Adaptive Unsharp Masking in Image Enhancement

Student: Pieter Jan Eilers, s2381575

University: Rijksuniversiteit Groningen
Course code: WBCS13000
Primary supervisor: Dr. M.H.F. Wilkinson
Secondary supervisor: Prof. Dr. M. Biehl

June 19, 2019

Contents

1 Introduction **3**

2 Unsharp Masking **3**

3 Related Concepts **5**
3.1 Connectivity 5
3.1.1 Connectivity classes 6
3.2 Connected operators 6
3.2.1 Attribute Filters 6
3.2.2 Extensions to Greyscale 7
3.3 Max-tree Algorithm 7
3.4 Hyperconnectivity 9
3.5 Atrribute Filters Based on k-flat Zones 9
3.6 Reading and Writing Images 12

4 Methods **12**
4.1 Gaussian blurring 12
4.2 High-pass Filter 13
4.3 Max-tree Filtering 14
4.3.1 Sum of Squared Greylevels 14
4.4 Unsharp Masking Completion 15

5 Results **16**
5.1 Noise Reduction 17
5.2 Ringing Artefacts 19
5.3 Another Example 22

6 Thesis Conclusions **23**

7 Future Work **24**
7.1 Dynamic Lookup Table 24
7.2 Combination with Other Unsharp Masking Techniques 25

A Code Attachments **26**
A.1 kabsMaxTreeFilter.c 26
A.2 highPass.c 44
A.3 unsharpMasking.c 51

Abstract

Unsharp Masking is the concept of using high-pass filtering to enhance an image, this process adds a factor of the high-pass filtered version of the image to the original image. The main drawback to this method is that any noise present

in the image will also get enhanced. Connected filters are a form of morphological operators, used in morphological image analysis, based on *connectivity* and *hyperconnectivity*. In this thesis I have adapted an existing Max-Tree algorithm with new filtering methods that are better at detecting noise. I then apply this algorithm to several images of the sun to see that the amount of noise and artefacts is indeed reduced using this adapted algorithm.

1 Introduction

When we take a look at images produced in certain fields like astronomy, we would like these images to be sharp and have clear detail, however in reality this is often far from the case and images are blurry and unclear. Unsharp Masking is an existing technique used to increase the sharpness of an image, in other words, it will accentuate the details of an image. This technique however is far from perfect and has a number of drawbacks, the main drawback being that any noise present in the image will also be accentuated. There exist a multitude of ways to adapt and improve this technique and in this thesis I will focus on one specific technique, namely using connected filters in combination with max-trees and k-flat zones. Connected filters are morphological operators acting on connected components, think of openings and closings using structuring elements. A max-tree is a way to construct a hierarchical region-based representation of an image, which can then be easily filtered and then the image reconstructed. A k-flat zone is defined as a connected region of maximal extent, in which the total greylevel variation is no more than k [3].

In essence, the question this Bachelor's Thesis should answer is: "Can connected filters be used to improve on Unsharp Masking?"

To achieve this goal I adapted some existing code provided by [3] and developed some of my own algorithms, mainly for the Gaussian blurring and the process of Unsharp Masking itself. All was done in C, which means run-times are swift but we have to manage memory and make sure there are no leaks.

2 Unsharp Masking

Unsharp Masking is the process wherein a blurred version or mask of the original image is combined with the original image. It improves the visual quality of the image using high-pass filtering and dynamic range compression. In more detail, if we consider the original image f, we can produce a low-pass filter of f by averaging each pixel with its neighbour pixels, using a kernel that is based on a Gaussian curve, also known as Gaussian blurring. Then we can subtract this low-pass filter from the original image, the result is called a high-pass filter. This result can then be multiplied by a factor β and then be added to the original image. The β represents the factor of sharpening. A higher β results in a sharper image, however this will also increase potential noise. This can be

more formally presented as

$$f' = \alpha * f + \beta * (f - L_p(f)) \tag{1}$$

where f' is the new image after Unsharp Masking and $L_p(f)$ is the low-pass filter or Gaussian blur of the original image f.

There are two main drawbacks of Unsharp Masking. Firstly, it will amplify all the details, including any noise. Secondly, it will leaving ringing artefacts around objects with high contrast in the image. In this document I will primarily be focusing on images from astronomy. These images have in common that there usually are one or more bright components combined with a darker background containing potential noise.

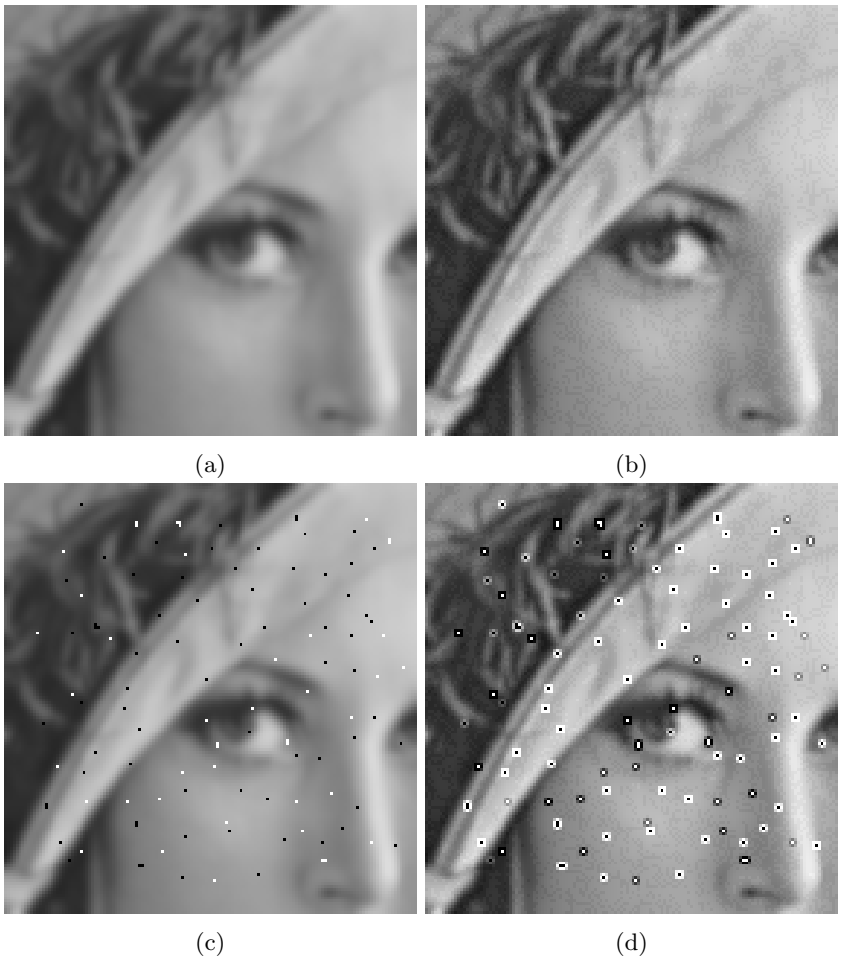


Figure 1: Effects of Unsharp Masking on a blurred image (a) Cropped part of the classic Lena image (b) After Unsharp Masking (c) Added noise (d) Effects of Unsharp Masking on noise.

3 Related Concepts

3.1 Connectivity

In the context of discrete image analysis, the set-theoretic concept of connectivity describes the way pixels are grouped to form so-called connected component or flat-zones. Connected components in a greyscale image are regions of constant intensity (equal greyscale value) in which pixels are characterised by path-wise connected relation. In a 2-dimensional discrete space $\mathbb{Z}^2$, pixels are typically either 4 or 8-connected [4]. 4-connected means a pixel is connected horizontally and vertically, in terms of pixel coordinates this means every pixel that has the

coordinates $(x \pm 1, y)$ or $(x, y \pm 1)$ is connected to the pixel at (x, y), whereas 8-connected pixels are also connected diagonally, meaning every pixel that has the coordinates $(x \pm 1, y \pm 1)$ or is connected to the pixel at (x, y).

3.1.1 Connectivity classes

We can use this idea of connectivity in the context of discrete image analysis to group pixels into meaningful structures. A common way to address this is in the context of set-theory and mathematical morphology is through *connections* or *connectivity classes* as described in [4][3].
Definition 1: Let E be an arbitrary non-empty set. A connectivity class or connection $\mathcal{C}$ on E is any family $\mathcal{P}(E)$ that satisfies:

1) $\emptyset \in \mathcal{C}$ and for all $x \in E$, $\{x\} \in \mathcal{C}$
2) for any $C_i \subseteq \mathcal{C}$, $\bigcap_i \neq \emptyset \Rightarrow \bigcup_i C_i \in \mathcal{C}$

This definition shows that both the empty set and the singleton sets are connected and any union of sets which have a non-empty intersection is also connected. A member of $\mathcal{C}$ is called a *connected set*. Connected components or grains are accessed by means of a *connectivity opening* Γ_x, which is an operator which when given a point $x \in E$ extracts the union of all connected sets within an image X that have x in their intersection [4]

$$\Gamma_x(X) = \bigcup \{C_i \in \mathcal{C} | x \in C_i, C_i \subseteq X\} \tag{2}$$

for every $X \subseteq E$, yielding the largest connected subset of X which contains x.

3.2 Connected operators

Connected morphological operators are a set of powerful and computationally efficient tools used in image filtering. Classic examples are extensive and anti-extensive reconstructions using a structuring element, also known as openings and closings. Typically, a connected operator acting on an image is a transformation that enlarges the partition of the space created by the flat zones of the image [8].

3.2.1 Attribute Filters

In most definitions, a morphological operator is called a filter if it is idempotent, i.e. $\Psi(\Psi(X)) = \Psi(X)$. Attribute filters come in in two varieties, increasing and non-increasing. An filter is increasing if for any two sets $X, Y \subseteq E$, $X \subseteq Y$ implies $\Psi(X) \subseteq \Psi(Y)$ [3]. Whereas with a non-increasing filter this is not necessarily the case. In attribute filters, connected components are either preserved unmodified if they meet an attribute criterion Λ or removed if they do not. This means no new components can be added. Attribute criteria for connected components $C_x \subseteq X$ are usually given in the form of

$$\Lambda(C_x) = (Attr(C_x \geq \lambda) \tag{3}$$

Where *Attr* is some real-value attribute of C_x and λ an attribute threshold, implemented by means of a trivial opening $\Gamma^\Lambda : \mathcal{C} \rightarrow \mathcal{C}$. For a connected component C_x, Γ_Λ returns C_x if $\Lambda(C_x)$ is true and $\emptyset$ otherwise [4][3].
Definition 2: The binary attribute opening Γ^Λ of a set X with inceasing criterion Λ is given by

$$\Gamma^\Lambda(X) = \bigcup_{x \in X} \Gamma_\Lambda(\Gamma_x(X)) \tag{4}$$

If Λ is non-increasing, we refer to it as an *attribute thinning* rather than an attribute opening [9].

3.2.2 Extensions to Greyscale

Connected filters extend to greyscale by threshold superposition [2]. Given a grey-scale image $f : E \rightarrow \mathbb{R}$, thresholding f in an increasing order from h_{min} to h_{max} produces a stack of nested binary sets where each binary image at level h is given by

$$T_h(f) = \{x \in E | f(x) \geq h\} \tag{5}$$

and for any two levels $h_1 < h_2 \Rightarrow T_{h_1}(f) \subseteq T_{h_1}(f)$. Now the response of the greyscale counterpart of a binary increasing filter $\Psi^\Lambda(x)$ on each point x is given by

$$\Psi^\Lambda(f)(x) = \sup\{h | x \in \Psi^\Lambda(t_h(f))\} \tag{6}$$

This operator Ψ^Λ will assign to each x the highest threshold at which it still belongs to a connected component which satisfies a certain attribute criterion Λ.
Now we can define three types of components that are used for defining connected filter in greyscale. Given a greyscale image f, a *peak component* P_h at level h is a connected component of the thresholded image $T_h(f)$, meaning that for some x

$$P_h^x(f) = \Gamma_x(T_h(f)) \tag{7}$$

and a *regional maximum* M_h at level h is a component or flat-zone which has no neighbours a larger intensity that h. A flat-zone F_h is a connected component of the set of pixels with level equal to h, i.e. [3][9].

$$F_h^x(f) = \Gamma(\{x \in E | f(x) = h\}) \tag{8}$$

Greyscale attribute openings act on peak components while connected filters generally act on flat-zones [8].

3.3 Max-tree Algorithm

One way of efficiently implementing an anti-extensive attribute filter is to use tree-based algorithms for greyscale image representation, originally introduced by Salembier et al. [7]. The Max-Tree algorithm is a process in which the construction of the tree and computation of node attributes is independent from the filtering process. This tree structure encodes the set-theoretical notion

of connectivity and its greyscale extension. Given a greyscale image f. the tree structure represents the nesting order of its peak components P_h^i. The nodes C_h^i, at level h with index i, correspond to sets of pixels for which there exists a unique mapping to peak components [3].

$$C_h^i = \{x \in P_h^i | f(x) = h\} \tag{9}$$

Every Max-Tree node except for the root will point to its parent at level $h' < h$. The root node points at level h_{min} to itself. Max-tree nodes contain four important fields: *Level* storing the original level h, *NewLevel* storing the new value after filtering. *Attribute* containing the value used for filtering, and *Parent* which points to the address of its parent [3]. The tree is recursively constructed by means of a flooding function using hierarchical FIFO queues as described in [4][3][9]. It is initiated by first accessing the root pixel from the queue at level h_{min} and then proceeds with flooding nodes along the different root paths that emerge during this process. This function, upon receiving a pixel from the queue, updates the *Attribute* value of the node currently being flooded, it then inspects its neighbouring pixels and places them in the appropriate queue. If a neighbouring pixel is at a higher level $h' > h$, flooding proceeds recursively at h'. When a node is fully flooded, it will be finalised by determining its *Parent* and *Attribute* fields. The process terminates when all pixels have been processed.

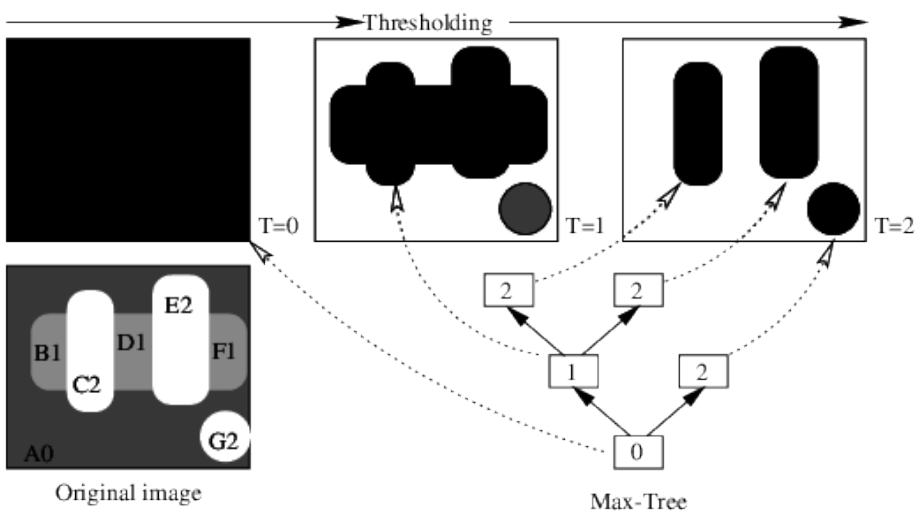


Figure 2: Max-tree representation of a greyscale image, original image from [10].

After the flooding is completed, we can start the filtering process using a filter function which visits each node, starting from the root. At every node, the *Attribute* field is used to evaluate Λ against a given thresholdλ. Nodes failing this criterion are removed by lowering the greyscale value of its member pixels. By what degree these values are lowered depends on the filtering rule chosen.

3.4 Hyperconnectivity

Before we explain k-flat zones, we have to introduce the theory behind it, i.e. hyperconnectivity [11, 1]. Hyperconnectivity extends the notion of connectivity by relaxing the second condition of Def. 1 where instead of needing a strict non-empty intersection of sets for their union to be connected, *hyperconnectivity classes* involve a degree of overlap, specified by an *overlap criterion*.

Definition 3: An overlap criterion in $\mathcal{P}(E)$ is a mapping $\perp$: $\mathcal{P}(\mathcal{P}(E)) \rightarrow \{0,1\}$ such that $\perp$ is decreasing, i.e. for any $\mathcal{A}, \mathcal{B} \subseteq \mathcal{P}(E)$

$$\mathcal{A} \subseteq \mathcal{B} \Rightarrow \perp(\mathcal{B}) \leq \perp(\mathcal{A}) \tag{10}$$

Any $\mathcal{A} \subseteq \mathcal{P}(E)$ for which $\perp(\mathcal{A})$ is 1 said to be overlapping, otherwise $\mathcal{A}$ is non-overlapping. We can interpret (10) as ensuring that a non-overlapping family cannot become overlapping by adding more sets. Next, we define the hyperconnectivity class [3].

Definition 4: A hyperconnectivity class $\mathcal{H} \subseteq \mathcal{P}(E)$ with an overlap criterion $\perp$ is a family of sets with the following properties:

1) $\emptyset \in \mathcal{H}$ and for all $x \in E$, $\{x\} \in \mathcal{H}$

2) for any $\{A_j\} \subseteq \mathcal{H}$ for which $\perp(\{A_j\}) = 1 \Rightarrow \bigcup A_j \in \mathcal{H}$

Members of hyperconnectivity class are called hyperconnected. Now we can use this underlying theory of an overlap criterion and use it to define a more practical example, namely k-flat zones. This is the most relevant definition since the filters used in this project are designed around the concept of k-flat zones.

Definition 5: A k-flat zone $F_{h,k}$ at level h and depth k is the set of all path-wise connected pixels, marked from $x \in E$, with intensities from h down to $h-k$.

$$F_{h,k}(x) = \Gamma_x(\{p \in E | h-k \leq f(p) \leq h\}) \tag{11}$$

A 1-D example is shown in Fig. 2, using a k of 2. See [3] to see how a hyperconnectivity class can be derived from k-flat zones.

3.5 Atrribute Filters Based on k-flat Zones

If we replace regular peak components with k-peak components, we have to consider what the k-flat-zone equivalent of a regional maximum constitutes. A regional maximum at level M_h at level$h$ is a 0-flat zones with neighbours of strictly lower greylevel. A k-regional maximum $M_{h,k}$ is a k-flat zone at level h which has neighbours of greylevel strictly smaller than $h-k$.

When we consider the threshold decomposition of a greyscale image f. A component $P_{h,k}$ is preserved if $Attr(P_{h-k}) \geq \lambda$ and this decision will propagate to all regular peak components from P_{h-k} up to P_h, independent of whether each one of them separately satisfies Λ or not. It is obvious that k-peak components are only defined from k levels above the background. Regular peak components P_h which are not members of some k-peak component, i.e. $P_h + k = \emptyset$ are then

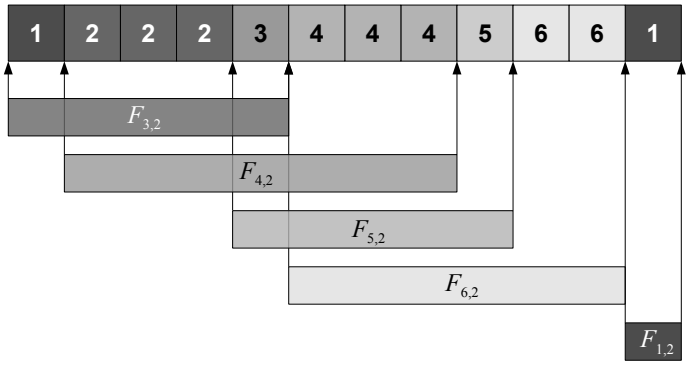


Figure 3: A signal and its k-flat zones for k = 2, each shaded box and associated arrows indicate a 2-flat zone, original from [3].

rejected because of low contrast. A modified criterion Λ_k can be derived from any criterion Λ used for regular Max-Tree filtering as

$$\Lambda_k(P_h) = \begin{cases} \Lambda(P_h) & \text{if } \exists x \in P_h L : f(x) \geq h + k \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

This first checks if the contrast within a peak component is sufficient, and only then computes $\Lambda(P_h)$. However, just modifying this criterion is not sufficient because it violates idempotence. For example, let P_h be preserved because it meets Λ_k but its descendants do not. If these descendants are not restored up to a level k above the current P_h, another application of the filter would remove P_h because it automatically fails Λ_k. The classical subtractive Max-Tree filtering rule is therefore modified, in a way that propagates preserve decisions based on Λ_k upwards within a range k above the preserved ancestors. This is known as k-subtractive filtering which is extensively explained in [3], the pseudocode for this k-subtractive filtering is shown below.

Algorithm 1 The k-Subtractive filtering rule.

```
procedure kSubtractive (var t : MaxTree , λ : AttributeType, k : GreyLevel)
    /* input: Max-Tree t of an image f, attribute threshold λ and k-flat value */

    var root, node, parnode ; /* type MaxNode, pointers to root, current node and its parent */
    var l, difflevel ;        /* type GreyLevel, level and level difference */

    /* process root */
    root = t.root
    l = root.Level    /* sets l = h_min */
    if root.Attribute > λ then
        root.kprime = k /* preserve and set maximal k-restoration level */
        root.NewLevel = root.Level
    else
        root.kprime = 0
        root.NewLevel = 0
    endif
    for all levels l starting at h_min + 1 to h_max do
        for all nodes node at each level l do
            parnode = node.parent
            difflevel = node.Level − parnode → Level
            if node.PeakLevel − parnode.Level > k AND node.Attribute > λ then
                /* preserve and set maximal k-restoration level */
                node.NewLevel = parnode.NewLevel + difflevel
                node.kprime = k
            else
                if difflevel > parnode.kprime then
                    node.NewLevel = parnode.NewLevel + parnode.kprime
                    node.kprime = 0
                    /* k-restoration completed */
                else
                    node.NewLevel = parnode.NewLevel + difflevel
                    node.kprime = parnode.kprime − difflevel
                    /* remainder to be k-restored */
                endif
            endif
        endfor
    endfor
    for all pixels p restitute the filtered image
```

3.6 Reading and Writing Images

For the purpose of reading and writing images I use a library called `freeimage.h` and two functions called `readTIFF(..)` and `writeTIFF(..)`. These two functions are provided by the code received from Prof. Wilkinson. `readTIFF(..)` has the ability to read in an image with multiple channels or planes, i.e. an RGB image. The way it does this is by using a 2D array for the image, where the rows of the array represent the number of planes. The image itself is then read in as a 1D array, per plane. For the purpose of this project, only greyscale images are used, which only have 1 channel. Having a 1D array representation of an image can be tricky, so one could change `readTIFF(..)` to read in the image as a 2D array, I decided not to do this, because this would remove a dynamic element.

4 Methods

For the purpose of this Bachelor Project we would ideally be able to remove all noise and unwanted artefacts without influencing the important parts of the image we want enhanced after the Unsharp Masking process is done. In reality removing all unwanted noise and artefacts is likely impossible so instead we will aim to reduce it as much as possible.

Before we start using any attribute filtering we have to understand what the formula shown in (1) means. If we take a look at the right part of (1), $f - L_p(f)$, means we subtract the low-pass or blurred version of the original image from the original image, this process is called high-pass filtering. At pixels with high contrast to its neighbours, this is going to leave these neighbouring pixels with either high positive or negative values, when this part is then added to the original image, is where ringing artefacts may appear. A bright star on a darker but not completely black background for example.

Noise in the background means that in the high-pass filter of f pixels are going to have low positive or negative values, when these get multiplied by a significant β and added to the original, we can get noise amplification. So we realise that we have to use our max-tree algorithm with attribute filtering on high-pass filter of $f(1)$. In the following section we will first describe the blurring method.

4.1 Gaussian blurring

There are already many tools developed to produce a Gaussian blur of an image, like *OpenCV* in Python or when using *GIMP* or *Photoshop*, however I decided to develop my own v ersion of Gaussian blurring in C, mainly because OpenCV does not work well with 16-bit images, which is what I will mostly be using. Also it makes for quicker usage and easier debugging than when using a tool

like GIMP or Photoshop. The idea of the Gaussian blur is fairly simple, first we can build a Gaussian kernel x wide by using this formula:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}$$

for 1 dimension where the kernel size must be must be uneven and this formula:

$$G(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

in 2 dimensions where the length and width of the kernel must be uneven. However the first part of both formulas is not necessary in practice, since we normalise the kernel. This is what happens in the `double *generateKernel(double stdDev)` part of the program `highPass.c` in the appendix. Before we use this kernel for the Gaussian blurring, we can use the fact that the kernel is symmetrical and its rank is 1, this means we can split up our 2D convolution, for example:

$$\frac{1}{36} * \begin{bmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{bmatrix} = \frac{1}{36} * \begin{bmatrix} 1 & 4 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 4 \\ 1 \end{bmatrix}$$

So instead of convolving with a 2D kernel, we can do it in 2 passes with a 1D kernel, first in the horizontal direction, and then in the vertical direction. This reduces the complexity from $O(k^2 w_{image} h_{image})$ to $O(k w_{image} h_{image})$, where h is height, w is width and k is the length of the kernel. Because we work with an integer-based representation of greylevels, this only works when every value in the kernel is > 1, so we divide the complete kernel by the smallest value, after blurring we divide the result by the sum of the complete kernel to normalise again. So in the first pass, for every pixel (x, y) we multiply $(x + i, y)$ by the value of the kernel with the same offset, starting in the middle, with i in range $(-k/2, k/2)$, where k is the length of the kernel. and then add these results to the pixel. In the second pass we do the same thing, only in the vertical direction, so for $(x, y + i)$. Using this method we can generate a Gaussian blur, however we first have to realise what to do with edge cases.

There are several methods we can use to deal with the edge cases, the method I chose was to extend the image at the edges by $k/2$ where k is the length of the kernel. We let the extended edges have the pixel values of the old edges, this is done in the function `extendImage()`, now we can convolve the image with the kernel without any problems by only looping over the pixel values of the original image and applying the Gaussian blur as explained above. One thing to realise is that after blurring in one direction, the resulting image has to be extended again before we blur in the other direction.

4.2 High-pass Filter

Looking at the right part of (1), $f - L_p(f)$ is going to leave us with positive or negative values, however, an image can not contain any negative values, so

we save our our positive and negative values to images representing the positive part and an image representing the negative, by multiplying the negative values by -1. Then we can use these images in our Max-tree filtering algorithm. These values will usually be quite small, so the resulting image might be mostly black. For debugging purposes, it can be useful to multiply all values by a certain β, however if we do that, we have to make sure to not overflow, as shown in the function `void highPass()`.

4.3 Max-tree Filtering

Now that we have our positive and negative parts, we can start our filtering process using the max-tree filtering algorithm. As explained before, the flooding algorithm, `int flood(..)`, builds our max-tree. The original filtering algorithm `void MaxTreeFilter(..)` uses area attribute filtering, however, area is not the attribute we care about, so we choose a different attribute. For the purpose of this project I have chosen the sum-of-squared-greylevels attribute.

4.3.1 Sum of Squared Greylevels

If we make the assumption that for an object or group of pixels to be seen as something other than noise, there has to be a certain amount of contrast within it. While in regular k-flat filtering, if a peak component is not part of a k-peak component, this means there is not enough contrast and the node is removed, however, in many cases, we also want to remove a node when the contrast is too high. Another way to measure contrast is to, per node, track the of sum of squared greylevels. So whenever a new pixel is processed we increase the `SumGreyLevel` variable by `h*h` with `h` being the current greylevel. This process ensures that pixels with higher greylevels will have significantly higher attribute values.

We adapt our filtering algorithm to properly deal with this new attribute. In `void MaxTreeProcessNode(..)` we first calculate the propagation range `k` by checking if the sum of squared greylevels is high enough for a certain node, if this is not the case k becomes 0, if it is high enough `k` becomes equal to the `k` given as an argument. Now we can use this `k` to check if, for a certain node, the difference between its `PeakLevel`, and its parent's `NewLevel` is larger than `k`. When this is the case, we change the node's `NewLevel` to that of its parent. This ensure that things like ringing artefacts, which will usually have a high enough sum of squared greylevels will also get removed.
One thing to realise is that this operation is not actually idempotent, so technically it is a connected operator and not a filter. However we can reach a sort of idempotency by repeating the process until no more change is found.

4.4 Unsharp Masking Completion

After our max-tree filtering, all that is left is to complete our Unsharp Masking so it becomes visible what kind of effects our adapted Max-tree filtering algorithm has delivered on the final result. This part is pretty trivial since we know our images representing our positive and negative values are completely disjoint, meaning $Pos \cap Neg = \emptyset$ where Pos represents the positive values after high-pass filtering and Neg the negative values. So all we have to do is add the positive and negative parts multiplied by a given β to our original image which can be confined into one line,

$$ORI(x, y) = ORI(x, y) + \beta * (Pos(x, y) - Neg(x, y)) \tag{13}$$

where the limits for a pixel value are $(0, N)$. So if any underflow or overflow occurs, we flatten to 0 and N respectively. N in a 16-bit greyscale image is $2^{16} - 1$.

5 Results

Firstly, we look at the effects of our algorithm on a normal greyscale image, specifically the classic Lenna picture. This image is an 8-bit greyscale image, using a β of 2, the image already has some large changes in contrast. When we apply our filtering using sum of squared greylevels, using a Λ of 200, we can see that it does remove some of the noisiness in the face and background. However, when we look at her hat, some of the low contrast parts have also been reduced in sharpness.

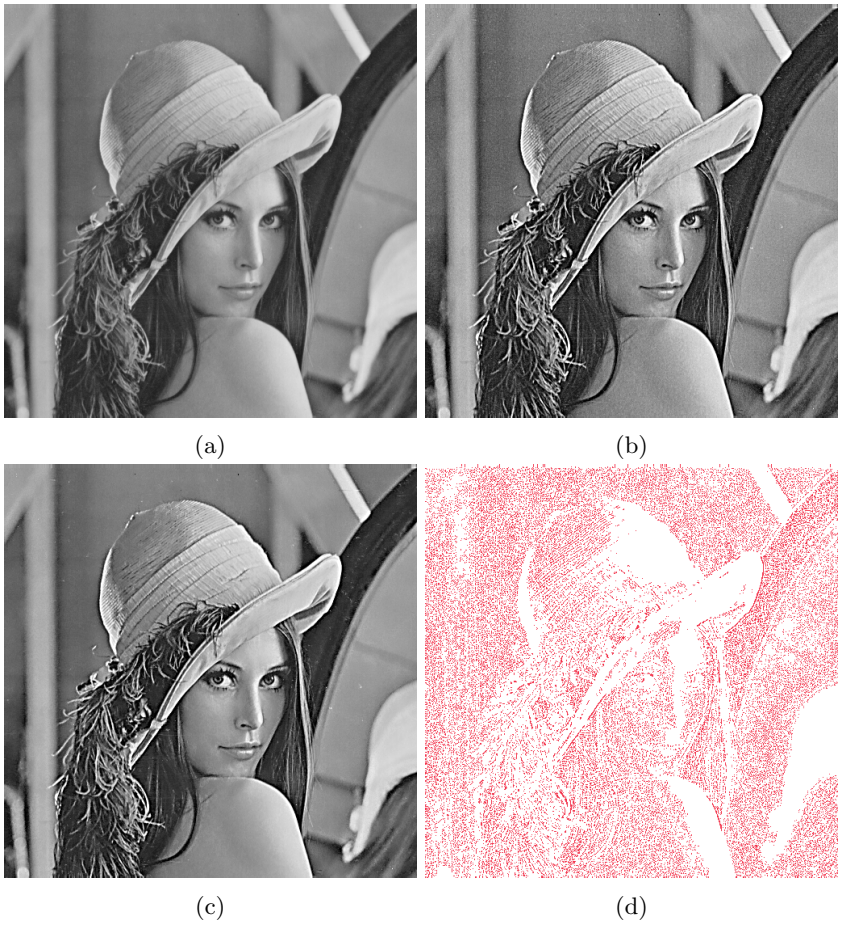
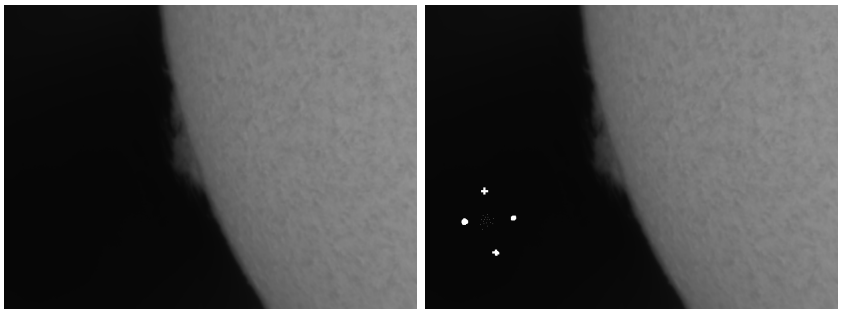


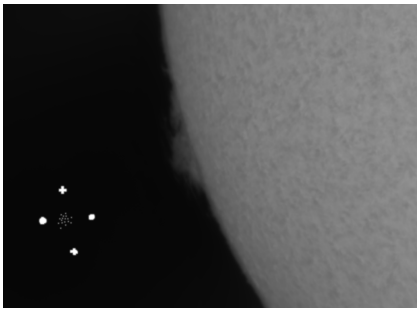
Figure 4: (a) Lenna Image in 8-bit greyscale (b) Unsharp Masking using a β of 2 without any filtering (c) Unsharp masking after filtering (d) difference between b and c

While the filtering can have a visually pleasing effect on images like the Lenna picture in Fig. 5, these images are not very useful for the purpose of

research, so for the following experiments I will use an image showcasing a blurry view of surface of the sun against an otherwise black background. Also some adaptations of this image, adding artificial stars.



(a) (b)



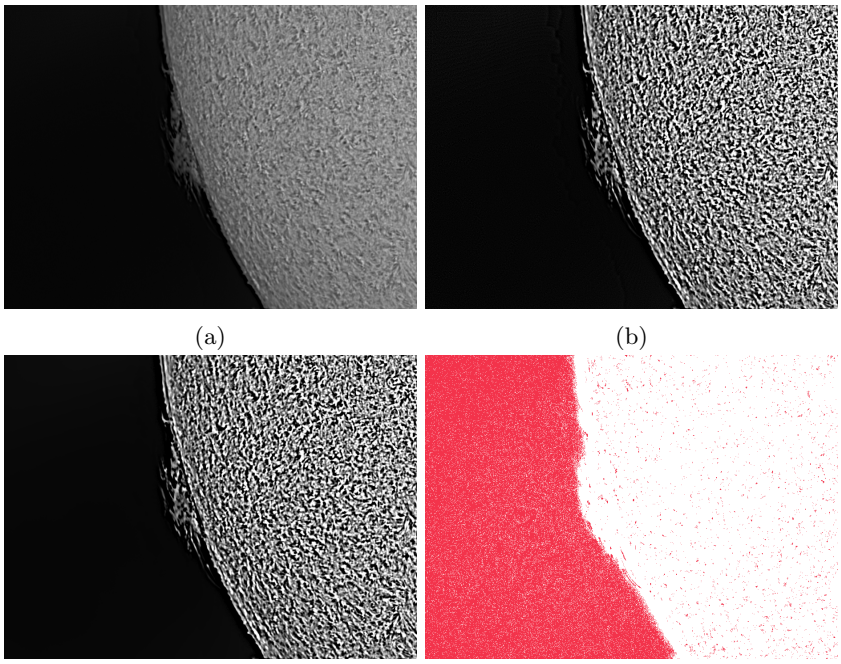
(c)

Figure 5: (a) Image of surface of the sun, (b) Image of surface of the sun with added stars, (c) Blurred stars.

5.1 Noise Reduction

Firstly, lets take a look at our unaltered image and see what happens when we execute the Unsharp Masking process without any filtering. Looking at Figure 7a. we can see that using a β of 25, the sun already shows more detail and the noise in the background is hardly noticeable, however there is definitely some noise present. If we pump up the β to 150, the sun obviously starts to look weird, however we can see more clearly the noise present in the sharpened image.

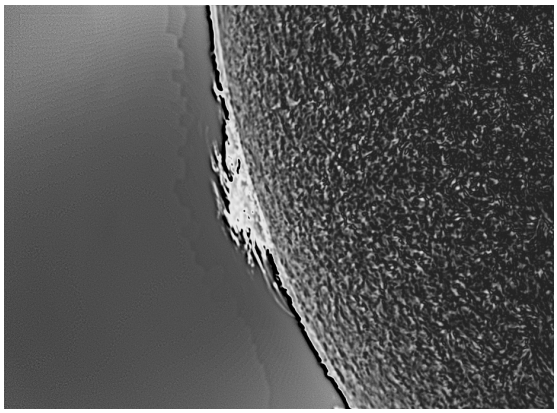
When we apply our max-tree filtering, using a λ of 7500, we can see that most of the noise gets reduced to almost invisible status. This is especially clear when we look at the difference in Figure 6d. This example has already given us an idea of the effects our filtering technique can have on noise present in the original image, however it is still hard to see with such a dark background.



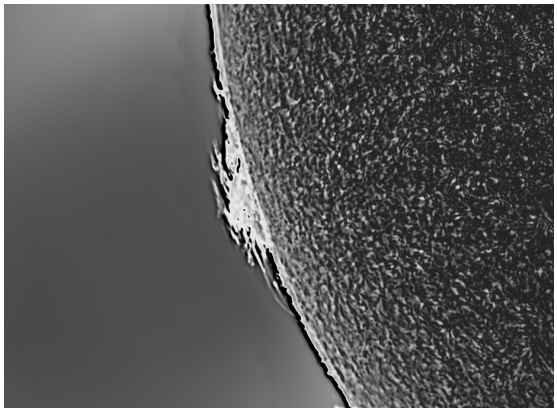
(a) (b) (c) (d)

Figure 6: (a) Unsharp Masking using a β of 25, (b) Unsharp Masking using a β of 150 (c) New version after filtering (d) difference between b and c

To give a more visually interesting look at the effects we can, using a tool like *GIMP*, switch around the colour curves. This changes the values of the lookup table of the image, such that high values are lowered in intensity and vice versa, this means that bright regions will become darker and darker regions, like the background, will become brighter, which will make it easier to see the difference between the filtered and unfiltered image. Taking a look at Figure 7a, which is the same unfiltered Unsharped Masked image as Figure 6a, only the colour curves are swapped after the process is done, we can clearly see the noise that is present in the background. However, after filtering, we can see a sharp decrease. There is definitely still some kind of noise present, as we can see some gradient difference in the background, which can probably be attributed to Poisson noise. However, this noise is already present in the original image. Another side effect is the large ringing artefact surrounding the edges of the sun.



(a)



(b)

Figure 7: (a) Unsharp Masking using a β of 25 with swapped colour curves (b) New version after filtering

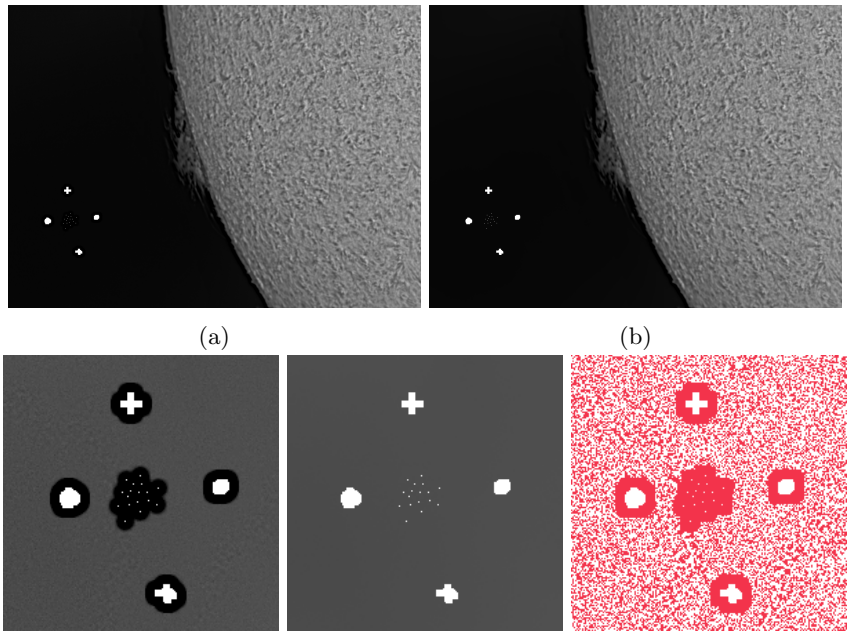
5.2 Ringing Artefacts

After the Unsharp Masking process is done, ringing artefacts can appear around objects with large contrast to the background, like our artificial stars in Figure 5b. Larger artefacts mean larger kernel size used in the blurring of the image. For our example in Figure 8, a kernel size of 15 is used. The stars in Figure 8a have these artefacts around them, they are hardly noticeable however, since the background is already very dark. When we switch around our colour curves again and take a look at Figure 8c, we can see more clearly the ringing artefacts that are left after Unsharp Masking. These artefacts come from amplifying the negative part of the high-pass filtering stage, since they appear around bright object on a dark background. So we have to choose an appropriate k to use when filtering this negative part. As we can see In Figure 8d, these artefacts have been completely filtered out, thanks to our k-flat filtering, using a k of

2000. Figure 8e showcases the difference between the original negative part of the high-pass and the filtered one.

One thing to note is that the artificial stars added to the image are completely clear and sharp and do not have the same kind of blurring as the sun has. So we have to wonder if we can still filter out artefacts when we apply some kind of blurring to these celestial bodies. Taking a look at Figure 9a and 9b, it is clear that it can indeed still work with the level of blurring used.

One thing to note is that the artificial stars added to the image are completely

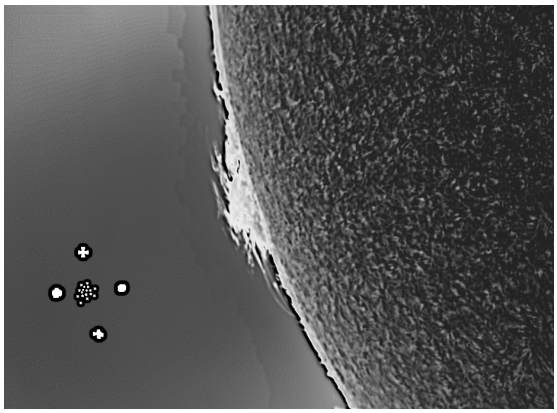


(a) (b)

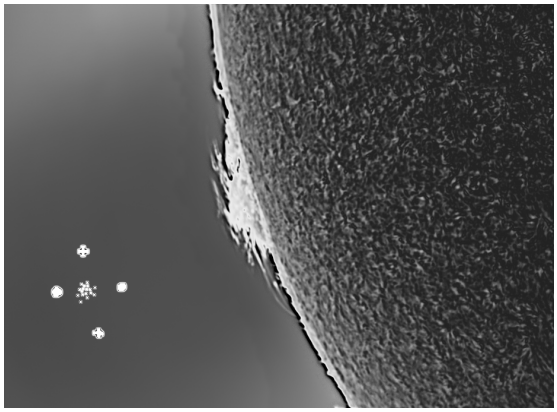
(c) (d) (e)

Figure 8: (a) Unsharp Masking using a β of 15, (b) New version after filtering, (c) zoomed in part of a, (d) Zoomed in part of b, (e) Difference between the negative parts of the high-pass filter of Figure 5b.

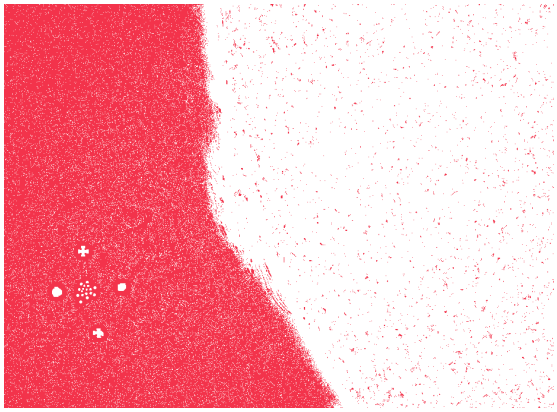
clear and sharp and do not have the same kind of blurring as the sun has. So we have to wonder if we can still filter out artefacts when we apply some kind of blurring to these celestial bodies. Taking a look at Figure 9a and 9b, it is clear that it can indeed still work with the level of blurring used. However we have to use a fairly high k, 27500 in this case, when filtering the positive part, because these artificial stars still have really high contrast compared to the background, meaning they can be mistaken for other things, like ringing artefacts.



(a)



(b)



(c)

Figure 9: (a) Unsharp Masking using a β of 15 with swapped colour curves, (b) New version after filtering, (c) Difference.

5.3 Another Example

For completeness sake, Figure 10 shows a different view of the sun, only the object itself is a little bit darker. We can see that, using a λ of 5000, the same kind of results are produced. One thing to note is that around the ringing artefact around the sun, some noise is still present, this is likely because some of the noise in the negative part of the high-pass is connected to the large ringing artefact, giving it a high attribute value.

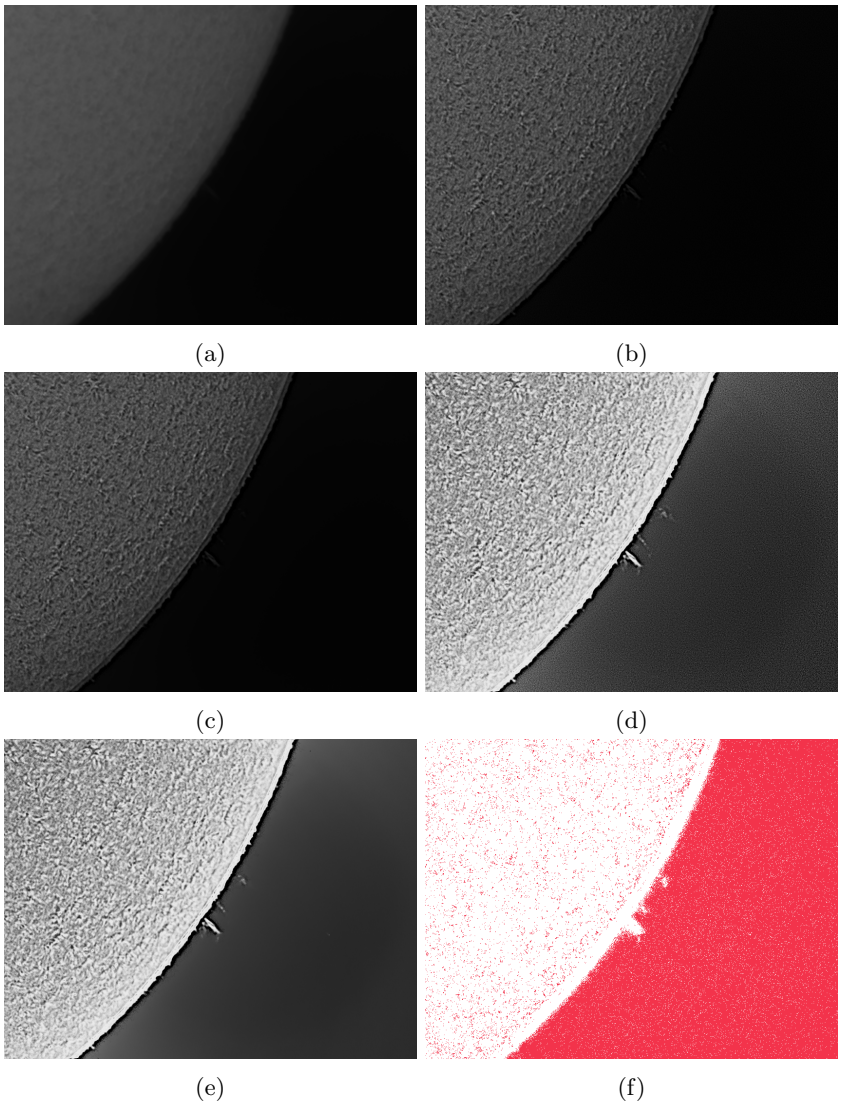


Figure 10: (a) Image of the surface of the sun against a dark background, (b) Unsharp Masking with a β of 25, (c) New version after filtering, (d) Swapped colour curves, (e) New version after filtering (f) Difference between b and c

6 Thesis Conclusions

Connected filters or attribute filters have already shown great promise in the world of image enhancement and the field of astronomy, and I do believe that a case can be made for its use in Unsharp Masking. Using the limited examples I have shown here, we can see that, when using attribute filtering with a sum-

of-squared-greylevels attribute, a significant portion of any noise present in the background can be reduced to a very acceptable level after Unsharp Masking without affecting too much the parts we want to be sharpened. The other big side effect in ringing artefacts can also be helped using k-flat filtering, as they will completely disappear in the filtered images. However, in the examples I used, this only concerns artefacts around small objects, not around the larger one, the sun in this case. To accurately remove these larger ringing artefacts, a more dynamic type of k-flat filtering may be the answer. More examples could have been used to showcase the effects connected filters can have on Unsharp Masking, though it is hard to find good examples to use.

Using C together with the `freeimage.h` library provides a quick and fairly approachable way to work with 16-bit greyscale images. The implementation of my high-pass filter and Unsharp Masking provide an easy way to debug and produce desired images. Some speed-ups are likely achievable since computation time was not my main concern for this project.

To answer the question posed in the Introduction: "Can connected filters be used to improve on Unsharp Masking?" The answer would be yes, connected filters are an incredibly useful and robust tool in the context of image enhancement and the results shown here illustrate that with the right implementation, they can have an considerable impact in the context of Unsharp Masking.

7 Future Work

As mentioned above, the process of differentiating the noise from actual structures is key to producing good results, the methods described here can be improved upon, especially when looking at the type of k-flat filtering used which is probably too rudimentary. A type of variable k-filtering as described in [3] where $k(h) = k_p h^p$ might be a useful tool in detecting the type of photon-noise found in images of astronomy. Another way could be to use a tool to calculate the significance of a structure using a statistical test to see if it can be characterised as noise and then be handled accordingly. A version of this is used in the *mtobjects* project [12].

7.1 Dynamic Lookup Table

When researching proper ways to detect noise using max-Tree filtering, it can be taken one step further. The way the max-tree filtering algorithm works now is that nodes either get removed by changing the greylevel of a child to that of its surviving ancestor, or by leaving it unaltered. After this process is done the image gets restituted from the filtered max-tree. A possible improvement on this is, when in the filtering process, something is detected that can be removed, to not only remove it, but also reduce the value in the lookup table of the actual image. This way noise can be reduced to an even further extent.

7.2 Combination with Other Unsharp Masking Techniques

Obivously, using connected operators and max-tree filtering is not the only way to improve on Unsharp Masking, as there are many adaptive techniques already out there. It may be entirely possible that combining the implementations described in this Thesis with other ones like Nonlinear Unsharp [6] or Cubic Unsharp Masking [5] can improve the final result.

References

[1] Ulisses Braga-Neto and John Goutsias. "A Theoretical Tour of Connectivity in Image Processing and Analysis". In: *Journal of Mathematical Imaging and Vision* 19.1 (July 2003), pp. 5–31. ISSN: 1573-7683. DOI: 10.1023/A:1024476403183. URL: https://doi.org/10.1023/A:1024476403183.

[2] P. Maragos and R.D. Ziff. "Threshold superposition in morphological image analysis systems". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1.5 (1990), pp. 498–504.

[3] G. K. Ouzounis and M. H. F. Wilkinson. "Hyperconnected Attribute Filters Based on k-Flat Zones". In: 33.2 (2011), pp. 224–239.

[4] Georgios K. Ouzounis and Michael H.F. Wilkinson. "Mask-Based Second-Generation Connectivity and Attribute Filters". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29 (2007), pp. 990–1004.

[5] Giovanni Ramponi. "A cubic unsharp masking technique for contrast enhancement". In: *Signal Processing* 67.2 (1998), pp. 211–222. ISSN: 0165-1684. DOI: https://doi.org/10.1016/S0165-1684(98)00038-3. URL: http://www.sciencedirect.com/science/article/pii/S0165168498000383.

[6] Giovanni (Gianni Ramponi et al. "Nonlinear unsharp masking methods for image contrast enhancement". In: *J. Electronic Imaging* 5 (July 1996), pp. 353–366. DOI: 10.1117/12.242618.

[7] P. Salembier, A. Oliveras, and L. Garrido. "Antiextensive connected operators for image and sequence processing". In: *IEEE Transactions on Image Processing* 7.4 (Apr. 1998), pp. 555–570. ISSN: 1057-7149. DOI: 10.1109/83.663500.

[8] P. Salembier and J. Serra. "Flat zones filtering, connected operators, and filters by reconstruction". In: *IEEE Transactions on Image Processing* 4.8 (Aug. 1995), pp. 1153–1160. ISSN: 1057-7149. DOI: 10.1109/83.403422.

[9] P. Salembier and M. H. F. Wilkinson. "Connected Operators: A review of region-based morphological image processing techniques". In: *IEEE signal processing magazine* 26 (2009), pp. 136–157.

[10] Philippe Salembier and Luis Garrido. "Connected Operators Based On Region-Tree Pruning". In: (June 2000). DOI: 10.1007/0-306-47025-X_19.

[11] Jean Serra. "Connectivity on Complete Lattices". In: *Journal of Mathematical Imaging and Vision* 9.3 (Nov. 1998), pp. 231–251. ISSN: 1573-7683. DOI: 10.1023/A:1008324520475. URL: https://doi.org/10.1023/A:1008324520475.

[12] P. Teeninga et al. "Statistical attribute filtering to detect faint extended astronomical sources". In: *Mathematical Morphology - Theory and Applications* 1.1 (Jan. 2016), pp. 100–115. ISSN: 2353-3390. DOI: 10.1515/mathm-2016-0006.

A Code Attachments

A.1 kabsMaxTreeFilter.c

```
/*
    ********************************************************
    */
/*

    */
/* Area/shape preserving extensive and anti-extensive
        */
/* operators using Max-Trees
                                        */
/*

    */
/* by Georgios K Ouzounis - Jan. 2004
                            */
/* original Max-Tree code by Erik Urbach June 2000.
            */
/*

    */
/*
    ********************************************************
    */

#include <assert.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
#include <sys/times.h>
#include <unistd.h>
#include <math.h>

#include <FreeImage.h>


typedef short bool;
#define false 0
#define true  1

typedef unsigned char ubyte;

typedef unsigned short greyval;

#define NUMLEVELS     65536
#define CONNECTIVITY  4

double kbase, background = 0, alpha=0;

typedef struct FIFOQueue
{
   long *Pixels;
   long Head;
   long Tail; /* First free place in queue, or -1 when the queue is full */
} FIFOQueue;



/*** Max-Tree definition ***/

typedef struct MaxNode MaxNode;
typedef struct MaxNode *MaxTree;

/* MaxNode Status */
#define MTS_Ok       0
#define MTS_Deleted  1

struct MaxNode
{
  long Parent;
  long Status;
  long SumGreyLevel;
  int PeakLevel;
```

```
  int kPrime;
  int Level;
  int NewLevel;
};

MaxTree Tree;



/*** Globals used by Max-Tree ***/

long numbernodes[NUMLEVELS]; /* Number of nodes C^k_h at level h */
greyval **P_ORI;  /* Denotes the original gray level value of pixel p in processed image*/
greyval **ORI;    /* Denotes the original ... in the unproceesed input image */

/* STATUS stores the information of the pixel status: the pixel can be
 * NotAnalyzed, InTheQueue or assigned to node k at level h. In this
 * last case STATUS(p)=k. */
#define ST_NotAnalyzed  -1
#define ST_InTheQueue   -2
long *STATUS;

bool NodeAtLevel[NUMLEVELS];

/*** HQueue globals ***/
long  MaxPixelsPerLevel[NUMLEVELS];   /* Original image  */
long PMaxPixelsPerLevel[NUMLEVELS];   /* Processed image */
FIFOQueue HQueue[NUMLEVELS];

/*** Other globals ***/
long ImageWidth;
long ImageHeight;
long ImageSize;
long NumPlanes;

long NumPixelsBelowLevel[NUMLEVELS];
long PNumPixelsBelowLevel[NUMLEVELS];

```

```
#define ReadPixel(t,i,h)  (t[NumPixelsBelowLevel[h]+i].NewLevel)

/*** Queue functions ***/
#define HQueueFirst(h)     (HQueue[h].Pixels[HQueue[h].Head++])
#define HQueueAdd(h,p)     HQueue[h].Pixels[HQueue[h].Tail++] = p
#define HQueueNotEmpty(h)  (HQueue[h].Head != HQueue[h].Tail)
void HQueueCreate(void);
void HQueueDelete(void);

/*********************************************/
/* HQueue is an array segmented based on the   */
/* processed image histogram.                  */
/*********************************************/

void HQueueCreate(void)
{
   int i;

   HQueue->Pixels = calloc((size_t)2*ImageSize, sizeof(long));
   assert(HQueue->Pixels != NULL);
   HQueue->Head = HQueue->Tail = 0;
   for (i=1; i<NUMLEVELS; i++)
   {
      HQueue[i].Pixels = HQueue[i-1].Pixels + PMaxPixelsPerLevel[i-1] + MaxPixelsPerLevel[i-1];
      HQueue[i].Head = HQueue[i].Tail = 0;
   }

} /* HQueueCreate */


/*********************************************/
/* Image I/O functions                         */
/*********************************************/



/** Generic image loader
@param lpszPathName Pointer to the full file name
@param flag Optional load flag constant
```

```
@return Returns the loaded dib if successful, returns NULL otherwise
*/
FIBITMAP* GenericLoader(const char* lpszPathName, int flag) {
  FREE_IMAGE_FORMAT fif = FIF_UNKNOWN;
/*  check the file signature and deduce its format */
/* (the second argument is currently not used by FreeImage) */
  fif = FreeImage_GetFileType(lpszPathName, 0);
  if(fif == FIF_UNKNOWN) {
    /* no signature ? */
    /* try to guess the file format from the file extension */
    fif = FreeImage_GetFIFFromFilename(lpszPathName);
  }
  /* check that the plugin has reading capabilities ... */
  if((fif != FIF_UNKNOWN) && FreeImage_FIFSupportsReading(fif)) {
    /* ok, let's load the file */
    FIBITMAP *dib = FreeImage_Load(fif, lpszPathName, flag);
    /* unless a bad file format, we are done ! */
    return dib;
  }
  return NULL;
}

greyval **ReadTIFF(char *fnm, long *width, long *height, long *numplanes){
   FIBITMAP *dib = GenericLoader(fnm,0);
   unsigned long  bitsperpixel;
   greyval **im;
   unsigned int x,y,i,j,imsize;
   if (dib == NULL) return NULL;

   bitsperpixel =  FreeImage_GetBPP(dib);
   *numplanes=1;
   if ((bitsperpixel==24) ||(bitsperpixel==48))
     *numplanes=3;
   *height = FreeImage_GetHeight(dib),
   *width = FreeImage_GetWidth(dib);

   imsize = (*width)*(*height);
   im = calloc((size_t)*numplanes, sizeof(greyval *));
```

```
  assert(im!=NULL);
  for (i=0;i<*numplanes;i++){
    im[i] = calloc((size_t)imsize, sizeof(greyval));
    assert(im[i]!=NULL);
  }
  switch(bitsperpixel) {
  case 8 :
    i=0;
    for(y = 0; y <*height; y++) {
      BYTE *bits = (BYTE *)FreeImage_GetScanLine(dib, *height - y -1);
      /*printf("y=%d\n",y);*/
      for(x = 0; x < *width; x++,i++) {
    im[0][i] = bits[x];
      }
    }

    FreeImage_Unload(dib);
    return im;
  case 16 :
    i=0;
    for(y = 0; y < *height; y++) {
      unsigned short *bits = (unsigned short *) FreeImage_GetScanLine(dib,*height - y -1);
      for(x = 0; x < *width; x++,i++) {
    im[0][i] = bits[x];
      }
    }
    FreeImage_Unload(dib);
    return im;
  case 24 :
    i=0;
    for(y = 0; y <*height; y++) {
      BYTE *bits = (BYTE *)FreeImage_GetScanLine(dib, *height - y -1);
      /*printf("y=%d\n",y);*/
      for(x = 0; x < *width; x++,i++) {
        for (j=0;j<*numplanes;j++)
      im[j][i] = bits[3*x+j];
      }
    }

    FreeImage_Unload(dib);
    return im;
  case 48 :
    i=0;
```

```
      for(y = 0; y <*height; y++) {
        unsigned short *bits = (unsigned short *) FreeImage_GetScanLine(dib, *height - y -1);
        /*printf("y=%d\n",y); */
        for(x = 0; x < *width; x++,i++) {
          for (j=0;j<*numplanes;j++)
        im[j][i] = bits[3*x+j];
        }
      }

      FreeImage_Unload(dib);
      return im;

    default :
      FreeImage_Unload(dib);
      fprintf(stderr, "unsupported format\n"); exit(-1);
      return NULL;
    }

}


void WriteTIFF(char *fname, long width, long height, int bitspp, long numplanes){
  FIBITMAP *outmap;
  long i,j,y,x,plane;
  FREE_IMAGE_FORMAT fif = FreeImage_GetFIFFromFilename(fname);

  if (bitspp == 8){
    ubyte *imagebuf;
    RGBQUAD *pal;
    outmap = FreeImage_AllocateT(FIT_BITMAP,width,height,numplanes*bitspp,0xFF,0xFF,0xFF);
    if (numplanes==1){
      pal = FreeImage_GetPalette(outmap);
      for (i = 0; i < 256; i++) {
    pal[i].rgbRed = i;
    pal[i].rgbGreen = i;
    pal[i].rgbBlue = i;
      }
    }
    i = 0;
    for (j=height-1; j>=0; j--){
      imagebuf = FreeImage_GetScanLine(outmap,j);
```

```
      for (x=0;x<width;x++,i++)
        for (plane=0;plane<numplanes;plane++)
     imagebuf[numplanes*x+plane]=ORI[plane][i];
    }

  } else {
    greyval *imagebuf;
    if (numplanes==1)
      outmap = FreeImage_AllocateT(FIT_UINT16,width,
          height,16,0xFFFF,0xFFFF,0xFFFF);
    else
      outmap = FreeImage_AllocateT(FIT_RGB16,width,
          height,48,0xFFFF,0xFFFF,0xFFFF);

    i = 0;
    for (j=height-1; j>=0; j--){
      imagebuf = (greyval *)FreeImage_GetScanLine(
          outmap,j);
      for (x=0;x<width;x++,i++)
        for (plane=0;plane<numplanes;plane++)
     imagebuf[numplanes*x+plane]=ORI[plane][i];

    }
  }

  FreeImage_Save(fif,outmap,fname,0);
  FreeImage_Unload(outmap);

}


void ReadPGM(char *fname,short buffer)
{
   long width, height;
   int c;


   if(buffer==1)
     {
       ORI = ReadTIFF( fname, &width, &height, &
           NumPlanes);
       ImageWidth = width;
       ImageHeight = height;
       ImageSize = width*height;

     }
```

```
   else
     {
       P_ORI = ReadTIFF(fname, &width, &height, &NumPlanes);
       ImageWidth = width;
       ImageHeight = height;
       ImageSize = width*height;
     }

} /* ReadPGM */


greyval FindMax(greyval *buf, long width, long height) {
  greyval max = buf[0];
  long i,size = width*height;
  for (i = 1; i<size;i++){
    max = (buf[i]>max) ? buf[i] : max;
  }
  return max;
}


void WritePGM(char *fname, long width, long height, long numplanes)
{


  if (FindMax(ORI[0],width,height)>255)
    WriteTIFF(fname, width, height, 16, numplanes);
  else
    WriteTIFF(fname, width, height, 8, numplanes);


} /* WritePGM */

/*********************************************/
/* Setting up the Max-Tree structure         */
/* MaxPixelsPerLevel already initialized     */
/*********************************************/

MaxTree MaxTreeCreate(long size)
{
   MaxTree t;
   int i;

```

```
   t = calloc((size_t)2*size, sizeof(MaxNode));
   if (t)
   {
      *NumPixelsBelowLevel = 0;
      for (i=1; i<NUMLEVELS; i++)
      {
         NumPixelsBelowLevel[i] = NumPixelsBelowLevel[
            i-1] + MaxPixelsPerLevel[i-1] +
            PMaxPixelsPerLevel[i-1];
      }
   }
   return(t);
} /* MaxTreeCreate */

/*************************************************/
/* Finds a pixel of minimum intensity in the image */
/* MaxPixelsPerLevel : Histogram of original image */
/* PMaxPixelsPerLevel: Histogram of processed image*/
/*************************************************/

long FindHMinPixel(greyval *ORI,greyval *P_ORI)
/*  */
{
   long i, p=0;

   /* initialise level entries */
   for (i=0; i<NUMLEVELS; i++)
     {
       MaxPixelsPerLevel[i] = 0;
       PMaxPixelsPerLevel[i] = 0;
     }
   /* compute histogram */
   for (i=0; i<ImageSize; i++)
   {
      MaxPixelsPerLevel[ORI[i]]++;
      PMaxPixelsPerLevel[P_ORI[i]]++;
      if (P_ORI[i]<P_ORI[p])  p = i;
   }
   return(p);
} /* FindHMinPixel */

/*********************************************/
/* Retrieves the neighbors of pixel p          */
/*********************************************/

int GetNeighbors(long p, long *neighbors)
```

```
{
   long x;
   int n=0;

   x = p % ImageWidth;
   if (x<(ImageWidth-1))  neighbors[n++] = p+1;
   if (p>=ImageWidth)     neighbors[n++] = p-
      ImageWidth;
   if (x>0)               neighbors[n++] = p-1;
   p += ImageWidth;
   if (p<ImageSize)       neighbors[n++] = p;
   return(n);
} /* GetNeighbors */


/*********************************************/
/* Max-Tree Creation: Here it is the modified  */
/* image that is being processed               */
/*********************************************/

int flood(greyval *ORI, greyval *P_ORI, int h
, int *thispeaklevel, long *thisSumGreyLevel)
{
   long neighbors[CONNECTIVITY];
   long p, q,idx;
   int numneighbors, i, m, level, peaklevel = h,
      childpeaklevel;
   long sumGreyLevel = *thisSumGreyLevel,
      childSumGreylevel;
   MaxNode *node;

   while(HQueueNotEmpty(h)){
     p = HQueueFirst(h);
     STATUS[p] = numbernodes[ORI[p]]; /* in all cases
        */

     if(ORI[p]!=h){/* unstable regions */
       level = ORI[p];
       NodeAtLevel[level]=true;

       /* Detect the node */
       node = Tree + (NumPixelsBelowLevel[level] +
          numbernodes[level]);
       peaklevel = (peaklevel<level)? level :
          peaklevel;
       if(level>h){ /* Anti-extensive case */
```

```

    /* set the singleton node at ORI[p]*/
    node->Parent = NumPixelsBelowLevel[h] + numbernodes[h];
    node->Status = MTS_Ok;
    node->Level  = level;
    node->PeakLevel  = level;
    numbernodes[level] += 1;    /* increase this for next time */
    NodeAtLevel[level] = false; /* finalise this node */;
    sumGreyLevel += h*h;           /* Need to update the parent area also  */\

      }
    }else /* Stable or background regions */
      sumGreyLevel += h*h;
    /* process the neighbors */
    numneighbors = GetNeighbors(p, neighbors);
    for (i=0; i<numneighbors; i++){
      q = neighbors[i];
      if (STATUS[q]==ST_NotAnalyzed){
    HQueueAdd(P_ORI[q],q);
    STATUS[q] = ST_InTheQueue;
    NodeAtLevel[P_ORI[q]] = true;
    /* The neighbor is greater than p */
    if (P_ORI[q] > P_ORI[p])
      {
        m = P_ORI[q];
        childpeaklevel = m;
            childSumGreylevel = 0;
        do
          {
        m = flood(ORI, P_ORI, m, &childpeaklevel, &childSumGreylevel);
          } while (m!=h);
            sumGreyLevel += childSumGreylevel;
        peaklevel = (peaklevel<childpeaklevel)? childpeaklevel : peaklevel;
      }
      }
    }
  }/* end of while loop*/

  /* Retrieve the parent and finalise the node*/
  numbernodes[h] = numbernodes[h]+1;
```

```
468    m = h-1;
469    while ((m>=0) && (NodeAtLevel[m]==false))  m--;
470    if (m>=0)
471      {
472        node = Tree + (NumPixelsBelowLevel[h] +
             numbernodes[h]-1);
473        node->Parent = NumPixelsBelowLevel[m] +
             numbernodes[m];
474      }
475    else
476      {
477        idx = NumPixelsBelowLevel[h];
478        node = Tree + idx;
479        node->Parent = idx;
480      }
481    node->Status = MTS_Ok;
482    node->Level = h;
483    node->PeakLevel = h;
484    node->SumGreyLevel += sumGreyLevel;
485    NodeAtLevel[h] = false;
486    *thispeaklevel = node->PeakLevel;
487    *thisSumGreyLevel = node->SumGreyLevel;
488    return(m);
489 } /* flood */
490
491
492 /*********************************************/
493 /* Filtering Functions                       */
494 /*********************************************/
495
496 typedef long (*kFunc)(long, long, long);
497
498 long kflatConstant(long greylevel, long sog, long
      lambda2){
499   return (long) kbase;
500 }
501
502 long kflatLinear(long greylevel, long sog, long
      lambda2){
503
504   return (long) (kbase*(double)greylevel/100.0);
505 }
506
507 long kflatConditional(long greylevel, long sog, long
      lambda) {
508   return (sog > lambda ? kbase :  0);
```

```
}



void MaxTreeProcessNode(MaxTree t, long lambda, long idx, kFunc kf, long maxarea)
/* Deletes node idx and possible one or more of its parents. */
{
  long k,parent, diflevel,newlevel;

   parent = t[idx].Parent;

   diflevel = t[idx].Level - t[parent].Level;
   newlevel = t[parent].NewLevel + diflevel;
   k = (*kf)(diflevel, t[idx].SumGreyLevel, lambda);
   if (t[idx].PeakLevel - t[parent].Level> k){
       /* initiate absorption */
       t[idx].NewLevel = t[parent].NewLevel;
       t[idx].kPrime = -k;
   } else {
     if (t[parent].kPrime<0){
       t[idx].NewLevel = t[parent].NewLevel;
       t[idx].kPrime = 0;
     } else {
       if (diflevel>t[parent].kPrime){
     t[idx].NewLevel = t[parent].NewLevel + t[parent].kPrime;
     t[idx].kPrime = 0;
       } else {
     t[idx].NewLevel = t[parent].NewLevel + diflevel;
     t[idx].kPrime = t[parent].kPrime - diflevel;
       }
     }
   }

} /* MaxTreeProcessNode */

void SetPeakLevels(MaxTree t)
{
  long i, idx,parent;
   int h;

   for ( h= NUMLEVELS-1; h>=0;h-- )
   {
      for (i=0; i<numbernodes[h]; i++)
```

```
      {
     idx = NumPixelsBelowLevel[h] + i;
         parent = t[idx].Parent;
         if (t[parent].PeakLevel < t[idx].PeakLevel)
       t[parent].PeakLevel = t[idx].PeakLevel;
      }

   }

} /* SetPeakLevels */


void MaxTreeFilter(greyval *ORI, MaxTree t, long
    lambda, kFunc k, long maxarea)
{
   long i, idx;
   int h=0;

   SetPeakLevels(t);
   while (numbernodes[h]==0)
     h++;                    /* find root*/


   /*process root*/


   idx = NumPixelsBelowLevel[h];

   t[idx].kPrime = (*k)(t[idx].Level, t[idx].
       SumGreyLevel, lambda);
   t[idx].NewLevel = t[idx].Level;

   h++;

   while ( h< NUMLEVELS )
   {
      for (i=0; i<numbernodes[h]; i++)
      {
     idx = NumPixelsBelowLevel[h] + i;
         MaxTreeProcessNode(t, lambda, idx, k,maxarea)
             ;
      }
      h++;
   }
   /* Modify the output image */
   for (i=0; i<ImageSize; i++)
```

```
     ORI[i] = ReadPixel(t, STATUS[i], ORI[i]);

} /* MaxTreeFilter */

/*
    ------------------
```

```
    int l,i;
    long idx,area,level,parent,status, parent_level, parent_node;

    fp = fopen("MaxTreeReport.txt","a");
    fprintf(fp,"\n--- Start of Session ---\n");
    fprintf(fp,"Filenames: '%s' - '%s'\n",fname1, fname2);
    fprintf(fp,"Image Size %ld\n",ImageSize);
    fprintf(fp,"h_min:%d\n",hmin);

    for (l=NUMLEVELS-1; l>=0; l--)
    {
      for (i=0; i<numbernodes[l]; i++)
      {
          idx = NumPixelsBelowLevel[l] + i;
          level  = t[idx].Level;
          parent = t[idx].Parent;
          /* parent level*/
          parent_level = t[parent].Level;
          parent_node =  parent - NumPixelsBelowLevel[parent_level];
          status = t[idx].Status;
          /*if(status==0)*/
            fprintf(fp,"level: %d, node: %d, area: %ld, parent_level: %ld, parent_node: %ld, status: %ld\n",l,i,area,parent_level, parent_node,status);
      }
    }
    fprintf(fp,"--- End of Session ---\n");
    fclose(fp);
}

int main(int argc, char *argv[])
{
  long i, p,plane;
   long area=0,k=0, maxarea;
   int peaklevel;
   long sumGreyLevel;
   long lambda;;
   int hmin;
   short buffer;
  struct tms timestruct;
  clock_t start;
  long ticksize=sysconf(_SC_CLK_TCK);
```

```
  float museconds;

   if (argc<5)
   {
      printf("Usage: %s <original inputimage> <processed input image> <outputfile> <lambda> [k] \n", argv[0]);
      exit(0);
   }
   lambda = atol(argv[4]);
   FreeImage_Initialise(0);

   ReadPGM(argv[1],buffer=1); /* ORI image buffer */
   ReadPGM(argv[2],buffer=2); /* P_ORI  image buffer */
   /*printf("\n   Read Buffers Ready\n");*/
   if (argc>5) k=atoi(argv[5]);
   /* STATUS keeps track of each pixel's labelling */
   /*printf("\n   Levels initialised\n");   */

   kbase = k;

   /*printf("%f\n",kbase);*/
  times(&timestruct);
  start=timestruct.tms_utime;

   for (plane=0; plane<NumPlanes; plane++){

     STATUS = calloc((size_t)ImageSize, sizeof(long));
     assert(STATUS!=NULL);
     for (i=0; i<ImageSize; i++)  STATUS[i] = ST_NotAnalyzed;
     InitLevels();
     p = FindHMinPixel(ORI[plane],P_ORI[plane]);
     hmin = P_ORI[plane][p];
     peaklevel = hmin;
     sumGreyLevel = 0;
     NodeAtLevel[hmin] = true;
     HQueueCreate();
     /*   printf("\n   Hierarchical Queues Ready\n"); */

     HQueueAdd(hmin, p);
     STATUS[p] = ST_InTheQueue;
     Tree = MaxTreeCreate(ImageSize);
     assert(Tree!=NULL);
```

```
      /*printf("\n   Max-Tree structure Ready\n");*/

      flood(ORI[plane],P_ORI[plane],hmin, &peaklevel, &sumGreyLevel);
      /*printf("\n   MaxTree Structure Completed!\n");*/

      MaxTreeFilter(ORI[plane],Tree, lambda, kflatConditional, maxarea);
      /*   printf("\n   Area Opening Completed!\n");*/

      /* Generate a log file */
      /* CreateLog(Tree,hmin,argv[1],argv[2]); */

      free(Tree);
      free(HQueue->Pixels);
      free(STATUS);
    }
  times(&timestruct);
  museconds=1E6*(timestruct.tms_utime-start)/(float) (ticksize);
  printf ("%8ld %10.4f\n", lambda, museconds/1000000);
  WritePGM(argv[3], ImageWidth, ImageHeight, NumPlanes);



    for(plane=0; plane<NumPlanes; plane++){
      free(ORI[plane]);
      free(P_ORI[plane]);
    }
    free(ORI);
    free(P_ORI);
    FreeImage_DeInitialise();
    return(0);

} /* main */
```

A.2 highPass.c

```
#include <assert.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <sys/times.h>
#include <unistd.h>
#include <FreeImage.h>
#include <math.h>

long ImageWidth;
long ImageHeight;
long eWidth;
long eHeight;
long ImageSize;
long NumPlanes;
int ksize;

typedef unsigned long greyval;
typedef unsigned short greyvalold;
typedef unsigned char ubyte;

greyvalold N;

#define NUMLEVELS     65536

greyval **ORI;    /* Denotes the original ... in the
    unproceesed input image */
greyval **new;    /* Denotes the output image */

/* Getting standard deviation */

double getStdDev () {;
  return 0.3*((ksize-1)*0.5 - 1) + 0.8;
}

/* Sum of array */

double sumArray(double *arr, int size) {
  int i;
  double sum = 0;
  for (i = 0; i < size; i++){
    sum+= arr[i];
  }
  return sum;
}

/** Generating kernel used for gaussian blurring
 * @param sigma, the standard deviation
 * @returns the kernel
 */
```

```


double *generateKernel(double sigma) {
  int k = ksize;
  int kh = ksize/2;
  double *kernel = malloc(k*k*sizeof(double));


  int i = 0, j = 0;
  double alpha = (double) -(((-kh)*(-kh)) + ((-kh)*(-kh)));
  double div = exp(alpha/(2*sigma*sigma));
  for(i = 0; i < k; i++) {
    for (j = 0; j < k; j++) {
      alpha = (double) -(((i-kh)*(i-kh)) + ((j-kh)*(j-kh)));
      kernel[i*k+j] = exp(alpha/(2*sigma*sigma))/div;
    }
  }
  /*for(i = 0; i < k; i++) {
    for(j = 0; j < k; j++) {
      printf("%lf ", kernel[i*k + j]);
    }
    printf("\n");
  }*/

  return kernel;
}

/* Extending image edges for easier gaussian blurring */


greyval **extendImage(greyval **img) {
  greyval **eImage = calloc((size_t)NumPlanes, sizeof(greyval *));
  assert(eImage != NULL);
  long imsize = (eWidth) * (eHeight);
  int i,j, plane;
  for (i=0;i<NumPlanes;i++){
    eImage[i] = calloc((size_t)imsize, sizeof(greyval));
    assert(eImage[i] != NULL);
  }

  for (plane = 0; plane < NumPlanes; plane++) {
```

```

    //Horizontal Edges

    for(i = ksize/2; i < eWidth - ksize/2; i++) {
      for(j = 0 ; j < ksize/2;j++) {
        eImage[plane][i + j*eWidth] = img[plane][i-ksize/2];
      }
    }

    for(i = ksize/2; i < eWidth - ksize/2; i++) {
      for(j = 0 ; j < ksize/2;j++) {
        eImage[plane][i + (j+eHeight-ksize/2)*eWidth] = img[plane][i+(ImageHeight-1)*ImageWidth-ksize/2];
      }
    }


    //Vertical edges

    for(i = ksize/2; i < eHeight - ksize/2; i++) {
      for(j = 0 ; j < ksize/2;j++) {
        eImage[plane][i*eWidth + j] = img[plane][ImageWidth*(i-ksize/2)];
      }
    }

    for(i = ksize/2; i < eHeight - ksize/2; i++) {
      for(j = eWidth -ksize/2 ; j < eWidth;j++) {
        eImage[plane][i*eWidth + j] = img[plane][ImageWidth*(i-ksize/2) +ImageWidth -1];
      }
    }

    //Rest of image;

    for(i = ksize/2; i < eHeight- ksize/2; i++) {
      for(j = ksize/2; j < eWidth - ksize/2; j++) {
        eImage[plane][i * eWidth + j] = img[plane][(i-ksize/2)*ImageWidth + j-ksize/2];
      }
    }
  }
  return eImage;
}
```

```

/* Gaussian blurring of image */

void gaussianBlur() {
  int i,j,k, l, plane;
  int kh = ksize/2;
  greyval **eImage = extendImage(ORI);
  long imsize = ImageWidth*ImageHeight;
  new = calloc((size_t)NumPlanes, sizeof(greyval *));

  assert(new!=NULL);
  for (i=0;i<NumPlanes;i++){
    new[i] = calloc((size_t)imsize, sizeof(greyval));
    assert(new[i]!=NULL);
  }
  double *kernel = generateKernel(getStdDev());
  double sum = sumArray(kernel,ksize*ksize);
  //printf("sum:%lf\n", sum);

  /* Blurring is happening here */


  for (plane = 0; plane < NumPlanes; plane++) {


    // Horizontal
    for(i = 0; i < ImageHeight; i++) {
      for(j = 0; j < ImageWidth; j++) {
        for(k = 0; k < ksize; k++) {

          new[plane][i*ImageWidth + j] += kernel[k] * eImage[plane][(i+kh)*eWidth + k+j];

        }
      }
    }
    eImage = extendImage(new); //extending resulting image again
    // Vertical
     for(i = 0; i < ImageHeight; i++) {
      for(j = 0; j < ImageWidth; j++) {
        new[plane][i*ImageWidth + j] = 0;
        for(k = 0; k < ksize; k++) {

          new[plane][i*ImageWidth + j] += kernel[k] * eImage[plane][i*eWidth + k*eWidth +kh+j];
```

```

        }
        new[plane][i*ImageWidth+j]/=sum; //
            Normalizing
      }
    }
  }

    for(plane=0; plane<NumPlanes; plane++){
     free(eImage[plane]);
   }
  free(eImage);
  free(kernel);
  }

/* Original image minus low pass filter */


void highPass(int beta) {
  greyvalold **pos, **neg;
  long i, plane;
  long imsize = ImageWidth*ImageHeight;

  //Positive values
  pos = calloc((size_t)NumPlanes, sizeof(greyvalold *)
      );
  assert(pos!=NULL);
  for (i=0;i<NumPlanes;i++){
    pos[i] = calloc((size_t)imsize, sizeof(greyvalold)
        );
    assert(pos[i]!=NULL);
  }

  //Negative values

  neg = calloc((size_t)NumPlanes, sizeof(greyvalold *)
      );
  assert(neg!=NULL);
  for (i=0;i<NumPlanes;i++){
    neg[i] = calloc((size_t)imsize, sizeof(greyvalold)
        );
    assert(neg[i]!=NULL);
  }

  //High-pass
  for (plane = 0; plane < NumPlanes; plane++) {
```

```
    for(i = 0; i <ImageWidth*ImageHeight; i++) {
      int pixel =  (ORI[plane][i] - new[plane][i]);
      if(pixel < 0) {
        neg[plane][i] = (pixel*beta*-1 > N) ? N :(greyvalold) pixel*-1*beta;
      } else {
        pos[plane][i] = (pixel*beta > N) ? N : (greyvalold) pixel*beta;
      }
    }
  }

  //Writing images

  if ( N> 255) {
    WriteTIFF("pos.tif", ImageWidth, ImageHeight, 16, NumPlanes, pos);
    WriteTIFF("neg.tif", ImageWidth, ImageHeight, 16, NumPlanes, neg);
  } else {
    WriteTIFF("pos.tif", ImageWidth, ImageHeight, 8, NumPlanes, pos);
    WriteTIFF("neg.tif", ImageWidth, ImageHeight, 8, NumPlanes, neg);

  }


  for(plane=0; plane<NumPlanes; plane++){
    free(pos[plane]);
    free(neg[plane]);
  }

  free(pos);
  free(neg);

}

int main(int argc, char *argv[]) {

  FreeImage_Initialise(0);
  int i,plane, beta =1;

  if (argc < 3)  {
    printf("Usage: %s <inputfile> <ksize> [beta]\n", argv[0]);
```

```
    exit(0);
  }
  ksize = atoi(argv[2]);
  if(ksize % 2 == 0) {
    printf("ksize must be odd\n");
    exit(0);
  }

  if (argc  == 4) {
    beta = atoi(argv[3]);
  }

  long width, height;
  ORI = ReadTIFF(argv[1], &width, &height, &NumPlanes);
  N = (findMax(ORI[0], width, height) > 255 ? NUMLEVELS-1 : 255);
  ImageWidth = width;
  ImageHeight = height;
  eWidth = width + ksize -1;
  eHeight = height + ksize -1;
  gaussianBlur();
  highPass(beta);

 // printf("ImageHeight: %ld, ImageWidth: %ld\n", ImageHeight, ImageWidth);

  for(plane=0; plane<NumPlanes; plane++){
     free(ORI[plane]);
     free(new[plane]);
   }

  free(ORI);
  free(new);

  FreeImage_DeInitialise();
}
```

A.3 unsharpMasking.c

```


#include <assert.h>
#include <stddef.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <unistd.h>
#include <FreeImage.h>
#include <math.h>

long ImageWidth;
long ImageHeight;

long ImageSize;
long NumPlanes;

typedef unsigned short greyval;
typedef unsigned char ubyte;

#define NUMLEVELS     65536

greyval N;

greyval **ORI;    /* Denotes the original ... in the unproceesed input image */
greyval **neg;    /* Denotes the processed input image */
greyval **pos;


/* Unsharp Masking*/

void usm (int beta) {
  int i, plane;
  for (plane = 0; plane < NumPlanes; plane++) {
    for(i = 0; i < ImageSize; i++) {
      int pixel = (int) ((int)(ORI[0][i] + pos[0][i]*beta - neg[0][i]*beta));
      if(pixel > N) {
        ORI[plane][i] = N;
      } else if(pixel < 0) {
        ORI[plane][i] = 0;
      } else {
        ORI[plane][i] = (greyval) pixel;
      }
    }
  }
}

```

```

int main(int argc, char *argv[]) {

  FreeImage_Initialise(0);
  int beta = 1;
  if (argc < 5) {
    printf ("Usage: %s original positive negative beta \n", argv[0]);
  }

  ORI =ReadTIFF(argv[1], &ImageWidth, &ImageHeight, &NumPlanes);
  pos = ReadTIFF(argv[2], &ImageWidth, &ImageHeight, &NumPlanes);
  neg = ReadTIFF(argv[3], &ImageWidth, &ImageHeight, &NumPlanes);
  N = (findMax(ORI[0], ImageWidth, ImageHeight) > 255 ? NUMLEVELS-1 : 255);


  ImageSize = ImageWidth * ImageHeight;

  if(argc == 5) {
    beta = atoi(argv[4]);
  }

  usm(beta);

  FreeImage_DeInitialise();
  if (N > 255) {
    WriteTIFF("usm.tif", ImageWidth, ImageHeight, 16, NumPlanes, ORI);
  } else {
    WriteTIFF("usm.tif", ImageWidth, ImageHeight, 8, NumPlanes, ORI);
  }

  for(plane=0; plane<NumPlanes; plane++){
     free(ORI[plane]);
     free(pos[plane]);
     free(neg[plane]);
   }

  free(ORI);
  free(pos);
  free(neg);
```

```

}
```