Global optical flow estimation with an event camera using contrast maximization

In this exercise we perform contrast maximization in the simplest case: when all edges are moving with the same velocity $\mathbf{v} = (v_x, v_y)^\top$ on the image plane.

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. The files to be edited are: image_warped_events.cpp, optim_contrast_gsl.cpp and util.cpp.

1.1 How does the code work? Workflow of the code

- The main file (node.cpp) creates an object of the type GlobalFlowEstimator (defined in global_flow_estimator.h), which processes the events and publishes the results.
- The parameters of the method (specified in the launch file) are set on this object's constructor, using the function loadBaseOptions().
- 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, by means of the function maximizeContrast().
- The global flow vector $\mathbf{v}$ of the motion is estimated by the function maximizeContrast(), specified in the file optim_contrast_gsl.cpp. This function calls functions from the GSL library (gsl_multimin_fdfminimizer) to search for the flow vector that maximize the contrast function (actually GSL searches for the minima of (-1)·contrast).
  - The optimization routine calls the function to compute the contrast, which in turn needs to call the function to compute the IWE with the current events. The IWE is computed in the functions of the file image_warped_events.cpp, using event warping and bilinear voting (see Eqs. (2)).
  - The optimization routine uses finite differences (numerical derivatives) to approximate the derivative of the contrast function. The derivatives are used by the conjugate gradient method (optimization routine). There is no need to edit code related to the computation of the derivatives.

  - In maximizeContrast() you just need to make sure that you pass the right input vector and return the proper estimated flow. There is no need to modify other parts of this function.

- After processing the current packet of events, the IWEs (without and with motion compensation) are published side-by-side, along with the estimated global flow $\mathbf{v}$ (in a different topic). Then, the observation "window" of events is shifted to process the next (more recent) packet / subset of events (Operations on the queue of events). We take the flow $\mathbf{v}_{p-1}$ obtained from the previous packet of events to initialize the optimization of the flow $\mathbf{v}_p$ of the current packet of events, thus assuming that the velocity does not change much between consecutive packets.

1.2 Event warping according to optical flow motion

Each event $e_k = (\mathbf{x}_k, t_k, p_k)$ is transformed / warped to $e'_k = (\mathbf{x}'_k, t_{\text{ref}}, p_k)$, with $\mathbf{x} = (x, y)^\top$ the coordinates on the image plane. This transformation represents the effect of "transporting" the event from its space-time location $(\mathbf{x}_k, t_k)$ to a new location $(\mathbf{x}'_k, t_{\text{ref}})$ according to a trajectory defined by the motion parameters $\mathbf{v}$ (a straight line in space-time).

$$\mathbf{x}'_k = \mathbf{x}_k - (t_k - t_{\text{ref}})\mathbf{v}$$

(This is Eq. (1) in the CVPR 2018 paper). Note that the the timestamp of the event, t_k, affects the amount of displacement (warping). This warping is typically a few (3-4 pixels on the image plane). Thus, all events are transported to a common time, t_{ref} (we use as t_{ref} the timestamps of the first event in the packet), and we now operate with them, creating a histogram or image, the so-called image of warped events (the image "lives" in time t_{ref}), by summing

$$I(\mathbf{x}) = \sum_k b_k \ker(\mathbf{x} - \mathbf{x}'_k). \tag{1}$$

$\ker(\mathbf{x})$ is a kernel (a "voting function") that spreads the contribution of each warped event to its closest pixels (we use a combination of bilinear voting and Gaussian smoothing; a Gaussian kernel could be used directly, but it would be more expensive).

If polarity is used, we set $b_k = p_k$, otherwise we set $b_k = 1$ and the IWE simply counts warped events per pixel. Please review the slides provided on ISIS in past weeks; we have seen motion compensation before.

The contrast function is simply the variance of all the pixels of the IWE. Alternatively, we can use the mean of the squares of all the pixels as a contrast function, too. There are many objective (contrast) functions that could be used, as shown in this CVPR 2019 paper.

1.3 Bilinear voting

Recall the linear voting strategy that we applied to the temporal dimension of the voxel grid at the end of Exercise 2. In this exercise, we instead apply voting to the spatial dimensions (x and y). How to carry this operation was already specified in the document *linear_voting_doc.pdf* accompanying Exercise 2. We briefly review here.

To compute the image of warped events (IWE), we implement bilinear voting (also known as forward mapping in image processing or "splatting" in computer graphics) in function accumulateWarpedEvent(). Each event is accumulated in the IWE by splitting its vote among its four closest

pixels (a pixel is a 2D "bin"), according to fractional distances δ_x and δ_y along each dimension. The voting scheme becomes

$$\begin{aligned} \text{bin}_{i,j} &\mathrel{+}= (1-\delta_x)(1-\delta_y) \\ \text{bin}_{i+1,j} &\mathrel{+}= \delta_x(1-\delta_y) \\ \text{bin}_{i,j+1} &\mathrel{+}= (1-\delta_x)\delta_y \\ \text{bin}_{i+1,j+1} &\mathrel{+}= \delta_x\delta_y \end{aligned} \tag{2}$$

where indices i and j correspond to the x and y directions of the image, respectively. The weights on the right hand side of (2) are the same as those used in bilinear interpolation. Equations (2) correspond to the IWE without using polarity. In case of using the event polarity to compute the IWE, each event should split its polarity into the 4 closest pixels, in a similar way to Eq. (2). Once all events are accumulated in the IWE, each pixel will hold a "balance" of polarities. We have seen this type of image in Exercise 2 (albeit without motion compensation).

2 What to submit?

1. **Source code**: zip file with the completed ROS node dvs_global_flow, 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 (4 and 5).

   (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 on the provided perspective file (gflow.perspective, which shows the IWE without and with motion compensation, and the plot of the published global flow $\mathbf{v}$).

   (a) Sequences to use: choose two among slider_far.bag, slider_close.bag, slider_hdr_far.bag, slider_hdr_close.bag, boxes_translation.

   (b) Play with the parameters of the method: type of contrast function (variance or mean square), number of events per packet, number of events to slide for the next package, with or without polarity, etc. These parameters defined in the launch file.

   (c) Show variation of four parameters, and two values per parameter. Specify, either in the video or in an attached text file, what parameters are used in the experiment. For example: you could have a "base" experiment, with parameters $E_1 = \{p_1, p_2, \ldots, p_4\}$ and show new experiments by only changing one parameter at a time, $E_2 = \{p'_1, p_2, p_3, p_4\}$, $E_3 = \{p_1, p'_2, p_3, p_4\}$, $E_4 = \{p_1, p_2, p'_3, p_4\}$, $E_5 = \{p_1, p_2, p_3, p'_4\}$. *More importantly, try to understand* how the parameters affect the method and its performance on flow estimation.