3-DOF SLAM: "Simultaneous Mosaicing and Tracking with an Event Camera"

In this exercise we partially re-implement the paper [1], which can be interpreted as a particular case of Simultaneous Localization and Mapping (SLAM) when the camera undergoes purely rotational motion (that is, 3-DOF instead of the 6-DOF of rotation and translation). "Localization" refers to the estimation of the pose of the camera (the oraientation in this simplified problem), and "Mapping" refers to the estimation of a map of the scene (i.e., the mosaic or panoramic image). Since the camera is does not translate, there is no parallax, and therefore it is not possible to estimate depth. Therefore, the mapping part builds only a brightness map of the scene, as opposed to a combined brightness plus depth map of the scene.

How are we tackling the problem?

The implementation can be divided in two parts (just like [1] describes the method): the localization function (section 3.1) and the mapping function (section 3.2). We will use *synthetic*, noise-free data produced by an event camera simulator, such as [2] or [3]. The synthetic data consists of events and control poses (translation and orientation). Events are provided in a ROS bag file published in a topic, in the way that events would be published by an event camera driver. Ground truth control poses at discrete times are specified in a txt file and are just used for prototyping.

We will first implement the mosaicing part using events and ground truth poses. At the end of this step, we will have created a "ground truth" panoramic map (in floating point values) that we can then use to prototype the camera tracking part. If time allows, we will combine both previous steps, getting rid of the ground truth poses, to yield a SLAM algorithm. During initialization, before map or poses are available, bootstrapping will be required to kick start the SLAM method.

1 Code provided

A skeleton of the code is provided in the following GitLab repository. Your task is to complete the code, at the locations indicated by "FILL in". Hints of what functions to use are provided next to the corresponding line.

1.1 How does the code work? Workflow of the code

- The file with the main() function (mosaic_node.cpp) creates an object of the type Mosaic (defined in mosaic.h), which processes the events and publishes the results.
- The parameters of the method are set on the Mosaic object constructor. Currently there are parameters that can be changed using dynamic reconfigure, but this is secondary (no need to worry about them for now).

- Incoming events are processed by the eventsCallback() function associated to the event topic subscriber. Events are appended to a queue and processed in groups ("packets") whenever there are enough events. Why in packets? Even though the method described in the paper can process one event at a time, there are some operations that can become expensive if performed on a per-event basis. For example, computing the rotation corresponding to each event, with the rotation changing at microsecond resolution is unnecessary. Reconstructing the brightness map using a Poisson solver that processes the entire gradient map at microsecond resolution is also unnecessary.

- Let us focus on the *mosaicing* part. The eventsCallback() function mainly calls processEventForMap() and publishMap(). The function processEventForMap() is (to be) implemented in the file mapping.cpp and it performs equations (6) to (15) in [1] (per-pixel EKF update of the mosaic gradient map).

  - Some functions to perform geometric operations involving 3D points and 2D points are (to be) implemented in the file geometry.cpp. For example, precomputeBearingVectors() computes the 3D point / direction corresponding to each pixel of the event camera (called $\mathtt{K}^{-1}\dot{\mathbf{p}}_c$ in Eq.(4) of [1]), and project_EquirectangularProjection() computes the 2D point $\mathbf{p}_m$ in the map corresponding to the projection of a given world (3D) point $\mathbf{p}_w$ (denoted by $\mathbf{p}_m = \pi(\mathbf{p}_w)$ in Eq.(4) of [1]).
  - To speed up the process of computing the point $\mathbf{p}_m^{(t-\tau_c)}$ in the paper (map point corresponding to the previous timestamp at the pixel of the current event), the rotation corresponding to the latest event at the pixel is stored in a "map of rotations", of the same size as the sensor resolution, where each pixel stores a rotation matrix. As events are processed, the timestamp map and this map of rotations need to be updated.
  - The Poisson integration of the mosaic gradient map is implemented using code that is available online. These files are already included in the git repository, and an interface to use them is partially implemented in the file reconstruction.cpp. Since this Poisson solver is "expensive" because it solves the Poisson equation from scratch every time it is called and since the brightness mosaic is not explicitly used in the mosaicing part, for now let us only compute it only if there is at least one subscriber in the corresponding publishing topic.

- After processing the current packet of events, the sensor time map, mosaic gradient map and other quantities are published (in different topics). Publishing is done in the file publish.cpp. Then, the processing "window" of events is shifted to process the next (more recent) subset of events (Operations on the queue of events). The mosaic gradient map and its covariance are continuously updated by each incoming event, and so they remain as shared variables throughout the iteration.

2 Event warping according to rotational motion and equirectangular projection

As mentioned, the function project_EquirectangularProjection() computes the 2D point $\mathbf{p}_m$ in the mosaic corresponding to the projection of a given world (3D) point $\mathbf{p}_w = (X, Y, Z)^\top$ (see Fig.1).

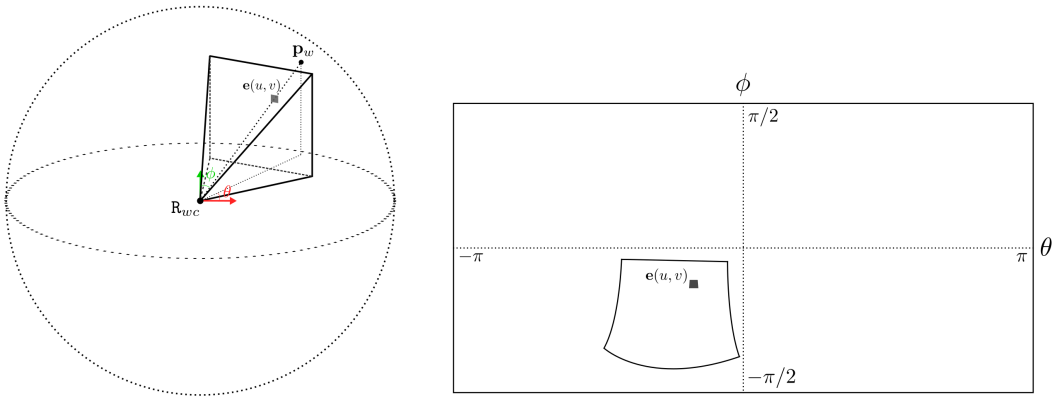

Figure 1: Left: geometry of the rotating camera, event on the image plane $\boldsymbol{e}(u,v)$ and the world point $\mathbf{p}_w$ on a sphere that projects onto $\boldsymbol{e}$. Right: projection of the sensor on the mosaic. The deformed rectangle corresponds to the current field of view (FOV) of the event camera, and it also shows the location corresponding to event $\boldsymbol{e}$. Figure adapted from [4].

Taking into account the width and height of the mosaic (in pixel units), w_m and h_m respectively, the projection is given by equations:

$$\mathbf{p}_m = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{w_m}{2} + f_x \arctan(X/Z) \\ \frac{h_m}{2} + f_y \arcsin(Y/\sqrt{X^2+Y^2+Z^2}) \end{pmatrix}$$

where $f_x = w_m/(2\pi)$ and $f_y = h_m/\pi$ are scale factors that convert the continuous ranges of the horizontal and vertical mosaic coordinates, from $[-\pi, \pi]$ and $[-\pi/2, \pi/2]$ (Fig. 1) to $[0, w_m]$ and $[0, h_m]$ respectively.

Remember to use atan2() to achieve the range of the horizontal coordinate θ in $[-\pi, \pi]$ (if you simply use the function atan(), you will only span half of the desired range).

Remarks:

- In the BVMC 2014 there is a typo in Eq. (9): polarity is missing. Polarity ($\pm$) is used in Eq. (7) and then it disappears in Eq. (9).
- In the mosaicing part no bilinear voting is used. Each event updates only the gradient (and its covariance) at a single pixel of the mosaic. For this, we use a mosaic of sufficient resolution (2048×1024 pixels with respect to a sensor resolution of the DVS128: 128×128 pixels).
- Remember to modify line 16 of *poses_grountruth.cpp* to be able to read the txt file with the ground truth poses in your computer.
- Ground truth poses (variable *poses_*) are stored in an std::map structure. You do not need to worry about this. Code is provided for the interpolation of poses if the time of the event is between the times of two ground truth poses.
- Publishers and subscribers are set in the constructor like it has been done in previous exercises (using the *nh_* node handler).

- In *mosaic.cpp*, please set
  measure_contrast_ = false;
  this boolean variable selects between two measurement functions. The measurement function in the BMVC paper (Eqs. (8) and (9) in the paper) corresponds to the option "*false*", and it has an associated noise variance *var_R_ = 1e4* seconds^(-2).

- Each pixel in the mosaic has an EKF that estimates the brightness gradient. This is a 2D state; its covariance is a 2×2 matrix. The covariance matrix is symmetric and positive definite, therefore it has only three different entries a, b and c:

$$P = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$$

  The covariance matrices of all mosaic pixels are stored in variable *grad_map_covar_*, which is a float 3-channel array. The 1st channel stores all the a entries, the 2nd channel stores all the b entries, and the 3rd channel stores all the c entries. This is a way in which the covariance matrices are efficiently stored (there is no need use a 4-channel array). The covariance matrices are updated using the EKF equations, which should be implemented in *processEventForMap*(). In the code, Pg is supposed to read the current covariance, compute the new one (using EKF eqs) and then be stored in grad_map_covar_ for the next iteration.

3 What to submit?

1. **Source code**: zip file with the completed ROS node dvs_mosaic, so that it can be copied into a catkin workspace that includes the package dependencies and can be copied, compiled and run, as in previous exercises.

   (a) **Alternatively, you may submit code in Python**, if you are more comfortable in that programming language than in C++. Please specify in a file (e.g., README.md) how to run your code (dependencies, etc.)

2. **Video** of the execution of the code, showing the different published topics of the method. A sample video is provided on the ISIS page. Feel free to show the different published topics using just 1-2 image views; there is no need to always show the reconstructed brightness mosaic (because it is relatively slow compared to other parts of the code). If the parameters have been changed from the default ones provided, please specify them.

References

[1] Kim, H., Handa, A., Benosman, R., Ieng, S.-H., Davison, A., "Simultaneous Mosaicing and Tracking with an Event Camera", British Machine Vision Conference (BMVC) 2014.

[2] Mueggler, E., Rebecq, H., Gallego, G., Delbruck, T., Scaramuzza, T., "The Event-Camera Dataset and Simulator: Event-based Data for Pose Estimation, Visual Odometry, and SLAM". International Journal of Robotics Research, Vol. 36, Issue 2, pages 142-149, Feb. 2017. http://rpg.ifi.uzh.ch/davis_data.html

[3] Rebecq, H., Gehrig, D., Scaramuzza, D., "ESIM: an Open Event Camera Simulator". Conference on Robot Learning (CoRL), Zurich, 2018. http://rpg.ifi.uzh.ch/esim.html

[4] Kim, H., "Real-time visual SLAM with an event camera", Ph.D. thesis, Imperial College London, UK, 2018. http://hdl.handle.net/10044/1/59704