```
struct CV_EXPORTS MatchesInfo {
            MatchesInfo();

            MatchesInfo(const MatchesInfo &other);

            MatchesInfo &operator=(const MatchesInfo &other);

            int src_img_idx, dst_img_idx;          //!< Images indices (optional)
            std::vector<DMatch> matches;
            std::vector<uchar> inliers_mask;       //!< Geometrically consistent matches mask
            int num_inliers;                              //!< Number of geometrically consistent
matches
            Mat H;                                        //!< Estimated transformation
            double confidence;                            //!< Confidence two images are from the
same panorama
        };
MatchesInfo &MatchesInfo::operator=(const MatchesInfo &other) {
            src_img_idx = other.src_img_idx;
            dst_img_idx = other.dst_img_idx;
            matches = other.matches;
            inliers_mask = other.inliers_mask;
            num_inliers = other.num_inliers;
            H = other.H.clone();
            confidence = other.confidence;
            return *this;
        }
/** @brief Class for matching keypoint descriptors

query descriptor index, train descriptor index, train image index, and distance between
descriptors.
*/
class CV_EXPORTS_W_SIMPLE DMatch
{
public:
    CV_WRAP DMatch();
    CV_WRAP DMatch(int _queryIdx, int _trainIdx, float _distance);
    CV_WRAP DMatch(int _queryIdx, int _trainIdx, int _imgIdx, float _distance);

    CV_PROP_RW int queryIdx; // query descriptor index
    CV_PROP_RW int trainIdx; // train descriptor index
    CV_PROP_RW int imgIdx;    // train image index
    CV_PROP_RW float distance;

    // less is better
    bool operator<(const DMatch &m) const;
};
/** @brief Feature matchers base class. */
        class CV_EXPORTS FeaturesMatcher {
        public:
            virtual ~FeaturesMatcher() {}

            /** @overload
            @param features1 First image features
            @param features2 Second image features
            @param matches_info Found matches
            */
            void operator()(const ImageFeatures &features1, const ImageFeatures &features2,
                            MatchesInfo &matches_info) { match(features1, features2,
matches_info); }

            /** @brief Performs images matching.

            @param features Features of the source images
            @param pairwise_matches Found pairwise matches
            @param mask Mask indicating which image pairs must be matched

            The function is parallelized with the TBB library.

            @sa detail::MatchesInfo
            */
            void operator()(const std::vector<ImageFeatures> &features,
std::vector<MatchesInfo> &pairwise_matches,
                            const cv::UMat &mask = cv::UMat());

            /** @return True, if it's possible to use the same matcher instance in parallel, false
otherwise
            */
            bool isThreadSafe() const { return is_thread_safe_; }

            /** @brief Frees unused memory allocated before if there is any.
            */
```

```
                virtual void collectGarbage() {}

          protected:
                FeaturesMatcher(bool is_thread_safe = false) : is_thread_safe_(is_thread_safe) {}

                /** @brief This method must implement matching logic in order to make the
wrappers
                detail::FeaturesMatcher::operator()_ work.

                @param features1 first image features
                @param features2 second image features
                @param matches_info found matches
                  */
                virtual void match(const ImageFeatures &features1, const ImageFeatures &features2,
                                         MatchesInfo &matches_info) = 0;

                bool is_thread_safe_;
          };

/** @brief Features matcher which finds two best matches for each feature and leaves the best one
only if the
ratio between descriptor distances is greater than the threshold match_conf

@sa detail::FeaturesMatcher
  */
          class CV_EXPORTS BestOf2NearestMatcher : public FeaturesMatcher {
          public:
                /** @brief Constructs a "best of 2 nearest" matcher.

                @param try_use_gpu Should try to use GPU or not
                @param match_conf Match distances ration threshold
                @param num_matches_thresh1 Minimum number of matches required for the 2D
projective transform
                estimation used in the inliers classification step
                @param num_matches_thresh2 Minimum number of matches required for the 2D
projective transform
                re-estimation on inliers
                  */
                BestOf2NearestMatcher(bool try_use_gpu = false, float match_conf = 0.3f, int
num_matches_thresh1 = 6,
                                             int num_matches_thresh2 = 6);
```

```
            void collectGarbage();

        protected:
            void match(const ImageFeatures &features1, const ImageFeatures &features2,
                        MatchesInfo &matches_info);

            int num_matches_thresh1_;
            int num_matches_thresh2_;
            Ptr<FeaturesMatcher> impl_;
        };
/** @brief Features matcher similar to cv::detail::BestOf2NearestMatcher which
finds two best matches for each feature and leaves the best one only if the
ratio between descriptor distances is greater than the threshold match_conf.

Unlike cv::detail::BestOf2NearestMatcher this matcher uses affine
transformation (affine transformation estimate will be placed in matches_info).

@sa cv::detail::FeaturesMatcher cv::detail::BestOf2NearestMatcher
 */
        class CV_EXPORTS AffineBestOf2NearestMatcher : public BestOf2NearestMatcher {
        public:
            /** @brief Constructs a "best of 2 nearest" matcher that expects affine
transformation
            between images

            @param full_affine whether to use full affine transformation with 6 degress of
freedom or reduced
            transformation with 4 degrees of freedom using only rotation, translation and uniform
scaling
            @param try_use_gpu Should try to use GPU or not
            @param match_conf Match distances ration threshold
            @param num_matches_thresh1 Minimum number of matches required for the 2D
affine transform
            estimation used in the inliers classification step

            @sa cv::estimateAffine2D cv::estimateAffinePartial2D
             */
            AffineBestOf2NearestMatcher(bool full_affine = false, bool try_use_gpu = false,
                                         float match_conf = 0.3f, int num_matches_thresh1
= 6) :
```

```
                    BestOf2NearestMatcher(try_use_gpu, match_conf, num_matches_thresh1,
num_matches_thresh1),
                    full_affine_(full_affine) {}

        protected:
            void match(const ImageFeatures &features1, const ImageFeatures &features2,
                       MatchesInfo &matches_info);

            bool full_affine_;
        };
BestOf2NearestMatcher::BestOf2NearestMatcher(bool try_use_gpu, float match_conf, int
num_matches_thresh1,
                                                     int num_matches_thresh2) {
            CV_UNUSED(try_use_gpu);

#ifdef HAVE_OPENCV_CUDAFEATURES2D
            if (try_use_gpu && getCudaEnabledDeviceCount() > 0)
            {
                impl_ = makePtr<GpuMatcher>(match_conf);
            }
            else
#endif
            {
                impl_ = makePtr<CpuMatcher>(match_conf);
            }

            is_thread_safe_ = impl_->isThreadSafe();
            num_matches_thresh1_ = num_matches_thresh1;
            num_matches_thresh2_ = num_matches_thresh2;
        }


        void BestOf2NearestMatcher::match(const ImageFeatures &features1, const
ImageFeatures &features2,
                                          MatchesInfo &matches_info) {

            (*impl_)(features1, features2, matches_info);

            // Check if it makes sense to find homography
            if (matches_info.matches.size() < static_cast<size_t>(num_matches_thresh1_))
                return;
```

```
// Construct point-point correspondences for homography estimation
Mat src_points(1, static_cast<int>(matches_info.matches.size()), CV_32FC2);
Mat dst_points(1, static_cast<int>(matches_info.matches.size()), CV_32FC2);
for (size_t i = 0; i < matches_info.matches.size(); ++i) {
    const DMatch &m = matches_info.matches[i];

    Point2f p = features1.keypoints[m.queryIdx].pt;
    p.x -= features1.img_size.width * 0.5f;
    p.y -= features1.img_size.height * 0.5f;
    src_points.at<Point2f>(0, static_cast<int>(i)) = p;

    p = features2.keypoints[m.trainIdx].pt;
    p.x -= features2.img_size.width * 0.5f;
    p.y -= features2.img_size.height * 0.5f;
    dst_points.at<Point2f>(0, static_cast<int>(i)) = p;
}

// Find pair-wise motion
matches_info.H = findHomography(src_points, dst_points, matches_info.inliers_mask, RANSAC);
if (matches_info.H.empty() ||
    std::abs(determinant(matches_info.H)) < std::numeric_limits<double>::epsilon())
    return;

// Find number of inliers
matches_info.num_inliers = 0;
for (size_t i = 0; i < matches_info.inliers_mask.size(); ++i)
    if (matches_info.inliers_mask[i])
        matches_info.num_inliers++;

// These coeffs are from paper M. Brown and D. Lowe. "Automatic Panoramic Image Stitching
// using Invariant Features"
matches_info.confidence = matches_info.num_inliers / (8 + 0.3 * matches_info.matches.size());

// Set zero confidence to remove matches between too close images, as they don't provide
// additional information anyway. The threshold was set experimentally.
matches_info.confidence = matches_info.confidence > 3. ? 0. :
```

```
matches_info.confidence;

            // Check if we should try to refine motion
            if (matches_info.num_inliers < num_matches_thresh2_)
                return;

            // Construct point-point correspondences for inliers only
            src_points.create(1, matches_info.num_inliers, CV_32FC2);
            dst_points.create(1, matches_info.num_inliers, CV_32FC2);
            int inlier_idx = 0;
            for (size_t i = 0; i < matches_info.matches.size(); ++i) {
                if (!matches_info.inliers_mask[i])
                    continue;

                const DMatch &m = matches_info.matches[i];

                Point2f p = features1.keypoints[m.queryIdx].pt;
                p.x -= features1.img_size.width * 0.5f;
                p.y -= features1.img_size.height * 0.5f;
                src_points.at<Point2f>(0, inlier_idx) = p;

                p = features2.keypoints[m.trainIdx].pt;
                p.x -= features2.img_size.width * 0.5f;
                p.y -= features2.img_size.height * 0.5f;
                dst_points.at<Point2f>(0, inlier_idx) = p;

                inlier_idx++;
            }

            // Rerun motion estimation on inliers only
            matches_info.H = findHomography(src_points, dst_points, RANSAC);
        }

        void BestOf2NearestMatcher::collectGarbage() {
            impl_->collectGarbage();
        }
// These two classes are aimed to find features matches only, not to
// estimate homography

        class CpuMatcher : public FeaturesMatcher {
        public:
            CpuMatcher(float match_conf) : FeaturesMatcher(true), match_conf_(match_conf) {}

            void match(const ImageFeatures &features1, const ImageFeatures &features2,
                        MatchesInfo &matches_info);

        private:
            float match_conf_;
        };
void
        CpuMatcher::match(const ImageFeatures &features1, const ImageFeatures &features2,
MatchesInfo &matches_info) {
            CV_Assert(features1.descriptors.type() == features2.descriptors.type());
            CV_Assert(features2.descriptors.depth() == CV_8U || features2.descriptors.depth() ==
CV_32F);

#ifdef HAVE_TEGRA_OPTIMIZATION
            if (tegra::useTegra() && tegra::match2nearest(features1, features2, matches_info,
match_conf_))
                return;
#endif

            matches_info.matches.clear();
            Ptr<cv::DescriptorMatcher> matcher;
#if 0 // TODO check this
            if (ocl::isOpenCLActivated())
            {
                matcher = makePtr<BFMatcher>((int)NORM_L2);
            }
            else
#endif
            {
                Ptr<flann::IndexParams> indexParams = makePtr<flann::KDTreeIndexParams>();
                Ptr<flann::SearchParams> searchParams = makePtr<flann::SearchParams>();

                if (features2.descriptors.depth() == CV_8U) {
                    indexParams->setAlgorithm(cvflann::FLANN_INDEX_LSH);
                    searchParams->setAlgorithm(cvflann::FLANN_INDEX_LSH);
                }

                matcher = makePtr<FlannBasedMatcher>(indexParams, searchParams);
            }
            std::vector<std::vector<DMatch> > pair_matches;
            MatchesSet matches;

            // Find 1->2 matches
            matcher->knnMatch(features1.descriptors, features2.descriptors, pair_matches, 2);
            for (size_t i = 0; i < pair_matches.size(); ++i) {
                if (pair_matches[i].size() < 2)
                    continue;
                const DMatch &m0 = pair_matches[i][0];
                const DMatch &m1 = pair_matches[i][1];
                if (m0.distance < (1.f - match_conf_) * m1.distance) {
                    matches_info.matches.push_back(m0);
                    matches.insert(std::make_pair(m0.queryIdx, m0.trainIdx));
                }
            }
            LOG("\n1->2 matches: " << matches_info.matches.size() << endl);

            // Find 2->1 matches
            pair_matches.clear();
            matcher->knnMatch(features2.descriptors, features1.descriptors, pair_matches, 2);
            for (size_t i = 0; i < pair_matches.size(); ++i) {
                if (pair_matches[i].size() < 2)
                    continue;
                const DMatch &m0 = pair_matches[i][0];
                const DMatch &m1 = pair_matches[i][1];
                if (m0.distance < (1.f - match_conf_) * m1.distance)
                    if (matches.find(std::make_pair(m0.trainIdx, m0.queryIdx)) ==
matches.end())
                        matches_info.matches.push_back(DMatch(m0.trainIdx, m0.queryIdx,
m0.distance));
            }
            LOG("1->2 & 2->1 matches: " << matches_info.matches.size() << endl);
        }
void AffineBestOf2NearestMatcher::match(const ImageFeatures &features1, const ImageFeatures
&features2,
                                                    MatchesInfo &matches_info) {
            (*impl_)(features1, features2, matches_info);
            // Check if it makes sense to find transform
            if (matches_info.matches.size() < static_cast<size_t>(num_matches_thresh1_))
                return;
            // Construct point-point correspondences for transform estimation
            Mat src_points(1, static_cast<int>(matches_info.matches.size()), CV_32FC2);
            Mat dst_points(1, static_cast<int>(matches_info.matches.size()), CV_32FC2);
            for (size_t i = 0; i < matches_info.matches.size(); ++i) {
                const cv::DMatch &m = matches_info.matches[i];
                src_points.at<Point2f>(0, static_cast<int>(i)) = features1.keypoints[m.queryIdx].pt;
                dst_points.at<Point2f>(0, static_cast<int>(i)) = features2.keypoints[m.trainIdx].pt;
            }
            // Find pair-wise motion
            if (full_affine_)
                matches_info.H = estimateAffine2D(src_points, dst_points, matches_info.inliers_mask);
            else
                matches_info.H = estimateAffinePartial2D(src_points, dst_points, matches_info.inliers_mask);
            if (matches_info.H.empty()) {
                // could not find transformation
                matches_info.confidence = 0;
                matches_info.num_inliers = 0;
                return;
            }
            // Find number of inliers
            matches_info.num_inliers = 0;
            for (size_t i = 0; i < matches_info.inliers_mask.size(); ++i)
                if (matches_info.inliers_mask[i])
                    matches_info.num_inliers++;
            // These coeffs are from paper M. Brown and D. Lowe. "Automatic Panoramic
            // Image Stitching using Invariant Features"
            matches_info.confidence =
                    matches_info.num_inliers / (8 + 0.3 * matches_info.matches.size());

            /* should we remove matches between too close images? */
            // matches_info.confidence = matches_info.confidence > 3. ? 0. : matches_info.confidence;

            // extend H to represent linear transformation in homogeneous coordinates
            matches_info.H.push_back(Mat::zeros(1, 3, CV_64F));
            matches_info.H.at<double>(2, 2) = 1;
        }
```