```cpp
class CV_EXPORTS_W Estimator {
    public:
        virtual ~Estimator() {}

        /** @brief Estimates camera parameters.

        @param features Features of images
        @param pairwise_matches Pairwise matches of images
        @param cameras Estimated camera parameters
        @return True in case of success, false otherwise
         */
        CV_WRAP_AS(apply)

        bool operator()(const std::vector<ImageFeatures> &features,
                        const std::vector<MatchesInfo> &pairwise_matches,
                        CV_OUT CV_IN_OUT std::vector<CameraParams> &cameras) {
            return estimate(features, pairwise_matches, cameras);
        }

    protected:
        /** @brief This method must implement camera parameters estimation logic in order
to make the wrapper
        detail::Estimator::operator()_ work.

        @param features Features of images
        @param pairwise_matches Pairwise matches of images
        @param cameras Estimated camera parameters
        @return True in case of success, false otherwise
         */
        virtual bool estimate(const std::vector<ImageFeatures> &features,
                              const std::vector<MatchesInfo> &pairwise_matches,
                              CV_OUT std::vector<CameraParams> &cameras) = 0;
};

/** @brief Homography based rotation estimator.
 */
    class CV_EXPORTS_W HomographyBasedEstimator : public Estimator {
    public:
        CV_WRAP HomographyBasedEstimator(bool is_focals_estimated = false)
                : is_focals_estimated_(is_focals_estimated) {}
```

```
        private:
                virtual bool estimate(const std::vector<ImageFeatures> &features,
                                        const std::vector<MatchesInfo> &pairwise_matches,
                                        std::vector<CameraParams> &cameras);


                bool is_focals_estimated_;
        };
```

/** @brief Affine transformation based estimator.

This estimator uses pairwise transformations estimated by matcher to estimate
final transformation for each camera.

@sa cv::detail::HomographyBasedEstimator
 */

```
        class CV_EXPORTS_W AffineBasedEstimator : public Estimator {
        public:
                CV_WRAP AffineBasedEstimator() {}


        private:
                virtual bool estimate(const std::vector<ImageFeatures> &features,
                                        const std::vector<MatchesInfo> &pairwise_matches,
                                        std::vector<CameraParams> &cameras);
        };
bool HomographyBasedEstimator::estimate(
                const std::vector<ImageFeatures> &features,
                const std::vector<MatchesInfo> &pairwise_matches,
                std::vector<CameraParams> &cameras) {
            LOGLN("Estimating rotations...");
#if ENABLE_LOG
            int64 t = getTickCount();
#endif

            const int num_images = static_cast<int>(features.size());

            if (!is_focals_estimated_) {
                // Estimate focal length and set it for all cameras
                std::vector<double> focals;
                cv::vi_detail::estimateFocal(features, pairwise_matches, focals);
                cameras.assign(num_images, CameraParams());
                for (int i = 0; i < num_images; ++i)cameras[i].focal = focals[i];
```

```
//                    printValue(cameras, "a22");
        } else {
            MATH21_ASSERT(0, "debug");
            for (int i = 0; i < num_images; ++i) {
                cameras[i].ppx -= 0.5 * features[i].img_size.width;
                cameras[i].ppy -= 0.5 * features[i].img_size.height;
            }
        }

        // Restore global motion
        Graph span_tree;
        std::vector<int> span_tree_centers;
        vi_detail::findMaxSpanningTree(num_images,       pairwise_matches,       span_tree,
span_tree_centers);
        span_tree.walkBreadthFirst(span_tree_centers[0],          CalcRotation(num_images,
pairwise_matches, cameras));

        // As calculations were performed under assumption that p.p. is in image center
        for (int i = 0; i < num_images; ++i) {
            cameras[i].ppx += 0.5 * features[i].img_size.width;
            cameras[i].ppy += 0.5 * features[i].img_size.height;
        }

        LOGLN("Estimating rotations, time: " << ((getTickCount() - t) / getTickFrequency()) << "
sec");

        return true;
    }
    struct CalcRotation {
        CalcRotation(int _num_images, const std::vector<MatchesInfo> &_pairwise_matches,
                     std::vector<CameraParams> &_cameras)
            : num_images(_num_images),  pairwise_matches(&_pairwise_matches[0]),
cameras(&_cameras[0]) {}

        void operator()(const GraphEdge &edge) {
            int pair_idx = edge.from * num_images + edge.to;

            Mat_<double> K_from = Mat::eye(3, 3, CV_64F);
            K_from(0, 0) = cameras[edge.from].focal;
            K_from(1, 1) = cameras[edge.from].focal * cameras[edge.from].aspect;
            K_from(0, 2) = cameras[edge.from].ppx;
            K_from(1, 2) = cameras[edge.from].ppy;
```

```cpp
                Mat_<double> K_to = Mat::eye(3, 3, CV_64F);
                K_to(0, 0) = cameras[edge.to].focal;
                K_to(1, 1) = cameras[edge.to].focal * cameras[edge.to].aspect;
                K_to(0, 2) = cameras[edge.to].ppx;
                K_to(1, 2) = cameras[edge.to].ppy;

                Mat R = K_from.inv() * pairwise_matches[pair_idx].H.inv() * K_to;
                cameras[edge.to].R = cameras[edge.from].R * R;
            }

        int num_images;
        const MatchesInfo *pairwise_matches;
        CameraParams *cameras;
    };
bool AffineBasedEstimator::estimate(const std::vector<ImageFeatures> &features,
                                    const             std::vector<MatchesInfo>
&pairwise_matches,
                                        std::vector<CameraParams> &cameras) {
        cameras.assign(features.size(), CameraParams());
        const int num_images = static_cast<int>(features.size());

        // find maximum spaning tree on pairwise matches
        Graph span_tree;
        std::vector<int> span_tree_centers;
        // uses number of inliers as weights
        vi_detail::findMaxSpanningTree(num_images, pairwise_matches, span_tree,
                                    span_tree_centers);

        // compute final transform by chaining H together
        span_tree.walkBreadthFirst(
                span_tree_centers[0],
                CalcAffineTransform(num_images, pairwise_matches, cameras));
        // this estimator never fails
        return true;
    }
/**
 * @brief Functor calculating final transformation by chaining linear transformations
 */
    struct CalcAffineTransform {
        CalcAffineTransform(int _num_images,
```

```cpp
                              const std::vector<MatchesInfo> &_pairwise_matches,
                              std::vector<CameraParams> &_cameras)
                : num_images(_num_images),  pairwise_matches(&_pairwise_matches[0]),
cameras(&_cameras[0]) {}

        void operator()(const GraphEdge &edge) {
            int pair_idx = edge.from * num_images + edge.to;
            cameras[edge.to].R = cameras[edge.from].R * pairwise_matches[pair_idx].H;
        }

        int num_images;
        const MatchesInfo *pairwise_matches;
        CameraParams *cameras;
    };
class Graph {
        public:
        Graph(int num_vertices = 0) { create(num_vertices); }

        void create(int num_vertices) { edges_.assign(num_vertices, std::list<GraphEdge>()); }

        int numVertices() const { return static_cast<int>(edges_.size()); }

        void addEdge(int from, int to, float weight);

        template<typename B>
        B forEach(B body) const;

        template<typename B>
        B walkBreadthFirst(int from, B body) const;

    private:
        std::vector<std::list<GraphEdge> > edges_;
    };
struct GraphEdge {
        GraphEdge(int from, int to, float weight);

        bool operator<(const GraphEdge &other) const { return weight < other.weight; }

        bool operator>(const GraphEdge &other) const { return weight > other.weight; }

        int from, to;
```

```cpp
        float weight;
    };
inline GraphEdge::GraphEdge(int _from, int _to, float _weight) : from(_from), to(_to), weight(_weight)
{}
void Graph::addEdge(int from, int to, float weight) {
        edges_[from].push_back(GraphEdge(from, to, weight));
    }
template<typename B>
    B Graph::forEach(B body) const {
        for (int i = 0; i < numVertices(); ++i) {
            std::list<GraphEdge>::const_iterator edge = edges_[i].begin();
            for (; edge != edges_[i].end(); ++edge)
                body(*edge);
        }
        return body;
    }


    template<typename B>
    B Graph::walkBreadthFirst(int from, B body) const {
        std::vector<bool> was(numVertices(), false);
        std::queue<int> vertices;

        was[from] = true;
        vertices.push(from);

        while (!vertices.empty()) {
            int vertex = vertices.front();
            vertices.pop();
            std::list<GraphEdge>::const_iterator edge = edges_[vertex].begin();
            for (; edge != edges_[vertex].end(); ++edge) {
                if (!was[edge->to]) {
                    body(*edge);
                    was[edge->to] = true;
                    vertices.push(edge->to);
                }
            }
        }
        return body;
    }
```

```
void focalsFromHomography(const Mat &H, double &f0, double &f1, bool &f0_ok, bool &f1_ok) {
         CV_Assert(H.type() == CV_64F && H.size() == Size(3, 3));

         const double *h = H.ptr<double>();

         double d1, d2; // Denominators
         double v1, v2; // Focal squares value candidates

         f1_ok = true;
         d1 = h[6] * h[7];
         d2 = (h[7] - h[6]) * (h[7] + h[6]);
         v1 = -(h[0] * h[1] + h[3] * h[4]) / d1;
         v2 = (h[0] * h[0] + h[3] * h[3] - h[1] * h[1] - h[4] * h[4]) / d2;
         if (v1 < v2) std::swap(v1, v2);
         if (v1 > 0 && v2 > 0) f1 = std::sqrt(std::abs(d1) > std::abs(d2) ? v1 : v2);
         else if (v1 > 0) f1 = std::sqrt(v1);
         else f1_ok = false;

         f0_ok = true;
         d1 = h[0] * h[3] + h[1] * h[4];
         d2 = h[0] * h[0] + h[1] * h[1] - h[3] * h[3] - h[4] * h[4];
         v1 = -h[2] * h[5] / d1;
         v2 = (h[5] * h[5] - h[2] * h[2]) / d2;
         if (v1 < v2) std::swap(v1, v2);
         if (v1 > 0 && v2 > 0) f0 = std::sqrt(std::abs(d1) > std::abs(d2) ? v1 : v2);
         else if (v1 > 0) f0 = std::sqrt(v1);
         else f0_ok = false;
     }


         void    estimateFocal(const    std::vector<ImageFeatures>    &features,    const
std::vector<MatchesInfo> &pairwise_matches,
                         std::vector<double> &focals) {
         const int num_images = static_cast<int>(features.size());
         focals.resize(num_images);

         std::vector<double> all_focals;

         for (int i = 0; i < num_images; ++i) {
             for (int j = 0; j < num_images; ++j) {
                 const MatchesInfo &m = pairwise_matches[i * num_images + j];
```

```
                if (m.H.empty())
                    continue;
                double f0, f1;
                bool f0ok, f1ok;
                focalsFromHomography(m.H, f0, f1, f0ok, f1ok);
                if (f0ok && f1ok)
                    all_focals.push_back(std::sqrt(f0 * f1));
            }
        }

        if (static_cast<int>(all_focals.size()) >= num_images - 1) {
            double median;

            std::sort(all_focals.begin(), all_focals.end());
            if (all_focals.size() % 2 == 1)
                median = all_focals[all_focals.size() / 2];
            else
                median = (all_focals[all_focals.size() / 2 - 1] + all_focals[all_focals.size() / 2])
* 0.5;

            for (int i = 0; i < num_images; ++i)
                focals[i] = median;
        } else {
            LOGLN("Can't estimate focal length, will use naive approach");
            double focals_sum = 0;
            for (int i = 0; i < num_images; ++i)
                focals_sum += features[i].img_size.width + features[i].img_size.height;
            for (int i = 0; i < num_images; ++i)
                focals[i] = focals_sum / num_images;
        }
    }
```

```cpp
void findMaxSpanningTree(int num_images, const std::vector<MatchesInfo> &pairwise_matches,
                         Graph &span_tree, std::vector<int> &centers) {
    Graph graph(num_images);
    std::vector<GraphEdge> edges;

    // Construct images graph and remember its edges
    for (int i = 0; i < num_images; ++i) {
        for (int j = 0; j < num_images; ++j) {
            if (pairwise_matches[i * num_images + j].H.empty())
                continue;
            float conf = static_cast<float>(pairwise_matches[i * num_images +
j].num_inliers);

            graph.addEdge(i, j, conf);
            edges.push_back(GraphEdge(i, j, conf));
        }
    }

    DisjointSets comps(num_images);
    span_tree.create(num_images);
    std::vector<int> span_tree_powers(num_images, 0);

    // Find maximum spanning tree
    sort(edges.begin(), edges.end(), std::greater<GraphEdge>());
    for (size_t i = 0; i < edges.size(); ++i) {
        int comp1 = comps.findSetByElem(edges[i].from);
        int comp2 = comps.findSetByElem(edges[i].to);
        if (comp1 != comp2) {
            comps.mergeSets(comp1, comp2);
            span_tree.addEdge(edges[i].from, edges[i].to, edges[i].weight);
            span_tree.addEdge(edges[i].to, edges[i].from, edges[i].weight);
            span_tree_powers[edges[i].from]++;
            span_tree_powers[edges[i].to]++;
        }
    }

    // Find spanning tree leafs
    std::vector<int> span_tree_leafs;
    for (int i = 0; i < num_images; ++i)
        if (span_tree_powers[i] == 1)
            span_tree_leafs.push_back(i);
```

```cpp
// Find maximum distance from each spanning tree vertex
std::vector<int> max_dists(num_images, 0);
std::vector<int> cur_dists;
for (size_t i = 0; i < span_tree_leafs.size(); ++i) {
    cur_dists.assign(num_images, 0);
    span_tree.walkBreadthFirst(span_tree_leafs[i], IncDistance(cur_dists));
    for (int j = 0; j < num_images; ++j)
        max_dists[j] = std::max(max_dists[j], cur_dists[j]);
}

// Find min-max distance
int min_max_dist = max_dists[0];
for (int i = 1; i < num_images; ++i)
    if (min_max_dist > max_dists[i])
        min_max_dist = max_dists[i];

// Find spanning tree centers
centers.clear();
for (int i = 0; i < num_images; ++i)
    if (max_dists[i] == min_max_dist)
        centers.push_back(i);
CV_Assert(centers.size() > 0 && centers.size() <= 2);
}
```