```
class CV_EXPORTS_W BundleAdjusterBase : public Estimator {
        public:
            CV_WRAP const Mat refinementMask() const { return refinement_mask_.clone(); }

            CV_WRAP void setRefinementMask(const Mat &mask) {
                CV_Assert(mask.type() == CV_8U && mask.size() == Size(3, 3));
                refinement_mask_ = mask.clone();
            }

            CV_WRAP double confThresh() const { return conf_thresh_; }

            CV_WRAP void setConfThresh(double conf_thresh) { conf_thresh_ = conf_thresh; }

            CV_WRAP TermCriteria termCriteria() { return term_criteria_; }

            CV_WRAP void setTermCriteria(const TermCriteria &term_criteria) { term_criteria_ =
term_criteria; }

        protected:
            /** @brief Construct a bundle adjuster base instance.

            @param num_params_per_cam Number of parameters per camera
            @param num_errs_per_measurement Number of error terms (components) per
match
             */
            BundleAdjusterBase(int num_params_per_cam, int num_errs_per_measurement)
                        : num_images_(0), total_num_matches_(0),
                          num_params_per_cam_(num_params_per_cam),
                          num_errs_per_measurement_(num_errs_per_measurement),
                          features_(0), pairwise_matches_(0), conf_thresh_(0) {
                setRefinementMask(Mat::ones(3, 3, CV_8U));
                setConfThresh(1.);
                setTermCriteria(TermCriteria(TermCriteria::EPS + TermCriteria::COUNT, 1000,
DBL_EPSILON));
            }

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

```
/** @brief Sets initial camera parameter to refine.

@param cameras Camera parameters
  */
virtual void setUpInitialCameraParams(const std::vector<CameraParams> &cameras) =
0;

/** @brief Gets the refined camera parameters.

@param cameras Refined camera parameters
  */
virtual void obtainRefinedCameraParams(std::vector<CameraParams> &cameras)
const = 0;

/** @brief Calculates error vector.

@param err Error column-vector of length total_num_matches \*
num_errs_per_measurement
  */
virtual void calcError(Mat &err) = 0;

/** @brief Calculates the cost function jacobian.

@param jac Jacobian matrix of dimensions
(total_num_matches \* num_errs_per_measurement) x (num_images \*
num_params_per_cam)
  */
virtual void calcJacobian(Mat &jac) = 0;

// 3x3 8U mask, where 0 means don't refine respective parameter, != 0 means refine
Mat refinement_mask_;

int num_images_;
int total_num_matches_;

int num_params_per_cam_;
int num_errs_per_measurement_;

const ImageFeatures *features_;
const MatchesInfo *pairwise_matches_;
```

```
        // Threshold to filter out poorly matched image pairs
        double conf_thresh_;

        //Levenberg-Marquardt algorithm termination criteria
        TermCriteria term_criteria_;

        // Camera parameters matrix (CV_64F)
        Mat cam_params_;

        // Connected images pairs
        std::vector<std::pair<int, int> > edges_;
    };
/** @brief Implementation of the camera parameters refinement algorithm which minimizes sum of
the reprojection
error squares

It can estimate focal length, aspect ratio, principal point.
You can affect only on them via the refinement mask.
 */
    class CV_EXPORTS_W BundleAdjusterReproj : public BundleAdjusterBase {
    public:
        CV_WRAP BundleAdjusterReproj() : BundleAdjusterBase(7, 2) {}

    private:
        void setUpInitialCameraParams(const std::vector<CameraParams> &cameras);

        void obtainRefinedCameraParams(std::vector<CameraParams> &cameras) const;

        void calcError(Mat &err);

        void calcJacobian(Mat &jac);

        Mat err1_, err2_;
    };
/** @brief Implementation of the camera parameters refinement algorithm which minimizes sum of
the distances
between the rays passing through the camera center and a feature. :

It can estimate focal length. It ignores the refinement mask for now.
 */
    class CV_EXPORTS_W BundleAdjusterRay : public BundleAdjusterBase {
```

```
    public:
        CV_WRAP BundleAdjusterRay() : BundleAdjusterBase(4, 3) {}

    private:
        void setUpInitialCameraParams(const std::vector<CameraParams> &cameras);

        void obtainRefinedCameraParams(std::vector<CameraParams> &cameras) const;

        void calcError(Mat &err);

        void calcJacobian(Mat &jac);

        Mat err1_, err2_;
    };
/** @brief Bundle adjuster that expects affine transformation
represented in homogeneous coordinates in R for each camera param. Implements
camera parameters refinement algorithm which minimizes sum of the reprojection
error squares

It estimates all transformation parameters. Refinement mask is ignored.

@sa AffineBasedEstimator AffineBestOf2NearestMatcher BundleAdjusterAffinePartial
 */
    class CV_EXPORTS_W BundleAdjusterAffine : public BundleAdjusterBase {
    public:
        CV_WRAP BundleAdjusterAffine() : BundleAdjusterBase(6, 2) {}

    private:
        void setUpInitialCameraParams(const std::vector<CameraParams> &cameras);

        void obtainRefinedCameraParams(std::vector<CameraParams> &cameras) const;

        void calcError(Mat &err);

        void calcJacobian(Mat &jac);

        Mat err1_, err2_;
    };
/** @brief Bundle adjuster that expects affine transformation with 4 DOF
represented in homogeneous coordinates in R for each camera param. Implements
camera parameters refinement algorithm which minimizes sum of the reprojection
```

error squares

It estimates all transformation parameters. Refinement mask is ignored.

@sa AffineBasedEstimator AffineBestOf2NearestMatcher BundleAdjusterAffine
 */

```
    class CV_EXPORTS_W BundleAdjusterAffinePartial : public BundleAdjusterBase {
    public:
        CV_WRAP BundleAdjusterAffinePartial() : BundleAdjusterBase(4, 2) {}

    private:
        void setUpInitialCameraParams(const std::vector<CameraParams> &cameras);

        void obtainRefinedCameraParams(std::vector<CameraParams> &cameras) const;

        void calcError(Mat &err);

        void calcJacobian(Mat &jac);

        Mat err1_, err2_;
    };
bool BundleAdjusterBase::estimate(const std::vector<ImageFeatures> &features,
                                  const std::vector<MatchesInfo> &pairwise_matches,
                                  std::vector<CameraParams> &cameras) {
#if USE_OPENCV_CVMAT
        LOG_CHAT("Bundle adjustment");
#if ENABLE_LOG
        int64 t = getTickCount();
#endif

        num_images_ = static_cast<int>(features.size());
        features_ = &features[0];
        pairwise_matches_ = &pairwise_matches[0];

        setUpInitialCameraParams(cameras);

        // Leave only consistent image pairs
        edges_.clear();
        for (int i = 0; i < num_images_ - 1; ++i) {
            for (int j = i + 1; j < num_images_; ++j) {
```

```
                const MatchesInfo &matches_info = pairwise_matches_[i * num_images_ +
j];
                if (matches_info.confidence > conf_thresh_)
                    edges_.push_back(std::make_pair(i, j));
            }
        }

        // Compute number of correspondences
        total_num_matches_ = 0;
        for (size_t i = 0; i < edges_.size(); ++i)
            total_num_matches_ += static_cast<int>(pairwise_matches[edges_[i].first *
num_images_ +

edges_[i].second].num_inliers);

        ASSERT(total_num_matches_, "No image left");
        CvLevMarq2 solver(num_images_ * num_params_per_cam_,
                          total_num_matches_ * num_errs_per_measurement_,
                          cvTermCriteria(term_criteria_));

        Mat err, jac;
        CvMat matParams = cvMat(cam_params_);
        cvCopy(&matParams, solver.param);

        int iter = 0;
        for (;;) {
            const CvMat *_param = 0;
            CvMat *_jac = 0;
            CvMat *_err = 0;

            bool proceed = solver.update(_param, _jac, _err);

            cvCopy(_param, &matParams);

            if (!proceed || !_err)
                break;

            if (_jac) {
                calcJacobian(jac);
                CvMat tmp = cvMat(jac);
                cvCopy(&tmp, _jac);
```

```
		}

		if (_err) {
			calcError(err);
			LOG_CHAT(".");
			iter++;
			CvMat tmp = cvMat(err);
			cvCopy(&tmp, _err);
		}
	}

	LOGLN_CHAT("");
	LOGLN_CHAT("Bundle adjustment, final RMS error: " << std::sqrt(err.dot(err) /
total_num_matches_));
	LOGLN_CHAT("Bundle adjustment, iterations done: " << iter);

	// Check if all camera parameters are valid
	bool ok = true;
	for (int i = 0; i < cam_params_.rows; ++i) {
		if (cvIsNaN(cam_params_.at<double>(i, 0))) {
			ok = false;
			break;
		}
	}
	if (!ok)
		return false;

	obtainRefinedCameraParams(cameras);

	// Normalize motion to center image
	Graph span_tree;
	std::vector<int> span_tree_centers;
	vi_detail::findMaxSpanningTree(num_images_, pairwise_matches, span_tree,
span_tree_centers);
	Mat R_inv = cameras[span_tree_centers[0]].R.inv();
	for (int i = 0; i < num_images_; ++i)
		cameras[i].R = R_inv * cameras[i].R;

	LOGLN_CHAT("Bundle adjustment, time: " << ((getTickCount() - t) / getTickFrequency())
<< " sec");
	return true;
```

```
#else
                ASSERT(0, "USE_OPENCV_CVMAT not enabled!");
                return false;
#endif
        }
void BundleAdjusterReproj::setUpInitialCameraParams(const std::vector<CameraParams> &cameras)
{
                cam_params_.create(num_images_ * 7, 1, CV_64F);
                SVD svd;
                for (int i = 0; i < num_images_; ++i) {
                    cam_params_.at<double>(i * 7, 0) = cameras[i].focal;
                    cam_params_.at<double>(i * 7 + 1, 0) = cameras[i].ppx;
                    cam_params_.at<double>(i * 7 + 2, 0) = cameras[i].ppy;
                    cam_params_.at<double>(i * 7 + 3, 0) = cameras[i].aspect;

                    svd(cameras[i].R, SVD::FULL_UV);
                    Mat R = svd.u * svd.vt;
                    if (determinant(R) < 0)
                        R *= -1;

                    Mat rvec;
                    Rodrigues(R, rvec);
                    CV_Assert(rvec.type() == CV_32F);
                    cam_params_.at<double>(i * 7 + 4, 0) = rvec.at<float>(0, 0);
                    cam_params_.at<double>(i * 7 + 5, 0) = rvec.at<float>(1, 0);
                    cam_params_.at<double>(i * 7 + 6, 0) = rvec.at<float>(2, 0);
                }
        }
void BundleAdjusterReproj::obtainRefinedCameraParams(std::vector<CameraParams> &cameras)
const {
                for (int i = 0; i < num_images_; ++i) {
                    cameras[i].focal = cam_params_.at<double>(i * 7, 0);
                    cameras[i].ppx = cam_params_.at<double>(i * 7 + 1, 0);
                    cameras[i].ppy = cam_params_.at<double>(i * 7 + 2, 0);
                    cameras[i].aspect = cam_params_.at<double>(i * 7 + 3, 0);

                    Mat rvec(3, 1, CV_64F);
                    rvec.at<double>(0, 0) = cam_params_.at<double>(i * 7 + 4, 0);
                    rvec.at<double>(1, 0) = cam_params_.at<double>(i * 7 + 5, 0);
                    rvec.at<double>(2, 0) = cam_params_.at<double>(i * 7 + 6, 0);
                    Rodrigues(rvec, cameras[i].R);
```

```
            Mat tmp;
            cameras[i].R.convertTo(tmp, CV_32F);
            cameras[i].R = tmp;
        }
    }
void BundleAdjusterReproj::calcError(Mat &err) {
        err.create(total_num_matches_ * 2, 1, CV_64F);

        int match_idx = 0;
        for (size_t edge_idx = 0; edge_idx < edges_.size(); ++edge_idx) {
            int i = edges_[edge_idx].first;
            int j = edges_[edge_idx].second;
            double f1 = cam_params_.at<double>(i * 7, 0);
            double f2 = cam_params_.at<double>(j * 7, 0);
            double ppx1 = cam_params_.at<double>(i * 7 + 1, 0);
            double ppx2 = cam_params_.at<double>(j * 7 + 1, 0);
            double ppy1 = cam_params_.at<double>(i * 7 + 2, 0);
            double ppy2 = cam_params_.at<double>(j * 7 + 2, 0);
            double a1 = cam_params_.at<double>(i * 7 + 3, 0);
            double a2 = cam_params_.at<double>(j * 7 + 3, 0);

            double R1[9];
            Mat R1_(3, 3, CV_64F, R1);
            Mat rvec(3, 1, CV_64F);
            rvec.at<double>(0, 0) = cam_params_.at<double>(i * 7 + 4, 0);
            rvec.at<double>(1, 0) = cam_params_.at<double>(i * 7 + 5, 0);
            rvec.at<double>(2, 0) = cam_params_.at<double>(i * 7 + 6, 0);
            Rodrigues(rvec, R1_);

            double R2[9];
            Mat R2_(3, 3, CV_64F, R2);
            rvec.at<double>(0, 0) = cam_params_.at<double>(j * 7 + 4, 0);
            rvec.at<double>(1, 0) = cam_params_.at<double>(j * 7 + 5, 0);
            rvec.at<double>(2, 0) = cam_params_.at<double>(j * 7 + 6, 0);
            Rodrigues(rvec, R2_);

            const ImageFeatures &features1_d = features_[i];
            const ImageFeatures &features2_d = features_[j];
            const MatchesInfo &matches_info = pairwise_matches_[i * num_images_ + j];
```

```
            Mat_<double> K1 = Mat::eye(3, 3, CV_64F);
            K1(0, 0) = f1;
            K1(0, 2) = ppx1;
            K1(1, 1) = f1 * a1;
            K1(1, 2) = ppy1;

            Mat_<double> K2 = Mat::eye(3, 3, CV_64F);
            K2(0, 0) = f2;
            K2(0, 2) = ppx2;
            K2(1, 1) = f2 * a2;
            K2(1, 2) = ppy2;

            std::vector<KeyPoint> keypoints1 = features1_d.keypoints;
            std::vector<KeyPoint> keypoints2 = features2_d.keypoints;
            Mat_<double> H = K2 * R2_.inv() * R1_ * K1.inv();

            for (size_t k = 0; k < matches_info.matches.size(); ++k) {
                if (!matches_info.inliers_mask[k])
                    continue;

                const DMatch &m = matches_info.matches[k];
                Point2f p1 = keypoints1[m.queryIdx].pt;
                Point2f p2 = keypoints2[m.trainIdx].pt;
                double x = H(0, 0) * p1.x + H(0, 1) * p1.y + H(0, 2);
                double y = H(1, 0) * p1.x + H(1, 1) * p1.y + H(1, 2);
                double z = H(2, 0) * p1.x + H(2, 1) * p1.y + H(2, 2);

                err.at<double>(2 * match_idx, 0) = p2.x - x / z;
                err.at<double>(2 * match_idx + 1, 0) = p2.y - y / z;
                match_idx++;
            }
        }
    }
void BundleAdjusterReproj::calcJacobian(Mat &jac) {
        jac.create(total_num_matches_ * 2, num_images_ * 7, CV_64F);
        jac.setTo(0);

        double val;
        const double step = 1e-4;

        for (int i = 0; i < num_images_; ++i) {
```

```
if (refinement_mask_.at<uchar>(0, 0)) {
        val = cam_params_.at<double>(i * 7, 0);
        cam_params_.at<double>(i * 7, 0) = val - step;
        calcError(err1_);
        cam_params_.at<double>(i * 7, 0) = val + step;
        calcError(err2_);
        calcDeriv(err1_, err2_, 2 * step, jac.col(i * 7));
        cam_params_.at<double>(i * 7, 0) = val;
}
if (refinement_mask_.at<uchar>(0, 2)) {
        val = cam_params_.at<double>(i * 7 + 1, 0);
        cam_params_.at<double>(i * 7 + 1, 0) = val - step;
        calcError(err1_);
        cam_params_.at<double>(i * 7 + 1, 0) = val + step;
        calcError(err2_);
        calcDeriv(err1_, err2_, 2 * step, jac.col(i * 7 + 1));
        cam_params_.at<double>(i * 7 + 1, 0) = val;
}
if (refinement_mask_.at<uchar>(1, 2)) {
        val = cam_params_.at<double>(i * 7 + 2, 0);
        cam_params_.at<double>(i * 7 + 2, 0) = val - step;
        calcError(err1_);
        cam_params_.at<double>(i * 7 + 2, 0) = val + step;
        calcError(err2_);
        calcDeriv(err1_, err2_, 2 * step, jac.col(i * 7 + 2));
        cam_params_.at<double>(i * 7 + 2, 0) = val;
}
if (refinement_mask_.at<uchar>(1, 1)) {
        val = cam_params_.at<double>(i * 7 + 3, 0);
        cam_params_.at<double>(i * 7 + 3, 0) = val - step;
        calcError(err1_);
        cam_params_.at<double>(i * 7 + 3, 0) = val + step;
        calcError(err2_);
        calcDeriv(err1_, err2_, 2 * step, jac.col(i * 7 + 3));
        cam_params_.at<double>(i * 7 + 3, 0) = val;
}
for (int j = 4; j < 7; ++j) {
        val = cam_params_.at<double>(i * 7 + j, 0);
        cam_params_.at<double>(i * 7 + j, 0) = val - step;
        calcError(err1_);
        cam_params_.at<double>(i * 7 + j, 0) = val + step;
```

```
                    calcError(err2_);
                    calcDeriv(err1_, err2_, 2 * step, jac.col(i * 7 + j));
                    cam_params_.at<double>(i * 7 + j, 0) = val;
                }
            }
        }
void BundleAdjusterRay::setUpInitialCameraParams(const std::vector<CameraParams> &cameras) {
            cam_params_.create(num_images_ * 4, 1, CV_64F);
            SVD svd;
            for (int i = 0; i < num_images_; ++i) {
                cam_params_.at<double>(i * 4, 0) = cameras[i].focal;

                svd(cameras[i].R, SVD::FULL_UV);
                Mat R = svd.u * svd.vt;
                if (determinant(R) < 0)
                    R *= -1;

                Mat rvec;
                Rodrigues(R, rvec);
                CV_Assert(rvec.type() == CV_32F);
                cam_params_.at<double>(i * 4 + 1, 0) = rvec.at<float>(0, 0);
                cam_params_.at<double>(i * 4 + 2, 0) = rvec.at<float>(1, 0);
                cam_params_.at<double>(i * 4 + 3, 0) = rvec.at<float>(2, 0);
            }
        }
void BundleAdjusterRay::obtainRefinedCameraParams(std::vector<CameraParams> &cameras) const
{
            for (int i = 0; i < num_images_; ++i) {
                cameras[i].focal = cam_params_.at<double>(i * 4, 0);

                Mat rvec(3, 1, CV_64F);
                rvec.at<double>(0, 0) = cam_params_.at<double>(i * 4 + 1, 0);
                rvec.at<double>(1, 0) = cam_params_.at<double>(i * 4 + 2, 0);
                rvec.at<double>(2, 0) = cam_params_.at<double>(i * 4 + 3, 0);
                Rodrigues(rvec, cameras[i].R);

                Mat tmp;
                cameras[i].R.convertTo(tmp, CV_32F);
                cameras[i].R = tmp;
            }
        }
```

```
void BundleAdjusterRay::calcError(Mat &err) {
        err.create(total_num_matches_ * 3, 1, CV_64F);

        int match_idx = 0;
        for (size_t edge_idx = 0; edge_idx < edges_.size(); ++edge_idx) {
            int i = edges_[edge_idx].first;
            int j = edges_[edge_idx].second;
            double f1 = cam_params_.at<double>(i * 4, 0);
            double f2 = cam_params_.at<double>(j * 4, 0);

            double R1[9];
            Mat R1_(3, 3, CV_64F, R1);
            Mat rvec(3, 1, CV_64F);
            rvec.at<double>(0, 0) = cam_params_.at<double>(i * 4 + 1, 0);
            rvec.at<double>(1, 0) = cam_params_.at<double>(i * 4 + 2, 0);
            rvec.at<double>(2, 0) = cam_params_.at<double>(i * 4 + 3, 0);
            Rodrigues(rvec, R1_);

            double R2[9];
            Mat R2_(3, 3, CV_64F, R2);
            rvec.at<double>(0, 0) = cam_params_.at<double>(j * 4 + 1, 0);
            rvec.at<double>(1, 0) = cam_params_.at<double>(j * 4 + 2, 0);
            rvec.at<double>(2, 0) = cam_params_.at<double>(j * 4 + 3, 0);
            Rodrigues(rvec, R2_);

            const ImageFeatures &features1 = features_[i];
            const ImageFeatures &features2 = features_[j];
            const MatchesInfo &matches_info = pairwise_matches_[i * num_images_ + j];

            Mat_<double> K1 = Mat::eye(3, 3, CV_64F);
            K1(0, 0) = f1;
            K1(0, 2) = features1.img_size.width * 0.5;
            K1(1, 1) = f1;
            K1(1, 2) = features1.img_size.height * 0.5;

            Mat_<double> K2 = Mat::eye(3, 3, CV_64F);
            K2(0, 0) = f2;
            K2(0, 2) = features2.img_size.width * 0.5;
            K2(1, 1) = f2;
            K2(1, 2) = features2.img_size.height * 0.5;
```

```
Mat_<double> H1 = R1_ * K1.inv();
Mat_<double> H2 = R2_ * K2.inv();

for (size_t k = 0; k < matches_info.matches.size(); ++k) {
    if (!matches_info.inliers_mask[k])
        continue;

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

    Point2f p1 = features1.keypoints[m.queryIdx].pt;
    double x1 = H1(0, 0) * p1.x + H1(0, 1) * p1.y + H1(0, 2);
    double y1 = H1(1, 0) * p1.x + H1(1, 1) * p1.y + H1(1, 2);
    double z1 = H1(2, 0) * p1.x + H1(2, 1) * p1.y + H1(2, 2);
    double len = std::sqrt(x1 * x1 + y1 * y1 + z1 * z1);
    x1 /= len;
    y1 /= len;
    z1 /= len;

    Point2f p2 = features2.keypoints[m.trainIdx].pt;
    double x2 = H2(0, 0) * p2.x + H2(0, 1) * p2.y + H2(0, 2);
    double y2 = H2(1, 0) * p2.x + H2(1, 1) * p2.y + H2(1, 2);
    double z2 = H2(2, 0) * p2.x + H2(2, 1) * p2.y + H2(2, 2);
    len = std::sqrt(x2 * x2 + y2 * y2 + z2 * z2);
    x2 /= len;
    y2 /= len;
    z2 /= len;

    double mult = std::sqrt(f1 * f2);
    err.at<double>(3 * match_idx, 0) = mult * (x1 - x2);
    err.at<double>(3 * match_idx + 1, 0) = mult * (y1 - y2);
    err.at<double>(3 * match_idx + 2, 0) = mult * (z1 - z2);

    match_idx++;
}
}
}
void BundleAdjusterRay::calcJacobian(Mat &jac) {
    jac.create(total_num_matches_ * 3, num_images_ * 4, CV_64F);

    double val;
    const double step = 1e-3;
```

```
            for (int i = 0; i < num_images_; ++i) {
                for (int j = 0; j < 4; ++j) {
                    val = cam_params_.at<double>(i * 4 + j, 0);
                    cam_params_.at<double>(i * 4 + j, 0) = val - step;
                    calcError(err1_);
                    cam_params_.at<double>(i * 4 + j, 0) = val + step;
                    calcError(err2_);
                    calcDeriv(err1_, err2_, 2 * step, jac.col(i * 4 + j));
                    cam_params_.at<double>(i * 4 + j, 0) = val;
                }
            }
        }
void BundleAdjusterAffine::setUpInitialCameraParams(const std::vector<CameraParams> &cameras)
{
            cam_params_.create(num_images_ * 6, 1, CV_64F);
            for (size_t i = 0; i < static_cast<size_t>(num_images_); ++i) {
                CV_Assert(cameras[i].R.type() == CV_32F);
                // cameras[i].R is
                //      a b tx
                //      c d ty
                //      0 0 1. (optional)
                // cam_params_ model for LevMarq is
                //      (a, b, tx, c, d, ty)
                Mat params(2, 3, CV_64F, cam_params_.ptr<double>() + i * 6);
                cameras[i].R.rowRange(0, 2).convertTo(params, CV_64F);
            }
        }
void BundleAdjusterAffine::obtainRefinedCameraParams(std::vector<CameraParams> &cameras)
const {
            for (int i = 0; i < num_images_; ++i) {
                // cameras[i].R will be
                //      a b tx
                //      c d ty
                //      0 0 1
                cameras[i].R = Mat::eye(3, 3, CV_32F);
                Mat params = cam_params_.rowRange(i * 6, i * 6 + 6).reshape(1, 2);
                params.convertTo(cameras[i].R.rowRange(0, 2), CV_32F);
            }
        }
void BundleAdjusterAffine::calcError(Mat &err) {
```

```
err.create(total_num_matches_ * 2, 1, CV_64F);

int match_idx = 0;
for (size_t edge_idx = 0; edge_idx < edges_.size(); ++edge_idx) {
    size_t i = edges_[edge_idx].first;
    size_t j = edges_[edge_idx].second;

    const ImageFeatures &features1 = features_[i];
    const ImageFeatures &features2 = features_[j];
    const MatchesInfo &matches_info = pairwise_matches_[i * num_images_ + j];

    Mat H1(2, 3, CV_64F, cam_params_.ptr<double>() + i * 6);
    Mat H2(2, 3, CV_64F, cam_params_.ptr<double>() + j * 6);

    // invert H1
    Mat H1_inv;
    invertAffineTransform(H1, H1_inv);

    // convert to representation in homogeneous coordinates
    Mat last_row = Mat::zeros(1, 3, CV_64F);
    last_row.at<double>(2) = 1.;
    H1_inv.push_back(last_row);
    H2.push_back(last_row);

    Mat_<double> H = H1_inv * H2;

    for (size_t k = 0; k < matches_info.matches.size(); ++k) {
        if (!matches_info.inliers_mask[k])
            continue;

        const DMatch &m = matches_info.matches[k];
        const Point2f &p1 = features1.keypoints[m.queryIdx].pt;
        const Point2f &p2 = features2.keypoints[m.trainIdx].pt;

        double x = H(0, 0) * p1.x + H(0, 1) * p1.y + H(0, 2);
        double y = H(1, 0) * p1.x + H(1, 1) * p1.y + H(1, 2);

        err.at<double>(2 * match_idx + 0, 0) = p2.x - x;
        err.at<double>(2 * match_idx + 1, 0) = p2.y - y;

        ++match_idx;
```

```
                }
            }
        }
void BundleAdjusterAffine::calcJacobian(Mat &jac) {
            jac.create(total_num_matches_ * 2, num_images_ * 6, CV_64F);

            double val;
            const double step = 1e-4;

            for (int i = 0; i < num_images_; ++i) {
                for (int j = 0; j < 6; ++j) {
                    val = cam_params_.at<double>(i * 6 + j, 0);
                    cam_params_.at<double>(i * 6 + j, 0) = val - step;
                    calcError(err1_);
                    cam_params_.at<double>(i * 6 + j, 0) = val + step;
                    calcError(err2_);
                    calcDeriv(err1_, err2_, 2 * step, jac.col(i * 6 + j));
                    cam_params_.at<double>(i * 6 + j, 0) = val;
                }
            }
        }
void BundleAdjusterAffinePartial::setUpInitialCameraParams(const std::vector<CameraParams>
&cameras) {
            cam_params_.create(num_images_ * 4, 1, CV_64F);
            for (size_t i = 0; i < static_cast<size_t>(num_images_); ++i) {
                CV_Assert(cameras[i].R.type() == CV_32F);
                // cameras[i].R is
                //      a -b tx
                //      b  a ty
                //      0  0 1. (optional)
                // cam_params_ model for LevMarq is
                //      (a, b, tx, ty)
                double *params = cam_params_.ptr<double>() + i * 4;
                params[0] = cameras[i].R.at<float>(0, 0);
                params[1] = cameras[i].R.at<float>(1, 0);
                params[2] = cameras[i].R.at<float>(0, 2);
                params[3] = cameras[i].R.at<float>(1, 2);
            }
        }
void BundleAdjusterAffinePartial::obtainRefinedCameraParams(std::vector<CameraParams>
&cameras) const {
```

```
			for (size_t i = 0; i < static_cast<size_t>(num_images_); ++i) {
				// cameras[i].R will be
				//	a -b tx
				//	b   a ty
				//	0   0 1
				// cam_params_ model for LevMarq is
				//	(a, b, tx, ty)
				const double *params = cam_params_.ptr<double>() + i * 4;
				double transform_buf[9] =
					{
						params[0], -params[1], params[2],
						params[1], params[0], params[3],
						0., 0., 1.
					};
				Mat transform(3, 3, CV_64F, transform_buf);
				transform.convertTo(cameras[i].R, CV_32F);
			}
		}
void BundleAdjusterAffinePartial::calcError(Mat &err) {
			err.create(total_num_matches_ * 2, 1, CV_64F);

			int match_idx = 0;
			for (size_t edge_idx = 0; edge_idx < edges_.size(); ++edge_idx) {
				size_t i = edges_[edge_idx].first;
				size_t j = edges_[edge_idx].second;

				const ImageFeatures &features1 = features_[i];
				const ImageFeatures &features2 = features_[j];
				const MatchesInfo &matches_info = pairwise_matches_[i * num_images_ + j];

				const double *H1_ptr = cam_params_.ptr<double>() + i * 4;
				double H1_buf[9] =
					{
						H1_ptr[0], -H1_ptr[1], H1_ptr[2],
						H1_ptr[1], H1_ptr[0], H1_ptr[3],
						0., 0., 1.
					};
				Mat H1(3, 3, CV_64F, H1_buf);
				const double *H2_ptr = cam_params_.ptr<double>() + j * 4;
				double H2_buf[9] =
					{
```

```
                        H2_ptr[0], -H2_ptr[1], H2_ptr[2],
                        H2_ptr[1], H2_ptr[0], H2_ptr[3],
                        0., 0., 1.
                };
            Mat H2(3, 3, CV_64F, H2_buf);

            // invert H1
            Mat H1_aff(H1, Range(0, 2));
            double H1_inv_buf[6];
            Mat H1_inv(2, 3, CV_64F, H1_inv_buf);
            invertAffineTransform(H1_aff, H1_inv);
            H1_inv.copyTo(H1_aff);

            Mat_<double> H = H1 * H2;

            for (size_t k = 0; k < matches_info.matches.size(); ++k) {
                if (!matches_info.inliers_mask[k])
                    continue;

                const DMatch &m = matches_info.matches[k];
                const Point2f &p1 = features1.keypoints[m.queryIdx].pt;
                const Point2f &p2 = features2.keypoints[m.trainIdx].pt;

                double x = H(0, 0) * p1.x + H(0, 1) * p1.y + H(0, 2);
                double y = H(1, 0) * p1.x + H(1, 1) * p1.y + H(1, 2);

                err.at<double>(2 * match_idx + 0, 0) = p2.x - x;
                err.at<double>(2 * match_idx + 1, 0) = p2.y - y;

                ++match_idx;
            }
        }
    }
void BundleAdjusterAffinePartial::calcJacobian(Mat &jac) {
        jac.create(total_num_matches_ * 2, num_images_ * 4, CV_64F);

        double val;
        const double step = 1e-4;

        for (int i = 0; i < num_images_; ++i) {
            for (int j = 0; j < 4; ++j) {
```

```
            val = cam_params_.at<double>(i * 4 + j, 0);
            cam_params_.at<double>(i * 4 + j, 0) = val - step;
            calcError(err1_);
            cam_params_.at<double>(i * 4 + j, 0) = val + step;
            calcError(err2_);
            calcDeriv(err1_, err2_, 2 * step, jac.col(i * 4 + j));
            cam_params_.at<double>(i * 4 + j, 0) = val;
        }
    }
}
```