```
#include "opencv2/ximgproc.hpp"
#include "stereo_match.h"

static cv::Rect computeROI(cv::Size2i src_sz, cv::Ptr<cv::StereoMatcher> matcher_instance) {
        int min_disparity = matcher_instance->getMinDisparity();
        int num_disparities = matcher_instance->getNumDisparities();
        int block_size = matcher_instance->getBlockSize();

        int bs2 = block_size / 2;
        int minD = min_disparity, maxD = min_disparity + num_disparities - 1;

        int xmin = maxD + bs2;
        int xmax = src_sz.width + minD - bs2;
        int ymin = bs2;
        int ymax = src_sz.height - bs2;

        cv::Rect r(xmin, ymin, xmax - xmin, ymax - ymin);
        return r;
}

static const cv::String keys =
                "{help   h   usage   ?   |                                              |   print   this
message                                                          }"
                "{@left                              |../data/aloeL.jpg    |    left    view    of    the
stereopair                                                 }"
                "{@right                             |../data/aloeR.jpg    |    right    view    of    the
stereopair                                                 }"
                "{GT                    |../data/aloeGT.png|  optional ground-truth disparity (MPI-Sintel or
Middlebury format) }"
                "{dst_path              |None                   | optional path to save the resulting filtered
disparity map              }"
                "{dst_raw_path      |None                   | optional path to save raw disparity map before
filtering              }"
                "{algorithm              |bm                              |  stereo  matching  method  (bm  or
sgbm)                                              }"
                "{filter                 |wls_conf                 | used post-filtering (wls_conf or wls_no_conf or
fbs_conf)              }"
                "{no-display                 |                                              |   don't   display
results                                                          }"
                "{no-downscale       |                               | force stereo matching on full-sized views to
improve quality           }"
```

```
        "{dst_conf_path    |None                  | optional path to save the confidence map used
in filtering         }"
        "{vis_mult          |1.0                     | coefficient used to scale disparity map
visualizations            }"
        "{max_disparity     |160                                   |  parameter  of  stereo
matching                                     }"
        "{window_size          |-1                                  |  parameter  of  stereo
matching                                     }"
        "{wls_lambda              |8000.0                            |  parameter  of  wls
post-filtering                               }"
        "{wls_sigma               |1.5                               |  parameter  of  wls
post-filtering                               }"
        "{fbs_spatial           |16.0                                |  parameter  of  fbs
post-filtering                               }"
        "{fbs_luma                 |8.0                              |  parameter  of  fbs
post-filtering                               }"
        "{fbs_chroma              |8.0                               |  parameter  of  fbs
post-filtering                               }"
        "{fbs_lambda              |128.0                             |  parameter  of  fbs
post-filtering                               }";

int camera_disparity_filtering_demo() {
    const char *path1, *path2;
    path1 = "cvdata/ambush_5_left.jpg";
    path2 = "cvdata/ambush_5_right.jpg";
    cv::Mat img1 = cv::imread(path1, cv::IMREAD_COLOR);
    cv::Mat img2 = cv::imread(path2, cv::IMREAD_COLOR);
    MatR disparityMatrix, confMatrix;
    camera_disparity_filtering(img1, img2, disparityMatrix, confMatrix, "", "", "", "");
    savePointCloud(disparityMatrix, MatR(), img1, get_point_cloud_name(264).c_str());
    return 1;
}

int camera_disparity_filtering(const cv::Mat &left,
                               const cv::Mat &right,
                               MatR &dispaityMatrix,
                               MatR &confMatrix,
                               const std::string &GT_path,
                               const std::string &dst_path,
                               const std::string &dst_raw_path,
                               const std::string &dst_conf_path) {
```

```
    cv::String algo = "sgbm";
    cv::String filter = "wls_conf";
//     bool no_display = 1;
    bool no_display = 0;
    bool no_downscale = 0;
//     bool no_downscale = 1;
//     int max_disp = 160;
//     int max_disp = 320;
    int max_disp = 640;
//     int max_disp = 480;
    double lambda = 8000.0;
    double sigma = 1.5;
    double fbs_spatial = 16.0;
    double fbs_luma = 8.0;
    double fbs_chroma = 8.0;
    double fbs_lambda = 128.0;
    double vis_mult = 1.0;

    int wsize;

    if (algo == "sgbm")
        wsize = 3; //default window size for SGBM
    else if (!no_downscale && algo == "bm" && filter == "wls_conf")
        wsize = 7; //default window size for BM on downscaled views (downscaling is performed
only for wls_conf)
    else
        wsize = 15; //default window size for BM on full-sized views

    bool noGT;
    cv::Mat GT_disp;
    if (GT_path.empty())
        noGT = true;
    else {
        noGT = false;
        if (cv::ximgproc::readGT(GT_path, GT_disp) != 0) {
            std::cout << "Cannot read ground truth image file: " << GT_path << std::endl;
            return -1;
        }
    }

    cv::Mat left_for_matcher, right_for_matcher;
    cv::Mat left_disp, right_disp;
    cv::Mat filtered_disp, solved_disp, solved_filtered_disp;
    cv::Mat conf_map = cv::Mat(left.rows, left.cols, CV_8U);
    conf_map = cv::Scalar(255);
    cv::Rect ROI;
    cv::Ptr<cv::ximgproc::DisparityWLSFilter> wls_filter;
    double matching_time, filtering_time;
    double solving_time = 0;
    if (max_disp <= 0 || max_disp % 16 != 0) {
        std::cout << "Incorrect max_disparity value: it should be positive and divisible by 16";
        return -1;
    }
    if (wsize <= 0 || wsize % 2 != 1) {
        std::cout << "Incorrect window_size value: it should be positive and odd";
        return -1;
    }

    if (filter == "wls_conf") // filtering with confidence (significantly better quality than wls_no_conf)
    {
        if (!no_downscale) {
            // downscale the views to speed-up the matching stage, as we will need to compute both left
            // and right disparity maps for confidence map computation
            //! [downscale]
            max_disp /= 2;
            if (max_disp % 16 != 0)
                max_disp += 16 - (max_disp % 16);
            resize(left, left_for_matcher, cv::Size(), 0.5, 0.5, VISION_INTER_LINEAR_EXACT);
            resize(right, right_for_matcher, cv::Size(), 0.5, 0.5, VISION_INTER_LINEAR_EXACT);
            //! [downscale]
        } else {
            left_for_matcher = left.clone();
            right_for_matcher = right.clone();
        }

        if (algo == "bm") {
            //! [matching]
            cv::Ptr<cv::StereoBM> left_matcher = cv::StereoBM::create(max_disp, wsize);
            wls_filter = cv::ximgproc::createDisparityWLSFilter(left_matcher);
            cv::Ptr<cv::StereoMatcher> right_matcher = cv::ximgproc::createRightMatcher(left_matcher);
            cvtColor(left_for_matcher, left_for_matcher, cv::COLOR_BGR2GRAY);
            cvtColor(right_for_matcher, right_for_matcher, cv::COLOR_BGR2GRAY);

            matching_time = (double) cv::getTickCount();
            left_matcher->compute(left_for_matcher, right_for_matcher, left_disp);
            right_matcher->compute(right_for_matcher, left_for_matcher, right_disp);
            matching_time = ((double) cv::getTickCount() - matching_time) /
cv::getTickFrequency();
            //! [matching]
        } else if (algo == "sgbm") {
            cv::Ptr<cv::StereoSGBM> left_matcher = cv::StereoSGBM::create(0, max_disp, wsize);
            left_matcher->setP1(24 * wsize * wsize);
            left_matcher->setP2(96 * wsize * wsize);
            left_matcher->setPreFilterCap(63);
            left_matcher->setMode(cv::StereoSGBM::MODE_SGBM_3WAY);
            wls_filter = cv::ximgproc::createDisparityWLSFilter(left_matcher);
            cv::Ptr<cv::StereoMatcher> right_matcher =
cv::ximgproc::createRightMatcher(left_matcher);

            matching_time = (double) cv::getTickCount();
            left_matcher->compute(left_for_matcher, right_for_matcher, left_disp);
            right_matcher->compute(right_for_matcher, left_for_matcher, right_disp);
            matching_time = ((double) cv::getTickCount() - matching_time) /
cv::getTickFrequency();
        } else {
            std::cout << "Unsupported algorithm";
            return -1;
        }

        //! [filtering]
        wls_filter->setLambda(lambda);
        wls_filter->setSigmaColor(sigma);
        filtering_time = (double) cv::getTickCount();
        wls_filter->filter(left_disp, left, filtered_disp, right_disp);
        filtering_time = ((double) cv::getTickCount() - filtering_time) / cv::getTickFrequency();
        //! [filtering]
        conf_map = wls_filter->getConfidenceMap();

        // Get the ROI that was used in the last filter call:
        ROI = wls_filter->getROI();
        if (!no_downscale) {
            // upscale raw disparity and ROI back for a proper comparison:
            resize(left_disp, left_disp, cv::Size(), 2.0, 2.0, VISION_INTER_LINEAR_EXACT);
            left_disp = left_disp * 2.0;
            ROI = cv::Rect(ROI.x * 2, ROI.y * 2, ROI.width * 2, ROI.height * 2);
        }
    } else if (filter == "fbs_conf") // filtering with fbs and confidence using also wls pre-processing
    {
        if (!no_downscale) {
            // downscale the views to speed-up the matching stage, as we will need to compute
both left
            // and right disparity maps for confidence map computation
            //! [downscale_wls]
            max_disp /= 2;
            if (max_disp % 16 != 0)
                max_disp += 16 - (max_disp % 16);
            resize(left, left_for_matcher, cv::Size(), 0.5, 0.5);
            resize(right, right_for_matcher, cv::Size(), 0.5, 0.5);
            //! [downscale_wls]
        } else {
            left_for_matcher = left.clone();
            right_for_matcher = right.clone();
        }

        if (algo == "bm") {
            //! [matching_wls]
            cv::Ptr<cv::StereoBM> left_matcher = cv::StereoBM::create(max_disp, wsize);
            wls_filter = cv::ximgproc::createDisparityWLSFilter(left_matcher);
            cv::Ptr<cv::StereoMatcher>                    right_matcher                    =
cv::ximgproc::createRightMatcher(left_matcher);

            cvtColor(left_for_matcher, left_for_matcher, cv::COLOR_BGR2GRAY);
            cvtColor(right_for_matcher, right_for_matcher, cv::COLOR_BGR2GRAY);

            matching_time = (double) cv::getTickCount();
            left_matcher->compute(left_for_matcher, right_for_matcher, left_disp);
            right_matcher->compute(right_for_matcher, left_for_matcher, right_disp);
            matching_time    =    ((double)    cv::getTickCount()    -    matching_time)    /
cv::getTickFrequency();
            //! [matching_wls]
        } else if (algo == "sgbm") {
            cv::Ptr<cv::StereoSGBM> left_matcher = cv::StereoSGBM::create(0, max_disp, wsize);
            left_matcher->setP1(24 * wsize * wsize);
            left_matcher->setP2(96 * wsize * wsize);
            left_matcher->setPreFilterCap(63);
            left_matcher->setMode(cv::StereoSGBM::MODE_SGBM_3WAY);
            wls_filter = cv::ximgproc::createDisparityWLSFilter(left_matcher);
            cv::Ptr<cv::StereoMatcher> right_matcher =
cv::ximgproc::createRightMatcher(left_matcher);

            matching_time = (double) cv::getTickCount();
            left_matcher->compute(left_for_matcher, right_for_matcher, left_disp);
            right_matcher->compute(right_for_matcher, left_for_matcher, right_disp);
            matching_time = ((double) cv::getTickCount() - matching_time) /
cv::getTickFrequency();
        } else {
            std::cout << "Unsupported algorithm";
            return -1;
        }

        //! [filtering_wls]
        wls_filter->setLambda(lambda);
        wls_filter->setSigmaColor(sigma);
        filtering_time = (double) cv::getTickCount();
        wls_filter->filter(left_disp, left, filtered_disp, right_disp);
        filtering_time = ((double) cv::getTickCount() - filtering_time) / cv::getTickFrequency();
        //! [filtering_wls]

        conf_map = wls_filter->getConfidenceMap();

        cv::Mat left_disp_resized;
        resize(left_disp, left_disp_resized, left.size());

        // Get the ROI that was used in the last filter call:
        ROI = wls_filter->getROI();
        if (!no_downscale) {
            // upscale raw disparity and ROI back for a proper comparison:
            resize(left_disp, left_disp, cv::Size(), 2.0, 2.0);
            left_disp = left_disp * 2.0;
            left_disp_resized = left_disp_resized * 2.0;
            ROI = cv::Rect(ROI.x * 2, ROI.y * 2, ROI.width * 2, ROI.height * 2);
        }
```

```
#ifdef HAVE_EIGEN
        //! [filtering_fbs]
        solving_time = (double)cv::getTickCount();
        fastBilateralSolverFilter(left, left_disp_resized, conf_map/255.0f, solved_disp, fbs_spatial,
fbs_luma, fbs_chroma, fbs_lambda);
        solving_time = ((double)cv::getTickCount() - solving_time)/cv::getTickFrequency();
        //! [filtering_fbs]

        //! [filtering_wls2fbs]
        fastBilateralSolverFilter(left, filtered_disp, conf_map/255.0f, solved_filtered_disp,
fbs_spatial, fbs_luma, fbs_chroma, fbs_lambda);
        //! [filtering_wls2fbs]
#else
        (void) fbs_spatial;
        (void) fbs_luma;
        (void) fbs_chroma;
        (void) fbs_lambda;
#endif
    } else if (filter == "wls_no_conf") {
        /* There is no convenience function for the case of filtering with no confidence, so we
        will need to set the ROI and matcher parameters manually */

        left_for_matcher = left.clone();
        right_for_matcher = right.clone();

        if (algo == "bm") {
            cv::Ptr<cv::StereoBM> matcher = cv::StereoBM::create(max_disp, wsize);
            matcher->setTextureThreshold(0);
            matcher->setUniquenessRatio(0);
            cvtColor(left_for_matcher, left_for_matcher, cv::COLOR_BGR2GRAY);
            cvtColor(right_for_matcher, right_for_matcher, cv::COLOR_BGR2GRAY);
            ROI = computeROI(left_for_matcher.size(), matcher);
            wls_filter = cv::ximgproc::createDisparityWLSFilterGeneric(false);
            wls_filter->setDepthDiscontinuityRadius((int) ceil(0.33 * wsize));

            matching_time = (double) cv::getTickCount();
            matcher->compute(left_for_matcher, right_for_matcher, left_disp);
            matching_time = ((double) cv::getTickCount() - matching_time) /
cv::getTickFrequency();
        } else if (algo == "sgbm") {
            cv::Ptr<cv::StereoSGBM> matcher = cv::StereoSGBM::create(0, max_disp, wsize);
            matcher->setUniquenessRatio(0);
            matcher->setDisp12MaxDiff(1000000);
            matcher->setSpeckleWindowSize(0);
            matcher->setP1(24 * wsize * wsize);
            matcher->setP2(96 * wsize * wsize);
            matcher->setMode(cv::StereoSGBM::MODE_SGBM_3WAY);
            ROI = computeROI(left_for_matcher.size(), matcher);
            wls_filter = cv::ximgproc::createDisparityWLSFilterGeneric(false);
            wls_filter->setDepthDiscontinuityRadius((int) ceil(0.5 * wsize));

            matching_time = (double) cv::getTickCount();
            matcher->compute(left_for_matcher, right_for_matcher, left_disp);
            matching_time = ((double) cv::getTickCount() - matching_time) / cv::getTickFrequency();
        } else {
            std::cout << "Unsupported algorithm";
            return -1;
        }
        wls_filter->setLambda(lambda);
        wls_filter->setSigmaColor(sigma);
        filtering_time = (double) cv::getTickCount();
        wls_filter->filter(left_disp, left, filtered_disp, cv::Mat(), ROI);
        filtering_time = ((double) cv::getTickCount() - filtering_time) / cv::getTickFrequency();
    } else {
        std::cout << "Unsupported filter";
        return -1;
    }
    if (isDebugEnabled()) {
        //collect and print all the stats:
        std::cout.precision(2);
        std::cout << "Matching time:    " << matching_time << "s" << std::endl;
        std::cout << "Filtering time: " << filtering_time << "s" << std::endl;
        std::cout << "Solving time: " << solving_time << "s" << std::endl;
        std::cout << std::endl;
    }

    double MSE_before, percent_bad_before, MSE_after, percent_bad_after;
    if (!noGT) {
        MSE_before = cv::ximgproc::computeMSE(GT_disp, left_disp, ROI);
        percent_bad_before = cv::ximgproc::computeBadPixelPercent(GT_disp, left_disp, ROI);
        MSE_after = cv::ximgproc::computeMSE(GT_disp, filtered_disp, ROI);
        percent_bad_after = cv::ximgproc::computeBadPixelPercent(GT_disp, filtered_disp, ROI);
        std::cout.precision(5);
        std::cout << "MSE before filtering: " << MSE_before << std::endl;
        std::cout << "MSE after filtering:    " << MSE_after << std::endl;
        std::cout << std::endl;
        std::cout.precision(3);
        std::cout << "Percent of bad pixels before filtering: " << percent_bad_before << std::endl;
        std::cout << "Percent of bad pixels after filtering:    " << percent_bad_after << std::endl;
    }
    if (!dst_path.empty()) {
        cv::Mat filtered_disp_vis;
        cv::ximgproc::getDisparityVis(filtered_disp, filtered_disp_vis, vis_mult);
        imwrite(dst_path, filtered_disp_vis);
    }
    if (!dst_raw_path.empty()) {
        cv::Mat raw_disp_vis;
        cv::ximgproc::getDisparityVis(left_disp, raw_disp_vis, vis_mult);
        imwrite(dst_raw_path, raw_disp_vis);
    }
    if (!dst_conf_path.empty()) {
        imwrite(dst_conf_path, conf_map);
    }

#if IS_SHOWING
    if (!no_display) {
        vi_cv_namedWindow("left", cv::WINDOW_NORMAL);
        vi_cv_imshow("left", left);
        vi_cv_namedWindow("right", cv::WINDOW_NORMAL);
        vi_cv_imshow("right", right);

        if (!noGT) {
            cv::Mat GT_disp_vis;
            cv::ximgproc::getDisparityVis(GT_disp, GT_disp_vis, vis_mult);
            vi_cv_namedWindow("ground-truth disparity", cv::WINDOW_NORMAL);
            vi_cv_imshow("ground-truth disparity", GT_disp_vis);
        }

        //! [visualization]
        cv::Mat raw_disp_vis;
        cv::ximgproc::getDisparityVis(left_disp, raw_disp_vis, vis_mult);
        vi_cv_namedWindow("raw disparity", cv::WINDOW_NORMAL);
        vi_cv_imshow("raw disparity", raw_disp_vis);
        cv::imwrite(getAbsPath("raw_disparity") + ".jpg", raw_disp_vis);
        cv::Mat filtered_disp_vis;
        cv::ximgproc::getDisparityVis(filtered_disp, filtered_disp_vis, vis_mult);
        vi_cv_namedWindow("filtered disparity", cv::WINDOW_NORMAL);
        vi_cv_imshow("filtered disparity", filtered_disp_vis);
        cv::imwrite(getAbsPath("filtered_disparity") + ".jpg", filtered_disp_vis);

        if (!solved_disp.empty()) {
            cv::Mat solved_disp_vis;
            cv::ximgproc::getDisparityVis(solved_disp, solved_disp_vis, vis_mult);
            vi_cv_namedWindow("solved disparity", cv::WINDOW_NORMAL);
            vi_cv_imshow("solved disparity", solved_disp_vis);

            cv::Mat solved_filtered_disp_vis;
            cv::ximgproc::getDisparityVis(solved_filtered_disp, solved_filtered_disp_vis, vis_mult);
            vi_cv_namedWindow("solved wls disparity", cv::WINDOW_NORMAL);
            vi_cv_imshow("solved wls disparity", solved_filtered_disp_vis);
        }

        while (1) {
            char key = (char) vi_cv_waitKey();
            if (key == 27 || key == 'q' || key == 'Q') // 'ESC'
                break;
        }
        //! [visualization]
    }
#endif

    cv::Mat disparity;
    filtered_disp.convertTo(disparity, CV_32F, 1 / 16.0f);
//      printInfo(conf_map);
    conf_map = conf_map * (1 / 255.0f);
    toMat(disparity, dispaityMatrix);
    confMatrix.swap(toM(conf_map).get<TenR>());
    return 0;
}
```