**MMVE Project – Technical Documentation**

**Introduction**

Before I can explain the implementation, I first need to give an intuitive explanation to what the algorithm does. A more theoretical explanation can be found in ref-1.

Part 1 of the algorithm

Given a convex body (or rather, a group of pixels representing a convex body), the algorithm first finds an arbitrary **starting point** inside the body, using quadrant search (basically – binary search in 2 dimensions). It also receives an epsilon (eps1), and will not search squares smaller than that epsilon - for example, if eps1=5, a body that covers an 5x5 pixel area will always be found, but a 3x100 body might be missed. Once it finds the **starting point**, it repeats the following:

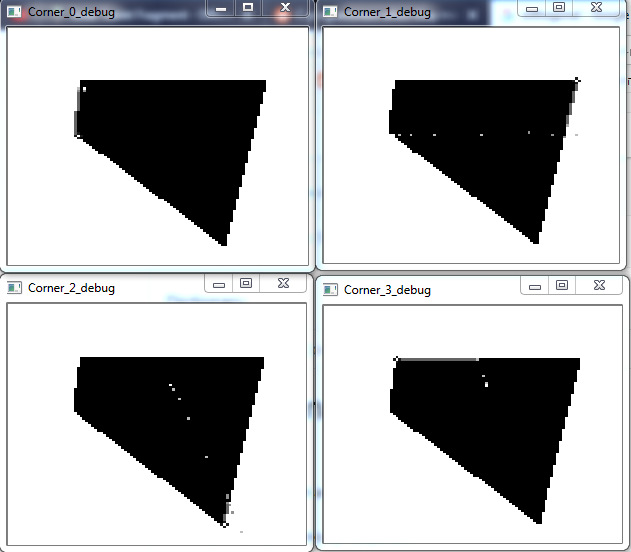
Chose a direction (the first 2 are arbitrary opposite directions, the next 2 are orthogonal to the line between the first 2 points found).

Do binary hops in the chosen direction, until you find the border of the image. Obviously the hops grow x2 until you leave the image, then shrink x2 until you reach a **border point** back inside the image.

Once you found the border, crawl along the direction which maximizes your distance from the starting point (such direction has to exist and be singular, since the body is convex), and once you reach a point from which any movement decreeces the distance, you save it as a **corner point**.

The **starting point** is then updated to be the avarage of all points found so far (at 1, 2 and 4 points).

Once the initial 4 points **p[i]** are found, assign equal weights **u[i]** of **¼** to each point.

**Finding the first 4 points**

Part 2 of the algorithm

From here, the following has to be repeated:

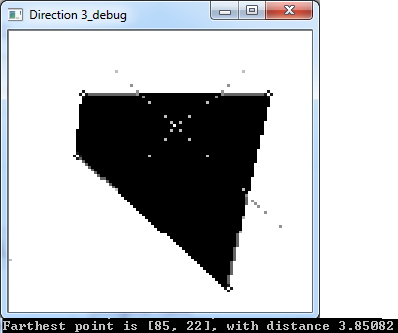
Calculate the ellipsoid matrix Q – equation (28) in my version of the article. Remember d = 2!

Use SVD(Q) (Singular Value Decomposition) to extract the 2x2 matrix V(t) from Q.

From V(t) – which represents the ellipsoid rotation- calculate the angles of the main/secondary axis. The way to do this is use [atan2](http://www.cplusplus.com/reference/cmath/atan2) on each column in the matrix.

Along the line of each angle, find the border and crawl to the farthest corner at that angle you can – **however**, the distance function should be the distance function from the ellipse, defined by Q and c (the ellipse center). It is written as equation (45) in my version of the article.

Also, remember that the center is now calculated by multiplying each **corner point p[i]** by its **weight u[i]**.

 **Image of finding 4 points – 1 in each direction**

Now, chose the farthest point (one with the highest distance). It might be an existing point, or a completely new one.

Calculate **beta** – defined at step 5 of algorithm 2 in my version of the article. It is defined as

where **k** is the distance of the point we found, and **d** is 2.

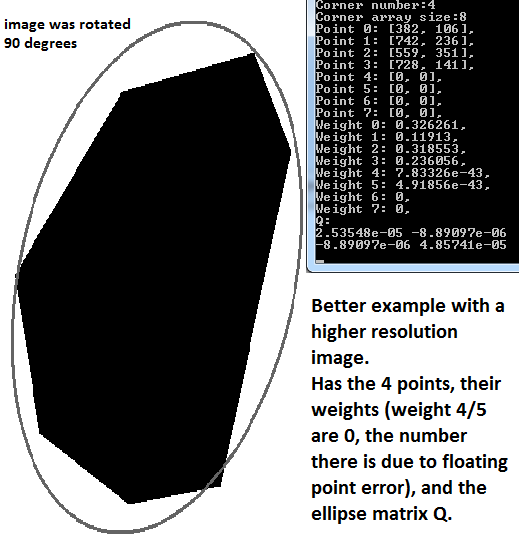
**For** j= 0 **to** <number of corner points>, **do** u[j] = u[j]\*(1-**beta**);

**If** the point we found didn’t exist, **add** a new point, and set its weight to be **beta**

**If** the point we found was an existing point, add **beta** to its weight

Repeat Part 2 until the **distance** between u and u\_old (previous iteration) is smaller than epsilon 2 (eps2). The distance is defined:

for all the **n corner points** found so far.

**Final output**

**\*Note that the final ellipse is the one BEFORE dividing by (1-epsilon) to make it fully enclosing. It is this way to allow more flexibility.**

**\*\*Drawn by deriving solely from Q the input needed for cv::ellipse()**

**Basic usage**

**imgCrawler.exe - for full usage options**

**Examples:**

**imgCrawler.exe test.bmp,**

**imgCrawler.exe 2.bmp 4 0.08 1 0 1000**

**Class Attributes**

**Mat image** – The input image on which the operations are executed. The image **must** be grayscale, target object **should** be black (0) – although the object color is configurable – and the rest of the image should be white (255)

**Mat Q** – Ellipse matrix. More details the article. Examples of usage can be found in many central methods.

**Point startingLoc** – The “start” point – used to return to the center after crawling to find new points.

**Point currLoc** – The “current” pixel in the image, on which the we “focus”.

**Point tempLoc** – Used to save a location to return to – for example when getting to a border and having to check 2 points due to a straight angle and a straight border.

**Point \* p** – Dynamic array of **corner points.** Those points are what defines the ellipse.

**Float \* u** – Array of relative weights for each **corner point.** For each point there must be a weight.

**Bool debug** – Indicates whether the program is running in debug mode.

**Int debugCounter** – Used to count unique touched pixels – currently not fully implemented.

**Int pointArrSize** – Since p/u are stored in **dynamic** arrays, pointArrSize saves current array size

**Int pointArrCounter** – p/u are actually stacks from which items are never removed. So pointArrCounter references the current number of items in the stack.

**Int color** – Target object color. For a completely black object, target color should be 0, for example.

**Double movementAngle**– The “current” movement angle – represents the direction we need to be moving at a specific point in time. In degrees between 180 -> -180

**Int mode** – Unimplemented. See main project document for possible future uses.

//Bellow are points relative to current position. You can look at it as: 8 1 2 or in the real world 6 5 4 since y starts from the upper left corner and goes down 7 0 3 7 0 3

6 5 4 8 1 2

**Functions**

**Main functionality**

**imgCrawler::imgCrawler(Mat image);**

Constructor. Takes the image you want to work on as input. Note that the image needs to already be stored in an OpenCV2 matrix.

.

**bool imgCrawler::run(int color, int eps1, double eps2, int x = 0, int y = 0);**

Runs the algorithm on the image the object was constructed with.

Color is the target image color, eps1 is the width of the initial search grid in pixels (objects smaller than eps1 x eps1 pixels might be missed), eps2 is the epsilon specific to the algorithm, and x/y are coordinates you can give as a “hint” to which column/row to start looking at – they will be checked first.

**bool imgCrawler::findStartPoint(int color, int eps, int x = 0, int y = 0);**

Finds starting point – takes color/eps/x/y from “run”.

Without x/y, will do the quadrant search described in the introduction, with x or y alone will check the relevant column/row first using binary search, with both x and y will check that pixel first, too.

**bool imgCrawler::findCorner(int num);**

**AFTER** the current angle is set, jumps to the border in that direction, then crawls from the border pixel to the farthest corner – using L2 distance from this->startingPoint.

**bool imgCrawler::jumpToBorder(Mat tempMat = Mat::zeros(1,1,CV\_32F));**

Starting from this->currLoc, performs binary hops in the direction of this->movementAngle until it’s outside of the image, then shrinking binary hops back/forward (depending on whether it’s inside the image) to find a border point inside the image.

**bool imgCrawler::crawlToCorner(Mat tempMat = Mat::zeros(1, 1, CV\_32F), bool elipsDist = false);**

Once on a **border point** (will not work otherwise), crawls to the farthest corner using either L2 distance from this->startingLoc, or distance from the center of the ellipse **only if** Q has already been initialized and elipsDist == true.

**bool imgCrawler::handleBump(bool elipsDist = false);**

The object is not really a convex body – it is at best a pixel approximation that represents a convex body. Due to this, moving along a “line”, you might actually get “stuck” on small “bumps” that aren’t there in the real world, but are there due to pixelization.

This function handles that.

**void imgCrawler::calcQ();**

Calculates Q, the matrix that represents the ellipse. All it needs is at least 1 point p with a weight u (obviously works on as many points as there are).

**void imgCrawler::setAngle(Point from, Point to);**

Sets this->movementAngle to be the (calculated) angle between pixels **from** and **to**. Otherwise, the user can manually set this->movementAngle.

**bool imgCrawler::moveCurrent(int whereTo);**

Moves current location to a nearby pixel, according to the map on the attributes page.

**Utility**

**bool imgCrawler::inShape(int where = 0, Point p = Point(-1,-1));**

Checks whether a pixel is inside the object. “where” is the relative position of the pixel according to the map on the attributes page, and “p” is the point relative to which we are checking. The default point is the current location (this->currLoc).

**Point imgCrawler::getPointAt(int whereTo = 0, Point start = Point(-1, -1));**

Returns a point relative to “start”, or current location if start isn’t passed.

Same directional rules apply as “inShape”.

**bool imgCrawler::checkPointLegal(Point p);**

Checks if point p is inside the boundaries of the image.

**bool imgCrawler::checkBorderPoint(Point p = Point(-1,-1));**

Checks if point p (default – current location) is either

within the object with one of its neighboring pixels outside the object, or

outside the object with one of its neighboring pixels inside the object.

**int\* imgCrawler::getAngleDirections(int angle = 361);**

Once this->movementAngle is set, this function returns an array of size [9], corresponding to the map on the attribute page, where each member is 1 or 0.

A member 0 means movement in **that direction** is **impossible** given the current angle.

A member of 1 means the opposite.

Can also take “angle” as a parameter.

**float imgCrawler::pointDist(Point p1, Point p2);**

Calculates the Euclidian (L2) distance between 2 points.

**float imgCrawler::elipsDist(Point p);**

Calculates the distance of a point p from the center of the ellipse which is represented by this->Q. **Q must be set**, or else this will crush!

**double getAngleData(bool convertToRad = true);**

A getter for this->movementAngle that can also convert an angle to radians by default.

**void imgCrawler::expandArr();**

Expands the dynamic arrays where p/u are stored.

**void imgCrawler::setDebug(bool val);**

Indicates the program to run in debug mode (or exit debug mode if val == false). **RECOMMENDED FOR LEARNING**

**void imgCrawler::drawCross(cv::Mat image, Point p);**

Draws a small black/white cross at point p in a given image. Used for debugging mainly.

**void imgCrawler::printState();**

A mainly debugging function that returns the current state of the class.

**float\* imgCrawler::checkDistNearMe(bool onlyBorder = true, bool onlyInShape = true, bool toPrint = true);**

A pure debugging function that returns an array that follows the same rules as **getAngleDirections,** but in each member is instead the L2 distance of that pixel from the stating point.

**bool findStartPointInLine(int color, int eps, int x, int y, int d, bool col);**

Helper function for initial starting point finding.

**bool findStartPointByQuadrats(int color, int eps, int x, int y, int dx, int dy);**

Helper function for initial starting point finding.

References:

1. [**Minimum-volume enclosing ellipsoids and core sets**](https://www.researchgate.net/publication/225220681_Minimum-Volume_Enclosing_Ellipsoids_and_Core_Sets) **– by P. Kumar, E. Alper Yuldirim, 2004**