```
#include "Labeling.h"
#include "SlantCorrector.h"

//! コンストラクタ
SlantCorrector::SlantCorrector(
 const cv::Mat& imgGray)                //!> 入力画像
    :   m_slant(DBL_MAX),
        m_imgCorrected()
{
    m_slant = detectSlant(imgGray);

    if (m_slant != DBL_MAX) {
        m_imgCorrected = correctsSlant(imgGray, m_slant);
    }
}

//! デストラクタ
SlantCorrector::~SlantCorrector()
{
}

//! 傾き角度取得
double SlantCorrector::slant() const
{
    return m_slant;
}

//! 傾き補正画像取得
cv::Mat SlantCorrector::correctedImage() const
{
    return m_imgCorrected;
}

//! 傾き検出
double SlantCorrector::detectSlant(
 const cv::Mat& imgGray) const      //!> 入力画像
{
    //二値化
    cv::Mat imgBin;
//  cv::adaptiveThreshold(imgGray, imgBin, 255, CV_ADAPTIVE_THRESH_GAUSSIAN_C, CV_THRESH_BINARY_INV, 7, 32);
    cv::threshold(imgGray, imgBin, 0, 255, CV_THRESH_BINARY_INV | CV_THRESH_OTSU);

#if 0
    cv::namedWindow("二値化", CV_WINDOW_AUTOSIZE|CV_WINDOW_FREERATIO);
    cv::imshow("二値化", imgBin);
    cv::waitKey(0);
    cv::destroyWindow("二値化");
```

```
#endif

    //二値化した画像に対してラベリングを実行
    cv::Mat label(imgBin.size(), CV_16SC1); //ラベリング結果
    LabelingBS  labeling;
    labeling.Exec(imgBin.data, (short *)label.data, imgBin.cols, imgBin.rows, false, 0);

#if 1   //ラベリング結果を表示
{
    cv::Mat imgDebug;
    cv::cvtColor(imgGray, imgDebug, CV_GRAY2RGB);

    //ラベリング結果を描画
    for (int i = 0; i < labeling.GetNumOfRegions(); i++) {
        RegionInfoBS* info = labeling.GetResultRegionInfo(i);

        cv::Point p1;           //矩形の左上座標
        info->GetMin(p1.x, p1.y);
        cv::Point p2;           //矩形の右下座標
        info->GetMax(p2.x, p2.y);

        cv::rectangle(imgDebug, p1, p2, CV_RGB(255, 0, 0));
    }

    cv::namedWindow("ラベリング結果", CV_WINDOW_AUTOSIZE|CV_WINDOW_FREERATIO);
    cv::imshow("ラベリング結果", imgDebug);
    cv::waitKey(0);
    cv::destroyWindow("ラベリング結果");
}
#endif

    //各領域の下辺中央の点を二値画像に描画
    cv::Mat tmp = cv::Mat::zeros(imgBin.size(), CV_8UC1);
    for (int i = 0; i < labeling.GetNumOfRegions(); i++) {
        RegionInfoBS* info = labeling.GetResultRegionInfo(i);

        cv::Point p1;           //矩形の左上座標
        info->GetMin(p1.x, p1.y);
        cv::Point p2;           //矩形の右下座標
        info->GetMax(p2.x, p2.y);

        //下辺中央の点を描画
        cv::line(tmp, cv::Point((p1.x + p2.x) / 2, p2.y), cv::Point((p1.x + p2.x) / 2, p2.y), 255);
    }

#if 1   //各領域の下辺中央の点を表示
    cv::namedWindow("下辺中央の点", CV_WINDOW_AUTOSIZE|CV_WINDOW_FREERATIO);
```

```
    cv::imshow("下辺中央の点", tmp);
    cv::waitKey(0);
    cv::destroyWindow("下辺中央の点");
#endif

    //閾値を徐々に厳しくしながら直線検出
    //変動係数が一定以下になったら打ち切り
    double slant = DBL_MAX;
    std::vector<cv::Vec2f> lines;

    for (int thre = 2; thre < labeling.GetNumOfRegions(); thre++) {
        //直線検出
        cv::HoughLines(tmp, lines, 1, CV_PI/180/10, thre);
        if (lines.empty()) {
            return DBL_MAX;     //傾き検出不可
        }

        //thetaの平均を求める
        float ave = 0.0;
        for (size_t i = 0; i < lines.size(); i++) {
            ave += lines[i][1];
        }
        ave = ave / lines.size();

        //thetaの標準偏差を求める
        float sigma = 0.0;
        for (size_t i = 0; i < lines.size(); i++) {
            sigma += pow(lines[i][1] - ave, (float)2.0);
        }
        sigma = sqrt(sigma / lines.size());

        //変動係数を求める
        float cv = sigma / ave;
        printf("thre:%d, 変動係数:%f, 平均角度:%f度, 線の数:%d, 領域数:%d¥n", thre, cv, ave * 180.0 / CV_PI, lines.size(),
labeling.GetNumOfRegions());

        //変動係数が閾値未満になったらthetaの平均を傾きとする
        if (cv < 0.002) {
            slant = ave;
            break;
        }
    }

#if 1   //傾き検出結果を表示
{
    cv::Mat imgDebug;
    cv::cvtColor(imgGray, imgDebug, CV_GRAY2RGB);
```

```
    //直線検出結果を描画
    for (size_t i = 0; i < lines.size(); i++) {
        float rho = lines[i][0];
        float theta = lines[i][1];
        double a = cos(theta);
        double b = sin(theta);
        double x0 = a*rho;
        double y0 = b*rho;
        cv::Point pt1(  cvRound(x0 + imgDebug.cols*(-b)),
                        cvRound(y0 + imgDebug.cols*(a)));
        cv::Point pt2(  cvRound(x0 - imgDebug.cols*(-b)),
                        cvRound(y0 - imgDebug.cols*(a)));
        cv::line(imgDebug, pt1, pt2, 255, 1, 8);
    }

    cv::namedWindow("傾き検出", CV_WINDOW_AUTOSIZE|CV_WINDOW_FREERATIO);
    cv::imshow("傾き検出", imgDebug);
    cv::waitKey(0);
    cv::destroyWindow("傾き検出");
}
#endif

    return slant;
}

//! 傾き補正
cv::Mat SlantCorrector::correctsSlant(
 const cv::Mat& imgGray,                //!> 入力画像
 double slant) const                    //!> 傾き角度
{
    //元画像をコピー
    cv::Mat imgCorrectedSlant = imgGray.clone();

    //傾き角度があれば回転して補正
    if (slant != 0.0) {
        //回転角[deg]
        double angle = (slant * 180 / CV_PI) - 90.0;
        //回転中心
        cv::Point2f center((float)(imgGray.cols * 0.5), (float)(imgGray.rows * 0.5));

        // 以上の条件から2次元の回転行列を計算
        cv::Mat affine_matrix = cv::getRotationMatrix2D(center, angle, 1.0);

        //回転
        cv::warpAffine(imgGray, imgCorrectedSlant, affine_matrix, imgGray.size(), cv::INTER_CUBIC, cv::BORDER_CONSTANT,
cv::Scalar(255,255,255));
```

```
    }

#if 1   //傾き補正結果を表示
    cv::namedWindow("傾き補正結果", CV_WINDOW_AUTOSIZE|CV_WINDOW_FREERATIO);
    cv::imshow("傾き補正結果", imgCorrectedSlant);
    cv::waitKey(0);
    cv::destroyWindow("傾き補正結果");
#endif

    return imgCorrectedSlant;
}
```

```
#include "Labeling.h"
#include "TRange.h"
#include "LineDetector.h"

//! コンストラクタ
LineDetector::LineDetector(
 const cv::Mat& imgGray)                    //!> 入力画像
    :   m_lines()
{
    //行認識
    m_lines = getLineRects(imgGray);
}

//! デストラクタ
LineDetector::~LineDetector()
{
}

//! 行取得
std::vector<cv::Rect> LineDetector::lines()
{
    return m_lines;
}

//! 行認識
std::vector<cv::Rect> LineDetector::getLineRects(
 const cv::Mat& imgGray) const              //!> 入力画像
{
    std::vector<cv::Rect>   rects;

    //二値化
    cv::Mat imgBin;
//  cv::adaptiveThreshold(imgGray, imgBin, 255, CV_ADAPTIVE_THRESH_GAUSSIAN_C, CV_THRESH_BINARY_INV, 7, 32);
    cv::threshold(imgGray, imgBin, 0, 255, CV_THRESH_BINARY_INV | CV_THRESH_OTSU);

#if 0
    cv::namedWindow("二値化", CV_WINDOW_AUTOSIZE|CV_WINDOW_FREERATIO);
    cv::imshow("二値化", imgBin);
    cv::waitKey(0);
    cv::destroyWindow("二値化");
#endif

    //二値化した画像に対してラベリングを実行する
    cv::Mat label(imgBin.size(), CV_16SC1); //ラベリング結果
    LabelingBS  labeling;
    labeling.Exec(imgBin.data, (short *)label.data, imgBin.cols, imgBin.rows, false, 0);
```

```
    //ラベリング結果をcv::Rectのvectorにコピー
    for (int i = 0; i < labeling.GetNumOfRegions(); i++) {
        RegionInfoBS* info1 = labeling.GetResultRegionInfo(i);

        cv::Point p1;               //矩形の左上座標
        info1->GetMin(p1.x, p1.y);
        cv::Point p2;               //矩形の右下座標
        info1->GetMax(p2.x, p2.y);

        rects.push_back(cv::Rect(p1, p2));
    }

    //高さの降順にソート
    std::sort(rects.begin(), rects.end(), GreaterRectHeight());

    //横方向に重なっている矩形を統合していく
    while (!rects.empty()) {
        size_t count = 0;   //矩形を統合した回数

        for (size_t i = 0; i < rects.size() - 1; i++) {
            for (size_t j = i+1; j < rects.size(); j++) {
                TRange<double> r1(rects[i].y, rects[i].y + rects[i].height);     //大きい矩形の縦の範囲
                TRange<double> r2(rects[j].y, rects[j].y + rects[j].height);     //小さい矩形の縦の範囲
                TRange<double> rc = r1.intersected(r2);                          //縦の範囲が重なっている範囲
                if (!rc.isNull()) {
                    if (rc.size() >= r2.size() * 0.5) {
                        //小さい矩形の縦の範囲が50%以上重なっていたら統合
                        rects[i] |= rects[j];
                        count++;

                        //統合した矩形を削除
                        rects.erase(rects.begin() + j);
                    }
                }
            }
        }

        //重なり合う矩形がなくなったら終わる
        if (count == 0) {
            break;
        }
    }

    //Y座標の昇順にソート
    std::sort(rects.begin(), rects.end(), LessRectY());

#if 1   //行検出結果を表示
```

```
    cv::Mat imgDebug;
    cv::cvtColor(imgGray, imgDebug, CV_GRAY2RGB);

    //領域を描画
    for (size_t i = 0; i < rects.size(); i++) {
        cv::rectangle(imgDebug, rects[i], CV_RGB(255, 0, 0));
    }

    cv::namedWindow("行検出結果", CV_WINDOW_AUTOSIZE);
    cv::imshow("行検出結果", imgDebug);
    cv::waitKey(0);
    cv::destroyWindow("行検出結果");
#endif

    //行の矩形の配列を返す
    return rects;
}
```

```
#ifndef TRANGE_H
#define TRANGE_H

//! 範囲クラス
template<typename _Tp> class TRange
{
public:
    TRange() : first(0), last(-1)
    {
    };
    TRange(_Tp _first, _Tp _last) : first(_first), last(_last)
    {
    };

    TRange intersected(const TRange& r) const
    {
        TRange intersect;

        if (size() >= r.size()) {
            if ((first <= r.first && last >= r.first)
             || (first <= r.last && last >= r.last)) {
                if (first >= r.first) {
                    intersect.first = first;
                } else {
                    intersect.first = r.first;
                }

                if (last <= r.last) {
                    intersect.last = last;
                } else {
                    intersect.last = r.last;
                }
            }
        } else {
            if ((r.first <= first && r.last >= first)
             || (r.first <= last && r.last >= last)) {
                if (first <= r.first) {
                    intersect.first = r.first;
                } else {
                    intersect.first = first;
                }

                if (last >= r.last) {
                    intersect.last = r.last;
                } else {
                    intersect.last = last;
                }
```

```
            }
        }

        return intersect;
    }

    bool isNull() const
    {
        return (last < first);
    }

    _Tp size() const
    {
        return last - first;
    }

private:
    _Tp first;
    _Tp last;
};

#endif  //TRANGE_H
```