```
void waveCorrect(std::vector<Mat> &rmats, WaveCorrectKind kind) {
            LOGLN("Wave correcting...");
#if ENABLE_LOG
            int64 t = getTickCount();
#endif
            if (rmats.size() <= 1) {
                LOGLN("Wave correcting, time: " << ((getTickCount() - t) / getTickFrequency()) <<
" sec");
                return;
            }

            if (kind == WAVE_CORRECT_AUTO) {
                kind = autoDetectWaveCorrectKind(rmats);
            }

            Mat moment = Mat::zeros(3, 3, CV_32F);
            for (size_t i = 0; i < rmats.size(); ++i) {
                Mat col = rmats[i].col(0);
                moment += col * col.t();
            }

            Mat eigen_vals, eigen_vecs;
            eigen(moment, eigen_vals, eigen_vecs);

            Mat rg1;
            if (kind == WAVE_CORRECT_HORIZ)
                rg1 = eigen_vecs.row(2).t();
            else if (kind == WAVE_CORRECT_VERT)
                rg1 = eigen_vecs.row(0).t();
            else
                CV_Error(CV_StsBadArg, "unsupported kind of wave correction");

            Mat img_k = Mat::zeros(3, 1, CV_32F);
            for (size_t i = 0; i < rmats.size(); ++i)
                img_k += rmats[i].col(2);
            Mat rg0 = rg1.cross(img_k);
            double rg0_norm = norm(rg0);

            if (rg0_norm <= DBL_MIN) {
                return;
            }
```

```
                rg0 /= rg0_norm;

                Mat rg2 = rg0.cross(rg1);

                double conf = 0;
                if (kind == WAVE_CORRECT_HORIZ) {
                        for (size_t i = 0; i < rmats.size(); ++i)
                                conf += rg0.dot(rmats[i].col(0));
                        if (conf < 0) {
                                rg0 *= -1;
                                rg1 *= -1;
                        }
                } else if (kind == WAVE_CORRECT_VERT) {
                        for (size_t i = 0; i < rmats.size(); ++i)
                                conf -= rg1.dot(rmats[i].col(0));
                        if (conf < 0) {
                                rg0 *= -1;
                                rg1 *= -1;
                        }
                }

                Mat R = Mat::zeros(3, 3, CV_32F);
                Mat tmp = R.row(0);
                Mat(rg0.t()).copyTo(tmp);
                tmp = R.row(1);
                Mat(rg1.t()).copyTo(tmp);
                tmp = R.row(2);
                Mat(rg2.t()).copyTo(tmp);

                for (size_t i = 0; i < rmats.size(); ++i)
                        rmats[i] = R * rmats[i];

                LOGLN("Wave correcting, time: " << ((getTickCount() - t) / getTickFrequency()) << "
sec");
        }
WaveCorrectKind autoDetectWaveCorrectKind(const std::vector<Mat> &rmats) {
                std::vector<float> xs, ys;
                xs.reserve(rmats.size());
                ys.reserve(rmats.size());
```

```
            // Project a [0, 0, 1, 1] point to the camera image frame
            // Ignore intrinsic parameters and camera translation as they
            // have little influence
            // This also means we can simply use "rmat.col(2)" as the
            // projected point homogeneous coordinate
            for (const Mat &rmat: rmats) {
                CV_Assert(rmat.type() == CV_32F);
                xs.push_back(rmat.at<float>(0, 2) / rmat.at<float>(2, 2));
                ys.push_back(rmat.at<float>(1, 2) / rmat.at<float>(2, 2));
            }

            // Calculate the delta between the max and min values for
            // both the X and Y axis
            auto min_max_x = std::minmax_element(xs.begin(), xs.end());
            auto min_max_y = std::minmax_element(ys.begin(), ys.end());
            double delta_x = *min_max_x.second - *min_max_x.first;
            double delta_y = *min_max_y.second - *min_max_y.first;

            // If the Y delta is the biggest, it means the images
            // mostly span along the vertical axis: correct this axis
            if (delta_y > delta_x) {
                LOGLN("    using vertical wave correction");
                return WAVE_CORRECT_VERT;
            } else {
                LOGLN("    using horizontal wave correction");
                return WAVE_CORRECT_HORIZ;
            }
        }
```

The function cv::eigen calculates just eigenvalues, or eigenvalues and eigenvectors of the symmetric matrix src:

```
@code
    src*eigenvectors.row(i).t() = eigenvalues.at<srcType>(i)*eigenvectors.row(i).t()
@endcode
```

@note in the new and the old interfaces different ordering of eigenvalues and eigenvectors parameters is used.

@param src input matrix that must have CV_32FC1 or CV_64FC1 type, square size and be symmetrical
(src ^T^ == src).

@param eigenvalues output vector of eigenvalues of the same type as src; the eigenvalues are stored in the descending order.

@param eigenvectors output matrix of eigenvectors; it has the same size and type as src; the

eigenvectors are stored as subsequent matrix rows, in the same order as the corresponding eigenvalues.
@sa completeSymm , PCA
*/

```
CV_EXPORTS_W bool eigen(InputArray src, OutputArray eigenvalues,
                        OutputArray eigenvectors = noArray());
```