# COMP2036 Assignment: Image Contour Extraction

Tristan Aubrey-Jones

November 20, 2006

## 1 Implementation

I have chosen to implement the Greedy Image Contour Extraction algorithm [1] using MATLAB as it has native matrix manipulation support and will enable fast prototyping. All of my MATLAB functions and code can be found in the attached ZIP file. The algorithm implementation can be found in '*greedyice.m*', '*greedyice_movecontour.m*' and '*greedyice_choosebeta.m*'. To invoke it call:

```
newcon = greedyice(con, img, alpha, beta, gamma);
```

## 2 Performance

### 2.1 Accuracy

In order to test the accuracy of the algorithm I wrote a test function, *testice*, which allows you to input a contour and run the algorithm on an image. A number of test applications of the algorithm[1] can be found in section 5.1.

The algorithm performs quite well for simple geometric shapes, although I have noted cases where the algorithm does not terminate with dense contours, as the contour points vibrate along the edges. For the forth sample the contour is identifying a triangle, but due to the density of the initial contour it is forced to 'buldge' outward as it tries to satisfy the $e_{curv}$ term of the energy functional.

When the algorithm is applied to real photographs it becomes less reliable due to the rich colour and texture variation. The first example photograph has high contrast with a cream doll on a blue background and so the algorithm performs well. When the size of the neighbourhood size $m$ is 8 it finds the very outer bound of the doll, and when $m$ is increased to 24 it accurately identifies the doll's face. Despite what might be expected the algorithm sometimes performs better with a lower value of $m$ like in the last example where the algorithm identifies the person when $m$ is 8, but not when $m$ is 24. The other real world examples are less accurate as the algorithm tends either to get caught on points external to the object, or fail to identify objects with smooth or blurred boundries.

[1] Including the implementation extensions described in 3

## 2.2 Time Complexity

In order to estimate how the time that the algorithm takes scales with $n$ (the number of points in the contour) I have written a function *icetimer*, which invokes the algorithm repeatedly using circular initial contours and calculates the average cpu time taken for each value of $n$ in a range.

The first figure in 5.2 shows a plot of some sample data collected using *icetimer*. This data does not appear to be linear, but can be fitted closely by a quadratic curve (see second figure) and so I would expect the algorithm to scale in $O(n^2)$. This is as expected as the complexity for minimising the energy functional from amoung a point's neighbours scales in $O(mn)$, and this process it repeated $n$ times for each new contour that is found. So if the convergence rate is not proportional to $n$ the algorithm can be said to scale in $O(mn^2)$.

# 3 Implementation Extensions

The main limitation with the greedy approach is that contour points get caught in local minima. This is not a big issue for synthesised images, with simple geometric shapes, but with real world photographs this is a major limitation. In order to address this issue I have altered the energy functional, by pre-processing the image matrix, to extract meaningful image data, and remove noise.

## 3.1 Edge Filter

In an image, objects can be identified by their edges which consist of sharp changes in colour or pattern. At a point on an edge the colour gradient will be high, most likely at a local maxima. We can therefore take $e_{image}$ not as the colour intensity at a given pixel, but instead as the gradient of the colour at that point. This can be approximated by summing the differences in colour (or intensity) between a pixel and its immediate neighbours:

$$eimage_{x,y} = |im_{x,y} - im_{x-1,y}| + |im_{x,y} - im_{x+1,y}| + |im_{x,y} - im_{x,y-1}| + |im_{x,y} - im_{x,y+1}|$$

This approach works for both grayscale and colour images as when applied to a colour image the image value at any point is a column vector (a point in 3D RGB space) and so the difference between values gives the scalar distance between the two colours. This makes it particularly useful as it does not matter what the colour of the object is, as long as it is different to the background, it can be identified. In my implementation the function that performs this process can be found in '$imedgefilter.m$'.

## 3.2 Noise Reduction

With most images the edge filter above returns an image of very low values, with some high values along edges. In order to estimate an object's contour within an image all that is needed is this edge data. We can therefore introduce a threshold for the $e_{image}$ values[1] ,

[1] values of $e_{image}$ are normalized to [0,1] and inverted before use in the algorithm such that edges are found at intensity maxima.

below which the value is set to 0, to eliminate noise. In my implementation the function that performs this process can be found in '$imnoisefilter.m$'.

### 3.3 $E_{area}$ Term

One algorithm limitation that I have noted is that the algorithm fails to fit concave shapes like the one in section 5.4 of the appendix. In order to try and correct this I added an $E_{area}$ term to the energy functional that would minimise the area of the contour. Computing the actual area within the contour would be very expensive and so instead I decided to minimise the distance between each point and the contour's center; thus forcing the contour to contract. The weight $\delta$ for this term is only set to 1 if the image intensity at the point is below a certain value (0 otherwise) so the contour should only contract when it finds a gap in the edge.

$$E_j = \alpha_j E_{cont,j} + \beta_j E_{curv,j} + \gamma_j E_{image,j} + \delta_j E_{area,j}$$

The effect of this extra term can also be found in section 5.4. Here you can see that although the contour has contracted to find the interior of the object, the corners have been missed off, and the contour points have bunched toward the center of the object. Considering these disadvantages I think it would be better to use this algorithm to find the outer boundry of an object, and then use a more accurate algorithm within this outer boundry to find any concave areas in the object edge.

## 4 References

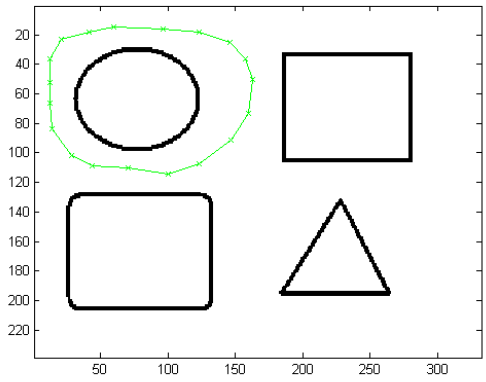
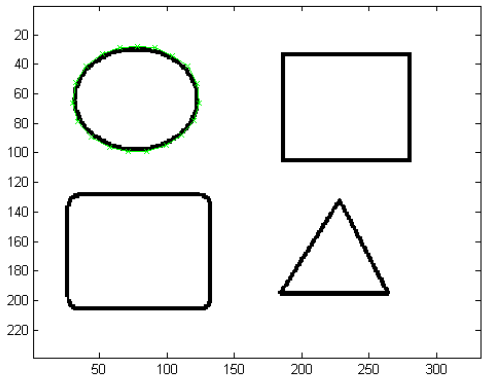
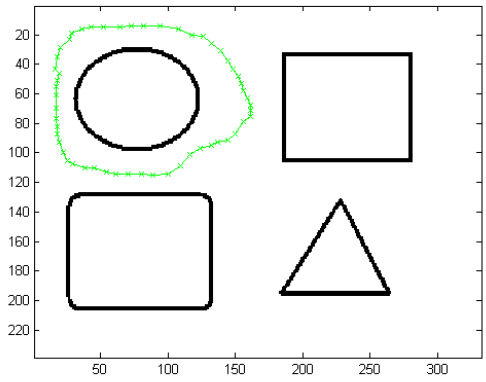
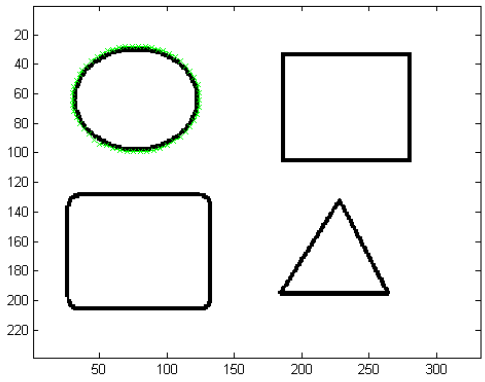
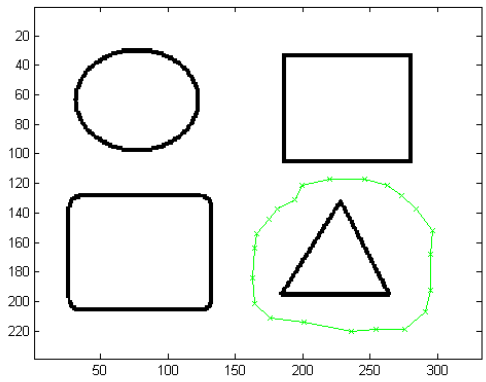
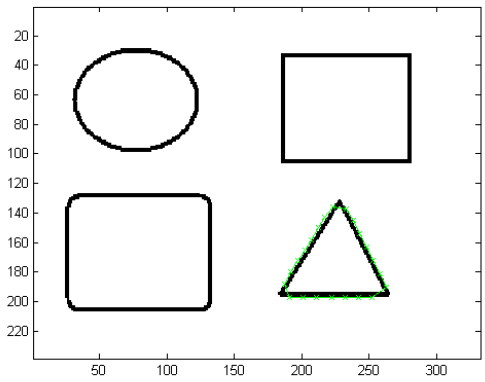
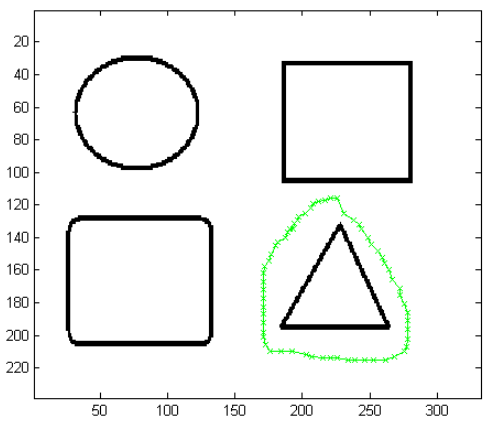
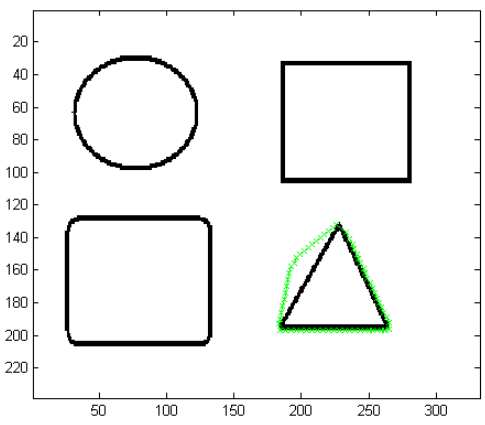
1. D.J. Williams and M. Shah. A fast algorithm for active contours and curvature estimation. *CVGIP: Image Understanding*, 55(1):1426, 1992.

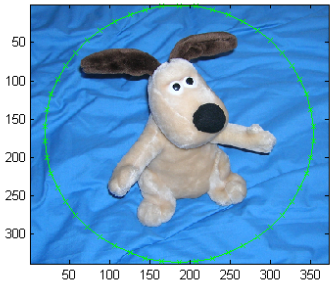
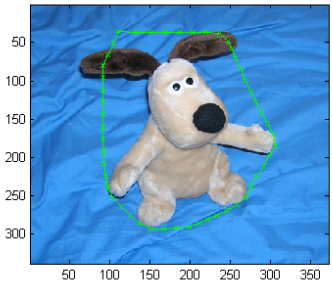
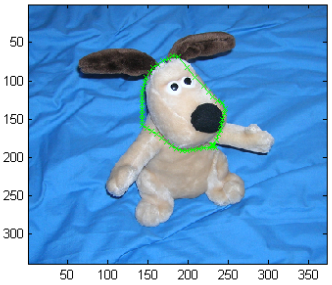
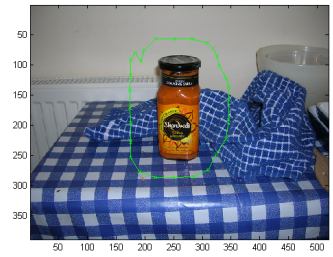
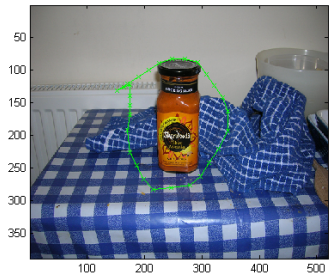
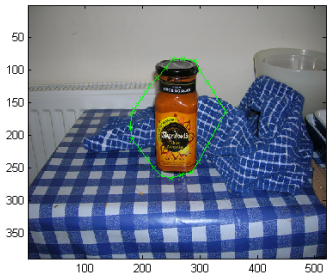
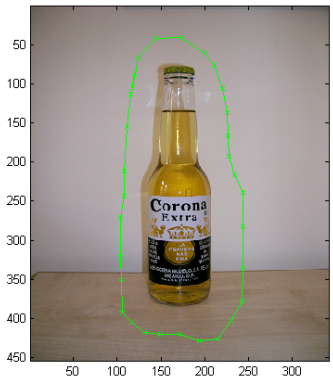
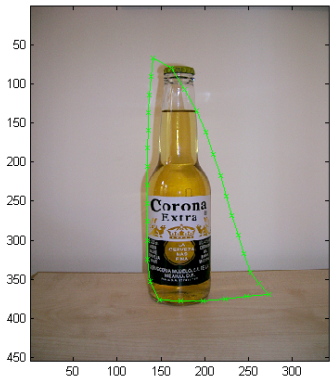
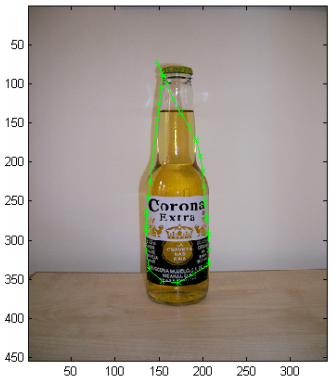
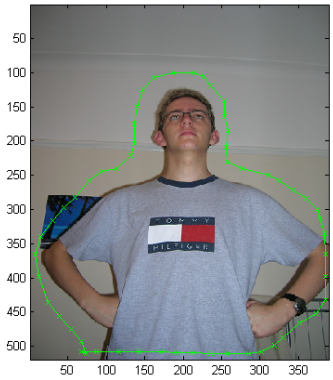
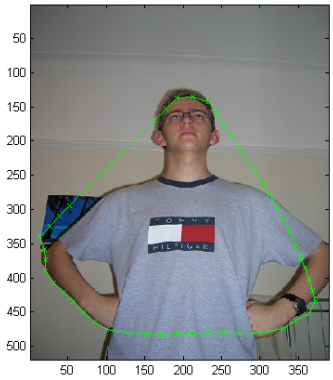
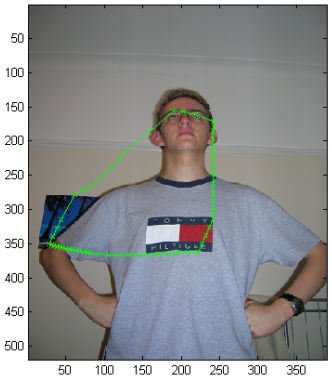
# 5 Appendix

## 5.1 Sample images

Initial contour

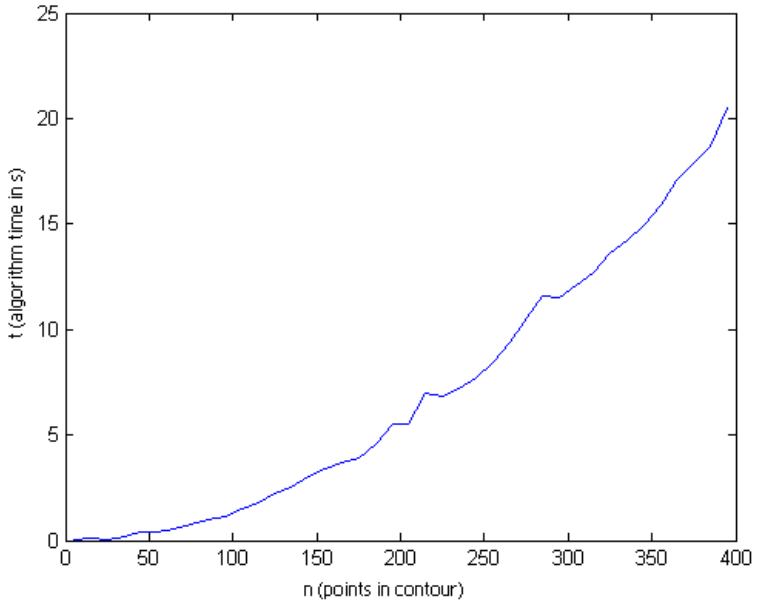
Final contour

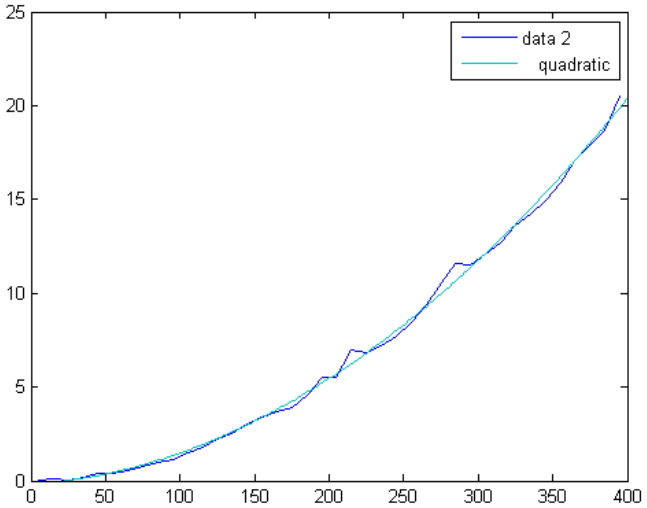
| Initial contour | $m = 8$ | $m = 24$ |
|---|---|---|
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## 5.2 Time complexity

These graphs show how the algorithm's time scales with n, the number of points in the contour:




*icetimer* sample

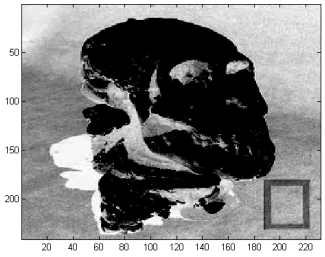



Quadratic fit of points

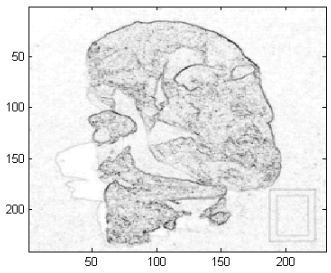
## 5.3 Image filters

The images below show the affect of applying different image pre-processing techniques in order to reduce noise and aid the algorithm in converging on the global optima; finding the object contour.

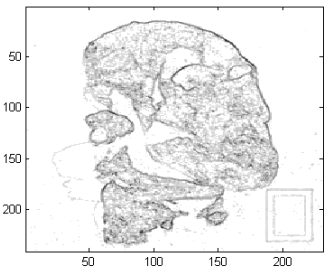
Original image



Edge filter

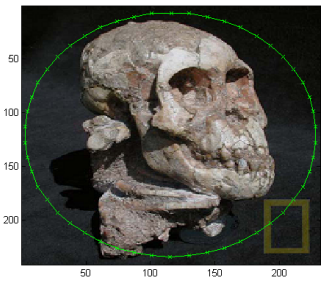


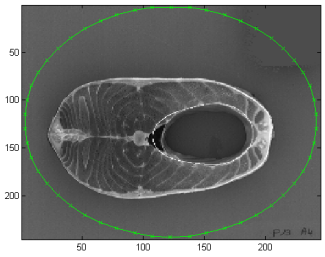
Edge and noise filters



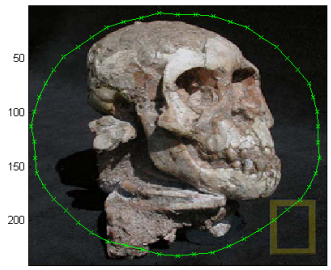
Below are the results of the algorithm run, with and without this initial pre-processing:

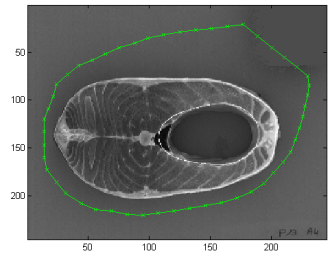
Original contour



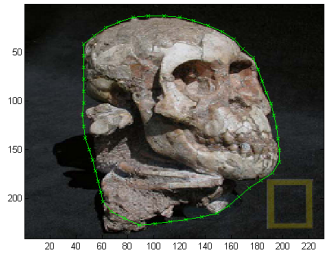


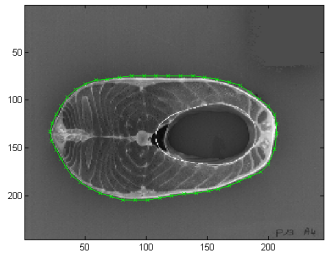
Result without filtering





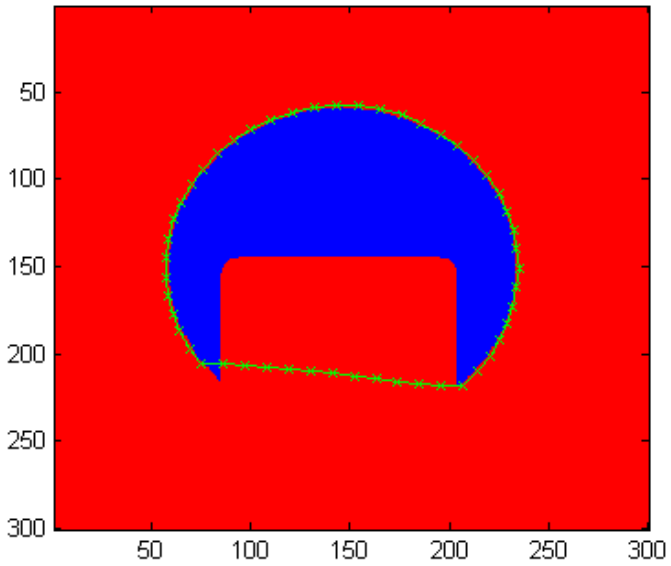
Result with filtering



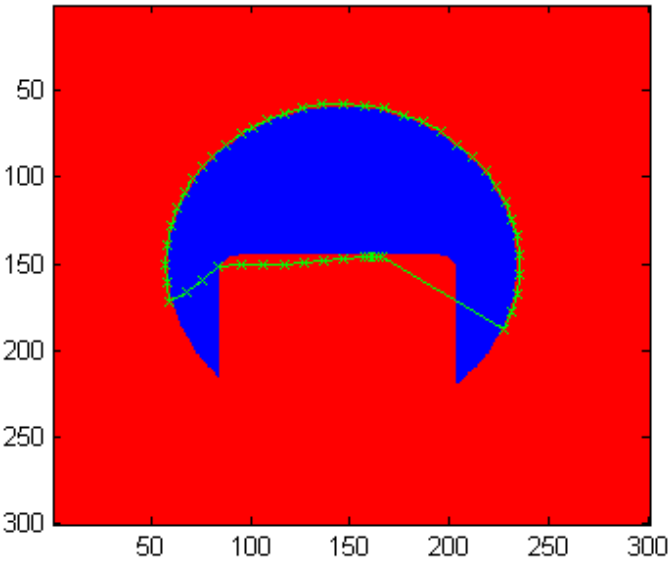


## 5.4 $E_{area}$ Term

These images show the affect of introducing a term to attract to the contour's center to energy functional.




ICE on concave shape




ICE with $E_{area}$ term