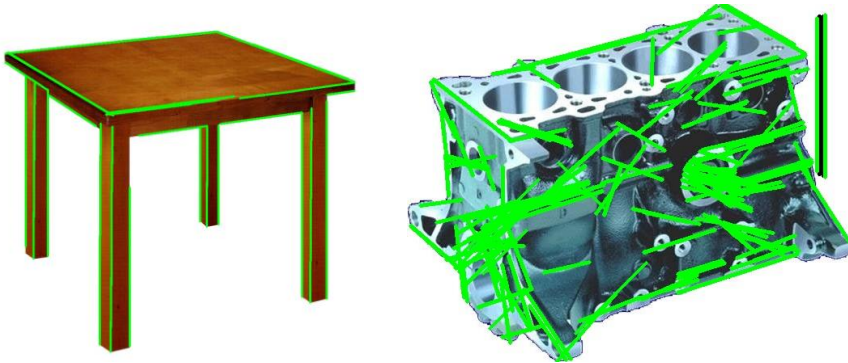
```
.IA.HoughRecta > main > for (int i = 0; i < lines.rows(); i++) >
out - InteligenciaArtificial2 (run) x
run:
Líneas detectadas y guardadas en 'resultadoRecta.jpg'
BUILD SUCCESSFUL (total time: 1 second)
```

→ Conclusión

Los resultados observados sobre los bloques de motor son muy inadecuadas, desde no observar líneas hasta llenar la imagen de las mismas tapando la misma, y por su puesto,

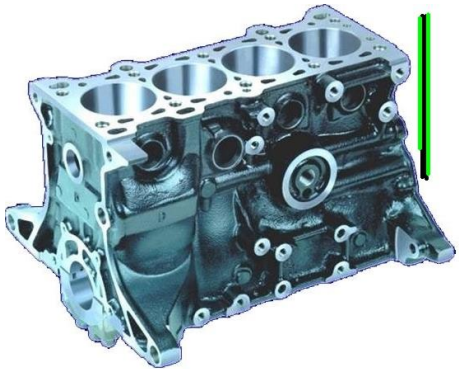
pasando por detectar como rectas cosas que no lo son.. Dejo además, un ejemplo de como se podría ajustar sobre una mesa, imagen que considero más adecuada.

```
Imgproc.HoughLinesP(edges, lines, 1, Math.PI / 180, 50, 50, 10);
```



De esta comparación, pensando en la resolución del caso (o conclusión del TP04) planteo una baliza, a modo de señalización para que el robot pueda detectar (o tener) una referencia de la posición y ubicación el motor.

```
Imgproc.HoughLinesP(edges, lines, 1, Math.PI / 180, 100, 100, 1);
```



3. Prototipo para detección de círculos

La librería es muy robusta por lo que la implementación no lleva más de algunos renglones. Además, no hay muchas diferencias con el caso de rectas.. Considero la siguiente:

→ en lugar de detectar rectas, detecta círculos, cuya radio puede definirse

```
// Definir el tamaño mínimo y máximo de los círculos
int minRadius = 19; // Radio mínimo con 19 solo detecta el deseado
int maxRadius = 30; // Radio máximo

// Detectar círculos
Mat circles = new Mat();
Imgproc.HoughCircles(blurred, circles, Imgproc.CV_HOUGH_GRADIENT, 1,
    blurred.rows()/8, 100, 30, minRadius, maxRadius);

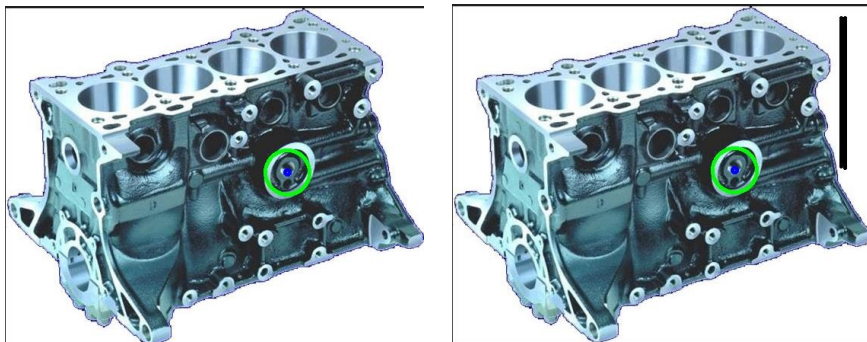
// Dibujar los círculos detectados
for (int i = 0; i < circles.cols(); i++) {
    double[] circle = circles.get(0, i);
    int x = (int) circle[0];
    int y = (int) circle[1];
    int radius = (int) circle[2];
```

Como se puede apreciar, la detección es casi perfecta, considerando incluso el ángulo de la imagen. Me pareció adecuado dibujar el centro (en azul), ya que es justamente el orificio deseado.

```
int minRadius = 19; // Radio mínimo con 19 solo detecta el deseado
```

```
int maxRadius = 30; // Radio máximo
```

```
Imgproc.HoughCircles(blurred, circles, Imgproc.CV_HOUGH_GRADIENT, 1, blurred.rows()/8,
    100, 30, minRadius, maxRadius);
```



La baliza no afecta la detección del círculo.

4. Balance de los tres enfoques tratados

Los modelos fueron tratados durante la materia, por lo que podemos resumir las siguientes características principales.

A. Enfoque Heurístico/Exhaustivo

Ambos métodos que utilizan reglas empíricas o algoritmos de búsqueda para encontrar soluciones. El enfoque exhaustivo evalúa todas las combinaciones posibles, mientras que el heurístico aplica reglas para reducir el espacio de búsqueda.

Dentro de las ventajas podemos encontrar que es fácil de entender e implementar, especialmente para problemas bien definidos y se puede adaptar a diversas situaciones y requerimientos específicos. Aunque contamos con algunas desventajas que podrían hacer que no pueda utilizarse en muchos problemas de la vida cotidiana ya sea por volverse impracticable, con un aumento en la complejidad del problema, resultando en un alto tiempo de cómputo o por la eficacia de los métodos heurísticos que dependen del conocimiento previo sobre el problema.

B. Redes de Hopfield

Redes neuronales recurrentes que funcionan como sistemas de memoria asociativa, útiles para resolver problemas de optimización y reconocimiento de patrones.

Sus principales ventajas son el procesamiento de información en paralelo, lo que puede resultar en tiempos de respuesta rápidos y su robustez, ya que tolera ruido en los datos y son capaces de encontrar patrones en información incompleta. Sus principales desventajas son la necesidad de un entrenamiento previo, lo que nuevamente, en la vida real puede ser impracticable o difícil de decidir cuanto sería adecuado o suficiente; por otro lado, puedo mencionar la posibilidad de encontrar “mínimos locales”, la incapacidad de encontrar soluciones óptimas o completas. Como yo mismo evidencié en el TP3, en el cual tuve que cambiar el patrón ya que el modelo no era capaz de reconocerlo.

C. Transformada de Hough

Un método para detectar formas geométricas en imágenes, utilizado para encontrar líneas y circunferencias.

Sus ventajas están a la vista, como dejo evidencia en el presente trabajo, es tolerante al ruido, se puede ajustar para encontrar las rectas y circunferencias deseadas, por lo que es muy efectiva para detectar formas en presencia de ruido y oclusiones; además, cuenta con una gran versatilidad ya que Puede adaptarse para detectar diferentes tipos de formas, lo que la hace útil en diversas aplicaciones.

Según pude averiguar, durante la etapa de indagación para desarrollar el trabajo, se puede aplicar a la detección de carriles en una autopista. Algo que ya se encuentra disponible en el mercado, dando cuenta de sus posibilidades en el mundo real.

En cuanto a sus inconvenientes, tenemos obviamente la complejidad computacional, puede ser intensiva en términos de tiempo y recursos, especialmente en imágenes de alta resolución. Relacionado a esto está la sensibilidad a la configuración, los resultados dependen de la elección de parámetros, como los umbrales y resoluciones de la propia imagen, lo que puede ser complicado de manejar.

Recomendación

Si bien la respuesta puede resultar obvia, tocamos la transformación de Hough en el módulo de robótica, voy a detenerme a analizar el problema desde diferentes ángulos, basado en mi opinión y conocimiento previo de la industria automotriz. A lo largo de la materia hemos ido viendo los usos más comunes de los modelos de IA desarrollados.

El problema planteado es el de un brazo robótico que debe colocar una pieza en el orificio indicado de un bloque de aluminio (material con el que se hacen los motores actualmente). De donde saca la pieza? Como llega el motor ahí? El brazo está fijo o su base debe moverse para un punto a otro?.. Y otras preguntas que se responden conociendo una línea de montaje de la industria.

El propio bloque lo hacen máquinas con una precisión de micrones (μm -micrómetros), los motores actuales soportan compresiones muy superiores a las de hace varios años. Por este motivo podemos confiar de la precisión de una línea de montaje de la industria, donde incluso la desviación puede ser conocida, tratada y corregida. Por ejemplo, cada dos días, la línea se retrasa un milímetro (1 mm).

Yo creo que el problema es perfectamente conocido, los estados solución son tres: punto de partida del brazo, ubicación de la pieza y ubicación del orificio, que luego podrían transformarse en dos, ya que el punto de partida podría ser el propio lugar de la pieza o el motor. El problema no cambia ya que hará siempre lo mismo, y las ubicaciones se pueden mantener a lo largo del tiempo, o corregir, si se detecta que desviación de la posición por fricción cada determinado tiempo.

Mi recomendación es utilizar un *enfoque heurístico*.

Será trabajoso inicialmente, requiere mucho trabajo manual para poder dar las coordenadas correctas al brazo, pero es trabajo amortizado en la vida útil de modelo (de motor) fabricado, no requerirá cambios hasta que cambie el motor. Cualquier otra solución también requiere trabajo de un programador capaz, y una complejidad asociada semejante; pero también, un número de piezas extra costosas sensibles al desgaste por uso. Su capacidad de procesamiento también sería mayor con cualquier otra solución, lo que significa mayor inversión inicial de HW y SW, no solo cámaras. Esto se puede resolver con un SO integrado en un HW básico.

Finalmente, el brazo robótico tiene preprogramados los movimientos y no necesita “pensar” que hacer con cada motor que se le presenta.

Bibliografía

<https://www.youtube.com/watch?v=zbyn57jgWNg>