

Tarea 2

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

Integrantes: Joaquín Zepeda
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Santiago de Chile

Índice de Contenidos

1. Introducción	1
2. Desarrollo	2
2.1. Parte 1: Cálculo de puntos de interés Harris	2
2.2. Parte 2: Reconocimiento de objetos particulares	6
3. Conclusión	13
4. Anexos	14
Referencias	26

Índice de Figuras

1. Imágenes de prueba	2
2. Salida del filtro de harris para las imágenes de prueba.	4
3. Puntos de interés en las imágenes de prueba.	5
4. Puntos de interés en las imágenes de prueba rotadas 30°.	5
5. Puntos de interés en las imágenes de prueba rotadas 30°.	5
6. Calces de las imágenes con el detector SIFT que viene implementado en opencv y con los calces compatibles usando RANSAC.	7
7. Reconocimiento de objetos particulares, estos se identifican con los romboídes.	10
8. Prueba 1.	11
9. Prueba 2.	12
10. Prueba 3.	13

1. Introducción

El procesamiento digital de imágenes tiene un papel importante en la sociedad, siendo estas una base para el reconocimiento de objetos/personas, diagnosticar condiciones médicas, astronomía, etc. este corresponde a un conjunto de técnicas que permiten cambiar la información que contiene la imagen con el fin de tener una mejor representación de esta y/o resaltar o suprimir alguna característica. Esta disciplina dio sus comienzos cuando comenzaron a digitalizarse las imágenes, es decir, se representaron las imágenes mediante matrices las cuales se guardan en memoria. Dentro de computadoras se pueden operar técnicas matemáticas y cálculos los cuales en la actualidad se puede realizar medianamente rápido, convirtiéndose en una disciplina muy útil en múltiples áreas. En el desarrollo de esta disciplina, aparecieron los puntos de interés en la imagen que tienen como objetivo identificar zonas claves.

El objetivo de esta tarea es implementar un detector que calcule los puntos de interés mediante el método de Harris e implementar un detector de objetos usando puntos de interés y descriptores SIFT, usando RANSAC, esto con el fin de poner en práctica los conceptos vistos en clases y poder apreciar en diferentes situaciones los resultados de estos métodos. Además, se busca evaluar y analizar la robustez de estos métodos de detección con respecto a las rotaciones y/o traslaciones, lo cual es algo común en la vida diaria pues no se tienen fotos ideales en general.

Esta tarea se desarrolla utilizando el lenguaje de programación Python y además se completan funciones en Cython, lo cual nos permite escribir código en C/C++ desde Python permitiendo tener un programa más rápido, a continuación se muestran y analizan los resultados en la sección de Desarrollo para luego finalizar con las conclusiones.

2. Desarrollo

2.1. Parte 1: Cálculo de puntos de interés Harris

Para esta sección se utilizan las siguientes imágenes:



(a) Img1



(b) Img2



(c) Img3

Figura 1: Imágenes de prueba

- a) En este ítem se completan las funciones *grad_x()* y *grad_y()* que calculan el gradiente en x e y respectivamente. Para esto se utiliza una aproximación de la derivada discreta, la cual se calcula a partir de la **convolución** de la imagen con el kernel que representa la derivada centrada discreta aproximada [1]. Estos kernels corresponden a:

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

$$K_y = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \quad (2)$$

```

1 cpdef np.ndarray[np.float32_t, ndim=2] gradx(np.ndarray[np.float32_t, ndim=2] input):
2     # Calculo aproximado del gradiente en x de una imagen
3     cdef int rows,cols, i,j
4
5     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.shape[1]],
6             dtype = np.float32)
7
8     # tamano de la imagen
9     rows = input.shape[0]
10    cols = input.shape[1]
11
12    for i in range(rows):
13        for j in range(2,cols-2):
14            output[i][j-2] = -1*input[i][j-2]+0*input[i][j-1]+1*input[i][j]
15
16    return output

```

```

17 cpdef np.ndarray[np.float32_t, ndim=2] grady(np.ndarray[np.float32_t, ndim=2] input):
18     # Calculo aproximado del gradiente en y de una imagen
19     cdef int rows,cols, i,j
20     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.shape[1]],
21     ↪ dtype = np.float32)
22
23     # tamano de la imagen
24     rows = input.shape[0]
25     cols = input.shape[1]
26
27     for j in range(cols):
28         for i in range(2,rows-2):
29             output[i-2][j] = -1*input[i-2][j]+0*input[i-1][j]+1*input[i][j]
30
31     return output

```

Se realiza la convolución recorriendo completamente la matriz.

- b) Para el cálculo del filtro de Harris, se utiliza el *cornerness_{harris}* pixel a pixel, para esto se utiliza la siguiente expresión en función del determinante (det) y la traza(tr):

$$\text{cornerness}_{\text{harris}} = \text{det} - 0.04 * \text{tr}^2 \quad (3)$$

En donde:

$$\text{det} = m_{xx} * m_{yy} - m_{xy}^2 \quad (4)$$

$$\text{tr} = m_{xx} + m_{yy} \quad (5)$$

A continuación se muestra el extracto de código donde se realiza esta función:

```

1 cpdef np.ndarray[np.float32_t, ndim=2] harris(np.ndarray[np.float32_t, ndim=2] mxx, np.
2     ↪ ndarray[np.float32_t, ndim=2] mxy, np.ndarray[np.float32_t, ndim=2] myy):
3     # POR HACER: Calcular el filtro de Harris a partir de (mxx, mxy, myy)
4     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([mxx.shape[0], mxx.shape[1]],
5     ↪ dtype = np.float32)
6
7     #recorremos pixel a pixel
8     for i in range(mxx.shape[0]):
9         for j in range(mxx.shape[1]):
10            output[i][j] = mxx[i][j]*myy[i][j] - mxy[i][j]**(2) - 0.04*(mxx[i][j] + myy[i][j])**2
11    return output

```

- c) En este ítem se completa la función *getMaxima()* la cual selecciona los máximos locales de la salida del filtro de harris. Para esto se seleccionan como máximos locales los puntos que cumplen con 2 condiciones:

1. El punto a revisar es un máximo local con respecto a sus 8 vecinos.

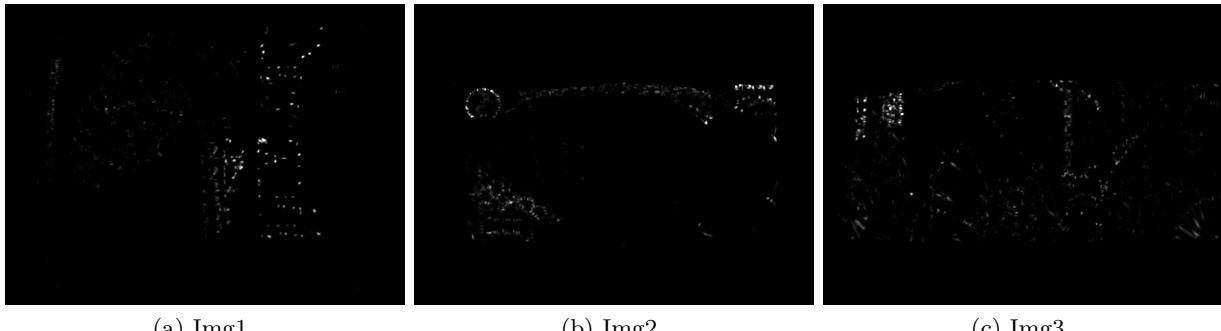
2. El punto a revisar es mayor a un valor umbral.

```

1 cpdef np.ndarray[np.float32_t, ndim=2] getMaxima(np.ndarray[np.float32_t, ndim=2] h,
2     ↪ float val):
3     cdef int rows,cols, r,c
4     # tamano de la imagen
5     rows = h.shape[0]
6     cols = h.shape[1]
7     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([h.shape[0], h.shape[1]], dtype
8     ↪ = np.float32)
9     for r in range(1,rows-1):
10        for c in range(1,cols-1):
11            if h[r,c]>val and h[r,c]>=max([h[r-1,c-1],h[r-1,c],h[r-1,c+1] ,h[r,c-1] ,h[r,c+1] ,h[r+1,c
12            ↪ -1] ,h[r+1,c],h[r+1,c+1]]):
13                output[r,c] = 1
14
15
16    return output

```

- d) Los resultados de las imágenes filtradas por el filtro de harris se muestran a continuación, se pueden observar imágenes con un mayor porcentaje de negro, pero con puntos/pixeles blancos indicando que son puntos de interés.



(a) Img1

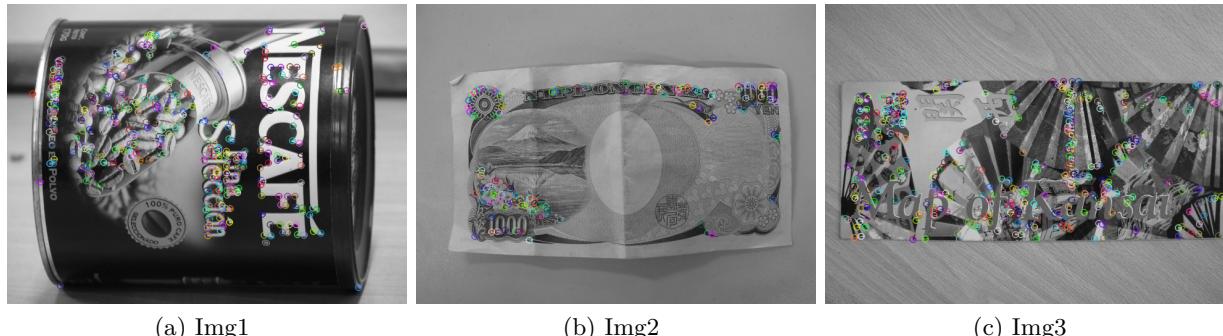
(b) Img2

(c) Img3

Figura 2: Salida del filtro de harris para las imágenes de prueba.

- e) Finalmente en las figuras 3 y 4 se observan los resultados del *harrisDetector()* en las imágenes de prueba utilizando $1e5$, $1e5$ y $3.5e4$ como valores umbrales para las 3 imágenes de prueba respectivamente.

Con respecto a la invarianza/robustez de los puntos de interés detectados mediante Harris cuando la imagen es rotada en 30 grados, se puede observar que en la imagen 2, figura 5 los puntos de interés en general se mantienen en la imagen rotada, pero también algunos desaparecen en la imagen rotada, por ejemplo algunos puntos de la esquina inferior derecha desaparecen con la imagen rotada, a pesar de esto la consistencia en general es bastante buena. Por otro lado en las imágenes 1 y 3 los resultados de consistencia son más evidentes, los puntos se mantienen en general en las zonas claves que se muestran lo cual era lo esperado. A pesar de esto, no se detectaron todos los bordes, esto se puede deber al filtrado que se le realizó o al valor del umbral seleccionado.

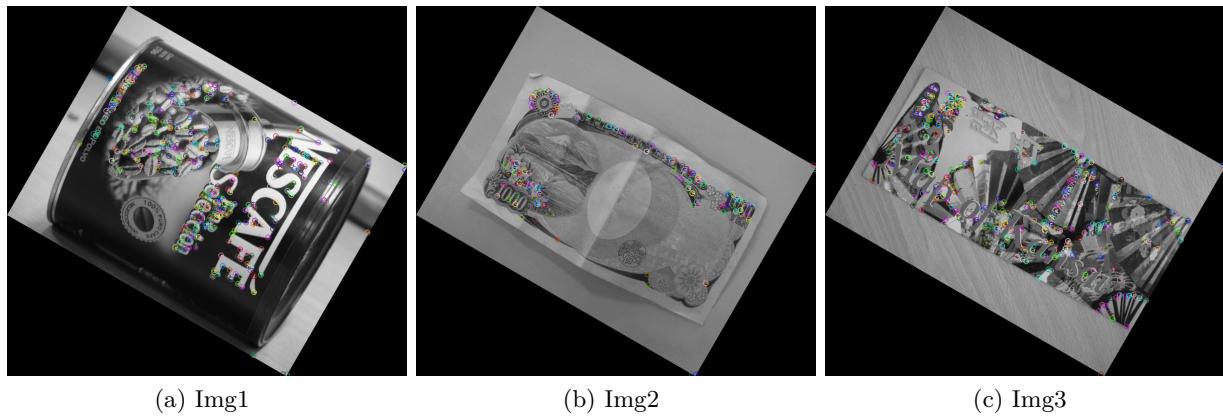


(a) Img1

(b) Img2

(c) Img3

Figura 3: Puntos de interés en las imágenes de prueba.

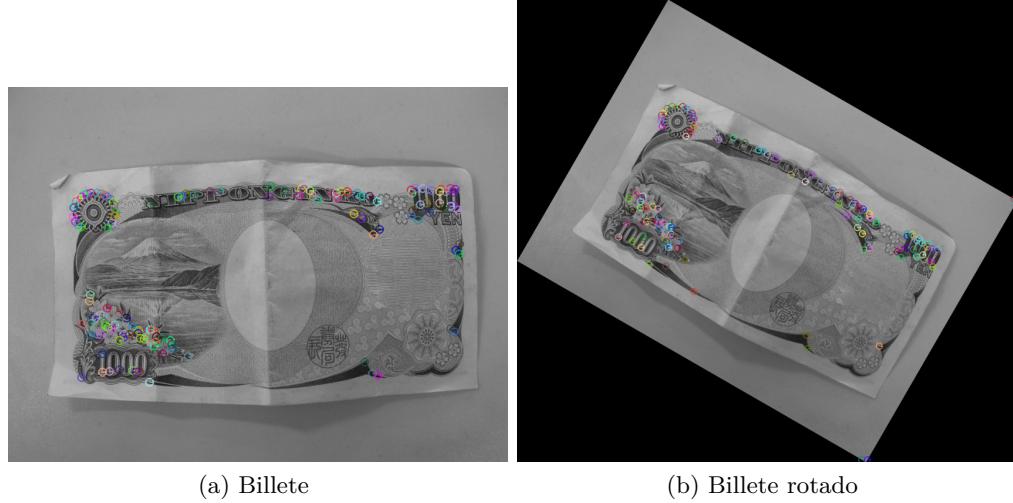


(a) Img1

(b) Img2

(c) Img3

Figura 4: Puntos de interés en las imágenes de prueba rotadas 30°.



(a) Billete

(b) Billete rotado

Figura 5: Puntos de interés en las imágenes de prueba rotadas 30°.

2.2. Parte 2: Reconocimiento de objetos particulares

1. Se implementa la función *genTransform()* la cual permite generar una transformación de semejanza a partir de un calce. Para completar esta función se rescatan los puntos, el tamaño y los ángulos a partir de los atributos que poseen los keypoints. Además, la transformación de semejanza que proyecta un punto de interés se determina a partir de las ecuaciones presentadas en el punto 5 de los antecedentes generales que presenta el enunciado, estás son:

$$e = \frac{\sigma_{pru}}{\sigma_{ref}} \quad (6)$$

$$\theta = \varphi_{pru} - \varphi_{ref} \quad (7)$$

$$tx = x_{pru} - e(x_{ref}\cos(\theta) - y_{ref}\sin(\theta)) \quad (8)$$

$$ty = y_{pru} - e(x_{ref}\sin(\theta) + y_{ref}\cos(\theta)) \quad (9)$$

Luego se retornan estos valores como (e, θ, tx, ty) .

```

1 import numpy as np
2
3 def genTransform(match, keypoints1, keypoints2):
4     # Calcular transformacion de semejanza (e,theta,tx,ty) a partir de un calce "match".
5     # La idea es que los puntos de train son de referencia y los de query son los de prueba
6     trainIdx = match.trainIdx
7     queryIdx = match.queryIdx
8
9     # Rescatamos los valores a partir de los atributos de la clase
10    # Puntos de referencia
11    (xt, yt) = keypoints1[trainIdx].pt
12    #puntos de prueba
13    (xr, yr) = keypoints2[queryIdx].pt
14
15    e = keypoints2[queryIdx].size / keypoints1[trainIdx].size
16    theta = keypoints2[queryIdx].angle - keypoints1[trainIdx].angle
17    tx = xr - e * (xt * np.cos(theta) - yt * np.sin(theta))
18    ty = yr - e * (xt * np.sin(theta) + yt * np.cos(theta))
19    return (e, theta, tx, ty)

```

2. El objetivo de este ítem es obtener la transformación de semejanza con mayor consenso y los calces compatibles usando RANSAC.

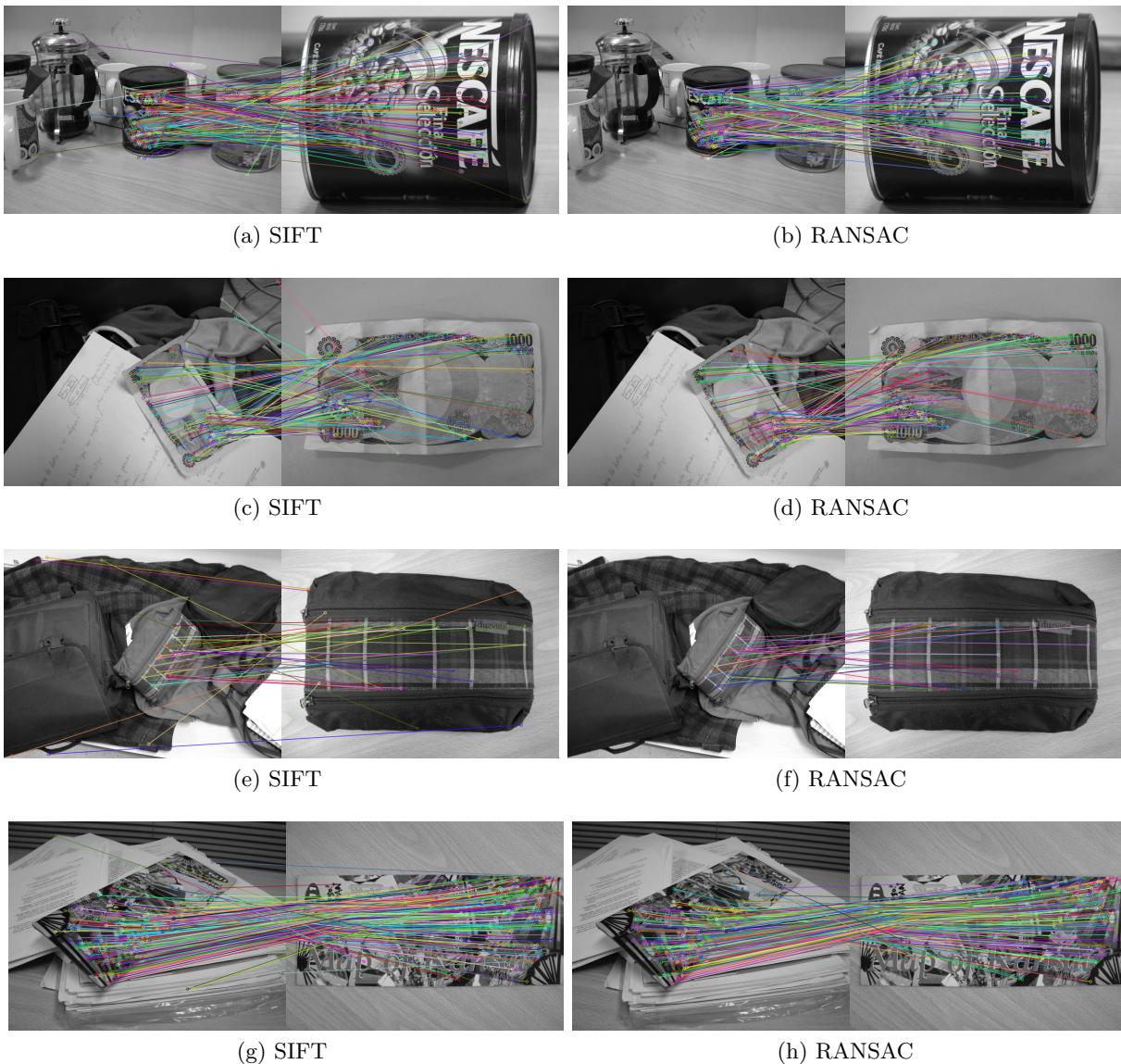


Figura 6: Calces de las imágenes con el detector SIFT que viene implementado en opencv y con los calces compatibles usando RANSAC.

Como se puede observar en la figura 6, se obtienen mejores resultados de los calces usando RANSAC, en efecto en hay algunos calces fuera de la figura a detectar en el detector de SIFT, en cambio usando RANSAC todos los calces en los 4 pares de imágenes de prueba corresponden a puntos dentro de los objetos a detectar que es lo que se esperaba. Es importante al programar RANSAC el de obtener la transformación de semejanza con mayor consenso, pues así se obtienen los mejores resultados. La función *RANSAC()*. A continuación se muestra la implementación del las funciones *computeConsensus()* y *RANSAC()* las cuales calculan los calces compatibles con la hipótesis y selecciona la hipótesis con el mayor consenso respectivamente.

```
1 def computeConsensus(matches, keypoints1, keypoints2, e, theta, tx, ty, umbralpos):  
2     # Calcular el número de matches compatibles con la transformación (e, theta, tx, ty)
```

```
    ↳ indicada
3   # Debe devolver el número de matches compatibles, y una lista con los matches
    ↳ compatibles

4
5   matches_c = []
6   for match in matches:
7       trainIdx = match.trainIdx
8       queryIdx = match.queryIdx
9       (xt, yt) = keypoints1[trainIdx].pt
10      (xr, yr) = keypoints2[queryIdx].pt

11
12      d1 = e*( np.cos(theta)*xt - np.sin(theta)*yt ) + tx
13      d2 = e*( np.sin(theta)*xt + np.cos(theta)*yt ) + ty

14
15      d = np.sqrt( (d1-xr)**2 + (d2-yr)**2 )
16      if d < umbralpos:
17          matches_c.append(match)

18
19      # se retorna un conteo de los matches y la lista que los contiene
20      return (len(matches_c), matches_c)

21
22      import random

23
24      def ransac(matches, keypoints1, keypoints2):
25          accepted = []

26
27          #definimos los parametros para luego ir revisndo los resultados
28          intentos = 100
29          umbralpos = 80
30          umbralcons = 30
31          n = len(matches)

32
33          L_nums = []
34          L_accepted = []

35
36          for _ in range(intentos):
37              #elegimos un indice al azar con random.randint
38              i = random.randint(0,n-1)
39              match = matches[i]
40              (e, theta, tx, ty) = genTransform(match, keypoints1, keypoints2)

41
42              (num,accepted) = computeConsensus(matches, keypoints1, keypoints2, e, theta, tx, ty,
43              ↳ umbralpos)
44              #si el numero de matches es mayor al umbral, se retorna la lista de calces considerados
45              ↳ correctos
46              if num > umbralcons:
47                  L_nums.append(num)
48                  L_accepted.append(accepted)

49      #encontramos el indice de la transformación de semejanza con mayor consenso y los calces
```

```

49     ↪ compatibles
50     imax = L_nums.index(max(L_nums))
51
52     #retornamos la lista de calces de mayor concenso
53     return L_accepted[imax]

```

3. A partir del conjunto de calces, se implementa la función que encuentra la transformación afín usando mínimos cuadrados a partir de un conjunto de calces, esto se realiza utilizando las ecuaciones definidas en el ítem 3 de los antecedentes generales. A continuación se presenta la definición y ecuaciones para calcular la transformación afín:

$$\begin{pmatrix} x_{PRU} \\ y_{PRU} \end{pmatrix} = \begin{pmatrix} m_{XX} & m_{XY} \\ m_{YX} & m_{YY} \end{pmatrix} \begin{pmatrix} x_{REF} \\ y_{REF} \end{pmatrix} + \begin{pmatrix} t_X \\ t_Y \end{pmatrix}$$

Alternativamente, dejando los parámetros de la transformación afín separados en un vector, se puede describir como: O en forma equivalente como:

$$Ax = b$$

De este modo, si se conoce A y b , y se quiere resolver el siguiente problema:

$$\operatorname{argmin}_x \|Ax - b\|^2$$

la solución se puede obtener como:

$$x^* = (A^T A)^{-1} A^T b \quad (10)$$

A partir de esto se completa la función *calcAfin()* y se utiliza el código que se entregó con el enunciado para graficar las imágenes con sus respectivos romboídes que identifican los objetos.

```

1   def calcAfin(matches, keypoints1, keypoints2):
2       # Por hacer: calcular la transformacion afin mediante minimos cuadrados a partir de "
3       ↪ matches"
4       # x* = (A'A)^(-1)A'b
5       a = []
6       b = []
7
8       for match in matches:
9           #print("aqui")
10          trainIdx = match.trainIdx
11          queryIdx = match.queryIdx
12
13          (xt, yt) = keypoints1[trainIdx].pt
14          (xr, yr) = keypoints2[queryIdx].pt
15
16          a.append([xt, yt, 0, 0, 1, 0])
17          a.append([0, 0, xt, yt, 0, 1])
18          b.append(xr)
19          b.append(yr)

```

```

19
20 A = np.array(a)
21
22 B = np.array(b)
23 At = np.transpose(A)
24 x = np.matmul(np.matmul(np.linalg.inv((np.matmul(At,A))),At),B)
25
26 return x
27

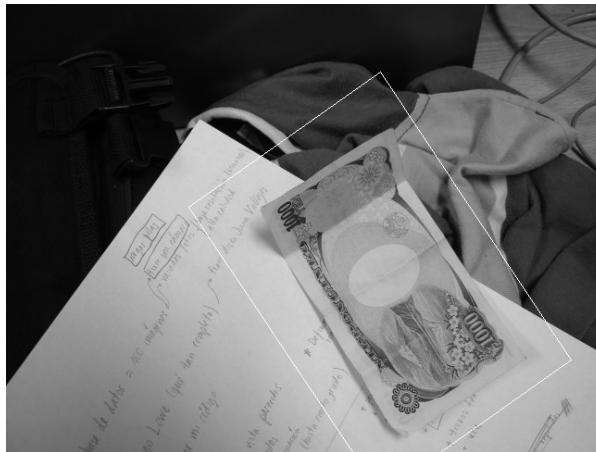
```



(a) Img1



(b) Img2



(c) Img3



(d) Img4

Figura 7: Reconocimiento de objetos particulares, estos se identifican con los romboideos.

Como se puede observar en la figura 7, los objetos con los parámetros recomendados fueron posibles de identificar, lo cual era lo que se buscaba, a pesar de esto imágenes como la imagen 4 de esta figura, en donde no se logra encuadrar completamente el objeto, por otro lado la imagen del billete también no se encuadra de buena manera. Esto se debe a la distribución de los calces,

los cuales no son uniformes en todo el objeto.

4. Pruebas:

- **Prueba 1:** Utilizando 10 intentos, un umbralpos = 10 y umbralcons = 10, se obtienen los siguientes resultados:

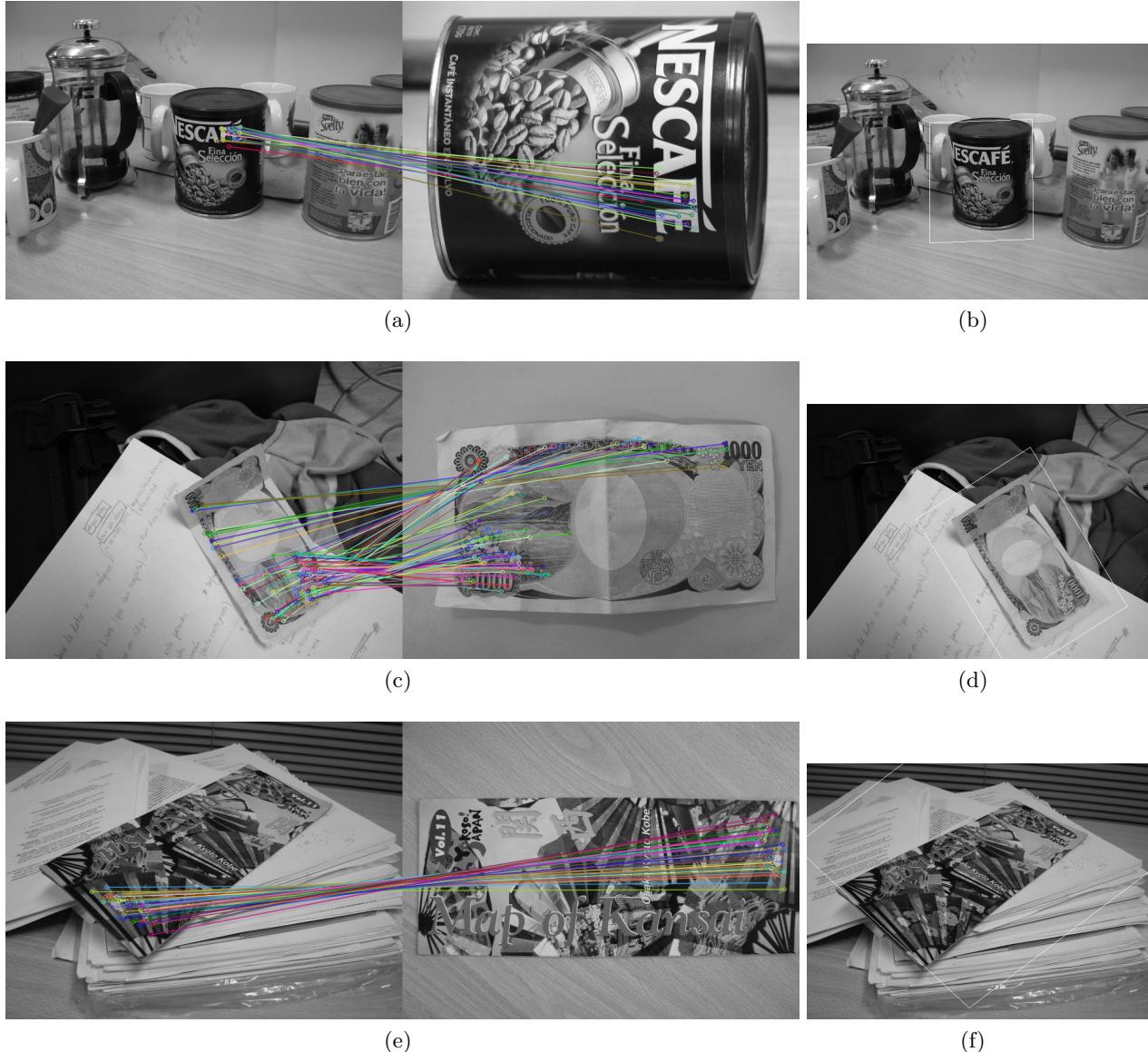


Figura 8: Prueba 1.

Como se puede observar en la figura 8, debido a la poca cantidad de intentos no se observan pocos calces, incluso la imagen del bolso no logró encontrar un calce que cumpliera con las condiciones de los umbrales de posición y consenso, por lo que esta imagen no se muestra en los resultados. Las imágenes del café de Nescafe y del Billete logran ser detectados de forma exitosa, pero se observan que son pocos los matches en las figuras. Por otro lado en

la imagen (f) de esta figura se observa un mal encuadre del objeto, esto se debe a la poca cantidad de calces, estos no logran representar la figura de forma efectiva.

Prueba 2: Utilizando 100 intentos, un umbralpos = 100 y umbralcons = 100, se obtienen los siguientes resultados:

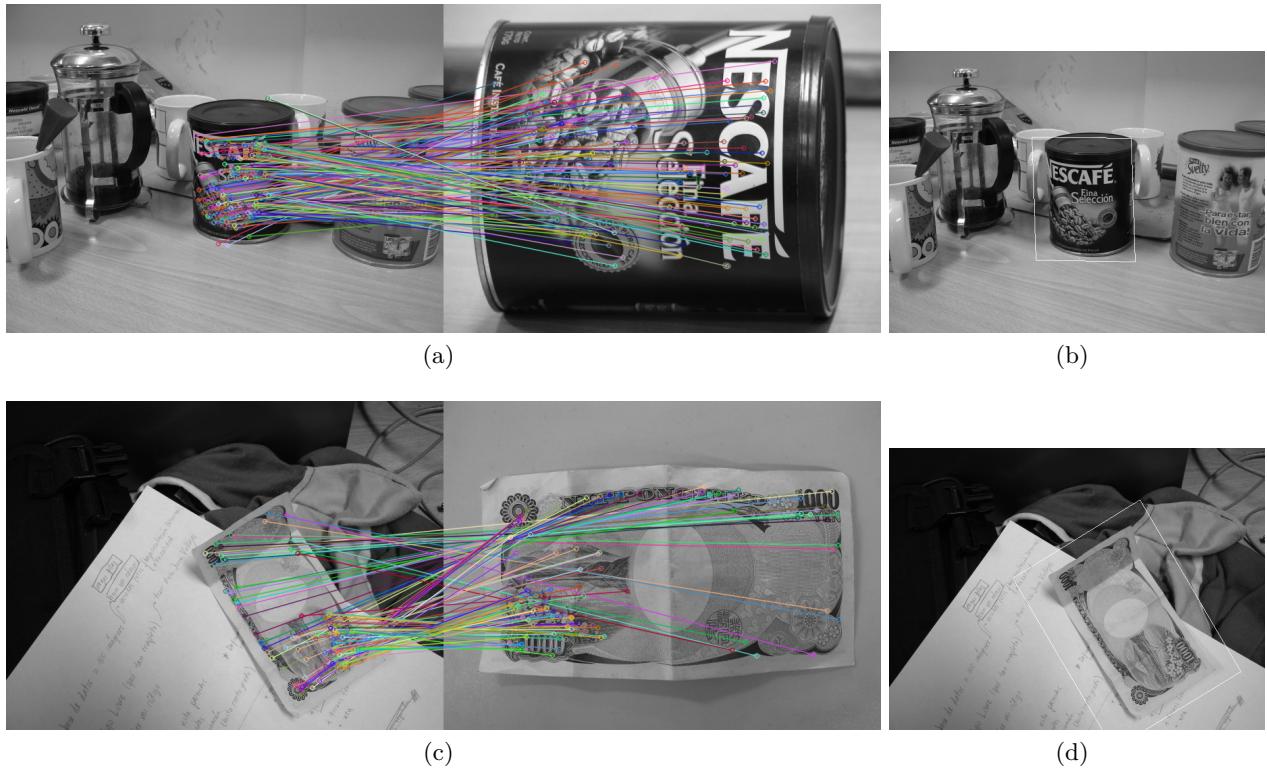


Figura 9: Prueba 2.

Como se puede observar en la figura 9, se aumentó el número de calces en las figuras, esto debido al aumento de los umbrales, a pesar del aumento de los umbrales, los calces siguen estando dentro de las figuras por lo que las figuras logran ser reconocidas de una mejor manera que en el item 3 (el cuadro está más uniforme en las figuras).

- **Prueba 3:** Utilizando 100 intentos, un umbralpos = 5 y umbralcons = 50, se obtienen los siguientes resultados:

Como se puede observar en la figura 10, se aumentó el número de calces en las figuras a pesar de disminuir el umbral de posición, esto entrega puntos más precisos lo cual permite obtener una buena detección del objeto.

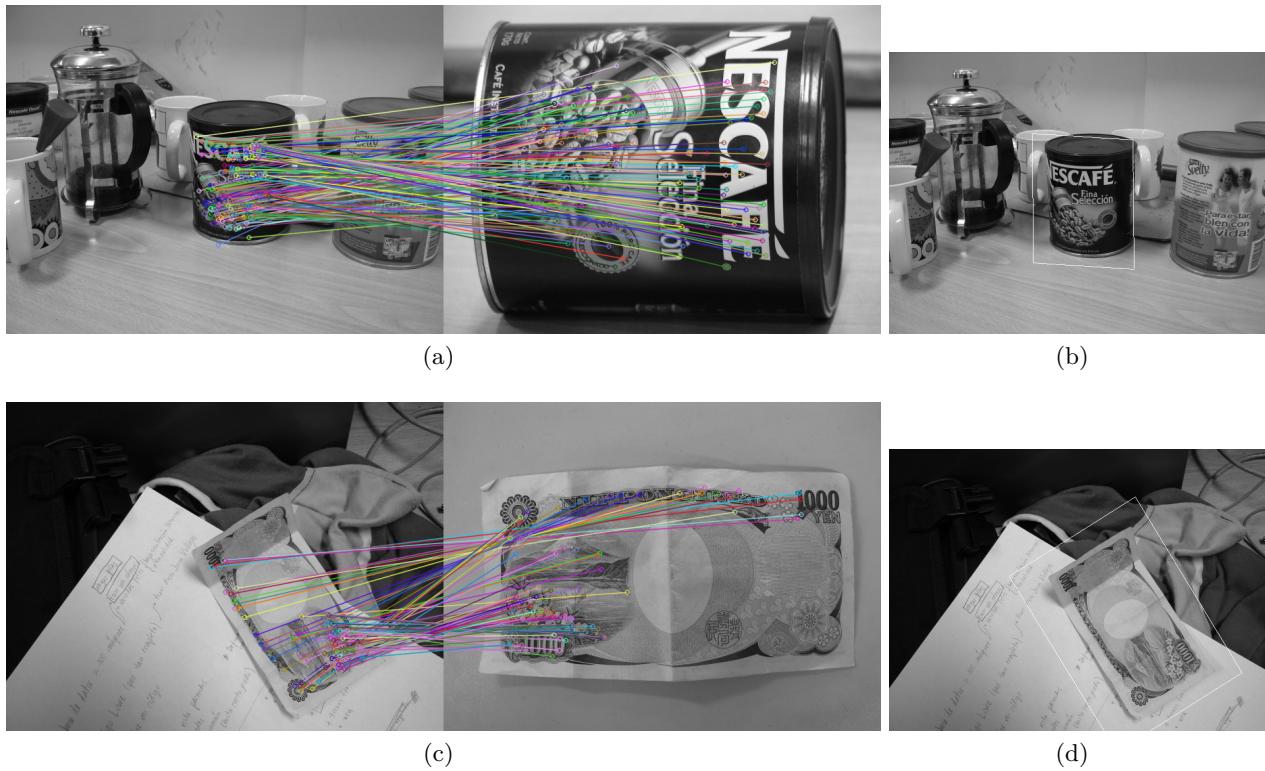


Figura 10: Prueba 3.

3. Conclusión

Luego de desarrollar esta tarea se logró implementar un detector que calcula los puntos de interés mediante el método de Harris e implementar un detector de objetos usando puntos de interés y descriptores SIFT, usando RANSAC, de forma exitosa cumpliéndose así los objetivos de esta tarea. Además se observaron los resultados en cuatro pares de imágenes de pruebas las cuales permitieron comprender de forma práctica el efecto de los detectores de puntos y de objetos, los efectos de variar los parámetros y los resultados para las distintas condiciones. Por otro lado, como se esperaba estos detectores respondieron de forma robusta ante las rotaciones y/o traslaciones, lo cual es una de las ventajas de usar estos métodos, pues en general no se tienen fotos ideales para identificar objetos.

Fue posible poner en práctica los conceptos y técnicas vistas en clases, programarlas desde cero en Python e incluso programar con Cython lo cual permitió mejorar los tiempos de ejecución del programa, la utilización de Cython es más importante de lo que uno piensa pues el procesamiento de imágenes puede tomar mucho tiempo en caso de programarse solo con Python (convirtiendo poco eficiente las implementaciones).

4. Anexos

```
1 # -*- coding: utf-8 -*-
2 """tarea_2_JZ.ipynb
3
4 Automatically generated by Colaboratory.
5
6 Original file is located at
7 https://colab.research.google.com/drive/1yfzyS4LhX5CbJ7OSJdZGTIQnAypB8pqp
8 """
9
10 from google.colab import files
11 #uploaded = files.upload()
12
13 ls
14
15 !pip install ipython-autotime
16
17 # Commented out IPython magic to ensure Python compatibility.
18 # %load_ext autotime
19
20 !pip install -U opencv-python
21
22 """## Parte 1: Cálculo de puntos de interés Harris"""
23
24 # Commented out IPython magic to ensure Python compatibility.
25 # %load_ext Cython
26
27 # Commented out IPython magic to ensure Python compatibility.
28 # %%cython
29 # import cython
30 # import numpy as np
31 # cimport numpy as np
32 #
33 # cpdef np.ndarray[np.float32_t, ndim=2] gradx(np.ndarray[np.float32_t, ndim=2] input):
34 #     # POR HACER: calcular el gradiente en x
35 #     cdef int rows,cols, i,j
36 #
37 #     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.shape[1]], dtype =
38 #     ↪ np.float32)
39 #
40 #     # tamano de la imagen
41 #     rows = input.shape[0]
42 #     cols = input.shape[1]
43 #
44 #     for i in range(rows):
45 #         for j in range(2,cols-2):
46 #             output[i][j-2] = -1*input[i][j-2]+0*input[i][j-1]+1*input[i][j]
47 #
48 #     return output
```

```
48
49 # Commented out IPython magic to ensure Python compatibility.
50 # %cython
51 # import cython
52 # import numpy as np
53 # cimport numpy as np
54 #
55 # cpdef np.ndarray[np.float32_t, ndim=2] grady(np.ndarray[np.float32_t, ndim=2] input):
56 #     # POR HACER: Calcular el gradiente en y
57 #     cdef int rows,cols, i,j
58 #     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.shape[1]], dtype =
59 #             np.float32)
60 #
61 #     # tamano de la imagen
62 #     rows = input.shape[0]
63 #     cols = input.shape[1]
64 #
65 #     for j in range(cols):
66 #         for i in range(2,rows-2):
67 #             output[i-2][j] = -1*input[i-2][j]+0*input[i-1][j]+1*input[i][j]
68 #
69 #     return output
70
71 # Commented out IPython magic to ensure Python compatibility.
72 # %cython
73 # import cython
74 # import numpy as np
75 # cimport numpy as np
76 #
77 # cpdef np.ndarray[np.float32_t, ndim=2] product(np.ndarray[np.float32_t, ndim=2] input1, np.
78 #         ndarray[np.float32_t, ndim=2] input2):
79 #     # POR HACER: generar una matriz que contenga el producto entre input1 e input2, pixel a
80 #             pixel
81 #     cdef int rows,cols, i,j
82 #
83 #     # tamano de la imagen
84 #     rows = input1.shape[0]
85 #     cols = input1.shape[1]
86 #
87 #     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input1.shape[0], input1.shape[1]], dtype =
88 #             np.float32)
89 #
90 #     for i in range(rows):
91 #         for j in range(cols):
92 #             output[i][j] = input1[i][j]*input2[i][j]
93 #
94 #     return output
95
96 # Commented out IPython magic to ensure Python compatibility.
97 # %cython
98 # import cython
99 # import numpy as np
100 # cimport numpy as np
```

```

95 #
96 # cpdef np.ndarray[np.float32_t, ndim=2] harris(np.ndarray[np.float32_t, ndim=2] mxx, np.ndarray
97 #   ↪ [np.float32_t, ndim=2] mxy, np.ndarray[np.float32_t, ndim=2] myy):
98 #   # POR HACER: Calcular el filtro de Harris a partir de (mxx, mxy, myy)
99 #   cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([mxx.shape[0], mxx.shape[1]], dtype =
100 #   ↪ np.float32)
101 #
102 #   #recorremos pixel a pixel
103 #   for i in range(mxx.shape[0]):
104 #       for j in range(mxx.shape[1]):
105 #           output[i][j] = mxx[i][j]*myy[i][j] - mxy[i][j]**(2) - 0.04*(mxx[i][j] + myy[i][j])**2
106 #   return output
107 #
108 # Commented out IPython magic to ensure Python compatibility.
109 # %cython
110 # import cython
111 # import numpy as np
112 # cimport numpy as np
113 #
114 # cpdef np.ndarray[np.float32_t, ndim=2] getMaxima(np.ndarray[np.float32_t, ndim=2] h, float val)
115 #   ↪ :
116 #   # POR HACER: Crear una imagen, inicialmente llena con ceros.
117 #   # Luego, hacer 1 los valores en los cuales se cumplen dos condiciones:
118 #   #   h[r,c] es un maximo local respecto a sus 8 vecinos
119 #   #   h[r,c] supera el valor val
120 #   cdef int rows,cols, r,c
121 #   # tamano de la imagen
122 #   rows = h.shape[0]
123 #   cols = h.shape[1]
124 #   cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([h.shape[0], h.shape[1]], dtype = np.
125 #   ↪ float32)
126 #   for r in range(1,rows-1):
127 #       for c in range(1,cols-1):
128 #           if h[r,c]>val and h[r,c]>=max([h[r-1,c-1],h[r-1,c],h[r-1,c+1] ,h[r,c-1] ,h[r,c+1] ,h[r+1,c-1] ,h[r+1,
129 #           ↪ c],h[r+1,c+1]]):
130 #               output[r,c] = 1
131 #
132 #   return output
133 #
134 # Esta funcion genera keypoints a partir de una imagen de entrada
135 # La imagen de entrada es la salida del filtro de Harris
136 # No es necesario modificar esta funcion
137 def getKeyPoints(input):
138     output = []
139     input_rows = input.shape[0]
140     input_cols = input.shape[1]
141     for r in range(input_rows):
142         for c in range(input_cols):
143             if input[r,c] > 0:
144                 kp = cv2.KeyPoint()
145                 kp.pt = (c,r)

```

```
141     kp.size = 10
142     kp.angle = 0
143     output.append(kp)
144 return output
145
146 import cv2
147 def harrisDetector(input, val):
148     input = np.float32( input )
149     # Por hacer: calcular el filtro de Harris
150     # Hay que realizar los siguientes pasos:
151     # 1) Suavizar la imagen de entrada con cv.GaussianBlur( )
152     suav = cv2.GaussianBlur(input,(5,5),3)
153     # 2) Calcular gradientes imx e imy (usando funciones de cython gradx y grady definidas arriba)
154     imx = gradx(suav)
155     imy = grady(suav)
156     # 3) Calcular momentos usando la funcion product( )
157     #   imxx = imx*imx (pixel a pixel)
158     imxx = product(imx,imx)
159     #   imxy = imx*imy (pixel a pixel)
160     imxy = product(imx,imy)
161     #   imyy = imy*imy (pixel a pixel)
162     imyy = product(imy,imy)
163     # 4) Suavizar momentos imxx, imxy, imyy con cv.GaussianBlur( )
164     simxx = cv2.GaussianBlur(imxx,(5,5),1.5)
165     simxy = cv2.GaussianBlur(imxy,(5,5),1.5)
166     simyy = cv2.GaussianBlur(imyy,(5,5),1.5)
167     # 5) Aplicar el filtro de Harris (usando funcion de Cython harris definida arriba)
168     output = harris(simxx,simxy,simyy)
169     # 6) Encontrar puntos maximos usando getMaxima( )
170     maximos = getMaxima(output,val)
171     # 7) Generar el listado de puntos usando getKeyPoints( )
172     points = getKeyPoints(maximos)
173     #points = getKeyPoints(output)
174     # 8) Devolver los puntos e interes y la imagen filtrada
175
176 return points, output
177
178 def do_rotate(img, angle):
179     h = img.shape[0]
180     w = img.shape[1]
181     cx = w // 2
182     cy = h // 2
183     m = cv2.getRotationMatrix2D((cx, cy), -angle, 1.0)
184     cosa = np.cos(angle * np.pi / 180.0)
185     sina = np.sin(angle * np.pi / 180.0)
186     nw = int((h * sina) + (w * cosa))
187     nh = int((h * cosa) + (w * sina))
188     m[0,2] += (nw / 2) - cx
189     m[1,2] += (nh / 2) - cy
190     return cv2.warpAffine(img, m, (nw, nh))
191
```

```
192 import numpy as np
193 import cv2
194 from google.colab.patches import cv2_imshow
195
196 img1 = cv2.imread('uch010a.jpg',0)
197 img2 = do_rotate(img1, 30)
198
199
200 kp1, h1 = harrisDetector(img1, 1e5)
201 kp2, h2 = harrisDetector(img2, 1e5)
202
203 res1 = cv2.drawKeypoints(img1, kp1, img1, None, cv2.
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
204 res2 = cv2.drawKeypoints(img2, kp2, img2, None, cv2.
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
205
206
207 print('Resultados de filtro de Harris')
208 cv2_imshow(h1 * 500 / np.max(np.max(h1)))
209 cv2_imshow(h2 * 500 / np.max(np.max(h2)))
210 print('Puntos de interes')
211 cv2_imshow(res1)
212 cv2_imshow(res2)
213
214 img1 = cv2.imread('uch098a.jpg',0)
215 img2 = do_rotate(img1, 30)
216
217
218 kp1, h1 = harrisDetector(img1, 1e5)
219 kp2, h2 = harrisDetector(img2, 1e5)
220
221 res1 = cv2.drawKeypoints(img1, kp1, img1, None, cv2.
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
222 res2 = cv2.drawKeypoints(img2, kp2, img2, None, cv2.
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
223
224
225 print('Resultados de filtro de Harris')
226 cv2_imshow(h1 * 500 / np.max(np.max(h1)))
227 cv2_imshow(h2 * 500 / np.max(np.max(h2)))
228 print('Puntos de interes')
229 cv2_imshow(res1)
230 cv2_imshow(res2)
231
232 img1 = cv2.imread('uch020a.jpg',0)
233 img2 = do_rotate(img1, 30)
234
235
236 kp1, h1 = harrisDetector(img1, 3.5e4)
237 kp2, h2 = harrisDetector(img2, 3.5e4)
238
```

```
239 res1 = cv2.drawKeypoints(img1, kp1, img1, None, cv2.  
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)  
240 res2 = cv2.drawKeypoints(img2, kp2, img2, None, cv2.  
    ↪ DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)  
241  
242  
243 print('Resultados de filtro de Harris')  
244 cv2_imshow(h1 * 500 / np.max(np.max(h1)))  
245 cv2_imshow(h2 * 500 / np.max(np.max(h2)))  
246 print('Puntos de interes')  
247 cv2_imshow(res1)  
248 cv2_imshow(res2)  
249  
250 """## Parte 2: Reconocimiento de objetos particulares"""  
251  
252 import math  
253 import numpy as np  
254  
255 def genTransform(match, keypoints1, keypoints2):  
256     # Calcular transformacion de semejanza (e,theta,tx,ty) a partir de un calce "match".  
257     #La idea es que los puntos de train son de referencia y los de query son los de prueba  
258     trainIdx = match.trainIdx  
259     queryIdx = match.queryIdx  
260  
261     # Rescatamos los valores a partir de los atributos de la clase  
262     # Puntos de referencia  
263     (xt, yt) = keypoints1[trainIdx].pt  
264     #puntos de prueba  
265     (xr, yr) = keypoints2[queryIdx].pt  
266  
267     e = keypoints2[queryIdx].size/keypoints1[trainIdx].size  
268     theta = keypoints2[queryIdx].angle-keypoints1[trainIdx].angle  
269     tx = xr-e*(xt*np.cos(theta)-yt*np.sin(theta))  
270     ty = yr-e*(xt*np.sin(theta)+yt*np.cos(theta))  
271     return (e, theta, tx, ty)  
272  
273 def computeConsensus(matches, keypoints1, keypoints2, e, theta, tx, ty, umbralpos):  
274     # Calcular el número de matches compatibles con la transformación (e, theta, tx, ty) indicada  
275     # Debe devolver el número de matches compatibles, y una lista con los matches compatibles  
276  
277     matches_c = []  
278     for match in matches:  
279         trainIdx = match.trainIdx  
280         queryIdx = match.queryIdx  
281         (xt, yt) = keypoints1[trainIdx].pt  
282         (xr, yr) = keypoints2[queryIdx].pt  
283  
284         d1 = e*( np.cos(theta)*xt - np.sin(theta)*yt ) + tx  
285         d2 = e*( np.sin(theta)*xt + np.cos(theta)*yt ) + ty  
286  
287         d = np.sqrt( (d1-xr)**2 + (d2-yr)**2)
```

```
288     if d < umbralpos:
289         matches_c.append(match)
290
291     # se retorna un conteo de los matches y la lista que los contiene
292     return (len(matches_c), matches_c)
293
294 # Por hacer:
295 # Implementar RANSAC. Se debe elegir un calce al azar y calcular su transformacion de semejanza
#   ↪ (e,theta,tx,ty)
296 # usando la funcion genTransform( ) definida arriba
297 # Luego se debe evaluar el consenso de la transformacion usando computeConsensus()
298 # Esto se debe repetir varias veces
299 # Una vez hechos todos los intentos, se debe analizar la hipotesis que tuvo el mayor consenso
300 # Si dicho consenso es mayor a un umbral, se aceptan los calces del consenso como correctos
301 # Los calces considerados correctos se deben guardar en "accepted"
302
303 import random
304
305 def ransac(matches, keypoints1, keypoints2):
306     accepted = []
307
308     #definimos los parametros para luego ir revisndo los resultados
309     intentos = 100
310     umbralpos = 10
311     umbralcons = 50
312     n = len(matches)
313
314     L_nums = []
315     L_accepted = []
316
317     for _ in range(intentos):
318         #elegimos un indice al azar con random.randint
319         i = random.randint(0,n-1)
320         match = matches[i]
321         (e, theta, tx, ty) = genTransform(match, keypoints1, keypoints2)
322
323         (num,accepted) = computeConsensus(matches, keypoints1, keypoints2, e, theta, tx, ty,
#           ↪ umbralpos)
324         #si el numero de matches es mayor al umbral, se retorna la lista de calces considerados correctos
325         if num > umbralcons:
326             L_nums.append(num)
327             L_accepted.append(accepted)
328
329
330     #encontramos el indice de la transformacion de semejanza con mayor consenso y los calces
#       ↪ compatibles
331     imax = L_nums.index(max(L_nums))
332
333     #retornamos la lista de calces de mayor concenso
334     return L_accepted[imax]
335
```

```
336 def calcAfin(matches, keypoints1, keypoints2):  
337     # Por hacer: calcular la transformacion afin mediante minimos cuadrados a partir de "matches"  
338     #  $x^* = (A'A)^{-1}A'b$   
339     a = []  
340     b = []  
341  
342     for match in matches:  
343         #print("aqui")  
344         trainIdx = match.trainIdx  
345         queryIdx = match.queryIdx  
346  
347         (xt, yt) = keypoints1[trainIdx].pt  
348         (xr, yr) = keypoints2[queryIdx].pt  
349  
350         a.append([xt, yt, 0, 0, 1, 0])  
351         a.append([0, 0, xt, yt, 0, 1])  
352         b.append(xr)  
353         b.append(yr)  
354  
355     A = np.array(a)  
356  
357     B = np.array(b)  
358     At = np.transpose(A)  
359     x = np.matmul(np.matmul(np.linalg.inv((np.matmul(At,A))),At),B)  
360  
361     return x  
362  
363     # Esta función devuelve la imagen "input1" con un romboide dibujado  
364     # El romboide representa el rectangulo de la imagen "input2" proyectada en la imagen "input1"  
365     # La proyección se realiza a partir de la transformación "transf"  
366  
367 def drawProjAfin(transf, input1, input2):  
368  
369     #le hacemos un reshape pues habian problemas con las dimensiones  
370     transf = np.array([[transf[0],transf[1],transf[4]],[transf[2],transf[3],transf[5]]])  
371  
372     w = input2.shape[1];  
373     h = input2.shape[0];  
374     sq1 = [0,0]  
375     sq2 = [w-1,0]  
376     sq3 = [w-1,h-1]  
377     sq4 = [0,h-1];  
378     p1 = [0,0]  
379     p2 = [0,0]  
380     p3 = [0,0]  
381     p4 = [0,0]  
382     print("Tamano " + str(transf.shape[1]) + " x " + str(transf.shape[0]));  
383     print("Posicion " + str(transf[0,2]) + ", " + str(transf[1,2]));  
384     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]));
```

385

```

386 p1[0] = transf[0,0] * sq1[0] + transf[0,1] * sq1[1] + transf[0,2];
387 p1[1] = transf[1,0] * sq1[0] + transf[1,1] * sq1[1] + transf[1,2];
388 p2[0] = transf[0,0] * sq2[0] + transf[0,1] * sq2[1] + transf[0,2];
389 p2[1] = transf[1,0] * sq2[0] + transf[1,1] * sq2[1] + transf[1,2];
390 p3[0] = transf[0,0] * sq3[0] + transf[0,1] * sq3[1] + transf[0,2];
391 p3[1] = transf[1,0] * sq3[0] + transf[1,1] * sq3[1] + transf[1,2];
392 p4[0] = transf[0,0] * sq4[0] + transf[0,1] * sq4[1] + transf[0,2];
393 p4[1] = transf[1,0] * sq4[0] + transf[1,1] * sq4[1] + transf[1,2];
394
395 p1 = (int(p1[0]), int(p1[1]))
396 p2 = (int(p2[0]), int(p2[1]))
397 p3 = (int(p3[0]), int(p3[1]))
398 p4 = (int(p4[0]), int(p4[1]))
399
400
401 out = np.copy(input1);
402 cv2.line(out, p1, p2, (255,255,255));
403 cv2.line(out, p2, p3, (255,255,255));
404 cv2.line(out, p3, p4, (255,255,255));
405 cv2.line(out, p4, p1, (255,255,255));
406 return out;
407
408 # Esta funcion ya esta lista, no debe ser modificada
409 def filterMatches(matches):
410     # Apply ratio test
411     points1 = []
412     points2 = []
413     good = []
414
415     for m,n in matches:
416         if m.distance < 0.75*n.distance: # 0.75
417             good.append(m)
418             points1.append(kp1[m.queryIdx].pt)
419             points2.append(kp2[m.trainIdx].pt)
420     return np.array(points1), np.array(points2), good
421
422 """# Probamos con 4 imagenes"""
423
424 import numpy as np
425 import cv2
426 from matplotlib import pyplot as plt
427 from google.colab.patches import cv2_imshow
428
429 img2 = cv2.imread('uch010a.jpg',0)
430 img1 = cv2.imread('uch010b.jpg',0)
431
432 # Initiate SIFT detector
433 sift = cv2.SIFT_create()
434
435 # find the keypoints and descriptors with SIFT
436 kp1, des1 = sift.detectAndCompute(img1,None)

```

```
437 kp2, des2 = sift.detectAndCompute(img2,None)
438
439 ##### BFMatcher with default params
440 bf = cv2.BFMatcher()
441 matches = bf.knnMatch(des1,des2, k=2)
442 points1, points2, good = filterMatches(matches)
443
444 img_match = cv2.drawMatches(img1,kp1,img2,kp2,good, None, flags=cv2.
445     ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
446
447 cv2_imshow(img_match)
448
449 # Por hacer:
450 # 1) Obtener los calces aceptados a partir de la función ransac(good, kp2, kp1)
451 accepted = ransac(good,kp2,kp1)
452 # 2) Dibujar los calces aceptados
453 img_match = cv2.drawMatches(img1,kp1,img2,kp2,accepted, None, flags=cv2.
454     ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
455 cv2_imshow(img_match)
456 # 3) Calcular la transformacion afin
457 Afin = calcAfin(accepted, kp2, kp1)
458 # 4) Dibujar la imagen con el romboide superpuesto
459 cv2_imshow(drawProjAfin(Afin,img1,img2))
460
461
462 import numpy as np
463 import cv2
464 from matplotlib import pyplot as plt
465 from google.colab.patches import cv2_imshow
466
467 img2 = cv2.imread('uch001a.jpg',0)
468 img1 = cv2.imread('uch001b.jpg',0)
469
470 # Initiate SIFT detector
471 sift = cv2.SIFT_create()
472
473 # find the keypoints and descriptors with SIFT
474 kp1, des1 = sift.detectAndCompute(img1,None)
475 kp2, des2 = sift.detectAndCompute(img2,None)
476
477 ##### BFMatcher with default params
478 bf = cv2.BFMatcher()
479 matches = bf.knnMatch(des1,des2, k=2)
480 points1, points2, good = filterMatches(matches)
481
482 img_match = cv2.drawMatches(img1,kp1,img2,kp2,good, None, flags=cv2.
483     ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
484
```

```
485 cv2_imshow(img_match)
486
487 # Por hacer:
488 # 1) Obtener los calces aceptados a partir de la función ransac(good, kp2, kp1)
489 accepted = ransac(good,kp2,kp1)
490 # 2) Dibujar los calces aceptados
491 img_match = cv2.drawMatches(img1,kp1,img2,kp2,accepted, None, flags=cv2.
    ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
492 cv2_imshow(img_match)
493 # 3) Calcular la transformacion afin
494 Afin = calcAfin(accepted, kp2, kp1)
495 # 4) Dibujar la imagen con el romboide superpuesto
496 cv2_imshow(drawProjAfin(Afin,img1,img2))
497
498 import numpy as np
499 import cv2
500 from matplotlib import pyplot as plt
501 from google.colab.patches import cv2_imshow
502
503 img2 = cv2.imread('uch020a.jpg',0)
504 img1 = cv2.imread('uch020b.jpg',0)
505
506 # Initiate SIFT detector
507 sift = cv2.SIFT_create()
508
509 # find the keypoints and descriptors with SIFT
510 kp1, des1 = sift.detectAndCompute(img1,None)
511 kp2, des2 = sift.detectAndCompute(img2,None)
512
513 ##### BFMatcher with default params
514 bf = cv2.BFMatcher()
515 matches = bf.knnMatch(des1,des2, k=2)
516 points1, points2, good = filterMatches(matches)
517
518 img_match = cv2.drawMatches(img1,kp1,img2,kp2,good, None, flags=cv2.
    ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
519
520
521 cv2_imshow(img_match)
522
523 # Por hacer:
524 # 1) Obtener los calces aceptados a partir de la función ransac(good, kp2, kp1)
525 accepted = ransac(good,kp2,kp1)
526 # 2) Dibujar los calces aceptados
527 img_match = cv2.drawMatches(img1,kp1,img2,kp2,accepted, None, flags=cv2.
    ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
528 cv2_imshow(img_match)
529 # 3) Calcular la transformacion afin
530 Afin = calcAfin(accepted, kp2, kp1)
531 # 4) Dibujar la imagen con el romboide superpuesto
532 cv2_imshow(drawProjAfin(Afin,img1,img2))
```

```
533
534 import numpy as np
535 import cv2
536 from matplotlib import pyplot as plt
537 from google.colab.patches import cv2_imshow
538
539 img2 = cv2.imread('uch098a.jpg',0)
540 img1 = cv2.imread('uch098b.jpg',0)
541
542 # Initiate SIFT detector
543 sift = cv2.SIFT_create()
544
545 # find the keypoints and descriptors with SIFT
546 kp1, des1 = sift.detectAndCompute(img1,None)
547 kp2, des2 = sift.detectAndCompute(img2,None)
548
549 ##### BFMatcher with default params
550 bf = cv2.BFMatcher()
551 matches = bf.knnMatch(des1,des2, k=2)
552 points1, points2, good = filterMatches(matches)
553
554 img_match = cv2.drawMatches(img1,kp1,img2,kp2,good, None, flags=cv2.
    ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
555
556
557 cv2_imshow(img_match)
558
559 # Por hacer:
560 # 1) Obtener los calces aceptados a partir de la función ransac(good, kp2, kp1)
561 accepted = ransac(good,kp2,kp1)
562 # 2) Dibujar los calces aceptados
563 img_match = cv2.drawMatches(img1,kp1,img2,kp2,accepted, None, flags=cv2.
    ↪ DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
564 cv2_imshow(img_match)
565 # 3) Calcular la transformacion afin
566 Afin = calcAfin(accepted, kp2, kp1)
567 # 4) Dibujar la imagen con el romboide superpuesto
568 cv2_imshow(drawProjAfin(Afin,img1,img2))
```

Referencias

- [1] Giuseppe Pio Cannata. Image derivative. Towards Data Science. Disponible en: <https://towardsdatascience.com/image-derivative-8a07a4118550>