ДОДАТОК А. ПРОГРАМНИЙ КОД ПРОЕКТУ

**package** com.lab111.picturetwister.vectorized;

**import** java.util.HashMap;

**import** com.lab111.picturetwister.core.Inaccurate;

**import** com.lab111.picturetwister.core.Stat;

/\*\*

\* Клас описує точку - як кут пікселя.

\*

\* **@author** Бедь Анатолій Михайлович

\*

\*/

**public** **class** Point {

/\*\*

\* Координата Х на сітці

\*/

**private** **double** x;

/\*\*

\* Координата y на сітці

\*/

**private** **double** y;

/\*\*

\* Мапа, яка зберігає "сусідів" даної точки, у вигляді ключ - точка, ключі:

\* left - точка зліва right - точка српава top - точка зверху bottom - точка

\* знизу

\*/

**private** HashMap<String, Point> neighboringPoints;

/\*\*

\* Конструктор, який створює точку з координатами (x, y) на сітці

\* **@param** x абсциса на сітці

\* **@param** y ордината на сітці

\*/

**public** Point(**double** x, **double** y) {

**this**.x = x;

**this**.y = y;

neighboringPoints = **new** HashMap<String, Point>(4);

}

/\*\*

\* Метод повертає точку яка є сусідньою зліва для даної точки

\* **@return** сусідню точку зліва

\*/

**public** Point getPointLeft() {

**return** neighboringPoints.get("left");

}

/\*\*

\* Метод встановлює точку яка є сусідньою зліва для даної точки

\* **@param** pointLeft сусіднья зліва точка для даної точки

\*/

**public** **void** setPointLeft(Point pointLeft) {

neighboringPoints.put("left", pointLeft);

}

/\*\*

\* Метод повертає точку яка є сусідньою справа для даної точки

\* **@return** сусідню точку справа

\*/

**public** Point getPointRight() {

**return** neighboringPoints.get("right");

}

/\*\*

\* Метод встановлює точку яка є сусідньою справа для даної точки

\* **@param** pointLeft сусіднья справа точка для даної точки

\*/

**public** **void** setPointRight(Point pointRight) {

neighboringPoints.put("right", pointRight);

}

/\*\*

\* Метод повертає точку яка є сусідньою зверху для даної точки

\* **@return** сусідню точку зверху

\*/

**public** Point getPointTop() {

**return** neighboringPoints.get("top");

}

/\*\*

\* Метод встановлює точку яка є сусідньою зверху для даної точки

\* **@param** pointLeft сусіднья зверху точка для даної точки

\*/

**public** **void** setPointTop(Point pointTop) {

neighboringPoints.put("top", pointTop);

}

/\*\*

\* Метод повертає точку яка є сусідньою знизу для даної точки

\* **@return** сусідню точку знизу

\*/

**public** Point getPointBottom() {

**return** neighboringPoints.get("bottom");

}

/\*\*

\* Метод встановлює точку яка є сусідньою знизу для даної точки

\* **@param** pointLeft сусіднья знизу точка для даної точки

\*/

**public** **void** setPointBottom(Point pointBottom) {

neighboringPoints.put("bottom", pointBottom);

}

/\*\*

\* Метод повертає координату x

\* **@return** координату x

\*/

**public** **double** getX() {

**return** x;

}

/\*\*

\* **@return** the y

\*/

**public** **double** getY() {

**return** y;

}

/\*\*

\* toString, що тут скажеш

\*/

@Override

**public** String toString() {

**return** "( " + x + ", " + y + " )";

}

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

Point point = **new** Point(**this**.x, **this**.y);

point.setPointBottom(**this**.getPointBottom());

point.setPointLeft(**this**.getPointLeft());

point.setPointRight(**this**.getPointRight());

point.setPointTop(**this**.getPointTop());

**return** point;

}

/\*\*

\* **@param** x

\* the x to set

\*/

**public** **void** setX(**double** x) {

**this**.x = x;

}

/\*\*

\* **@param** y

\* the y to set

\*/

**public** **void** setY(**double** y) {

**this**.y = y;

}

/\*\*

\* Метод перевіряє, чи входить дана точка в прямокутник утворений двома точками,

\* які зєднані діагоналлю цього прямокутника

\* **@param** vertex1 перша точка

\* **@param** vertex2 друга точка

\* **@return** true, якщо точка лежить всередині прямокутника і false, якщо не лежить

\*/

**public** **boolean** inRect(Point vertex1, Point vertex2) {

**if** (vertex1 == **null** || vertex2 == **null**)

**return** **false**;

**double** left = Stat.*min*(vertex1.getX(), vertex2.getX());

**double** right = Stat.*max*(vertex1.getX(), vertex2.getX());

**double** top = Stat.*min*(vertex1.getY(), vertex2.getY());

**double** bottom = Stat.*max*(vertex1.getY(), vertex2.getY());

**double** x = **this**.getX();

**double** y = **this**.getY();

**return** Inaccurate.*moreOrEquals*(x, left)

&& Inaccurate.*lessOrEquals*(x, right)

&& Inaccurate.*moreOrEquals*(y, top)

&& Inaccurate.*lessOrEquals*(y, bottom);

}

}

**package** com.lab111.picturetwister.vectorized;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** org.apache.commons.math3.geometry.euclidean.twod.Vector2D;

**import** com.lab111.picturetwister.core.Inaccurate;

**import** com.lab111.picturetwister.core.Linear;

/\*\*

\* Клас описує "оболочку", яка складається з точок Point і може бути довільної

\* форми.

\* **@author** Бедь Анатолій Михайлович

\*/

**public** **class** ConvexHull {

/\*\*

\* Список кутів векторного пікселя

\*/

**protected** ArrayList<Point> peakPixel;

/\*\*

\* Атрибути

\*/

**protected** HashMap<String, Object> attribute;

**public** ConvexHull() {

attribute = **new** HashMap<String, Object>();

peakPixel = **new** ArrayList<Point>();

}

/\*\*

\* Конструктор

\*

\* **@param** points

\* список точок

\*/

**public** ConvexHull(ArrayList<Point> points) {

**this**.peakPixel = points;

attribute = **new** HashMap<String, Object>();

}

/\*\*

\* Конструктор

\*

\* **@param** points

\* перелік точок

\*/

**public** ConvexHull(Point... points) {

**this**.peakPixel = **new** ArrayList<Point>();

**for** (Point p : points) {

**this**.peakPixel.add(p);

}

attribute = **new** HashMap<String, Object>();

}

/\*\*

\* Метод для отримання контуру оболочки

\*

\* **@return** список всіх вершин оболочки

\*/

**public** ArrayList<Point> getPeakPixel() {

**return** peakPixel;

}

/\*\*

\* Метод для отримання атрибутів оболочки

\*

\* **@param** key ключ атрибута

\* **@param** value значення атрибута

\*/

**public** <T> T getAttribute(String key) {

**return** (T) attribute.get(key);

}

/\*\*

\* Метод для встановлення атрибутів оболочки

\* **@param** key ключ атрибута

\* **@param** value

\* значення атрибута

\*/

**public** **void** setAttribute(String key, Object value) {

attribute.put(key, value);

}

/\*\*

\* Метод рахує периметер оболочки

\* **@return** периметер оболочки

\*/

**public** **double** getPerimeter() {

**double** res = 0;

**for** (**int** i = 0; i < peakPixel.size() - 1; i++) {

res += Linear.*createVector*(peakPixel.get(i), peakPixel.get(i + 1))

.getNorm();

}

res += Linear.*createVector*(peakPixel.get(peakPixel.size() - 1),

peakPixel.get(0)).getNorm();

setAttribute("p", res);

**return** res;

}

/\*\*

\* Метод рахує площу оболочки

\* **@return** площу оболочки

\*/

**public** **double** getArea() {

**if** (peakPixel.size() == 3) {

Vector2D f = Linear

.*createVector*(peakPixel.get(0), peakPixel.get(1));

Vector2D s = Linear

.*createVector*(peakPixel.get(0), peakPixel.get(2));

setAttribute("a", Math.*abs*(Linear.*product*(f, s)) / 2);

**return** Math.*abs*(Linear.*product*(f, s)) / 2;

}

ArrayList<ConvexHull> splitting = split();

**double** res = 0;

**for** (ConvexHull c : splitting)

res += c.getArea();

**return** res;

}

/\*\*

\* Метод перевіряє, чи знаходиться точка всередині оболочки, яка описується

\* ConvexHull

\* **@param** p точка

\* **@return** true, якщо p - внутрішня точка оболочки, або якщо вона лежить на

\* контурі, false якщо точка лежить поза оболочкою.

\*/

**public** **boolean** isInnerPoint(Point p) {

**if** (peakPixel.size() == 3) {

**double** s = **new** ConvexHull(peakPixel.get(0), peakPixel.get(1), p)

.getArea()

+ **new** ConvexHull(peakPixel.get(1), peakPixel.get(2), p)

.getArea()

+ **new** ConvexHull(peakPixel.get(2), peakPixel.get(0), p)

.getArea();

**return** Inaccurate.*equals*(s, getArea());

}

ArrayList<ConvexHull> splitting = split();

**boolean** res = **false**;

**for** (ConvexHull c : splitting)

res |= c.isInnerPoint(p);

**return** res;

}

**private** **static** **boolean** triangleIsValid(ConvexHull t,

ArrayList<Point> contour) {

Point f = t.getPeakPixel().get(0);

Point s = t.getPeakPixel().get(2);

Vector2D fv = Linear.*createVector*(t.getPeakPixel().get(0), t

.getPeakPixel().get(1));

Vector2D sv = Linear.*createVector*(t.getPeakPixel().get(1), t

.getPeakPixel().get(2));

**if** (Linear.*product*(fv, sv) <= 0)

**return** **false**;

**for** (Point p : contour) {

**if** (!t.getPeakPixel().contains(p) && t.isInnerPoint(p))

**return** **false**;

}

**if** (!t.getPeakPixel().contains(contour.get(0))

&& !t.getPeakPixel().contains(contour.get(contour.size() - 1))) {

**if** (Linear.*segmentIntersection*(f, s,

contour.get(contour.size() - 1), contour.get(0)) != **null**)

**return** **false**;

}

**for** (**int** i = 0; i < contour.size() - 1; i++) {

**if** (!t.getPeakPixel().contains(contour.get(i))

&& !t.getPeakPixel().contains(contour.get(i + 1))) {

**if** (Linear.*segmentIntersection*(f, s, contour.get(i),

contour.get(i + 1)) != **null**)

**return** **false**;

}

}

**return** **true**;

}

**private** **static** **boolean** triangleIsValid(ConvexHull t, ConvexHull f) {

**return** *triangleIsValid*(t, f.getPeakPixel());

}

**private** **static** ConvexHull bestTriangle(ConvexHull a, ConvexHull b) {

**if** (a == **null** && b == **null**)

**return** **null**;

**if** (a == **null** && b != **null**)

**return** b;

**if** (a != **null** && b == **null**)

**return** a;

**double** a\_ = a.getAttribute("factor");

**double** b\_ = b.getAttribute("factor");

**return** (a\_ < b\_) ? b : a;

}

**private** **static** ConvexHull getTriangle(ConvexHull figure) {

ConvexHull res = **null**;

ArrayList<Point> contour = figure.getPeakPixel();

**for** (**int** i = 1; i < contour.size() - 1; i++) {

ConvexHull temp = **new** ConvexHull(contour.get(i - 1),

contour.get(i), contour.get(i + 1));

**if** (*triangleIsValid*(temp, figure)) {

temp.setAttribute("factor",

temp.getArea() / temp.getPerimeter());

res = *bestTriangle*(res, temp);

}

}

ConvexHull temp = **new** ConvexHull(contour.get(contour.size() - 1),

contour.get(0), contour.get(1));

**if** (*triangleIsValid*(temp, figure)) {

temp.setAttribute("factor", temp.getArea() / temp.getPerimeter());

res = *bestTriangle*(res, temp);

}

temp = **new** ConvexHull(contour.get(contour.size() - 2),

contour.get(contour.size() - 1), contour.get(0));

**if** (*triangleIsValid*(temp, figure)) {

temp.setAttribute("factor", temp.getArea() / temp.getPerimeter());

res = *bestTriangle*(res, temp);

}

**if**(!(res != **null**)){

System.*out*.println(figure);

}

**return** res;

}

/\*\*

\* Метод для розбиття оболочки на трикутники. Він базується на наступному

\* алгоритмові.<br/>

\* 1) Проходимо контур по всім точкам і вибираємо трикутники із трьох

\* сусідніх точок. Нехай це будуть точки A(i-1), Ai, A(i+1), тоді точки

\* A(i-1) і A(i+1) зєднуємо хордою. Таким чином можна отримати N різних

\* трикутників для контуру що складається з N точок.<br/>

\* 2) Кожен трикутник перевіряємо на правильність. Якщо хорда A(i-1)A(i+1)

\* проходить не всередині контуру то такий трикутник не проходить перевірку

\* на валідність.<br/>

\* 3) Вибираємо той трикутник у якого відношення площі до периметру є

\* найбільшим (це дозволяє вибрати трикутник найбільш наближений до

\* рівностороннього) Заносимо цей трикутник до списку. <br/>

\* 4) Видаляємо з контуру Ai - точку трикутника, який ми вибрали як

\* "найкращий". <br/>

\* 5) Повторюємо попередні пункти до тих пір поки в контурі не залишиться

\* три точки. <br/>

\* 6) З'єднавши ці три точки отримуємо останній трикутник і заносимо його до

\* списку. <br/>

\*

\* **@return** список трикутників, або null якщо оболочка містить тільки дві

\* точки

\*/

**public** ArrayList<ConvexHull> split() {

**if** (**this**.getPeakPixel().size() <= 2)

**return** **null**;

ArrayList<ConvexHull> res = **new** ArrayList<ConvexHull>();

ConvexHull temp = **new** ConvexHull((ArrayList<Point>) **this**.getPeakPixel()

.clone());

**for** (; temp.getPeakPixel().size() > 3;) {

ConvexHull t = *getTriangle*(temp);

res.add(t);

temp.getPeakPixel().remove(t.getPeakPixel().get(1));

}

temp.getArea();

res.add(temp);

**return** res;

}

/\*\*

\* Метод повертає Point, шо є точкою центр мас. Можливі чотири випадки: <br/>

\* 1) ConvexHull містить тільки одну точку. В цьому випадку повертаємо її

\* саму <br/>

\* 2) ConvexHull містить дві точки. В цьому випадку повертаємо середину

\* відрізка, що сполучає ці дві точки <br/>

\* 3) ConvexHull містить три точки. В цьому випадку маємо трикутник, тому

\* повертаємо точку перетину медіан, яка і буде точкою центер мас трикутника <br/>

\* 4) ConvexHull містить більше трьох точок. В цьому випадку розбиваємо наш

\* полігон на трикутники. Центер мас вираховується як середнє арифметичне

\* центра мас всіх трикутників, з урахуванням вагових коефіцієнтів, якими в

\* даному випадку є площі трикутників

\*

\* **@return** Point - точку центер мас

\*/

**public** Point getCenterOfMass() {

**if** (peakPixel.size() == 1)

**return** peakPixel.get(0);

**if** (peakPixel.size() == 2)

**return** Linear.*middlePointOfSegment*(peakPixel.get(0),

peakPixel.get(1));

**if** (peakPixel.size() == 3) {

Point a1 = peakPixel.get(0);

Point b1 = Linear.*middlePointOfSegment*(peakPixel.get(1),

peakPixel.get(2));

Point a2 = peakPixel.get(1);

Point b2 = Linear.*middlePointOfSegment*(peakPixel.get(2),

peakPixel.get(0));

**return** Linear.*segmentIntersection*(a1, b1, a2, b2);

}

ArrayList<ConvexHull> splitting = split();

**double** x\_ = 0;

**double** y\_ = 0;

**for** (ConvexHull c : splitting) {

Point p = c.getCenterOfMass();

**double** a = c.getArea();

x\_ += p.getX() \* a;

y\_ += p.getY() \* a;

}

x\_ /= getArea();

y\_ /= getArea();

**return** **new** Point(x\_, y\_);

}

**public** String toString() {

String res = "[ ";

**for** (Point p : peakPixel) {

res += p.toString() + " ";

}

**return** res + " ]: " + attribute.toString();

}

}

**package** com.lab111.picturetwister.vectorized;

**import** java.util.ArrayList;

/\*\*

\* Клас, який описує векторний піксель, наслідується від класа ConvexHull

\* **@author** Бедь Анатолій Михайлович

\*

\*/

**public** **class** VectorPixel **extends** ConvexHull{

**public** VectorPixel(ArrayList<Point> points) {

**super**(points);

}

/\*\*

\* Метод встановлює колір векторного пікселя

\* **@param** color

\*/

**public** **void** setColor(**int** color){

attribute.put("color", color);

}

/\*\*

\* Метод повертає колір векторного пікселя

\* **@return**

\*/

**public** **int** getColor(){

**return** (**int**) attribute.get("color");

}

/\*\*

\* Метод повертає добуток "ваги" пікселя на його центер мас

\* **@return**

\*/

**public** **double**[] getWeightPixel(){

Point c = getCenterOfMass();

**return** **new** **double**[] {c.getX()\*getColor(), c.getY()\*getColor()};

}

/\*\*

\* Метод повертає точку з заданої позиції

\* **@param** poz позиція точки яку хочемо отримати

\* **@return** точку в позиції poz

\*/

**public** Point getPoint(**int** poz){

**return** peakPixel.get(poz);

}

@Override

**public** String toString() {

**return** "VectorPixel [peakPixel=" + peakPixel.toString() + ", color=" + attribute.get("color") + "]";

}

}

**package** com.lab111.picturetwister.vectorized;

**import** java.awt.Color;

**import** java.awt.Graphics2D;

**import** java.awt.image.BufferedImage;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.lab111.picturetwister.core.Function;

**import** com.lab111.picturetwister.core.Linear;

**import** com.lab111.picturetwister.core.Stat;

**import** com.lab111.picturetwister.imageutil.ImageUtil;

/\*\*

\* Клас описує сітку точок, зв'язаних між собою

\* **@author** Бедь Анатолій Михайлович

\*/

**public** **class** Reticle {

**private** Point[][] points;

**private** Function weightFunction;

**private** VectorPixel[][] vectorPixels;

**private** MatrixTransform transform;

**public** **int**[][] colorMatrix;

**private** **int** height;

**private** **int** width;

/\*\*

\* Конструктор, який створює сітку

\*

\* **@param** colorsRGB

\* - матриця кольорів

\*/

**public** Reticle(**int**[][] colorsRGB) {

points = *createPointSet*(colorsRGB[0].length, colorsRGB.length);

colorMatrix = colorsRGB;

height = colorsRGB.length;

width = colorsRGB[0].length;

vectorPixels = *createConvexHullsSet*(colorMatrix, points);

transform = **new** MatrixTransform(**this**);

}

/\*\*

\* Метод повертає сітку точок, в якій кожна точка знає про свої сусідні

\* точки зверху, зліва, справа, знизу

\*

\* **@param** width ширина зображення

\* **@param** height висота зображення

\* **@return** Points[][] - сітка точок

\*/

**public** **static** Point[][] createPointSet(**int** width, **int** height) {

Point[][] points = **new** Point[height + 1][width + 1];

/\*

\* Створюємо точку в координатах (0, 0)

\*/

points[0][0] = **new** Point(0, 0);

/\*

\* Заповнення першого рядка матриці зразу вказуємо для даної точки

\* першого рядка її сусідню точку зліва

\*/

**for** (**int** j = 1; j < width + 1; j++) {

points[0][j] = **new** Point(0, j);

points[0][j].setPointLeft(points[0][j - 1]);

}

/\*

\* Заповнення першого стовпця матриці зразу вказуємо для даної точки

\* першого стовпця її сусідню точку зверху

\*/

**for** (**int** i = 1; i < height + 1; i++) {

points[i][0] = **new** Point(i, 0);

points[i][0].setPointTop(points[i - 1][0]);

}

/\*

\* Заповнення всіх інших елементів матриці Створюємо точку. Для даної

\* точки вказуємо її сусідні точки зверху та зліва для точки яка

\* знаходиться зліва по діагоналі вказуємо сусідні точки знизу і справа.

\* Після виконання циклів, для точок останнього стовпця не буде вказано

\* іхніх сусідніх точок знизу, а для точок останнього рядка - сусідніх

\* точок справа

\*/

**for** (**int** i = 1; i < height + 1; i++)

**for** (**int** j = 1; j < width + 1; j++) {

points[i][j] = **new** Point(i, j);

points[i][j].setPointTop(points[i - 1][j]);

points[i][j].setPointLeft(points[i][j - 1]);

points[i - 1][j - 1].setPointRight(points[i - 1][j]);

points[i - 1][j - 1].setPointBottom(points[i][j - 1]);

}

/\*

\* Вказуємо для точок останнього стовпця сусідні їм точок знизу

\*/

**for** (**int** i = 0; i < height; i++) {

points[i][width].setPointBottom(points[i + 1][width]);

}

/\*

\* Вказуємо для точок останнього рядка сусідніх їм точки справа

\*/

**for** (**int** j = 0; j < width; j++) {

points[height][j].setPointRight(points[height][j + 1]);

}

**return** points;

}

/\*\*

\* Метод повертає матрицю векторних пікселів

\*

\* **@param** intMatrix матриця кольорів

\* **@param** points сітка точок

\* **@return**

\*/

**public** **static** VectorPixel[][] createConvexHullsSet(**int**[][] intMatrix,

Point[][] points) {

**int** width = intMatrix[0].length;

**int** height = intMatrix.length;

VectorPixel[][] vectorPixels = **new** VectorPixel[height][width];

ArrayList<Point> pointList;

/\*

\* Кладемо в список чотири точки(вершини пікселя) і створюємо векторний

\* піксель, який кладемо в масив.

\*/

**for** (**int** i = 0; i < height; i++) {

**for** (**int** j = 0; j < width; j++) {

pointList = **new** ArrayList<Point>(4);

pointList.add(points[i][j]);

pointList.add(points[i][j + 1]);

pointList.add(points[i + 1][j + 1]);

pointList.add(points[i + 1][j]);

vectorPixels[i][j] = **new** VectorPixel(pointList);

vectorPixels[i][j].setColor(intMatrix[i][j]);

}

}

**return** vectorPixels;

}

/\*\*

\* метод для маштабування сітки

\*

\* **@param** mx

\* параметр збільшення по осі Ох

\* **@param** my

\* параметр збільшення по осі Оу

\*/

**public** **void** scale(**double** mx, **double** my) {

transform.scale(mx, my);

width = (**int**) (width \* my);

height = (**int**) (height \* mx);

}

/\*\*

\* метод для здійснення повороту сітки на кут

\*

\* **@param** angle

\* кут на який потрібно повернути сітку

\*/

**public** **void** rotate(**double** angle) {

transform.rotate(angle);

}

/\*\*

\* метод, який дозволяє перемістити сітку

\*

\* **@param** transferX

\* відстань на яку треба перемістити сітку по координаті X

\* **@param** transferY

\* відстань на яку треба перемістити сітку по координаті Y

\*/

**public** **void** transfer(**double** transferX, **double** transferY) {

transform.transfer(transferX, transferY);

}

/\*

\* Метод, який ламає сітку.

\*/

**public** **void** deformation() {

VectorPixel[][] hulls = **this**.cloneConvexHulls();

**double** x = 0;

**double** y = 0;

**for** (**int** i = 1; i < vectorPixels.length - 1; i++) {

**for** (**int** j = 1; j < vectorPixels[0].length - 1; j++) {

**if** (Stat.*max*(Stat.*difference*(hulls[i][j].getColor(),

hulls[i - 1][j - 1].getColor()), Stat.*difference*(

hulls[i][j].getColor(), hulls[i - 1][j].getColor()),

Stat.*difference*(hulls[i][j].getColor(),

hulls[i - 1][j + 1].getColor()), Stat

.*difference*(hulls[i][j].getColor(),

hulls[i][j - 1].getColor()), Stat

.*difference*(hulls[i][j].getColor(),

hulls[i][j + 1].getColor()), Stat

.*difference*(hulls[i][j].getColor(),

hulls[i + 1][j - 1].getColor()), Stat

.*difference*(hulls[i][j].getColor(),

hulls[i + 1][j].getColor()), Stat

.*difference*(hulls[i][j].getColor(),

hulls[i + 1][j + 1].getColor())) > 25) {

x = 0;

y = 0;

x = Stat.*sum*(hulls[i][j].getWeightPixel()[0],

hulls[i][j + 1].getWeightPixel()[0],

hulls[i + 1][j].getWeightPixel()[0],

hulls[i + 1][j + 1].getWeightPixel()[0]);

y = Stat.*sum*(hulls[i][j].getWeightPixel()[1],

hulls[i][j + 1].getWeightPixel()[1],

hulls[i + 1][j].getWeightPixel()[1],

hulls[i + 1][j + 1].getWeightPixel()[1]);

**double** mass = Stat.*sum*(hulls[i][j].getColor(),

hulls[i][j + 1].getColor(),

hulls[i + 1][j].getColor(),

hulls[i + 1][j + 1].getColor());

x = x / mass;

y = y / mass;

vectorPixels[i][j].getPoint(2).setX(x);

vectorPixels[i][j].getPoint(2).setY(y);

}

}

}

}

/\*\*

\* Метод для деформації сітки

\*/

**public** **void** deformation1() {

**for** (**int** i = 0; i < points.length; i++) {

**for** (**int** j = 0; j < points[0].length; j++) {

ArrayList<VectorPixel> connected = getConnectedPixels(points[i][j]);

**double**[] c = **new** **double**[connected.size()];

**for** (**int** k = 0; k < c.length; k++) {

c[k] = connected.get(k).getColor();

}

**if** (Stat.*difference*(Stat.*max*(c), Stat.*min*(c)) > 25) {

**for** (**int** k = 0; k < c.length; k++) {

// connected.get(k).setAttribute("w", 129-Math.abs(new

// Double(connected.get(k).getColor())-128));

**double** w = 129 - Math.*abs*(**new** Double(connected.get(k)

.getColor()) - 128);

w = w \* w;

connected.get(k).setAttribute("w", w);

}

Point res = Linear.*centerOfMassForSys*(connected);

points[i][j].setX(res.getX());

points[i][j].setY(res.getY());

}

}

}

}

/\*\*

\* Метод для отримання сітки зв'язаних точок

\*

\* **@return** повертає сітку зв'язаних точок

\*/

**public** Point[][] getPoints() {

**return** points;

}

/\*\*

\* Метод для отримання векторних пікселів з сітки

\* **@return** vectorPixels

\*/

**public** VectorPixel[][] getConvexHulls() {

**return** vectorPixels;

}

/\*\*

\* метод повертає матрицю координат у вигляді Ax Cx ..... Px Ay Cy ..... Py

\* Bx Dx ..... Lx By Dy ..... Ly Mx Nx ..... Qx My Ny ..... Qy

\* ............... ............... Rx Tx ...... Hx Ry Ty ...... Hy

\*

\* де точки A(Ax,Ay), B(Bx,By), D(Dx,Dy), C(Cx,Cy) є кутами пікселя (і так

\* далі відповідно)

\*

\* **@return**

\*/

**public** **double**[][] getPointsXY() {

**double**[][] vector = **new** **double**[2 \* points.length][points[0].length];

**int** k = 0;

**for** (**int** i = 0; i < points.length; i++) {

**for** (**int** j = 0; j < points[0].length; j++) {

vector[k][j] = points[i][j].getX();

vector[k + 1][j] = points[i][j].getY();

}

k = k + 2;

}

**return** vector;

}

/\*\*

\* Метод для отримання копії ConvexHull[][]

\* **@return** повертає копію ConvexHull[][], що містить Reticle

\*/

**public** VectorPixel[][] cloneConvexHulls() {

Point[][] points = **this**.clonePoints();

**int** width = colorMatrix[0].length;

**int** height = colorMatrix.length;

VectorPixel[][] vectorPixels = **new** VectorPixel[height][width];

ArrayList<Point> pointList;

/\*

\* Кладемо в список чотири точки(вершини пікселя) і створюємо векторний

\* піксель, який кладемо в масив.

\*/

**for** (**int** i = 0; i < height; i++) {

**for** (**int** j = 0; j < width; j++) {

pointList = **new** ArrayList<Point>(4);

pointList.add(points[i][j]);

pointList.add(points[i][j + 1]);

pointList.add(points[i + 1][j + 1]);

pointList.add(points[i + 1][j]);

vectorPixels[i][j] = **new** VectorPixel(pointList);

vectorPixels[i][j].setColor(colorMatrix[i][j]);

}

}

**return** vectorPixels;

}

/\*\*

\* Метод для отримання копії сітки зв'язаних між собою точок

\* **@return** повертає копію сітки точок

\*/

**public** Point[][] clonePoints() {

**int** width = colorMatrix[0].length;

**int** height = colorMatrix.length;

Point[][] points = **new** Point[height + 1][width + 1];

/\*

\* Створюємо точку в координатах (0, 0)

\*/

points[0][0] = **new** Point(**this**.points[0][0].getX(),

**this**.points[0][0].getY());

/\*

\* Заповнення першого рядка матриці зразу вказуємо для даної точки

\* першого рядка її сусідню точку зліва

\*/

**for** (**int** j = 1; j < width + 1; j++) {

points[0][j] = **new** Point(**this**.points[0][j].getX(),

**this**.points[0][j].getY());

points[0][j].setPointLeft(points[0][j - 1]);

}

/\*

\* Заповнення першого стовпця матриці зразу вказуємо для даної точки

\* першого стовпця її сусідню точку зверху

\*/

**for** (**int** i = 1; i < height + 1; i++) {

points[i][0] = **new** Point(**this**.points[i][0].getX(),

**this**.points[i][0].getY());

points[i][0].setPointTop(points[i - 1][0]);

}

/\*

\* Заповнення всіх інших елементів матриці Створюємо точку. Для даної

\* точки вказуємо її сусідні точки зверху та зліва для точки яка

\* знаходиться зліва по діагоналі вказуємо сусідні точки знизу і справа.

\* Після виконання циклів, для точок останнього стовпця не буде вказано

\* іхніх сусідніх точок знизу, а для точок останнього рядка - сусідніх

\* точок справа

\*/

**for** (**int** i = 1; i < height + 1; i++)

**for** (**int** j = 1; j < width + 1; j++) {

points[i][j] = **new** Point(**this**.points[i][j].getX(),

**this**.points[i][j].getY());

points[i][j].setPointTop(points[i - 1][j]);

points[i][j].setPointLeft(points[i][j - 1]);

points[i - 1][j - 1].setPointRight(points[i - 1][j]);

points[i - 1][j - 1].setPointBottom(points[i][j - 1]);

}

/\*

\* Вказуємо для точок останнього стовпця сусідні їм точок знизу

\*/

**for** (**int** i = 0; i < height; i++) {

points[i][width].setPointBottom(points[i + 1][width]);

}

/\*

\* Вказуємо для точок останнього рядка сусідніх їм точки справа

\*/

**for** (**int** j = 0; j < width; j++) {

points[height][j].setPointRight(points[height][j + 1]);

}

**return** points;

}

/\*\*

\* Метод для отримання матриці кольорів

\*

\* **@return** colorповертає матрицю кольорів

\*/

**public** **int**[][] getColorMatrix() {

**return** colorMatrix;

}

/\*\*

\* Мето для отримання висоти сітки

\* **@return** висоту сітки

\*/

**public** **int** getHeight() {

**return** height;

}

/\*\*

\* Мето для отримання ширини сітки

\* **@return** ширину сітки

\*/

**public** **int** getWidth() {

**return** width;

}

/\*\*

\* Метод для отримання вагової фінкції

\* **@return** вагову функцію

\*/

**public** Function getWeightFunction() {

**return** weightFunction;

}

/\*\*

\* Мето для встановлення вагової функції

\* **@param** вагова функція

\*/

**public** **void** setWeightFunction(Function weightFunction) {

**this**.weightFunction = weightFunction;

}

/\*\*

\* Мето для отримання пікселів до яких входить задана точка p

\*

\* **@param** p

\* точка

\* **@return** список пікселів до яких вжодить задана точка

\*/

**public** ArrayList<VectorPixel> getConnectedPixels(Point p) {

ArrayList<VectorPixel> res = **new** ArrayList<VectorPixel>();

**int** i\_ = -1;

**int** j\_ = -1;

**for** (**int** i = 0; i < points.length; i++) {

**for** (**int** j = 0; j < points[0].length; j++) {

**if** (p == points[i][j]) {

i\_ = i;

j\_ = j;

**break**;

}

}

**if** (i\_ >= 0)

**break**;

}

**if** (i\_ < 0)

**return** res;

**if** (((i\_ - 1) >= 0) && (j\_ - 1) >= 0)

res.add(vectorPixels[i\_ - 1][j\_ - 1]);

**if** (((i\_) < vectorPixels.length) && (j\_ - 1) >= 0)

res.add(vectorPixels[i\_][j\_ - 1]);

**if** (((i\_ - 1) >= 0) && (j\_) < vectorPixels[0].length)

res.add(vectorPixels[i\_ - 1][j\_]);

**if** (((i\_) < vectorPixels.length) && (j\_) < vectorPixels[0].length)

res.add(vectorPixels[i\_][j\_]);

**return** res;

}

/\*\*

\* Метод для виділення контурних ланцюгів

\* **@param** porog параметр

\* **@return** контурний ланцюг

\*/

**public** ConvexHull searchCounture(**int** porog) {

List<ConvexHull> listCounture = **new** ArrayList<ConvexHull>();

ConvexHull hull = **new** ConvexHull();

**int** color1;

**int** color2;

**for** (**int** i = 1; i < points.length - 2; i++) {

**for** (**int** j = 1; j < points[i].length - 2; j++) {

color1 = 0x0000ff & vectorPixels[i][j].getColor();

color2 = vectorPixels[i - 1][j].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(1)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(1));

}

color2 = vectorPixels[i - 1][j - 1].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(0)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(0));

}

color2 = vectorPixels[i][j - 1].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(0)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(0));

}

color2 = vectorPixels[i + 1][j].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(3)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(3));

}

color2 = vectorPixels[i + 1][j].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(3)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(3));

}

color2 = vectorPixels[i + 1][j + 1].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(2)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(2));

}

color2 = vectorPixels[i][j + 1].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(2)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(2));

}

color2 = vectorPixels[i - 1][j + 1].getColor() & 0x0000ff;

**if** (Stat.*difference*(color2, color1) > porog) {

**if** (!hull.peakPixel

.contains(vectorPixels[i][j].getPoint(1)))

hull.peakPixel.add(vectorPixels[i][j].getPoint(1));

}

}

}

listCounture.add(hull);

**return** hull;

}

}

**package** com.lab111.picturetwister.vectorized;

/\*\*

\* Клас що описує матрицю перетоврень

\*

\* **@author** Бедь Анатолій Михайлович

\*

\*/

**public** **class** MatrixTransform {

**private** **double**[][] matrix;

**private** Reticle reticle;

**public** MatrixTransform(Reticle reticle) {

**this**.reticle = reticle;

matrix = **new** **double**[][] { { 1, 0, 0 },

{ 0, 1, 0 },

{ 0, 0, 1 } };

}

/\*\*

\* Метод для маштабування сітки побудованої з точок Point

\* **@param** mx коефіцієнт маштабування по осі X

\* **@param** my коефіцієнт маштабування по осі Y

\* **@return** двовимірний масив точок промаштабованої Reticle

\*/

**public** Point[][] scale(**double** mx, **double** my) {

setScale(mx, my);

Point[][] points = reticle.getPoints();

**for** (**int** i = 0; i < points.length; i++) {

**for** (**int** j = 0; j < points[0].length; j++) {

points[i][j].setX(points[i][j].getX() \* matrix[0][0]);

points[i][j].setY(points[i][j].getY() \* matrix[1][1]);

}

}

**return** points;

}

/\*\*

\* Метод для встановлення коефіцієнтів маштабування

\* **@param** mx коефіцієнт маштабування по осі X

\* **@param** my коефіцієнт маштабування по осі Y

\*/

**public** **void** setScale(**double** mx, **double** my) {

matrix[0][0] = mx;

matrix[1][1] = my;

}

/\*\*

\* Метод для встановлення кута повороту сітки

\* **@param** angle кут повороту заданий в градусах.

\*/

**public** **void** setRotate(**double** angle) {

angle = angle \* Math.*PI* / 180;

matrix[0][0] = Math.*cos*(angle);

matrix[1][1] = Math.*cos*(angle);

matrix[0][1] = Math.*sin*(angle);

matrix[1][0] = -Math.*sin*(angle);

matrix[2][2] = 1;

}

/\*\*

\* Метод для повороту сітки

\* **@param** angle кут повороту заданий в градусах.

\* **@return** двовимірний масив точок повернутої на кут angle Reticle

\*/

**public** Point[][] rotate(**double** angle) {

setRotate(angle);

Point[][] points = reticle.getPoints();

**for** (**int** i = 0; i < points.length; i++) {

**for** (**int** j = 0; j < points[0].length; j++) {

points[i][j].setX(points[i][j].getX() \* matrix[0][0]

+ points[i][j].getY() \* matrix[0][1]);

points[i][j].setY(points[i][j].getX() \* matrix[1][0]

+ points[i][j].getY() \* matrix[1][1]);

}

}

**return** points;

}

/\*\*

\* Метод для переміщення сітки

\*

\* **@param** transferX відстань для переносу по осі X задана в пікселях

\* **@param** transferY відстань для переносу по осі Y задана в пікселях

\* **@return** двовимірний масив точок переміщеної Reticle

\*/

**public** Point[][] transfer(**double** transferX, **double** transferY) {

setTransfer(transferX, transferY);

Point[][] points = reticle.getPoints();

**for** (**int** i = 0; i < points.length; i++) {

**for** (**int** j = 0; j < points[0].length; j++) {

points[i][j].setX(points[i][j].getX() + matrix[0][2]);

points[i][j].setY(points[i][j].getY() + matrix[1][2]);

}

}

**return** points;

}

/\*\*

\* Метод для встановлення значень переміщення сітки

\* **@param** transferX відстань для переносу по осі X задана в пікселях

\* **@param** transferY відстань для переносу по осі Y задана в пікселях

\*/

**public** **void** setTransfer(**double** transferX, **double** transferY) {

matrix[0][2] = transferX;

matrix[1][2] = transferY;

**for** (**int** i = 0; i < matrix.length; i++) {

matrix[i][i] = 1;

}

}

}

**package** com.lab111.picturetwister.core;

/\*\*

\* Даний інтерфейс описує функцію

\* **@author**

\*

\*/

**public** **interface** Function {

**public** **double** getValue(**double**... args);

}

**package** com.lab111.picturetwister.core;

/\*\*

\* Клас з статичними методами порівняння двох величин

\* **@author**

\*

\*/

**public** **class** Inaccurate {

**public** **static** **double** *ACCURACY* = 0.000001;

**public** Inaccurate() {

// **TODO** Auto-generated constructor stub

}

/\*\*

\* Перевіряє два числа на рівність з точністю 0.000001

\* **@param** a перше число

\* **@param** b друге число

\* **@return** true, якщо a рівне b і false, якщо a не рівне b

\*/

**public** **static** **boolean** equals(**double** a, **double** b) {

**return** Math.*abs*(a - b) <= *ACCURACY*;

}

/\*\*

\* Перевіряє чи a < b з точністю 0.000001

\* **@param** a перше число

\* **@param** b друге число

\* **@return** true, якщо a мешне b і false, якщо a більше або рівне b

\*/

**public** **static** **boolean** less(**double** a, **double** b) {

**return** (b - a) > *ACCURACY*;

}

/\*\*

\* Перевіряє чи a > b з точністю 0.000001

\* **@param** a перше число

\* **@param** b друге число

\* **@return** true, якщо a більше b і false, якщо a менше або рівне b

\*/

**public** **static** **boolean** more(**double** a, **double** b) {

**return** (a - b) > *ACCURACY*;

}

/\*\*

\* Перевіряє чи a <= b з точністю 0.000001

\* **@param** a перше число

\* **@param** b друге число

\* **@return** true, якщо a менше або рівне b і false, якщо a більше b

\*/

**public** **static** **boolean** lessOrEquals(**double** a, **double** b) {

**return** *less*(a, b) || *equals*(a, b);

}

/\*\*

\* Перевіряє чи a >= b з точністю 0.000001

\* **@param** a перше число

\* **@param** b друге число

\* **@return** true, якщо a більше або рівне b і false, якщо a менше b

\*/

**public** **static** **boolean** moreOrEquals(**double** a, **double** b) {

**return** *more*(a, b) || *equals*(a, b);

}

}

**package** com.lab111.picturetwister.core;

**import** java.util.ArrayList;

**import** org.apache.commons.math3.geometry.euclidean.twod.Line;

**import** org.apache.commons.math3.geometry.euclidean.twod.Vector2D;

**import** com.lab111.picturetwister.vectorized.ConvexHull;

**import** com.lab111.picturetwister.vectorized.Point;

/\*\*

\* Клас який містить статичні методи для роботи з геометрією

\* **@author**

\*

\*/

**public** **class** Linear {

/\*\*

\* Метод створю екземпляр класа Vector2D, який описує вектор

\* **@param** a точка початку вектора

\* **@param** b точка кінця вектора

\* **@return** 2D векто (екземпляр класа Vector2D)

\*/

**public** **static** Vector2D createVector(Point a,Point b){

**return** **new** Vector2D(b.getX()-a.getX(), b.getY()-a.getY());

}

/\*\*

\* Метод для аналізу взаємного розташування двох векторів

\* **@param** first перший вектор

\* **@param** second другий вектор

\* **@return** повертає додатнє число, якщо перший вектор випереджає другий,

\* і відємне коли другий випереджає перший,

\* якщо вектора дивляться в одному напрямку то повертає 0

\*

\*/

**public** **static** **double** product(Vector2D first,Vector2D second){

**return** -first.getX()\*second.getY()+second.getX()\*first.getY();

}

/\*\*

\* Метод знахотидь середину відрізка

\* **@param** p1 перша точка

\* **@param** p2 друга точка

\* **@return** Point - точку, яка є серединою відрізка що з'єднує точки p1 і p2

\*/

**public** **static** Point middlePointOfSegment(Point p1, Point p2){

**return** **new** Point(Stat.*average*(p1.getX(),p2.getX()), Stat.*average*(p1.getY(),p2.getY()));

}

/\*\*

\* Метод перетворює вектор в точку

\* **@param** p точка

\* **@return** вектор, початок якого точка (0, 0), а кінець точка p; повертає null, якщо p null

\*

\*/

**public** **static** Vector2D toVector (Point p){

**return** (p == **null**) ? **null** : **new** Vector2D(p.getX(),p.getY());

}

/\*\*

\* Метод перетворє вектор з координатами (x, y) в точку з координатами (x, y)

\* **@param** v вектор з координатами (x, y)

\* **@return** точка з координатами (x, y), де x,y - кординати вектора v

\*/

**public** **static** Point toPoint (Vector2D v){

**return** (v == **null**) ? **null** : **new** Point(v.getX(),v.getY());

}

/\*\*

\* Метод повертає екземпляр класа Line, який описує лінію

\* **@param** p1 перша точка

\* **@param** p2 друга точка

\* **@return** лінію, яка проходить чере точки p1, p2

\*/

**public** **static** Line lineModel(Point p1, Point p2){

**return** **new** Line(*toVector*(p1), *toVector*(p2));

}

/\*\*

\* Метод повертає екземпляр класа Point - точку перетину двох ліній

\* **@param** l1 перша лінія

\* **@param** l2 друга лінія

\* **@return** точка перетину ліній l1 і l2, повертає null якщо лінії не перетинаються

\*/

**public** **static** Point lineIntersection(Line l1, Line l2){

**return** *toPoint*(l1.intersection(l2));

}

/\*\*

\* Метод знаходить точку перетину двох відрізків a1b1 i a2b2

\* **@param** a1 перша точка

\* **@param** b1 друга точка

\* **@param** a2 третя точка

\* **@param** b2 четверта точка

\* **@return** точку перетину двох відрізків, або null якщо відрізки не перетинеаються

\*/

**public** **static** Point segmentIntersection(Point a1, Point b1, Point a2, Point b2){

Line l1 = *lineModel*(a1, b1);

Line l2 = *lineModel*(a2, b2);

Point p = *lineIntersection*(l1, l2);

**if** (p == **null**) **return** **null**;

**if** ( !p.inRect(a1, b1) || !p.inRect(a2, b2)) **return** **null**;

**return** p;

}

**public** **static** Point centerOfMassForPlanarObj(ConvexHull object){

// **TODO** сделать centerOfMassForPlanarObj

**return** **null**;

}

/\*\*

\* Метод знаходить точку центер мас системи з оболочок ConvexHull або його наслідників.<br/>

\* **@param** objects лінійний список з ConvexHull або його наслідників

\* **@return** точку центер мас

\*/

**public** **static** Point centerOfMassForSys(ArrayList<? **extends** ConvexHull> objects){

**double** x\_ = 0;

**double** y\_ = 0;

**double** w\_ = 0;

**for**(ConvexHull c :objects){

Point p = c.getCenterOfMass();

**double** a = c.getAttribute("w");

x\_ += p.getX()\*a;

y\_ += p.getY()\*a;

w\_ += a;

}

x\_ /= w\_;

y\_ /= w\_;

**return** **new** Point(x\_,y\_);

}

**public** **static** Point perpendicularLine(Line line, Point point){

**return** **null**;

}

}

**package** com.lab111.picturetwister.core;

**import** org.apache.commons.math3.stat.StatUtils;

/\*\*

\* Клас з статичними методами математичної статистики

\* **@author**

\*

\*/

**public** **class** Stat {

**public** **static** **double** *ACCURACY* = 0.00000000000001;

**public** Stat() {

}

/\*\*

\* Метод повертає суму елементів

\* **@param** values вхідний масив

\* **@return** суму елементів, або 0 якщо не знадано жодного параметра

\*/

**public** **static** **double** sum(**double**... values) {

**double** result = 0;

**for** (**double** value : values)

result += value;

**return** result;

}

/\*\*

\* Метод шукає максимальний елемент

\* **@param** values вхідний масив

\* **@return** максимальний з елементів, або 0 якщо не знадано жодного параметра

\*/

**public** **static** **double** max(**double**... values) {

**double** result = values[0];

**for** (**double** value : values)

result = (result < value) ? value : result;

**return** result;

}

/\*\*

\* Метод шукає мінімальний елемент

\* **@param** values вхідний масив

\* **@return** мінімальний з елементів, або 0 якщо не знадано жодного параметра

\*/

**public** **static** **double** min(**double**... values) {

**double** result = values[0];

**for** (**double** value : values)

result = (result > value) ? value : result;

**return** result;

}

/\*\*

\* Метод шукає індекс максимального елемента

\* **@param** values вхідний масив

\* **@return** індекс максимального елемента, або 0 якщо не знадано жодного параметра

\*/

**public** **static** **int** maxIndex(**double**... values) {

**double** max = values[0];

**int** res = 0;

**for** (**int** i = 1; i < values.length; i++) {

**if** (max < values[i]) {

max = values[i];

res = i;

}

}

**return** res;

}

/\*\*

\* Метод шукає індекс мінімального елемента

\* **@param** values вхідний масив

\* **@return** індекс мінімального елемента, або 0 якщо не знадано жодного параметра

\*/

**public** **static** **int** minIndex(**double**... values) {

**double** min = values[0];

**int** res = 0;

**for** (**int** i = 1; i < values.length; i++) {

**if** (min > values[i]) {

min = values[i];

res = i;

}

}

**return** res;

}

/\*\*

\* Метод рахує модуль різниці двох чисел

\* **@param** a перше число

\* **@param** b друге число

\* **@return** модуль різниці a і b

\*/

**public** **static** **double** difference(**double** a, **double** b) {

**return** Math.*abs*(a - b);

}

/\*\*

\* Метод шукає середнє арифметичне

\* **@param** values вхідний масив

\* **@return** середнє арифметичне елементів у вхідному масиві, або Double.NaN, якщо масив порожній.

\*/

**public** **static** **double** average(**double**... values) {

**return** StatUtils.*mean*(values);

}

/\*\*

\* Метод шукає середньоквадратичне відхилення

\* **@param** values вхідний масив

\* **@return** середньоквадратичне відхилення

\*/

**public** **static** **double** stdDeviance(**double**... values) {

**return** Math.*sqrt*(StatUtils.*variance*(values));

}}

ДОДАТОК Б. СТРУКТУРА ПРОЕКТУ

