University of Portsmouth

School of Energy and Electronic Engineering

Simultaneous Visual Identification and Tracking of household objects

Iason, Tzanetatos

926418

MSc Electronic Systems Engineering

Dissertation

Department of Electrical, Mechanical and Electronic Engineering

Faculty of Engineering and Architecture

Marousi Campus, Athens, Greece

2019/2020

Statement of Originality:

This dissertation is submitted in partial fulfilment of the requirements for the Engineering Programmes *Master of Science* at the *University of Portsmouth*.

I, the author and undersigned, declare that this dissertation is my own original work. If ideas and/or wording, resources and so on, from another source were used, this is explicitly referenced within the text.

I give permission for this dissertation to be photocopied and made available for interlibrary loans for the purpose of research.

I give permission for this dissertation, and the electronic source, to be used for academic and research purposes, as considered necessary to fulfil the requirements of the University Regulations, Procedures and Codes of Practice.

Signed:

Surname, Name: Tzanetatos Iason

Date: 23/09/2019

Statement of Supervision

In my opinion the dissertation is sufficiently well prepared to be presented to the Board of Examiners, and no immediate issues have been identified. In particular, it is certified that:

- The dissertation does not exceed the prescribed maximum word limit; Yes ☒ No ☐
- The research and writing embodied are those of the student except where due reference is made in the text (similarity rate less than 20%); Yes ☒ No ☐
- Any assistance provided to the student during the research phase has been appropriately described and acknowledged; Yes ☒ No ☐
- Any assistance provided to the student in the implementation stages has been appropriately described and acknowledged. Yes ☒ No ☐

Supervisor (Name Surname): Dr G. Chliveros

Supervisor Signature: A. N. Other

Date: 23/12/2019

Acknowledgements

I would like to express my deepest gratitude to my supervisor Dr. Georgios Chliveros for his support for this dissertation. Whenever I had some questions or ideas regarding the implementation of the research or writing, he was always available. He allowed me to explore my ideas with an indescribable amount of freedom, interfering only when he deemed it necessary. Thank you.

I would like to thank my parents for providing me the opportunity to further continue with my academic course. Without their support, I would not have been able to attend to this MSc programme.

Finally, I would like to express by gratitude to my friends and partner for supporting me in their own way, during the development of my research.

Abstract

The field of computer vision in the recent years has been rapidly expanding. Algorithms and systems that are capable performing impressive results, that in some cases outperform the human visual perception, have been already implemented and are constantly improved. However, most recent implementations utilize Artificial Neural Networks, that are abstracts in nature, thus the implemented models are difficult to interrogate on the criteria that led to their results.

The main objective of this project is the implementation of a human interpretable system, that is capable of identifying and detected any number of objects of interest, reliably. The implementation of the system is unsupervised in nature, meaning that no assumptions are made over the properties of the object that are of interest to be detected. By further adding features to the project, it can be implemented in industrial applications, driver assistive applications, etc.

The system's performance is only limited to the illumination of the scene, and it can occasionally fail to detect objects that have the same or similar colour of the background's scene.

Keywords: Computer Vision, OpenCV, Machine Learning, Data Analysis, Signal Processing, Unsupervised Learning, Sequential Covering Algorithms

Table of Contents

List of Figures

List of Tables

List of Abbreviations

KS test	Kolmogorov-Smirnov Statistical Test
BGR	Blue Green Red Colour Coding
MoG2	Mixture of Gaussian Kernels Algorithm
Std	Standard Deviation
CDF	Cumulative Density Function
GMM	Gaussian Mixture Model
IIHS	Insurance Institute for Highway Safety
ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks

PAGE INTENTIONALLY LEFT BLANK

Chapter 1: Introduction – Aims and Objectives

1.1 Background

The field of computer vision has been rapidly developing in the past decades. The field of applications is limitless, from surveillance, safety systems, traffic monitoring systems, automated driver assistive systems, or even human identification. Coupled with the field of machine learning that is currently experiencing a rapid development, with new systems being developed every year, outperforming the previous systems, and in some cases even outperforming the human visual perception. Currently, the most dominant methodology of implementing systems that achieve state-of-the-art results, is the utilization of Artificial Neural Networks (ANN). While ANN are capable of producing high accuracy results, there is one major drawback that starts to emerge, in particular in the applications of autonomous cars; due to their abstractive nature, the inner workings of ANN are almost human uninterpretable, thus the models cannot be interrogated on their results.

In the case of several implementations of autonomous cars that are already available in the commercial market, certain unwanted behaviours of the systems have emerged, and it is, as of now, still unclear how to proceed in order to correct said behaviours. On-road and track test performed by the IIHS institute found several major issues ranging from minor to extremely severe. Some notable examples of severe issues are as such. In some cases where already stopped cars are in the front of the vehicle, the adaptive cruise control failed to attempt to stop. In other test scenarios, the cars briefly decelerated due to false positive obstacles. The IIHS also found that some models failed to apply the autobrake function of the system, resulting in a collision. Finally, in another test, some cars failed to stay inside the lane due to the curvature of the road [9].

While these issues and implementations are beyond the scope of this dissertation, they serve as an indication that the abstractive nature of ANN can create unforeseeable issues. This also highlights the importance of a model to be interrogative and human interpretable.

1.2 Aims and Objectives

Based on the content that has been presented in the previous subchapter, the aims and objectives of this dissertation can be clearly defined. The overall aim of the system is the implementation of an unsupervised algorithm, via human interpretable rule-based means, that is capable of simultaneously identify and track any number of objects in a non-static background. As previously highlighted, large emphasis is given on

the fact that the system must be human interpretable and interrogative, in order to be able to deduce and understand as clearly as possible the system's responses.

In order to achieve the aim of the project, the following objectives must be implemented, before the above-mentioned aim. A reliable background subtraction by a mixture of Gaussian functions will be initially implemented in order to detect the foreground of interest. Statistical tests and metrics, indicating whether the background subtraction has successfully detected the objects of interest, or the objects have yet to be detected will complement the background subtractor. The results of these statistical tests and operations are critical since they are responsible for deducing the results of the background subtraction; therefore, it is also crucial to be as reliable as possible. The final objective to the project is the implementation of a segmentation algorithm, in order to clearly separate multiple objects that the system considers them to be one object.

1.3 Dissertation Outline

The dissertation's structure is segmented into six chapters. In Chapter 1 the main subject that the dissertation attempts to provide a solution is presented and briefly discussed. The aims and objectives are also presented in a clearly defined manner. In Chapter 2 the current state-of-the-art implementations that tackle the same subject, along with previously related works that this dissertation is based are discussed and the differences between the already made solutions and this dissertation's proposed methodology are highlighted. Chapter 3 provides a detailed analysis over the core methodology, the relevant works that this dissertation is based, and how said works are utilized. In Chapter 4, some initial experiments are performed, parts that were considered for this implementation are presented and discussed, and their results along with the results of the experiments are presented. Chapter 5 presents extensive experiments on various conditions; the results are presented, discussed and evaluated in detail. Chapter 6 presents the conclusions of the results of the implementation, its limitations and potential future work that could further develop the proposed system.

Chapter 2: Literature Review

2.1 State-of-the-art Implementations

The fields of computer vision and machine learning have been extensively explored by a variety of different approaches and implementations. In recent years, most of the state-of-the-art implementations have been performed by means of neural networks. The main architecture that is commonly used is the convolution neural networks, utilized to extract the desired features and classify the detected objects. The current state-of-the-art implementations that used the convolution architecture have either utilized improved versions of the network's architecture, such as R-CNN, or incorporated the convolution architecture as a part of the overall proposed network's architecture.

A notable example of such implementations is the YOLO project, that is capable of detecting up to 9000 classes of objects in real-time, utilizing a single convolutional network in order to estimate the position and the class of the detected objects. Since its initial implementation, the project has been further developed in order to increase its performance at the cost of its processing speed [18, 19]. Despite its reduction in speed, it still retains state-of-the-art status, and is capable of processing frames at 30 frames per second [19].

As it can be noted, there are many parameters that can be exploited in order to achieve state-of-the-art results. Such parameters are the network's dimensions that can be changed accordingly in order to improve the performance. The EfficientNet implementation utilizes a compound coefficient to uniformly scale the network's dimensions; by utilizing neural architecture search, the study designed a model, scaled its dimensions accordingly and concluded to the design of a family of models denoted as EfficientNets [23].

Another noteworthy implementation of manipulating the network's design is the RMDL implementation that finds the most appropriate architecture of the network in order to achieve the most efficient processing speed as possible. The implementation is not limited to only image processing; it can process a variety of input data [11]. An implementation of architecture manipulation is the construction of two identical CNN

where the outputs of both networks are compared, and the appropriate detection is performed. The implementation achieved state-of-the-art results on detecting a specific object in a cluttered environment. [28]

The MixMatch study proposed a semi-supervised learning network, classifying each object by “guessing low-entropy labels for data-augmented unlabelled examples” [2].

There are other implementations that aim to improve and innovate on pore core ideas, instead of focusing on implementing new features. One such notable example is the implementation of an unsupervised implementation that is capable of performing classification and segmentation. The system has, as the authors note, achieved global status for state-of-the-art and it is capable of processing other forms of data besides images. [10]

2.2 Relevant implementations

Improved Adaptive Gaussian Mixture Model (MoG2)

The idea of modelling a pixel’s value as a mixture of Gaussian kernels, has been already investigated by [4] and [6]. While the main idea of retaining a number of previously present mixture of Gaussian kernels that determine the background model, several issues were not able to be addressed by the proposed implementations

The improvements that are presented in [27] further improve the performance and processing speed both in static and dynamic frames. By recursively updating the model and implementing a constantly adaptive history of background model, the proposed implementation is capable of producing reliable and optimal results. It is important to note that, over time, present objects will also be subtracted from the scene, since the algorithm implements an exponential decaying effect, to suppress the influence of previous samples. The number of Gaussian kernels is also estimated recursively. The algorithm is both suited for static and adaptive background scenes. A noteworthy feature of the implementation is its improvement on the detection of shadows. The authors also

further developed another Gaussian kernel estimator, that is better suited for dynamic background scenes [27]

2.3 Summary Review

A number of state-of-the-art implementations have been briefly presented. While this project does not aim to implement or study ANN, it serves as an important indication on the capabilities found in the computer vision field. It also highlights the importance, as previously mentioned, on human interpretability

Chapter 3: Methodology and Implementation

3.1 Methodology

The proposed implementation utilizes a single camera sensor that is responsible for "feeding" frames into the algorithm, in order to detect potential objects of interest that are currently present in the foreground of the frame. The proposed system assumes that the camera's position is non static, therefore the background will always be composed of different scenes or colours. This feature implies the need of an adaptive foreground detection (or background subtraction) algorithm; other image processing techniques such as morphological operations might be of interest in order to extract the silhouettes of the present objects. Finally, assuming that the contours of the detected objects have been captured, in the case where different objects appear to be connected, an algorithmic approach must be implemented in order to distinguish between the seemingly connected objects.

The proposed algorithm will be implemented using the Python programming language, along with the libraries NumPy, OpenCV and Scikit-. All utilized software is open source and free to use for academic usage, as per their respective licences' definition. The library NumPy was utilized for data structure; OpenCV for its background subtraction functions, image processing functions and data acquisition; Scikit-Learn for its segmentation algorithms that was utilized in Chapter 4. Besides these libraries, the presented is work of this dissertation's author making.

In terms of hardware, a USB connected camera will be utilized in order to acquire frames to test the results of the developed algorithms, as well as test the results of the final implementation, via real-time frame streaming.

A detailed analysis of the most fundamental parts of the proposed implementation are discussed below, and an overview of the methodology is presented and discussed, showcasing how the most essential parts of the implementation tie up together, in order to produce the desired results and accomplish the dissertation's main objective.

3.2 Hypothesis Generation (MoG2)

In order to detect and capture the foreground of interest that might contain an N number of objects that it is desired to identify and track, an appropriate solution that adapts its response, based on the complexity of the background, is presented that [] have implemented. As previously mentioned, the reasoning behind considering an adaptive solution is due to the fact that no assumptions are made over the statistics of the background present in the stream of input frames. In other words, the complexity and the colours that consist the background are unknown to the system and the same is true for the colour, shape and complexion of the object or objects of interest.

By utilizing the Mixture of Gaussians approach that [28] has developed, the segmentation of background/foreground can be achieved over an interval of some adaptation time. The decision whether the value of a pixel, denoted as x, at time t, belongs to the background is defined as [28]:

$$p\left(\vec{x}^{(t)}\middle|BG\right) > C_{thr} \qquad (1)$$

where C_{thr} is defined as a threshold value and BG the background. To estimate the background classifier, a finite number of previous frames (i.e. $C_{Fame} - n$, where C_{Frame} is the current frame and n the number of previous frames), denoted as X, are required.

The collection of previous frames, X, has the form of $X_t = \{x^t, \dots, x^{t-T}\}$, where T the overall time period of the frame set and t some time frame. The frame set X_T is constantly updated with the addition of the current frame, and the oldest frame, along with its coefficients and statistical properties is discarded. The estimated background classifier is defined as:

$$\hat{p}(\vec{x}^{(t)}|X_T, BG) \qquad (2)$$

and it is constantly re estimated for each new frame that is added into the frame set. It is expected that some pixel values might belong to the foreground cluster, therefore the estimated background classifier is denoted as:

$$\hat{p}(\vec{x}^{(t)}|X_T, BG + FG) \qquad (3)$$

As [28] notes, the only assumption that is made between the frames contained in the data set is that they are independent. By utilizing the Gaussian Mixture Model (GMM) with M number of clusters, the estimated background classifier equation (3) is [27]:

$$\hat{p}(\vec{x}|X_T, BG + FG) = \sum_{m=1}^{M} \hat{\pi}_m N(\vec{x}; \hat{\vec{\mu}}_m, \hat{\sigma}_m^2 I) \qquad (4)$$

$\hat{\vec{\mu}}_1, \dots, \hat{\vec{\mu}}_m$ are the estimated mean values and $\hat{\sigma}_1^2, \dots, \hat{\sigma}_m^2$ are the estimated variances. The coefficients $\hat{\pi}_m$ have positive values and their total sum is equal to 1. Assuming that a new frame has been entered into the frame set, the estimated coefficients $\hat{\pi}_m$, mean values $\hat{\vec{\mu}}_m$ and variances $\hat{\sigma}_m^2$ are updated recursively as such in the implementation of [29]:

$$\hat{\pi}_m \leftarrow \hat{\pi}_m + a(o_m^{(t)} - \hat{\pi}_m) \qquad (5)$$

$$\hat{\vec{\mu}}_m \leftarrow \hat{\vec{\mu}}_m + o_m^{(t)}(a/\hat{\pi}_m)\vec{\delta}_m \qquad (6)$$

$$\hat{\sigma}_m^2 \leftarrow \hat{\sigma}_m^2 + o_m^{(t)}(\tfrac{a}{\hat{\pi}_m})(\vec{\delta}_m^T\vec{\delta}_m - \hat{\sigma}_m^2) \qquad (7)$$

The [27] define the term $\vec{\delta}_m$ as $\vec{\delta}_m = \vec{x}^{(t)} - \hat{\vec{\mu}}_m$, constant term $\alpha \cong {}^1/_T$ that as [27] note "describes an exponential decaying envelope that is used to limit the influence of the old data", in this implementation data refers to the pixel values that the newly entered frame contains. Finally, the variable term $o_m^{(t)}$ defines in which cluster the new frame's pixel values belongs to, setting said variable to 1 for the cluster with the largest coefficient $\hat{\pi}_m$, and 0 for all remaining clusters. To decide in which cluster the pixel values belong, the

metric of the squared Mahalanobis distance is evaluated and compared with some threshold value; the distance equation is defined as [27]:

$$D_m^2\left(\vec{x}^{(t)}\right) = \vec{\delta}_m^T \vec{\delta}_m / \hat{\sigma}_m^2 \qquad (8)$$

As the [27] note, if no clusters fulfill the results of the evaluated distance, a new cluster is created by executing the following equations that [28] define:

$$\hat{\pi}_{M+1} = a \qquad (9)$$

$$\hat{\vec{\mu}}_{M+1} = \vec{x}^{(t)} \qquad (10)$$

$$\hat{\sigma}_{M+1} = \sigma_0 \qquad (11)$$

where σ_0 "some initial appropriate value" [27]. In the case when the maximum number of clusters have been reached, [27] removes the clusters that has the smallest $\hat{\pi}_M$.

Considering the above discussed equations, the approximation of the Background classifier, as defined by [27], "by the first B largest clusters" is:

$$p(\vec{x}|X_T, BG) \sim \sum_{m=1}^{B} \hat{\pi}_m N(\vec{x}; \hat{\vec{\mu}}_m, \hat{\sigma}_m^2 I) \qquad (12)$$

By arranging the clusters to, as [] define, "descending weights $\hat{\pi}_m$" (i.e. coefficients), the definition of the background classifier for the "first B largest clusters" is:

$$B = \underset{b}{\operatorname{argmin}}\left(\sum_{m=1}^{b} \hat{\pi}_m > (1 - C_f)\right) \qquad (13)$$

The [28] defines C_f as an indicator of "the maximum portion of the data that can belong to the foreground model without influencing the background model" [28].

However, one potential issue arises, based on the above mentioned; assuming that a new, previously unseen, object enters into the frame, the coefficient of its cluster will be constantly increasing until its value becomes greater than or equal to the variable C_f. When that occurs, the cluster is considered as background and the object is gradually removed from the frame, in an exponential manner as previously discussed.

One solution to this issue is to simply assume that the MoG2 algorithm has converged when a number of frames, equal to the predefined period T, have passed, and evaluate the final resulting frame. This approach however is hypothesised to work only on a static background, and it does not guarantee that enough information about the object

of interest will be present. A more appropriate solution to this issue, as [28] denote, is to estimate the number of frames the object would remain present in the foreground by evaluating

$$\log(1 - C_f)/\log(1 - a) \qquad (14)$$

While equation (14) is hypothesized to produce the desired response of the system, further information is needed in order to make the final decision on whether the MoG2 algorithm has converged appropriately and/or enough information of the object of interest is present. One solution for this issue would be to further process the captured contour to confirm whether the contour corresponds with the physical object. However, as previously stated, no assumptions are made over the shape and colour of the objects of interest, therefore this classification cannot be determined without some knowledge of the shape and/or colour of certain objects. This approach could limit the number of detected objects to a predefined number of classes, and in some cases fail to detect objects due to the object's distance from the camera, or some other factor. The methodology that was utilized in this dissertation is discussed in the following subchapters.

The coefficients $\hat{\pi}_m$ also denote the amount of pixel values that belongs to some cluster of the GMM. Therefore, coefficients $\hat{\pi}_m$ can also be considered as an indication of the probability that some pixel value belongs to some cluster of the GMM. As [] note $\hat{\pi}_m$ "define an underlying multinomial distribution". The definition of the number of pixels that belong to some cluster of the GMM, m, is [27]:

$$n_m = \sum_{i=1}^{t} o_m^{(i)} \qquad (15)$$

where $o_m^{(i)}$, as previously defined, indicates the ownership of the pixel values to the GMM clusters and index t denoted the amount of pixel values. Therefore, the relation between coefficients $\hat{\pi}_m$ and the identifier of the number of pixels, n_m, gives the estimate from the amount of pixels t [28]:

$$\hat{\pi}_m^{(t)} = \frac{n_m}{t} = \frac{1}{t}\sum_{i=1}^{t} o_m^{(i)} \qquad (16)$$

Equation (16) can be re written in a recursive manner, as [27] note "of the estimate $\hat{\pi}_m^{(t-1)}$ for t-1 samples and the ownership $o_m^{(t)}$ of the last sample":

$$\hat{\pi}_m^{(t)} = \hat{\pi}_m^{(t-1)} + 1/t(o_m^{(t)} - \hat{\pi}_m^{(t-1)}) \quad (17)$$

By equating 1/t to a=1/T more emphasis is given to the newly acquired frames and the older frames have an exponential decaying effect, as previously discussed, on equation (17) [28]. It is important to include the influence of "prior knowledge for multinomial distribution" [28] by utilizing the Dirichlet prior $P = \prod_{m=1}^{M} \pi_m^{c_m}$ []. Coefficients c_m indicate the amount of "prior evidence" [28] for pixel values that belong to a certain cluster m. The c_m coefficients have negative values ($c_m = -c$); the reasoning behind this implementation is, in order to support the existence of cluster m, the data must support this assumption [28]. Therefore, it is concluded that:

$$\hat{\pi}_m^{(t)} = \frac{1}{K}\left(\sum_{i=1}^{t} o_m^{(i)} - c\right) \qquad (18)$$

As [] define, $K = \sum_{m=1}^{M}\left(\sum_{i=1}^{t} o_m^{(t)} - c\right) = t - M_c$ equation (18) is then:

$$\pi_m^{(t)} = \frac{\hat{\Pi}_m - c/t}{1 - Mc/t} \qquad (19)$$

In [27] define $\hat{\Pi}_m = \frac{1}{t}\sum_{i=1}^{t} o_m^{(t)}$ the "ML estimate from (16) and the bias from the prior is introduced thought c/t" where ML is defined as Maximum Likelihood. By defining $\frac{c}{t} = c_T = c/T$ the bias remains constant for the predefined frame set T. The recursive equation of (18) can be written as [27]:

$$\pi_m^{(t)} = \pi_m^{(t-1)} + \frac{1}{t\left(\frac{o_m^{(t)}}{1 - Mc_t} - \pi_m^{(t-1)}\right)} - 1/t\left(\frac{c_T}{1 - Mc_t}\right) \qquad (20)$$

The [27] makes the hypothesis that it is expected a small number of clusters M exist and bias c_t has a small value, for simplicity it can be assumed that $1 - Mc_t \approx 1$. The final recursive equation that the MoG2 utilizes to update $\pi_m^{(t)}$, taking the place of (5):

$$\hat{\pi}_m \leftarrow \hat{\pi}_m + a\left(o_m^{(t)} - \hat{\pi}_m\right) - ac_T \qquad (21)$$

After each recursion the $\hat{\pi}_m$ coefficients need to be normalized to sum to one, as previously discussed [28].

The [27] describe the steps of the algorithm as such: the GMM starts with one cluster "centered on the first sample" and new clusters are concluded by using the previously discussed methodology. Clusters that are not supported from the data will be suppressed by the Dirichlet prior and clusters with negative coefficients $\hat{\pi}_m$ are removed []. To support a cluster with the coefficient a=1/T, c=0.01*T samples must provide said evidence and $c_t = 0.01$ [28].

The proposed system of this dissertation utilizes the above mentioned from the OpenCV's already built-in MoG2 function [3]. While the authors [27] presented a "balloon variable kernel density estimation" that is better suited for complex, non-static background frames, a more generic approach that makes no assumptions is preferred (i.e. MoG2). The total frame set, previously denoted as X with time period T, is kept at its default value, 500 frames; OpenCV denotes time period T as "history". It can be concluded from the previous mentioned that, by increasing the "history" length, the time it takes for the algorithm to remove a potentially detected object is increased, since oversampling occurs. However, this practise would increase memory consumption, hence why the decision to keep the default value.

For this implementation, the detection of shadows is considered to be not useful, therefore it is disabled. The learning rate is kept at its default value as well, -1, indicating that an automatically chosen learning rate will be implemented. Keeping the background model constant would be useful if the position of the camera, along with the illumination of the frame, would remain static. Re-evaluating the background model completely after each time period T, is assumed to be optimal for applications where the camera is always non-stationary. The camera's position, for this implementation, is assumed to vary, therefore considered optimal to have some learning rate that also varies. The squared Mahalanobis distance threshold is kept to its default value, 16 [].

The resulting spatial filtering coefficients (i.e. mask) evaluated from the MoG2 model, have the form of 255 if the pixel is considered to be part of the foreground model, 0 if it is part of the background model and 127 if pixel is considered to be part of an object's shadow. The dimensions of the mask are $[N \times M]$, where N and M correspond to the input frame's height and width dimensions. While the resulting mask is considered as a grayscale image, it can also be considered as a binary image, since the background pixels have 0 values and foreground 255, as previously mentioned. The mask is applied by either multiplying the mask with the current frame, pixel-by-pixel, or by performing a logical AND operation [3, 7]; in this implementation, the latter operation is performed

due to its less computational expensive nature in comparison to the multiplication operation.

3.3 Forming General to Specific Rules

In order to estimate the number of objects present, the number of Gaussian kernels present in each colour channel must be detected. The Gaussian kernels of interest can be found by evaluating the histograms of each colour channel of the filtered frame (i.e. background subtracted). An algorithm, that the author of this dissertation has developed, similar to the CN2 algorithm, developed by [12], and an alternative version described in [12], has been implemented. The developed algorithm is part of the Sequential Covering Algorithms [12]. The core idea of this category of algorithms is learning one rule, discarding the data it covered, then iterate the procedure until a condition is satisfied.

As their name suggests, the input data are segmented into a smaller set of data and a rule covering this specific set of input data is learned. The algorithm learns one rule at each iteration, removes the trained data and continues with the iteration, until some hypothesis condition is satisfied [12]. The search of the rule that satisfies some attribute's condition is greedy in its nature, since no backtracking occurs, therefore the learned rule does not guarantee the most optimal results, since the search must be adapted to any set of inputs [12].

To initialize the search for the best rule, determined by some performance metric and variable attribute value, the most general rule must be defined that is common for each sequence of input data. This method is referred to as general to specific rule search. [12]. The Learn-One-Rule algorithm implements the general to specific rule search, creating at each step of the search a single descendant that further tests the attribute's performance. The system learns the rule that yields the best performance [12].

To minimize the greediness of the algorithm, a beam search can be introduced into the algorithm, that keeps a list of the best rule candidates found so far, instead of a constantly updated single best rule [12]. However, this does not guarantee that the learned rules will produce optimal results.

The search steps required to learn a set of n rules that contain k attribute tests, as [12] denotes, “in their preconditions” can be estimated by evaluating n*k. The advantages of this type of algorithms are, the direction of the search is clearly defined by a single maximally generated hypothesis [12], the trained data are evaluated after the evaluation of the attribute hypotheses and finally, potential noisy data have minimal influence on the algorithm’s performance since the search is based on the performance of the hypothesis [12]. As [12] notes, it is possible to further process the resulting rules by performing some post prune operations to remove rules that potentially produce suboptimal results, and some preconditions can be stopped from influencing the search in some clearly defined cases [12].

The CN2 algorithm utilizes some methodologies that are found in the AQ and ID3 algorithms. More specifically, it implements the search method of the AQ algorithm, and evaluates the specializations in a similar manner with the ID3 algorithm [5]. The algorithm proceeds with a top-down search and stops the iteration of learning new rules if some statistical metric indicates that it is not significant to do so; this procedure is similar with decision tree pruning methodologies. CN2 produces ordered “if-then” rules in a similar manner as to learn one rule methodology. As [5] states, a drawback of ordered rules is the lack of rule interpretation since each rule depends upon its preceding rule.

The algorithm operates in the same iterated manner as previously described, with notable features being a pruned general to specific search, the retention of best attributes, denoted as stars, a beam search of the retained complexes (i.e. variable space of the star) [5]. At each iteration the star is trimmed until no remaining candidates remain. The heuristic decisions that are made during the training process are as follows. By utilizing the “information-theoretic” entropy:

$$Entropy = -\sum_i \pi \log_2 \pi \qquad (22)$$

from the evaluated probability distribution of the current data, the algorithm can determine the quality of the complex, and solve potential conflicts should the maximum size of the star is achieved [5]. Entropy also guides the direction of the search to more significant rules.

The other heuristic decision is the evaluation of significance for the evaluated complex. The significance is evaluated by calculating the frequency distribution “of

examples among classes satisfying a give complex" [5], denoted as $F = [f_1, \dots, f_n]$ and the expected distribution $E = [e_1, .., e_n]$; by evaluating the likelihood ratio statistic:

$$2 \sum_{i=1}^{n} \log(\frac{f_i}{e_i}) \qquad (23)$$

the system can determine whether potential regularities are due to chance. This evaluation, under certain conditions, is distributed similar to chi square with n-1 degrees of freedom [].

By utilizing the two heuristic functions, the algorithm can determine "whether complexes found during search are both "good" and reliable" [5].

As previously discussed, the developed algorithm is a Sequential Covering Algorithm therefore, each colour channel's histogram is segmented into 8 plain and overlapping windows, containing 32 values. The overlapping values start at the median of each plain window's values, and the last overlapping window contains only 16 values. The search begins by first determining the windows that are of interest; said windows are defined with the following criteria in mind:

- Mean value of window is greater than 0
- Max value of window does not reside at the tails of the variable space
- Max value is over 0.4

A rule-based tree evaluates performs a general search as to which window contains the greatest max value. More specific rules test which window fulfils the rest of the criteria and captures the window of interest. Should a conflict arise and both windows have the same max value and both windows fulfil the above criteria, the window with the greatest mean value is selected.

The resulting windows of interest are fed into the general to specific learn one rule function of the algorithm. The evaluation of the similarity on the current plain and overlapping windows occurs by the KS statistical test, which is discussed in the next subchapter, instead of evaluating entropy and/or significance. Should the predefined criteria suggest the alternative hypothesis is true, the algorithm outputs the already evaluated windows of interest. This can be regarded as the most common rule, that was described previously. In the case where the null hypothesis is true, the learn one rule portion of the function iterates, further expanding the plain and overlapping windows;

The testing of the KS test results on the newly expanded windows occurs, the evaluated values that do not fulfil the alternative hypothesis criteria are discarded, and the algorithm returns the amount and direction of the expansion that is required to occur on the previously selected windows of interest.

It can be noted that the proposed algorithm behaves similarly with the CN2 algorithm and is part of the sequential covering algorithms, previously described. However, in its current implementation, the algorithm does not employ any post processing pruning methodologies, that are found in the CN2 algorithm, nor employs any statistical measure of the present information.

3.4 Hypothesis Evaluation (KS-Test)

The Kolmogorov-Smirnov statistical test is a non-parametric goodness of fit test, that evaluates the similarity of the shape between two distributions. The KS test evaluates the deviation of the observed cumulative distribution function (CDF) from the hypothesized CDF [14, 17]. The general mathematical steps that constitute the KS test are as follows. The sample data are rearranged in an ascending manner of magnitude. The observed CDF is evaluated by []:

$$F^0\left[x_{(i)}\right] = i/n \qquad (24)$$

By utilizing the hypothesized CDF, the theoretical distribution $F_x(x)$ can be obtained; the difference between the two CDF's is evaluated by calculating []:

$$D = \max_{1<i<n}\{\left|F^0\left[x_{(i)}\right] - F_x(x)\right|\} \qquad (25)$$

Afterwards, the significance level, denoted as a, is defined. The null hypothesis (H_0) is rejected if D > a and the alternative hypothesis is accepted (H_1). Should D result be less than significance level a, the null hypothesis is accepted. This step of the test is similar to the x^2 distribution test. However, the chi squared test is a "large-sample test" [14, 17] while the KS test can be performed on any size of samples. It can also be noted that the statistic metric D does not depend on the tested distributions, hence the non-parametric behaviour. There are some noteworthy disadvantages of the KS test; the KS test as [14, 17] denotes, "is strictly valid only for continuous distributions". However, the

most important drawback of the KS test is, due to its evaluation process, it tends to disregard significant differences that are found at the tails of the distributions in favour of differences that can be found at the centre of the tested distributions [14, 17, 20]. Therefore, the KS test lacks in power in comparison to some other statistical test that uniformly tests the variable space [14, 17, 20].

For this implementation, assuming that the histograms of the variables have been evaluated, the CDF of both histograms is evaluated by [5]:

$$u_{ci} = \sum_{j=1}^{i} u_j / N_u \qquad (26)$$

Where u_{ci} the realization of the variable that is tested, and N_u the sample's population. Therefore (23) is evaluated by:

$$T_{ks} = \max_i |u_{ci} - u'_{ci}| \qquad (27)$$

Where u'_{ci} denotes the CDF of the second distribution.

For this implementation the KS statistical test is utilized in two applications. As discussed in the previous subchapter, it is utilized as the attribute that indicates how many rules need to be learned in order to satisfy the attribute's condition. For this case, the H_0 hypothesis is defined as the shape of the two entered histograms overlaps significant (i.e. histograms have same shape) therefore further expansion of the current windows must occur, as discussed in the previous subchapter; H_1 hypothesis noted as the shape of the two histograms does not overlap significantly, therefore no further rules need be learned, simply output the resulted windows of interest.

The other application of the KS statistical test is to determine whether the MoG2 algorithm has converged to the object(s) of interest, therefore no further background subtraction is needed. For this application, the H_0 hypothesis is defined as the histograms of the current and past frame overlap significantly, therefore the MoG2 has converged to the desired point. H_1 hypothesis is defined as, the histograms of the current and past frame do not overlap significantly therefore, further background subtraction is required.

3.5 Methodology Overview

Based on what has been presented so far, the overview of the implemented methodology is briefly discussed, and its corresponding block diagram is presented below.

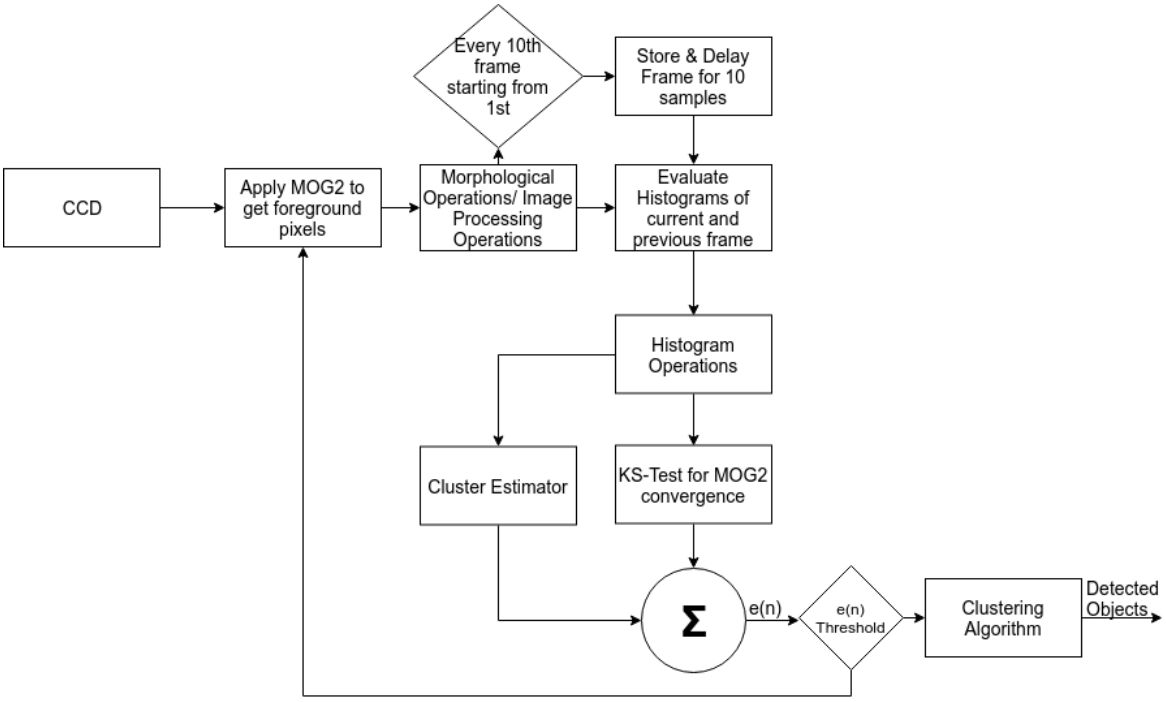


Figure 1 Overview of methodology

In the initial stages of the system, the input frame enters a hypothesis generator, namely the Mixture of Gaussian Kernels (MoG2) algorithm in order to detect the foreground. This process requires several samples, as previously noted, in order to completely subtract the background. The frame then is processed in order to capture the contour of the potential objects (i.e. foreground) that are present in the frame.

The evaluation of the BGR coloured histograms of the current frame and previous frame occurs. The evaluated histogram most dominant value for both frames is assumed to be 0 due to the background subtraction operation, and that effect can lead to overshadowing potential pixel values that contain useful information (i.e. detected objects). To overcome this issue, various filtering operations have been implemented and will be discussed in Chapter 4; the results of each filtering operation will be presented as well.

The normalization of both processed histograms occurs, and the results are fed into the Kolmogorov-Smirnov statistical test, that is responsible for evaluating if the MoG2 algorithm has converged; the normalized histograms are also fed into an algorithm that implements plain & overlapping windows for every colour channel.

The resulting windows are utilized as inputs for a rule-based tree, a part of the sequential covering algorithm as previously described, that is responsible for selecting the most potential windows of interest by forming general to specific rules. This process occurs for the windows of both histograms and for every colour channel.

The implementation of adaptive windows of interest by the Learn-One-Rule function, utilizing the KS statistical test attribute, occurs and the number of clusters for the clustering algorithm are evaluated based on the number of the resulting adaptive windows.

Other statistical means of concluding if the MoG2 algorithm has converged have been implemented as well and discussed, along with their results, in Chapter 5.

As a final stage, the results of the KS test of current and previous histograms, the number of clusters and the results of other statistical test for the MoG2 are evaluated appropriately, in order for the system to decide whether it has converged and the present objects have been optimally detected, or further subtraction is required. Should the system's decision be that the MoG2 has converged, the number of clusters and frame are fed into the clustering algorithm, in order to separate objects that are potentially connected to one another, as well as differentiate between the detected objects.

Chapter 4: Experimental Design and Investigation

This Chapter presents some initial experiments to evaluate the results of the core procedures described in the previous chapter, as well as the necessary implementations

that are needed in order to accomplish the aims and objectives that have been described in Chapter 1. This chapter also showcases some initial algorithmic approaches, their corresponding results, that have been conceived based on the resulting data of the core methodologies. Some of the experimental algorithms have been rejected in order to achieve the soft real-time performance in favour of less computational expensive algorithms. Various image processing operations, not mentioned in the previous chapter, have been implemented as well with the purpose of capturing the contours of interest and, in some cases, further filtering out the background from the foreground of interest (i.e. objects of interest).

Four experiments were performed, in total, with the camera being stationary and for the background a white cardboard construction was utilized, in order to keep the background model as simple as possible. For every experiment, the MoG2 firstly removed the entire background, and afterwards a series of objects were presented in the foreground. A frame was manually captured, when as little as possible background remained. All testing objects were household mugs, with the same shape but different colour. The first experiment involved the processing of a frame that contained only a blue mug along with some background artefacts. The experiment was repeated with a red mug, and similar algorithmic implementations were tested. A blue and red mug were placed close to one another in order to induce the problem of the algorithm considering two distinctly, for the human perception, objects as one. The same experiment was performed with a different set of colour mugs, namely blue and white, but in this experiment the blue mug was placed behind the white mug. The development and analysis for these experiments were performed in Python, utilizing the libraries NumPy, OpenCV, Scikit-Learn and Matplotlib. The frames were captured by a USB camera [3].

4.1 Image Processing Operations

The following image processing operations were performed prior to obtaining the frames utilized for the experiments discussed in the following subchapters. As previously mentioned, in order to apply the background subtractors resulting coefficients (i.e. mask) and filter the background from the frame, a logical AND operation

is performed. By utilizing this operation, pixel values that correspond to coefficients with zero values, become zero as well. Regions that are considered to be foreground have coefficient values equal to 255, as previously discussed, and are kept intact. The reasoning behind these values, besides the logical AND operation, is due to the data type of the frames being unsigned integer of 8 bin; the range of values for this type of integers is from 0 to 255 [3, 7]. This operation follows the truth table of the AND operation, as presented below.

Table 1 AND Truth Table

A	B	Out
0	0	0
1	0	0
1	1	1

The now filtered frame is morphologically dilated to close potential empty regions inside the contours of interest, thus closing the shape of contours. The structuring element for this operation is defined as a rectangle of dimensions 9x9. The resulting frame is binary in its form, and the mathematical definition of this operation is as such [22]:

$$g(x,y) = \begin{cases} 1, & if\ st\ hits\ f \\ 0, & otherwise \end{cases} \qquad (28)$$

where g denoted as the resulting binary image, f the input frame, and st the structuring element. The term “hits” refers to the condition that any pixel’s value inside the structuring element is non-zero. The entire kernel of the structuring element is filed with 1’s in this operation [22].

Afterwards, the morphological closing of the frame occurs. The morphological closing operation is dilation followed by erosion. Further dilating the frame ensures the complete enclosure of the contours of interest, and morphological erosion suppresses each contour’s expansion in its perimeter [24]. The mathematical definition of the erosion operation is as such [22, 24]:

$$g(x,y) = \begin{cases} 1, & if\ st\ fits\ f \\ 0, & otherwise \end{cases} \qquad (29)$$

The resulting binary frame that has been morphologically closed, is utilized as a mask and applied in the same manner as the background subtractor's mask, via logical AND operation [3,7, 24]. An overview of the processes that occur is presented in the form of a block diagram.

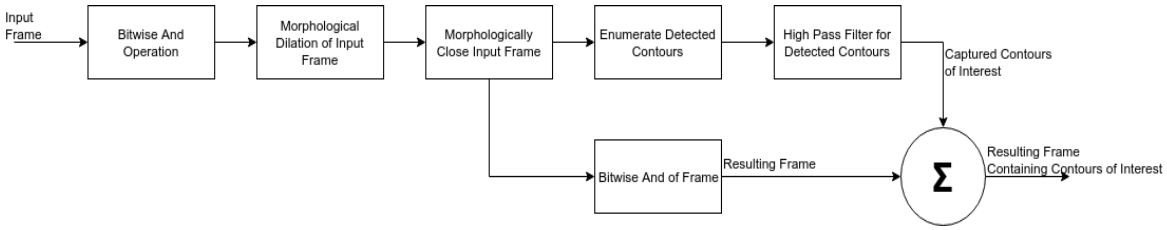


Figure 2 Image processing operations

4.2 First Experiment – Blue mug

As previously stated, a frame was manually captured while the MoG2 algorithm subtracted the background; the resulting frame contains a blue mug, presented below, along with some background artefacts.

Figure 3 Blue mug Frame.

The histograms of each colour channels for this frame were evaluate in order to determine the Gaussian kernels that are present in the current frame, indicating that a blue coloured object is present.

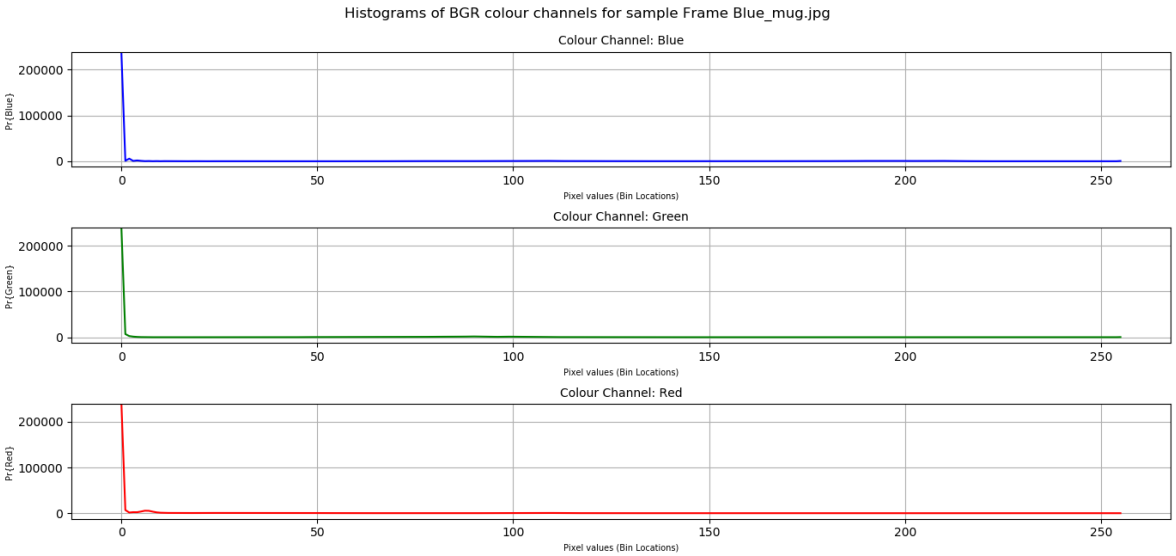


Figure 4 Histograms of each colour channel for Blue mug object.

It can be shown that the number of pixels that have value 0 overshadow any other present value, thus rendering the evaluation of present Gaussian kernels almost impossible. It would be expected that a large enough kernel would be observable in the blue channel, but this is not the case. Hence the need to somehow filter or suppress the effects of the zero valued pixels. By evaluating the FFT of each color channel's histogram, it results to the following figure.

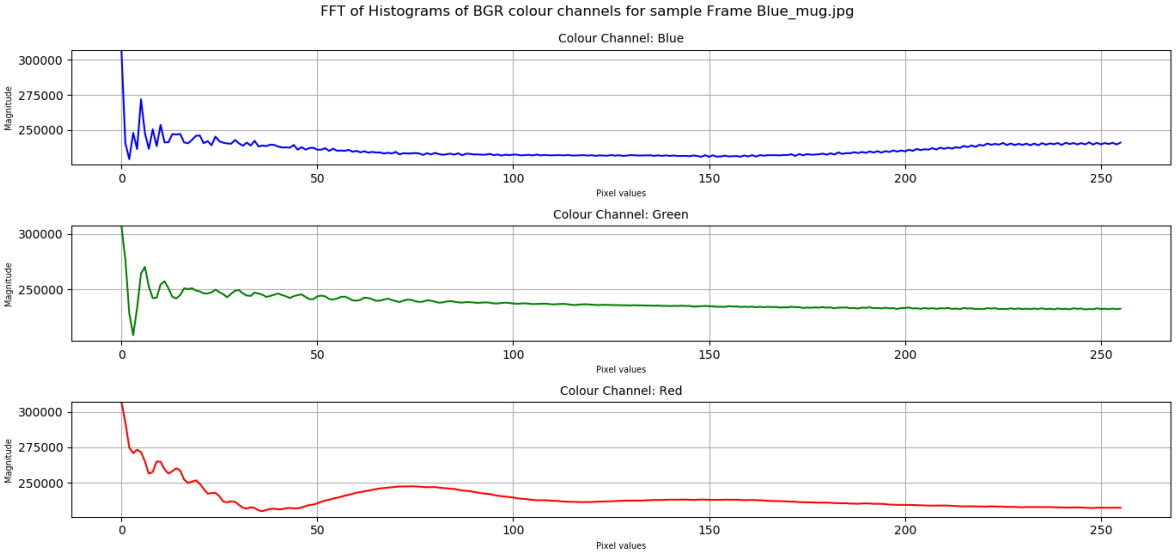


Figure 5 FFT evaluation of each color channel's histogram

By the evaluation of the present “frequencies”, in this application magnitude of present pixel values, it can be noted that the values of the histograms appear to be highly cluttered.

4.2.1 Histogram Filtering

To obtain any meaningful measurement from the histograms of the current frame, as it can be concluded from the histograms and their corresponding FFT evaluation, two filtering operations have been implemented. The first one being a spatial filter and the second a band pass FIR filter.

4.2.1.1 Spatial Filtering

The implemented spatial filter evaluated kernels of interest with dimensions of 3x3. In this application, kernels of interest are defined as kernels whose mean value is greater than zero. Kernels that their mean value evaluates to zero, were discarded. The mathematical definition for this operation is as follows:

$$k(x,y,z) = \begin{cases} f(x,y,z), \ when \frac{f(x,y,z)}{9} > 0 \\ 0, \ when \ \frac{f(x,y,z)}{9} = 0 \end{cases} \qquad (30)$$

where k the kernels of interest and f the blue mug frame; index z denotes the colour channel that the operations occur, since this filter is applied to every colour channel.
The results of this operation can be seen below.

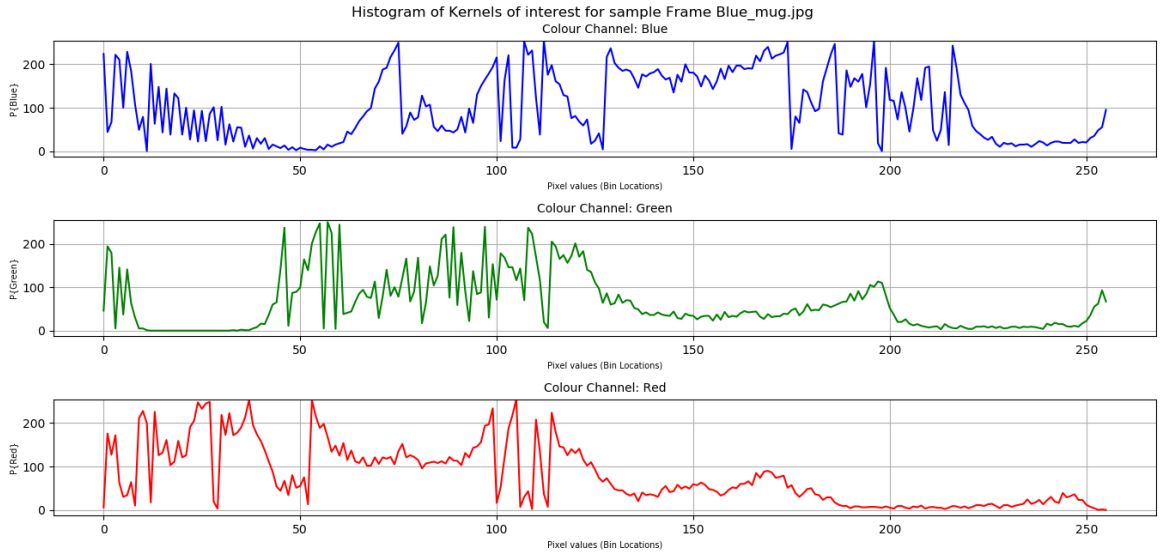


Figure 6 Spatial Filtering (Kernels of Interest) for Blue mug Frame

The implementation of this spatial filter can also be considered as a high pass filter, that allows any value greater than zero to pass. By evaluating the FFT of the now filtered histograms, it can be concluded that the filtering had a significant effect on the magnitude of pixel values, as shown below.

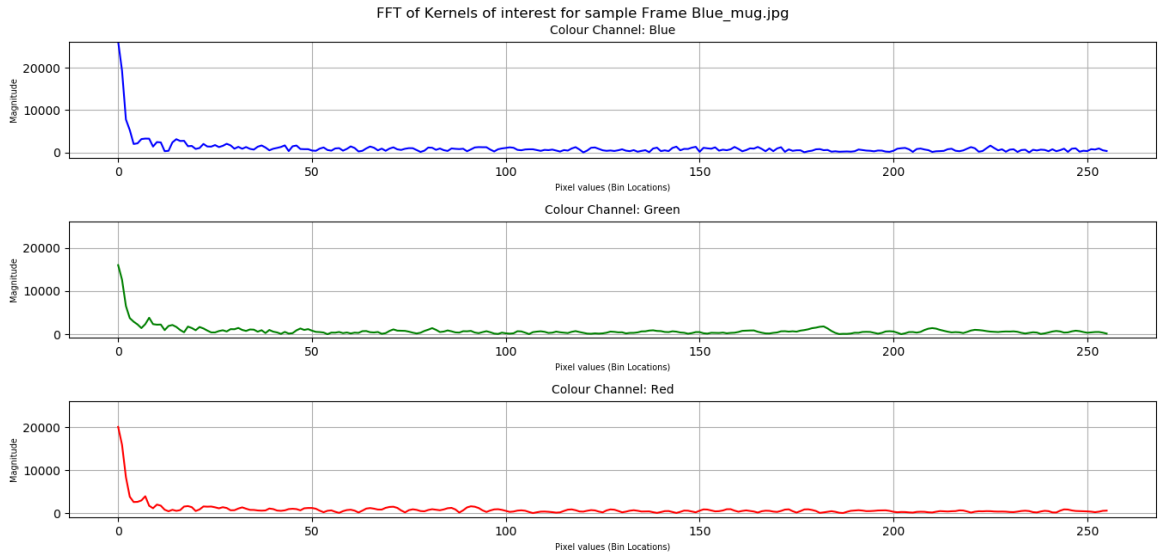


Figure 7 FFT of Spatial filtered histograms for Blue mug Frame

The resulting flowchart for the main structure of the algorithm is presented below.

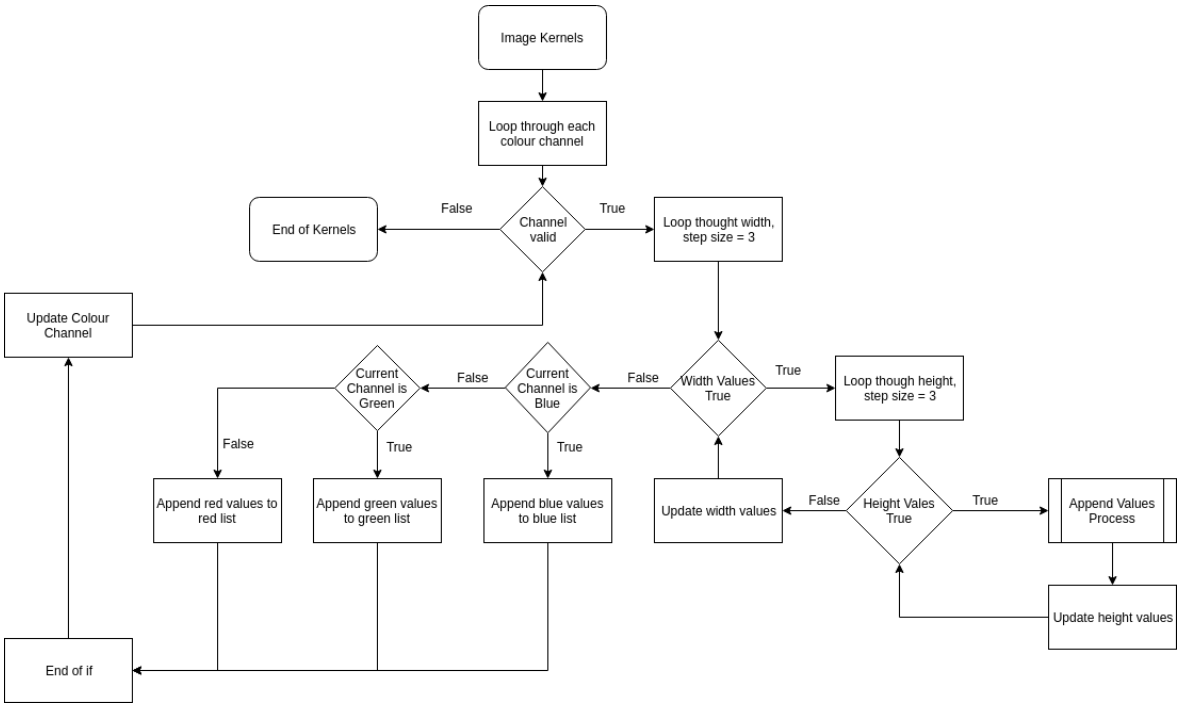


Figure 8 Spatial Filter main algorithm's structure

The flowchart for the Append Values Process is presented below.

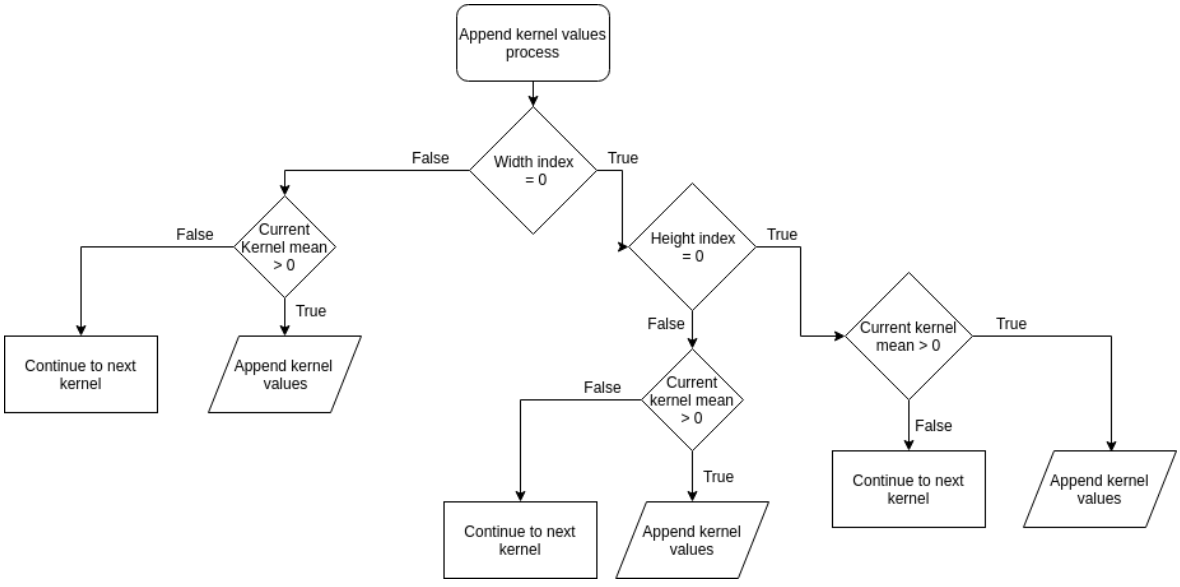


Figure 9 Append Values Process for Spatial Filter

The operation of the spatial filter is as follows. The algorithm iterates over each colour channel, and for each of the dimensions of the input frame, with step size of 3. The algorithm checks whether the iteration has reached any of the corners of the frame, evaluates the mean of the current kernel, and if the result is greater than zero, it appends

the kernel's values. Before the beginning of the iteration to the next colour channel, the resulting values are stored to the corresponding colour's list.

Based on the algorithm's results and its structure that has been show by its flowcharts, the following conclusions can be drawn. While the filtering has produced better results than the plain histograms, it appears to be somewhat cluttered with a lot of values that do not correspond to the hypothesized Gaussian kernel that represent the blue mug object. It is also important to note that this implementation might jeopardise the performance of soft real-time since it iterates over the entire frame's dimensions.

4.2.1.2 Band Pass FIR Filtering

A simpler, more robust and less computational expensive approach is the implementation of a band pass FIR filter. By implementing a FIR filter, it is ensured that reliable results will be produced. The upper and lower threshold of the filter must be evaluated for each colour channel individually, since it is assumed that the coefficients or thresholds for each colour channel will differ significantly [8, 13].

By evaluating the max value of each colour channel, starting from the third element, the overall max value, assumed to be the amount of zero valued pixels, and by performing:

$$lp_{thr} = \frac{hist\prime_{max}}{hist_{max}} \qquad (31)$$

where $hist'_{max}$ denotes the max value of the histogram that contains useful information and $hist_{max}$ the plain max value, assumed to be overshadowed by zero valued pixels, the low pass threshold can be evaluated.

To estimate the high pass threshold value, the assumption that the 20% of the max value containing useful information, will yield the desired results. Therefore, by evaluating:

$$hp_{thr} = hist'_{max} * 0.2 \qquad (32)$$

the high pass threshold can be obtained.

By obtaining the desired thresholds, the filter iterates over each colour channel's value and compares whether the current value is greater than the $hist'_{max}$ value or greater than hp_{thr} value and less or equal to $hist'_{max}$ value. Should the value be greater

than $hist'_{max}$, the current histogram value is multiplied by lp_{thr}. In the case where it fulfils the latter criterion, the histogram retains its value. For any other case, the histogram's value is set to zero [8, 13].

The algorithmic implementation of this filtering operation is presented in the following flowchart.

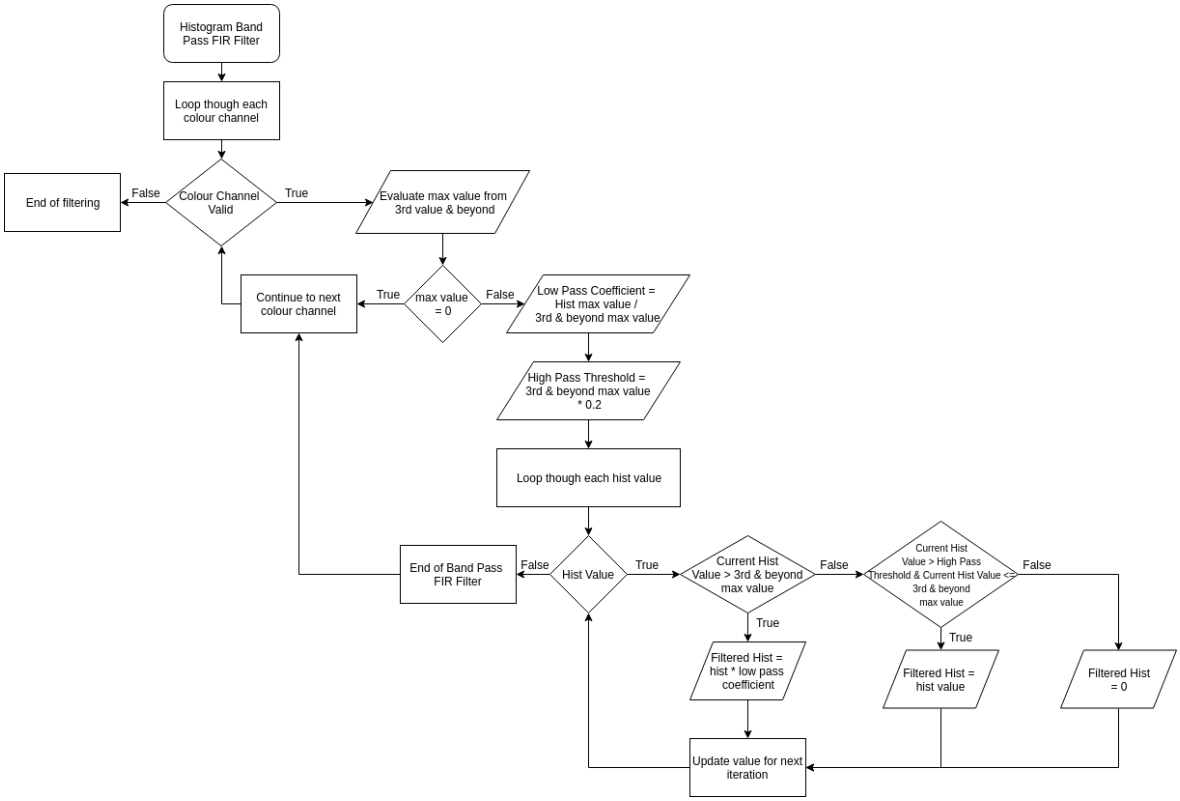


Figure 10 Flowchart of Band Pass FIR Filter

The process iterates over each colour channel, determines the max value of the present channel, evaluating from the third element. By utilizing equation (32) the low pass coefficient is evaluated and the algorithm loops for every value of the current histogram. By implementing if-then rules the algorithm determines whether the current value should be allowed to pass (i.e. high pass filter), multiplied by the low pass coefficient, or set to zero.

The resulting coefficients (i.e. thresholds), $hist'_{max}$ and $hist_{max}$ values for the blue mug frame are presented below.

Table 2 Results of Band Pass FIR Filter

$hist'_{max}$	$hist_{max}$	hp_{thr}	lp_{thr}
5691	235718	1138.2	0.0241433
2479	237490	495.8	0.0104383
5410	238982	1082	0.0226377

The resulting filtered histograms for each colour channel are presented below.

Figure 11 Band Pass FIR Filter Results

While the largest kernel appears to be in the red colour channel, it is hypothesized to be due to overshadowing effects that are still present in this channel. However, in the blue and red channel one cluster can be detected that might correspond to the present object. The cluster that is centred to pixel values 200 in the blue colour channel, is hypothesized to be the background artefacts that are present in the frame.

While it appears some Gaussian kernels that belong to the background cluster have passed into the filtered histograms, they are assumed to not have an effect on the next stages of the implementation, described in the following subchapters.

By evaluating the FFT for each colour channel, the following results are obtained.

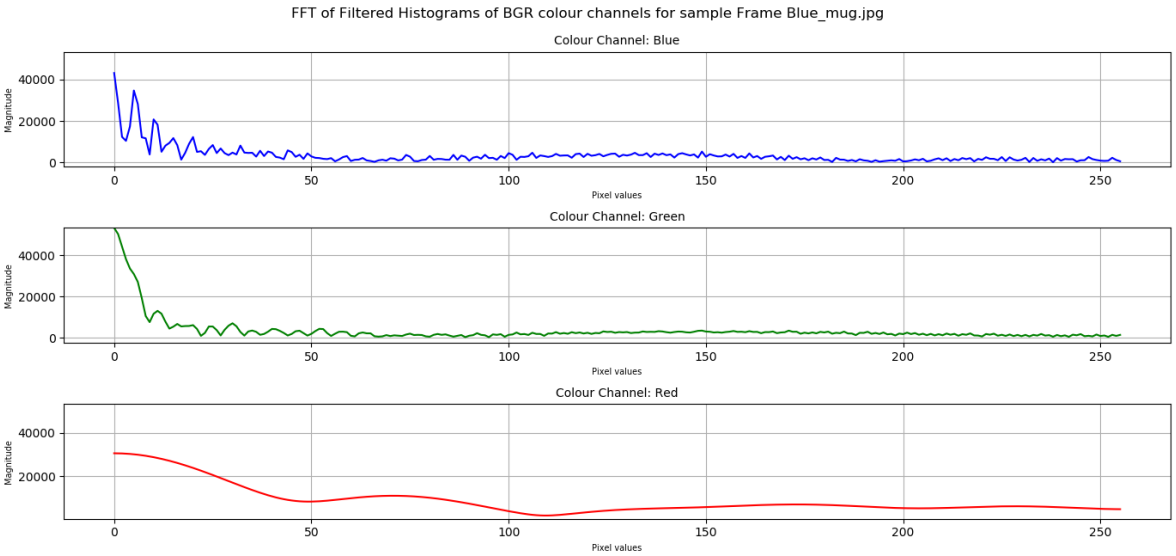


Figure 12 FFT of Band Pass FIR Filtered Histograms

Based on the results of the FFT of each colour channel, it can be noted that the filter produces more optimal results in comparison to the spatial filter implementation. Another benefit of this approach is less computational complexity, since the filter iterates over the histogram values that are very small in size in comparison to the actual frame's values.

4.2.2 Histogram Normalization

The normalization of each, now filtered, colour channel's histogram is an important step for the Sequential Covering Algorithm, as discussed in Chapter 3. This is due to how the algorithm threshold the minimum accepted max value of each window, further discussed below. The equation utilized for normalization is:

$$hist_{norm} = filt_{hist} - \min\left(filt_{hist}\right)/(\max(filt_{hist}) - \min(filt_{hist})) \qquad (33)$$

The results of the min and max values of each colour channel for the blue mug objects are as follows.

Table 3 Min and Max Values of filtered Histograms

Filtered Histogram Min	Filtered Histogram Max
0	5691
0	2479
0	5410

The resulting values correspond to the previous evaluated max values from the band pass FIR filter, as expected.

The flowchart of the algorithmic implementation is presented below.

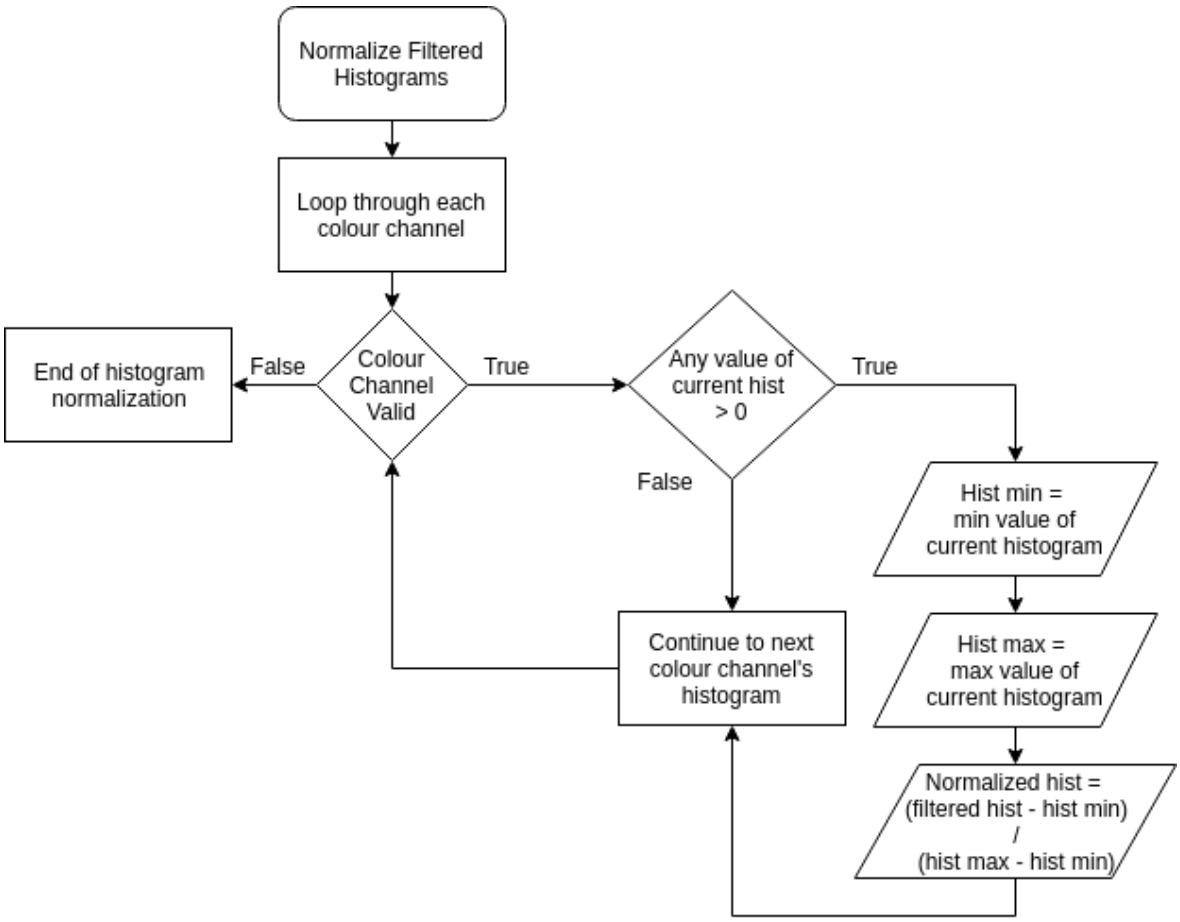


Figure 13 Normalization of Filtered Histograms

The implementation iterates over every colour channel, evaluating if the current colour channel is empty, to avoid zero division since an empty colour channel's max value would evaluate to zero, and perform equation (33).

The resulting histograms are presented below.

Figure 14 Normalized Filtered Histograms

The initial hypothesis that the Gaussian kernels found in the blue and green channel, appears to be correct based on the resulting normalized plots.

4.2.3 Plain and Overlapping Windows

As previously noted, the sequential covering algorithm that has been developed, utilizes a fraction of the input data at each iteration. For this reason, the implementation of plain and overlapping windows is crucial in order to deduce the windows that contain the information that is deemed as useful (i.e. windows of interest).

The reasoning behind the implementation of both plain and overlapping windows is to ensure that no information will be lost due to its segmentation. For each colour channel, its histograms are segmented into 8 windows of 32 values for the plain, and for the overlapping 7 windows of 32 values and the final window contains 16 values, as discussed in Chapter 3.

The structure of the algorithm is presented in the form of a flowchart, below.

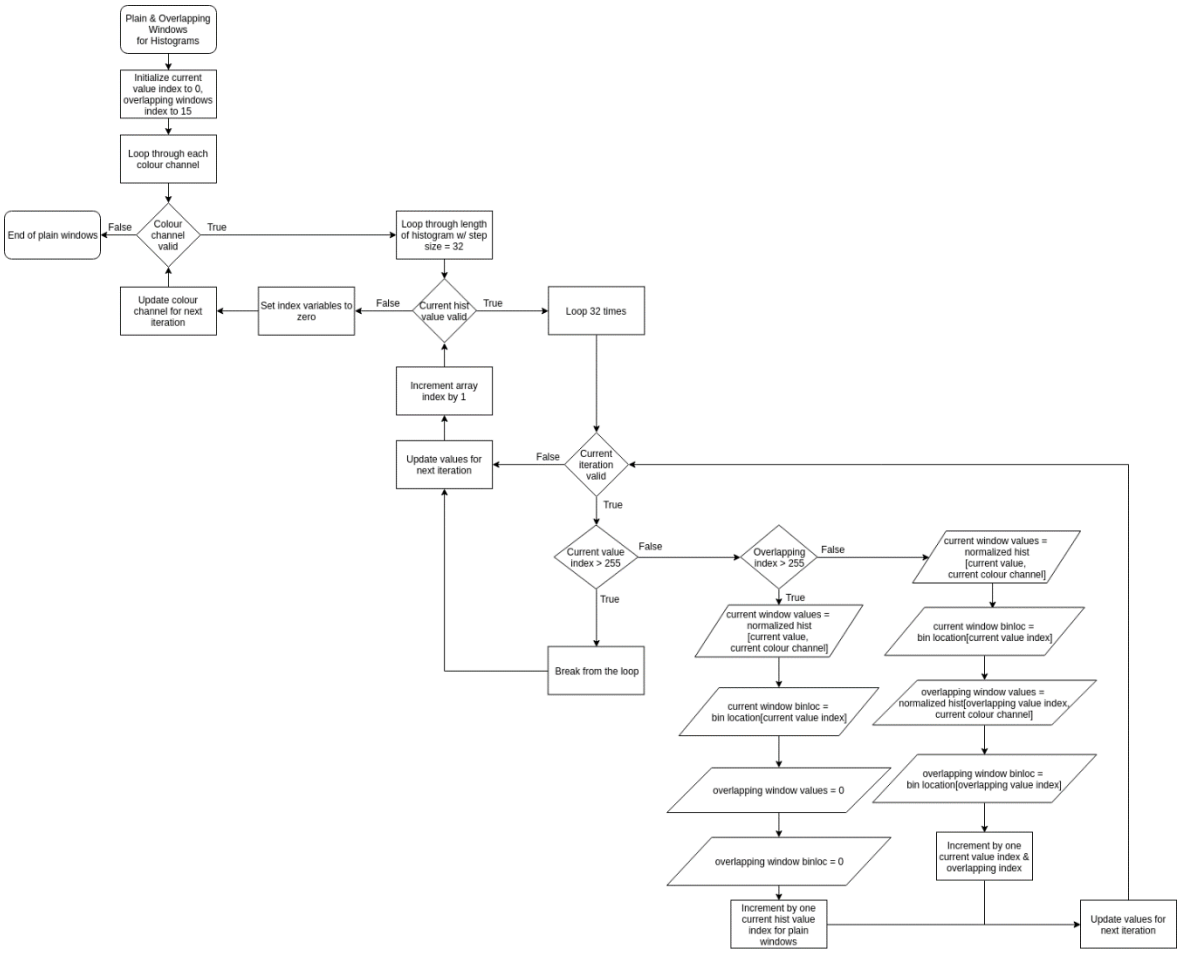
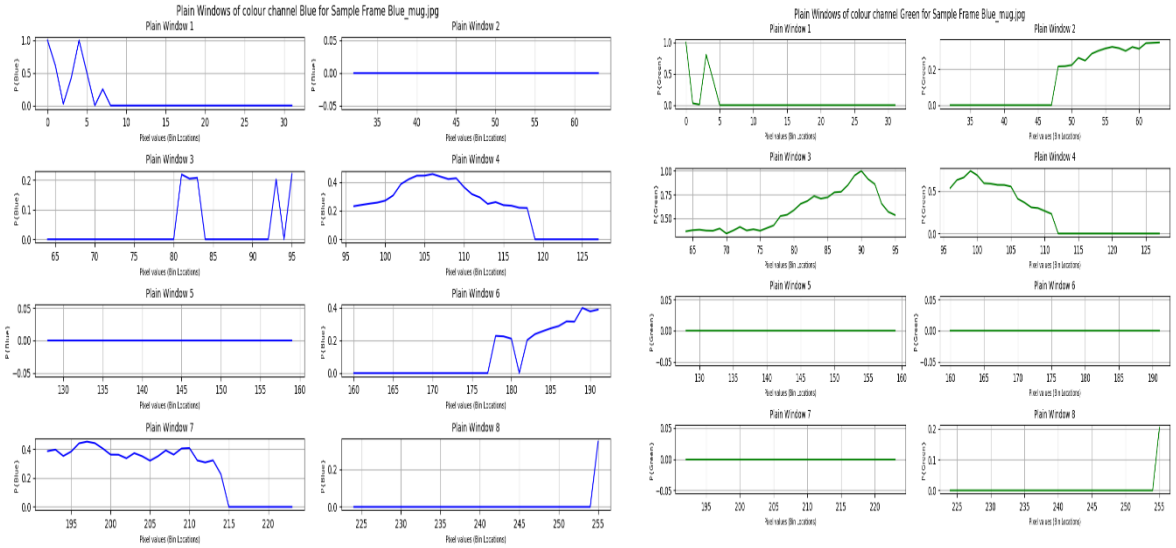


Figure 15 Implementation of plain and overlapping windows

While the algorithm performs the segmentation in a serial fashion, the implementation of both overlapping and plain is implemented in parallel.

The resulting plain windows for each colour channel are presented below.



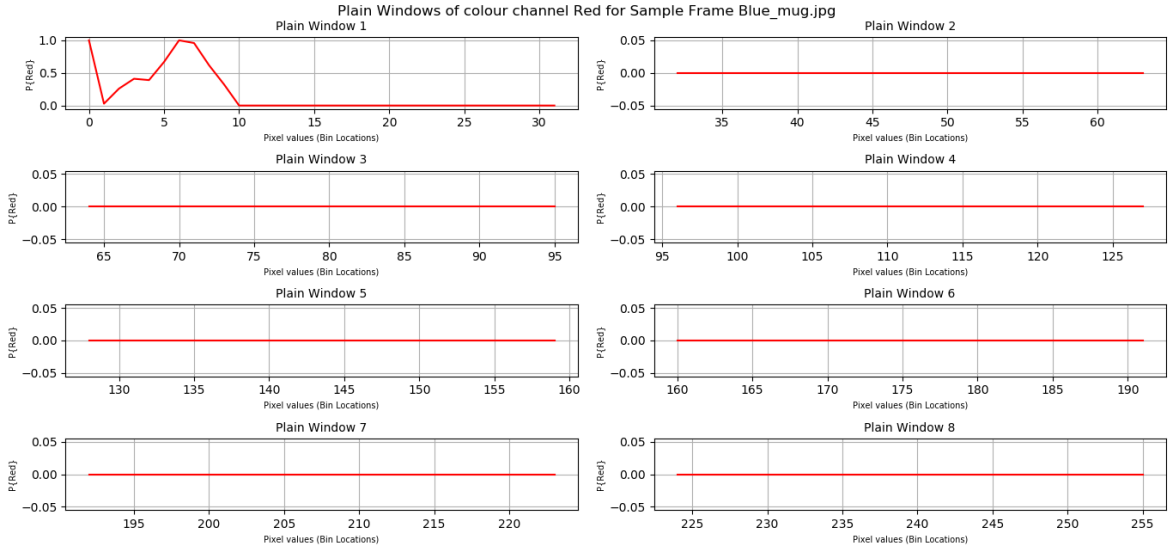


Figure 16 Plain Windows for Blue mugs Frame

The resulting overlapping windows are presented below.

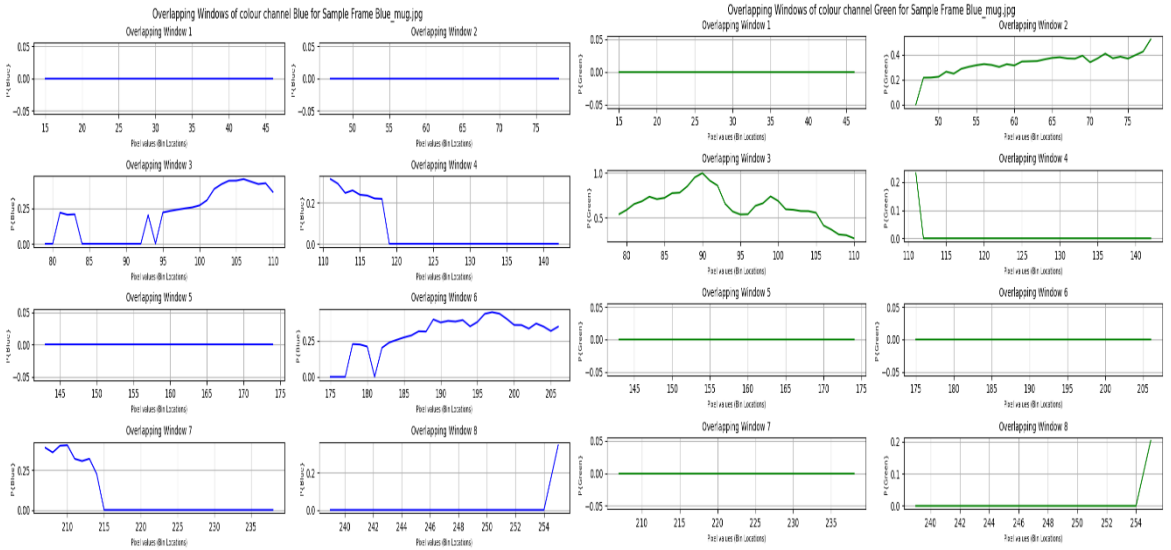


Figure 17 Overlapping Windows for Blue mug Frame

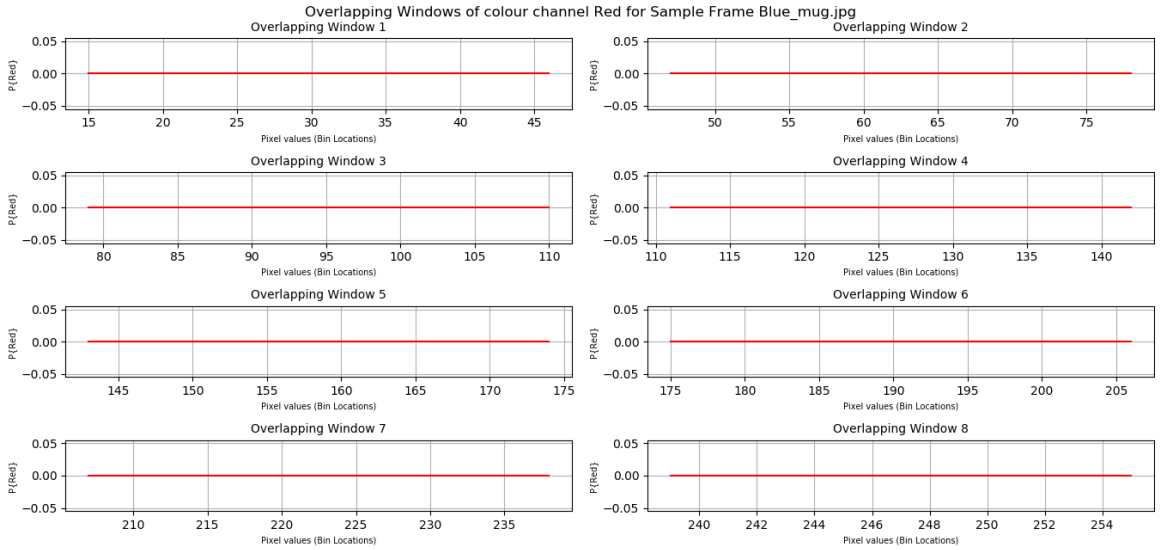


Figure 18 Red colour channel's overlapping windows for Blue mug sample frame

4.2.4 Windows of Interest

As described in Chapter 3, a general to specific rule, similar to the CN2 algorithm has been implemented in order to identify and capture the windows of interest. The criteria for classifying a window as of interest were clearly defined in Chapter 3 and are employed in the implementation of the algorithm.

The algorithm iterates over each colour channel and each plain and overlapping window. Should the algorithm occur the first set of windows, it evaluates the max value of the plain window starting from the 2nd element, to exclude the potential effects the background subtraction has on the histograms.

The logical evaluation of whether the max value resides in the tails of the variable space and the mean value of the window are performed in an iterative manner.
Afterwards the logical evaluation of the criteria, max value is beyond 0.4 and which window contains the max value are performed. The results are fed into the decision tree and the appropriate window, based on the logical results, is captured.

The flowchart of the main structure of the algorithm is presented in the next page.

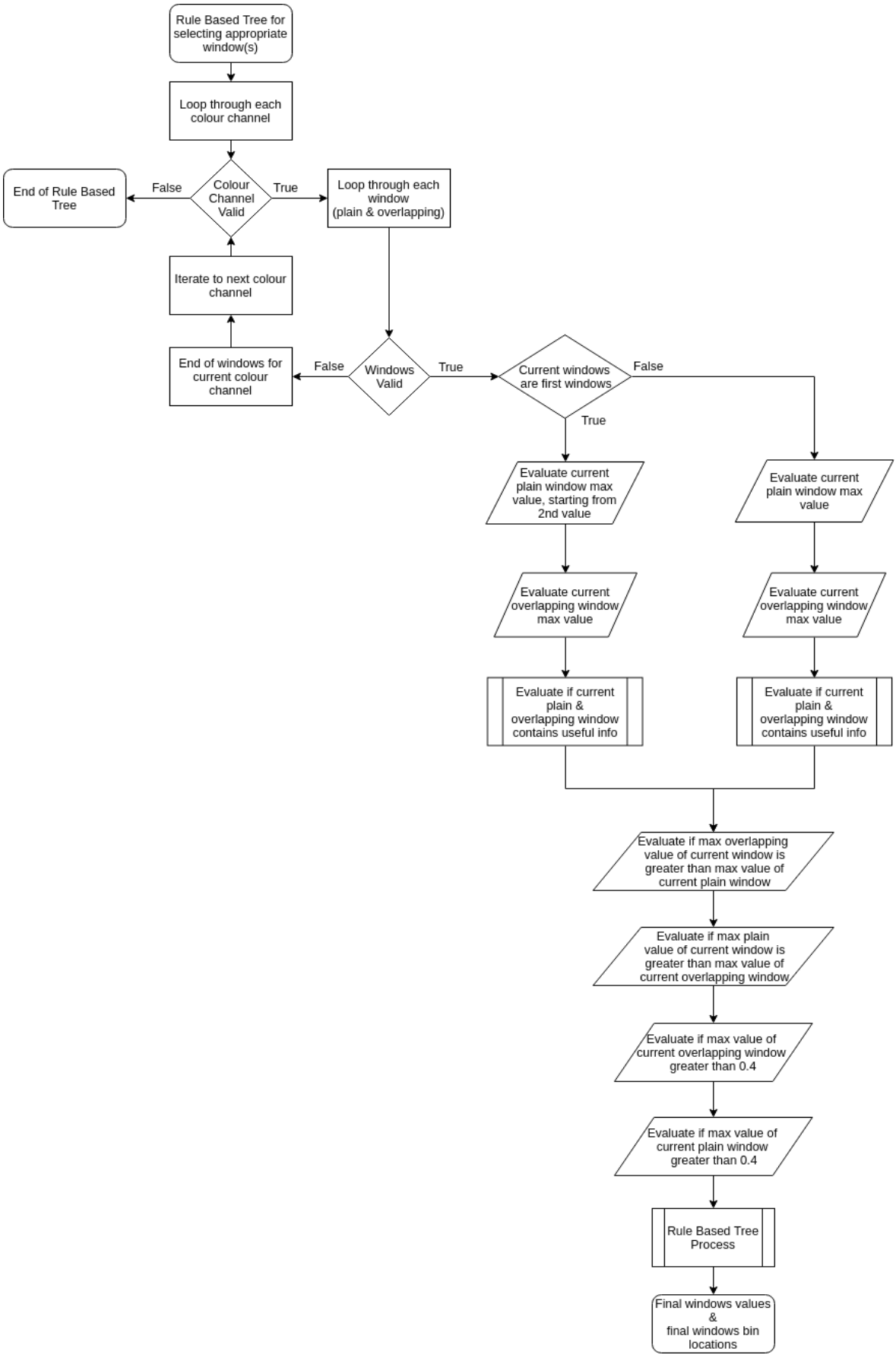


Figure 19 Skeleton of the General to Specific Rule Tree

The resulting adaptive windows for sample frame blue mug are as follows.

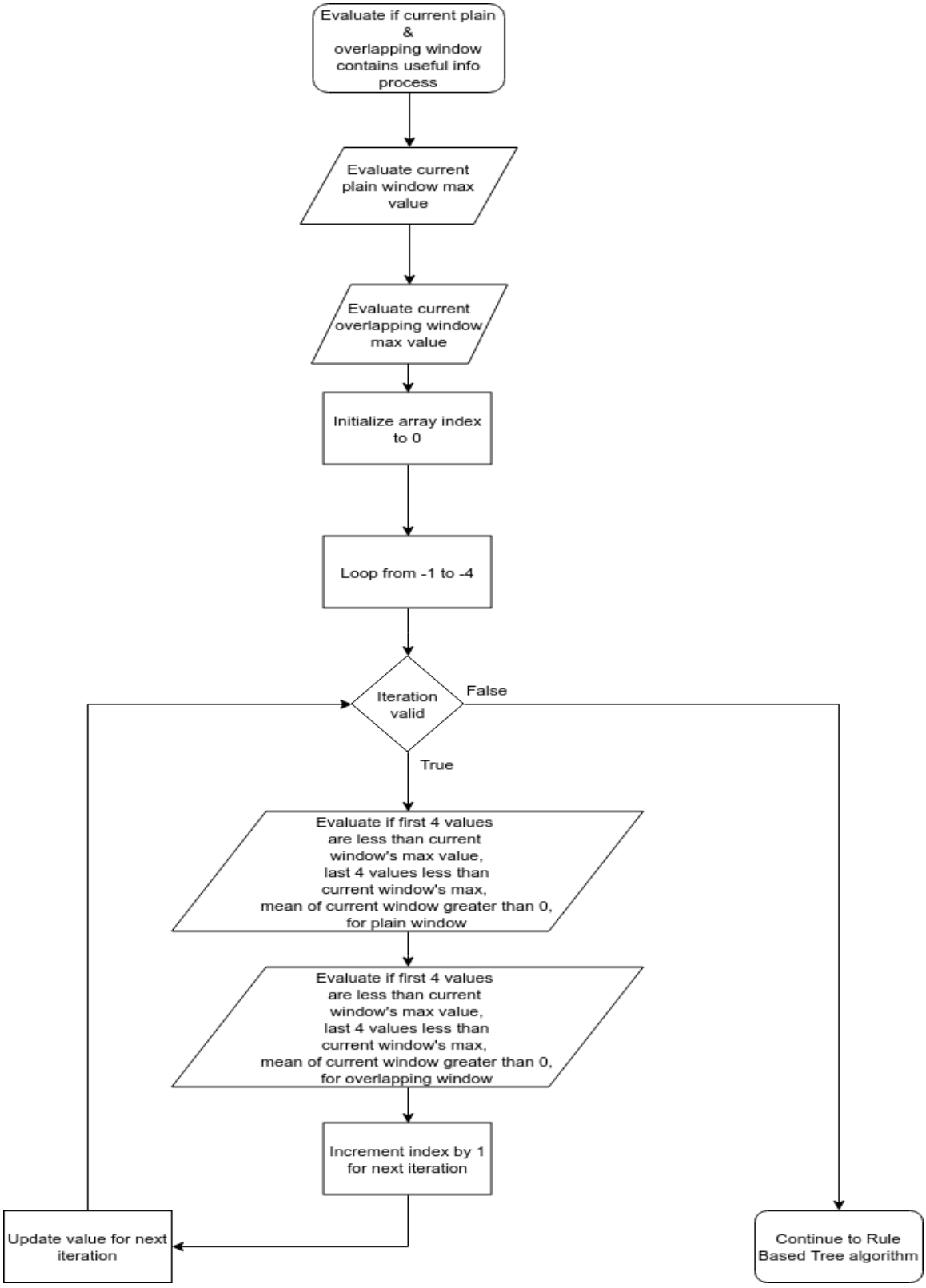


Figure 20 Flowchart for evaluating windows of interest

The flowchart for the rule-based tree process is presented below.

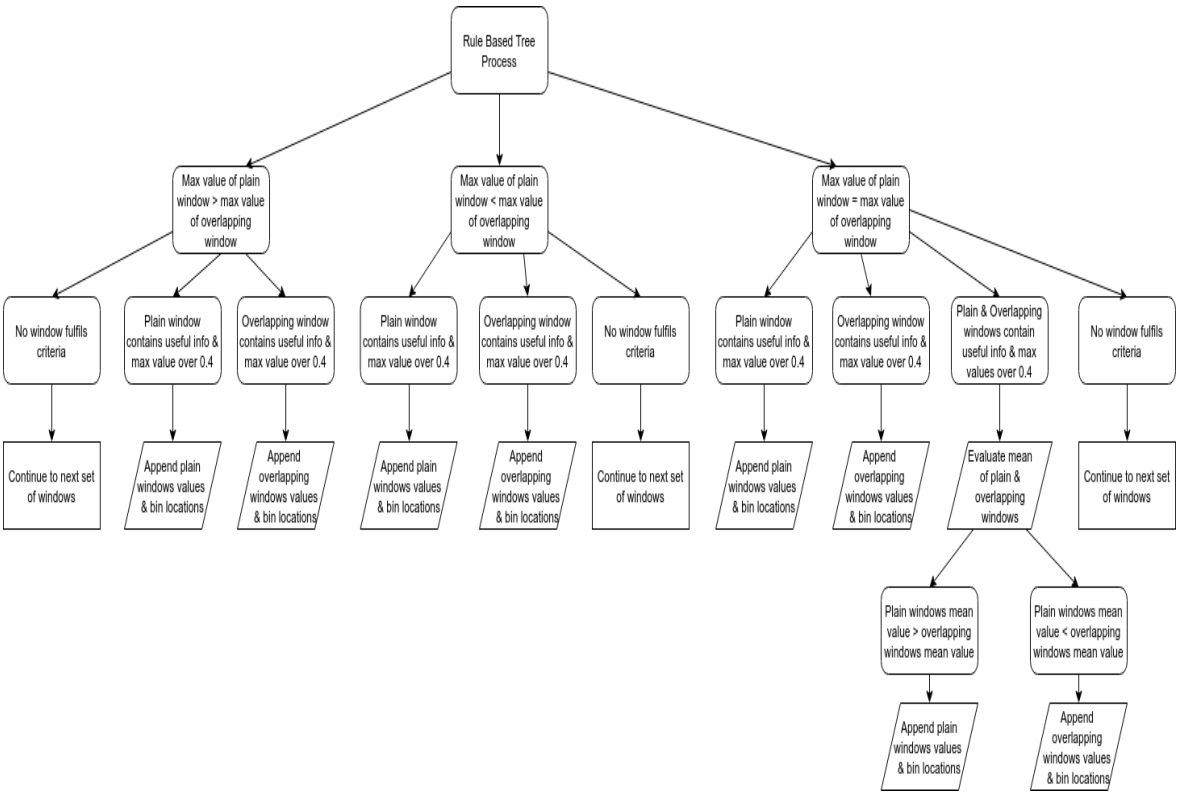


Figure 21 General to Specific Rule Tree Process

The resulting windows of interest for the blue mug frame are presented below.

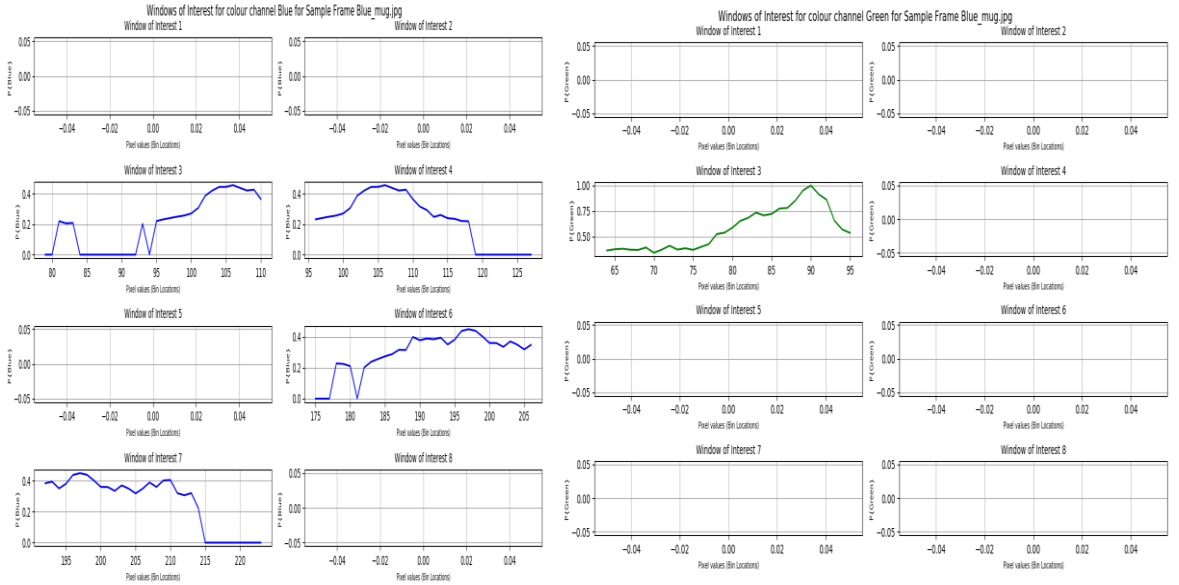


Figure 22 Windows of Interest of Blue mug frame for Blue and Green colour channels

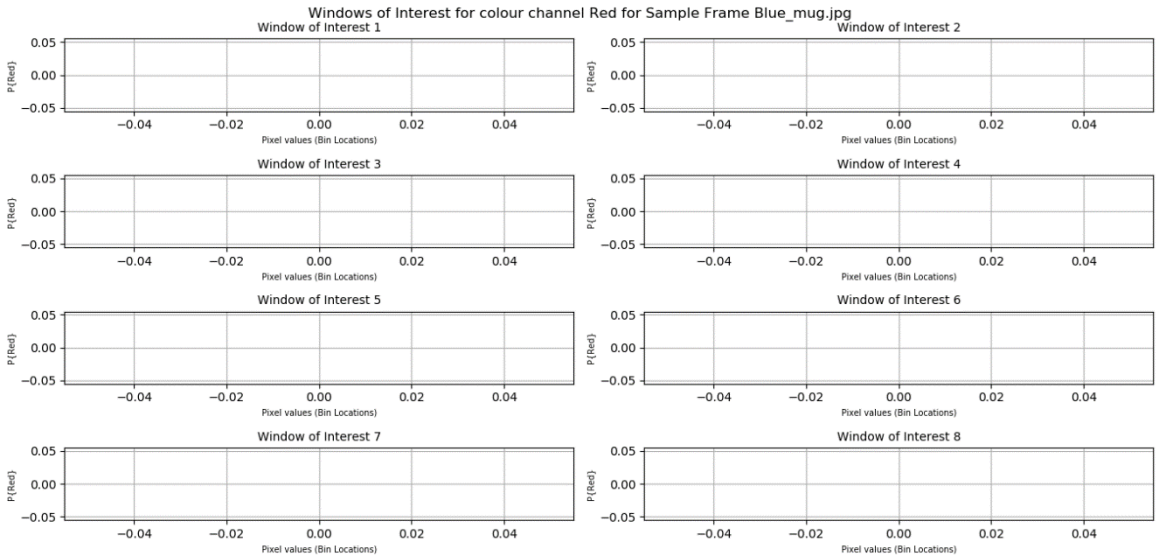


Figure 23 Windows of Interest of Blue mug frame for Red colour channel

4.2.5 Adaptive Windows of Interest

The function of the Learn One Rule function of the implemented algorithm occurs by evaluating the CDF of each plain and overlapping windows, and then performing the KS statistical test. As previously discussed, the null hypothesis is defined as the histograms shapes overlap significantly, and for the alternative hypothesis as the histograms shapes do not overlap significantly. To be more specific, the significance threshold a was placed above 0.5 for the alternative hypothesis; while it was previously described that the KS statistical test is not that powerful, the significance threshold was placed in this value to avoid the greediness of the algorithm.

The flowchart of the main structure of the implemented Learn One Rule function is presented in the next page

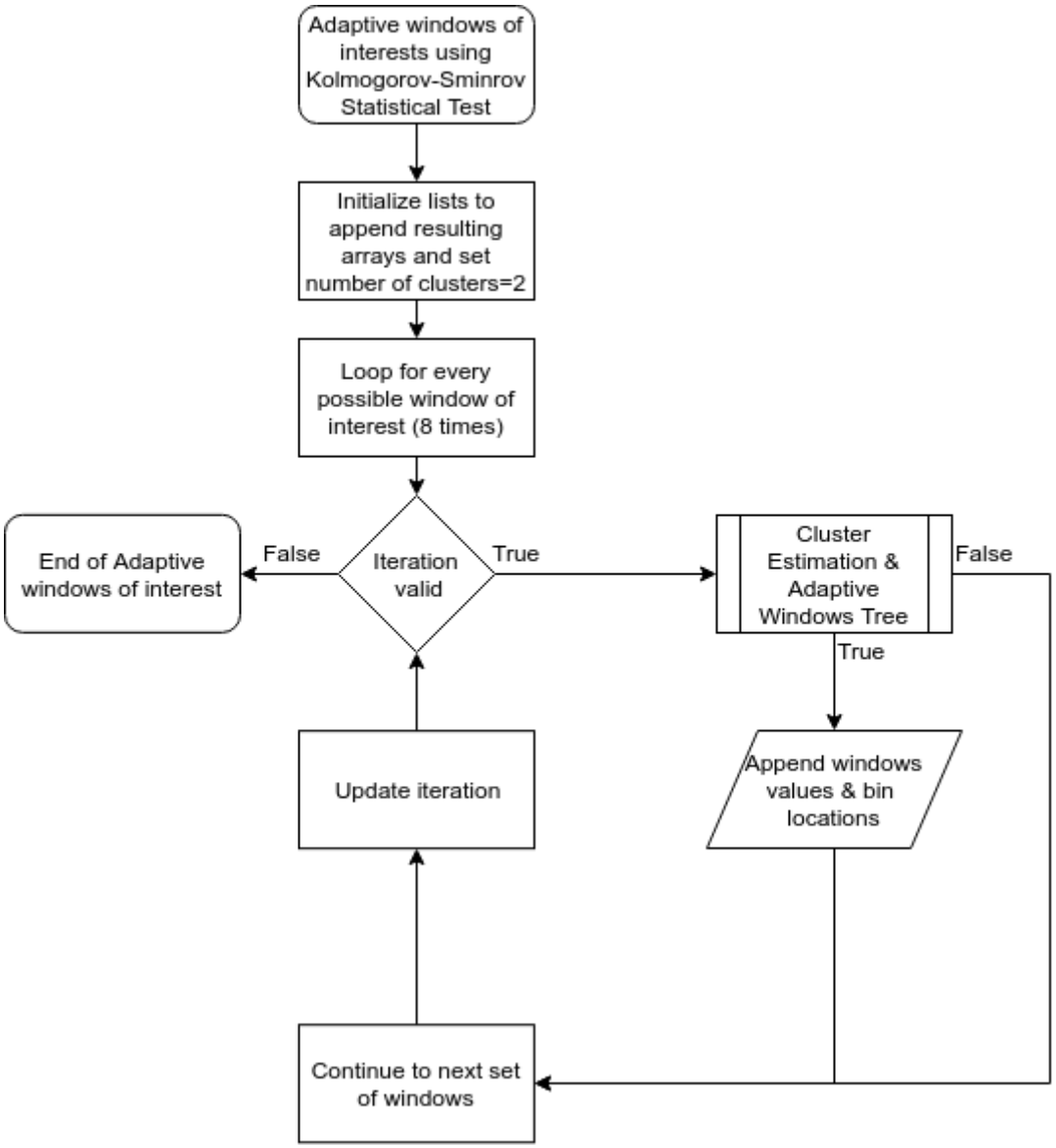


Figure 24 Main structure of the Learn One Rule

The cluster estimator & adaptive windows tree process flowchart is presented below.

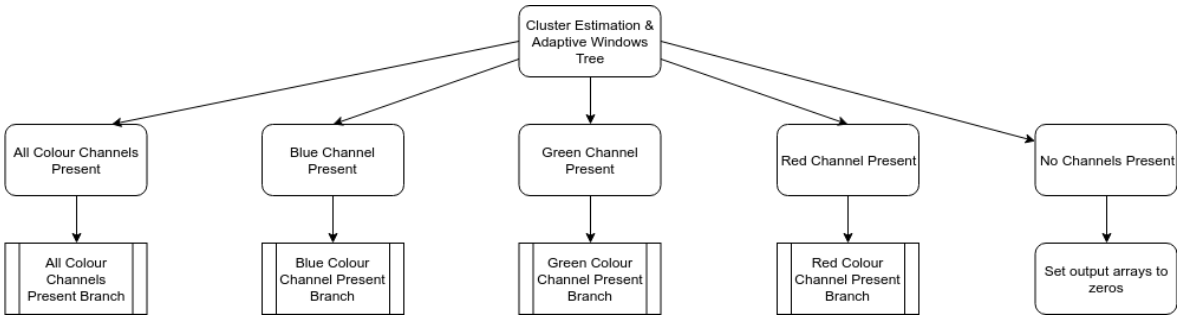


Figure 25 Main branch of the Cluster Estimator & Adaptive Windows Tree

While the implementation of a clustering algorithm is discussed on other experiments, where two objects are present, the estimator enumerates the resulting adaptive windows that have the same bin locations as one cluster, two windows with same bin locations as 2 clusters and 3 clusters for the case when all windows have unequal bin locations.

Below is presented the flowchart of branch for when all colour channels are present.

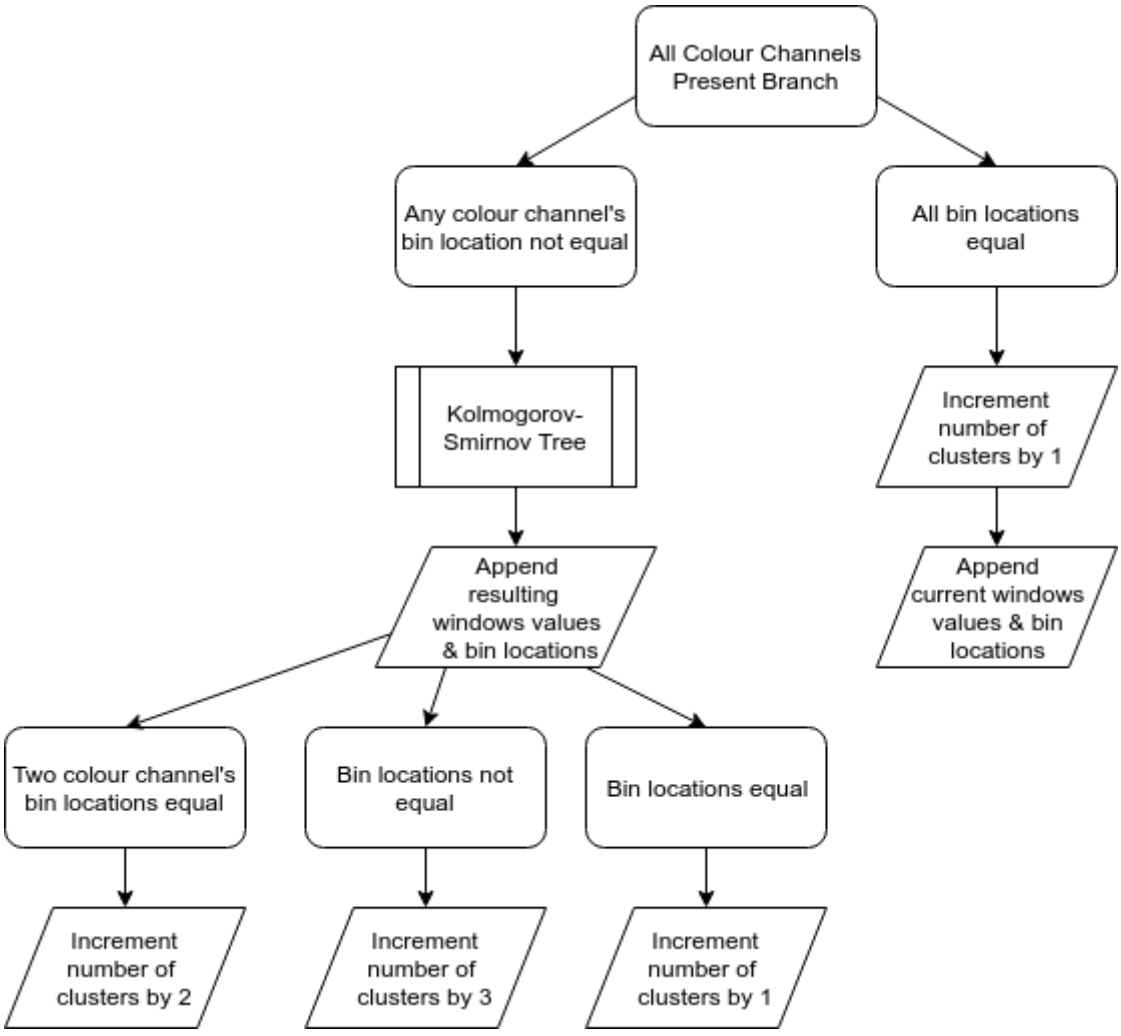


Figure 26 All colour channels Present Branch

The flowchart for the branches where at least one colour channel is present, is presented below.

Figure 27 A Colour Channel Present Branch (showcasing Blue colour channel)

The above flowchart describes the process for the case when the algorithm determines that the blue colour channel is present. The exact same structure is utilized for the two remaining colour channels.

The flowcharts for the Kolmogorov-Smirnov Tree found at the branch where all colour channels are present is as such.

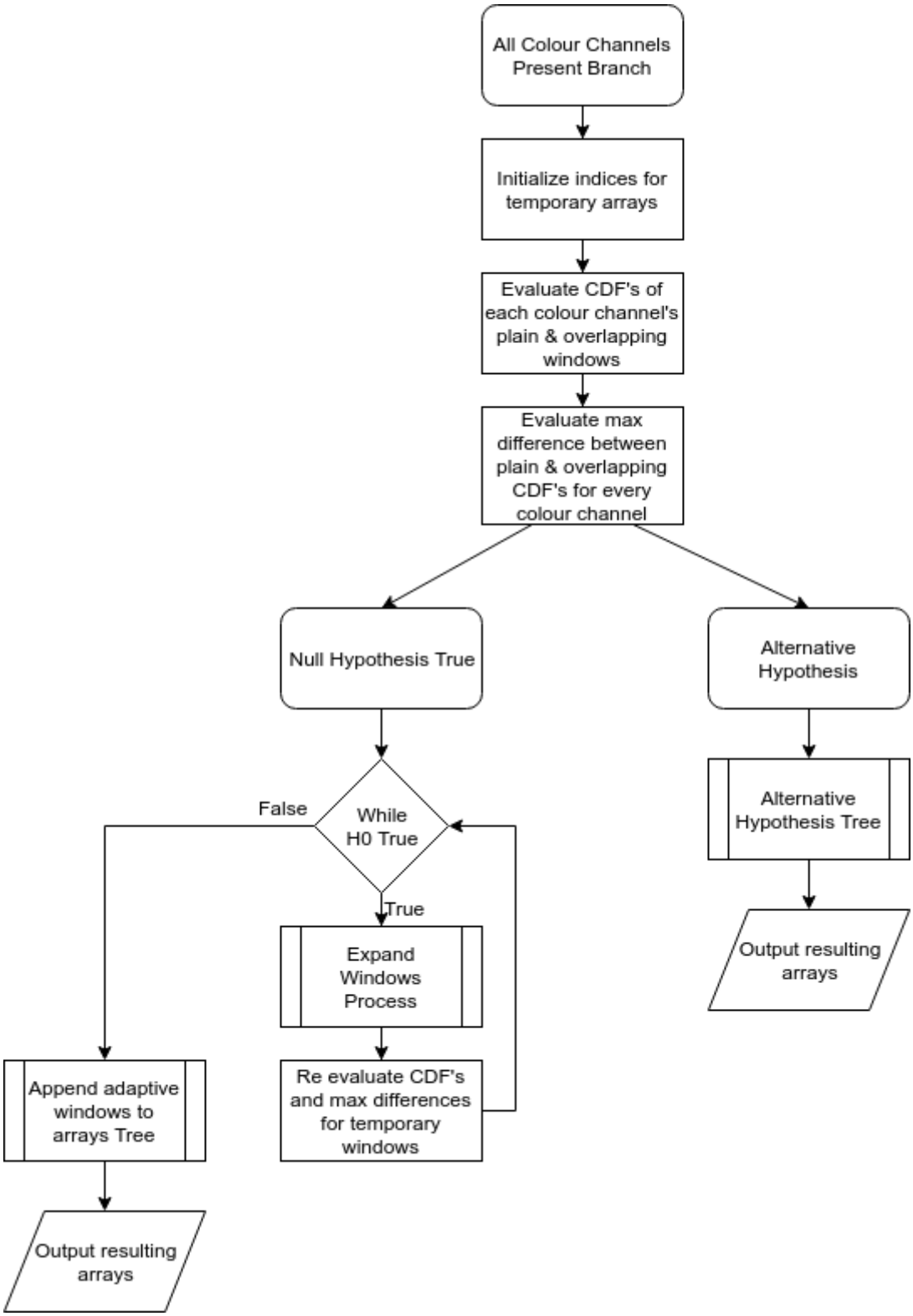


Figure 28 Kolmogorov-Smirnov Tree - All colour channels present Branch

The flowchart of the alternative hypothesis branch is as such.

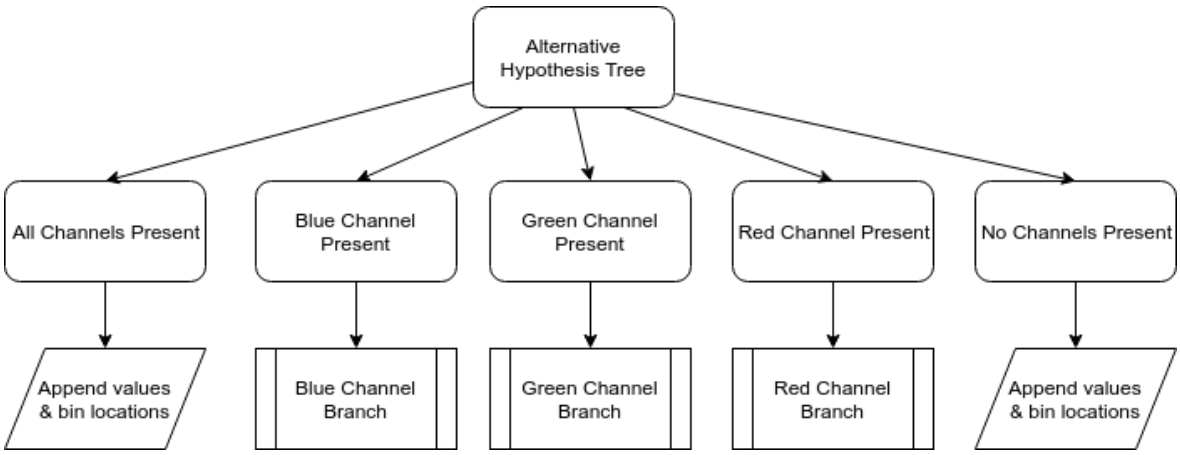


Figure 29 Alternative Hypothesis Branch

The branch of blue channel present for the alternative hypothesis branch is as shown below.

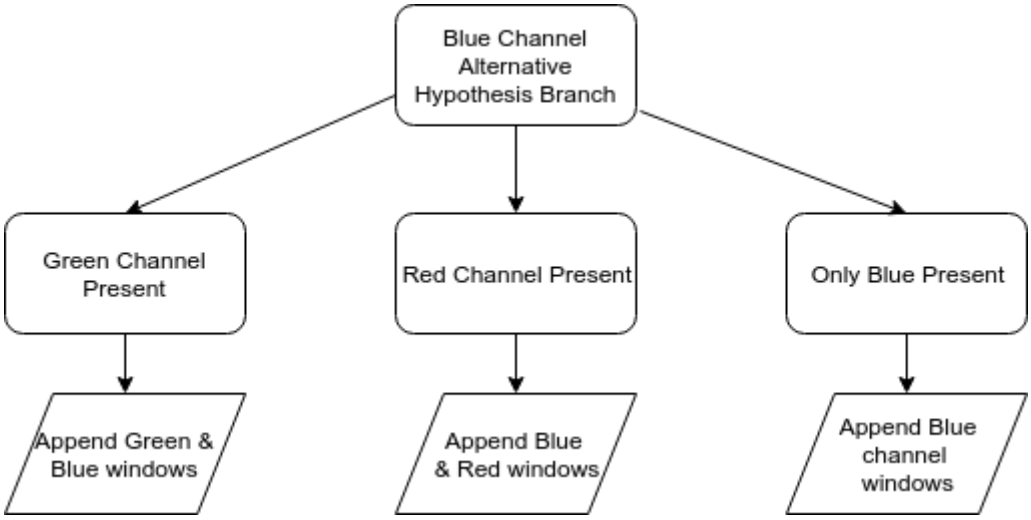


Figure 30 Blue Channel Present Branch of Alternative Hypothesis

The exact structure and operations occur for the rest of the colour channel branches.

The flowchart of the expand windows is as follows.

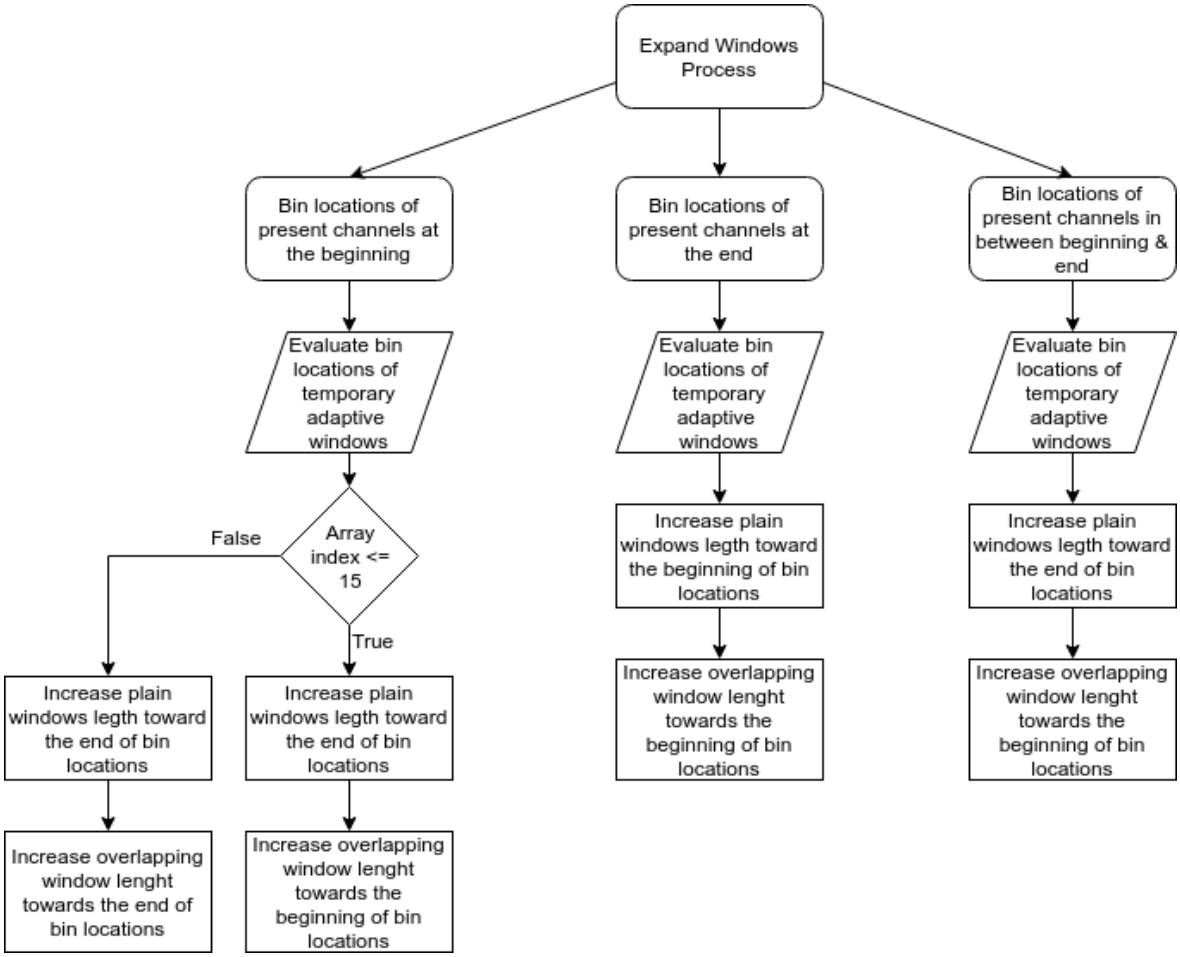


Figure 31 Expand windows process

The expand windows process is common for all colour channels present branches.

The append window values process is as follows.

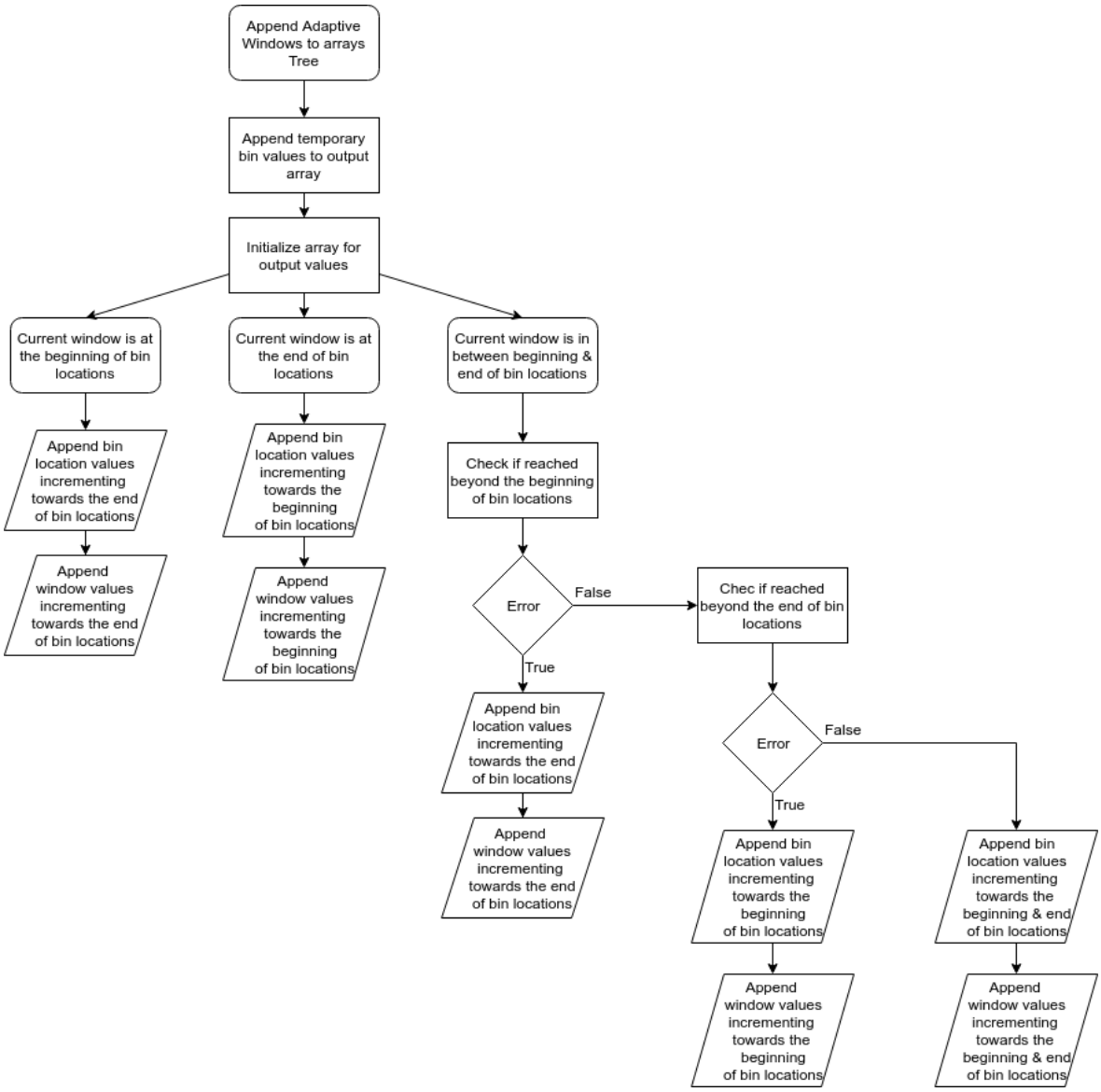


Figure 32 Append windows values Process

The process is common for every branch of colour channel present tree.

The flowchart for the null hypothesis branch of any colour channel, besides when all colour channels are present is as follows.

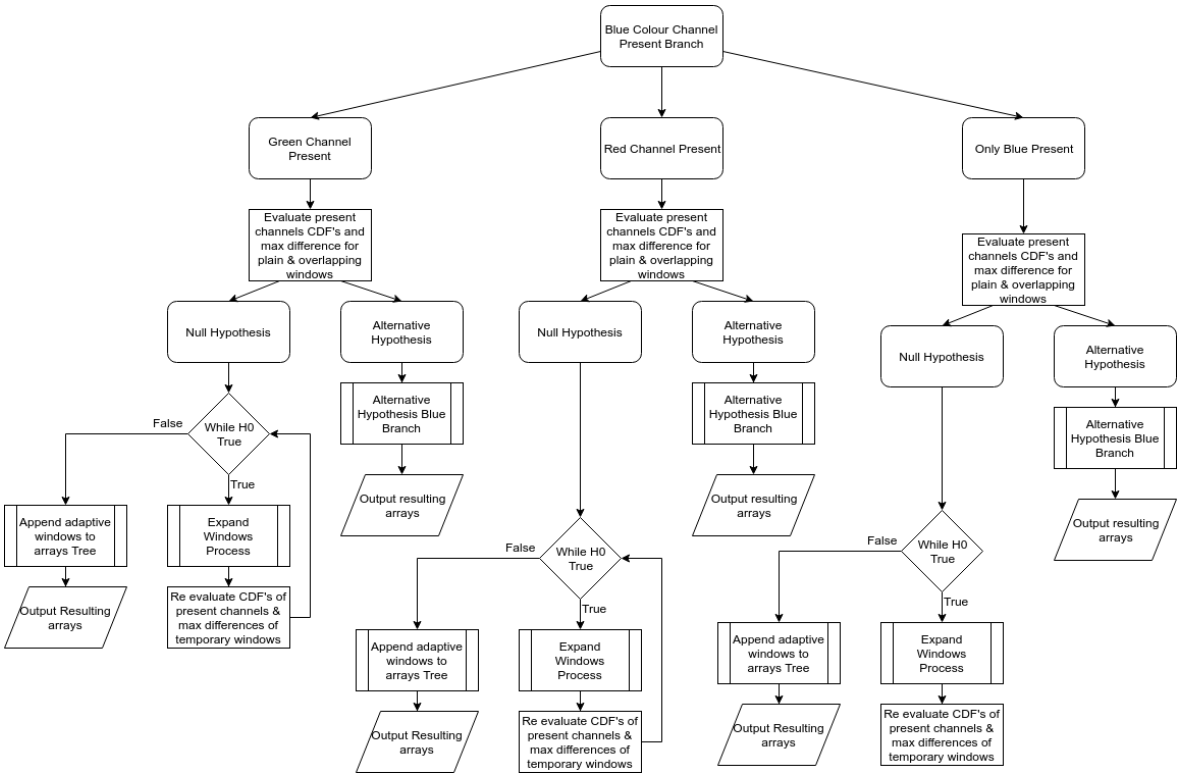


Figure 33 Null hypothesis branch for any single colour channel present

The structure and operations of this branch are common to every, single present colour channel.

Since there is no useful information on the Red colour channel, it is already known that the KS test will evaluate to the alternative hypothesis in this case.

The resulting adaptive windows for sample frame blue mug are as follows.

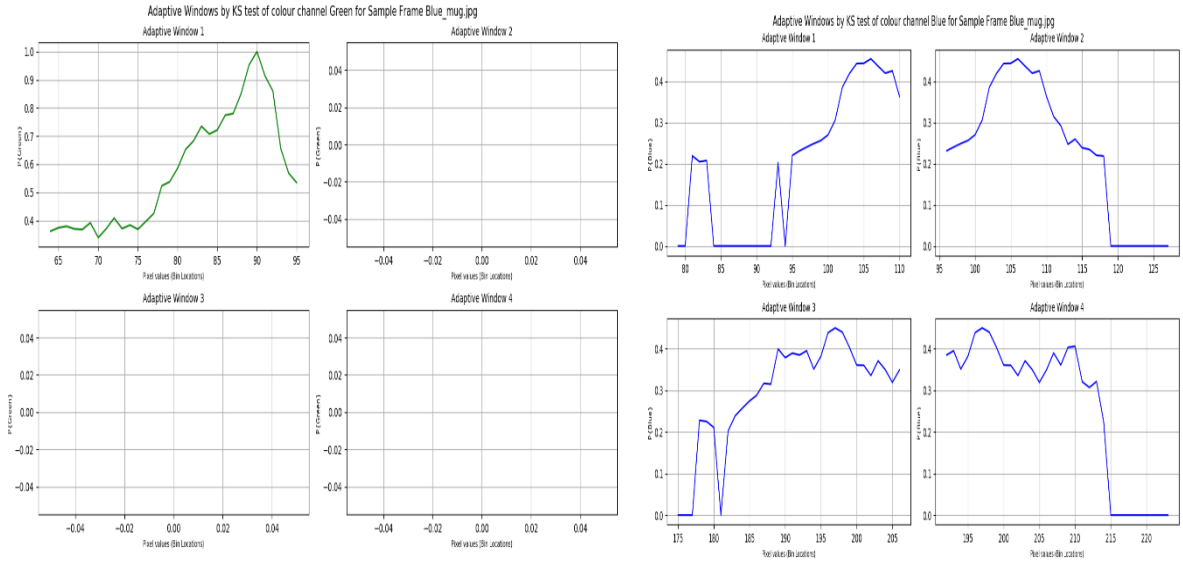


Figure 34 Adaptive Windows of Interest for Blue mug Sample Frame

4.3 Third Experiment – Blue and Red mugs

While no further improvements were made in the methodologies described in the previous experiment, the following subchapters serve to confirm that the results drawn are reliable.

The tested frame was capture via the same methods, previously described.

4.3.1 Histogram Operations

As discussed in the previous subchapter, the same histogram operations occur. However, since it was proven that the spatial filter produces suboptimal results, the band pass FIR filter is chosen for implementation.

The results of the plain histograms of the blue and red mugs testing frame are as shown below.

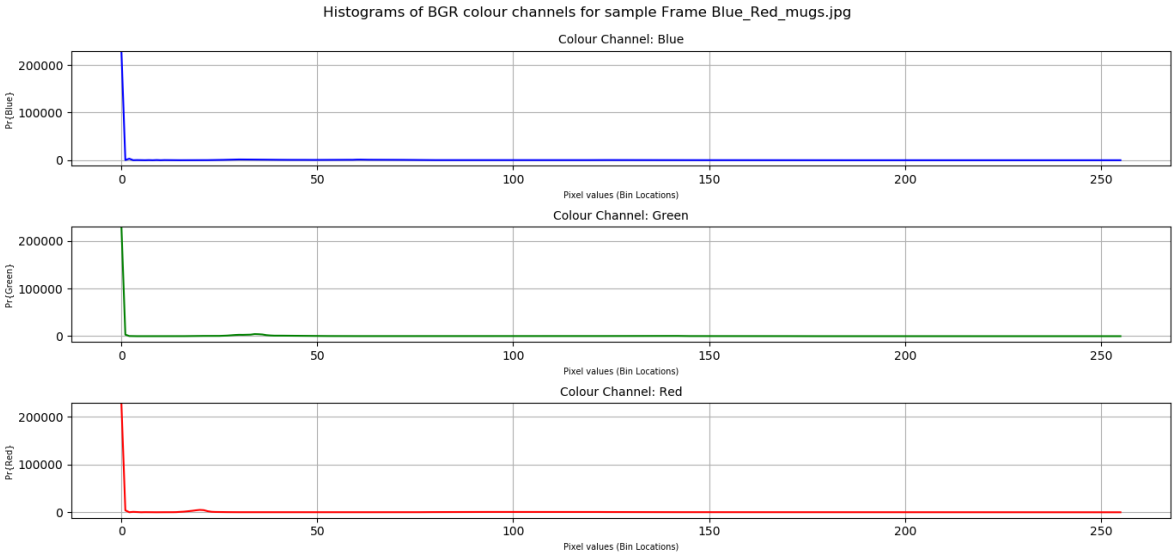


Figure 35Plain Histograms for Blue and Red mugs sample frame

The FFT of the unfiltered histograms is shown below.

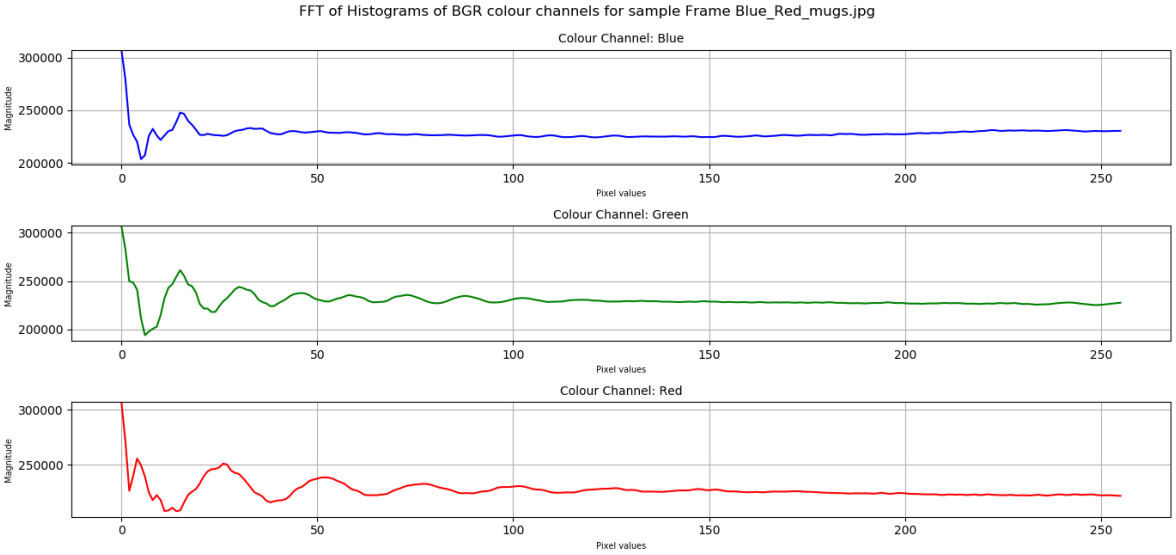


Figure 36 FFT of unfiltered histograms for Blue and Red mugs sample frame

The filtered histograms of the sample frame Blue and Red mugs is shown below.

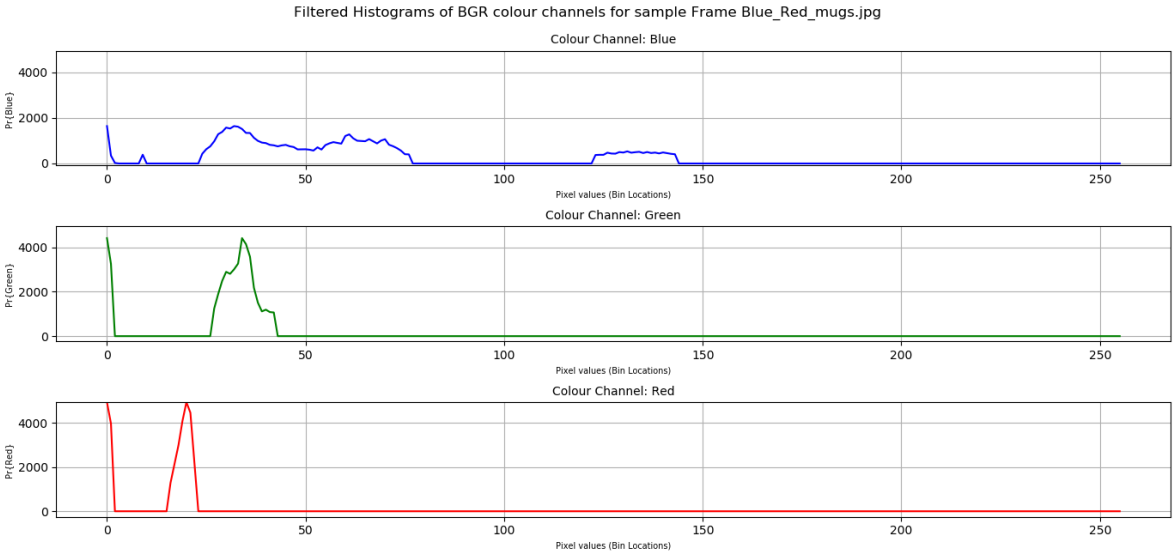


Figure 37 Filtered Histograms of Blue and Red mugs sample frame

While the filtered histogram clearly shows three distinctive Gaussian kernels at each colour channel, indicating that the kernels correspond with the objects of interest.

The FFT of the filtered histogram is as shown below.

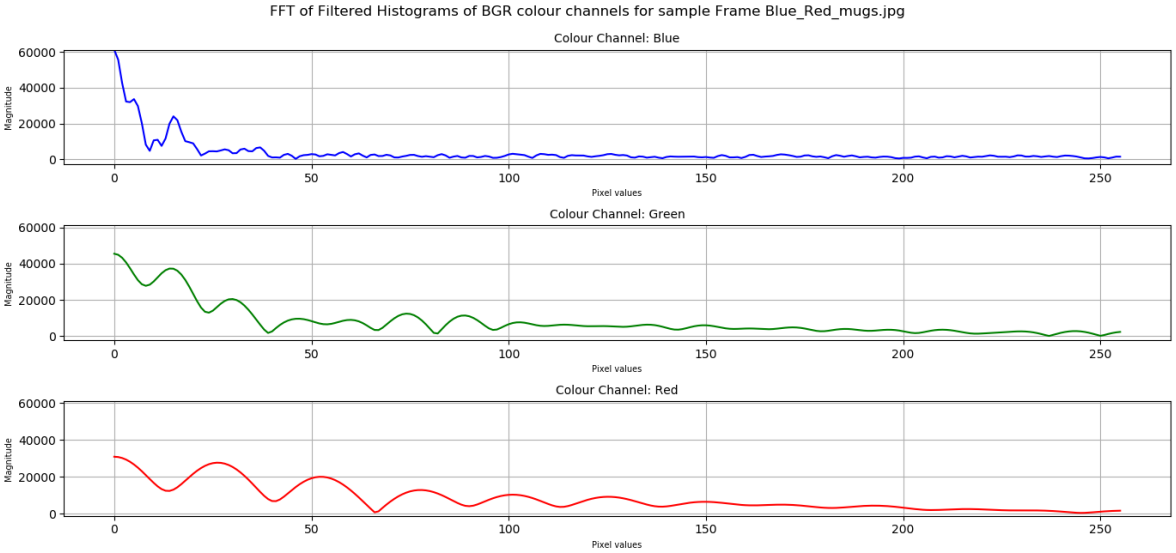


Figure 38 FFT of filtered histograms for Blue and Red mugs sample frame

The results of the FFT confirm that the filtering had the desired effect on the histograms.

The results of the normalized histograms are as such.

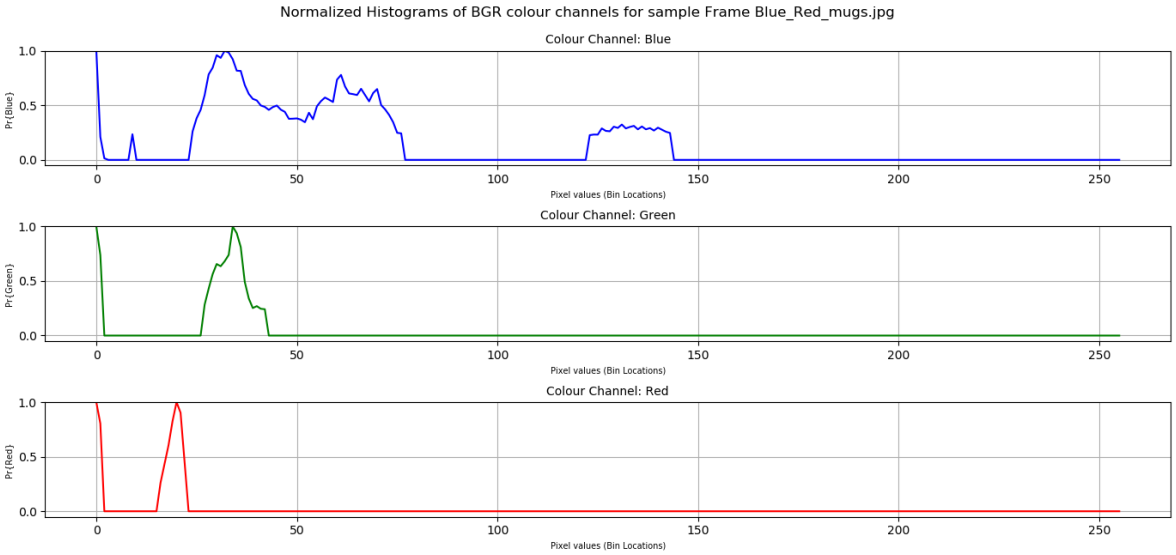


Figure 39 Normalized histograms for Blue and Red mugs sample frame

The plain windows of the sample frame of Blue and Red mugs are:

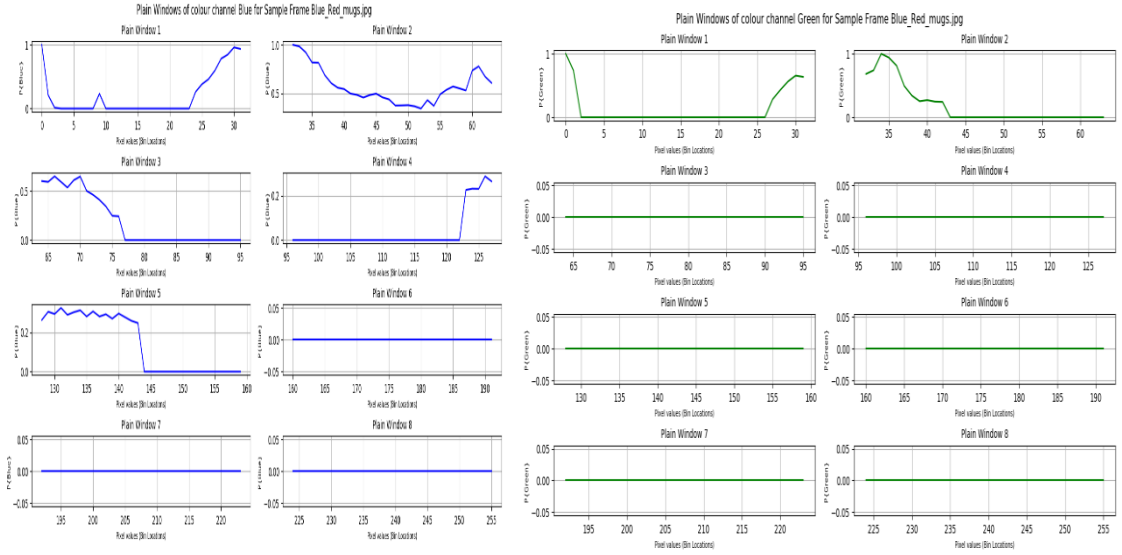


Figure 40 Blue and Green Plain Windows for Blue and Red mugs sample frame

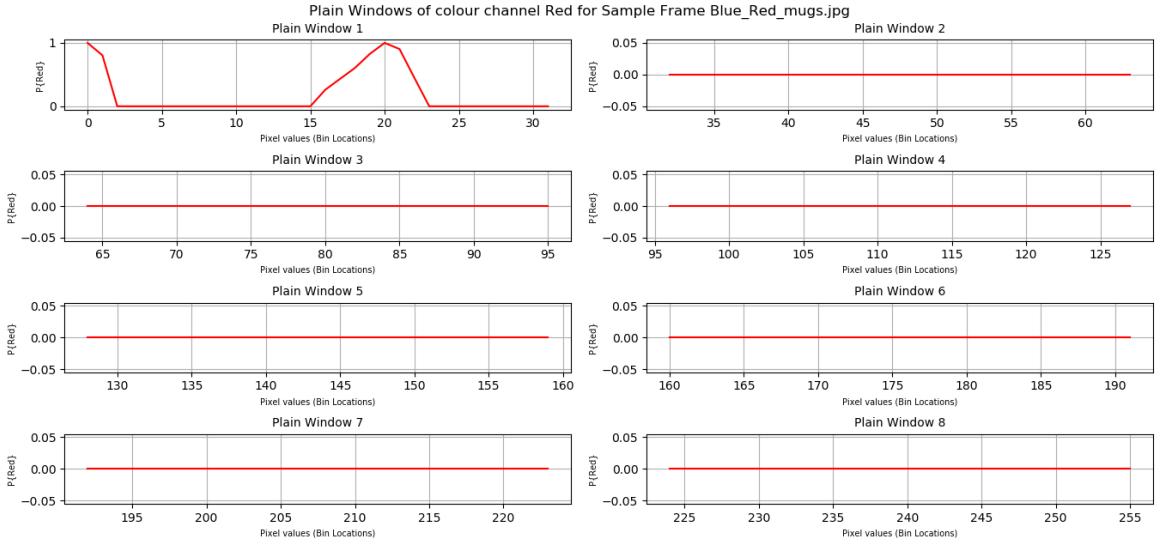


Figure 41 Red colour channel's Plain Windows for Blue and Red mugs sample frame

The results of the overlapping windows are shown below.

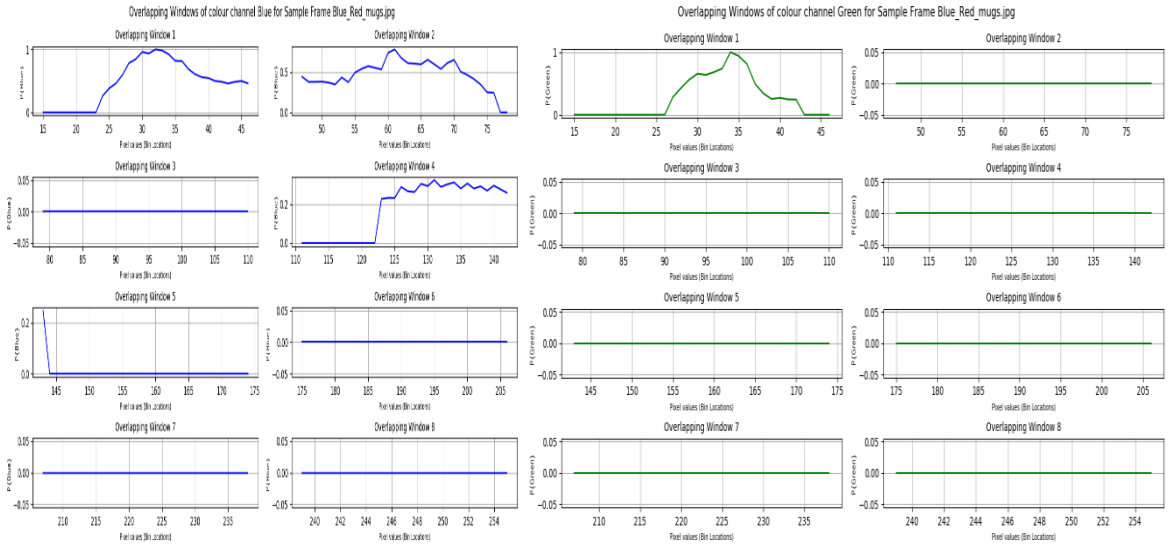


Figure 42 Blue and Green colour channel's Overlapping windows for Blue and Red sample frame

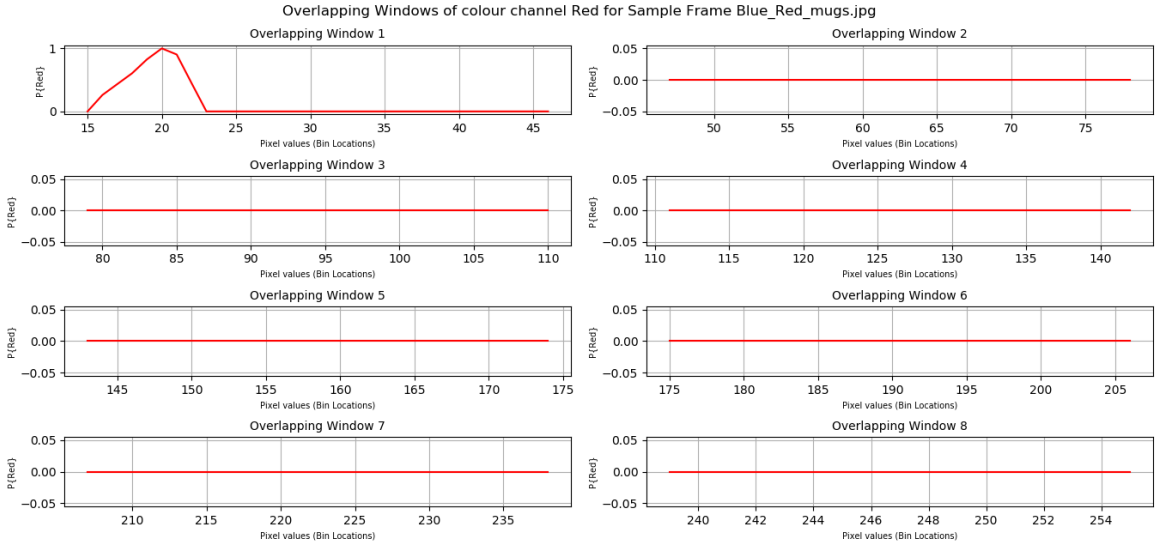


Figure 43 Red colour channel's Overlapping Windows for Blue and Red sample frame

4.3.2 Windows of Interest

Based on the previously described algorithm, that performs a general to specific search for windows of interest, its results are shown below.

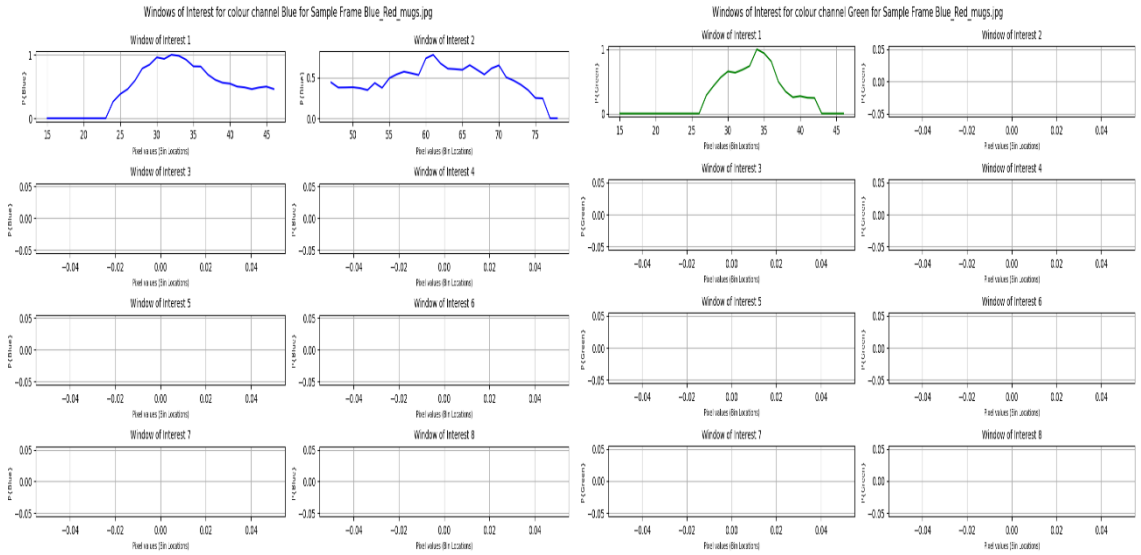


Figure 44 Blue and Green colour channel's Windows of Interest for Blue and Red sample frames

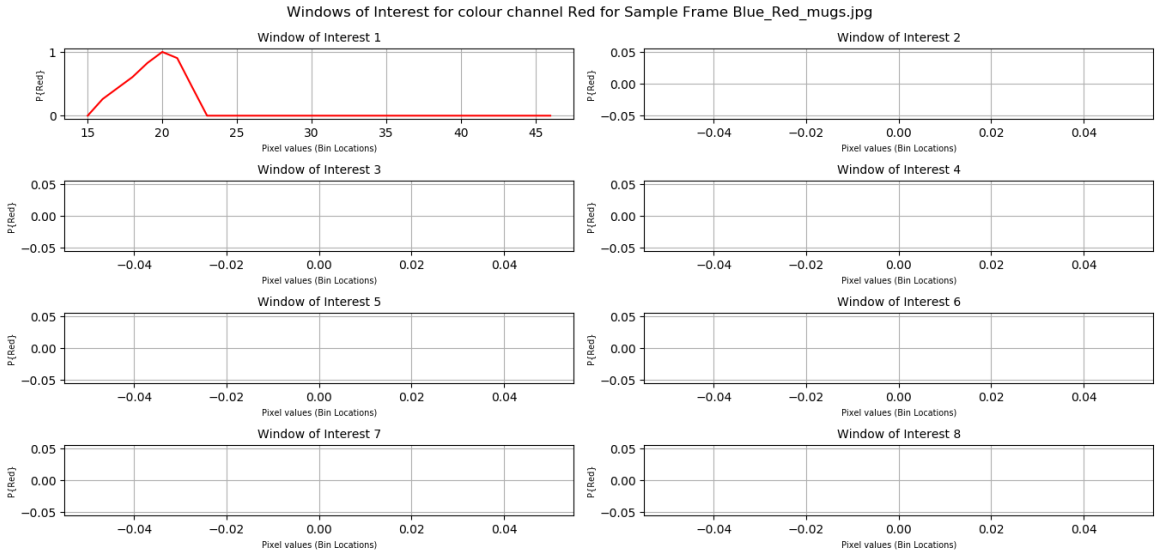


Figure 45 Red colour channel's Windows of Interest for Blue and Red mugs sample frame

4.3.3 Adaptive Windows of Interest

The resulting adaptive windows of interest for the sample frame Blue and Red mugs are as shown.

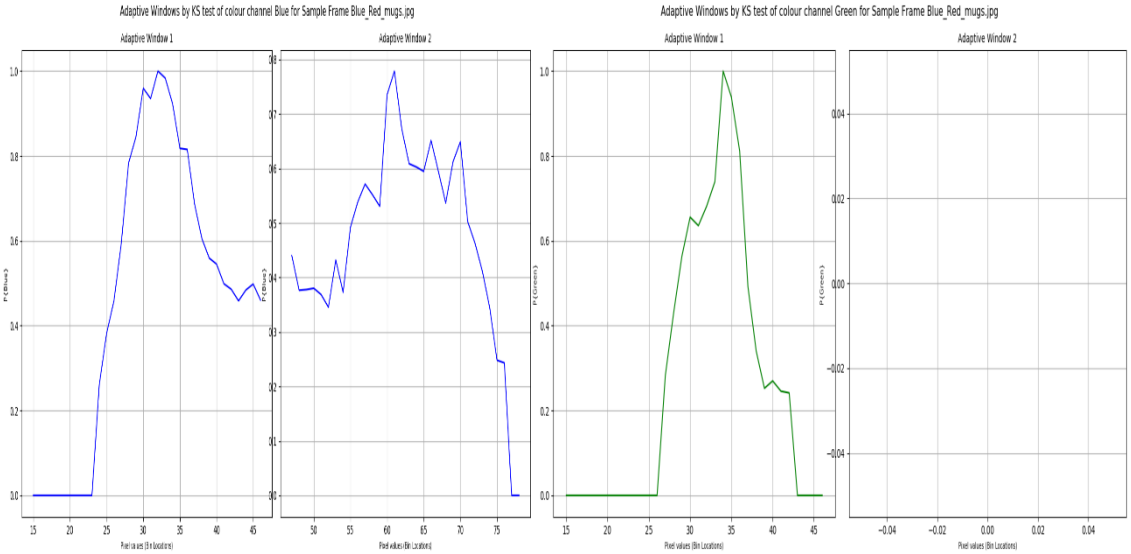


Figure 46 Adaptive Windows of Interest for Blue and Red mug sample frame

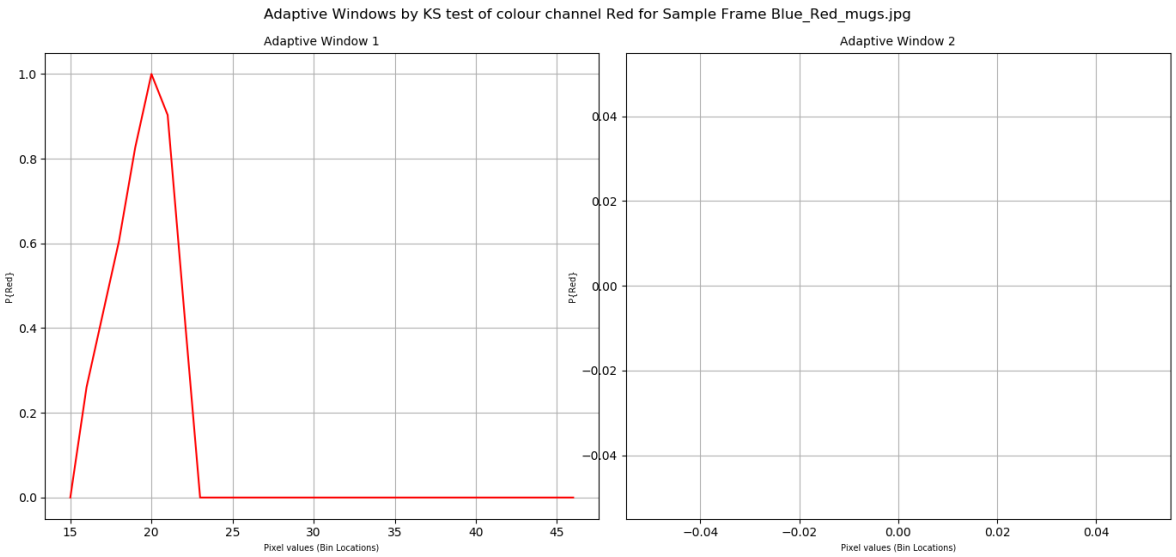


Figure 47 Red colour channel's Adaptive Windows of Interest for Blue and Red mug Sample frames

4.3.4 Cluster Estimation

This sample frame also serves as an indication of the need to segment detected objects that appear, for the algorithm's perspective as a single connected object. Further cluttering may occur if the shape and/or colour of the objects appears to be the same.

A cluster estimator was implementing by a decision three that incremented the number of clusters based on if the resulting adaptive windows of interest had the same bin values. Should the bin locations be equal, the number of clusters would be incremented by 1. Should all bin locations be unequal, the number of clusters would increase by three, and finally, should any two pairs of bin locations be equal, the number of clusters would increment by 2.

The results of the cluster estimation for the Blue and Red mugs sample frame is 4 clusters.

4.3.5 K-Means Clustering

For the segmentation of the detected objects, the k-means clustering algorithm was implemented since it is capable of performing unsupervised learning over a given set of data that no assumptions, other than the number of clusters present is made []. The object's shape or colour does not play a significant role on the algorithm's decision, since it evaluates distances of points [21].

For this project, the already made k-means cluster algorithm found in the OpenCV library was implemented, since its structure is optimized for image processing applications [3].

The mathematical steps for the k-means algorithm are as follows [21].

1. Initialize cluster centroids at random
2. Repeat until convergence:
3. For every i, set $c^i = arg\min_i ||x^{(i)} - \mu_j||^2$
4. For each j, set $\mu_j = \sum_{n=i}^{m} 1\{c^{(i)} = j\}x^i / \sum_{i=1}^{m} 1\{c^{(i)} = j\}$

For the implemented algorithm, the kmeans++ initialization was selected. The plain algorithm's initialization selected initial centres at random. The kmeans++ implementation, initializes the first centre at random, and the next centre at the most distant centre available [1].

The resulting segmentation based on the cluster estimation's response is as follows.

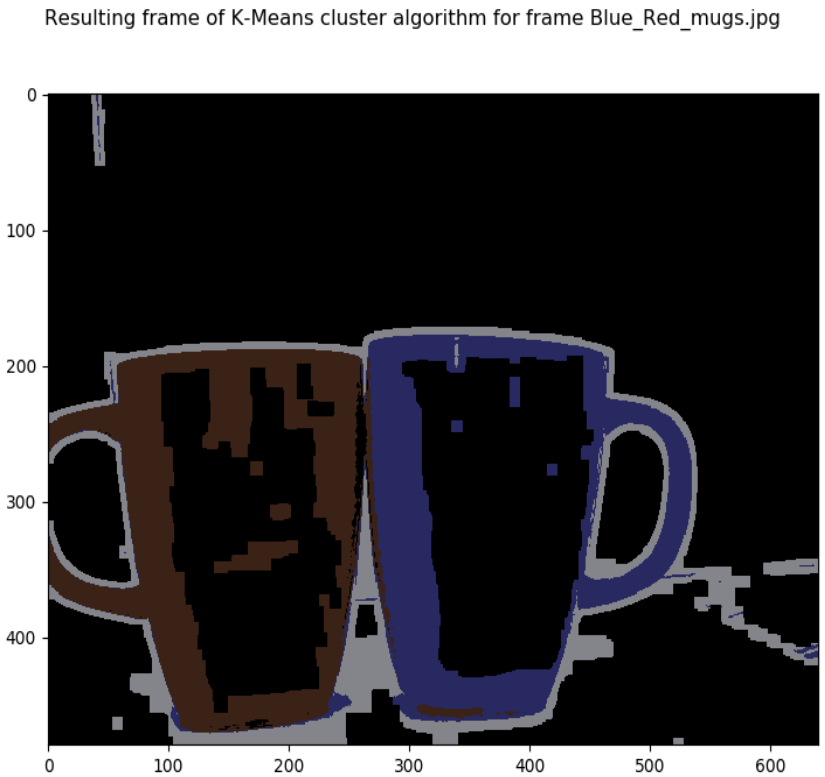


Figure 48 Resulting segmentation of the K-Means clustering algorithm for Blue and Red mug sample frame

Chapter 5: Results – Analysis, Testing, and Evaluation

In order to determine whether the algorithm has optimally captured the objects of interest, two statistical test implementations have been employed. The one being the evaluation of the deviation found in the adaptive windows of interest, and the other being the implementation of the KS test in the current and past frame, to determine whether the histogram shapes have been consistent, or further subtraction is required.

5.1 MoG2 Convergence (Detection of Objects of Interest)

The algorithmic implementation of the deviation test performs the following operations. The algorithm iterates over each resulting adaptive window of interest and evaluates its standard deviation. Should the windows values be non-empty, by evaluating its mean value, its deviation result is being compared to a threshold value. In the case where the deviation value is greater than or equal to the threshold's value, an index, indicating the number of windows with deviation value of interest, is incremented by one. Should the deviation value be lower than the threshold's value or the window contains no values, an index indicating the rejected windows is incremented by one. After the end of the iteration, the actual present values are being evaluated by calculating (h*3) - n_{reject}, where h denotes the number of windows, 3 indicates the number of colour channels and n_{reject} the number of rejected windows based on the previous threshold operation. The final step of the algorithm is to evaluate if the number of accepted windows is equal to or greater than the median value of the present channels. Should the number of accepted channels fulfil that criteria, the algorithm responds with "True", indicating that the MoG2 algorithm has converged to the object(s) of interest.

Should it fail the criteria, the algorithm responds with “False”, indicating that further subtraction is needed.

The implementation of the KS test for evaluation of the MoG2’s results, tests the current frame’s histogram shape with the 10th previous frame that the algorithm temporarily stores and updates. Should the null hypothesis be true, the algorithm returns “True”, indicating that the MoG2 algorithm has converged to the objects of interest. Should the alternative hypothesis be true, the algorithm responds with “False”, indicating that the MoG2 has not converged sufficiently.

A flowchart of the KS test for the MoG2 convergence is presented below.

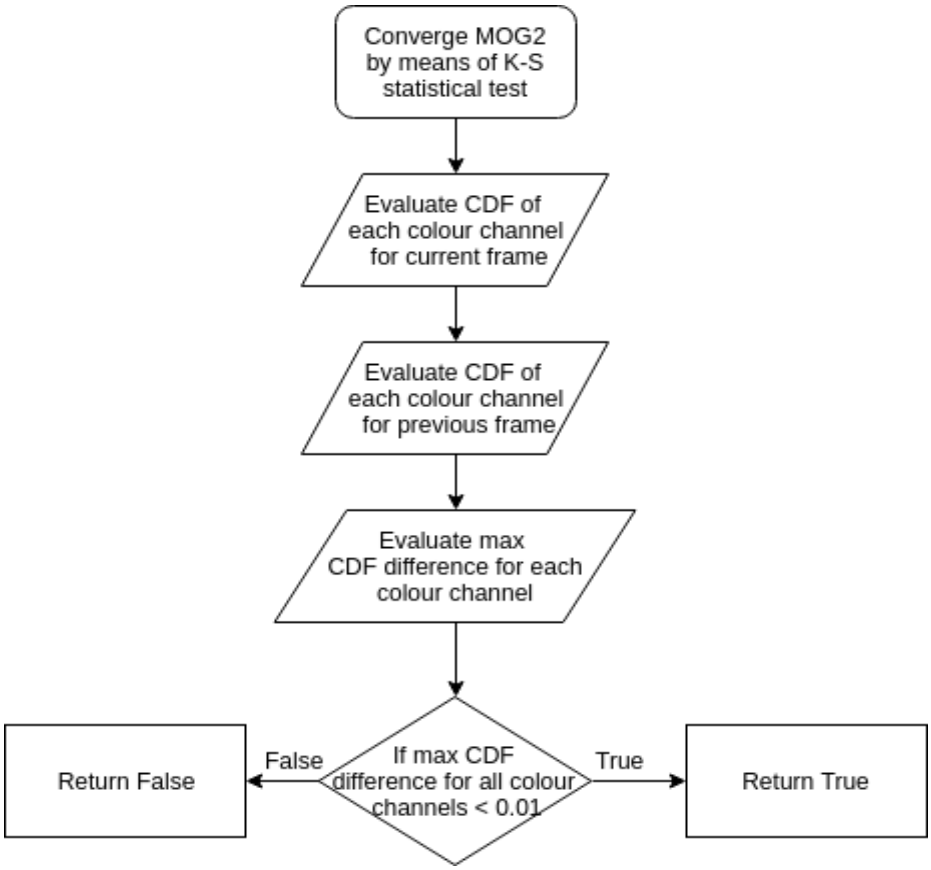


Figure 49 KS test for the MoG2's convergence status

Based on what has been presented in the previous chapter and this subchapter, the detailed methodology of this implementation is as follows.

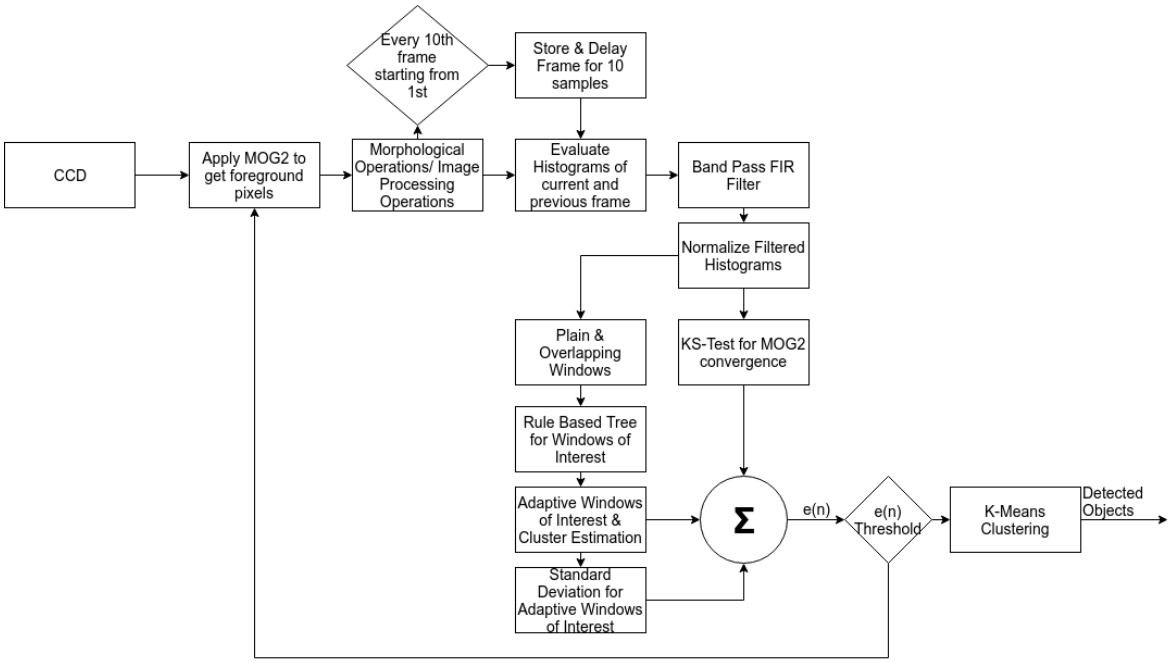


Figure 50 Block diagram of implemented methodology

The current frame of the system is fed into the MoG2 algorithm, to subtract the background. Image processing applications that filter the frame and capture the contours of interest, as discussed in the previous chapters, are implemented. The frame is temporary stored and updated at each 10^{th} iteration. At each 10^{th} iteration the histogram of both current and past frames occurs; both histograms are filtered and normalized and fed into the KS test that determines whether the system has converged to the objects of interest. The results are also fed into the Sequential covering algorithm, presented in previous chapters, and from there, the adaptive windows of interest are evaluated, the present number of clusters is estimated, and the deviation test for the present adaptive windows is implemented. Based on the results of the KS test and the deviation test, the system decides whether it has converged, or further subtraction is needed. Should the system has converged, the estimated number of clusters are fed into the k-means clustering and the resulting segmented frames are presented.

5.2 Static White Background

The testing of the resulting methodology occurs for a variety of mugs with same and different shapes and colours. In this subchapter, the presented tests have been performed on a controlled white background. In order to obtain the results, the k-means

cluster had to be removed since, in its current implementation it slows the performance of the system significantly, therefore, only the estimated clusters are presented.

5.2.1 Two mugs results

The tests consist of two mugs being present that initially have the same shape and colour in a white background. The results are presented below.

5.2.1.1 Same colour and shape

Two blue mugs

A sample of the test sequence raw data is presented below.

Figure 51 Sample sequence of two blue mugs

Figure 52 Captured contours for two blue mugs

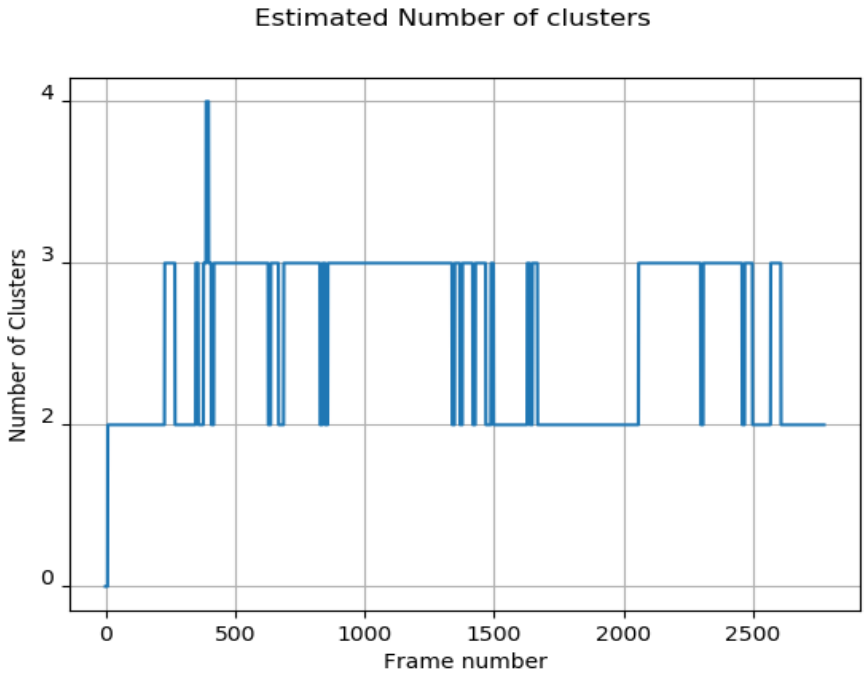


Figure 53 Number of estimated clusters over testing period for two blue mugs

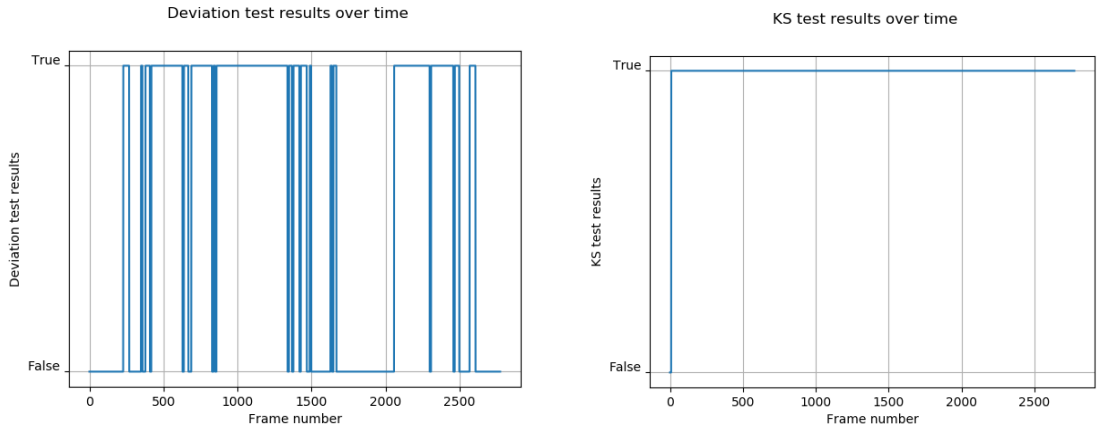


Figure 54 Deviation and KS test results over testing sequence

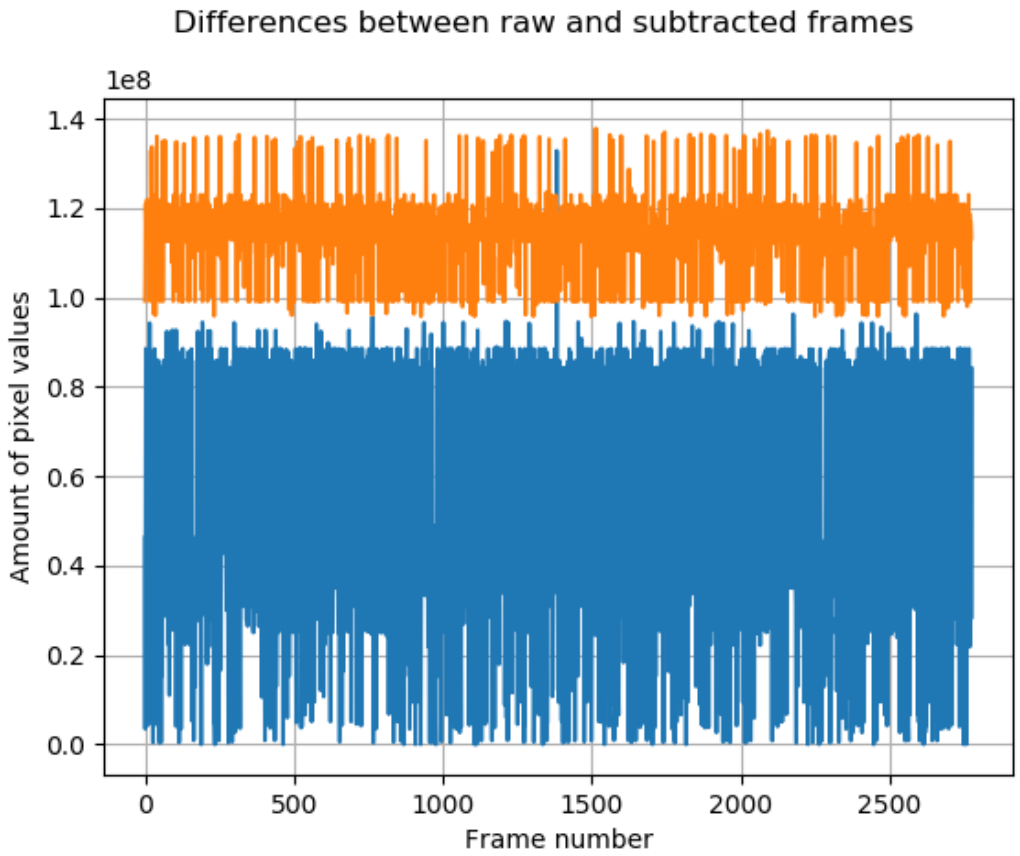


Figure 55 Total pixel differences over testing sequence

Based on the results of the testing sequence, it can be concluding that the KS test indicated that the algorithm has converged on the objects of interest. However, the deviation test indicates that the MoG2 has not yet converged on the objects of interest. The cluster estimator varied over the values of 2 and 3 clusters present; it is hypothesized that it corresponds to the actual present clusters since the variation can be explained by the variation of the illumination.

Two red mugs

Sample raw frames of the test sequence are presented below.

Figure 56 Raw frames of test sequence for two red mugs

Figure 57 Sample captured contours for test sequence two red mugs

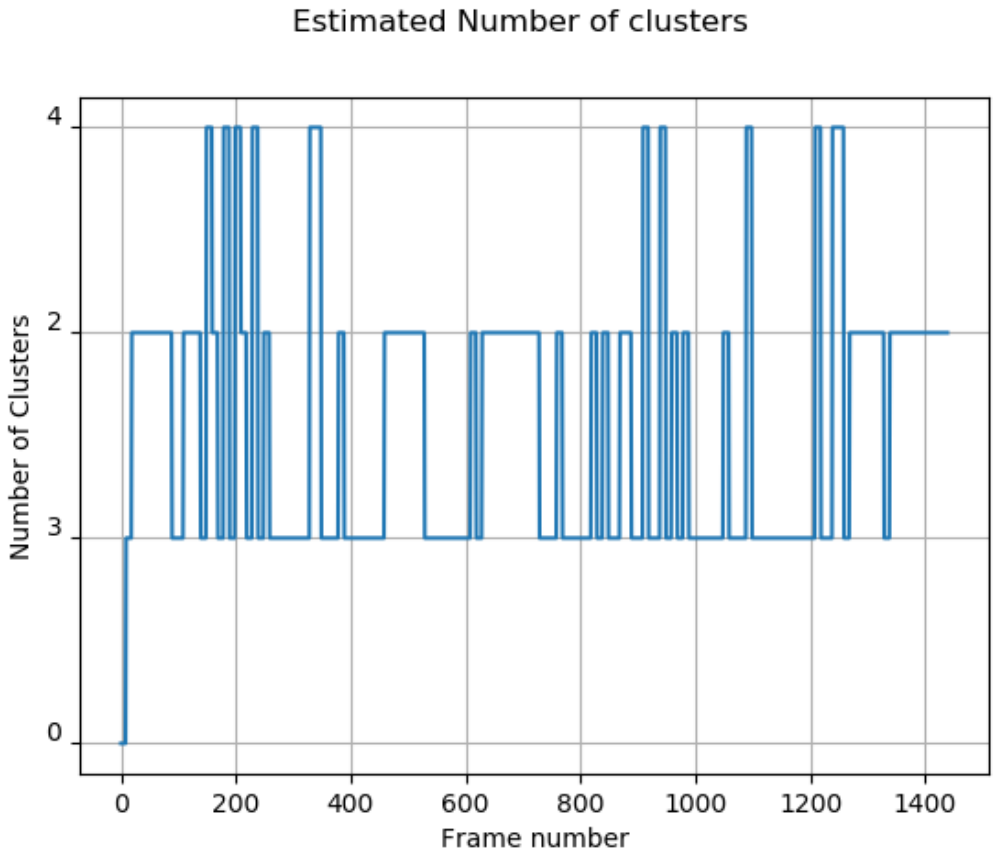


Figure 58 Estimated number of clusters for test sequence two red mugs

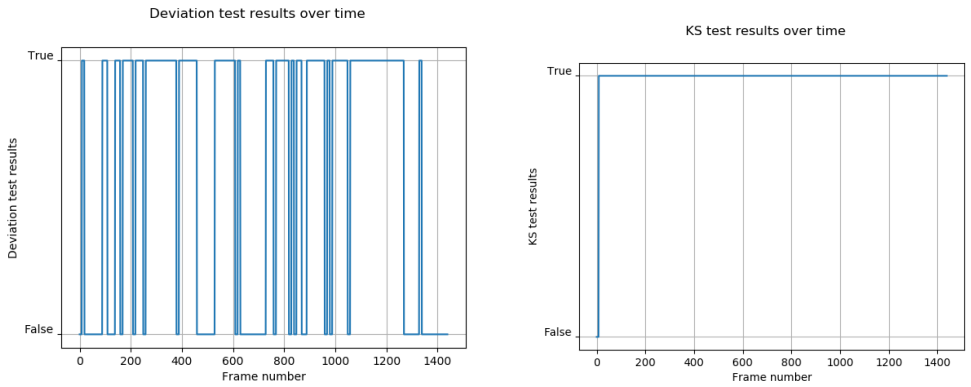


Figure 59 Deviation and KS test results over testing sequence of two red mugs

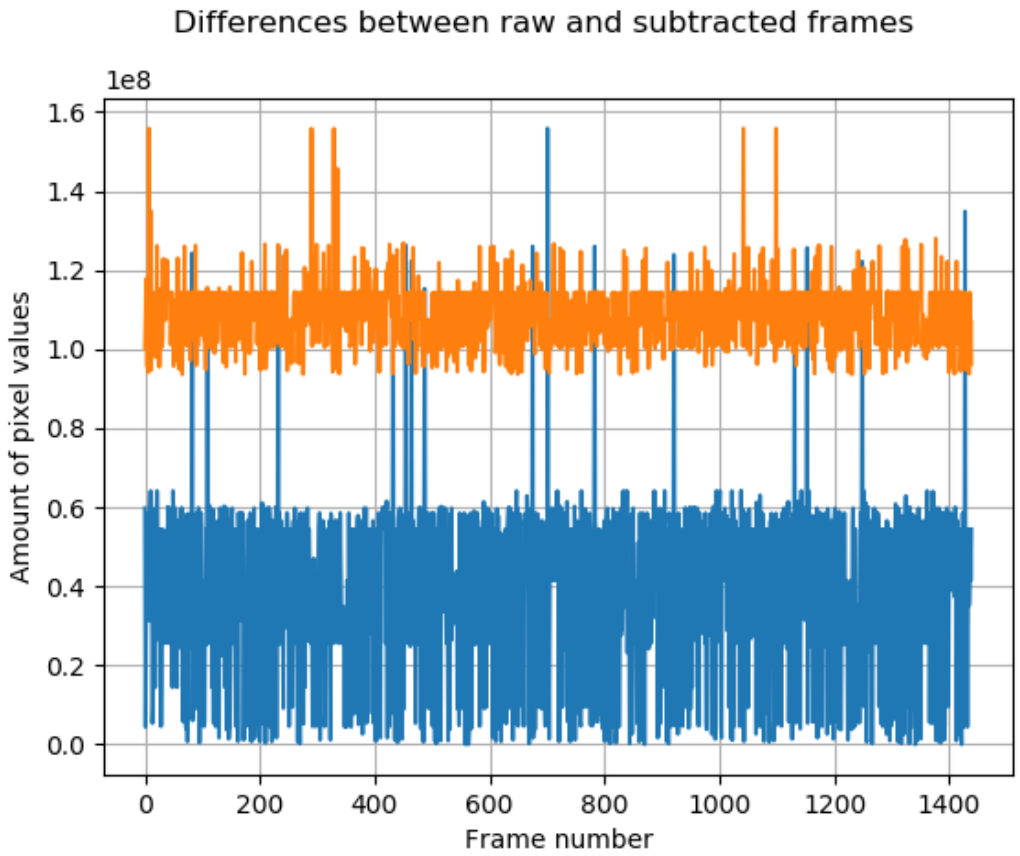


Figure 60 Differences between total pixel values of each frame for test sequence two red mugs

It can be concluded from the evaluated difference of raw frames and contours that in some small cases the algorithm has failed to completely capture the contours of interest. The previous conclusions on the deviation and KS test can be also drawn; a pattern appears to emerge over the results of these tests. The estimated clusters in this test varied in the same range of values as the previous test sequence.

Two white mugs

Figure 61 Raw sample sequence for two white mugs test

Figure 62 Sample captured contours for test sequence two white mugs

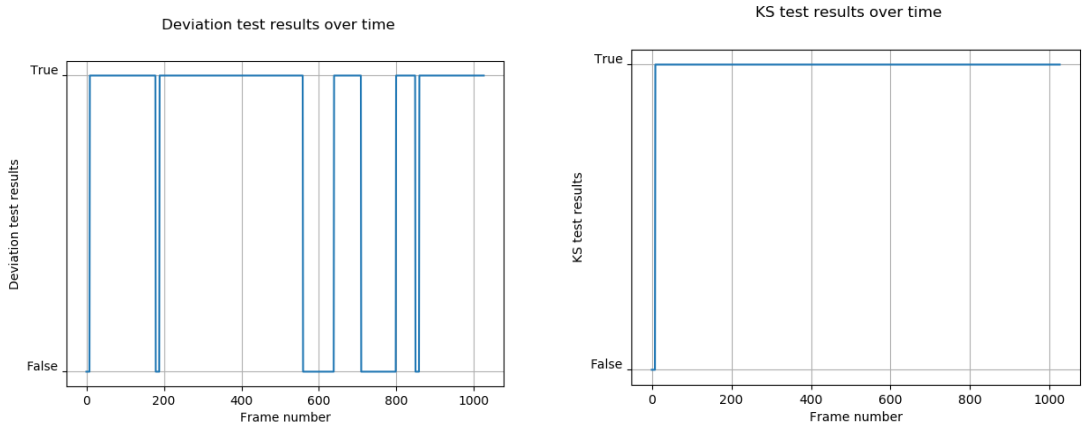


Figure 63 Deviation and KS test results of test sequence two white mugs

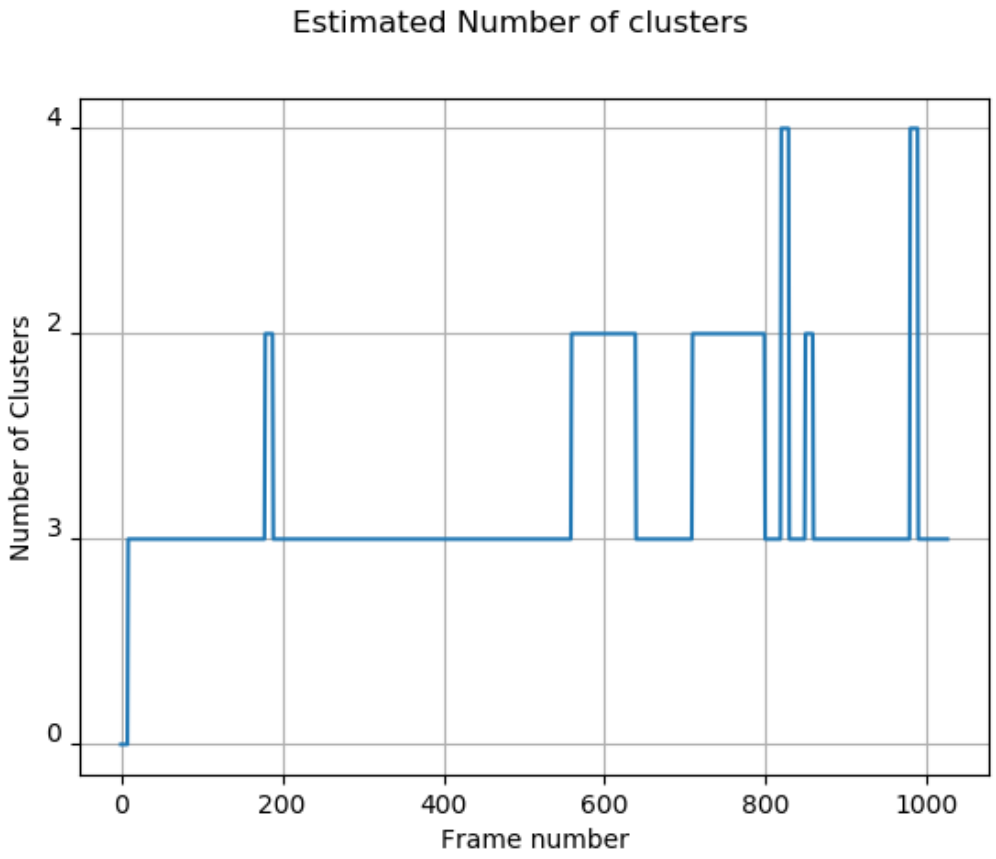


Figure 64 Estimated number of clusters over testing sequence of two white mugs

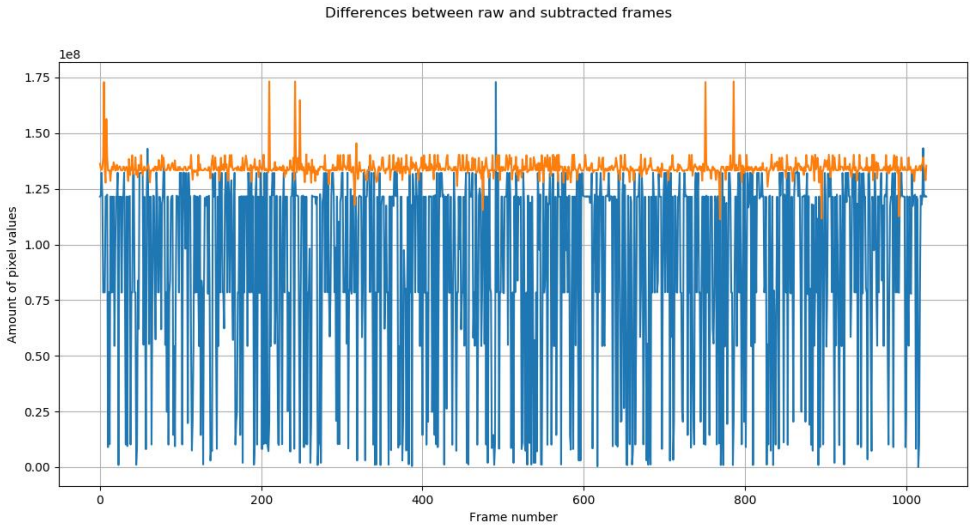


Figure 65 Difference of total pixels for test sequence two white mugs

Based on the results of the difference between the captured contours and raw frames, it can be concluded that the system could not optimally capture the contours of interest for two white mugs. It is hypothesized that this is due to the similarity of colour between the objects and the background. The KS test results hints that this statistical metric is not the optimal choice for a static white background whereas the deviation test appears to be more suitable for this application.

Same colour different shape

Figure 66 Sample raw frames for two white mugs with different shape

Figure 67 Sample frames for captured contours for test sequence of two white mugs with different shape

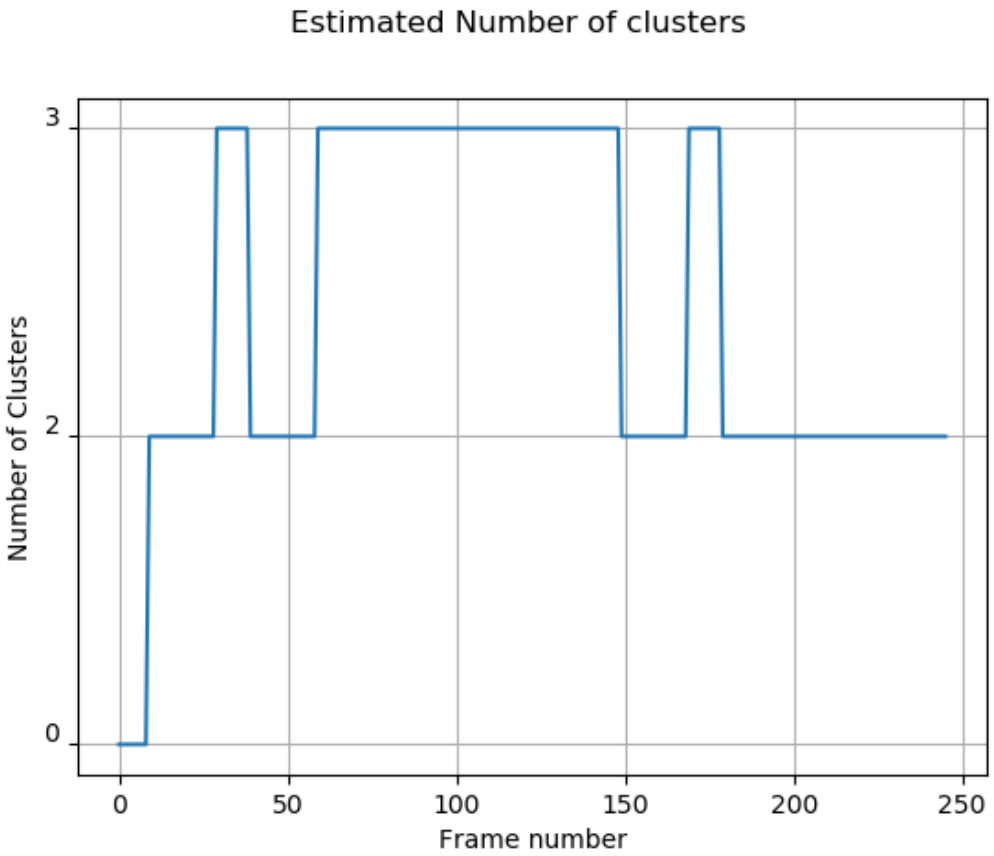


Figure 68 Estimated number of clusters for test sequence two white mugs different shape

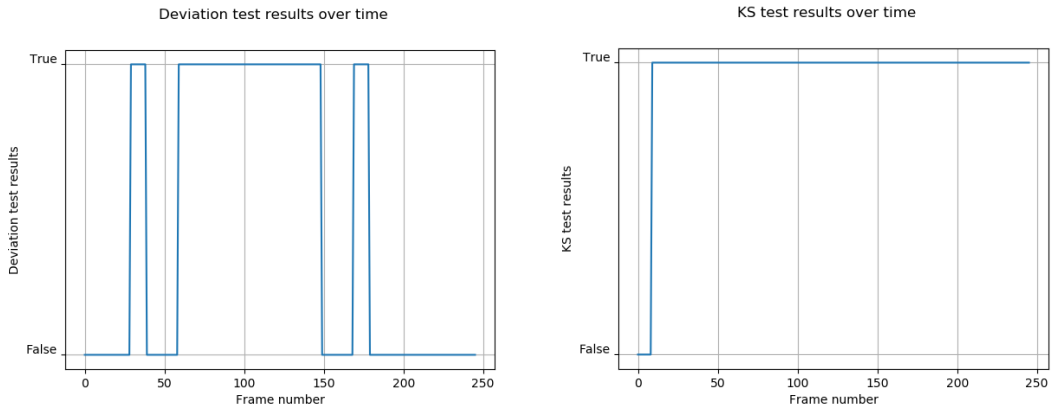


Figure 69 KS and Deviation test results for test sequence two white mugs different shapes

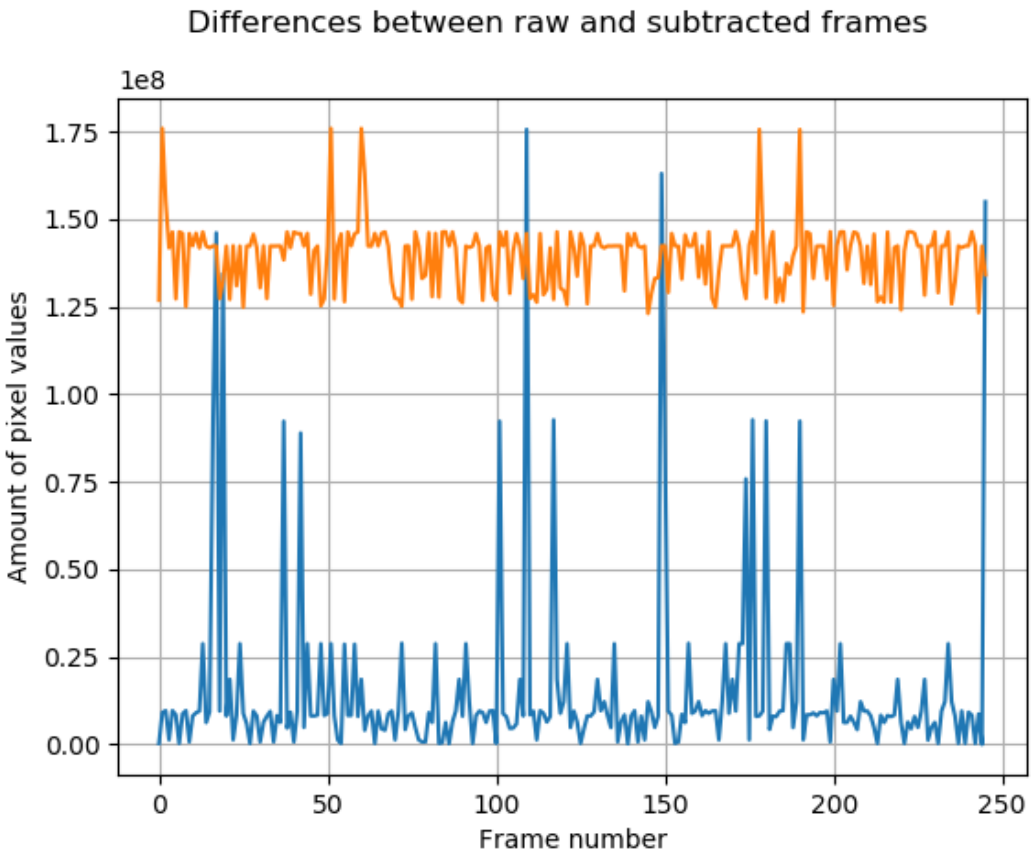


Figure 70 Difference between total pixel values of raw and contour frames for test sequence two white mugs different shape

Blue and red mugs with different shape

Figure 71 Sample frames for test sequence red and blue mug different shapes

Figure 72 Sample captured contours for test sequence red and blue mug with different shapes

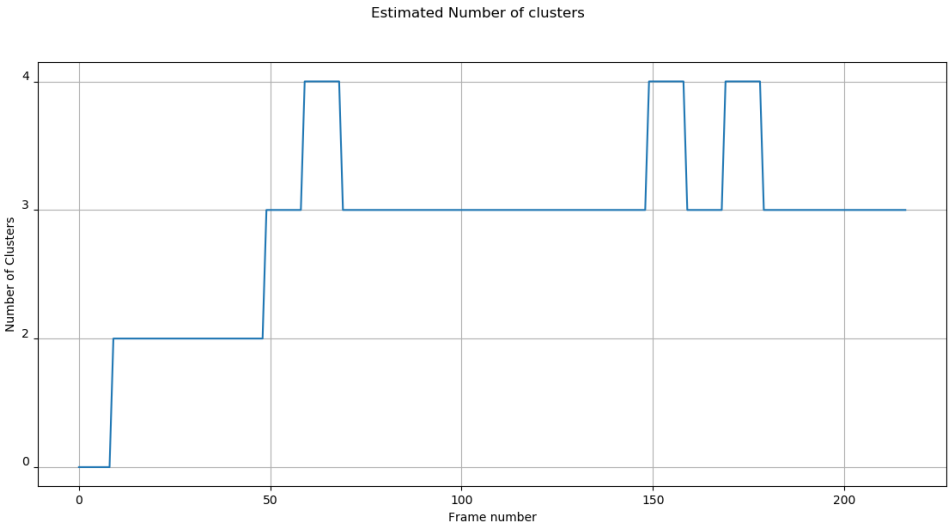


Figure 73 Estimated number of clusters

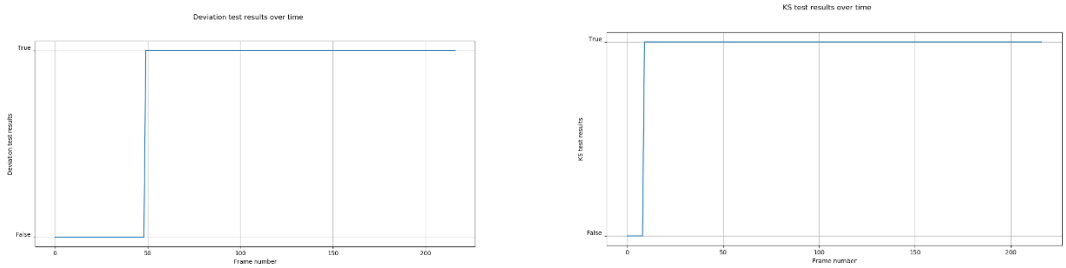


Figure 74 Deviation test and KS test for test sequence blue and red mugs with different shapes

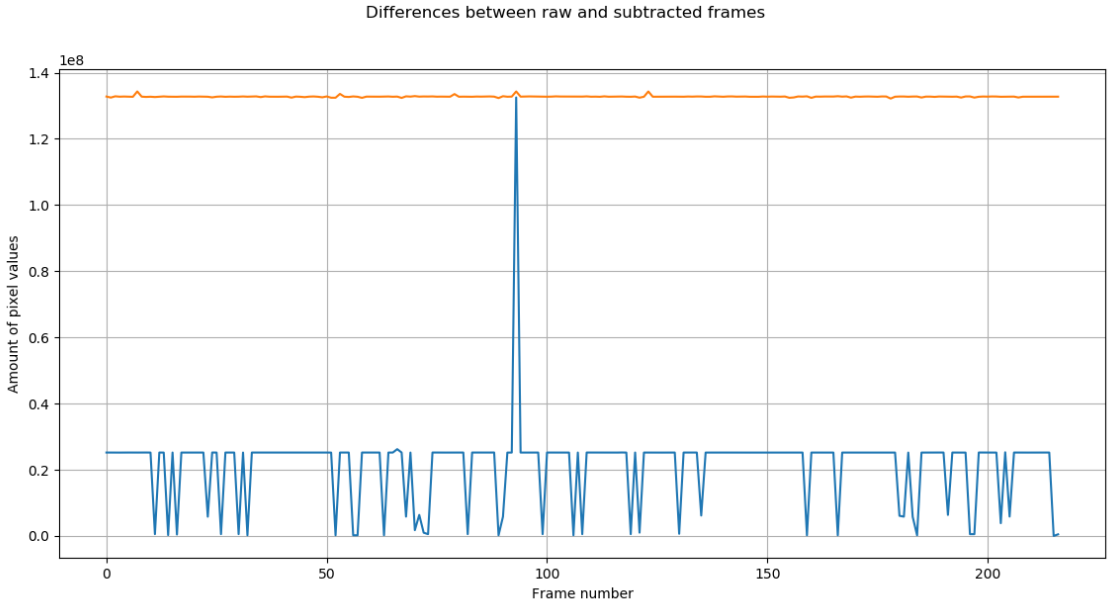


Figure 75 Differences between raw frame and captured contours of test sequence blue and red mug different shapes.

Based on the resulting differences between the raw frames and captured contours, it can be concluded that the algorithm has successfully captured the objects, occasionally subtracting it, but the algorithm's behavior has been swiftly adapted to the present objects. In this test sequence, the results of the KS test and deviation test overlapped significantly. The cluster estimation remained mostly constant, assuming that the small variation is due to illumination differences.

Chapter 6: Conclusions and Further Work

6.1 Conclusions

The implemented system is capable of performing an unsupervised identification and tracking of different shapes and sizes. The limit of the amount of object it is capable of tracking and identifying has not been reached by the tests that have been performed; it can be concluded that the implementation achieved its main goal with acceptable results. While the systems performance has not been benchmarked to give an exact percentage on its detection rate, in the case of a static background it only failed to detect the objects of interest in some occasions, namely for white coloured objects. However, it must be noted that in many occasions, should the objects remain present from the beginning of the test, the algorithm considers them part of the background, thereby subtracting them.

For the case study of static, white background, a number of household mugs of different colours and shapes were detected. In almost every test case, besides when only white mugs were present, the objects were successfully detected, with some small frames where parts of the background have not been completely removed.

Based on the tests that have been performed, it can be concluded that the major characteristics of the objects of interest that affect the system's performance is the similarity in colour.

Overall, based on the presented results, it can be concluded that the project's main aim and objectives have been achieved successfully.

System Limitations

In its current implementation, the system is susceptible to illumination changes that cause the identification of objects difficult. In some cases, some artefacts of the background were kept present, the shadow(s) of the detected object(s) was also not sufficiently removed, and in some rare cases, the implementation failed completely to keep the detected object present long enough to track it; finally, in some rare occasions, the system partially converged to the objects of interest, keeping only parts of the object present. In the case study of a white background, while the algorithm detected each test

object, the quality of the detection, besides the illumination factor, was lower when the object's colour was the same as the background's colour.

An initial, unknown time period must occur in order to fully subtract the background and retain only the objects of interest. This excludes the system for applications where the detection of already present, static objects is needed.

Should the background be cluttered (i.e. not somewhat unified colour) the cluster estimator underestimates the number of clusters that are present; another issue is the detection of false-positive artefacts. Finally, even when the background is kept static and in a unified state, the cluster estimator could provide a greater number of clusters than the actual number of clusters.

6.2 Further work

Revisions

There are several factors throughout each step of the implementation that could be improved by either further developing some of the algorithms or utilize other mathematical functions that produce more accurate results (e.g. Learn-One-Rule by means of KS-test). More specifically, while the Rule-Based Tree selects the desired windows of interest, further statistical metrics could be implemented should the conflict of both overlapping and plain windows contain the same amount of useful information. Metrics that measure the amount of information, such as entropy, could be of interest to solve such cases. As previously discussed, the Rule-Based Tree employs criteria, in its window selection, that will not produce greedy results. However, as noted by Chapter 4, in some cases it can misinterpret some windows as useful, therefore produce greedy results. By developing some pruning criteria or performing backtracking (e.g. check if values of current window are present in the previous, and values with bin locations out of range from previous window are zero), the algorithm is hypothesized to produce more accurate results. Further optimizations on the logical evaluations from serial to parallel could further increase the algorithm's computational performance.

The process of segmenting the histograms of each colour channel into windows could be entirely parallelized by utilizing the iteration indices.

The adaptive windows by means of the KS statistical test (i.e. Learn-One-Rule implementation) in its current implementation, expand each colour channel's windows

with the same metric; as explained in Chapter 3, this is a greedy behaviour of the algorithm, and by implementing some pruning criteria (e.g. implementing more If-else rules) or employing a beam search algorithm that stores the best hypotheses for each colour channel, the greediness can be eliminated, and the appropriate amount of expansion can be achieved for each window individually. Other statistical tests or information metrics could be tested for the purpose of improving the Lean-One-Rule algorithm's performance.

The cluster estimator for the K-Means algorithm could be further improved by converting the windows of interest from BGR to greyscale and estimate the exact number of Gaussian kernels that are present in the current frame. This would also improve the K-Means segmentation performance in environments where cluttered background is to be found.

While the system is able to detect the objects of interest, in multiple occasions failed to keep the objects present if they remained static, therefore further development is needed in order to retain the previously detected objects.

Further Development

By utilizing the ICP algorithm, the system could obtain the 3D measurements of the detected object. In combination with a Regression and Classification algorithm (e.g. CART algorithm) and some initial rules from a database, the system could perform as semi-supervised, based on the results of the ICP. This procedure could also provide an additional metric to conclude whether the MoG2 has captured the silhouette of an already known object. By having the 3D models of some known objects, the CART algorithm could select the appropriate model to project over the detected object, provided the ICP has produced reasonably correct results. The system could also be utilized as semi-unsupervised, creating new classes for object that has not encountered before by utilizing the results of the ICP along with the development of further if-else rules. Should the system encounter another object that appears to match some newly created, previously unknown class, the CART algorithm will probably identify said object to that particular class. Other metrics to determine if said object actually belongs to said

class could be employed, should the behaviour of the algorithm deem so (e.g. K-Means Clustering, K-Nearest Neighbours).

References

[1] Arthur, D. & Vassilvitskii, S. (2007), k-means++: the advantages of careful seeding, in 'SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms' , Society for Industrial and Applied Mathematics, Philadelphia, PA, USA , pp. 1027--1035 .

[2] Berthelot, D., Carlini, N., Goodfellow, I., Oliver, A., Papernot, N. and Raffel, C. (2019). MixMatch: A Holistic Approach to Semi-Supervised Learning. Google Research.

[3] Bradski, G., 2000. The OpenCV Library. Dr. Dobb's Journal of Software Tools.

[4] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149), Fort Collins, CO, USA, 1999, pp. 246-252 Vol. 2.

[5] Clark, P. and Niblett, T. (1989). *The CN2 Induction Algorithm*. Netherlands: Kluwer Academic Publishers

[6] Friedman, N., & Russell, S.J. (1997). Image Segmentation in Video Sequences: A Probabilistic Approach. ArXiv, abs/1302.1539.

[7] Fisher, R., Perkins, S., Walker, A. and Wolfart, E. (2004*). Image Processing Learning Resources*. [online] Homepages.inf.ed.ac.uk. Available at: https://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm [Accessed 23 Aug. 2019].

[8] Hayes, M. (1996). Statistical digital signal processing and modelling. New York: Wiley.

[9] IIHS-HLDI crash testing and highway safety. (2018). IIHS examines driver assistance features in road, track tests. [online] Available at: https://www.iihs.org/news/detail/iihs-examines-driver-assistance-features-in-road-track-tests [Accessed 14 Jun. 2019].

[10] Ji, X., Henriques, J. and Vedaldi, A. (2019). Invariant Information Clustering for Unsupervised Image Classification and Segmentation. University of Oxford.

[11] Kowsari, K., Heidarysafa, M., Brown, D., Meimandi, K. and Barnes, L. (2018). *RMDL: Random Multimodel Deep Learning for Classification.*

[12] Mitchell, T. (1997). Machine learning. New York: MacGraw-Hill.

[13] Monson, M. (1999). Schaum's outline of theory and problems digital signal processing. New York: McGraw-Hill.

[14] Ogunnaike, B. (2010). Random phenomena. Boca Raton, Fla.: CRC Press.

[15] Oliphant, T.E., 2006. A guide to NumPy, Trelgol Publishing USA.

[16] Pedregosa, F. et al., 2011. Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), pp.2825–2830.

[17] Porter, F. (2008). *Testing Consistency of Two Histograms*. [online] Pasadena, California 91125: California Institute of Technology. Available at: https://arxiv.org/abs/0804.0380 [Accessed 22 Jul. 2019].

[18] Redman J., Farhadi A. (2016) *YOLO9000: Better, Faster, Stronger*. Washington: University of Washington

[19] Redman J., Farhadi A. (2018) *YOLOv3: An Incremental Improvement*. Washington: University of Washington

[20] Soong, T. (2005). Fundamentals of probability and statistics for engineers. Chichester (England): John Wiley and Sons.

[21] Piech, C. (2013). CS221. [online] Stanford.edu. Available at: https://stanford.edu/~cpiech/cs221/handouts/kmeans.html [Accessed 25 Jul. 2019].

[22] Szeliski R. (2010) Computer Vision: Algorithms and Applications. [online] Available at: http://szeliski.org/Book/

[23] Tan, M. and Le, Q. (2019). *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.*

[24] The University of Auckland (2019). Morphological Image Processing. [online] Available at: https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm [Accessed 16 Jun. 2019].

[25] Van Rossum, G. & Drake Jr, F.L., 1995. Python tutorial, Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

[26] Wang, Q., Zhang, L., Bertinetto, L., Hu, W. and Torr, P. (2019). *Fast Online Object Tracking and Segmentation: A Unifying Approach*.

[27] Z. Zivkovic and Ferdinand van der Heijden. *Efficient adaptive density estimation per image pixel for the task of background subtraction*. Pattern recognition letters, 27(7):773–780, 2006.

[28] Z. Zivkovic (2004) *Improved adaptive gaussian mixture model for background subtraction*. In Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on, volume 2, pages 28–31. IEEE, 2004.

[29] Z. Zivkovic and F.van der Heijden, “Recursive Unsupervised Learning of Finite Mixture Models”, IEEE Trans. on PAMI, vol.26., no.5, 2004.

Appendices

Appendix A: Image kernels Function

```
# ------------------------------------------------------------------------ #
# Image Kernels Function
# ------------------------------------------------------------------------ #
def kernels_hist(frame):
    '''
    Function that evaluates the kernels of interest for the input frame and
    outputs the histogram of each colour channel that remains.

    Kernels size: 3x3

    Inputs:     frame: Image of any dimensions

    Outputs:    hist: Array of size 256x3 that contains the evaluated histograms
                      of each colour channel's kernels of interest.
    '''
    try:
        # Get input frame dimensions
        height, width, col = frame.shape
    except:
        # Get input grayscale frame dimensions
        height, width = frame.shape
        col = 1

    # Initialize list to store temp kernels
    img_kernel = []

    # Initialize lists to store resulting kernels that contain useful info
    b_kernel = []
    g_kernel = []
    r_kernel = []

    # Loop through colour channels of current frame
```

```
for c in range(col):
    # Loop through width of current frame
    for i in range(0, width, 3):
        # Loop through height of current frame
        for k in range(0, height, 3):
            # Case for when at the top-left corner of width
            if i == 0:
                # Case for when at the top-left corner of frame
                if k == 0:
                    # If kernel non-empty
                    if frame[0:3, 0:3, c].size > 0 == True:
                        # If kernel contains useful info
                        if np.any(frame[0:3, 0:3, c]) > 0 == True:
                            # Store Current kernel values
                            img_kernel.append(np.float32(frame[0:3, 0:3, c]))

                        # Else continue to next kernel
                        else:
                            continue
                    # If kernel empty, continue to next kernel
                    else:
                        continue
                # If beyond top-left image corner
                else:
                    # If kernel non-empty
                    if frame[k-3:k, 0:3, c].size > 0 == True:
                        # If kernel contains useful info
                        if np.any(frame[k-3:k, 0:3, c]) > 0 == True:
                            # Store Current kernel values
                            img_kernel.append(np.float32(frame[k-3:k, 0:3, c]))

                        # Else continue to next kernel
                        else:
```

```
                    continue
                # If kernel empty, continue to next kernel
                else:
                    continue
            # If not at the top-left corner of width
            else:
                # Check if kernel is non-empty
                if frame[k-3:k, i-3:i, c].size > 0 == True:
                    # If kernel contains useful info
                    if np.any(frame[k-3:k, i-3:i, c]) > 0 == True:
                        # Store current kernel values
                        img_kernel.append(np.float32(frame[k-3:k, i-3:i, c]))

                    # Else continue to next kernel
                    else:
                        continue
                # If kernel empty, continue to next kernel
                else:
                    continue
    # If blue channel's kernels have been evaluated
    if c == 0:
        # Store kernels to blue kernels list
        b_kernel = img_kernel.copy()

        # Clear list for next colour channel
        img_kernel.clear()

    # If green channel's kernels have been evaluated
    elif c == 1:
        # Store kernels to green kernels list
        g_kernel = img_kernel.copy()

        # Clear list for next colour channel
```

```
            img_kernel.clear()

        # If red channel's kernels have been evaluated
        else:
            # Store kernels to red kernels list
            r_kernel = img_kernel.copy()

            # Clear list
            img_kernel.clear()


    # Assign array to store kernel histograms
    hist = np.zeros((256, 3), dtype=np.uint32)


#    # Evaluate histogram of kernels
#    # Blue channel
    hist[::, 0] = np.bincount(np.asarray(b_kernel).ravel(),minlength=256)

#    # Green channel
    hist[::, 1] = np.bincount(np.asarray(g_kernel).ravel(),minlength=256)

#    # Red channel
    hist[::, 2] = np.bincount(np.asarray(r_kernel).ravel(),minlength=256)


    # Return resulting array
    return(hist)
```

Appendix B: Band Pass FIR Histogram Filter Function

```
# ------------------------------------------------------------------------ #
# Filter Histograms Function
# ------------------------------------------------------------------------ #
def hist_bpf(hist):
    '''
    Function that implements a Band Pass FIR filter in order to suppress the
    potential effects of the number of zero pixels has on the histograms.

    Since background subtraction/foreground detection occurs, it is expected that
    the most dominant pixel value is zero, therefore it has the potential to
    'overshadow' potential useful information.

    Inputs:          hist:     An array of assumed size 256x3 that contains the
                               evaluated histograms of each colour channel.

    Output:          filt_hist:  An array of same size as input array (256x3)
                               that contains the filtered histograms.
    '''
    # Get array dimensions
    height, width = hist.shape

    # Initialize array to store results
    filt_hist = np.zeros((256, 3), dtype=np.float32)

    # Loop through every colour channel
    for i in range(width):
        # Find max value of useful info
        info_hist_max = np.max(hist[3::, i])

        # Check if no objects present; continue to next colour channel
        if info_hist_max == 0:
            continue
```

```
    else:
        # Evaluate Low Pass FIR coefficient
        lp_fir_coeff = info_hist_max / np.max(hist[::, i])

        # Evaluate High Pass threshold
        hp_thresh = info_hist_max * 0.2

        # Loop through each histogram's value & implement Band Pass FIR Filter
        for k in range(height):
            # If current value greater than max value of useful info (Low Pass)
            if hist[k, i] > info_hist_max:
                # Multiply current value with the corresponding coefficient
                filt_hist[k, i] = np.uint32(hist[k, i] * lp_fir_coeff)
            # If current value greater than High Pass threshold & less than
            # or equal to max value of useful info (High Pass)
            elif hist[k, i] > hp_thresh and hist[k, i] <= info_hist_max:
                filt_hist[k, i] = hist[k, i]
            # Else set current value to zero
            else:
                filt_hist[k, i] = np.uint32(0)

# Return filtered histogram(s)
return(filt_hist)
```

Appendix C: Normalize Filtered Histograms Function

```
# ---------------------------------------------------------------------- #
# Normalize Histograms Functions
# ---------------------------------------------------------------------- #
def hist_norm(filt_hist):
    '''
    Function that normalizes histogram values. In its current implementation,
    it takes as an input an 256x3 array that contains the histograms of a
    RGB/BGR coloured image. In its current implementation, it is best to enter
    as an input an array of filtered histograms.

    Input:          hist: An array of assumed size 256x3 that contains the
                      evaluated histograms of each colour channel

    Output:         hist_norm: An array of same size as input array (256x3) that
                           contains the normalized input histograms.
    '''

    # Preallocate array for normalized histograms
    hist_norm = np.zeros((256, 3), dtype=np.float32)

    # Loop through each colour channel
    for i in range(0, 3):
        # Check if current colour channel's histogram is empty
        if np.any(filt_hist[:, i] > 0) == True:
            # Evaluate min value of current histogram
            hist_min = np.min(filt_hist[::, i])

            # Evaluate max value of current histogram
            hist_max = np.max(filt_hist[::, i])

            # Normalize current histogram
```

```
        hist_norm[:, i] = (filt_hist[:, i] - hist_min) / (hist_max - hist_min)
    # Else continue to next colour channel
    else:
        continue
# Return normalized histogram(s)
return(hist_norm)
```

Appendix D: Implement Plain & Overlapping Windows of Histograms Function

```
# ---------------------------------------------------------------------- #
# Implement windows for each Colour Channel Function
# ---------------------------------------------------------------------- #
def windows(hist_norm):
    '''
    Function that implements plain & overlapping windows containing values
    of each colour channel's histograms.

    Input:          hist_norm: An array of size 256x3 that contains the histograms
                               of each colour channel. In the current implementation
                               it is optimal for the histogram values to be normalized.

    Outputs:        win_vals: An array of size (8, 32, 3) that contains the values
                              of the input histograms.

                    win_binloc: An array of size (8, 32, 3) that contains the bin
                                locations of the corresponding plain values.

                    over_vals: An array of size (8, 32, 3) that contains the values
                               of the input histograms in an overlapping manner,
                               starting from the 15th element.

                    over_binloc: An array of size (8, 32, 3) that contains the bin
                                 locations of the corresponding overlapping values
    '''
    # Index variables for resulting arrays
    idx = 0
    cnt = 0
    # Overlapping index
    odx = 15
```

```
# Initialize plain window values
win_vals = np.zeros((8, 32, 3), dtype=np.float32)
# Initialize plain window location values
win_binloc = np.zeros((8, 32, 3), dtype=np.uint8)

# Initialize overlapping window values
over_vals = np.zeros((8, 32, 3), dtype=np.float32)
# Initialize overlapping window bin locations
over_binloc = np.zeros((8, 32, 3), dtype=np.uint8)

# Enumerate histogram bins
bin_c = np.arange(0, 256, dtype=np.uint8)

# Iterate over hist values, segmented into 8 parts of 32 vals
for c in range(0, 3):
    for i in range(0, 256, 32):
        for j in range(0, 32):
            # Break when actual max value
            if cnt > 255:
                break
            elif odx > 255:
                # Update plain windows
                win_vals[idx, j, c] = hist_norm[cnt, c]
                win_binloc[idx, j, c] = bin_c[cnt]

                # Set overlapping windows values & binlocs to zero since
                # out of range
                over_vals[idx, j, c] = np.uint32(0)
                over_binloc[idx, j, c] = np.uint8(0)
                cnt += 1
            else:
                # Plain windows
```

```
                win_vals[idx, j, c] = hist_norm[cnt, c]
                win_binloc[idx, j, c] = bin_c[cnt]
                # Overlapping windows
                over_vals[idx, j, c] = hist_norm[odx, c]
                over_binloc[idx, j, c] = bin_c[odx]
                cnt += 1
                odx += 1
        # Update array index
        idx += 1
    # Reset indices for next colour channel
    idx = 0
    cnt = 0
    odx = 15

# Get output arrays as a pandas series
out = pd.Series((win_vals, win_binloc, over_vals, over_binloc))

# Return resulting arrays
return(out)
```

Appendix E: Rule-Based Tree for selection of RGB Plain or Overlapping Windows of Interest

```
# ---------------------------------------------------------------------------- #
# Implement Rule-Based Tree on RGB Windows Function
# ---------------------------------------------------------------------------- #
def rule_tree(win_vals, over_vals, win_binloc, over_binloc):
    '''
    Implementation of a rule-based decision tree, in order to find regions of
    interest, and determine number of clusters. In its current implementation
    the algorithm assumes that the input arrays are of size (8x32x3). The only
    restrictions are the number of rows that must be taken into account & the
    arrays dimensions must be the same.

    Inputs: hist_norm - Normalized & filtered histogram of current frame.
            bin_c - Locations of histogram values (bin locations)

    Outputs: final_win_loc - Bin location for windows of interest.
            final_win_vals - Values of windows of interest.
            final_win_class - Max value of windows of interest.
    '''

    # Initialize array of max values of windows
    max_win_val = np.zeros((8, 1, 3), dtype=np.float32)

    # Initialize array of max values of overlapping windows
    max_over_val = np.zeros((8, 1, 3), dtype=np.float32)

    # Initialize arrays for determining if useful info is present
    win_use_info_cl = np.zeros((8, 4, 3), dtype=np.bool)
    over_use_info_cl = np.zeros((8, 4, 3), dtype=np.bool)

    # Initialize arrays for determining max value of current window
```

```
max_val_logic_over = np.zeros((8, 1, 3), dtype=np.bool)
max_val_logic_win = np.zeros((8, 1, 3), dtype=np.bool)

# Initialize arrays for value percentage
max_val_over_perc = np.zeros((8, 1, 3), dtype=np.bool)
max_val_win_perc = np.zeros((8, 1, 3), dtype=np.bool)

# Initialize arrays that store desired window locations, values & max vals
final_win_loc = np.zeros((8, 32, 3), dtype=np.uint8)
final_win_vals = np.zeros((8, 32, 3), dtype=np.float32)

for c in range(3):
    # Iterate over each row of windows(plain & overlapping)
    for i in range(8):
        # Case for determining if black object is present
        if i == 0:
            # Evaluate max value of current plain & overlapping window
            max_win_val[i, :, c] = np.max(win_vals[i, 2:, c])
            max_over_val[i, :, c] = np.max(over_vals[i, :, c])

            # Initialize array index
            c_idx = 0
            # Check if max value is near the ends of plain & overlapping window
            # And if window contains info
            for val in range(-1, -5, -1):
                # Overlapping window
                over_use_info_cl[i, c_idx, c] = np.logical_and.reduce((over_vals
                [i, val, c]< max_over_val[i, :, c], over_vals[i, c_idx, c]
                < max_over_val[i, :, c], np.mean(over_vals[i, :, c]) > 0))

                # Plain window
                win_use_info_cl[i, c_idx, c] = np.logical_and.reduce((
                win_vals[i, val,c] < max_win_val[i, :, c],
```

```
                np.mean(win_vals[i, :, c]) > 0,
                win_vals[i, c_idx, c] < max_win_val[i, :, c]))

                # Update array index
                c_idx += 1
        else:
            # Evaluate max value of plain & overlapping current window
            max_win_val[i, :, c] = np.max(win_vals[i, :, c])
            max_over_val[i, :, c] = np.max(over_vals[i, :, c])
            # Initialize index for useful info classification
            c_idx = 0
            # Check past and future values around max values if they contain
            # useful info
            for val in range(-1, -5, -1):
                # Store bool results for plain window criteria
                win_use_info_cl[i, c_idx, c] = np.logical_and.reduce((win_vals[i, val, c]
                < max_win_val[i, :, c], win_vals[i, c_idx, c] < max_win_val[i, :, c],
                np.mean(win_vals[i, :, c]) > 0))
                # Store bool results for overlapping window criteria
                over_use_info_cl[i, c_idx, c] = np.logical_and.reduce((over_vals[i, val,
c]
                < max_over_val[i, :, c], over_vals[i, c_idx, c] < max_over_val[i, :, c],
                np.mean(over_vals[i, :, c]) > 0))
                # Update index for checking future values
                c_idx += 1

        # Check if max overlapping value is greater than max plain value
        max_val_logic_over[i, :, c] = max_over_val[i, :, c] > max_win_val[i, :, c]

        # Check if max overlapping value is greater than 0.4
        max_val_over_perc[i, :, c] = max_over_val[i, :, c] > 0.4

        # Check if max plain value is greater than max overlapping value
```

```
max_val_logic_win[i, :, c] = max_win_val[i, :, c] > max_over_val[i, :, c]

# Check if max plain value is greater than 0.4
max_val_win_perc[i, :, c] = max_win_val[i, :, c] > 0.4

# If max plain value is greater than max overlapping
if max_val_logic_win[i, :, c] == True:

    # If plain contains useful info & over 0.4
    if win_use_info_cl[i, :, c].all() and max_val_win_perc[i, :, c]:
        # Get plain window location
        final_win_loc[i, :, c] = win_binloc[i, :, c]

        # Get Plain window values
        final_win_vals[i, :, c] = win_vals[i, :, c]



    # If overlapping contains useful info & over 0.4
    elif over_use_info_cl[i, :, c].all() and max_val_over_perc[i, :, c]:
        # Get current overlapping window bin locations
        final_win_loc[i, :, c] = over_binloc[i, :, c]

        # Get current overlapping window values
        final_win_vals[i, :, c] = over_vals[i, :, c]

    # If both windows do not fill criteria, continue to next windows
    else:
        continue

# If overlapping max value is greater than max plain
elif max_val_logic_over[i, :, c] == True:

    # If overlapping contains useful info & over 0.4
```

```
    if over_use_info_cl[i, :, c].all() and max_val_over_perc[i, :, c]:
        # Get current overlapping window's bin location
        final_win_loc[i, :, c] = over_binloc[i, :, c]

        # Get current overlapping window's values
        final_win_vals[i, :, c] = over_vals[i, :, c]

    # If plain contains useful info & over 0.4
    elif win_use_info_cl[i, :, c].all() and max_val_win_perc[i, :, c]:
        # Get plain window bin locations
        final_win_loc[i, :, c] = win_binloc[i, :, c]

        # Get plain window values
        final_win_vals[i, :, c] = win_vals[i, :, c]

    # If both windows do not fill criteria, continue to next windows
    else:
        continue

# If both max values are equal
elif max_val_logic_over[i, :, c] == max_val_logic_win[i, :, c]:
    # If plain contains useful info & over 0.4
    if win_use_info_cl[i, :, c].all() and max_val_win_perc[i, :, c]:
        # Get plain window bin location
        final_win_loc[i, :, c] = win_binloc[i, :, c]

        # Get plain window values
        final_win_vals[i, :, c] = win_vals[i, :, c]


    # If overlapping contains useful info & over 0.4
    elif over_use_info_cl[i, :, c].all() and max_val_over_perc[i, :, c]:
        # Get overlapping window's bin location
```

```
            final_win_loc[i, :, c] = over_binloc[i, :, c]

            # Get overlapping window's values
            final_win_vals[i, :, c] = over_vals[i, :, c]

        # Case when both windows fulfil criteria
        elif np.logical_and.reduce((over_use_info_cl[i,:,c].all() == True,
                                    max_val_over_perc[i,:,c] == True,
                                    win_use_info_cl[i,:,c].all() == True,
                                    max_val_win_perc[i,:,c] == True)):

            # Evaluate mean value of plain & overlapping windows
            plain_mean = np.mean(win_vals[i,:,c], dtype=np.float64)
            over_mean = np.mean(over_vals[i,:,c], dtype=np.float64)

            # Evaluate if plain mean greater than overlapping
            if plain_mean > over_mean:
                # Store plain windows values & bin locations
                final_win_loc[i,:,c] = win_binloc[i,:,c]

                final_win_vals[i,:,c] = win_vals[i,:,c]

            # Else overlapping window contains useful info
            else:
                # Store overlapping windows values & bin locations
                final_win_loc[i,:,c] = over_binloc[i,:,c]

                final_win_vals[i,:,c] = over_vals[i,:,c]

        # If both windows do not fill criteria, continue to next windows
        else:
            continue
```

```
        # Continue to next set of windows
        else:
            continue



# Get resulting arrays as a Pandas series
out = pd.Series((final_win_vals[:, :, :], final_win_loc[:, :, :]))

# Return resulting arrays
return(out)
```

Appendix F: K-Means Function (OpenCV library)

```
# ---------------------------------------------------------------------- #
# K-Means Implementation Function - OpenCV
# ---------------------------------------------------------------------- #
def kmeans_cv(frame, n_clusters):
    '''
    K-Means function that utilizes the already built-in function found in the
    OpenCV library. In the current implementation, the k-means clustering alg-
    orithm is utilized in order to separate potential present objects in the
    current frame. The algorithm utilizes the Kmeans++ initialization.
    The criteria for the K-Means are defined as, max number of iterations set to
    300, and the acceptable error rate is set to 1e-4.

    Inputs:         frame: Current frame; can be any size or colour type

                    n_clusters: Number of clusters for the algorithm to evaluate

    Outputs:        res_frame: Resulting frame from the k-means algorithm
    '''
    # Define criteria
    criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 300,
1e-4)

    # Flatten input frame
    inpt_cv = np.float32(frame.reshape((-1, 3)))

    # Fit current frame to the k-means algorithm
    ret,label,center = cv.kmeans(inpt_cv, n_clusters, None, criteria,
                                 10, cv.KMEANS_PP_CENTERS)

    # Obtain labels
    center = np.uint8(center)
```

```
# Evaluate new frame based on resulting labels
res_frame = center[label.flatten()]

# Reshape frame to its original dimensions
res_frame = res_frame.reshape((frame.shape))

# Return resulting array
return(res_frame)
```

Appendix G: K-Means Function (Scikit-Learn library)

```
# ---------------------------------------------------------------------- #
# K-Means Implementation Function - Scikit-Learn
# ---------------------------------------------------------------------- #
def kmeans_sk(frame, n_clusters):
    '''
    K-Means function that utilizes the already built-in function found in the
    scikit-learn library. In the current implementation, the k-means clustering
    algorithm is utilized in order to separate potential present objects in the
    current frame. This function utilizes the Kmeans++ initialization, as well
    as 2 CPU cores for faster processing. The criteria for the k-means are
    kept to their default values, which are 300 max iterations and acceptable
    error rate of 1e-4.

    Inputs:         frame: Current frame; can be any size or colour type

                    n_clusters: Number of clusters for the algorithm to evaluate

    Outputs:        res_frame: Resulting frame from the k-means algorithm
    '''
    # Get image dimensions
    heigh, width, _ =  frame.shape

    # Flatten image values for the k-means algorithm
    inpt = np.reshape(frame, (width * heigh, 3))

    # Initialize the k-means model
    kmeans = KMeans(n_clusters, init='k-means++', n_jobs=2)

    # Fit the input image into the model
    kmeans.fit(inpt)
```

```
# Predict the closest cluster each sample in input image belongs to
labels = kmeans.predict(inpt)

# Output separated objects into image
res_frame = np.zeros((heigh, width, 3), dtype=np.uint8)

# Initialize label index
label_idx = 0

# Loop through image dimensions
for i in range(heigh):
    for k in range(width):
        # At each iteration, select the corresponding cluster center of each label
        res_frame[i, k] = kmeans.cluster_centers_[labels[label_idx]]
        # Update label index
        label_idx += 1

# Return resulting frame
return(res_frame)
```

Appendix H: Kolmogorov-Smirnov Statistical Test for the MOG2 algorithm to converge

```
# ------------------------------------------------------------------------ #
# Kolmogorov-Sminrov Statistical Test to converge MOG2
# ------------------------------------------------------------------------ #
def stop_mog2(curr_hist, prev_hist):
    '''
    Kolmogorov-Smirnov Statistical test for MOG2 to converge.
    The current function evaluates the Cumulative Densities Functions
    of current & previous frame's histograms. Depending upon the results of
    the function, the MOG2 algorithm will either continue, or stop (e.g. converged).

    Alternative hypothesis (H1) is defined as histograms do not overlap significantly,
    therefore, the MOG2 algorithm has not converged yet.

    Null hypothesis (H0) is defined as histograms do overlap significantly,
    therefore, the MOG2 algorithm has converged, so it must stop.

    Inputs:         curr_hist: Histogram of current frame
                    prev_hist: Histogram of previous frame

    Outputs:        Tks_b : Difference current and previous histogram
                      for the Blue colour channel.

                    Tks_g : Difference between current and previous histogram
                      for the Green colour channel.

                    Tks_r : Difference between current and previous histogram
                      for the Red colour channel
    '''
    # Evaluate CDF of each colour channel for current frame
    curr_cdf_b = curr_hist[:, 0] / 256
```

```
curr_cdf_g = curr_hist[:, 1] / 256
curr_cdf_r = curr_hist[:, 2] / 256

# Evaluate CDF of each colour channel for previous frame
prev_cdf_b = prev_hist[:, 0] / 256
prev_cdf_g = prev_hist[:, 1] / 256
prev_cdf_r = prev_hist[:, 2] / 256

# Evaluate difference for Blue colour channel
Tks_b = np.max(np.abs(curr_cdf_b - prev_cdf_b))

# Evaluate difference for Green colour channel
Tks_g = np.max(np.abs(curr_cdf_g - prev_cdf_g))

# Evaluate difference for Red colour channel
Tks_r = np.max(np.abs(curr_cdf_r - prev_cdf_r))

# If Null Hypothesis true
if np.logical_and.reduce((Tks_b < 0.01,
                          Tks_g < 0.01,
                          Tks_r < 0.01)):
    return(True)
# If Alternative Hypothesis true
else:
    return(False)
```

Appendix I: Image Processing Function – Morphological Operations & Contour Capture

```
# ---------------------------------------------------------------------------- #
# Image Processing Function - Morphological Operations & Contour capture
# ---------------------------------------------------------------------------- #
def frame_proc(frame, fgmask, kernel, contour_size):
    '''
    Function that implements a number of morphological operations in order to
    capture the contours of the detected objects (i.e. contours of interest)

    The function first performs a bitwise self-addition of the input frame,
    utilizing the evaluated mask from the MOG2 algorithm.

    Afterwards, a morphological dilation is performed on the frame in order to
    close potential empty regions inside the contours of interest.

    The morphological closing is performed on the frame in order to capture the
    now filled contours of interest.

    The entire silhouette of each contour is evaluated and captured, in order
    to draw the detected contours; the contours that are below of a threshold
    value are deleted.

    Inputs:         frame: Input frame

                    fgmask: Evaluated mask for the current input frame

                    kernel: Kernel of size 9x9

                    contour_size: Threshold of accepted contours size,
                            defined as 60 pixels.
```

```
Outputs:        res2: Output frame, masked with the morphological closing
                      of input frame

                res: Frame that the contours are to be drawn.

                contours: The detected contours of current frame
'''
# Self bitwise operation on current frame
res = cv.bitwise_and(frame,frame, mask = fgmask)

# Morphologically dilate current frame
e_im = cv.dilate(fgmask, kernel, iterations = 1)

# Morphologically close current frame
e_im = cv.morphologyEx(e_im, cv.MORPH_CLOSE, kernel)

# Evaluate & capture each entire silhouettes
contours, hierarchy = cv.findContours(e_im, cv.RETR_EXTERNAL,
                                      cv.CHAIN_APPROX_SIMPLE)

# Remove contours that are lower than threshold's value
temp = []
num = 0
# Loop through each detected contours
for i in range(len(contours)):
    # If current contour size less than threshold's value, store contour
    if len(contours[i]) < contour_size:
        temp.append(i)
# Loop through each contour that is less than threshold's value
for i in temp:
    # Delete the contours that are less than threshold's value
    del contours[i - num]
    num = num + 1
```

```
# Perform bitwise and operation using the morphological processed frame as
# a mask
res2 = cv.bitwise_and(frame,frame, mask = e_im)

# Implement outputs as a pandas Series object
out = pd.Series((res2, res, contours))

# Return resulting frames and detected contours
return(out)
```

Appendix J: Histogram Standard Deviation for the MOG2 algorithm to converge

```
# ------------------------------------------------------------------------ #
# Histogram Standard Deviation to converge MOG2
# ------------------------------------------------------------------------ #
def hist_deviation(out_win_vals):
    '''
    Function that evaluates the deviation of each resulting adaptive window,
    in order to evaluate if the MOG2 algorithm needs to converge or to continue.
    The function identifies the useful windows and evaluates its deviation.
    Useful windows are defined as the windows that contain values.

    Inputs:         out_win_vals: A 3d array of unknown size, containing the
                                  values of the resulting adaptive windows.

    Outputs:        True if the majority of the deviations are greater than
                    the threshold value, defined as 0.21.

                    False if the majority of the deviations are less than the
                    threshold value.
    '''
    try:
        # Obtain dimensions of input array
        h, _, _ = out_win_vals.shape
    except:
        # If no windows of interest, return false
        return(False)
    else:

        # Initialize arrays to store results
        res_var = np.zeros((h, 1, 3), dtype=np.float64)
        res_logic = np.zeros((h, 1, 3), dtype=np.bool)
```

```
    # Initialize indices to store the number of windows that its deviation is
    # greater than or equal to the threshold value & the number of windows that
are empty
    n_valid = 0
    n_reject = 0

    # Loop through each colour channel
    for c in range(3):
        # Loop through each window
        for i in range(h):
            # Evaluate deviation of current window
            res_var[i, 0, c] = np.std(out_win_vals[i, :, c], dtype=np.float64)

            # Evaluate if current window is non empty
            if np.any(out_win_vals[i, :, c]) > 0:
                # If deviation of current window greater than or equal to thresh
                if res_var[i, 0, c] >= 0.21:
                    # Increase the number of valid windows
                    n_valid += 1
                    # Store Boolean result for debugging purposes
                    res_logic[i, 0, c] = True
                # If deviation of current window less than thresh
                else:
                    # Increase the number of rejected windows
                    n_reject += 1
                    # Store Boolean result for debugging purposes
                    res_logic[i, 0, c] = False
            # If current window empty, increase the number of rejected windows
            else:
                n_reject += 1

    # Evaluate the number of non-empty windows
```

```
n_actual = (h * 3) - n_reject

# Evaluate if number of valid windows greater than or equal to the mean of
# windows that contain useful info
if n_valid >= np.ceil(n_actual / 2):
    return(True)
else:
    return(False)
```

Appendix K: Adaptive windows of interest utilizing the Kolmogorov-Smirnov Statistical Test function & Cluster Estimation

```
# ------------------------------------------------------------------------ #
# Implement adaptive windows of interest by means of KS statistical test
# ------------------------------------------------------------------------ #
def adapt_win(final_win_vals, final_win_loc, hist_norm,
              win_vals, win_binloc, over_vals, over_binloc, deb_flg):
    '''
    Function that utilizes the Kolmogorov-Smirnov statistical test in order to
    implement adaptive windows that contain the resulting histograms of each
    colour channel. In its current implementation, the windows of interest,
    resulting from the Rule Based Tree (rule_tree) function are utilized.

    The Null Hypothesis (H0) is defined as the shape of plain & overlapping
    windows histograms overlap, therefore the current window expands accordingly
    until they no longer overlap significantly.

    The Alternative Hypothesis is defined as the shape of plain & overlapping
    windows do not overlap significantly, therefore no need to expand current
    window, keep current window as is

    In its current implementation, it is assumed that the input arrays have
    dimensions of (8x32x3); however the only restriction is the number of rows
    that the algorithm takes into consideration.


    Inputs:         final_win_vals: The values of the windows of interest
                                    that the rule based tree has deemed as
                                    appropriate.

                    final_win_loc: The bin locations of the corresponding
                                   windows of interest.
```

hist_norm: The normalized histogram of each colour channel; it is utilized for the expansion of the plain & overlapping windows that the KS test deems as of significantly similar shape.

win_vals: The resulting values of plain windows. The resulting array is passed into the 'ks_test' function.

win_binloc: The resulting bin locations of plain windows. The resulting array is passed into the 'ks_test' function.

over_vals: The resulting values of overlapping values. The resulting array is passed into the 'ks_test' function.

over_binloc: The resulting bin locations of overlapping windows. The resulting array is passed into the 'ks_test' function.

Outputs: out_binlocs: The resulting bin locations of the adaptive windows. In its current implementation, the bin locations are utilized to determine the number of clusters for the k-means algorithm

out_win_vals: The resulting values of the adaptive windows. In its current implementation, the values are utilized as a proof of concept and to enumerate the resulting number of clusters for the k-means algorithm.

```
                n_clusters: The number of clusters for the k-means algo-
                        rithm. The following assumptions are made
                        in order to obtain the number of clusters:
                        a) Windows of different colour channel but
                          with same bin location, count as 1 cluster
                        b) Windows of same or different colour chan-
                          nel but different bin location constitute
                          as one cluster each.
    '''
# ------------------------------------------------------------------------ #
# Kolmogorov-Smirnov Statistical Hypothesis test Function
# ------------------------------------------------------------------------ #
    def ks_test_tree(win_vals, win_binloc, over_vals, over_binloc, hist_norm,
            final_win_vals, final_win_loc, bin_c):
        '''
        Kolmogorov-Smirnov statistical test function. The statistical test evaluates
        the similarity between the plain and overlapping windows.

        Null Hypothesis is defined as the shape of plain & overlapping windows is
        significantly similar, therefore further expansion of current window is needed.

        Alternative Hypothesis is defined as the shape of plain & overlapping windows
        is not significantly similar, therefore current window does not require
        further expansion.

        Inputs:         win_vals: Array of size (1x32x3) that contains the plain
                            windows values.

                    win_binloc: Array of size(1x32x3) that contains the
                            bin locations of plain windows.
```

```
                over_vals: Array of size (1x32x3) that contains the
                    overlapping windows values

                over_binloc: Array of size (1x32x3) that contains the
                    bin locations of overlapping windows

                hist_norm: Array of size (256x3) that contains the
                    normalized histograms of each colour channels

                final_win_vals: Array of size(1x32x3) that contains the
                        values of the windows that the Rule Based
                        Tree deemed as of interest.

                final_win_loc: Array of size (1x32x3) that contains the
                        bin locations of the windows that the
                        Rule Based Tree deemed as of interest.

                bin_c: Array of size(256) that contains the total bin
                        locations of each histogram.(0-255)


        Outputs:        out_binlocs: Array of size (32x3) that contains the
                            adaptive windows bin locations.

                        out_win_vals: Array of size (32x3) that contains the
                            adaptive windows values.
        '''
# ------------------------------------------------------------------------ #
# Alternative Hypothesis (H1) Function
# ------------------------------------------------------------------------ #
        def alt_hyp(final_win_loc, final_win_vals):
            '''
            Alternative Hypothesis (H1) of Kolmogorov-Smirnov statistical test.
```

H1 is defined as the shape of input histograms is not significantly the same.

Inputs: final_win_loc: Input array that contains bin locations of histograms of interest, as deemed by 'rule_based_tree' function.

final_win_vals: Values of input array that contains the windows of interest, as deemed by 'rule_based_tree' function.

Outputs: out_binlocs: The resulting bin locations of output windows

out_win_vals: Array containing the resulting windows values.

'''

```
# Check if all colour channels are present
if np.logical_and.reduce((np.any(final_win_loc[:, 0]) != 0,
                          np.any(final_win_loc[:, 1]) != 0,
                          np.any(final_win_loc[:, 2]) != 0)):

    # Initialize array values
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Evaluate bin locations
    out_binlocs[:, 0] = final_win_loc[:, 0]  # Blue Channel
    out_binlocs[:, 1] = final_win_loc[:, 1]  # Green Channel
    out_binlocs[:, 2] = final_win_loc[:, 2]  # Red Channel

    # Evaluate windows values
    out_win_vals[:, 0] = final_win_vals[:, 0]  # Blue Channel
    out_win_vals[:, 1] = final_win_vals[:, 1]  # Green Channel
    out_win_vals[:, 2] = final_win_vals[:, 2]  # Red Channel
```

```
    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)
# Check if Blue channel present
elif np.any(final_win_loc[:, 0] != 0):
    # If Green & Blue Channels present
    if np.any(final_win_loc[i, 1]) != 0:
        # Initialize array values
        out_binlocs = np.zeros((32, 3), dtype=np.uint8)
        out_win_vals = np.zeros((32, 3), dtype=np.float32)

        # Evaluate bin locations
        out_binlocs[:, 0] = final_win_loc[:, 0]  # Blue Channel
        out_binlocs[:, 1] = final_win_loc[:, 1]  # Green Channel

        # Evaluate windows values
        out_win_vals[:, 0] = final_win_vals[:, 0]  # Blue Channel
        out_win_vals[:, 1] = final_win_vals[:, 1]  # Green Channel

        # Return resulting arrays as a pandas Series object
        out = pd.Series((out_binlocs, out_win_vals))

        # Return output
        return(out)
    # Check if Red & Blue Channels present
    elif np.any(final_win_loc[:, 2]) != 0:
        # Initialize array values
        out_binlocs = np.zeros((32, 3), dtype=np.uint8)
        out_win_vals = np.zeros((32, 3), dtype=np.float32)
```

```
            # Evaluate bin locations
            out_binlocs[:, 0] = final_win_loc[:, 0]  # Blue Channel
            out_binlocs[:, 2] = final_win_loc[:, 2]  # Red Channel

            # Evaluate windows values
            out_win_vals[:, 0] = final_win_vals[:, 0]  # Blue Channel
            out_win_vals[:, 2] = final_win_vals[:, 2]  # Red Channel

            # Return resulting arrays as a pandas Series object
            out = pd.Series((out_binlocs, out_win_vals))

            # Return output
            return(out)
        # If only Blue Channel present
        else:
            # Initialize array values
            out_binlocs = np.zeros((32, 3), dtype=np.uint8)
            out_win_vals = np.zeros((32, 3), dtype=np.float32)

            # Evaluate bin locations
            out_binlocs[:, 0] = final_win_loc[:, 0]  # Blue Channel

            # Evaluate windows values
            out_win_vals[:, 0] = final_win_vals[:, 0]  # Blue Channel

            # Return resulting arrays as a pandas Series object
            out = pd.Series((out_binlocs, out_win_vals))

            # Return output
            return(out)
    # If Green Channel present
    elif np.any(final_win_loc[:, 1]) != 0:
        # If Green & Blue present
```

```
if np.any(final_win_loc[:, 0]) != 0:
    # Initialize array values
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Evaluate bin locations
    out_binlocs[:, 0] = final_win_loc[:, 0]  # Blue Channel
    out_binlocs[:, 1] = final_win_loc[:, 1]  # Green Channel

    # Evaluate windows values
    out_win_vals[:, 0] = final_win_vals[:, 0]  # Blue Channel
    out_win_vals[:, 1] = final_win_vals[:, 1]  # Green Channel

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)
# If Red & Green Channels present
elif np.any(final_win_loc[:, 2]) != 0:
    # Initialize array values
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Evaluate bin locations
    out_binlocs[:, 1] = final_win_loc[:, 1]  # Green Channel
    out_binlocs[:, 2] = final_win_loc[:, 2]  # Red Channel

    # Evaluate windows values
    out_win_vals[:, 1] = final_win_vals[:, 1]  # Green Channel
    out_win_vals[:, 2] = final_win_vals[:, 2]  # Red Channel

    # Return resulting arrays as a pandas Series object
```

```
            out = pd.Series((out_binlocs, out_win_vals))

            # Return output
            return(out)
        # If Green Channel  present only
        else:
            # Initialize array values
            out_binlocs = np.zeros((32, 3), dtype=np.uint8)
            out_win_vals = np.zeros((32, 3), dtype=np.float32)

            # Evaluate bin locations
            out_binlocs[:, 1] = final_win_loc[:, 1]  # Green Channel

            # Evaluate windows values
            out_win_vals[:, 1] = final_win_vals[:, 1]  # Green Channel

            # Return resulting arrays as a pandas Series object
            out = pd.Series((out_binlocs, out_win_vals))

            # Return output
            return(out)
    # If Red channel present
    elif np.any(final_win_loc[:, 2]) != 0:
        # If Red & Blue present
        if np.any(final_win_loc[:, 0]) != 0:
            # Initialize array values
            out_binlocs = np.zeros((32, 3), dtype=np.uint8)
            out_win_vals = np.zeros((32, 3), dtype=np.float32)

            # Evaluate bin locations
            out_binlocs[:, 0] = final_win_loc[:, 0]  # Blue Channel
            out_binlocs[:, 2] = final_win_loc[:, 2]  # Red Channel
```

```
    # Evaluate windows values
    out_win_vals[:, 0] = final_win_vals[:, 0]  # Blue Channel
    out_win_vals[:, 2] = final_win_vals[:, 2]  # Red Channel

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)
# If Red & Green present
elif np.any(final_win_loc[:, 1]) != 0:

    # Initialize array values
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Evaluate bin locations
    out_binlocs[:, 1] = final_win_loc[:, 1]  # Green Channel
    out_binlocs[:, 2] = final_win_loc[:, 2]  # Red Channel

    # Evaluate windows values
    out_win_vals[:, 1] = final_win_vals[:, 1]  # Green Channel
    out_win_vals[:, 2] = final_win_vals[:, 2]  # Red Channel

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)
# If only Red Channel present
else:
    # Initialize array values
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
```

```
                out_win_vals = np.zeros((32, 3), dtype=np.float32)

                # Evaluate bin locations
                out_binlocs[:, 2] = final_win_loc[:, 2]  # Red Channel

                # Evaluate windows values
                out_win_vals[:, 2] = final_win_vals[:, 2]  # Red Channel

                # Return resulting arrays as a pandas Series object
                out = pd.Series((out_binlocs, out_win_vals))

                # Return output
                return(out)
            # If no colour channel present
            else:
                # Return zero filled arrays
                out_binlocs = np.zeros((32, 3), dtype=np.uint8)
                out_win_vals = np.zeros((32, 3), dtype=np.uint8)

                # Return resulting arrays as a pandas Series object
                out = pd.Series((out_binlocs, out_win_vals))

                # Return output
                return(out)
# ---------------------------------------------------------------------- #
# End of Alternative Hypothesis (H1) Function
# ---------------------------------------------------------------------- #
        # If all colour channels present
        if np.logical_and.reduce((np.any(final_win_loc[:, 0]) != 0,
                                  np.any(final_win_loc[:, 1]) != 0,
                                  np.any(final_win_loc[:, 2]) != 0)):
            # Initialize index for Colour Channels
            idx = 1
```

```
cdx = 1
# Evaluate Red channel's CDF
ur = win_vals[:, 2] / 32
vr = over_vals[:, 2] / 32

# Evaluate difference of plain & overlapping CDF's
Tks_r = np.max(np.abs(ur - vr))

# Evaluate Green channel's CDF
ug = win_vals[:, 1] / 32
vg = over_vals[:, 1] / 32

# Evaluate difference of plain & overlapping CDF's
Tks_g = np.max(np.abs(ug - vg))

# Evaluate Blue Channel's CDF
ub = win_vals[:, 0] / 32
vb = over_vals[:, 0] / 32

# Evaluate difference of plain & overlapping CDF's
Tks_b = np.max(np.abs(ub - vb))

# If Alternative Hypothesis (H1) true
if Tks_r > 0.5 and Tks_g > 0.5 and Tks_b > 0.5:
    # Initialize arrays
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Utilize alternative hypothesis function
    (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                          , final_win_vals[:, :])

    # Return resulting arrays as a pandas Series object
```

```
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_r < 0.5 or Tks_g < 0.5 or Tks_b < 0.5:
        # Initialize arrays to expand Channel values
        temp_win = np.zeros((32 + idx, 3), dtype=np.float32)
        temp_over = np.zeros((32 + idx, 3), dtype=np.float32)

        # If at beginning of histogram values(first window)
        if np.logical_or.reduce((final_win_loc[0, 2] == 0 ,
                                 final_win_loc[0, 1] == 0,
                                 final_win_loc[0, 0] == 0)):
            if idx <= 15:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[np.abs(15 - idx):47, 2]

                # Green channel Windows
                temp_win[:, 1] = hist_norm[0:32+idx, 1]
                temp_over[:, 1] = hist_norm[np.abs(15 - idx):47, 1]

                # Blue Channel Windows
                temp_win[:, 0] = hist_norm[0:32+idx, 0]
                temp_over[:, 0] = hist_norm[np.abs(15 - idx):47, 0]
            else:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[0:47+cdx, 2]
```

```
            # Green channel Windows
            temp_win[:, 1] = hist_norm[0:32+idx, 1]
            temp_over[:, 1] = hist_norm[0:47+cdx, 1]

            # Blue Channel Windows
            temp_win[:, 0] = hist_norm[0:32+idx, 0]
            temp_over[:, 0] = hist_norm[0:47+cdx, 0]
            # Increment index
            cdx += 1

    # If at the end of histogram values (last window)
    elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                               final_win_loc[0, 1] == 224,
                               final_win_loc[0, 0] == 224,
                               final_win_loc[0, 0] == 239,
                               final_win_loc[0, 1] == 239,
                               final_win_loc[0, 2] == 239)):
        # Red Channel Windows
        temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                        -idx: np.int(win_binloc[-1, 2])+1, 2]

        # Pad zeros to the end of the window
        temp_over[:, 2] = np.pad(hist_norm[over_binloc[0,2]
        - 1: over_binloc[16,2]+1, 2],
        (0, (len(temp_over) - (len(hist_norm[over_binloc[0,2]
        - 1: over_binloc[16,2]+1, 2])))), 'constant')



        # Green Channel Windows
        temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                        -idx: np.int(win_binloc[-1, 1])+1, 1]

        # Pad zeros to the end of the window
```

```
    temp_over[:, 1] = np.pad(hist_norm[np.int(over_binloc[ 0, 1])
                    -idx : np.int(over_binloc[16, 1])+1, 1],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 1])
                    -idx : np.int(over_binloc[16, 1])+1, 1])),
    'constant')

    # Blue Channel Windows
    temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                   -idx: np.int(win_binloc[-1, 0])+1, 0]

    # Pad zeros to the end of the window
    temp_over[:, 0] = np.pad(hist_norm[np.int(over_binloc[ 0, 0])
                    -idx : np.int(over_binloc[16, 0])+1, 0],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 0])
                    -idx : np.int(over_binloc[16, 0])+1, 0])),
    'constant')
else:
    # Red Channel Windows
    temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
             : np.int(win_binloc[-1, 2]) + idx, 2]

    temp_over[:, 2] = hist_norm[np.int(over_binloc[i, 0, 2])
                -idx : np.int(over_binloc[-1, 2]), 2]

    # Green Channel Windows
    temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
             : np.int(win_binloc[-1, 1]) + idx, 1]

    temp_over[:, 1] = hist_norm[np.int(over_binloc[0, 1])
              -idx : np.int(over_binloc[-1, 1]), 1]

    # Blue Channel Windows
    temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
```

```
                : np.int(win_binloc[-1, 0]) + idx, 0]

        temp_over[:, 0] = hist_norm[np.int(over_binloc[0, 0])
                -idx : np.int(over_binloc[-1, 0]), 0]

    # Re Evaluate Red channel
    ur = temp_win[:, 2] / len(temp_win)
    vr = temp_over[:, 2] / len(temp_over)
    Tks_r = np.max(np.abs(ur - vr))

    # Re Evaluate Green channel
    ug = temp_win[:, 1] / len(temp_win)
    vg = temp_over[:, 1] / len(temp_over)
    Tks_g = np.max(np.abs(ug - vg))

    # Re Evaluate Blue Channel
    ub = temp_win[:, 0] / len(temp_win)
    vb = temp_over[:, 0] / len(temp_over)
    Tks_b = np.max(np.abs(ub - vb))

    # Increase index value for next iteration
    idx += 1

# Initialize output arrays
out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

# If current window is first window
if final_win_loc[0, 0] == 0:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                np.int(final_win_loc[-1, 0]) + idx, 0]
```

```
    out_binlocs[:, 0] = bin_c[np.int(final_win_loc[0,0]):
        np.int(final_win_loc[-1,0]) + idx]
    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                    np.int(final_win_loc[-1, 1] + idx), 1]

    out_binlocs[:, 1] = bin_c[np.int(final_win_loc[0, 1]):
                    np.int(final_win_loc[-1, 1] + idx)]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                    np.int(final_win_loc[-1, 2] + idx), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                    np.int(final_win_loc[-1, 2] + idx)]

# If current window is last window
elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                        final_win_loc[0, 1] == 224,
                        final_win_loc[0, 0] == 224,
                        final_win_loc[0, 0] == 239,
                        final_win_loc[0, 1] == 239,
                        final_win_loc[0, 2] == 239)):
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                - idx):np.int(final_win_loc[-1, 0]), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                - idx):np.int(final_win_loc[-1, 0])]

    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                - idx):np.int(final_win_loc[-1, 1]), 1]
```

```
        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                        - idx):np.int(final_win_loc[-1, 1])]

        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2]), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2])]

    # If current window any window except first or last
    else:

        # Check if index reaches before beginning of values
        try:
            bin_c[np.int(final_win_loc[0, 0] - idx)] >= 0
        # If Error raised
        except:
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                            np.int(final_win_loc[-1, 0] + idx), 0]

            out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                            np.int(final_win_loc[-1, 0] + idx)]

            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                            np.int(final_win_loc[-1, 1] + idx), 1]

            out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                            np.int(final_win_loc[-1, 1] + idx)]
```

```
    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx)]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx)]

# If no errors occurred (not in the beginning of values)
# check index reaches beyond the end of values
else:
    try:
        bin_c[np.int(final_win_loc[-1, 0] + idx)] >= 255
        # If Error raised (beyond end of values)
    except:
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                            -idx):np.int(final_win_loc[-1, 0]), 0]

        out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                            -idx):np.int(final_win_loc[-1, 0])]

        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                            - idx):np.int(final_win_loc[-1, 1]), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                            - idx):np.int(final_win_loc[-1, 1])]

        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                            - idx):np.int(final_win_loc[-1, 2]), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
```

```
                - idx):np.int(final_win_loc[-1, 2])]

        # If no error raised (not beyond or below range of values)
        else:
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                    - idx):np.int(final_win_loc[-1, 0] + idx), 0]

            out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                    - idx):np.int(final_win_loc[-1, 0] + idx)]

            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1] + idx), 1]

            out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1] + idx)]

            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                    - idx):np.int(final_win_loc[-1, 2] + idx), 2]

            out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                    - idx):np.int(final_win_loc[-1, 2] + idx)]

        # Store resulting arrays as a pd Series object
        out = pd.Series((out_binlocs, out_win_vals))

        # Return resulting arrays
        return(out)

# Determine if Blue channel is present
if np.any(final_win_loc[:, 0]) != 0:
```

```
# Check if Red & Blue channels present only
if np.any(final_win_loc[:, 2]) != 0:
    # Initialize index for Colour Channels
    idx = 1
    cdx = 1
    # Evaluate Red channel's CDF
    ur = win_vals[:, 2] / 32
    vr = over_vals[:, 2] / 32

    # Evaluate difference of plain & overlapping CDF's
    Tks_r = np.max(np.abs(ur - vr))

    # Evaluate Blue Channel's CDF
    ub = win_vals[:, 0] / 32
    vb = over_vals[:, 0] / 32

    # Evaluate difference of plain & overlapping CDF's
    Tks_b = np.max(np.abs(ub - vb))

    # If Alternative Hypothesis (H1) true
    if Tks_r > 0.5 and Tks_b > 0.5:
        # Initialize arrays
        out_binlocs = np.zeros((32, 3), dtype=np.uint8)
        out_win_vals = np.zeros((32, 3), dtype=np.float32)

        # Utilize alternative hypothesis function
        (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                              , final_win_vals[:, :])

        # Return resulting arrays as a pandas Series object
        out = pd.Series((out_binlocs, out_win_vals))

        # Return output
```

```
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_r < 0.5 or Tks_b < 0.5:
        # Initialize arrays to expand Red & Blue Channels values
        temp_win = np.zeros((32 + idx, 3), dtype=np.float32)
        temp_over = np.zeros((32 + idx, 3), dtype=np.float32)



        # If at beginning of histogram values(first window)
        if np.logical_or(final_win_loc[0, 2] == 0 ,
                         final_win_loc[0, 0] == 0):

            if idx <= 15:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[np.abs(15 - idx):46, 2]

                # Blue Channel Windows
                temp_win[:, 0] = hist_norm[0:32+idx, 0]
                temp_over[:, 0] = hist_norm[np.abs(15 - idx):46, 0]
            else:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[0:46+cdx, 2]

                # Blue Channel Windows
                temp_win[:, 0] = hist_norm[0:32+idx, 0]
                temp_over[:, 0] = hist_norm[0:46+cdx, 0]
                # Increment index
                cdx += 1
```

```
# If at the end of histogram values (last window)
elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                final_win_loc[0, 0] == 224,
                final_win_loc[0, 0] == 239,
                final_win_loc[0, 2] == 239)):

    # Red Channel Windows
    temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                    -idx: np.int(win_binloc[-1, 2])+1, 2]

    temp_over[:, 2] = np.pad(hist_norm[np.int(over_binloc[0, 2])
    -idx: np.int(over_binloc[16, 2]), 2],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[0, 2])
    -idx: np.int(over_binloc[16, 2]), 2])), 'constant')



    # Blue Channel Windows
    temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                    -idx: np.int(win_binloc[-1, 0])+1, 0]

    temp_over[:, 0] = np.pad(hist_norm[np.int(over_binloc[ 0, 0])
                     -idx : np.int(over_binloc[16, 0]), 0],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 0])
                     -idx : np.int(over_binloc[16, 0]), 0])),
    'constant')

else:

    # Red Channel Windows
    temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
             : np.int(win_binloc[-1, 2]) + idx, 2]

    temp_over[:, 2] = hist_norm[np.int(over_binloc[i, 0, 2])
```

```
                -idx : np.int(over_binloc[-1, 2]), 2]

        # Blue Channel Windows
        temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                : np.int(win_binloc[-1, 0]) + idx, 0]

        temp_over[:, 0] = hist_norm[np.int(over_binloc[0, 0])
                -idx : np.int(over_binloc[-1, 0]), 0]

    # Re Evaluate Red channel
    ur = temp_win[:, 2] / len(temp_win)
    vr = temp_over[:, 2] / len(temp_over)
    Tks_r = np.max(np.abs(ur - vr))

    # Re Evaluate Blue Channel
    ub = temp_win[:, 0] / len(temp_win)
    vb = temp_over[:, 0] / len(temp_over)
    Tks_b = np.max(np.abs(ub - vb))

    # Increase index value for next iteration
    idx += 1

# Initialize output arrays
out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

# If current window is first window
if final_win_loc[0, 0] == 0:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
        np.int(final_win_loc[-1, 0] + idx), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
```

```
            np.int(final_win_loc[-1, 0] + idx)]

        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                             np.int(final_win_loc[-1, 2] + idx), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                             np.int(final_win_loc[-1, 2] + idx)]

    # If current window is last window
    elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                               final_win_loc[0, 0] == 224,
                               final_win_loc[0, 0] == 239,
                               final_win_loc[0, 2] == 239)):
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                           - idx):np.int(final_win_loc[-1, 0]), 0]

        out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                           - idx):np.int(final_win_loc[-1, 0])]

        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                           - idx):np.int(final_win_loc[-1, 2]), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                           - idx):np.int(final_win_loc[-1, 2])]

    # If current window any window except first or last
    else:

        # Check if index reaches before beginning of values
        try:
```

```
    bin_c[np.int(final_win_loc[0, 0] - idx)] >= 0
# If Error raised
except:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx)]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx)]

# If no errors occurred (not in the beginning of values)
# check index reaches beyond the end of values
else:
    try:
        bin_c[np.int(final_win_loc[-1, 0] + idx)] >= 255
    # If Error raised (beyond end of values)
    except:
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                        -idx):np.int(final_win_loc[-1, 0]), 0]

        out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                        -idx):np.int(final_win_loc[-1, 0])]

        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
```

```
                        - idx):np.int(final_win_loc[-1, 2]), 2]

                out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2])]

            # If no error raised (not beyond or below range of values)
            else:
                # Blue Channel values
                out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                        - idx):np.int(final_win_loc[-1, 0] + idx), 0]

                out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                        - idx):np.int(final_win_loc[-1, 0] + idx)]

                # Red Channel values
                out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2] + idx), 2]

                out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2] + idx)]

        # Store resulting arrays as a pd Series object
        out = pd.Series((out_binlocs, out_win_vals))

        # Return resulting arrays
        return(out)

# Check if Blue & Green channels are present
elif np.any(final_win_loc[:, 1]) != 0:
    # Initialize index for Colour Channels
    idx = 1
    cdx = 1
    # Evaluate Green channel's CDF
```

```
ug = win_vals[:, 1] / 32
vg = over_vals[:, 1] / 32

# Evaluate difference of plain & overlapping CDF's
Tks_g = np.max(np.abs(ug - vg))

# Evaluate Blue Channel's CDF
ub = win_vals[:, 0] / 32
vb = over_vals[:, 0] / 32

# Evaluate difference of plain & overlapping CDF's
Tks_b = np.max(np.abs(ub - vb))

# If Alternative Hypothesis (H1) true
if Tks_g > 0.5 and Tks_b > 0.5:
    # Initialize arrays
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Utilize alternative hypothesis function
    (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                , final_win_vals[:, :])

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_g < 0.5 or Tks_b < 0.5:
        # Initialize arrays to expand Green & Blue Channels window
```

```
temp_win = np.zeros((32 + idx, 1), dtype=np.float32)
temp_over = np.zeros((32 + idx, 1), dtype=np.float32)

# If at beginning of histogram values(first window)
if np.logical_or(final_win_loc[0, 1] == 0,
                 final_win_loc[0, 0] == 0):
    if idx <= 15:
        # Green channel Windows
        temp_win[:, 1] = hist_norm[0:32+idx, 1]
        temp_over[:, 1] = hist_norm[np.abs(15 - idx):46, 1]

        # Blue Channel Windows
        temp_win[:, 0] = hist_norm[0:32+idx, 0]
        temp_over[:, 0] = hist_norm[np.abs(15 - idx):46, 0]
    else:
        # Green channel Windows
        temp_win[:, 1] = hist_norm[0:32+idx, 1]
        temp_over[:, 1] = hist_norm[0:46+cdx, 1]

        # Blue Channel Windows
        temp_win[:, 0] = hist_norm[0:32+idx, 0]
        temp_over[:, 0] = hist_norm[0:46+cdx, 0]
        # Increment index
        cdx += 1

# If at the end of histogram values (last window)
elif np.logical_or.reduce((final_win_loc[0, 1] == 224,
                   final_win_loc[0, 0] == 224,
                   final_win_loc[0, 0] == 239,
                   final_win_loc[0, 1] == 239)):

    # Green Channel Windows
    temp_over[:, 1] = np.pad(hist_norm[np.int(over_binloc[0, 1])
```

```
                -idx: np.int(over_binloc[-1, 1])+1, 1],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[0, 1])
                -idx: np.int(over_binloc[16, 1]), 1])),
    'constant')

    temp_win[:, 1] = hist_norm[np.int(win_binloc[ 0, 1])
                 -idx : np.int(win_binloc[16, 1])+1, 1]

    # Blue Channel Windows
    temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                -idx: np.int(win_binloc[-1, 0])+1, 0]

    temp_over[:, 0] = np.pad(hist_norm[np.int(over_binloc[ 0, 0])
                 -idx : np.int(over_binloc[16, 0]), 0],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 0])
                 -idx : np.int(over_binloc[16, 0]), 0])),
    'constant')

else:

    # Green Channel Windows
    temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
            : np.int(win_binloc[-1, 1]) + idx, 1]

    temp_over[:, 1] = hist_norm[np.int(over_binloc[0, 1])
             -idx : np.int(over_binloc[-1, 1]), 1]

    # Blue Channel Windows
    temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
            : np.int(win_binloc[-1, 0]) + idx, 0]

    temp_over[:, 0] = hist_norm[np.int(over_binloc[0, 0])
             -idx : np.int(over_binloc[-1, 0]), 0]
```

```
    # Re Evaluate Green channel
    ug = temp_win[:, 1] / len(temp_win)
    vg = temp_over[:, 1] / len(temp_over)
    Tks_g = np.max(np.abs(ug - vg))

    # Re Evaluate Blue Channel
    ub = temp_win[:, 0] / len(temp_win)
    vb = temp_over[:, 0] / len(temp_over)
    Tks_b = np.max(np.abs(ub - vb))

    # Increase index value for next iteration
    idx += 1

# Initialize output arrays
out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

# If current window is first window
if final_win_loc[0, 0] == 0:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx)]

    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx)]
```

```
# If current window is last window
elif np.logical_or.reduce((final_win_loc[0, 1] == 224,
                           final_win_loc[0, 0] == 224,
                           final_win_loc[0, 0] == 239,
                           final_win_loc[0, 1] == 239)):
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                        - idx):np.int(final_win_loc[-1, 0]), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                        - idx):np.int(final_win_loc[-1, 0])]

    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                        - idx):np.int(final_win_loc[-1, 1]), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                        - idx):np.int(final_win_loc[-1, 1])]

# If current window any window except first or last
else:

    # Check if index reaches before beginning of values
    try:
        bin_c[np.int(final_win_loc[0, 0] - idx)] >= 0
    # If Error raised
    except:
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                                       np.int(final_win_loc[-1, 0] + idx), 0]

        out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
```

```
                    np.int(final_win_loc[-1, 0] + idx)]

        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                    np.int(final_win_loc[-1, 1] + idx), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                    np.int(final_win_loc[-1, 1] + idx)]

    # If no errors occurred (not in the beginning of values)
    # check index reaches beyond the end of values
    else:
        try:
            bin_c[np.int(final_win_loc[-1, 0] + idx)] >= 255
        # If Error raised (beyond end of values)
        except:
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                    -idx):np.int(final_win_loc[-1, 0]), 0]

            out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                    -idx):np.int(final_win_loc[-1, 0])]

            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1]), 1]

            out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1])]

        # If no error raised (not beyond or below range of values)
        else:
            # Blue Channel values
```

```
                out_win_vals[:, 0] = hist_norm[np.int(
                        final_win_loc[0, 0] - idx):np.int(
                        final_win_loc[-1, 0] + idx), 0]

                out_binlocs[:,0] = bin_c[np.int(
                        final_win_loc[0, 0] - idx):np.int(
                        final_win_loc[-1, 0] + idx)]

                # Green Channel values
                out_win_vals[:, 1] = hist_norm[np.int(
                        final_win_loc[0, 1] - idx):np.int(
                        final_win_loc[-1, 1] + idx), 1]

                out_binlocs[:,1] = bin_c[np.int(
                        final_win_loc[0, 1] - idx):np.int(
                        final_win_loc[-1, 1] + idx)]

        # Store resulting arrays as a pd Series object
        out = pd.Series((out_binlocs, out_win_vals))

        # Return resulting arrays
        return(out)

# If only Blue Channel present
else:
    # Initialize index for Colour Channels
    idx = 1
    cdx = 1
    # Evaluate difference of plain & overlapping CDF's
    Tks_g = np.max(np.abs(ug - vg))

    # Evaluate Blue Channel's CDF
    ub = win_vals[:, 0] / 32
```

```
vb = over_vals[:, 0] / 32

# Evaluate difference of plain & overlapping CDF's
Tks_b = np.max(np.abs(ub - vb))

# If Alternative Hypothesis (H1) true
if Tks_b > 0.5:
    # Initialize arrays
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Utilize alternative hypothesis function
    (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                , final_win_vals[:, :])

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_b < 0.5:
        # Initialize arrays to expand Blue Channel Window
        temp_win = np.zeros((32 + idx, 1), dtype=np.float32)
        temp_over = np.zeros((32 + idx, 1), dtype=np.float32)

        # If at beginning of histogram values(first window)
        if final_win_loc[0, 0] == 0:
            if idx <= 15:
                # Blue Channel Windows
                temp_win[:, 0] = hist_norm[0:32+idx, 0]
```

```
            temp_over[:, 0] = hist_norm[np.abs(15 - idx):46, 0]
        else:
            # Blue Channel Windows
            temp_win[:, 0] = hist_norm[0:32+idx, 0]
            temp_over[:, 0] = hist_norm[0:46+cdx, 0]
            # Increment index
            cdx += 1

    # If at the end of histogram values (last window)
    elif np.logical_or(final_win_loc[0, 0] == 224,
                       final_win_loc[0, 0] == 239):

        # Blue Channel Windows
        temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                        -idx: np.int(win_binloc[-1, 0])+1, 0]

        temp_over[:, 0] = np.pad(hist_norm[np.int(over_binloc[ 0, 0])
                         -idx : np.int(over_binloc[16, 0]), 0],
        (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 0])
                         -idx : np.int(over_binloc[16, 0]), 0])),
        'constant')

    else:

        # Blue Channel Windows
        temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                  : np.int(win_binloc[-1, 0]) + idx, 0]

        temp_over[:, 0] = hist_norm[np.int(over_binloc[0, 0])
                   -idx : np.int(over_binloc[-1, 0]), 0]

    # Re Evaluate Blue Channel
    ub = temp_win[:, 0] / len(temp_win)
```

```
    vb = temp_over[:, 0] / len(temp_over)
    Tks_b = np.max(np.abs(ub - vb))

    # Increase index value for next iteration
    idx += 1

# Initialize output arrays
out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

# If current window is first window
if final_win_loc[0, 0] == 0:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx)]

# If current window is last window
elif final_win_loc[0, 0] == 224 or final_win_loc[0,0] == 239:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                    - idx):np.int(final_win_loc[-1, 0]), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                    - idx):np.int(final_win_loc[-1, 0])]

# If current window any window except first or last
else:

    # Check if index reaches before beginning of values
    try:
```

```
    bin_c[np.int(final_win_loc[0, 0] - idx)] >= 0
# If Error raised
except:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx)]

# If no errors occurred (not in the beginning of values)
# check index reaches beyond the end of values
else:
    try:
        bin_c[np.int(final_win_loc[-1, 0] + idx)] >= 255
    # If Error raised (beyond end of values)
    except:
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                        -idx):np.int(final_win_loc[-1, 0]), 0]

        out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                        -idx):np.int(final_win_loc[-1, 0])]

    # If no error raised (not beyond or below range of values)
    else:
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(
                    final_win_loc[0, 0] - idx):np.int(
                    final_win_loc[-1, 0] + idx), 0]

        out_binlocs[:,0] = bin_c[np.int(
                    final_win_loc[0, 0] - idx):np.int(
```

```
                                    final_win_loc[-1, 0] + idx)]

            # Store resulting arrays as a pd Series object
            out = pd.Series((out_binlocs, out_win_vals))

            # Return resulting arrays
            return(out)
# If Green Channel present
elif np.any(final_win_loc[:, 1]) != 0:
    # If Green & Blue channels present
    if np.any(final_win_loc[:, 0]) != 0:
        # Initialize index for Colour Channels
        idx = 1
        cdx = 1
        # Evaluate Green channel's CDF
        ug = win_vals[:, 1] / 32
        vg = over_vals[:, 1] / 32

        # Evaluate difference of plain & overlapping CDF's
        Tks_g = np.max(np.abs(ug - vg))

        # Evaluate Blue Channel's CDF
        ub = win_vals[:, 0] / 32
        vb = over_vals[:, 0] / 32

        # Evaluate difference of plain & overlapping CDF's
        Tks_b = np.max(np.abs(ub - vb))

        # If Alternative Hypothesis (H1) true
        if Tks_g > 0.5 and Tks_b > 0.5:
            # Initialize arrays
            out_binlocs = np.zeros((32, 3), dtype=np.uint8)
            out_win_vals = np.zeros((32, 3), dtype=np.float32)
```

```
    # Utilize alternative hypothesis function
    (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                , final_win_vals[:, :])

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_g < 0.5 or Tks_b < 0.5:
        # Initialize arrays to expand Green & Blue Channels window
        temp_win = np.zeros((32 + idx, 1), dtype=np.float32)
        temp_over = np.zeros((32 + idx, 1), dtype=np.float32)

        # If at beginning of histogram values(first window)
        if np.logical_or(final_win_loc[0, 1] == 0,
                         final_win_loc[0, 0] == 0):
            if idx <= 15:
                # Green channel Windows
                temp_win[:, 1] = hist_norm[0:32+idx, 1]
                temp_over[:, 1] = hist_norm[np.abs(15 - idx):46, 1]

                # Blue Channel Windows
                temp_win[:, 0] = hist_norm[0:32+idx, 0]
                temp_over[:, 0] = hist_norm[np.abs(15 - idx):46, 0]
            else:
                # Green channel Windows
                temp_win[:, 1] = hist_norm[0:32+idx, 1]
                temp_over[:, 1] = hist_norm[0:46+cdx, 1]
```

```
        # Blue Channel Windows
        temp_win[:, 0] = hist_norm[0:32+idx, 0]
        temp_over[:, 0] = hist_norm[0:46+cdx, 0]
        # Increment index
        cdx += 1

# If at the end of histogram values (last window)
elif np.logical_or.reduce((final_win_loc[0, 1] == 224,
                final_win_loc[0, 0] == 224,
                final_win_loc[0, 0] == 239,
                final_win_loc[0, 1] == 239)):

    # Green Channel Windows
    temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                    -idx: np.int(win_binloc[-1, 1])+1, 1]

    temp_over[:, 1] = np.pad(hist_norm[np.int(over_binloc[ 0, 1])
                    -idx : np.int(over_binloc[16, 1]), 1],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 1])
                    -idx : np.int(over_binloc[16, 1]), 1])),
    'constant')

    # Blue Channel Windows
    temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                    -idx: np.int(win_binloc[-1, 0])+1, 0]

    temp_over[:, 0] = np.pad(hist_norm[np.int(over_binloc[ 0, 0])
                    -idx : np.int(over_binloc[16, 0]), 0],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 0])
                    -idx : np.int(over_binloc[16, 0]), 0])),
    'constant')
```

```
    else:

        # Green Channel Windows
        temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                : np.int(win_binloc[-1, 1]) + idx, 1]

        temp_over[:, 1] = hist_norm[np.int(over_binloc[0, 1])
                -idx : np.int(over_binloc[-1, 1]), 1]

        # Blue Channel Windows
        temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                : np.int(win_binloc[-1, 0]) + idx, 0]

        temp_over[:, 0] = hist_norm[np.int(over_binloc[0, 0])
                -idx : np.int(over_binloc[-1, 0]), 0]

    # Re Evaluate Green channel
    ug = temp_win[:, 1] / len(temp_win)
    vg = temp_over[:, 1] / len(temp_over)
    Tks_g = np.max(np.abs(ug - vg))

    # Re Evaluate Blue Channel
    ub = temp_win[:, 0] / len(temp_win)
    vb = temp_over[:, 0] / len(temp_over)
    Tks_b = np.max(np.abs(ub - vb))

    # Increase index value for next iteration
    idx += 1

# Initialize output arrays
out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)
```

```
# If current window is first window
if final_win_loc[0, 0] == 0:
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                        np.int(final_win_loc[-1, 0] + idx)]

    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx)]

# If current window is last window
elif np.logical_or.reduce((final_win_loc[0, 1] == 224,
                        final_win_loc[0, 0] == 224,
                        final_win_loc[0, 0] == 239,
                        final_win_loc[0, 1] == 239)):
    # Blue Channel values
    out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                    - idx):np.int(final_win_loc[-1, 0]), 0]

    out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                    - idx):np.int(final_win_loc[-1, 0])]

    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1]), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
```

```
                - idx):np.int(final_win_loc[-1, 1])]

# If current window any window except first or last
else:

    # Check if index reaches before beginning of values
    try:
        bin_c[np.int(final_win_loc[0, 0] - idx)] >= 0
    # If Error raised
    except:
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                            np.int(final_win_loc[-1, 0] + idx), 0]

        out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                            np.int(final_win_loc[-1, 0] + idx)]

        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                            np.int(final_win_loc[-1, 1] + idx), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                            np.int(final_win_loc[-1, 1] + idx)]

    # If no errors occurred (not in the beginning of values)
    # check index reaches beyond the end of values
    else:
        try:
            bin_c[np.int(final_win_loc[-1, 0] + idx)] >= 255
            # If Error raised (beyond end of values)
        except:
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
```

```
                    -idx):np.int(final_win_loc[-1, 0]), 0]

            out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                    -idx):np.int(final_win_loc[-1, 0])]

            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1]), 1]

            out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1])]

        # If no error raised (not beyond or below range of values)
        else:
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(
                    final_win_loc[0, 0] - idx):np.int(
                    final_win_loc[-1, 0] + idx), 0]

            out_binlocs[:,0] = bin_c[np.int(
                    final_win_loc[0, 0] - idx):np.int(
                    final_win_loc[-1, 0] + idx)]

            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(
                    final_win_loc[0, 1] - idx):np.int(
                    final_win_loc[-1, 1] + idx), 1]

            out_binlocs[:,1] = bin_c[np.int(
                    final_win_loc[0, 1] - idx):np.int(
                    final_win_loc[-1, 1] + idx)]

    # Store resulting arrays as a pd Series object
```

```
        out = pd.Series((out_binlocs, out_win_vals))

        # Return resulting arrays
        return(out)
# If Green & Red channels present
elif np.any(final_win_loc[:, 2]) != 0:
    # Initialize index for Colour Channels
    idx = 1
    cdx = 1
    # Evaluate Red channel's CDF
    ur = win_vals[:, 2] / 32
    vr = over_vals[:, 2] / 32

    # Evaluate difference of plain & overlapping CDF's
    Tks_r = np.max(np.abs(ur - vr))

    # Evaluate Green channel's CDF
    ug = win_vals[:, 1] / 32
    vg = over_vals[:, 1] / 32

    # Evaluate difference of plain & overlapping CDF's
    Tks_g = np.max(np.abs(ug - vg))

    # If Alternative Hypothesis (H1) true
    if Tks_r > 0.5 and Tks_g > 0.5:
        # Initialize arrays
        out_binlocs = np.zeros((32, 3), dtype=np.uint8)
        out_win_vals = np.zeros((32, 3), dtype=np.float32)

        # Utilize alternative hypothesis function
        (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                              , final_win_vals[:, :])
```

```
    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_r < 0.5 or Tks_g < 0.5:
        # Initialize arrays to expand Red & Green Channels values
        temp_win = np.zeros((32 + idx, 1), dtype=np.float32)
        temp_over = np.zeros((32 + idx, 1), dtype=np.float32)

        # If at beginning of histogram values(first window)
        if np.logical_or(final_win_loc[0, 2] == 0,
                         final_win_loc[0, 1] == 0):
            if idx <= 15:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[np.abs(15 - idx):46, 2]

                # Green channel Windows
                temp_win[:, 1] = hist_norm[0:32+idx, 1]
                temp_over[:, 1] = hist_norm[np.abs(15 - idx):46, 1]
            else:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[0:46+cdx, 2]

                # Green channel Windows
                temp_win[:, 1] = hist_norm[0:32+idx, 1]
                temp_over[:, 1] = hist_norm[0:46+cdx, 1]
                # Increment index
```

```
        cdx = 1

    # If at the end of histogram values (last window)
    elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                    final_win_loc[0, 1] == 224,
                    final_win_loc[0, 1] == 239,
                    final_win_loc[0, 2] == 239)):

        # Red Channel Windows
        temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                        -idx: np.int(win_binloc[-1, 2])+1, 2]

        temp_over[:, 2] = np.pad(hist_norm[np.int(over_binloc[0, 2])
        -idx: np.int(over_binloc[16, 2]), 2],
        (0, len(temp_over) - len(hist_norm[np.int(over_binloc[0, 2])
        -idx: np.int(over_binloc[16, 2]), 2])), 'constant')

        # Green Channel Windows
        temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                        -idx: np.int(win_binloc[-1, 1])+1, 1]

        temp_over[:, 1] = np.pad(hist_norm[np.int(over_binloc[ 0, 1])
                        -idx : np.int(over_binloc[16, 1]), 1],
        (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 1])
                        -idx : np.int(over_binloc[16, 1]), 1])),
        'constant')

    else:

        # Red Channel Windows
        temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                    : np.int(win_binloc[-1, 2]) + idx, 2]
```

```
            temp_over[:, 2] = hist_norm[np.int(over_binloc[i, 0, 2])
                        -idx : np.int(over_binloc[-1, 2]), 2]

            # Green Channel Windows
            temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                    : np.int(win_binloc[-1, 1]) + idx, 1]

            temp_over[:, 1] = hist_norm[np.int(over_binloc[0, 1])
                    -idx : np.int(over_binloc[-1, 1]), 1]



        # Re Evaluate Red channel
        ur = temp_win[:, 2] / len(temp_win)
        vr = temp_over[:, 2] / len(temp_over)
        Tks_r = np.max(np.abs(ur - vr))

        # Re Evaluate Green channel
        ug = temp_win[:, 1] / len(temp_win)
        vg = temp_over[:, 1] / len(temp_over)
        Tks_g = np.max(np.abs(ug - vg))



        # Increase index value for next iteration
        idx += 1

    # Initialize output arrays
    out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

    # If current window is first window
    if final_win_loc[0, 1] == 0:
        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
```

```
                    np.int(final_win_loc[-1, 1] + idx), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                    np.int(final_win_loc[-1, 1] + idx)]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                    np.int(final_win_loc[-1, 2] + idx), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                    np.int(final_win_loc[-1, 2] + idx)]

# If current window is last window
elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                    final_win_loc[0, 1] == 224,
                    final_win_loc[0, 1] == 239,
                    final_win_loc[0, 2] == 239)):
    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                - idx):np.int(final_win_loc[-1, 1]), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                - idx):np.int(final_win_loc[-1, 1])]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                - idx):np.int(final_win_loc[-1, 2]), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                - idx):np.int(final_win_loc[-1, 2])]

# If current window any window except first or last
else:
```

```
# Check if index reaches before beginning of values
try:
    bin_c[np.int(final_win_loc[0, 1] - idx)] >= 0
# If Error raised
except:
    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx)]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx)]

# If no errors occurred (not in the beginning of values)
# check index reaches beyond the end of values
else:
    try:
        bin_c[np.int(final_win_loc[-1, 1] + idx)] >= 255
    # If Error raised (beyond end of values)
    except:
        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1]), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1])]
```

```
            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2]), 2]

            out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2])]

        # If no error raised (not beyond or below range of values)
        else:
            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(
                    final_win_loc[0, 1] - idx):np.int(
                    final_win_loc[-1, 1] + idx), 1]

            out_binlocs[:,1] = bin_c[np.int(
                    final_win_loc[0, 1] - idx):np.int(
                    final_win_loc[-1, 1] + idx)]

            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(
                    final_win_loc[0, 2] - idx):np.int(
                    final_win_loc[-1, 2] + idx), 2]

            out_binlocs[:,2] = bin_c[np.int(
                    final_win_loc[0, 2] - idx):np.int(
                    final_win_loc[-1, 2] + idx)]

    # Store resulting arrays as a pd Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return resulting arrays
    return(out)
```

```
# If Green Channel only
else:
    # Initialize index for Colour Channels
    idx = 1
    cdx = 1
    # Evaluate Green channel's CDF
    ug = win_vals[:, 1] / 32
    vg = over_vals[:, 1] / 32

    # Evaluate difference of plain & overlapping CDF's
    Tks_g = np.max(np.abs(ug - vg))

    # If Alternative Hypothesis (H1) true
    if Tks_g > 0.5:
        # Initialize arrays
        out_binlocs = np.zeros((32, 3), dtype=np.uint8)
        out_win_vals = np.zeros((32, 3), dtype=np.float32)

        # Utilize alternative hypothesis function
        (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                              , final_win_vals[:, :])

        # Return resulting arrays as a pandas Series object
        out = pd.Series((out_binlocs, out_win_vals))

        # Return output
        return(out)

    # If Null Hypothesis (H0) true
    else:
        while Tks_g < 0.5:
            # Initialize arrays to expand Green Channel window
            temp_win = np.zeros((32 + idx, 1), dtype=np.float32)
```

```
temp_over = np.zeros((32 + idx, 1), dtype=np.float32)

# If at beginning of histogram values(first window)
if final_win_loc[0, 1] == 0:
    if idx <= 15:
        # Green channel Windows
        temp_win[:, 1] = hist_norm[0:32+idx, 1]
        temp_over[:, 1] = hist_norm[np.abs(15 - idx):46, 1]
    else:
        # Green channel Windows
        temp_win[:, 1] = hist_norm[0:32+idx, 1]
        temp_over[:, 1] = hist_norm[0:46+cdx, 1]
        # Increment index
        cdx += 1

# If at the end of histogram values (last window)
elif final_win_loc[0, 1] == 224 or final_win_loc[0,1] == 239:

    # Green Channel Windows
    temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                    -idx: np.int(win_binloc[-1, 1])+1, 1]

    temp_over[:, 1] = np.pad(hist_norm[np.int(over_binloc[ 0, 1])
                     -idx : np.int(over_binloc[16, 1]), 1],
(0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 1])
                     -idx : np.int(over_binloc[16, 1]), 1])),
'constant')

else:

    # Green Channel Windows
    temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
             : np.int(win_binloc[-1, 1]) + idx, 1]
```

```
            temp_over[:, 1] = hist_norm[np.int(over_binloc[0, 1])
                    -idx : np.int(over_binloc[-1, 1]), 1]

        # Re Evaluate Green channel
        ug = temp_win[:, 1] / len(temp_win)
        vg = temp_over[:, 1] / len(temp_over)
        Tks_g = np.max(np.abs(ug - vg))

        # Increase index value for next iteration
        idx += 1

    # Initialize output arrays
    out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

    # If current window is first window
    if final_win_loc[0, 1] == 0:
        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx)]

    # If current window is last window
    elif final_win_loc[0, 1] == 224 or final_win_loc[0,1] == 239:
        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1]), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1])]
```

```
# If current window any window except first or last
else:
    # Check if index reaches before beginning of values
    try:
        bin_c[np.int(final_win_loc[0, 1] - idx)] >= 0
    # If Error raised
    except:
        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                            np.int(final_win_loc[-1, 1] + idx), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                            np.int(final_win_loc[-1, 1] + idx)]

    # If no errors occurred (not in the beginning of values)
    # check index reaches beyond the end of values
    else:
        try:
            bin_c[np.int(final_win_loc[-1, 1] + idx)] >= 255
        # If Error raised (beyond end of values)
        except:
            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                        - idx):np.int(final_win_loc[-1, 1]), 1]

            out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                        - idx):np.int(final_win_loc[-1, 1])]

        # If no error raised (not beyond or below range of values)
        else:
            # Green Channel values
            out_win_vals[:, 1] = hist_norm[np.int(
```

```
                        final_win_loc[0, 1] - idx):np.int(
                        final_win_loc[-1, 1] + idx), 1]

                out_binlocs[:,1] = bin_c[np.int(
                        final_win_loc[0, 1] - idx):np.int(
                        final_win_loc[-1, 1] + idx)]

            # Store resulting arrays as a pd Series object
            out = pd.Series((out_binlocs, out_win_vals))

            # Return resulting arrays
            return(out)
    # If Red Chanel present
    elif np.any(final_win_loc[:, 2]) != 0:
        # If Red & Green Channels present
        if np.any(final_win_loc[:, 1]) != 0:
            # Initialize index for Colour Channels
            idx = 1
            cdx = 1
            # Evaluate Red channel's CDF
            ur = win_vals[:, 2] / 32
            vr = over_vals[:, 2] / 32

            # Evaluate difference of plain & overlapping CDF's
            Tks_r = np.max(np.abs(ur - vr))

            # Evaluate Green channel's CDF
            ug = win_vals[:, 1] / 32
            vg = over_vals[:, 1] / 32

            # Evaluate difference of plain & overlapping CDF's
            Tks_g = np.max(np.abs(ug - vg))
```

```
# If Alternative Hypothesis (H1) true
if Tks_r > 0.5 and Tks_g > 0.5:
    # Initialize arrays
    out_binlocs = np.zeros((32, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32, 3), dtype=np.float32)

    # Utilize alternative hypothesis function
    (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                , final_win_vals[:, :])

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_r < 0.5 or Tks_g < 0.5:
        # Initialize arrays to expand Red & Green Channels values
        temp_win = np.zeros((32 + idx, 3), dtype=np.float32)
        temp_over = np.zeros((32 + idx, 3), dtype=np.float32)

        # If at beginning of histogram values(first window)
        if np.logical_or(final_win_loc[0, 2] == 0,
                         final_win_loc[0, 1] == 0):
            if idx <= 15:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[np.abs(15 - idx):46, 2]

                # Green channel Windows
                temp_win[:, 1] = hist_norm[0:32+idx, 1]
```

```
            temp_over[:, 1] = hist_norm[np.abs(15 - idx):46, 1]
        else:
            # Red Channel Windows
            temp_win[:, 2] = hist_norm[0:32+idx, 2]
            temp_over[:, 2] = hist_norm[0:46+cdx, 2]

            # Green channel Windows
            temp_win[:, 1] = hist_norm[0:32+idx, 1]
            temp_over[:, 1] = hist_norm[0:46+cdx, 1]
            # Increment index
            cdx += 1

    # If at the end of histogram values (last window)
    elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                               final_win_loc[0, 1] == 224,
                               final_win_loc[0, 1] == 239,
                               final_win_loc[0, 2] == 239)):
        # Red Channel Windows
        temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                                   -idx: np.int(win_binloc[-1, 2])+1, 2]

        temp_over[:, 2] = np.pad(hist_norm[np.int(over_binloc[0, 2])
        -idx: np.int(over_binloc[16, 2]), 2],
        (0, len(temp_over) - len(hist_norm[np.int(over_binloc[0, 2])
        -idx: np.int(over_binloc[16, 2]), 2])), 'constant')

        # Green Channel Windows
        temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                                   -idx: np.int(win_binloc[-1, 1])+1, 1]

        temp_over[:, 1] = np.pad(hist_norm[np.int(over_binloc[ 0, 1])
                                    -idx : np.int(over_binloc[16, 1]), 1],
        (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 1])
```

```
                    -idx : np.int(over_binloc[16, 1]), 1])),
        'constant')

    else:

        # Red Channel Windows
        temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                : np.int(win_binloc[-1, 2]) + idx, 2]

        temp_over[:, 2] = hist_norm[np.int(over_binloc[i, 0, 2])
                    -idx : np.int(over_binloc[-1, 2]), 2]

        # Green Channel Windows
        temp_win[:, 1] = hist_norm[np.int(win_binloc[0, 1])
                : np.int(win_binloc[-1, 1]) + idx, 1]

        temp_over[:, 1] = hist_norm[np.int(over_binloc[0, 1])
                    -idx : np.int(over_binloc[-1, 1]), 1]

    # Re Evaluate Red channel
    ur = temp_win[:, 2] / len(temp_win)
    vr = temp_over[:, 2] / len(temp_over)
    Tks_r = np.max(np.abs(ur - vr))

    # Re Evaluate Green channel
    ug = temp_win[:, 1] / len(temp_win)
    vg = temp_over[:, 1] / len(temp_over)
    Tks_g = np.max(np.abs(ug - vg))

    # Increase index value for next iteration
    idx += 1

# Initialize output arrays
```

```
out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

# If current window is first window
if final_win_loc[0, 2] == 0:
    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx)]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx)]

# If current window is last window
elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                        final_win_loc[0, 1] == 224,
                        final_win_loc[0, 1] == 239,
                        final_win_loc[0, 2] == 239)):
    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1]), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                    - idx):np.int(final_win_loc[-1, 1])]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
```

```
                - idx):np.int(final_win_loc[-1, 2]), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                - idx):np.int(final_win_loc[-1, 2])]

# If current window any window except first or last
else:

    # Check if index reaches before beginning of values
    try:
        bin_c[np.int(final_win_loc[0, 2] - idx)] >= 0
    # If Error raised
    except:
        # Green Channel values
        out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx), 1]

        out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]):
                        np.int(final_win_loc[-1, 1] + idx)]

        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx)]

    # If no errors occurred (not in the beginning of values)
    # check index reaches beyond the end of values
    else:
        try:
            bin_c[np.int(final_win_loc[-1, 2] + idx)] >= 255
        # If Error raised (beyond end of values)
```

```
except:
    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(final_win_loc[0, 1]
                - idx):np.int(final_win_loc[-1, 1]), 1]

    out_binlocs[:,1] = bin_c[np.int(final_win_loc[0, 1]
                - idx):np.int(final_win_loc[-1, 1])]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                - idx):np.int(final_win_loc[-1, 2]), 2]

    out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                - idx):np.int(final_win_loc[-1, 2])]

# If no error raised (not beyond or below range of values)
else:
    # Green Channel values
    out_win_vals[:, 1] = hist_norm[np.int(
            final_win_loc[0, 1] - idx):np.int(
            final_win_loc[-1, 1] + idx), 1]

    out_binlocs[:,1] = bin_c[np.int(
            final_win_loc[0, 1] - idx):np.int(
            final_win_loc[-1, 1] + idx)]

    # Red Channel values
    out_win_vals[:, 2] = hist_norm[np.int(
            final_win_loc[0, 2] - idx):np.int(
            final_win_loc[-1, 2] + idx), 2]

    out_binlocs[:,2] = bin_c[np.int(
            final_win_loc[0, 2] - idx):np.int(
```

```
                                final_win_loc[-1, 2] + idx)]

            # Store resulting arrays as a pd Series object
            out = pd.Series((out_binlocs, out_win_vals))

            # Return resulting arrays
            return(out)
    # If Red & Blue channels present
    elif np.any(final_win_loc[:, 0]) != 0:
        # Initialize index for Colour Channels
        idx = 1
        cdx = 1
        # Evaluate Red channel's CDF
        ur = win_vals[:, 2] / 32
        vr = over_vals[:, 2] / 32

        # Evaluate difference of plain & overlapping CDF's
        Tks_r = np.max(np.abs(ur - vr))

        # Evaluate Blue Channel's CDF
        ub = win_vals[:, 0] / 32
        vb = over_vals[:, 0] / 32

        # Evaluate difference of plain & overlapping CDF's
        Tks_b = np.max(np.abs(ub - vb))

        # If Alternative Hypothesis (H1) true
        if Tks_r > 0.5 and Tks_b > 0.5:
            # Initialize arrays
            out_binlocs = np.zeros((32, 3), dtype=np.uint8)
            out_win_vals = np.zeros((32, 3), dtype=np.float32)

            # Utilize alternative hypothesis function
```

```
    (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                          , final_win_vals[:, :])

    # Return resulting arrays as a pandas Series object
    out = pd.Series((out_binlocs, out_win_vals))

    # Return output
    return(out)

# If Null Hypothesis (H0) true
else:
    while Tks_r < 0.5 or Tks_b < 0.5:
        # Initialize arrays to expand Red & Blue Channels values
        temp_win = np.zeros((32 + idx, 3), dtype=np.float32)
        temp_over = np.zeros((32 + idx, 3), dtype=np.float32)

        # If at beginning of histogram values(first window)
        if np.logical_or(final_win_loc[0, 2] == 0,
                         final_win_loc[0, 0] == 0):
            if idx <= 15:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[np.abs(15 - idx):46, 2]

                # Blue Channel Windows
                temp_win[:, 0] = hist_norm[0:32+idx, 0]
                temp_over[:, 0] = hist_norm[np.abs(15 - idx):46, 0]
            else:
                # Red Channel Windows
                temp_win[:, 2] = hist_norm[0:32+idx, 2]
                temp_over[:, 2] = hist_norm[0:46+cdx, 2]

                # Blue Channel Windows
```

```
        temp_win[:, 0] = hist_norm[0:32+idx, 0]
        temp_over[:, 0] = hist_norm[0:46+cdx, 0]
        # Increment index
        cdx += 1

# If at the end of histogram values (last window)
elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                final_win_loc[0, 0] == 224,
                final_win_loc[0, 0] == 239,
                final_win_loc[0, 2] == 239)):

    # Red Channel Windows
    temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                -idx: np.int(win_binloc[-1, 2])+1, 2]

    temp_over[:, 2] = np.pad(hist_norm[np.int(over_binloc[0, 2])
    -idx: np.int(over_binloc[16, 2]), 2],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[0, 2])
    -idx: np.int(over_binloc[16, 2]), 2])), 'constant')

    # Blue Channel Windows
    temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                -idx: np.int(win_binloc[-1, 0])+1, 0]

    temp_over[:, 0] = np.pad(hist_norm[np.int(over_binloc[ 0, 0])
                -idx : np.int(over_binloc[16, 0]), 0],
    (0, len(temp_over) - len(hist_norm[np.int(over_binloc[ 0, 0])
                -idx : np.int(over_binloc[16, 0]), 0])),
    'constant')

else:

    # Red Channel Windows
```

```
            temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                    : np.int(win_binloc[-1, 2]) + idx, 2]

            temp_over[:, 2] = hist_norm[np.int(over_binloc[i, 0, 2])
                        -idx : np.int(over_binloc[-1, 2]) , 2]

            # Blue Channel Windows
            temp_win[:, 0] = hist_norm[np.int(win_binloc[0, 0])
                    : np.int(win_binloc[-1, 0]) + idx, 0]

            temp_over[:, 0] = hist_norm[np.int(over_binloc[0, 0])
                    -idx : np.int(over_binloc[-1, 0]), 0]

        # Re Evaluate Red channel
        ur = temp_win[:, 2] / len(temp_win)
        vr = temp_over[:, 2] / len(temp_over)
        Tks_r = np.max(np.abs(ur - vr))

        # Re Evaluate Blue Channel
        ub = temp_win[:, 0] / len(temp_win)
        vb = temp_over[:, 0] / len(temp_over)
        Tks_b = np.max(np.abs(ub - vb))

        # Increase index value for next iteration
        idx += 1

    # Initialize output arrays
    out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

    # If current window is first window
    if final_win_loc[0, 0] == 0:
        # Blue Channel values
```

```
            out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                                           np.int(final_win_loc[-1, 0] + idx), 0]

            out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                                     np.int(final_win_loc[-1, 0] + idx)]

            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                                           np.int(final_win_loc[-1, 2] + idx), 2]

            out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                                     np.int(final_win_loc[-1, 2] + idx)]

        # If current window is last window
        elif np.logical_or.reduce((final_win_loc[0, 2] == 224 ,
                                   final_win_loc[0, 0] == 224,
                                   final_win_loc[0, 0] == 239,
                                   final_win_loc[0, 2] == 239)):
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                                 - idx):np.int(final_win_loc[-1, 0]), 0]

            out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
                                 - idx):np.int(final_win_loc[-1, 0])]

            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                                 - idx):np.int(final_win_loc[-1, 2]), 2]

            out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                                 - idx):np.int(final_win_loc[-1, 2])]

        # If current window any window except first or last
```

```
else:

    # Check if index reaches before beginning of values
    try:
        bin_c[np.int(final_win_loc[0, 0] - idx)] >= 0
    # If Error raised
    except:
        # Blue Channel values
        out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]):
                            np.int(final_win_loc[-1, 0] + idx), 0]

        out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]):
                            np.int(final_win_loc[-1, 0] + idx)]

        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                            np.int(final_win_loc[-1, 2] + idx), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                            np.int(final_win_loc[-1, 2] + idx)]

    # If no errors occurred (not in the beginning of values)
    # check index reaches beyond the end of values
    else:
        try:
            bin_c[np.int(final_win_loc[-1, 0] + idx)] >= 255
        # If Error raised (beyond end of values)
        except:
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(final_win_loc[0, 0]
                            -idx):np.int(final_win_loc[-1, 0]), 0]

            out_binlocs[:,0] = bin_c[np.int(final_win_loc[0, 0]
```

```
                    -idx):np.int(final_win_loc[-1, 0])]

            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2]), 2]

            out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2])]

        # If no error raised (not beyond or below range of values)
        else:
            # Blue Channel values
            out_win_vals[:, 0] = hist_norm[np.int(
                    final_win_loc[0, 0] - idx):np.int(
                    final_win_loc[-1, 0] + idx), 0]

            out_binlocs[:,0] = bin_c[np.int(
                    final_win_loc[0, 0] - idx):np.int(
                    final_win_loc[-1, 0] + idx)]

            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(
                    final_win_loc[0, 2] - idx):np.int(
                    final_win_loc[-1, 2] + idx), 2]

            out_binlocs[:,2] = bin_c[np.int(
                    final_win_loc[0, 2] - idx):np.int(
                    final_win_loc[-1, 2] + idx)]

# Store resulting arrays as a pd Series object
out = pd.Series((out_binlocs, out_win_vals))

# Return resulting arrays
```

```
        return(out)
# If only Red channel present
else:
    # Initialize index for Colour Channels
    idx = 1
    cdx = 1
    # Evaluate Red channel's CDF
    ur = win_vals[:, 2] / 32
    vr = over_vals[:, 2] / 32

    # Evaluate difference of plain & overlapping CDF's
    Tks_r = np.max(np.abs(ur - vr))

    # If Alternative Hypothesis (H1) true
    if Tks_r > 0.5:
        # Initialize arrays
        out_binlocs = np.zeros((32, 3), dtype=np.uint8)
        out_win_vals = np.zeros((32, 3), dtype=np.float32)

        # Utilize alternative hypothesis function
        (out_binlocs, out_win_vals) = alt_hyp(final_win_loc[:, :]
                                    , final_win_vals[:, :])

        # Return resulting arrays as a pandas Series object
        out = pd.Series((out_binlocs, out_win_vals))

        # Return output
        return(out)

    # If Null Hypothesis (H0) true
    else:
        while Tks_r < 0.5:
            # Initialize arrays to expand Red Channel values
```

```
temp_win = np.zeros((32 + idx, 3), dtype=np.float32)
temp_over = np.zeros((32 + idx, 3), dtype=np.float32)

# If at beginning of histogram values(first window)
if final_win_loc[0, 2] == 0:
    if idx <= 15:
        # Red Channel Windows
        temp_win[:, 2] = hist_norm[0:32+idx, 2]
        temp_over[:, 2] = hist_norm[np.abs(15 - idx):46, 2]
    else:
        # Red Channel Windows
        temp_win[:, 2] = hist_norm[0:32+idx, 2]
        temp_over[:, 2] = hist_norm[0:46+cdx, 2]
        # Increment index
        cdx += 1

# If at the end of histogram values (last window)
elif final_win_loc[0, 2] == 224 or final_win_loc[0,2] == 239:

    # Red Channel Windows
    temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
                    -idx: np.int(win_binloc[-1, 2])+1, 2]

    temp_over[:, 2] = np.pad(hist_norm[np.int(over_binloc[0, 2])
    -idx: np.int(over_binloc[16, 2]), 2],
(0, len(temp_over) - len(hist_norm[np.int(over_binloc[0, 2])
    -idx: np.int(over_binloc[16, 2]), 2])), 'constant')

else:

    # Red Channel Windows
    temp_win[:, 2] = hist_norm[np.int(win_binloc[0, 2])
              : np.int(win_binloc[-1, 2]) + idx, 2]
```

```
            temp_over[:, 2] = hist_norm[np.int(over_binloc[i, 0, 2])
                                        -idx : np.int(over_binloc[-1, 2]), 2]


        # Re Evaluate Red channel
        ur = temp_win[:, 2] / len(temp_win)
        vr = temp_over[:, 2] / len(temp_over)
        Tks_r = np.max(np.abs(ur - vr))

        # Increase index value for next iteration
        idx += 1

    # Initialize output arrays
    out_binlocs = np.zeros((32 + idx, 3), dtype=np.uint8)
    out_win_vals = np.zeros((32 + idx, 3), dtype=np.float32)

    # If current window is first window
    if final_win_loc[0, 2] == 0:
        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                                       np.int(final_win_loc[-1, 2] + idx), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                                 np.int(final_win_loc[-1, 2] + idx)]

    # If current window is last window
    elif final_win_loc[0, 2] == 224 or final_win_loc[0,2] == 239:
        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                                       - idx):np.int(final_win_loc[-1, 2]), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
```

```
                - idx):np.int(final_win_loc[-1, 2])]

# If current window any window except first or last
else:

    # Check if index reaches before beginning of values
    try:
        bin_c[np.int(final_win_loc[0, 2] - idx)] >= 0
    # If Error raised
    except:
        # Red Channel values
        out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx), 2]

        out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]):
                        np.int(final_win_loc[-1, 2] + idx)]

    # If no errors occurred (not in the beginning of values)
    # check index reaches beyond the end of values
    else:
        try:
            bin_c[np.int(final_win_loc[-1, 2] + idx)] >= 255
        # If Error raised (beyond end of values)
        except:
            # Red Channel values
            out_win_vals[:, 2] = hist_norm[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2]), 2]

            out_binlocs[:,2] = bin_c[np.int(final_win_loc[0, 2]
                        - idx):np.int(final_win_loc[-1, 2])]

        # If no error raised (not beyond or below range of values)
        else:
```

```
                    # Red Channel values
                    out_win_vals[:, 2] = hist_norm[np.int(
                        final_win_loc[0, 2] - idx):np.int(
                        final_win_loc[-1, 2] + idx), 2]

                    out_binlocs[:,2] = bin_c[np.int(
                        final_win_loc[0, 2] - idx):np.int(
                        final_win_loc[-1, 2] + idx)]

                # Store resulting arrays as a pd Series object
                out = pd.Series((out_binlocs, out_win_vals))

                # Return resulting arrays
                return(out)
        # If no colour channel present return zeros
        else:
            # Initialize arrays
            out_binlocs = np.zeros((32, 3), dtype=np.uint8)
            out_win_vals = np.zeros((32, 3), dtype=np.float32)

            # Store resulting arrays as a pd Series object
            out = pd.Series((out_binlocs, out_win_vals))

            # Return resulting arrays
            return(out)



# Initialize lists to output results
out_binlocs_l = []
out_win_vals_l = []

# Initialize number of clusters
```

```
n_clusters = 2    # Starting from 2 since background is a cluster & possible
                  # artefacts

# Enumerate histogram bins
bin_c = np.arange(0, 256, dtype=np.uint8)

for i in range(8):
    # If all channels present
    if np.logical_and.reduce((np.any(final_win_loc[i, :, 0]) != 0,
                              np.any(final_win_loc[i, :, 1]) != 0,
                              np.any(final_win_loc[i, :, 2]) != 0)):
        # Initialize lists to store values
        out_binlocs_list = []
        out_win_vals_list = []
        # Append bin locations
        out_binlocs_list.append(final_win_loc[i, :, :])

        # Append windows values
        out_win_vals_list.append(final_win_vals[i, :, :])

        # Convert resulting lists to pandas Series objects
        binlocs_pd = pd.Series(out_binlocs_list)
        win_vals_pd = pd.Series(out_win_vals_list)

        # Clear lists
        out_binlocs_list.clear()
        out_win_vals_list.clear()

        # Store results to arrays
        out_binlocs_temp = binlocs_pd[0]
        out_win_vals_temp = win_vals_pd[0]

        # Check if current windows have the same bin location for each
```

```
# Colour channel
if np.logical_and(out_binlocs_temp[:, 0].all() == out_binlocs_temp[:, 1].all(),
                  out_binlocs_temp[:, 1].all() == out_binlocs_temp[:, 2].all()):

    out_binlocs_l.append(out_binlocs_temp)
    out_win_vals_l.append(out_win_vals_temp)

    # Increment number of clusters by one
    n_clusters += 1

# If the bin location is not the same for any colour channel
else:
    # Perform KS-test
    out_binlocs, out_win_vals = ks_test_tree(win_vals[i, :, :],
        win_binloc[i, :, :], over_vals[i, :, :], over_binloc[i, :, :],
        hist_norm,final_win_vals[i, :, :],final_win_loc[i, :, :], bin_c)

    out_binlocs_l.append(out_binlocs)
    out_win_vals_l.append(out_win_vals)

    # Check if bin locations remain the same
    if np.logical_and(out_binlocs[:, 0].all() == out_binlocs[:, 1].all(),
                      out_binlocs[:, 1].all() == out_binlocs[:, 2].all()):
        # Increment number of clusters by 1
        n_clusters += 1
    # Check if Blue & Green bin locations same, but not Red's
    elif np.logical_and(out_binlocs[:, 0].all() == out_binlocs[:, 1].all(),
                        out_binlocs[:, 1].all() != out_binlocs[:, 2].all()):
        # Increment number of clusters by 2
        n_clusters += 2

    # Check if Blue & Red bin locations same, but not Green's
    elif np.logical_and(out_binlocs[:, 0].all() == out_binlocs[:, 2].all(),
```

```
                    out_binlocs[:, 2].all() != out_binlocs[:, 1].all()):
                # Increment number of cluster by 2
                n_clusters += 2

            # Check if Red & Green bin locations same, but not Blue's
            elif np.logical_and(out_binlocs[:, 1].all() == out_binlocs[:, 2].all(),
                    out_binlocs[:, 2].all() != out_binlocs[:, 0].all()):
                # Increment number of clusters by 2
                n_clusters += 2

            # If every colour's bin locations unequal
            else:
                # Increment number of clusters by 3
                n_clusters += 3

    # If Blue channel present
    elif np.any(final_win_loc[i, :, 0]) != 0:
    # If Blue and Red channels present
        if np.any(final_win_loc[i, :, 2]) != 0:
            # Initialize lists to store values
            out_binlocs_list = []
            out_win_vals_list = []

            # Append bin locations
            out_binlocs_list.append(final_win_loc[i, :, :])

            # Append windows values
            out_win_vals_list.append(final_win_vals[i, :, :])

            # Convert resulting lists to pandas Series objects
            binlocs_pd = pd.Series(out_binlocs_list)
            win_vals_pd = pd.Series(out_win_vals_list)
```

```
# Clear lists
out_binlocs_list.clear()
out_win_vals_list.clear()

# Store results to arrays
out_binlocs_temp = binlocs_pd[0]
out_win_vals_temp = win_vals_pd[0]

# Check if current windows have the same bin location for each
# Colour channel
if out_binlocs_temp[:, 0].all() == out_binlocs_temp[:, 2].all():

    out_binlocs_l.append(out_binlocs_temp)
    out_win_vals_l.append(out_win_vals_temp)

    # Increment number of clusters by one
    n_clusters += 1

# If the bin location is not the same for any colour channel
else:
    # Perform KS-test
    out_binlocs, out_win_vals = ks_test_tree(win_vals[i, :, :],
    win_binloc[i, :, :], over_vals[i, :, :], over_binloc[i, :, :],
    hist_norm,final_win_vals[i, :, :],final_win_loc[i, :, :], bin_c)

    out_binlocs_l.append(out_binlocs)
    out_win_vals_l.append(out_win_vals)

    # Check if bin locations remain the same
    if out_binlocs[:, 0].all() == out_binlocs[:, 2].all():
        # Increment number of clusters by 1
        n_clusters += 1
    # If Blue & Red colour bin locations unequal
```

```
            else:
                # Increment number of clusters by 2
                n_clusters += 2
    # If Blue & Green channels present
    elif np.any(final_win_loc[i, :, 1]) != 0:
        # Initialize lists to store values
        out_binlocs_list = []
        out_win_vals_list = []

        # Append bin locations
        out_binlocs_list.append(final_win_loc[i, :, :])

        # Append windows values
        out_win_vals_list.append(final_win_vals[i, :, :])

        # Convert resulting lists to pandas Series objects
        binlocs_pd = pd.Series(out_binlocs_list)
        win_vals_pd = pd.Series(out_win_vals_list)

        # Clear lists
        out_binlocs_list.clear()
        out_win_vals_list.clear()

        # Store results to arrays
        out_binlocs_temp = binlocs_pd[0]
        out_win_vals_temp = win_vals_pd[0]

        # Check if current windows have the same bin location for each
        # Colour channel
        if out_binlocs_temp[:, 0].all() == out_binlocs_temp[:, 1].all():

            out_binlocs_l.append(out_binlocs_temp)
            out_win_vals_l.append(out_win_vals_temp)
```

```
            # Increment number of clusters by one
            n_clusters += 1

        # If the bin location is not the same for any colour channel
        else:
            # Perform KS-test
            out_binlocs, out_win_vals = ks_test_tree(win_vals[i, :, :],
            win_binloc[i, :, :], over_vals[i, :, :], over_binloc[i, :, :],
            hist_norm,final_win_vals[i, :, :],final_win_loc[i, :, :], bin_c)

            out_binlocs_l.append(out_binlocs)
            out_win_vals_l.append(out_win_vals)

            # Check if bin locations remain the same
            if out_binlocs[:, 0].all() == out_binlocs[:, 1].all():
                # Increment number of clusters by 1
                n_clusters += 1
            # If Blue & Red colour bin locations unequal
            else:
                # Increment number of clusters by 2
                n_clusters += 2

    # If only Blue channel present
    else:
        # Initialize lists to store values
        out_binlocs_list = []
        out_win_vals_list = []

        # Append bin locations
        out_binlocs_list.append(final_win_loc[i, :, :])

        # Append windows values
```

```
            out_win_vals_list.append(final_win_vals[i, :, :])

            # Convert resulting lists to pandas Series objects
            binlocs_pd = pd.Series(out_binlocs_list)
            out_win_vals_pd = pd.Series(out_win_vals_list)

            # Clear lists
            out_binlocs_list.clear()
            out_win_vals_list.clear()

            # Store results to arrays
            out_binlocs_temp = binlocs_pd[0]
            out_win_vals_temp = out_win_vals_pd[0]

            out_binlocs_l.append(out_binlocs_temp)
            out_win_vals_l.append(out_win_vals_temp)

            # Increment number of clusters by 1
            n_clusters += 1
    # If Green channel present
    elif np.any(final_win_loc[i, :, 1]) != 0:
        # If Green & Blue channels present
        if np.any(final_win_loc[i, :, 0]) != 0:
            # Initialize lists to store values
            out_binlocs_list = []
            out_win_vals_list = []

            # Append bin locations
            out_binlocs_list.append(final_win_loc[i, :, :])

            # Append windows values
            out_win_vals_list.append(final_win_vals[i, :, :])
```

```
# Convert resulting lists to pandas Series objects
binlocs_pd = pd.Series(out_binlocs_list)
out_win_vals_pd = pd.Series(out_win_vals_list)

# Clear lists
out_binlocs_list.clear()
out_win_vals_list.clear()

# Store results to arrays
out_binlocs_temp = binlocs_pd[0]
out_win_vals_temp = out_win_vals_pd[0]

# Check if current windows have the same bin location for each
# Colour channel
if out_binlocs_temp[:, 0].all() == out_binlocs_temp[:, 1].all():

    out_binlocs_l.append(out_binlocs_temp)
    out_win_vals_l.append(out_win_vals_temp)
    # Increment number of clusters by one
    n_clusters += 1

# If the bin location is not the same for any colour channel
else:
    # Perform KS-test
    out_binlocs, out_win_vals = ks_test_tree(win_vals[i, :, :],
    win_binloc[i, :, :], over_vals[i, :, :], over_binloc[i, :, :],
    hist_norm,final_win_vals[i, :, :],final_win_loc[i, :, :], bin_c)

    out_binlocs_l.append(out_binlocs)
    out_win_vals_l.append(out_win_vals)

    # Check if bin locations remain the same
    if out_binlocs[:, 0].all() == out_binlocs[:, 1].all():
```

```
            # Increment number of clusters by 1
            n_clusters += 1
        # If Blue & Red colour bin locations unequal
        else:
            # Increment number of clusters by 2
            n_clusters += 2
# If Green & Red channels present
elif np.any(final_win_loc[i, :, 2]) != 0:
    # Initialize lists to store values
    out_binlocs_list = []
    out_win_vals_list = []

    # Append bin locations
    out_binlocs_list.append(final_win_loc[i, :, :])

    # Append windows values
    out_win_vals_list.append(final_win_vals[i, :, :])

    # Convert resulting lists to pandas Series objects
    binlocs_pd = pd.Series(out_binlocs_list)
    win_vals_pd = pd.Series(out_win_vals_list)

    # Clear lists
    out_binlocs_list.clear()
    out_win_vals_list.clear()

    # Store results to arrays
    out_binlocs_temp = binlocs_pd[0]
    out_win_vals_temp = win_vals_pd[0]

    # Check if current windows have the same bin location for each
    # Colour channel
    if out_binlocs_temp[:, 1].all() == out_binlocs_temp[:, 2].all():
```

```
            out_binlocs_l.append(out_binlocs_temp)
            out_win_vals_l.append(out_win_vals_temp)

            # Increment number of clusters by one
            n_clusters += 1

        # If the bin location is not the same for any colour channel
        else:
            # Perform KS-test
            out_binlocs, out_win_vals = ks_test_tree(win_vals[i, :, :],
            win_binloc[i, :, :], over_vals[i, :, :], over_binloc[i, :, :],
            hist_norm,final_win_vals[i, :, :],final_win_loc[i, :, :], bin_c)

            out_binlocs_l.append(out_binlocs)
            out_win_vals_l.append(out_win_vals)

            # Check if bin locations remain the same
            if out_binlocs[:, 1].all() == out_binlocs[:, 2].all():

                # Increment number of clusters by 1
                n_clusters += 1

            # If Green & Red colour bin locations unequal
            else:
                # Increment number of clusters by 2
                n_clusters += 2

    # If only Green channel present
    else:
        # Initialize lists to store values
        out_binlocs_list = []
        out_win_vals_list = []
```

```
            # Append bin locations
            out_binlocs_list.append(final_win_loc[i, :, :])

            # Append windows values
            out_win_vals_list.append(final_win_vals[i, :, :])

            # Convert resulting lists to pandas Series objects
            binlocs_pd = pd.Series(out_binlocs_list)
            win_vals_pd = pd.Series(out_win_vals_list)

            # Clear lists
            out_binlocs_list.clear()
            out_win_vals_list.clear()

            # Store results to arrays
            out_binlocs_temp = binlocs_pd[0]
            out_win_vals_temp = win_vals_pd[0]

            out_binlocs_l.append(out_binlocs_temp)
            out_win_vals_l.append(out_win_vals_temp)

            # Increment number of clusters by 1
            n_clusters += 1
    # If Red channel present
    elif np.any(final_win_loc[i, :, 2]) != 0:
        # If Red & Blue present
        if np.any(final_win_loc[i, :, 0]) != 0:
            # Initialize lists to store values
            out_binlocs_list = []
            out_win_vals_list = []

            # Append bin locations
```

```
out_binlocs_list.append(final_win_loc[i, :, :])

# Append windows values
out_win_vals_list.append(final_win_vals[i, :, :])

# Convert resulting lists to pandas Series objects
binlocs_pd = pd.Series(out_binlocs_list)
win_vals_pd = pd.Series(out_win_vals_list)

# Clear lists
out_binlocs_list.clear()
out_win_vals_list.clear()

# Store results to arrays
out_binlocs_temp = binlocs_pd[0]
out_win_vals_temp = win_vals_pd[0]

# Check if current windows have the same bin location for each
# Colour channel
if out_binlocs_temp[:, 0].all() == out_binlocs_temp[:, 2].all():

    out_binlocs_l.append(out_binlocs_temp)
    out_win_vals_l.append(out_win_vals_temp)

    # Increment number of clusters by one
    n_clusters += 1

# If the bin location is not the same for any colour channel
else:
    # Perform KS-test
    out_binlocs, out_win_vals = ks_test_tree(win_vals[i, :, :],
    win_binloc[i, :, :], over_vals[i, :, :], over_binloc[i, :, :],
    hist_norm,final_win_vals[i, :, :],final_win_loc[i, :, :], bin_c)
```

```
            out_binlocs_l.append(out_binlocs)
            out_win_vals_l.append(out_win_vals)

            # Check if bin locations remain the same
            if out_binlocs[:, 0].all() == out_binlocs[:, 2].all():
                # Increment number of clusters by 1
                n_clusters += 1
            # If Blue & Red colour bin locations unequal
            else:
                # Increment number of clusters by 2
                n_clusters += 2
    # If Red & Green channels present
    elif np.any(final_win_loc[i, :, 1]) != 0:
        # Initialize lists to store values
        out_binlocs_list = []
        out_win_vals_list = []

        # Append bin locations
        out_binlocs_list.append(final_win_loc[i, :, :])

        # Append windows values
        out_win_vals_list.append(final_win_vals[i, :, :])

        # Convert resulting lists to pandas Series objects
        binlocs_pd = pd.Series(out_binlocs_list)
        win_vals_pd = pd.Series(out_win_vals_list)

        # Clear lists
        out_binlocs_list.clear()
        out_win_vals_list.clear()

        # Store results to arrays
```

```
    out_binlocs_temp = binlocs_pd[0]
    out_win_vals_temp = win_vals_pd[0]

    # Check if current windows have the same bin location for each
    # Colour channel
    if out_binlocs_temp[:, 2].all() == out_binlocs_temp[:, 1].all():

        out_binlocs_l.append(out_binlocs_temp)
        out_win_vals_l.append(out_win_vals_temp)

        # Increment number of clusters by one
        n_clusters += 1

    # If the bin location is not the same for any colour channel
    else:
        # Perform KS-test
        out_binlocs, out_win_vals = ks_test_tree(win_vals[i, :, :],
        win_binloc[i, :, :], over_vals[i, :, :], over_binloc[i, :, :],
        hist_norm,final_win_vals[i, :, :],final_win_loc[i, :, :], bin_c)

        out_binlocs_l.append(out_binlocs)
        out_win_vals_l.append(out_win_vals)

        # Check if bin locations remain the same
        if out_binlocs[:, 2].all() == out_binlocs[:, 1].all():
            # Increment number of clusters by 1
            n_clusters += 1
        # If Blue & Red colour bin locations unequal
        else:
            # Increment number of clusters by 2
            n_clusters += 2

# If only Red channel present
```

```
    else:
        # Initialize lists to store values
        out_binlocs_list = []
        out_win_vals_list = []

        # Append bin locations
        out_binlocs_list.append(final_win_loc[i, :, :])

        # Append windows values
        out_win_vals_list.append(final_win_vals[i, :, :])

        # Convert resulting lists to pandas Series objects
        binlocs_pd = pd.Series(out_binlocs_list)
        win_vals_pd = pd.Series(out_win_vals_list)

        # Clear lists
        out_binlocs_list.clear()
        out_win_vals_list.clear()

        # Store results to arrays
        out_binlocs_temp = binlocs_pd[0]
        out_win_vals_temp = win_vals_pd[0]

        out_binlocs_l.append(out_binlocs_temp)
        out_win_vals_l.append(out_win_vals_temp)

        # Increment number of clusters by 1
        n_clusters += 1
# If no colour channel present
else:
    # Continue to next windows
    continue
```

```
# Convert output lists into pandas Series objects
out_binlocs_pd = pd.Series(out_binlocs_l)

out_win_vals_pd = pd.Series(out_win_vals_l)

# Test if any results are present
try:
    out_binlocs_pd[0]
except:
    # If not present, set outputs to zero
    out_binlocs = 0
    out_win_vals = 0
else:
    # Initialize output arrays
    out_binlocs = np.zeros((np.int8(len(out_binlocs_pd)),
                            np.int8(len(out_binlocs_pd[0])), 3), dtype=np.uint8)

    out_win_vals = np.zeros((np.int8(len(out_binlocs_pd)),
                             np.int8(len(out_binlocs_pd[0])), 3), dtype=np.float32)

    # Loop & store each window
    for i in range(len(out_binlocs_pd)):
        out_binlocs[i, :, :] = out_binlocs_pd[i]

        out_win_vals[i, :, :] = out_win_vals_pd[i]

# If flag is set to 1, output the resulting windows
if deb_flg == 1:
    out = pd.Series((out_binlocs, out_win_vals, n_clusters))
    return(out)
# Else, return the number of estimated clusters
else:
    return(n_clusters)
```

Appendix L: Main function

This is the main function file that captures a stream of frames (i.e. video) from an externally connected USB camera, and calls the implemented functions in the appropriate order.

```
import numpy as np
import cv2 as cv
import my_functions as f
import os

# Initialize frame stream source
source = f.user_inpt()
cap = cv.VideoCapture(source)

# 3 by 3 rectangular window for morph operators
kernel = np.ones((9, 9), dtype=np.uint8)
# Zivkovic MOG
fgbg = cv.createBackgroundSubtractorMOG2(detectShadows = False)

# Better Results by increasing the influence of past frames (?)
#cv.BackgroundSubtractorMOG2.setHistory(fgbg, 1000)
# contours px size to accept
contour_size = 60

# Iteration flag to check if MOG2 needs to stop
mog_flg = 1

# Initialize frame array
prev_frame = np.zeros((480, 640, 3), dtype=np.uint8)
res2 = np.zeros((480, 640, 3), dtype=np.uint8)

# Initialize state of statistical tests
```

```
mog2_stat = False
dev_stat = False
# Initialize mask array
prev_fgmask = np.zeros((480, 640), dtype=np.uint8)
# Initialize k-means frame
res_frame = np.zeros((480, 640, 3), dtype=np.uint8)
# Initialize number of clusters
n_clusters = 0

# Initialize arrays of histograms of current & previous frame
curr_hist = np.zeros((256, 3), dtype=np.uint32)
prev_hist = np.zeros((256, 3), dtype=np.uint32)

# Debugging counter for how many frames the MOG2 has converged
count = 0

# Get current working directory
main_path = os.getcwd()

while True:
    # Capture frame-by-frame
    ret, frame = cap.read()

    # Get input frame dimensions
    width, height, col = frame.shape

    # If 10th frame
    if mog_flg == 10:

        # Evaluate histograms of past & current frame
        for i in range(col):
            curr_hist[:, i] = np.reshape(cv.calcHist(res2, [i], None, [256],
                    [0, 256]), 256)
```

```
            prev_hist[::, i] = np.reshape(cv.calcHist(prev_frame, [i], None,
                [256], [0, 256]), 256)

        # Filter current and previous frames histograms
        filt_curr = f.hist_bpf(curr_hist)
        filt_prev = f.hist_bpf(prev_hist)

        # Normalize current & previous frames filtered histograms
        norm_curr = f.hist_norm(filt_curr)
        norm_prev = f.hist_norm(filt_prev)

        # Evaluate if K-S test indicates the MOG2 has converged
        mog2_stat = f.stop_mog2(norm_curr, norm_prev)

        # Implement plain & overlapping windows
        (win_vals, win_binloc, over_vals, over_binloc) = f.windows(norm_curr)

        # Rule based tree for windows of interest
        (final_win_vals, final_win_loc) = f.rule_tree(win_vals,
                        win_binloc, over_vals, over_binloc)

        # Adaptive windows via means of K-S statistical test
        _, out_win_vals, n_clusters = f.adapt_win(final_win_vals, final_win_loc,
norm_curr,
            win_vals, win_binloc, over_vals, over_binloc, deb_flg = 1)

        # Evaluate the result of the STD test
        dev_stat = f.hist_deviation(out_win_vals)

        # If MOG2 has converged
        if (dev_stat or mog2_stat) and n_clusters > 2:

            # Image processing function
```

```
            (res2, res, contours) = f.frame_proc(frame, prev_fgmask, kernel, contour_size)

            # Implement K-means clustering to segment detected objects
            res_frame = f.kmeans_cv(frame, n_clusters)
            # Increment counter by one
            count += 1
        else:
            #apply MOG2 and get foreground pixels
            fgmask = fgbg.apply(frame)
            (res2, res, contours) = f.frame_proc(frame, fgmask, kernel, contour_size)

    else:
        # If MOG2 has not converge
        if dev_stat == False and mog2_stat == False:
            #apply MOG2 and get foreground pixels
            fgmask = fgbg.apply(frame)
            (res2, res, contours) = f.frame_proc(frame, fgmask, kernel, contour_size)

        # Apply previous mask on the current frame
        else:
            (res2, res, contours) = f.frame_proc(frame, prev_fgmask, kernel, contour_size)
            # If number of clusters greater than two
            if n_clusters > 2:
                # Implement K-means clustering to segment detected objects
                res_frame = f.kmeans_cv(frame, n_clusters)

    # Show resulting frames
    cv.imshow('Foreground', res2)
    cv.drawContours(res, contours, -1, (0, 0, 255), 2)
    cv.imshow('Contours', res)
    cv.imshow('Raw Frame', frame)
```

```
    cv.imshow('K-Means results', res_frame)



    # Store current frame for next iteration
    if mog_flg == 1:
        prev_frame = res2

        # Increment flag for next iteration
        mog_flg += 1

        # If MOG2 has converged keep current mask
        if mog2_stat:
            prev_fgmask = fgmask

    # Reset flag for next iteration
    elif mog_flg == 10:
        mog_flg = 1

    else:
        # Increment by 1 the flag
        mog_flg += 1

    # To break, press the q key
    if cv.waitKey(1) & 0xFF == ord('q'):
        break

# Release capture & close all windows
cap.release()
cv.destroyAllWindows()
```