

Tarea 2

Cálculo de puntos de interés Harris y reconocimiento de objetos particulares
usando SIFT y RANSAC

Integrantes: Benjamín A. Irarrázabal T.

Profesor: Javier Ruiz del Solar

Auxiliar: Patricio Loncomilla

Ayudantes: José Díaz V.

Danilo Moreira

Javier Mosnaim Z.

Jhon Pilataxi

Fecha de realización: 21 de septiembre de 2022

Fecha de entrega: 21 de septiembre de 2022

Santiago de Chile

Índice de Contenidos

1. Introducción	1
2. Algoritmos	2
2.1. Detección de Puntos de Interés de Harris	2
2.2. Descriptores SIFT	2
2.3. Algoritmo RANSAC	2
3. Resultados	3
3.1. Puntos de Interés de Harris	4
3.2. Descriptores SIFT y BFMatcher	7
3.3. Descriptores SIFT y RANSAC	8
3.4. Romboideos Dibujados	9
3.5. Resultados con Variación de Parámetros RANSAC	9
4. Análisis de Resultados	13
4.1. Puntos de Interés de Harris	13
4.2. Descriptores SIFT y BFMatcher	14
4.3. Descriptores SIFT y Algoritmo RANSAC	14
4.4. Romboideos en objetos	14
4.5. Variación de Parámetros RANSAC	14
5. Conclusiones	16
Referencias	17
6. Anexos	18
6.1. Imágenes de referencia y prueba	18
6.2. Código Implementado	19
6.2.1. Puntos de Interés de Harris	19
6.2.2. Descriptores SIFT y Algoritmo RANSAC	23

Índice de Figuras

1. Resultados para Tarro de Nescafé	4
2. Resultados para Estuche	5
3. Resultados para Billete	6
4. Resultados usando Descriptores SIFT y BFMatcher de OpenCV	7
5. Resultados usando Descriptores SIFT y algoritmo RANSAC	8
6. Romboideos dibujados sobre las imágenes de prueba	9
7. Resultados luego de variar los tres parámetros principales de RANSAC	10
8. Resultados luego de variar los tres parámetros principales de RANSAC	11
9. Resultados luego de variar los tres parámetros principales de RANSAC	12
10. Resultados luego de variar los tres parámetros principales de RANSAC	13
11. Imágenes de referencia	18

12. Imágenes de prueba	19
----------------------------------	----

1. Introducción

El procesamiento de imágenes se compone por un conjunto de técnicas aplicadas a imágenes digitales para poder modificar su calidad o buscar información dentro de esta, por ejemplo, colores, bordes, entre otros. Estos métodos han tenido un gran impacto en variadas áreas tales como medicina, telecomunicaciones, industria e incluso entretenimiento. Dentro de estos procedimientos, se encuentra la detección de puntos de interés, técnica que posee diferentes métodos tales como el detector de Moravec, el de Shi-Tomasi, el de Harris, entre otros. Estos métodos, permiten encontrar pixeles importantes dentro de una imagen, principalmente, de esquinas del contenido de esta. Junto a estos, existen formas de reconocer objetos particulares dentro de una imagen a partir de otra de referencia, lo anterior mediante el uso de descriptores locales, como puede ser SIFT, SURF, etc.

El presente informe, tiene como objetivo el desarrollo de la primera tarea del curso Procesamiento Avanzado de Imágenes y busca, usando herramientas computacionales, el cálculo de puntos de interés de Harris y reconocer objetos particulares dentro de una imagen usando SIFT y RANSAC para encontrar un grupo de calces compatibles con una misma transformación entre las imágenes de la sección 6.1. Para esto, se comenzará dando una breve introducción de los algoritmos utilizados, para posteriormente mostrar los resultados obtenidos, haciendo referencia a la sección de código implicada en estos (ver sección 6.2). Luego, se presentará el análisis correspondiente para finalizar con las principales conclusiones de la experiencia.

2. Algoritmos

A continuación se introducen los algoritmos utilizados, dando a conocer las secciones de código correspondientes.

2.1. Detección de Puntos de Interés de Harris

Para esto se comienza con las funciones **gradx** y **grady**, las cuales toman una imagen de entrada y calculan la aproximación del gradiente en el eje X e Y respectivamente. Esta aproximación se realiza mediante una convolución entre la imagen y las siguientes máscaras.

$$K_x = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} \quad K_y = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \quad (1)$$

Posterior a estas operaciones, se tiene la función **product**, la cual recibe dos imágenes (matrices) y retorna la multiplicación pixel a pixel entre estas. Luego, está la función **harris**, encargada de recibir los momentos m_{xx} , m_{xy} y m_{yy} , y calcular la respuesta del filtro en base a la siguiente ecuación:

$$\text{cornerness}_{\text{harris}} = \det - 0.04 \cdot \text{tr}^2 \quad (2)$$

donde

$$\det = m_{xx} \cdot m_{yy} - m_{xy}^2 \quad \wedge \quad \text{tr} = m_{xx} + m_{yy} \quad (3)$$

Después, la función **getMaxima** recibe una imagen de entrada (salida de la función **harris**) y un valor umbral y recorre la imagen para analizar dos condiciones, que el valor actual supere el umbral establecido y que el valor actual sea maximo local con respecto a sus 8 vecinos más cercanos. Si este valor cumple ambas condiciones, se retorna un 1 en su posición respectiva en la imagen de salida, en caso contrario, se deja en 0. Finalmente, esta salida ingresa a la función predefinida **getKeyPoints** y retorna los puntos de interés para las imágenes consultadas. (para ver las funciones comentadas ir a sección 6.2.1)

2.2. Descriptores SIFT

Para los descriptores locales SIFT, se usaron funciones predefinidas de la librería OpenCV de Python, por ejemplo, para crear un detector, se utilizó **SIFT_create**, luego, para encontrar puntos de interés y descriptores locales se utilizó **detectAndCompute**, el cual es un método del detector creado anteriormente. Finalmente, se utilizó **BFMatcher** para trabajar con los calces en la imagen y posteriormente **drawMatches** para dibujar estos (ver sección 6.2.2).

2.3. Algoritmo RANSAC

Random Sample Consensus (RANSAC) es un algoritmo para ajustar modelos en presencia de outliers. Esto, aplicado a imágenes permite realizar calces paramétricos utilizando una imagen de referencia y una de prueba. En esta experiencia, este algoritmo se divide en las siguientes funciones.

- **genTransform**: esta recibe un calce y los puntos de interés de la imagen de referencia y de prueba (*keypoints*), luego, calcula la transformación de semejanza en base a las siguientes

igualdades, donde e corresponde al factor de escala entre las imágenes, θ al ángulo de rotación y (t_x, t_y) la traslación.

$$\begin{aligned} e &= \frac{\sigma_{PRU}}{\sigma_{REF}} \\ \theta &= \phi_{PRU} - \phi_{REF} \\ tx &= x_{PRU} - e(x_{REF}\cos(\theta) - y_{REF}\sin(\theta)) \\ ty &= y_{PRU} - e(x_{REF}\sin(\theta) + y_{REF}\cos(\theta)) \end{aligned}$$

- **computeConsensus:** genera una lista de calces aceptados en base a una distancia definida mediante un umbral entregado. En este caso, la distancia ocupada es la euclíadiana y en un principio se utiliza un umbral de consenso igual a 30. Junto a esto, la función entrega un número de calces aceptados, por lo tanto, siempre y cuando este número sea mayor al umbral de consenso se entrega la lista de calces.
- **ransac:** une las dos funciones anteriores y repite el consenso según la cantidad de intentos predefinida (en un inicio son 100), finalmente, entrega los calces aceptados según la función computeConsensus explicada anteriormente.
- **calcAfin:** busca realizar una transformación afín definida como sigue:

$$\begin{pmatrix} x_{REF} & y_{REF} & 0 & 0 & 1 & 0 \\ 0 & 0 & x_{REF} & y_{REF} & 0 & 1 \end{pmatrix} \begin{pmatrix} m_{XX} & m_{XY} & m_{YX} & m_{YY} & t_X & t_Y \end{pmatrix}^T = \begin{pmatrix} x_{PRU} \\ y_{PRU} \end{pmatrix} \quad (4)$$

donde la primera matriz se denota A , la segunda x y el resultado es un vector b ($Ax = b$). Con lo anterior, la solución de la transformación afín se puede encontrar como:

$$x^* = (A^T A)^{-1} A^T b \quad (5)$$

Cabe destacar, al observar la sección 6.2.2, que al finalizar la función, se realiza un reordenamiento del vector transformación, esto debido a que las operaciones entregan una matriz de tamaño $1x6$ y para poder utilizar la función drawProjAfin predefinida en el notebook se requiere de una matriz de transformación de $2x3$.

- Finalmente, las funciones, **filterMatches** y **drawProjAfin** vienen predefinidas en el notebook, pero realizan un filtrado de los calces (dependiendo de una distancia similar al caso de ransac) y el dibujo del romboide en cada imagen de prueba respectivamente.

3. Resultados

3.1. Puntos de Interés de Harris



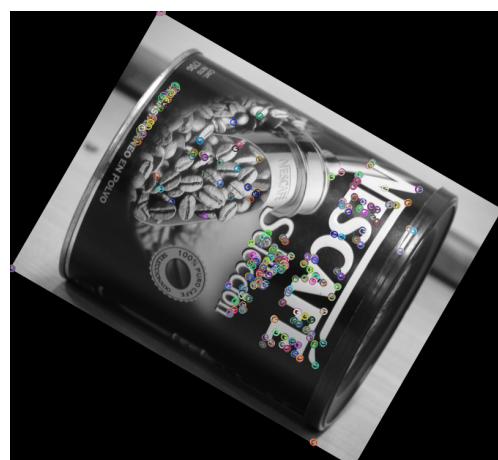
(a) Filtro a imagen original



(b) Filtro a imagen rotada



(c) Puntos de interés de imagen original



(d) Puntos de interés de imagen rotada

Figura 1: Resultados para Tarro de Nescafé

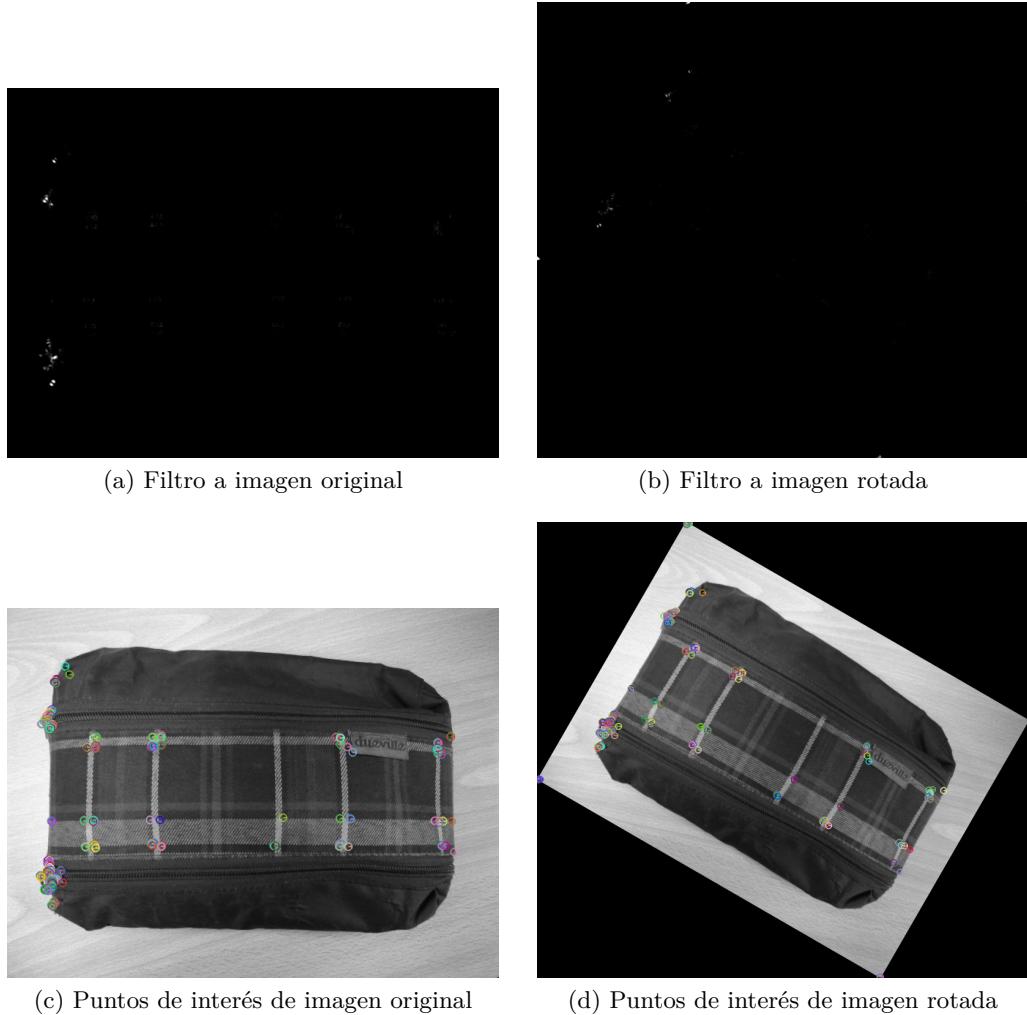


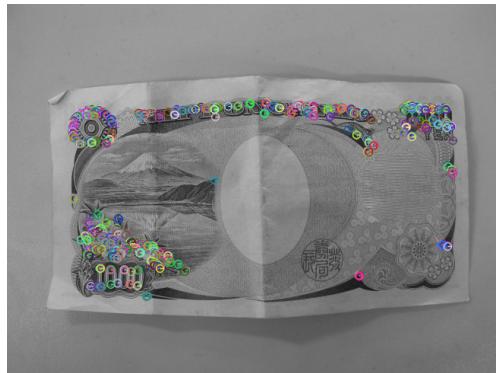
Figura 2: Resultados para Estuche



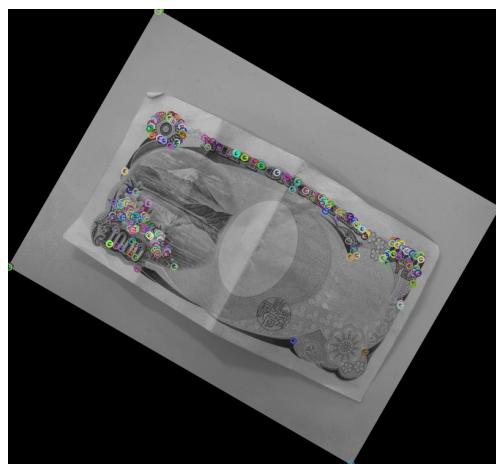
(a) Filtro a imagen original



(b) Filtro a imagen rotada



(c) Puntos de interés de imagen original



(d) Puntos de interés de imagen rotada

Figura 3: Resultados para Billete

3.2. Descriptores SIFT y BFMatcher

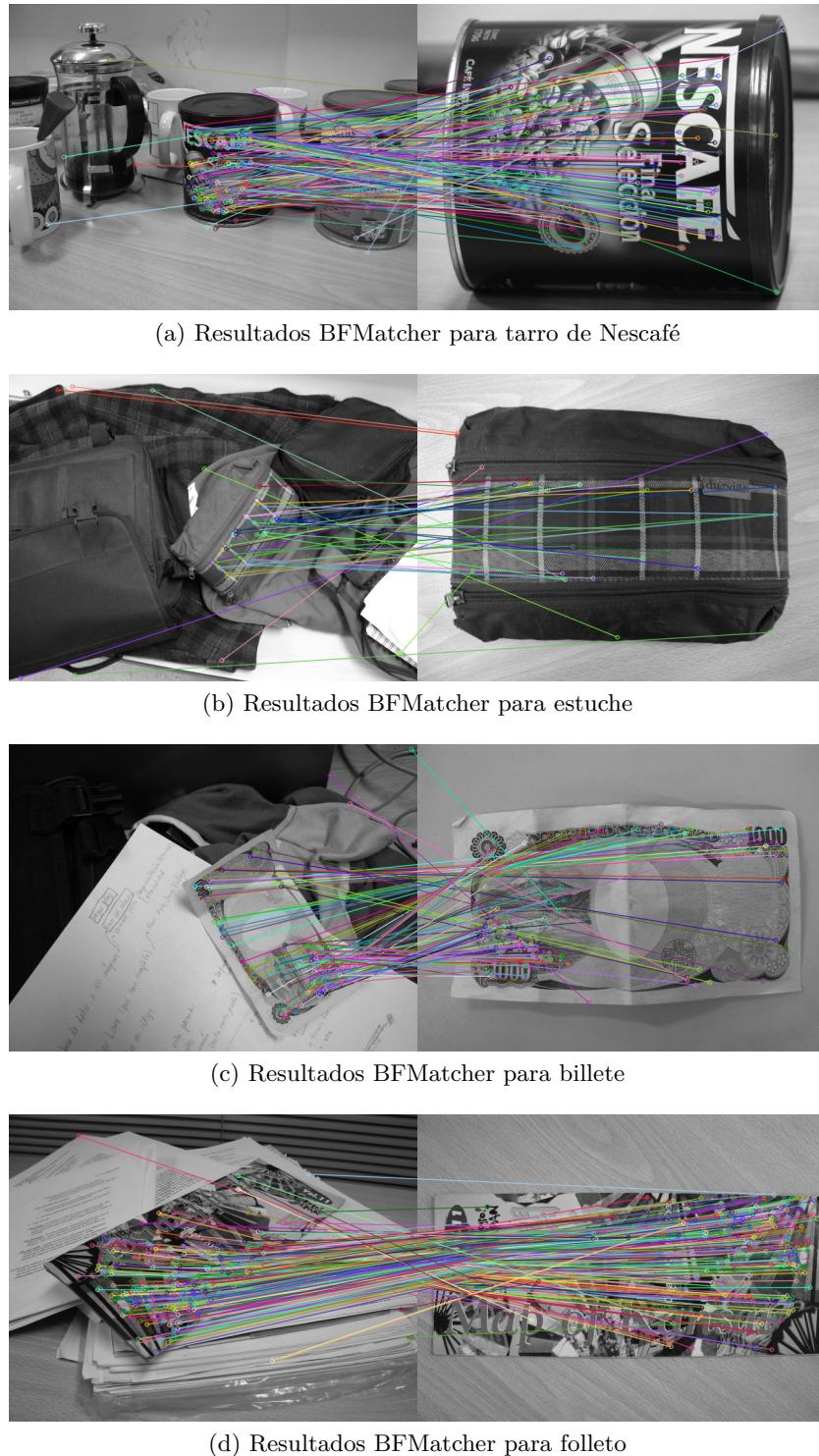
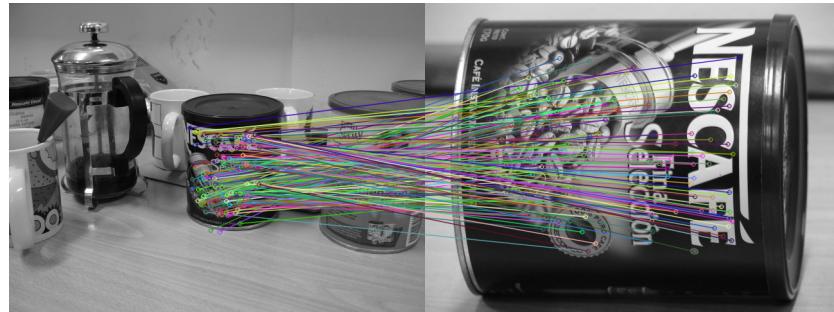


Figura 4: Resultados usando Descriptores SIFT y BFMatcher de OpenCV

3.3. Descriptores SIFT y RANSAC

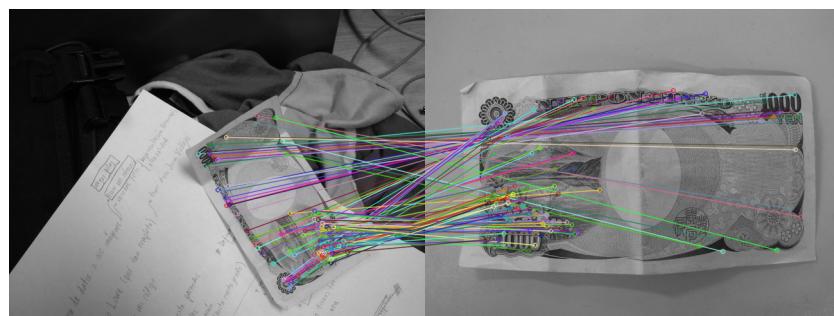
Para esta sección, se utilizaron 100 intentos, un umbral de error de posición de 100 y un umbral de consenso de 30.



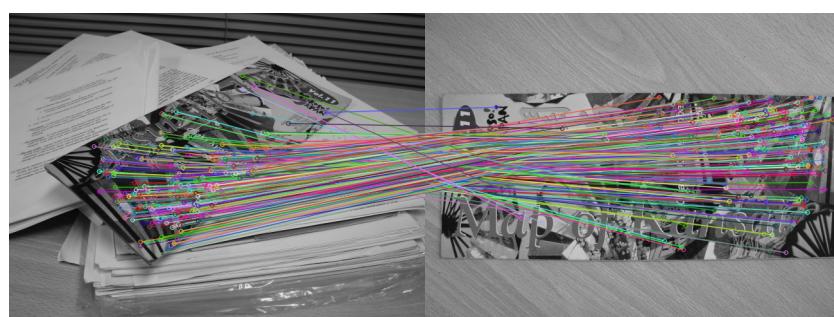
(a) Resultados RANSAC para tarro de Nescafé



(b) Resultados RANSAC para estuche



(c) Resultados RANSAC para billete



(d) Resultados RANSAC para folleto

Figura 5: Resultados usando Descriptores SIFT y algoritmo RANSAC

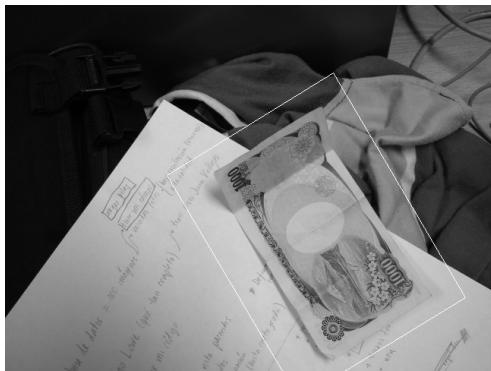
3.4. Romboídes Dibujados



(a) Romboide para tarro de Nescafé



(b) Romboide para estuche



(c) Romboide para billete



(d) Romboide para folleto

Figura 6: Romboídes dibujados sobre las imágenes de prueba

3.5. Resultados con Variación de Parámetros RANSAC

En este caso, se varían los parámetros establecidos en la sección anterior, para esto se utilizó lo siguiente:

- Intentos: 200
- Umbral de error de posición: 50
- Umbral de consenso: 40

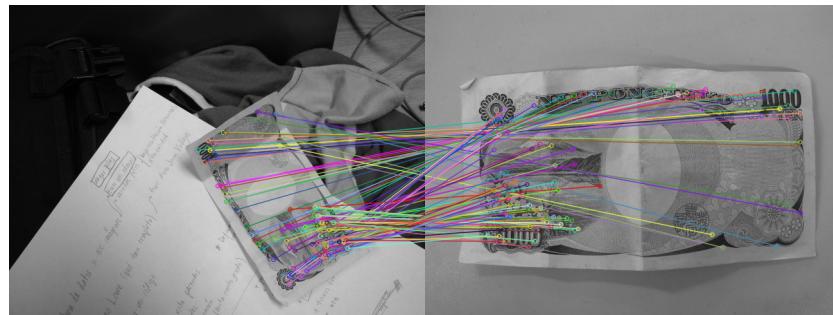
Con lo que se tienen los siguientes resultados para la imagen del tarro de Nescafé y del billete.



(a) Resultados RANSAC para tarro de Nescafé



(b) Romboide para tarro de Nescafé



(c) Resultados RANSAC para billete



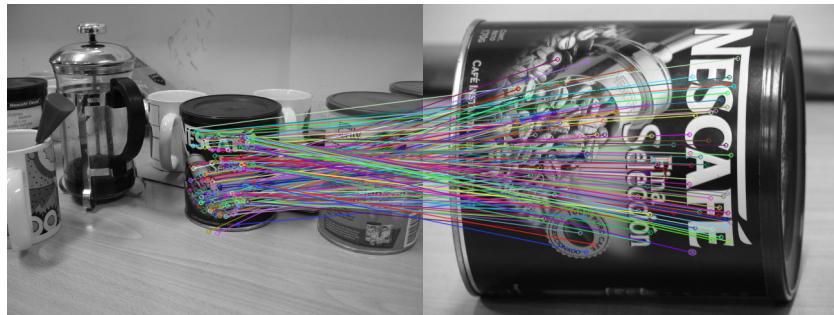
(d) Romboide para billete

Figura 7: Resultados luego de variar los tres parámetros principales de RANSAC

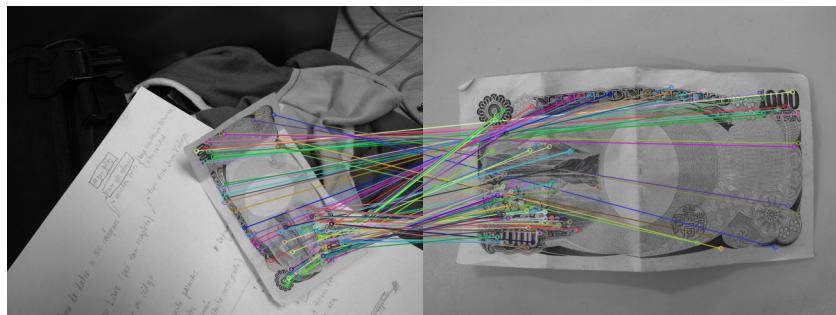
Luego, al variar los parámetros de la siguiente forma:

- Intentos: 100
- Umbral de error de posición: 20
- Umbral de consenso: 50

Se obtienen los siguientes resultados.



(a) Resultados RANSAC para tarro de Nescafé



(b) Resultados RANSAC para billete



(c) Romboide para tarro de Nescafé



(d) Romboide para billete

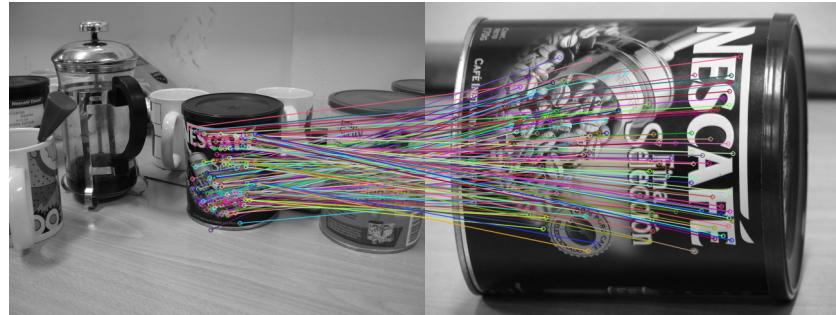
Figura 8: Resultados luego de variar los tres parámetros principales de RANSAC

Como penúltima prueba, se realizaron las siguientes modificaciones:

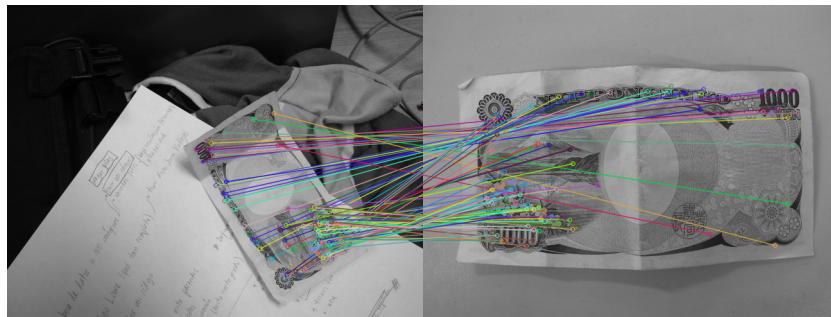
- Intentos: 500
- Umbral de error de posición: 10

- Umbral de consenso: 50

Esto, significa que el algoritmo está siendo más exigente, necesitando que la distancia entre calces sea mínima (10) y que el número de calces compatibles sea superior a 50. Los resultados son los siguientes:



(a) Resultados RANSAC para tarro de Nescafé



(b) Resultados RANSAC para billete



(c) Romboide para tarro de Nescafé



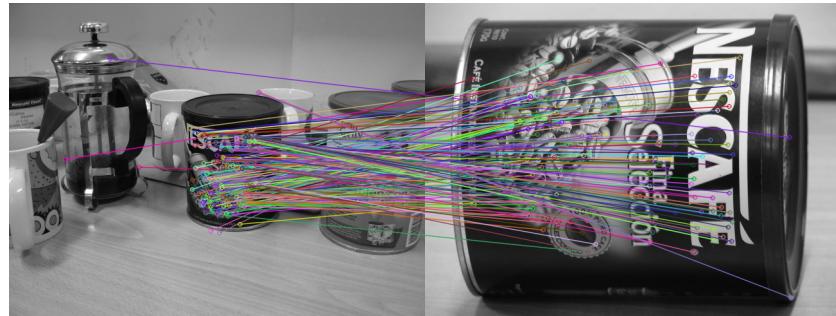
(d) Romboide para billete

Figura 9: Resultados luego de variar los tres parámetros principales de RANSAC

Finalmente, se realizó una prueba no exigente para analizar que sucede cuando permitimos que el error del algoritmo sea muy grande, esto se realizó con los siguientes parámetros.

- Intentos: 200
- Umbral de error de posición: 200
- Umbral de consenso: 10

Al implementar esto, se tienen los siguientes resultados.



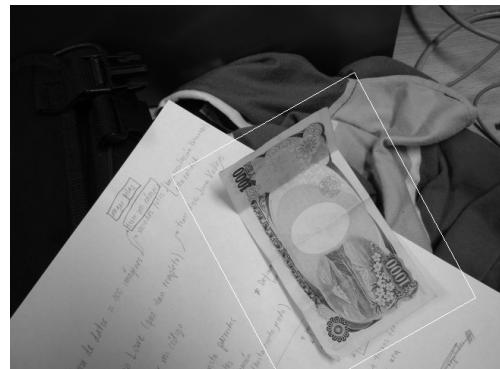
(a) Resultados RANSAC para tarro de Nescafé



(b) Resultados RANSAC para billete



(c) Romboide para tarro de Nescafé



(d) Romboide para billete

Figura 10: Resultados luego de variar los tres parámetros principales de RANSAC

4. Análisis de Resultados

4.1. Puntos de Interés de Harris

En primer lugar, al observar las imágenes (a) y (b) de las figuras 1, 2 y 3, se puede observar la detección de esquinas tanto para la imagen original como para la imagen rotada luego de ser aplicado el filtro de Harris. En estas, se visualizan pequeños puntos blancos que, en general, se mantienen con respecto a su versión rotada. Sin embargo, en algunos casos, la intensidad de estos puede verse afectada, lo cual puede ser causado debido a la aplicación de los gradientes individuales en x e y que

pueden variar según la orientación del objeto.

Luego, en las imágenes (c) y (d) de las mismas figuras (1, 2 y 3), se pueden observar los puntos de interés encontrados mediante las funciones getMáxima y getKeypoints (ver sección 6.2.1). En ambas, se puede notar que los puntos se mantienen en su mayoría tanto para la imagen original como rotada, concluyendo que el filtro de Harris es lo suficientemente robusto como para mantener estos resultados a pesar de que existan rotaciones en las imágenes.

4.2. Descriptores SIFT y BFMatcher

A partir de las imágenes (a, b, c, d) de la figura 4, se puede destacar lo siguiente al usar descriptores SIFT y BFMatcher de OpenCV. En primer lugar, para la imagen de Nescafé, se obtienen resultados bastante positivos, teniendo menos de 10 puntos de interés fuera de la imagen de consulta. Por otro lado, para la imagen del estuche, se presenta una cantidad considerable de puntos fuera del objetivo o que realizan un mal calce, por ejemplo, que desde el bolso (objeto extra) haya un match con la orilla del estuche. Luego, para la imagen del billete se obtienen muy buenos resultados, teniendo únicamente tres calces erróneos. Finalmente, para la imagen del folleto los resultados son bastante confusos visualmente, ya que se presenta una gran cantidad de calces entre puntos de ambas imágenes. No obstante, se observan sólo 6 calces fuera del objetivo, lo cual es aceptable. En la siguiente sección, se realizará la comparación correspondiente entre usar BFMatcher y el algoritmo RANSAC.

4.3. Descriptores SIFT y Algoritmo RANSAC

Al observar la figura 5 y las imágenes individuales (a, b, c, d) se puede destacar lo siguiente al usar descriptores SIFT y el algoritmo RANSAC. Para todas estas, se puede observar una gran mejoría con relación a los calces, debido a que todos estos se encuentran dentro del objetivo a reconocer. Al contrastar éstos con los resultados de la sección anterior, se puede concluir que el algoritmo RANSAC si mejora considerablemente los resultados para estos ejemplos en particular, evitando errores en los calces y permitiendo tener una mejor detección.

4.4. Romboídes en objetos

Al observar las imágenes individuales (a, b, c, d) de la figura 6, se puede destacar que en todas las imágenes el romboide logra encerrar gran parte del objeto detectado, presentando algunos errores para la imagen del estuche y del folleto. No obstante, estos resultados son positivos y aceptables para el experimento.

4.5. Variación de Parámetros RANSAC

Al observar las figuras 7, 8, 9 y 10, correspondientes a la variación de parámetros RANSAC, se puede notar que dependiendo de los parámetros, los resultados pueden ser aún más eficientes o más erróneos. Por ejemplo, en la primera variación, los resultados no presentan muchos cambios, debido a que el umbral de error de posición y consenso son cercanos al inicial. Por otro lado, al exigir mayor exactitud del algoritmo, como es el caso de la tercera variación, donde se define un bajo umbral de error y un alto valor de consenso, los calces son más precisos. Finalmente, como es el caso de la cuarta variación, donde el algoritmo se configura más flexible, con un alto umbral de error y un bajo umbral de consenso, se obtienen resultados erráticos, donde se presentan matches fuera del objeto

que se desea reconocer.

Para todos los casos, los romboídes dibujados permiten detectar eficientemente los objetos consultados, presentando ligeras variaciones según los parámetros.

Cabe destacar, que en esta sección se realizaron los experimentos para dos imágenes, sin embargo, llama la atención que para la imagen del estuche, en estas pruebas surgen errores debido a que el algoritmo no encuentra calces exigentes, esto posiblemente causado por el ruido presente en esta, ya que al fondo se puede distinguir una especie de chaqueta a cuadros que puede confundirse con el diseño del objeto, al igual que los otros bolsos presentes.

5. Conclusiones

Basandose en los resultados obtenidos se puede afirmar que se cumplió con el objetivo principal de la experiencia, el cual se enfocaba en implementar tanto un algoritmo de detección de puntos de interés con el método de Harris como un algoritmo capaz de detectar objetos usando descriptores SIFT y algoritmo RANSAC. Junto a esto, se logró comparar dos métodos de calces distintos como lo son BFMatcher de OpenCV y RANSAC implementado por el propio autor. Dentro de lo anterior, una de las complicaciones más grandes al desarrollar la tarea se originó en el manejo de los calces y los keypoints entregados por la librería utilizada, no obstante, se lograron solucionar los problemas y obtener resultados dentro de un rango aceptable.

En general, los resultados obtenidos fueron bastante positivos, logrando destacar puntos de interés y detectando objetos debidamente. No obstante, al momento de aplicar el filtro de Harris para la imagen del estuche, los puntos destacados en blanco son poco visibles, lo cual podría ser mejorado. Por otro lado, en la sección de los Romboides, para la imagen del estuche y del folleto se obtienen resultados un poco deficientes que podrían ser mejorados con una variación de los parámetros de los algoritmos.

Finalmente, cabe destacar que estas herramientas son de gran utilidad para reconocer y detectar objetos, permitiendo una gran variedad de aplicaciones, por ejemplo, podría utilizarse en una serie de imágenes (de un video) para detectar objetos que aparezcan momentáneamente y que sean importantes de visualizar, etc. No obstante, para esto se requeriría de una imagen de referencia (o varias de distintos ángulos en algunos casos).

Referencias

- [1] Fisher R. The RANSAC Algorithm. 2002. Disponible en: https://homepages.inf.ed.ac.uk/rbf/C_Vonline/LOCAL_COPIES/FISHER/RANSAC/
- [2] Universidad de Maryland, Departamento de Ciencias de Computación. RANSAC. Disponible en: <http://www.cs.umd.edu/~djacobs/CMSC426/RANSAC.pdf>

6. Anexos

6.1. Imágenes de referencia y prueba



(a) Estuche



(b) Nescafé



(c) Billete



(d) Folleto

Figura 11: Imágenes de referencia



Figura 12: Imágenes de prueba

6.2. Código Implementado

A continuación, se presenta el código implementado en el desarrollo de la tarea, cabe destacar, que al momento de imprimir (mostrar) los resultados, sólo se copia el fragmento utilizado para una imagen, ya que, para las demás simplemente se cambia el nombre.

Las secciones de código serán separadas según su función, es decir, se divide entre el código utilizado para los puntos de interés de Harris y luego para la aplicación de SIFT y RANSAC.

6.2.1. Puntos de Interés de Harris

```

1 % %cython
2 import cython
3 import numpy as np
4 cimport numpy as np
5
6 cpdef np.ndarray[np.float32_t, ndim=2] gradx(np.ndarray[np.float32_t, ndim=2] input):
7     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.shape[1]], dtype =
    ↪ np.float32)
8
9     cdef int i, j # indices para recorrer la imagen de entrada
10

```

```
11  for i in range(input.shape[0]): # se recorren todas las filas
12      for j in range(1, input.shape[1] - 1): # se evitan los bordes para calzar con la aproximación [-1, 0,
13          ↪ 1]
14          output[i,j] = (input[i,j+1] - input[i,j-1]) # se aplica la convolución
15
16
17
18 % %cython
19 import cython
20 import numpy as np
21 cimport numpy as np
22
23 cpdef np.ndarray[np.float32_t, ndim=2] grady(np.ndarray[np.float32_t, ndim=2] input):
24     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.shape[1]], dtype =
25         ↪ np.float32)
26
27     cdef int i, j # indices para recorrer la imagen de entrada
28
29     for i in range(1, input.shape[0] - 1): # se evitan los bordes para calzar con la aproximación [-1, 0,
30         ↪ 1]^T
31         for j in range(input.shape[1]): # se recorren todas las columnas
32             output[i,j] = (input[i+1,j] - input[i-1,j]) # se aplica la convolución
33
34     return output
35
36
37 % %cython
38 import cython
39 import numpy as np
40 cimport numpy as np
41
42 cpdef np.ndarray[np.float32_t, ndim=2] product(np.ndarray[np.float32_t, ndim=2] input1, np.
43     ↪ ndarray[np.float32_t, ndim=2] input2):
44
45     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input1.shape[0], input1.shape[1]], dtype =
46         ↪ np.float32)
47
48     cdef int i,j # indices para recorrer las imágenes, se asume que tienen los mismos tamaños
49
50     for i in range(input1.shape[0]):
51         for j in range(input1.shape[1]):
52             output[i,j] = input1[i,j] * input2[i,j] # Multiplicación pixel a pixel
53
54
55 % %cython
56 import cython
```

```

57 import numpy as np
58 cimport numpy as np
59
60 cpdef np.ndarray[np.float32_t, ndim=2] harris(np.ndarray[np.float32_t, ndim=2] mxx, np.ndarray[
61     ↪ np.float32_t, ndim=2] mxy, np.ndarray[np.float32_t, ndim=2] myy):
62
63     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([mxx.shape[0], mxx.shape[1]], dtype = np.
64     ↪ float32)
65     cdef int i, j # indices para recorrer las matrices mxx, mxy y myy.
66     cdef float det, tr # valores flotantes para guardar el valor det y tr para construir el cornerness de
67     ↪ harris
68
69     # se comienza iniciando los valores definidos en 0
70     det = 0
71     tr = 0
72     for i in range(mxx.shape[0]):
73         for j in range(mxx.shape[1]):
74             det = mxx[i,j] * myy[i,j] - (mxy[i,j] ** 2) # se calcula la variable det
75             tr = mxx[i,j] + myy[i,j] # se calcula la variable tr
76
77             output[i,j] = det - 0.04 * (tr ** 2) # se guarda el valor obtenido a partir de la combinación entre
78             ↪ det, tr y un factor (0.04)
79
80
81     %%cython
82     import cython
83     import numpy as np
84     cimport numpy as np
85
86     cpdef np.ndarray[np.float32_t, ndim=2] getMaxima(np.ndarray[np.float32_t, ndim=2] h, float val):
87
88         # Luego, hacer 1 los valores en los cuales se cumplen dos condiciones:
89         # h[r,c] es un maximo local respecto a sus 8 vecinos
90         # h[r,c] supera el valor val
91         cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([h.shape[0], h.shape[1]], dtype = np.float32
92             ↪ )
93         cdef int i, j, m, n # i, j recorren la matriz, mientras que m y n permiten recorrer bloques de 3x3
94             ↪ dentro de esta.
95
96         cdef float max, ij # se definen dos valores flotantes para facilitar la lectura
97
98         for i in range(1, h.shape[0] - 1):
99             for j in range(1, h.shape[1] - 1): # se recorre la imagen de entrada
100                 ij = h[i,j] # se guarda momentáneamente el valor en el cual estamos (posición i,j)
101                 max = 0 # se define un maximo
102                 for m in [-1, 0, 1]:
103                     for n in [-1, 0, 1]: # se recorren bloques de 3x3 (8 vecinos del valor actual i,j)

```

```
102     if h[i+m, j+n] >= ij: # si es mayor o igual al valor actual se guarda en la variable max
103         max = h[i+m, j+n]
104     if (max == ij) and (ij > val): # si el valor actual cumple ambas condiciones se cambia el 0 en la
105         → posición i,j por un 1.
106         output[i,j] = 1
107
108
109
110
111 # Esta funcion genera keypoints a partir de una imagen de entrada
112 # La imagen de entrada es la salida del filtro de Harris
113 # No es necesario modificar esta funcion
114 def getKeyPoints(input):
115     output = []
116     input_rows = input.shape[0]
117     input_cols = input.shape[1]
118     for r in range(input_rows):
119         for c in range(input_cols):
120             if input[r,c] > 0:
121                 kp = cv2.KeyPoint()
122                 kp.pt = (c,r)
123                 kp.size = 10
124                 kp.angle = 0
125                 output.append(kp)
126
127
128
129
130 import cv2
131 def harrisDetector(input, val):
132     input = np.float32( input )
133     # 1) Suavizar la imagen de entrada con cv.GaussianBlur( )
134     Blur_img = cv2.GaussianBlur(input, (5,5),2)
135     # 2) Calcular gradientes imx e imy (usando funciones de cython gradx y grady definidas arriba)
136     imx = gradx(Blur_img)
137     imy = grady(Blur_img)
138     # 3) Calcular momentos usando la funcion product( )
139     #    imxx = imx*imx (pixel a pixel)
140     #    imxy = imx*imy (pixel a pixel)
141     #    imyy = imy*imy (pixel a pixel)
142     imxx = product(imx, imx)
143     imxy = product(imx, imy)
144     imyy = product(imy, imy)
145     # 4) Suavizar momentos imxx, imxy, imyy con cv.GaussianBlur( )
146     mxx = cv2.GaussianBlur(imxx, (7,7), 1.2)
147     mxy = cv2.GaussianBlur(imxy, (7,7), 1.2)
148     myy = cv2.GaussianBlur(imyy, (7,7), 1.2)
149     # 5) Aplicar el filtro de Harris (usando funcion de Cython harris definida arriba)
150     post_harris = harris(mxx, mxy, myy)
151     # 6) Encontrar puntos maximos usando getMaxima()
```

```

152 max_points = getMaxima(post_harris, val)
153 # 7) Generar el listado de puntos usando getKeyPoints( )
154 points = getKeyPoints(max_points)
155 return points, post_harris
156
157
158
159 def do_rotate(img, angle):
160     h = img.shape[0]
161     w = img.shape[1]
162     cx = w // 2
163     cy = h // 2
164     m = cv2.getRotationMatrix2D((cx, cy), -angle, 1.0)
165     cosa = np.cos(angle * np.pi / 180.0)
166     sina = np.sin(angle * np.pi / 180.0)
167     nw = int((h * sina) + (w * cosa))
168     nh = int((h * cosa) + (w * sina))
169     m[0,2] += (nw / 2) - cx
170     m[1,2] += (nh / 2) - cy
171     return cv2.warpAffine(img, m, (nw, nh))
172
173
174 import numpy as np
175 import cv2
176 from google.colab.patches import cv2_imshow
177
178 img1 = cv2.imread('uch010a.jpg',0)
179 img2 = do_rotate(img1, 30)
180
181 kp1, h1 = harrisDetector(img1, 3e5)
182 kp2, h2 = harrisDetector(img2, 3e5)
183
184 res1 = cv2.drawKeypoints(img1, kp1, img1, None, cv2.
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
185 res2 = cv2.drawKeypoints(img2, kp2, img2, None, cv2.
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
186
187 print('Resultados de filtro de Harris')
188 cv2_imshow(h1 * 500 / np.max(np.max(h1)))
189 cv2_imshow(h2 * 500 / np.max(np.max(h2)))
190 print('Puntos de interes')
191 cv2_imshow(res1)
192 cv2_imshow(res2)

```

6.2.2. Descriptores SIFT y Algoritmo RANSAC

```

195 import math
196
197 def genTransform(match, keypoints1, keypoints2):
198     # Calcular transformacion de semejanza (e,theta,tx,ty) a partir de un calce "match".

```

```
199 # Se debe notar que los keypoints de OpenCV tienen orientacion en grados.  
200  
201 # se extraen índices de referencia y prueba  
202 indexref = match.trainIdx  
203 indexpru = match.queryIdx  
204  
205 # se definen las posiciones de referencia y prueba a partir de los keypoints  
206 (xref, yref) = keypoints1[indexref].pt  
207 (xpru, ypru) = keypoints2[indexpru].pt  
208  
209 # se define la escala de referencia y prueba  
210 sigmaref = keypoints1[indexref].size  
211 sigmapru = keypoints2[indexpru].size  
212  
213 # se definen los angulos phi para cada uno, se transforma a radianes para usar funciones  
    → predefinidas en python (math)  
214 phiref = (keypoints1[indexref].angle) * math.pi / 180  
215 phipru = (keypoints2[indexpru].angle) * math.pi / 180  
216  
217 # se define el factor de escala entre imágenes e  
218 e = sigmapru / sigmaref  
219  
220 # se define ángulo theta como la resta entre phi_pru y phi_ref  
221 theta = phipru - phiref  
222  
223 # se define la traslación (tx, ty)  
224 tx = xpru - e * (xref * math.cos(theta) - yref * math.sin(theta))  
225 ty = ypru - e * (xref * math.sin(theta) + yref * math.cos(theta))  
226  
227 return (e, theta, tx, ty) # se retorna la transformación de semejanza  
228  
229  
230 def computeConsensus(matches, keypoints1, keypoints2, e, theta, tx, ty, umbralpos):  
231     # Calcular el número de matches compatibles con la transformación (e, theta, tx, ty) indicada  
232     # Debe devolver el número de matches compatibles, y una lista con los matches compatibles  
233  
234     # se define lista que guarda los matches compatibles y un valor que los contará  
235     compatible_match = []  
236     number_of_matches = 0  
237  
238     for match in matches: # se recorren los matches  
239         indexref = match.trainIdx  
240         indexpru = match.queryIdx  
241  
242         (xref, yref) = keypoints1[indexref].pt  
243         (xpru, ypru) = keypoints2[indexpru].pt  
244  
245         x_t = tx + e * (xref * math.cos(theta) - yref * math.sin(theta))  
246         y_t = ty + e * (xref * math.sin(theta) + yref * math.cos(theta))  
247  
248         # se calcula la distancia euclidiana entre (x_t, y_t) y (xpru, ypru)
```

```
249 euclidean_dist = math.sqrt((x_t - xpru)**2 + (y_t - ypru)**2)
250
251 # se analiza la distancia con el umbral definido
252 if euclidean_dist < umbralpos:
253     number_of_matches += 1
254     compatible_match.append(match)
255
256 return (number_of_matches, compatible_match)
257
258 import random
259 def ransac(matches, keypoints1, keypoints2):
260     n_matches = []
261     accepted = []
262     intentos = 100
263     umbral_error = 100
264     umbral_consenso = 30
265
266     for i in range(intentos): # se repite el consenso según los intentos definidos
267
268         random_index = random.randint(0, len(matches)-1) # se busca un indice aleatorio para buscar el
269             ↪ calce al azar
270         match = matches[random_index] # se extrae el calce aleatorio
271         (e, theta, tx, ty) = genTransform(match, keypoints1, keypoints2) # se busca transformación de
272             ↪ semejanza
273
274         (count, calces) = computeConsensus(matches, keypoints1, keypoints2, e, theta, tx, ty,
275             ↪ umbral_error) # se realiza el consenso
276
277         if count > umbral_consenso: # se guardan los calces siempre y cuando sean mayores al umbral
278             n_matches.append(count)
279             accepted.append(calces)
280
281     return accepted[index]
282
283 def calcAfin(matches, keypoints1, keypoints2):
284     # Por hacer: calcular la transformacion afin mediante minimos cuadrados a partir de "matches"
285
286     # definimos matrices que ayudarán a calcular la transformación
287     A = []
288     b = []
289     for match in matches: # recorremos todos los calces
290         indexref = match.trainIdx
291         indexpru = match.queryIdx
292
293         (xref, yref) = keypoints1[indexref].pt
294         (xpru, ypru) = keypoints2[indexpru].pt
295
296         f1 = [xref, yref, 0, 0, 1, 0]
```

```
297 f2 = [0, 0, xref, yref, 0, 1]
298
299 A.append(f1)
300 A.append(f2)
301 b.append(xpru)
302 b.append(ypru)
303
304 A = np.array(A)
305 A_tr = np.transpose(A)
306 b = np.array(b)
307
308 output = np.matmul(np.matmul(np.linalg.inv(np.matmul(A_tr, A)), A_tr), b) # se realiza operaci
    ↪ ón ( $A^T * A$ ) $^{-1} * A^T * b$  (enunciado)
309
310 # reordenamiento
311 transformacion = []
312 fila1 = [output[0], output[1], output[4]]
313 fila2 = [output[2], output[3], output[5]]
314 transformacion.append(fila1)
315 transformacion.append(fila2)
316
317 transformacion = np.array(transformacion)
318
319 return transformacion
320
321 # Esta función devuelve la imagen "input1" con un romboide dibujado
322 # El romboide representa el rectángulo de la imagen "input2" proyectada en la imagen "input1"
323 # La proyección se realiza a partir de la transformación "transf"
324
325 def drawProjAfin(transf, input1, input2):
326     w = input2.shape[1];
327     h = input2.shape[0];
328     sq1 = [0,0]
329     sq2 = [w-1,0]
330     sq3 = [w-1,h-1]
331     sq4 = [0,h-1];
332     p1 = [0,0]
333     p2 = [0,0]
334     p3 = [0,0]
335     p4 = [0,0]
336     print("Tamano " + str(transf.shape[1]) + " x " + str(transf.shape[0]));
337     print("Posicion " + str(transf[0,2]) + " , " + str(transf[1,2]));
338     print("Matriz " + str(transf[0,0]) + " " + str(transf[0,1]) + " " + str(transf[0,2]) + " " + str(transf
    ↪ [1,0]) + " " + str(transf[1,1]) + " " + str(transf[1,2]));
339
340     p1[0] = transf[0,0] * sq1[0] + transf[0,1] * sq1[1] + transf[0,2];
341     p1[1] = transf[1,0] * sq1[0] + transf[1,1] * sq1[1] + transf[1,2];
342     p2[0] = transf[0,0] * sq2[0] + transf[0,1] * sq2[1] + transf[0,2];
343     p2[1] = transf[1,0] * sq2[0] + transf[1,1] * sq2[1] + transf[1,2];
344     p3[0] = transf[0,0] * sq3[0] + transf[0,1] * sq3[1] + transf[0,2];
345     p3[1] = transf[1,0] * sq3[0] + transf[1,1] * sq3[1] + transf[1,2];
```

```
346 p4[0] = transf[0,0] * sq4[0] + transf[0,1] * sq4[1] + transf[0,2];
347 p4[1] = transf[1,0] * sq4[0] + transf[1,1] * sq4[1] + transf[1,2];
348
349 p1 = (int(p1[0]), int(p1[1]))
350 p2 = (int(p2[0]), int(p2[1]))
351 p3 = (int(p3[0]), int(p3[1]))
352 p4 = (int(p4[0]), int(p4[1]))
353
354
355 out = np.copy(input1);
356 cv2.line(out, p1, p2, (255,255,255));
357 cv2.line(out, p2, p3, (255,255,255));
358 cv2.line(out, p3, p4, (255,255,255));
359 cv2.line(out, p4, p1, (255,255,255));
360 return out;
361
362 # Esta funcion ya esta lista, no debe ser modificada
363 def filterMatches(matches):
364     # Apply ratio test
365     points1 = []
366     points2 = []
367     good = []
368
369     for m,n in matches:
370         if m.distance < 0.75*n.distance: # 0.75
371             good.append(m)
372             points1.append(kp1[m.queryIdx].pt)
373             points2.append(kp2[m.trainIdx].pt)
374     return np.array(points1), np.array(points2), good
375
376
377 import numpy as np
378 import cv2
379 from matplotlib import pyplot as plt
380 from google.colab.patches import cv2_imshow
381
382 img2 = cv2.imread('uch010a.jpg',0)
383 img1 = cv2.imread('uch010b.jpg',0)
384
385 # Initiate SIFT detector
386 sift = cv2.SIFT_create()
387
388 # find the keypoints and descriptors with SIFT
389 kp1, des1 = sift.detectAndCompute(img1,None)
390 kp2, des2 = sift.detectAndCompute(img2,None)
391
392 ##### BFMatcher with default params
393 bf = cv2.BFMatcher()
394 matches = bf.knnMatch(des1,des2, k=2)
395 points1, points2, good = filterMatches(matches)
396
```

```
397 img_match = cv2.drawMatches(img1,kp1,img2,kp2,good, None, flags=cv2.  
    ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)  
398  
399 cv2_imshow(img_match)  
400  
401 # 1) Obtener los calces aceptados a partir de la función ransac(good, kp2, kp1)  
402 Ransac_points = ransac(good, kp2, kp1)  
403 # 2) Dibujar los calces aceptados  
404 calces_img = cv2.drawMatches(img1,kp1,img2,kp2,good, None, flags=cv2.  
    ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)  
405 cv2_imshow(calces_img)  
406 # 3) Calcular la transformacion afin  
407 transformacion = calcAfin(Ransac_points, kp2, kp1)  
408 # 4) Dibujar la imagen con el romboide superpuesto  
409 print(transformacion) # Para analizar los valores dentro de esta  
410 cv2_imshow(drawProjAfin(transformacion, img1, img2))
```