TP3 Robótica Móvil

Sacha Varela
Gerónimo González Marino

31 de octubre de 2025

Introducción

A continuación se presenta la resolución de cada ejercicio con los resultados finales obtenidos en forma de matrices e imágenes. Para el detalle se redirige al repositorio Git [1]. El dataset utilizado son los correspondientes a los escenarios V102 y M101 del conjunto EuroCMAV (es posible utilizar los demás escenarios del conjunto de datos si se actualizan correctamente las rutas de los archivos leídos).

La ejecución de los códigos provistos se puede realizar desde vscode cargando el ambiente de trabajo "Ejs", el cual posee un launch.json que permite ejecutar cada script con los argumentos correspondientes y distintas configuraciones directamente desde el IDE, utilizando la herramienta "Debug using launch.json". Otra opción es correr los scripts desde la terminal directamente con python3 y los argumentos deseados. También se recomienda ver los comentarios al final de cada script para comprender que acciones se deben realizar para su correcta ejecución.

Ejercicio 1

- Realizar una calibración de la cámara estéreo del robot. Se deben proveer los parámetros intrínsecos y extrínsecos.

Se utiliza el rosbag de calibración con checkerboard porque este es el tipo de tablero que admite el paquete de ros camera_calibration. El archivo está en el formato de bag de ROS1 (`.bag`) por lo que es necesario transformarlo al formato de ROS2 utilizando la librería `rosbags`. Para ello se utiliza el siguiente comando.

```
rosbags-convert --src cam_checkerboard.bag --dst cam_checkerboard_ros2
```

Dentro de los archivos asociados, busco el archivo `checkerboard_7x6.yaml` que contiene la descripción del patrón de calibración utilizado. Dicha información se encuentra documentada en [2], donde se presenta en el siguiente formato:

Listing 1: Configuración del patrón de calibración

```
target_type: 'checkerboard' #gridtype
targetCols: 6               #number of internal chessboard corners
```

```
targetRows: 7                #number of internal chessboard corners
rowSpacingMeters: 0.06       #size of one chessboard square [m]
colSpacingMeters: 0.06       #size of one chessboard square [m]
```

Para la obtención de los parámetros intrínsecos y extrínsecos de la cámara, se corrió en una consola el rosbag con el siguiente comando

```
ros2 bag play cam_checkerboard_ros2.db3
```

mientras en otra consola se corría el calibrador, con los parámetros del patrón de calibración, mediante el comando

```
ros2 run camera_calibration cameracalibrator \
--approximate=0.1 \
--size 6x7 \
--square 0.06 \
--no-service-check \
--ros-args \
-r left:=/cam0/image_raw \
-r right:=/cam1/image_raw \
-p camera:=stereo
```

Los parámetros de calibración obtenidos se encuentran en [3], bajo los nombres de left.yaml y right.yaml, cada archivo representa la calibración y mapa de rectificación de la cámara izquierda y derecha del arreglo stero respectivamente.

Si observamos las matrices de proyección obtenidas, para la cámara izquierda:

$$\begin{bmatrix} 442{,}17199 & 0. & 356{,}19366 & 0. \\ 0. & 442{,}17199 & 253{,}89066 & 0. \\ 0. & 0. & 1. & 0. \end{bmatrix} \tag{1}$$

Y de la derecha:

$$\begin{bmatrix} 442{,}17199 & 0. & 356{,}19366 & -48{,}47631 \\ 0. & 442{,}17199 & 253{,}89066 & 0. \\ 0. & 0. & 1. & 0. \end{bmatrix} \tag{2}$$

Donde se puede observar que el cero de la cámara se toma en la cámara izquierda, y la derecha se encuentra desplazada 10 cm,

$$d = \frac{442{,}17}{-48{,}47} = -0{,}1[m] \tag{3}$$

Ejercicio 2

Desarrollar un programa que lea un par de imágenes estéreo cualquiera y realice los siguientes pasos:

Inciso a)

-Rectificar las imágenes utilizando los parámetros intrínsecos y extrínsecos de la cámara stereo.

Debido a que los bag con las grabaciones de los frames de cada escenario no tienen los tópicos /cam/Camera_info, correspondientes a los datos de calibración de cada cámara, se realiza un script en python para utilizar las funciones de la biblioteca OpenCV, en particular la función `cv2.remap`.

Como es posible que los timestamp del header de los mensajes de cada cámara no sean exactamente iguales, primero es necesario utilizar la función ApproximateTimeSynchronizer para garantizar que llegue un frame de cada cámara al mismo tiempo y que estos sean correspondientes entre si.

Luego se cargan los parámetros de calibración desde los archivos .yaml y se generan los mapas de rectificación con la función `cv2.initUndistortRectifyMap`. Por último, se publica un tópico por cada imagen rectificada y se utiliza rviz para visualizarlos.

Para el código ver inciso-a.py

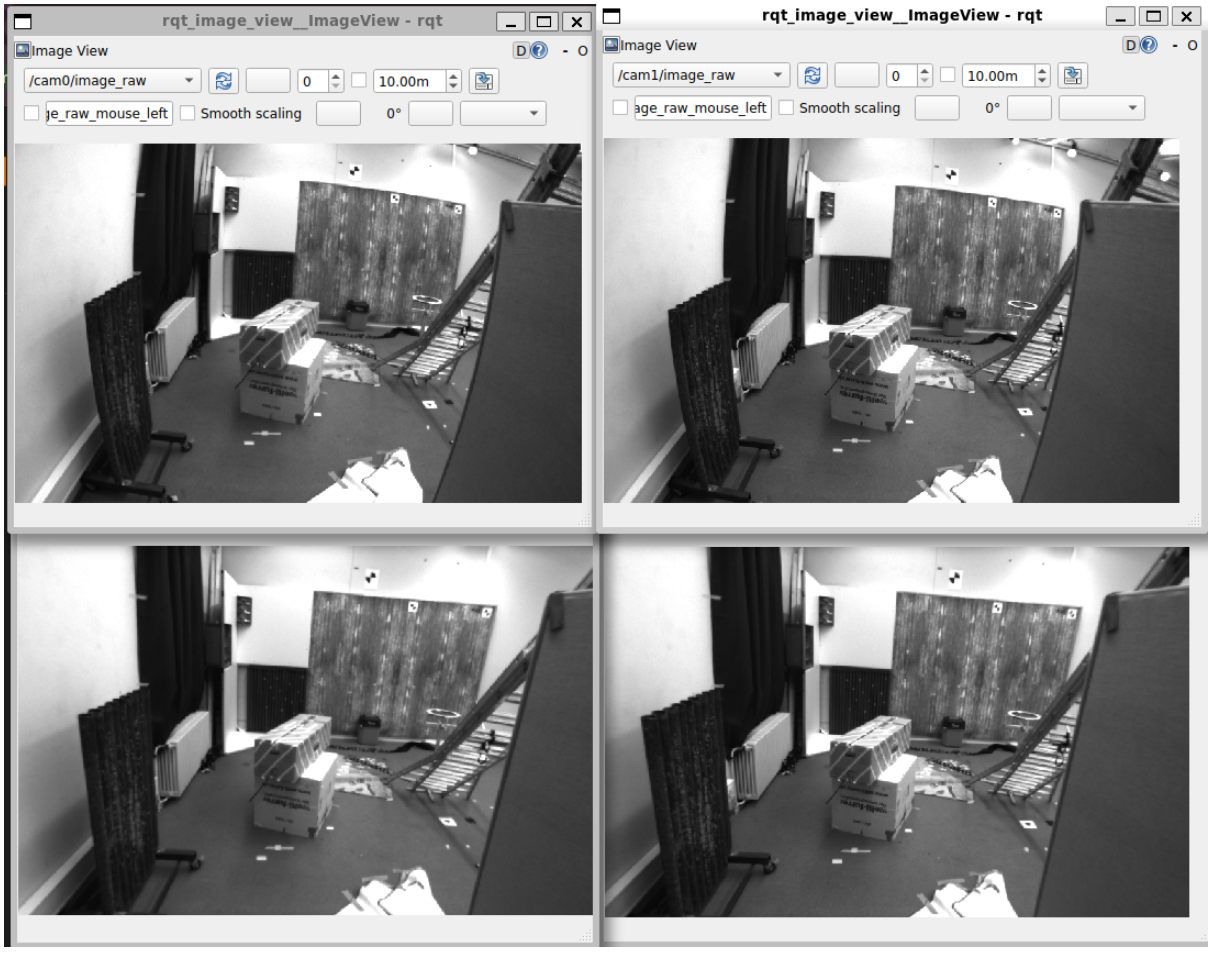

Figura 1: resultado inciso a

En la imagen se observan, arriba ambas cámaras sin rectificar, debajo, ambas cámaras rectificadas. si se presta a las lineas verticales de algunos de los muebles que aparecen en

el video la desdistorisón realizada se detecta fácilmente. Con las imágenes presentadas la distorsión del lente parecería ser de tipo Barrel.

incisos b y c)

- Extraer Features Visuales: Keypoints y descriptores. Seleccionar un detector de keypoints (FAST, ORB, SIFT, SURF, GFTT, BRISK, etc.) y un descriptor (BRIEF, ORB, BRISK, etc.), y extraer features en ambas imágenes. Capturar una imagen izquierda y derecha con los features extraídos. Agregar captura al informe.

- Buscar correspondencias visuales. Realizar la búsqueda de correspondencias entre los features de ambas imágenes (matching). Para esto se debe utilizar la función cv::BFMatcher::BFMatcher(). Visualizar todos los matches. Luego, visualizar todos los matches que tengan una distancia de matching menor a 30. Agregar capturas al informe.

Se utilizó el detector y descriptor ORB de la biblioteca OpenCV para detectar los features y luego se realizó un matcheo de fuerza bruta (BFMatcher) entre los features de cada imagen. Por último se filtró, según la distancia de hamming, los matches con una distancia menor a 30.

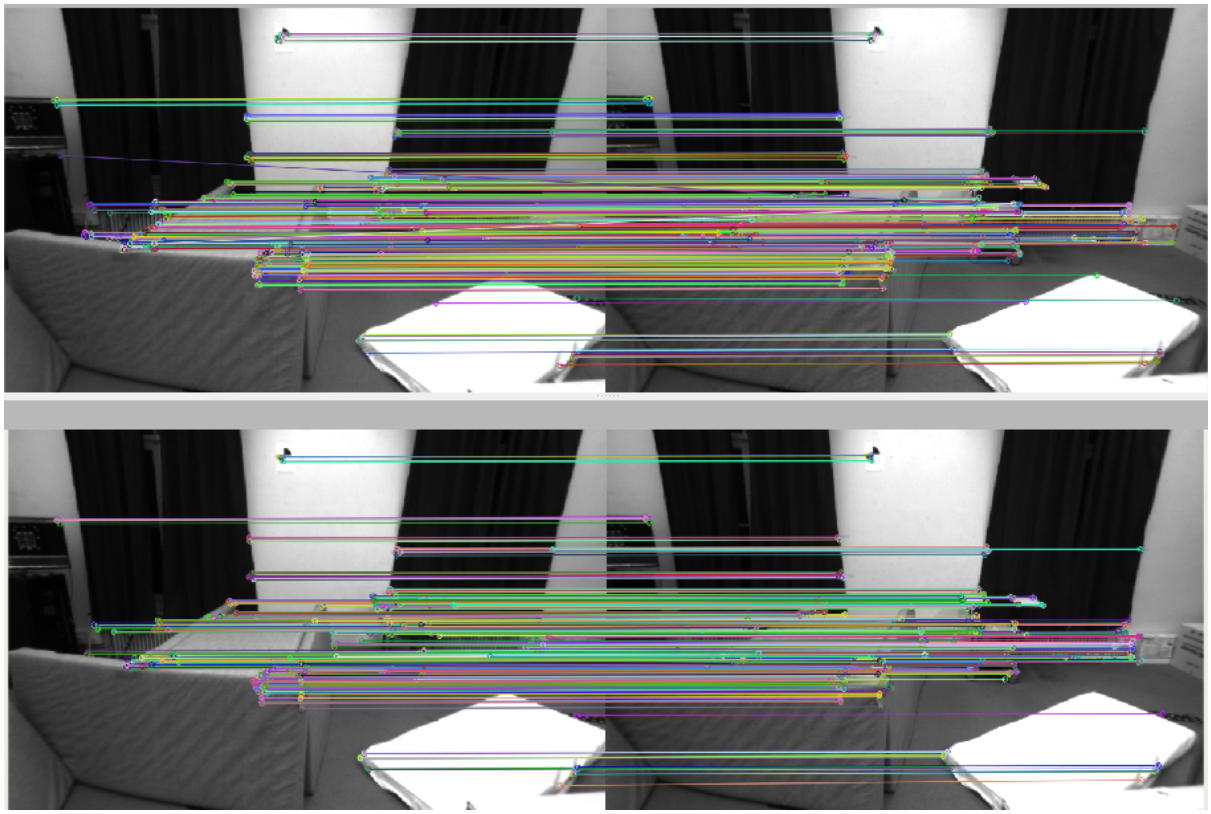

Figura 2: arriba: todos los matches y los features detectados. abajo: solo matches y features con distancia de hamming menor a 30. V102

En la imagen de arriba se puede apreciar cómo la cantidad de matches se reduce una vez aplicado el filtro. Además, se observa que antes del filtro el matcheo por fuerza bruta devuelve algunos resultados que no se encuentran sobre la línea epipolar horizontal, y

siendo que las cámaras están rectificadas esto no es posible. Una vez que se realiza el filtrado se observa menor cantidad de matcheos que no sean horizontales.

En el siguiente escenario, donde las texturas y la iluminación hacen la detección más difícil, podemos ver que una vez realizado el filtrado, la cantidad de matches se reduce drásticamente. También se observa el fenómeno de matches fuera de líneas epipolares.

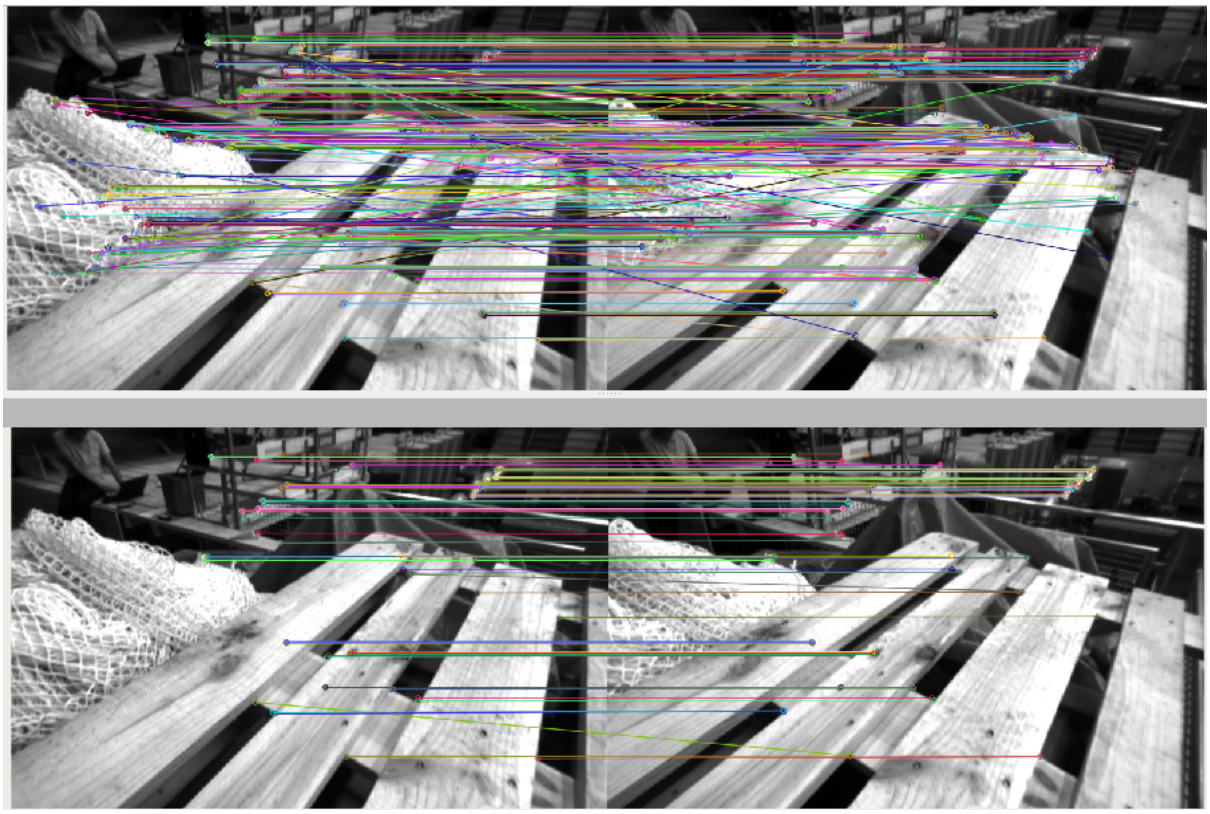

Figura 3: arriba: todos los matches y los features detectados. abajo: solo matches y features con distancia de hamming menor a 30. M101

Inciso d)

- Dadas las correspondencias visuales (matches) obtenidas en el paso anterior, realizar la triangulación de los features extraídos utilizando la función cv::sfm::triangulatePoints(). Para la visualización de la nube de puntos 3D se puede publicar un mensaje de tipo sensor_msgs/PointCloud24 y hacer uso de RViz. agregar capturas al informe.

La función `cv2.triangulatePoints` toma los matches ya filtrados y utiliza las matrices de proyección de cada cámara para calcular la profundidad de cada punto. Luego se publica la nube frame a frame y se visualiza en Rviz. Para visualizar correctamente es necesario definir el sistema de coordenadas fijo, para lo que se ejecuta la siguiente transformación en una terminal secundaria.

```
ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 map stereo_camera
```

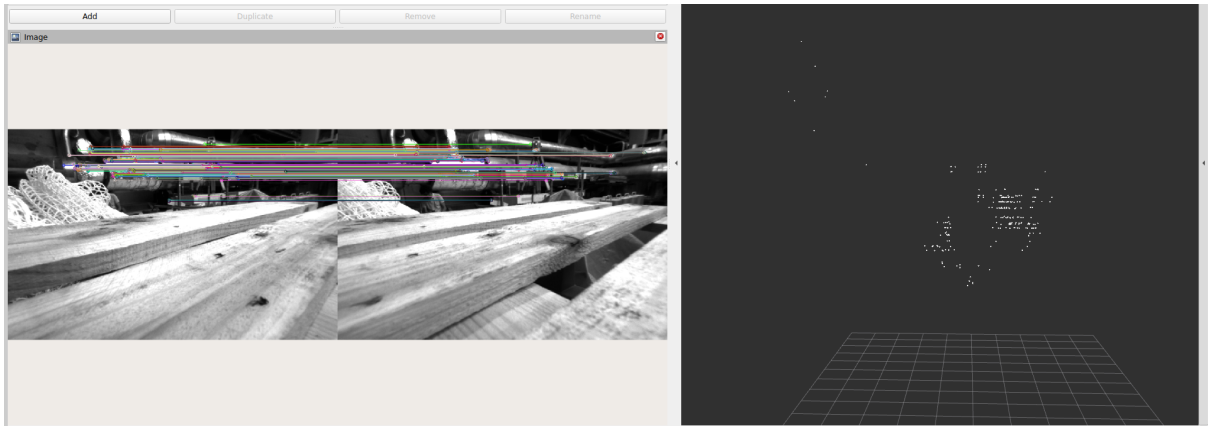

Figura 4: nube de puntos local y matches correspondientes

Al tener pocos matches, la nube de puntos es relativamente pequeña, además, se encuentra publicada en el marco local de la cámara por lo que varía frame a frame. En el ángulo presentado en la imagen de arriba se puede observar la diferencia de profundidad entre los features, producto de la triangulación.

inciso e)

- Aplicar RANSAC (Random sample consensus) para filtrar los matches espúreos y computar la Matriz homográfica que relaciona los puntos. Para esto puede utilizar la función cv::findHomography(). Para verificar el impacto del filtrado, visualizar los matches entre las imágenes nuevamente como en la nube de puntos 3D generada. También, visualizar en la imagen derecha los puntos de la imagen izquierda transformados por la matriz homográfica. Para esto último utilizar la función cv::perspectiveTransform(). Agregar capturas al informe.

Luego del filtrado aplicamos RANSAC para determinar el modelo geométrico que mejor se ajusta a las vistas de ambas cámaras, esto es, la matriz homográfica o la fundamental. Finalmente los puntos que se toman como válidos son solo los que se adecúan al modelo (inliers) y respetan la geometría epipolar.

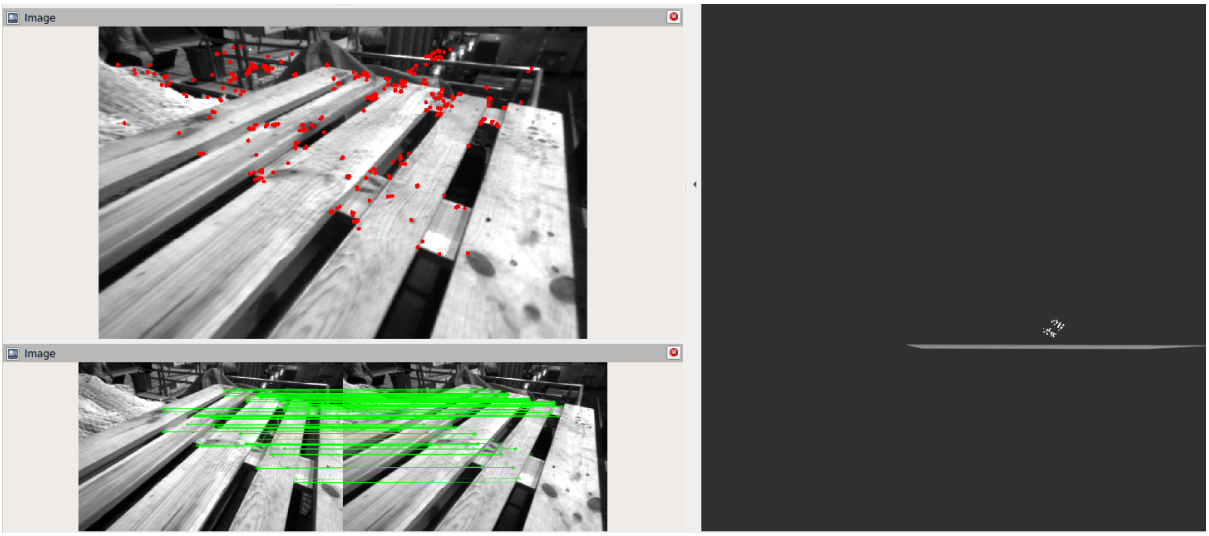

Figura 5: Matches, nube de puntos y proyección la otra imagen de los puntos inliers

En la imagen de arriba podemos observar que la nube de puntos se redujo, probablemente a causa de la eliminación de coincidencias espurias, ya no se observan matches que no cumplan con la geometría epipolar de las cámaras rectificadas y se pueden apreciar los features de la imagen izquierda proyectados en rojo sobre la imagen derecha.

inciso f) (opcional)

- Mapear el entorno utilizando las poses dada por el ground-truth del dataset. Visualizar el mapa reconstruido. Agregar captura al informe.

Como se comentó anteriormente, la nube de puntos se publica sobre el marco local de la cámara, por lo que si no se conoce la pose de la cámara no se podrán dibujar los puntos en su lugar en el espacio (a menos que se recurra a SLAM). En este caso tenemos acceso al conjunto de datos del ground truth de la cámara, por lo que podemos asociar cada nube de puntos correspondientes a un par de frames (L y R) con su posición en el espacio.

Para esto primero se crea un nodo publicador que levanta todos los datos del archivo .csv del ground truth, extrae los parámetros de traslación y rotación (x, y, z, qw, qx, qy, qz) y los publica en un mensaje de tipo POSE en el tópico /ground_truth/pose.

Luego, otro script toma los datos de los frames del bag (los procesa al igual que antes) y los multiplica por la matriz de la pose actual de la cámara (matriz de roto-traslación homogénea) que extrae del tópico /ground_truth/pose.

Para correr este inciso adecuadamente, el flujo de trabajo a seguir es:

- correr el script madre que inicializa los nodos de rectificación, publicadores y suscriptores
- correr el nodo publicador del ground truth.
- correr el bag del dataset con ros2 bag play /ruta/dataset

Es necesario ajustar un poco las frecuencias del publicador y las del bag para que el desfase entre la pose publicada y el par de frames correspondientes sea el menor posible.

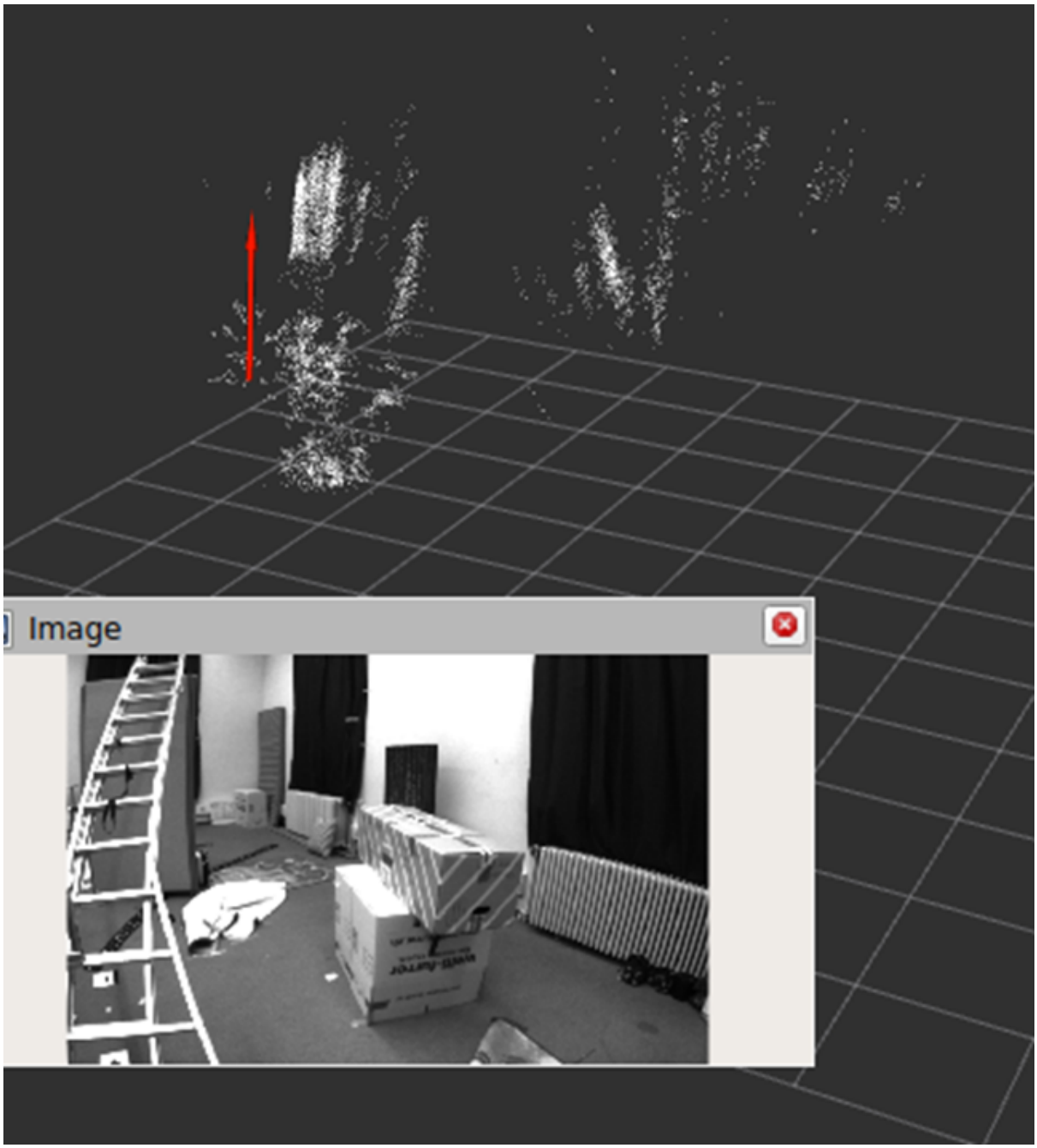

Figura 6: nube global de matches

incisos g y h)

- Computar el mapa de disparidad con las librerías utilizando la función cv::StereoMatcher::compute() Visualizar el mapa de disparidad. Agregar captura al informe.

- Utilizando el mapa de disparidad obtenido en el paso anterior realizar una reconstrucción densa de la escena observada utilizando la función cv::reprojectImageTo3D(). Para

esto debe utilizar la matríz de reproyección Q retornada por la función cv::stereoRectify(). Visualizar la nube de puntos 3D. Agregar captura al informe.

Ya no se utiliza ORB como detector y descriptor de features si no que se utiliza la clase cv2.StereoSGBM_create y la función self.stereo.compute() para computar el mapa de disparidad, la cual ya considera la geometría epipolar de las cámaras rectificadas y compara la intensidad de cada pixel en una misma línea horizontal para calcular disparidad. Luego la función reprojectImageTo3D() genera la nube de puntos.

Para visualizar los resultados se corre el script `inciso-g-h-i.py` con su configuración (launch.json) según el escenario que se desee reproducir y se visualizan los siguientes tópicos:

- /stereo/disparity para mapa de disparidad
- /stereo/dense_pointcloud para nube densa

Para visualizar la nube densa es recomendable comentar la sección de código que realiza un sub-muestreo para disminuir la cantidad de puntos (para máquinas con poca memoria disponible).

```
#Submuestreo para reducir cantidad de puntos
    if len(points) > 20000:
        idx = np.random.choice(len(points), 20000, replace=False)
        points = points[idx]
        colors = colors[idx]
```

A la nube se la gráfica con los colores de cada pixel de la imagen izquierda para que resulte más fácil comprender la imagen generada.

Para esto es necesario configurar la nube de puntos en rviz para que utilice:

- Style: points
- size(m): 0.01
- ColorTransformer: RGBF32 (Solo estará disponible una vez que se haya dado play al bag)

también es necesario publicar la siguiente transformación desde terminal

```
ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 world cam0
```

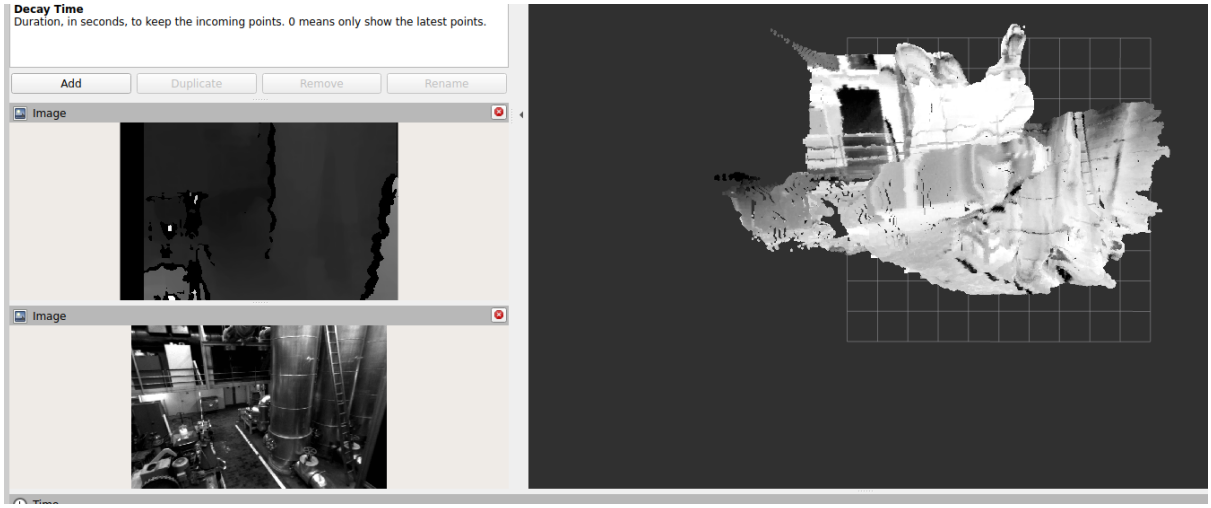

Figura 7: Mapa de disparidad, nube densa en marco loca, imagen izquierda rectificada para referencia

En la imagen se puede apreciar el mapa de disparidad encima de la imagen de la cámara rectificada, donde los colores más oscuros indican mayor lejanía. A la derecha se observa la nube densa (local) en los colores invertidos de los de la cámara (no sabemos por qué, pero así se visualiza mejor). Se puede ver que la imagen que se forma en la nube es la misma que la de la cámara y si se inspecciona la nube también se puede ver su profundidad.

Inciso i) (opcional)

- Mapear el entorno de manera densa utilizando las poses dada por el ground-truth del dataset. Visualizar el mapa reconstruido. Agregar captura al informe.

De la misma manera que en el inciso f se mapeó el entorno utilizando los features, ahora se realiza la misma acción pero publicando la nube densa (en vez de la nube de features del ORB) en cada pose del ground truth. En este caso, para mejorar la sincronización, ya no se corre un nodo que publique la pose en un tópico, sino que se levantan los datos directamente del .csv del ground truth, se desglosa en los valores necesarios (posición y rotación), se extrae el timestamp de cada pose y se define una función (`get_pose_at_time(self, t)`)que lo aproxima al más cercano de la imagen publicada (image_raw, ya que en los demás mensajes se recicla el header). Luego se obtiene la pose del ground truth para ese instante y se utiliza para transformar los puntos del marco local de la cámara al marco global.

Cabe destacar que el costo computacional de esta operación es elevado (por lo menos para nuestras PCs) ya que la nube global acumula puntos de una nube densa local y los publica en cada nuevo mensaje de tipo Point_cloud2.

Por esta razón, el código cuenta con 4 estrategias para reducir el costo computacional.

- Primero, existe un bloque que crea una submuestra aleatoria de los puntos de disparidad según si estos superan un límite establecido por el usuario (hardcodeado en la función, no se pasa como argumento), y utiliza este subconjunto para todas las publicaciones posteriores.

- se implementa una estrategia de throttling que evita procesar todos los frames, sino que establece un tiempo de espera entre frame procesado y el siguiente (hardcodeado en el init de la clase, no se pasa como argumento, la variable es `self.process_interval`.
- La nube densa global solo se publica cada un cierto número de frames (hardcodeado en la definición del callback, no se pasa como argumento) para no saturar la memoria.
- en la línea 201, donde se realiza una máscara sobre la disparidad para filtrar los valores inválidos, es posible definir un valor de disparidad mínimo más alto que el mínimo posible (`disparity.min()`), lo que filtra puntos muy lejanos. Se encontró que entre 25 y 50 se obtiene una buena nube densa global, pero aún con las 4 estrategias el costo computacional es alto.

Debajo se presentan los bloques de código de las 4 estrategias:

```
    #Submuestreo para reducir cantidad de puntos
        if len(points) > 20000:
            idx = np.random.choice(len(points), 20000, replace=False)
            points = points[idx]
            colors = colors[idx]
```

```
    def callback(self, msg_l, msg_r):
        # --- Control de frecuencia --- par no procesar todos los frames,
↪ genera ruido. (esto se llama throttling)
        now = time.time()
        if now - self.last_process_time < self.process_interval:
            return
        self.last_process_time = now

        self.frame_count += 1
```

```
    #tambien publico solo cada 15 frames para no saturar la memoria
            if len(self.global_cloud) > 0 and self.frame_count % 15 == 0:
                header.frame_id = "world"
                msg = point_cloud2.create_cloud(header, fields,
↪ self.global_cloud)
                self.pub_dense_cloudMAP.publish(msg)
                self.get_logger().info(f"Nube densa global publicada con
↪ {len(points_global)} puntos nuevos.")
```

```
        mask = (disparity > 70) & np.isfinite(disparity) # filtro valores
↪ invalidos nan o inf, puntos sin correspondencia valida
        points = points_3d[mask]
```

Es posible que alguna definición de variables o inicialización de contadores no se muestre dentro de estos bloques.

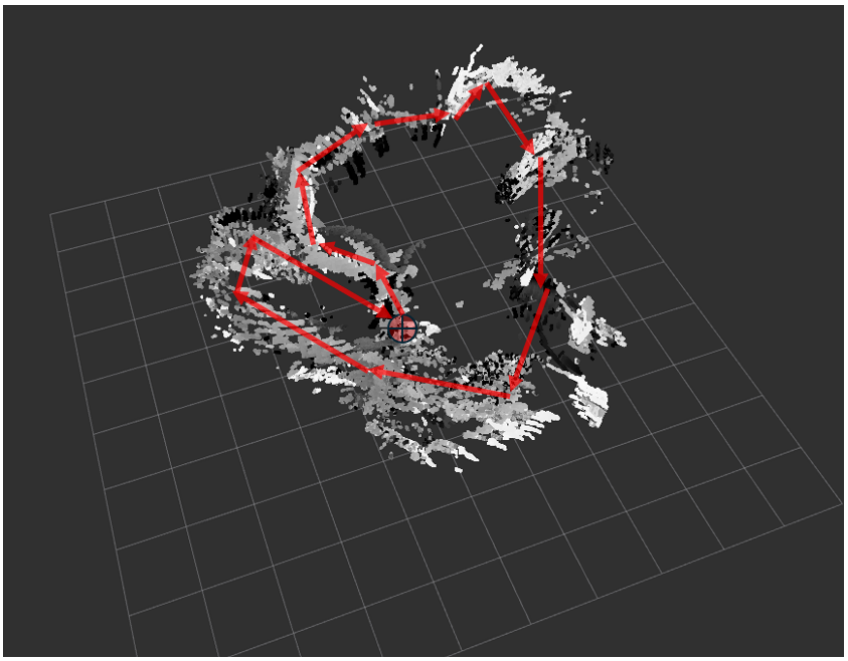

Figura 8: Mapeo Denso con disparidad $\dot{5}0$. trayectoria referenciada

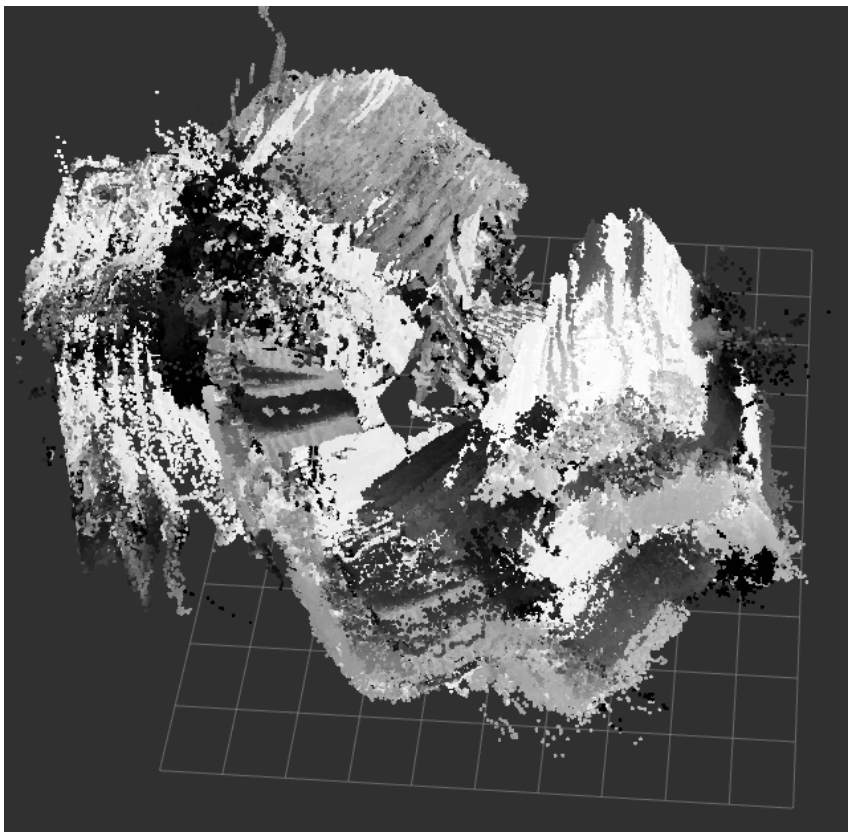

Figura 9: Mapeo denso con disparidad $\dot{1}0$

En las imágenes de arriba se observa, primero una nube densa global tomando solo puntos cuya disparidad sea mayor a 50, sobre la nube global se marca la trayectoria que se observa a medida que se publican los nuevos puntos, en concordancia con la observada en el video (escenario V201).

En la imagen de abajo se tomaron puntos con disparidad mayor a 10 (es decir más puntos posibles), donde se empieza a vislumbrar la representación 3D del entorno. La nube, en este caso, es de un tiempo menor de filmación, debido al consumo de memoria tuvo que frenarse antes, el escenario es el mismo.

Inciso j)

- Estimar la Pose utilizando Visión Monocular. Utilizando cv::recoverPose() estimar la transformación entre la cámara izquierda y la cámara derecha. Para esto deberá calcular la matriz esencial utilizando la función cv::findEssentialMat(). Notar que cv::recoverPose() retorna el vector unitario de traslación, por lo tanto deberá multiplicarlo por el factor de escala (en este caso el baseline entre las cámaras) para obtener la traslación estimada. Una vez hecho esto se pide:

- Visualizar la pose estimada de ambas cámaras. Agregar captura al informe.

- Estimar y visualizar la trayectoria realizada por la cámara izquierda utilizando como factor de escala la distancia entre cada par de frames dada por el ground-truth. Agregar captura al informe.

Para este punto, se utilizan los matches entre el frame de una cámara y el inmediato siguiente para calcular la matriz esencial, es decir la transformación geométrica entre ambas cámaras (misma cámara, distintos instantes). De esta manera, en vez de recuperar la pose entre dos cámaras en arreglo estero, utilizamos la matriz esencial para obtener la transformación geométrica entre un instante y otro y así estimar la trayectoria.

Comentarios: La lectura de los ground truth dentro del script fue algo que realizamos a lo último y no llegamos a implementarlo en todos los incisos que era necesario. La implementación se realiza de manera similar (casi igual) a como está realizada en el inciso e. Es por esto que algunas de las imágenes de mapeo tienen errores de sincronización entre frames y ground truth real.

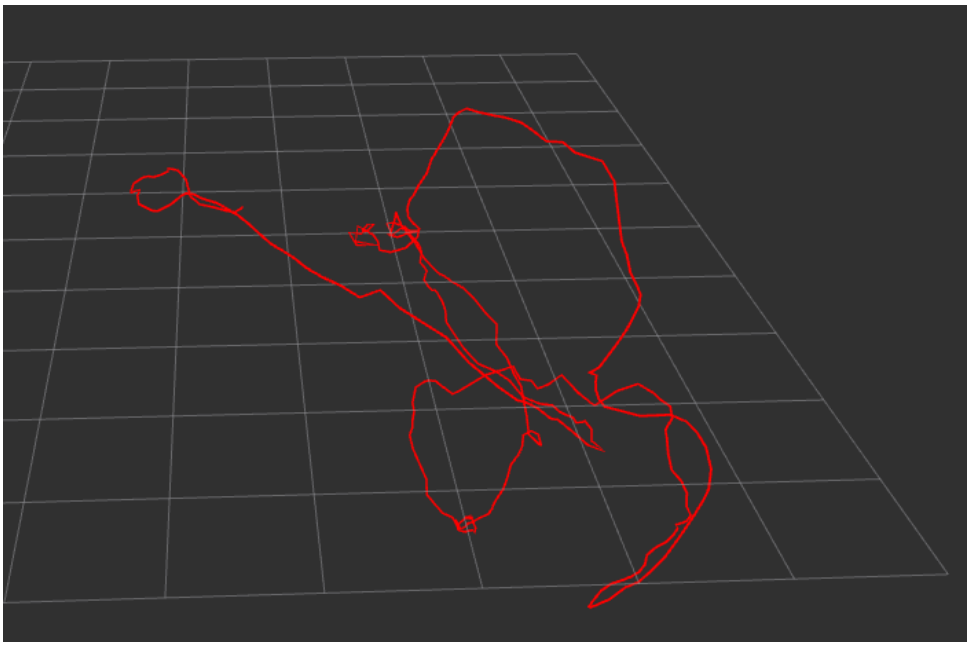

Figura 10: Trayectoria monocular

Referencias

[1] https://github.com/geronimogonzalez/Robotica-movil/tree/main

[2] http://robotics.ethz.ch/~asl-datasets/ijrr_euroc_mav_dataset/calibration_datasets/cam_checkerboard/checkerboard_7x6.yaml

[3] https://github.com/geronimogonzalez/Robotica-movil/tree/main/TP-3/cal-ros2