E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Optimizing the fuzzy transform using key points for image compression

Grado en Ingeniería Informática

Trabajo Fin de Grado

Mikel Galafate de la Llave

Daniel Paternain Dallo

Mikel Sesma Sara

Pamplona, 07/09/2021

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Abstract

In this project we work with the F-Transform and its application to image compression. With images that contain a single object with a constant background as a target, the main idea is to try to make a better approximation of the inverse F-Transform finding a better partition for each image applying a prior key point detection so that the F-Transform can be focused on the objects in the image. We make an optimization of the code for both the F-Transform and the approximation of its inverse, propose a method to find a better fuzzy partition and optimize it based on the magnitude of the first derivative of the image and lastly propose different methods that separate the object from the image to try to improve the results.

Keywords: Fuzzy Transform; Image compression; Fuzzy partition; F-Transform; Compression ratio

Contents

Introduction

Image compression is a very important technique in our daily life. The amount of data we need to store is growing every day and, even if the devices we use are evolving and have gone from a couple of kilobytes to hundreds of gigabytes and even terabytes, if we stored the original files without changing anything on them, the space we would need would be immense. Taking a picture with our phone and storing thousands of them in it, services using video stream e.g. video-on-demand services such as Netflix, HBO, Amazon Prime Video, etc. are some examples where image compression is needed. The use of streaming services would be impossible to carry by the providers with millions of simultaneous users. In fact, this kind of services were only possible thanks to image compression.

In order to reduce the amount of space needed for an image, there are two main techniques. These are image reduction, also called image scaling, and image compression.

The pursuit of image reduction is to reduce the spacial resolution in order to obtain the same image with a smaller amount of pixels. For example, turning a 1024×1024 image into a 512×512 image, and thus, making the file 4 times smaller, by means of different methods that take a neighbourhood of pixels and turn it into a single pixel that represents all of them such as the arithmetic mean, the geometric mean, min, max, etc. These can then be reconstructed with the use of image scaling algorithms e.g. interpolation. Image compression, though, reduces the amount of space used to store the data by completely modifying the data of the file.

There is a big variety of image compression algorithms and they can be classified in two different categories, lossy compression and lossless compression algorithms. As the name suggests, the former, also called irreversible compression algorithms, lose some of the data of the original file in the process. Therefore, the used compression ratio [1] determines the quality of the resulting image. Higher compression ratios lead to a lower quality in the resulting image and lower compression ratios to a higher quality image. Contrarily, lossless compression algorithms can retrieve the original file on its completeness. In exchange for this, the compression ratio we can obtain with them is much lower. For some tasks, it is important to use algorithms that are capable to recover all the information. If we want to compress a text file, it is not possible to use lossy compression, since the data would not make sense at all. The same happens when we compress several files into a single zip, rar, tar, 7z, etc. file, as we then want to get back the same files we had before. For images, lossy

[1] The data compression ratio defined as: $\text{CR} = \frac{\text{Uncompressed size}}{\text{Compressed size}}$. For a file compressed from size 45 to 5, both 9:1 and 9 are equivalent.

compression is preferred instead because the lost details are not very remarkable for the human eye, yet the achieved compression ratio is much higher in comparison with the lost data. A very well known lossy method for image compression is jpeg or jpg, based on the discrete cosine transform (DCT).

However, we do not always use lossy compression for images, for instance, if we are working with an image and cannot afford the loss of data because we need to work with it later. In this case, a well known algorithm that does not lose any detail of the image is png.

In this case, we are going to work with images and we are going to apply the Fuzzy Transform (shortened F-Transform) method to them, which is a lossy algorithm in between both image reduction and image compression concepts since, it does apply an image reduction but it can also change the structure of the original data in some way.

Originally, the F-Transform is computed by first fixing a number of nodes that will be the dimensions of our new image. This nodes are evenly distributed all along the image creating a grid over the image *Fig. 1*. This means that, when the new pixels are computed, all parts of the image are compressed uniformly with an even loss of data. Nevertheless, we know there are some kind of images where there is a specific part containing most of the important information. In *Fig. 2*, the bird contains the important information whereas, the background is almost constant and it is not as important. Therefore, if we compress the image using a lossy algorithm, it would be preferable that the lost data were part of the background keeping the maximum possible quality in the object of interest, the bird in this case.

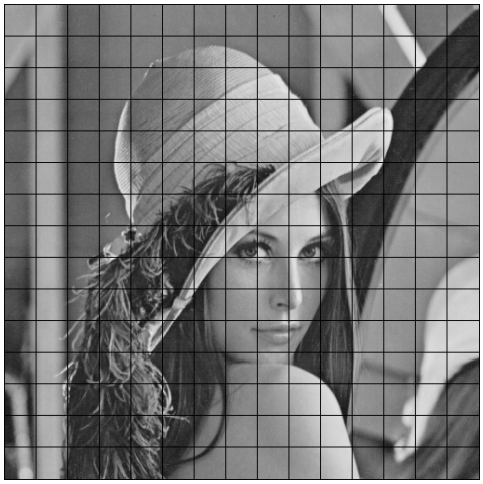


Fig. 1 Uniform grid over original image for a reduction from 512×512 to 16×16.

Fig. 2 Image with constant background and object of interest.

This Final Degree Project consists in developing an image compression algorithm trying to reduce the size of these while preserving the main properties of the previously described type of images. The algorithm is based on the method of the fuzzy transform but it performs a prior key point detection finding the object of interest to focus the attention of the fuzzy partition on it. The objective of our proposal is to keep as much information as possible from the original image, thus minimizing the error when applying the inverse transformation.

The project has been carried out in three different parts. First, working on the fuzzy transform and optimizing the algorithm. Second part is about optimizing the used nodes adapting them to the image with the first derivative of it. Lastly, part three to find the key points in the images to focus the transform in the objects.

Chapter 1

Fuzzy Transform

In this first chapter, we are going to introduce the concept of the Fuzzy Transform, give the definition and see an example of the application of the method. Then we are going to work on the optimization of the complexity of the code in order to speed up the calculations.

1.1 Definition of the Fuzzy Transform

Let us start by defining the concept of the Fuzzy Transform, a method for constructing approximations of functions based on the fuzzy logic introduced by L. Zadeh [1].

1.1.1 Fuzzy Partition

In order to define the Fuzzy Transform, we first need to introduce an important concept with which we are going to work throughout the whole project. The concept is a fuzzy partition of an interval $[a, b]$ we will consider as a universe.

Definition 1. Let $x_1 < \cdots < x_n$ be nodes within $[a, b]$, such that $x_1 = a$, $x_n = b$ and $n \geq 2$. We say that fuzzy sets $A_1, \ldots, A_n : [a, b] \to [0, 1]$ form a fuzzy partition of $[a, b]$ if they fulfill the following conditions for $k = 1, \ldots, n$: [2]

(1) $A_k(x_k) = 1$;

(2) $A_k(x) = 0$ if $x \notin (x_{k-1}, x_{k+1})$ where for the uniformity of denotation, we put $x_0 = a$ and $x_{n+1} = b$;

(3) $A_k(x)$ is continuous;

(4) $A_k(x)$, $k = 2, \ldots, n$, strictly increases on $[x_{k-1}, x_k]$ and $A_k(x)$, $k = 1, \ldots, n-1$ strictly increases on $[x_k, x_{k+1}]$;

(5) $\forall\, x \in [a, b]$, $\sum_{i=1}^{n} A_k(x) = 1$;

The membership functions $A_1, \ldots, A_n$ are called *basic functions*.

Let us remark that basic functions are specified by a set of nodes $x_1 < \cdots < x_n$ and the properties 1-5. The shape of basic functions is not predetermined and therefore, it can be chosen additionally according to further requirements (e.g. smoothness).

Basic functions are defined by the previous properties and some fixed nodes but that does not define the shape of the functions. For this project the used basic functions are defined by:

$$A_1(x) = \begin{cases} 1 - \frac{(x-x_1)}{h_1}, & x \in [x_1, x_2], \\ 0 & \text{otherwise} \end{cases}$$

$$A_k(x) = \begin{cases} \frac{(x-x_{k-1})}{h_{k-1}}, & x \in [x_{k-1}, x_k], \\ 1 - \frac{(x-x_k)}{h_k}, & x \in [x_k, x_{k+1}], \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

$$A_n(x) = \begin{cases} \frac{(x-x_{n-1})}{h_{n-1}}, & x \in [x_{n-1}, x_n], \\ 0 & \text{otherwise} \end{cases}$$

$$\text{where } k = 2, \ldots, n-1, \text{ and } h_k = x_{k+1} - x_k$$

Applying (1.1) to the image represented in *Fig. 1* scaling it from 512×512 to 16×16, the resulting fuzzy partition used to compute the F-Transform of the image would be the following:

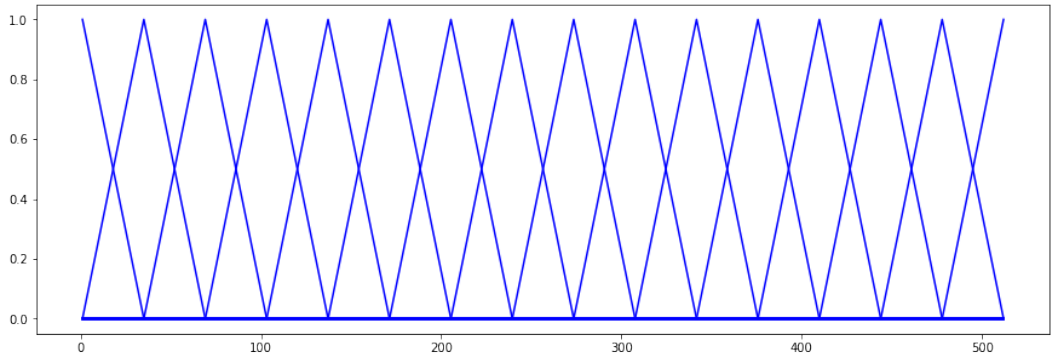


Fig. 1.1 Uniform fuzzy partition of the universe $[1, 512]$.

1.1.2 F-Transform

The F-Transform was introduced in [2] as a correspondence between a set of continuous functions on an interval $[a, b]$ and the set of n-dimensional vectors.

Let $C([a, b])$ be the set of continuous functions on the interval $[a, b]$.

Definition 2 Let $A_1, \ldots, A_n$ be basic functions which form a fuzzy partition of $[a, b]$ and f be any function from $C([a, b])$. We can say that the n-tuple of real

numbers $[F_1, \ldots, F_n]$ given by

$$F_k = \frac{\int_a^b f(x)A_k(x)\,dx}{\int_a^b A_k(x)\,dx} \tag{1.2}$$

is the F-Transform of f with respect to $A_1, \ldots, A_n$.

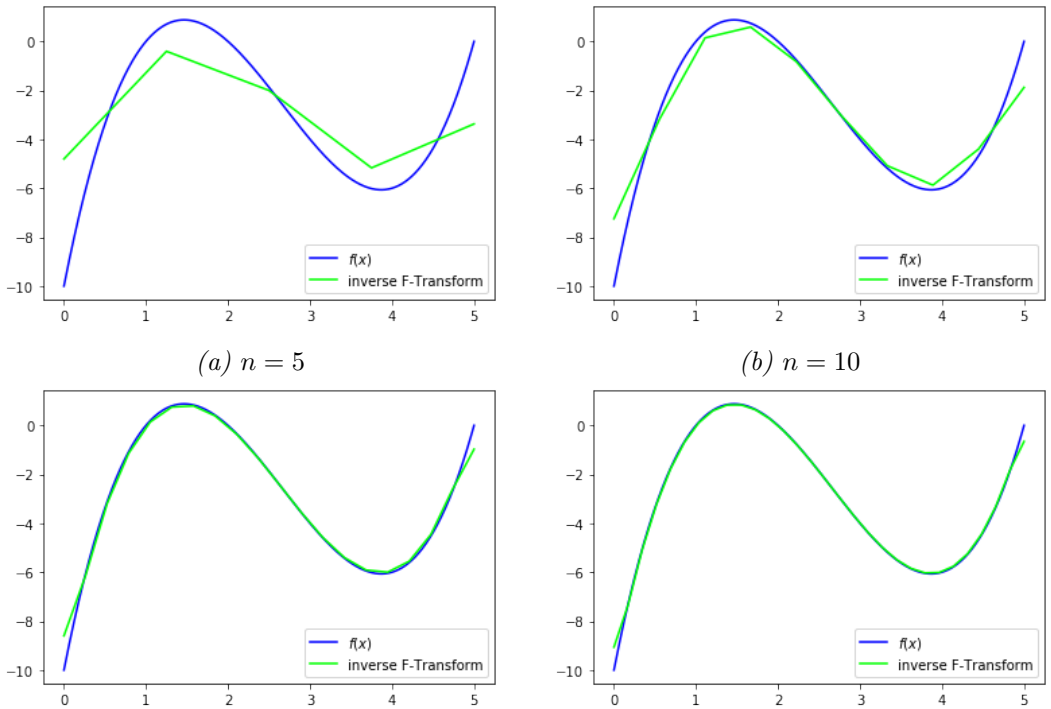
The inversion formula or inverse F-Transform, converts an n-dimensional vector into another continuous function approximating the original one.

Definition 3 Let $A_1, \ldots, A_n$ be basic functions which form a fuzzy partition of $[a, b]$ and f be a function from $C([a, b])$. Let $\mathbf{F}_n = [F_1, \ldots, F_n]$ be the F-Transform of f with respect to $A_1, \ldots, A_n$. Then the function given by

$$f_{F,n}(x) = \sum_{k=1}^{n} F_k A_k(x) \tag{1.3}$$

$\forall\, x \in [a, b]$ is called the inverse F-Transform.

It is obvious that the quality of the approximation of the F-Transform will depend on the number of nodes we use for the fuzzy partition. As we can see in *Fig. 1.2*, the more nodes we use, the better the approximation. In *Fig. 1.2(d)*, with $n = 30$, we see that the approximation almost exactly coincides with the original function.



(a) $n = 5$ *(b)* $n = 10$

(c) $n = 20$ *(d)* $n = 30$

Fig. 1.2 Different approximations of the function $f(x) = (x - 5)(x - 2)(x - 1)$ on the interval $[0, 5]$ given by the F-Transform with a different number of nodes n.

Considering a function f only defined on a set of points P and assuming P is

dense enough with respect to the used partition, then we can also define a discretized version of the F-Transform and its inverse F-Transform.

Definition 4 Let $f : [a, b] \to \mathbb{R}$ be a function given at points $p_1, \ldots, p_N \in [a, b]$ and let $A_1, \ldots, A_n$, $n < N$, be basic functions which form a fuzzy partition of $[a, b]$. We say that the n-tuple of real numbers $\mathbf{F}_n = [F_1, \ldots, F_n]$ is the discrete F-Transform of f with respect to $A_1, \ldots, A_n$ if

$$F_k = \frac{\sum_{i=1}^{n} f(p_i)A_k(p_i)}{\sum_{i=1}^{n} A_k(p_i)} \tag{1.4}$$

We can now define the inverse function for the discrete case.

Definition 5 Let $f : [a, b] \to \mathbb{R}$ be a function given at points $p_1, \ldots, p_N \in [a, b]$ and let $\mathbf{F}_n[f] = [F_1, \ldots, F_n]$ be the discrete F-Transform of f with respect to $A_1, \ldots, A_n$. Then the function

$$f_{F,n}(x) = \sum_{k=1}^{n} F_k A_k(x) \tag{1.5}$$

defined at $p_1, \ldots, p_N$ is called the discrete inverse F-Transform.

F-Transform of two or more variables

We can generalize the case of the one variable F-Transform and inverse F-Transform to two or more variables. Suppose the universe $[a, b] \times [c, d]$ and let $C([a, b] \times [c, d])$ be the set of continuous functions of two variables $f(x, y)$ on the interval $[a, b] \times [c, d]$

Definition 6 Let $A_1, \ldots, A_n$ be basic functions which form a fuzzy partition of $[a, b]$ and $B_1, \ldots, B_m$ be basic functions which form a fuzzy partition of $[c, d]$. Let $f(x, y)$ be any function from $C([a, b] \times [c, d])$. We say that the $n \times m$-matrix of real numbers $\mathbf{F}_{nm}[f] = F_{kl}$ is the integral F-Transform of f with respect to $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ if for each $k = 1, \ldots, n\,, l = 1, \ldots, m$,

$$F_{kl} = \frac{\int_c^d \int_a^b f(x, y)A_k(x)B_l(y)\,dx\,dy}{\int_c^d \int_a^b A_k(x)B_l(y)\,dx\,dy} \tag{1.6}$$

Discrete F-Transform. Application to image compression

Analogously to the one variable discrete case, for a function f only defined at some nodes $(p_i, q_j) \in [a, b] \times [c, d]$, we can define a discrete F-Transform for two or more variables. One of the applications of the F-Transform, and the one we are going to work with in this project, is the use of it for image compression. If we see an image as a discrete function of two variables f_I defined in $[1, N] \times [1, M]$, we can directly define the discrete F-Transform for image compression.

Definition 7 Let a function f be given at nodes $(p_i, q_j) \in [a, b] \times [c, d]\,, i = 1, \ldots, N\,, j = 1, \ldots, M$, and $A_1, \ldots, A_n$, $B_1, \ldots, B_m$ where $n < N$ and $m < M$, be basic functions which form fuzzy partitions of $[a, b]$ and $[c, d]$, respectively. Suppose

that sets P and Q of nodes are sufficiently dense with respect to the chosen partitions. We say that the $n \times m$-matrix of real numbers $\mathbf{F}_{nm}[f_I] = F_{kl}$ is the discrete F-Transform of f with respect to $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ if

$$F_{kl} = \frac{\sum_{j=1}^{M} \sum_{i=1}^{N} f_I(p_i, q_j) A_k(p_i) B_l(q_j)}{\sum_{j=1}^{M} \sum_{i=1}^{N} A_k(p_i) B_l(q_j)} \tag{1.7}$$

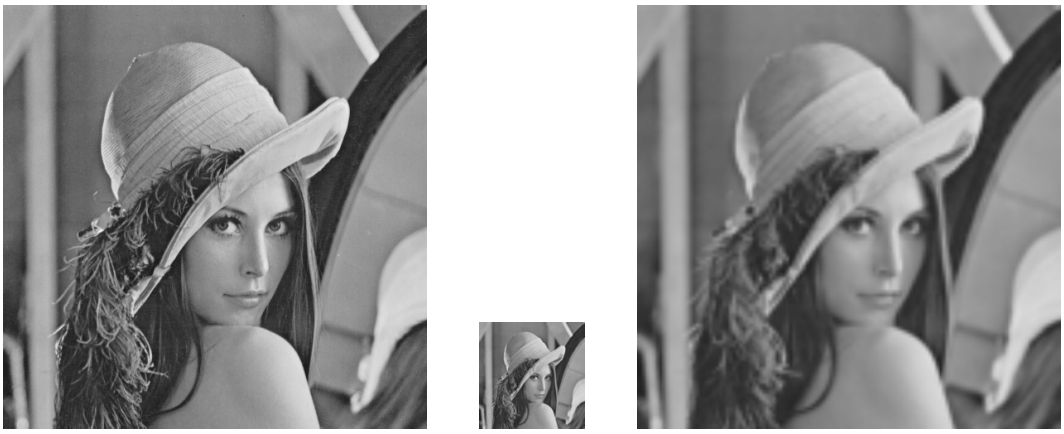
holds for all $k = 1, \ldots, n\,, l = 1, \ldots, m$.

This way, in order to recover an approximation of the original image, we can define the discrete inverse F-Transform.

Definition 8 Let $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ be basic functions which form fuzzy partitions of $[a, b]$ and $[c, d]$ respectively. Let f_I be a discrete function defined in $[1, N] \times [1, M]$ and $\mathbf{F}_{nm}[f_I]$ be the F-Transform of f_I with respect to $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$. Then the function

$$f^F_{nm}(p_i, q_j) = \sum_{k=1}^{n} \sum_{l=1}^{m} F_{kl} A_k(p_i) B_l(q_j) \tag{1.8}$$

is called the inverse F-Transform.



(a) *(b)* *(c)*

Fig. 1.3 The original image *(a)* is compressed with a 16 : 1 compression ratio *(b)*. The result of applying the inverse F-Transform is shown in *(c)*.

The example in *Fig. 1.3* is the result of applying the F-Transform to an image with a 16 : 1 compression ratio with a triangular fuzzy partition (eq. (1.1)). This means that the original 512×512 is reduced to a 128×128 image by means of eq. (1.7). The image in *Fig. 1.3(c)*, reconstructed with eq. (1.8), clearly shows that the F-Transform is a lossy compression algorithm since the loss of details and sharpness is obvious.

1.2 Code optimization

The previously described F-Transform in eq. (1.7) computes a single value of $\mathbf{F}$. For each value, two `for`-loops are needed that go through the image being

compressed and this must be done for every $(K, L) \in [1, n] \times [1, m]$ in matrix $\mathbf{F}$. This means the method needs 4 nested `for`-loops. This algorithm's complexity is therefore $\mathcal{O}(n^4)$ and we need to reduce it since such complexity is too high even when working with smaller images. We are now going to describe some methods that optimize the code of the F-Transform. Besides, there is also some optimization we can make to the algorithm in terms of computing the fuzzy partitions faster.

1.2.1 Redundancy in the calculations

As we have seen, to compute F_{KL} with eq. (1.7), the value of the fuzzy partition $A_K(p_i)$ is computed M times, once for each B_L and respectively, $B_L(q_j)$ N times for a given A_K. The same happens when we compute the inverse F-Transform eq. (1.8). We are going to try to find a way to avoid to do the same once and again.

F-Transform

To reduce the number of iterations, for a fixed F_{KL}, we are going to create an $N \times M$-matrix called $A_K B_L$ where $A_K B_L(i, j) = A_K(p_i) \cdot B_L(q_j)$. This way, once it is constructed, we can compute F_{KL} directly as the sum of the element-wise product[1] of the image and $A_K B_L$ divided by the sum of $A_K B_L$ (see eq. (1.9)) and thus, get rid of the two inner `for`-loops of the algorithm.

$$F_{KL} = \frac{\sum \sum im \odot A_K B_L}{\sum \sum A_K B_L} \tag{1.9}$$

Before constructing the matrix $A_K B_L$, we first need to define an $n \times N$-matrix A where $A[K, i] = A_K(p_i)$ and an $m \times M$-matrix B where $B[L, j] = B_L(q_j)$

$$A = \begin{pmatrix} A_1(p_1) & A_1(p_2) & \cdots & A_1(p_N) \\ A_2(p_1) & A_2(p_2) & \cdots & A_2(p_N) \\ \vdots & \vdots & \ddots & \vdots \\ A_n(p_1) & A_n(p_2) & \cdots & A_n(p_N) \end{pmatrix} B = \begin{pmatrix} B_1(q_1) & B_1(q_2) & \cdots & B_1(q_M) \\ B_2(q_1) & B_2(q_2) & \cdots & B_2(q_M) \\ \vdots & \vdots & \ddots & \vdots \\ B_m(q_1) & B_m(q_2) & \cdots & B_m(q_M) \end{pmatrix}$$

With these two matrices, for a given F_{KL} we construct $A_K B_L$ as $A_K B_L = A(K)^T \cdot B(L)$

$$A_K B_L = A(K)^T \cdot B(L) = \begin{pmatrix} A_K(p_1) \\ A_K(p_2) \\ \vdots \\ A_K(p_N) \end{pmatrix} \cdot \begin{pmatrix} B_L(q_1) & B_L(q_2) & \cdots & B_L(q_M) \end{pmatrix} =$$

$$= \begin{pmatrix} A_K(p_1) \cdot B_L(q_1) & A_K(p_1) \cdot B_L(q_2) & \cdots & A_K(p_1) \cdot B_L(q_M) \\ A_K(p_2) \cdot B_L(q_1) & A_K(p_2) \cdot B_L(q_2) & \cdots & A_K(p_2) \cdot B_L(q_M) \\ \vdots & \vdots & \ddots & \vdots \\ A_K(p_N) \cdot B_L(q_1) & A_K(p_N) \cdot B_L(q_2) & \cdots & A_K(p_N) \cdot B_L(q_M) \end{pmatrix}$$

[1] The element-wise or Hadamard product $\odot$

We can now apply eq. (1.9) to obtain the value of F_{KL}. By doing this we now do not need to use the `for`-loops that go through the image and only need to apply this for every (K, L) in order to obtain $\mathbf{F}_{nm}$.

Inverse F-Transform

The same way as with the F-Transform, to retrieve the original image with the inverse F-Transform with eq. (1.8), four `for`-loops are needed and we are going to get rid of the ones that, in this case, go through the matrix $\mathbf{F}_{nm}$, therefore removing the reiterative calculation of the fuzzy partitions.

We are again going to use the same A and B matrices but this time their transposed A^T and B^T matrices. The procedure is quite the same as with the F-Transform.

With these two matrices, for a given $f^F_{nm}(i, j)$ we construct an $n \times m$-matrix A_iB_j as $A_iB_j = A^T(i)^T \cdot B^T(j)$

$$A_iB_j = A^T(i)^T \cdot B^T(j) = \begin{pmatrix} A_1(p_i) \\ A_2(p_i) \\ \vdots \\ A_n(p_i) \end{pmatrix} \cdot \begin{pmatrix} B_1(q_j) & B_2(q_j) & \cdots & B_m(q_j) \end{pmatrix} =$$

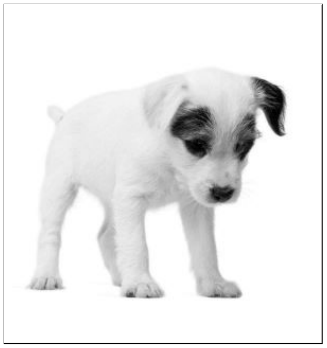
$$= \begin{pmatrix} A_1(p_i) \cdot B_1(q_j) & A_1(p_i) \cdot B_2(q_j) & \cdots & A_1(p_i) \cdot B_m(q_j) \\ A_2(p_i) \cdot B_1(q_j) & A_2(p_i) \cdot B_2(q_j) & \cdots & A_2(p_i) \cdot B_m(q_j) \\ \vdots & \vdots & \ddots & \vdots \\ A_n(p_i) \cdot B_1(q_j) & A_n(p_i) \cdot B_2(q_j) & \cdots & A_n(p_i) \cdot B_m(q_j) \end{pmatrix}$$

Now to compute $f^F_{nm}(i, j)$ we do the sum of the elements given by the result of the element-wise product of F and A_iB_j

$$f^F_{nm}(i, j) = \sum \sum F \odot A_iB_j \tag{1.10}$$

Results

We are going to compare the results of this new method with those of the original one. As we can see in *Fig. 1.4*, the Mean Squared Error (or MSE) of *Fig. 1.4(b)* and *Fig. 1.4(c)* are the same. Considering the original method as a reference, if we calculate the MSE of *Fig. 1.4(c)* with respect to *Fig. 1.4(b)* to see how similar they are, we see the MSE is $1.655 \cdot 10^{-26}$. This error is so small, we can consider them to be the same.



(a) Original image.

(b) Inverse with original method. $MSE = 208.833$

(c) Optimized inverse. $MSE = 208.833$

Fig. 1.4 Result of applying the two different introduced methods reducing (a) from 320×300 to 32×30 (100:1) and the Mean Squared Error.

1.2.2 Direct calculation

Based on section 1.2.1 we can find a quicker method that doesn't go through every F_{KL} but computes $\mathbf{F}_{nm}$ straightaway improving the algorithm and its complexity. This time, we are going to create a method that computes the F-Transform directly without using any `for`-loop.

F-Transform

In order to get rid of the `for`-loops, we are going to use matrices and their product. This time we are going to divide the task into two parts, first of all, calculate the $n \times m$-matrix F where $F(k,l) = \sum_{j=1}^{M} \sum_{i=1}^{N} f(i,j) \cdot A_k(p_i) \cdot B_l(q_j)$

Using the two same matrices A and B defined in section 1.2.1 we are going to calculate F as $F = A \cdot f \cdot B^T$ and we will get each value on it's position.

$$
\begin{aligned}
&F = A \cdot f \cdot B^T = \\
&= \begin{pmatrix} A_1(p_1) & \cdots & A_1(p_N) \\ \vdots & \ddots & \vdots \\ A_n(p_1) & \cdots & A_n(p_N) \end{pmatrix} \cdot \begin{pmatrix} f(1,1) & \cdots & f(1,M) \\ \vdots & \ddots & \vdots \\ f(N,1) & \cdots & f(N,M) \end{pmatrix} \cdot B^T = \\
&= \begin{pmatrix} \sum_{i=1}^{N} A_1(p_i) \cdot f(i,1) & \cdots & \sum_{i=1}^{N} A_i(p_i) \cdot f(i,M) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{N} A_1(p_i) \cdot f(i,1) & \cdots & \sum_{i=1}^{N} A_i(p_i) \cdot f(i,M) \end{pmatrix} \cdot \begin{pmatrix} B_1(q_1) & \cdots & B_m(q_1) \\ \vdots & \ddots & \vdots \\ B_1(q_M) & \cdots & B_m(q_M) \end{pmatrix} = \\
&= \begin{pmatrix} \sum_{j=1}^{M} \left(\sum_{i=1}^{N} A_1(p_i) \cdot f(i,j) \right) \cdot B_1(q_j) & \cdots & \sum_{j=1}^{M} \left(\sum_{i=1}^{N} A_1(p_i) \cdot f(i,j) \right) \cdot B_m(q_j) \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^{M} \left(\sum_{i=1}^{N} A_n(p_i) \cdot f(i,j) \right) \cdot B_1(q_j) & \cdots & \sum_{j=1}^{M} \left(\sum_{i=1}^{N} A_n(p_i) \cdot f(i,j) \right) \cdot B_m(q_j) \end{pmatrix}
\end{aligned}
$$

Once the numerator part of eq. (1.7) is done, we still need to calculate the denominator $\sum_{j=1}^{M} \sum_{i=1}^{N} A_k(p_i) B_l(q_j)$. We can think of it from the method explained in section 1.2.1.

We have that for a fixed F_{KL}

$$\begin{aligned} F_{KL} = \sum\sum A_K^T \cdot B_L &= a_{K1} \cdot b_{L1} + a_{K2} \cdot b_{L2} + \cdots + a_{KM} \cdot b_{LN} = \\ &= (a_{K1} + a_{k2} + \cdots + a_{KM}) \cdot (b_{L1} + b_{L2} + \cdots + b_{LM}) = \\ &= \sum A_K \cdot \sum B_L \end{aligned} \tag{1.11}$$

Once we have that $\sum\sum A_K^T \cdot B_L = \sum A_K \cdot \sum B_L$, we can generalize this to compute the whole matrix at once. We define the $n \times 1$-matrix $sA = A \cdot \mathbf{1}_N$ and $m \times 1$-matrix $sB = B \cdot \mathbf{1}_M$ which contain the row sum of each A and B.

$$sA = A \cdot \mathbf{1}_N = \begin{pmatrix} A_1(p_1) & \cdots & A_1(p_N) \\ \vdots & \ddots & \vdots \\ A_n(p_1) & \cdots & A_n(p_N) \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^N A_1(p_i) \\ \vdots \\ \sum_{i=1}^N A_n(p_i) \end{pmatrix}$$

$$sB = B \cdot \mathbf{1}_M = \begin{pmatrix} B_1(q_1) & \cdots & B_1(q_M) \\ \vdots & \ddots & \vdots \\ B_m(q_1) & \cdots & B_m(q_M) \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^M B_1(q_j) \\ \vdots \\ \sum_{j=1}^M B_m(q_j) \end{pmatrix}$$

If we now compute $sA \cdot sB^T$ we get

$$\begin{aligned} sA \cdot sB^T &= \begin{pmatrix} \sum_{i=1}^N A_1(p_i) \\ \vdots \\ \sum_{i=1}^N A_n(p_i) \end{pmatrix} \cdot \begin{pmatrix} \sum_{j=1}^M B_1(q_j) & \cdots & \sum_{j=1}^M B_m(q_j) \end{pmatrix} = \\ &= \begin{pmatrix} \sum_{i=1}^N A_1(p_i) \cdot \sum_{j=1}^M B_1(q_j) & \cdots & \sum_{i=1}^N A_1(p_i) \cdot \sum_{j=1}^M B_m(q_j) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^N A_n(p_i) \cdot \sum_{j=1}^M B_1(q_j) & \cdots & \sum_{i=1}^N A_n(p_i) \cdot \sum_{j=1}^M B_m(q_j) \end{pmatrix} \end{aligned}$$

Once we have both parts of eq. (1.7), we now only need to do the element-wise division[2] of the matrices to obtain the resulting F-Transform

$$\mathbf{F}_{nm} = (A \cdot f \cdot B^T) \oslash (sA \cdot sB^T) \tag{1.12}$$

Inverse F-Transform

To compute the inverse F-Transform straightaway we are going to apply a similar method as for the F-Transform. This time the equation to be applied (eq. (1.8)) is much simpler since we do not have a division, so we only need to do a couple of

[2]The element-wise or Hadamard division $\oslash$

matrix multiplications and calculate f^F_{nm} as $f^F_{nm} = A^T \cdot F \cdot B$.

$$
\begin{aligned}
f^F_{nm} &= A^T \cdot F \cdot B = \\
&= \begin{pmatrix} A_1(p_1) & \cdots & A_n(p_1) \\ \vdots & \ddots & \vdots \\ A_1(p_N) & \cdots & A_n(p_N) \end{pmatrix} \cdot \begin{pmatrix} F(1,1) & \cdots & F(1,M) \\ \vdots & \ddots & \vdots \\ F(N,1) & \cdots & F(N,M) \end{pmatrix} \cdot B = \\
&= \begin{pmatrix} \sum_{i=1}^n A_i(p_1) \cdot F(i,1) & \cdots & \sum_{i=1}^n A_i(p_1) \cdot F(i,m) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^n A_i(p_N) \cdot F(i,N) & \cdots & \sum_{i=1}^n A_i(p_N) \cdot F(i,m) \end{pmatrix} \cdot \begin{pmatrix} B_1(q_1) & \cdots & B_1(q_M) \\ \vdots & \ddots & \vdots \\ B_m(q_1) & \cdots & B_m(q_M) \end{pmatrix} = \\
&= \begin{pmatrix} \sum_{j=1}^m \left(\sum_{i=1}^n A_i(p_1) \cdot F(i,j) \right) \cdot B_j(q_1) & \cdots & \sum_{j=1}^m \left(\sum_{i=1}^n A_i(p_1) \cdot F(i,j) \right) \cdot B_j(q_M) \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^m \left(\sum_{i=1}^n A_i(p_N) \cdot F(i,j) \right) \cdot B_j(q_1) & \cdots & \sum_{j=1}^m \left(\sum_{i=1}^n A_i(p_N) \cdot F(i,j) \right) \cdot B_j(q_M) \end{pmatrix}
\end{aligned}
$$

Results

Let us compare the results obtained with this new method, the original and the previous one from subsection 1.2.1.

(a) Inverse with original method. $MSE = 208.833$

(b) Optimized inverse. $MSE = 208.833$

(c) Direct inverse. $MSE = 280.833$

Fig. 1.5 Result of applying the two different introduced methods reducing (a) from 320×300 to 32×30 (100:1) and the Mean Squared Error.

As we saw in *Fig. 1.4*, once again the MSE is the same for the three cases, so we will compare them with respect to the original inverse. The MSE in this case is $1.746 \cdot 10^{-26}$, which is slightly bigger than the previous one, but the difference is so small (about 10^{-27}), it is negligible.

1.2.3 Optimization on the computation of the fuzzy partition

To this point, the matrices A and B with the fuzzy partitions were computed applying eq. (1.1) to each element $A_K(p_i)$ and $B_L(q_j)$ of the matrix with two nested `for`-loops. The function receives as argument a K or L index, the point p_i or q_j for which the value is going to be computed and the nodes used for the partition. There is, though, a seamless faster way to compute the partitions taking advantage of the Python-supported Boolean indexing.

We are going to construct a new function that receives the K or L index, the nodes for the partition and a whole array of points $\mathbf{x} = (x_1, \ldots, x_n)$ where $x_i = i$ and it computes the whole partition for the given K or L at once. This way, we only need to go through the rows of the matrices A and B going from complexity $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

The only thing changing in this algorithm is the way in which we compute the matrices A and B in regard to the Python code and not to the theoretical part, which will provoke no change in the result. Therefore, comparing the results of the transform would not make sense. Instead, we are now going to compare the approximate execution time for each of the four methods to see if we did improve the complexity of the algorithm in the following table.

	Time F-Transform + Inverse	Total (ms)
Original Method	12min 24s + 6min 26s	1,130,000
Method 1.2.1	503ms + 908ms	1,311
Method 1.2.2	45.8ms + 34ms	79.8
Method 1.2.3	4.62ms + 5.18ms	9.8

Table 1.1 Execution time of each method.

As we see, the original method with $\mathcal{O}(n^4)$ is so much slower in comparison with the other three optimized methods. In fact, 861.94 times slower than method 1.2.1 and $1.15 \cdot 10^5$ times slower than the method presented in this subsection.

This way, Algorithm 1 shows the pseudocode of our optimized F-Transform.

Algorithm 1 Optimized Fuzzy Transform

Input: I: image of $N \times M$ pixels; $n < N$, $m < M$: Number of nodes in the fuzzy partition.

Output: F: image of $n \times m$ pixels containing the F-Transform of I.

1: nodesX := array of n equally distributed real numbers from 1 to N.
2: nodesY := array of m equally distributed real numbers from 1 to M.
3: A:= $n \times N$-matrix
4: B:= $m \times M$-matrix
5: rangeN := $\{1, \ldots, N\}$
6: rangeM := $\{1, \ldots, M\}$
7: **for** each k of $\{1, \ldots, m\}$ **do**
8: A := Apply the optimized function for the fuzzy partition (section 1.2.3) with k, rangeN and nodesX
9: **end for**
10: **for** each l of $\{1, \ldots, n\}$ **do**
11: B := Apply the optimized function for the fuzzy partition (section 1.2.3) with l, rangeM and nodesY
12: **end for**
13: $F := A \cdot I \cdot B.T$
14: sA := column array with the row sum of A
15: sB := column array with the row sum of B
16: $F := F \oslash (sA \cdot sB.T)$

Chapter 2

Optimization of the Fuzzy Partition

As seen in 1.1.1, the nodes to create a fuzzy partition of an interval $[a, b]$ are created uniformly dividing the interval in n disjoint subintervals. This has some obvious advantages. First, there is almost no cost on computing the partition compared with an optimized method that takes into account the information in each image. On the other hand, if we know the partitions are done uniformly, when we calculate the inverse F-Transform, we don't need to know how the partitions were distributed to do the F-Transform and it is possible to do it directly dividing it again uniformly. Therefore, we do not need to store the information of the partitions, which makes it even more efficient in terms of compression.

It is obvious though, that choosing an appropriate partition for each case will give a better transform since we are able to take into account the characteristics of each function or image. Let us define $f(x) = 5e^{-2(x-2)^2}$, a non-negative function on [0,10], which is constant except in the subinterval [0,4] as seen on *Fig. 2.1*.

By applying eq. (1.2) we are going to compute the integral inverse F-Transform with a triangular fuzzy partition (eq. (1.1)) with $n = 5$. As we can see in *Fig. 2.2*, the F-Transform we get does not fit at all to the original function $f(x)$ because most of the information is in the subinterval $[0, 4]$ but the uniform distribution of the nodes is giving the same weight to the subinterval $(4, 10]$ where the function is constant.

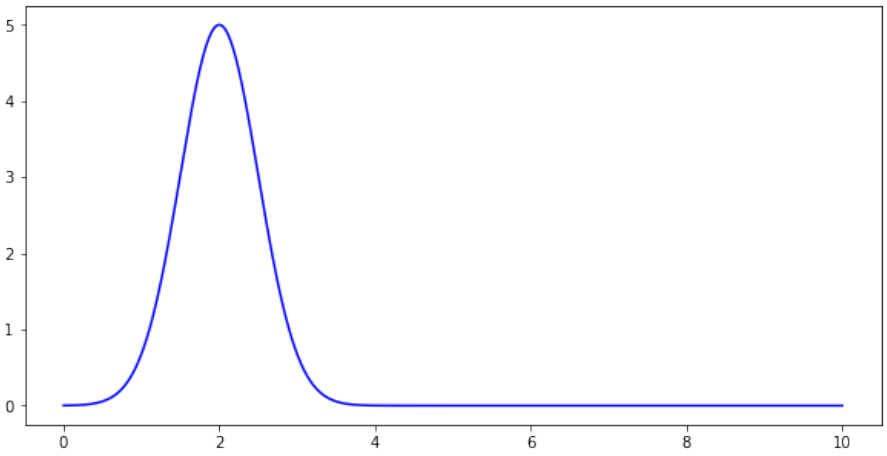


Fig. 2.1 Function $f(x) = 5e^{-2(x-2)^2}$ on the interval $[0, 10]$.

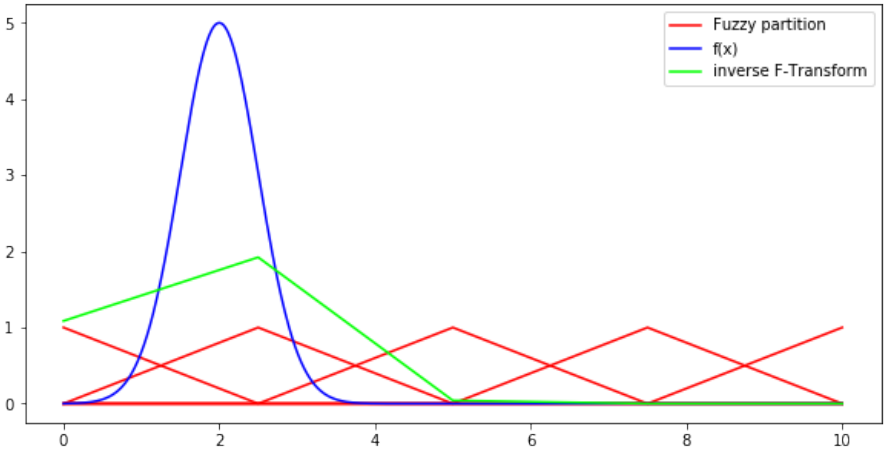


Fig. 2.2 Inverse F-Transform of $f(x)$ calculated with a uniform triangular fuzzy partition with $n = 5$.

If we think of $f(x)$ as a probability density function[1], we can then compute its cumulative density function $F(x)$.[2] Then, with $x_1 = a$ and $x_n = b$ we evenly distribute the rest of the nodes in the codomain of F $[0, 1]$ and, by approximating the inverse of F, we obtain the values for the nodes in the domain of $f(x)$.

[1] For a non-negative function f, we consider the probability density function as $g(x) = \frac{f(x)}{\int_{-\infty}^{\infty} f(x)dx}$

[2] $F(x) = \int_{-\infty}^{x} f(t)dt$

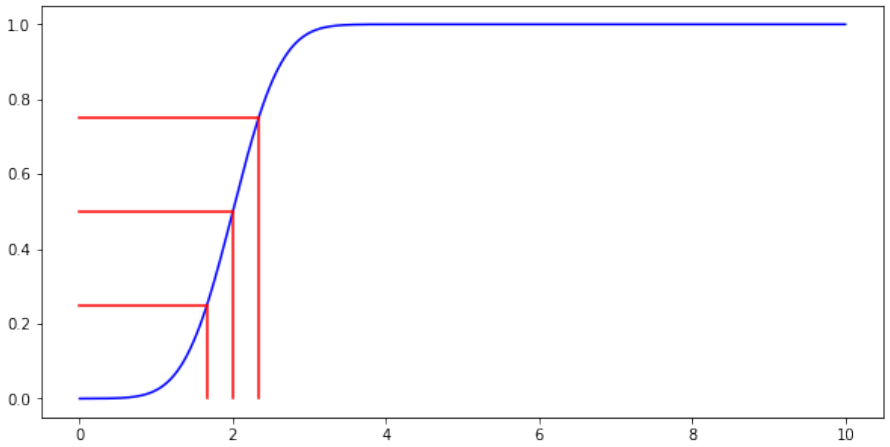


Fig. 2.3 Partition of the domain $[0, 10]$ based on the cumulative density function of $f(x)$, F.

As seen in *Fig. 2.3* the resulting nodes are $x_1 = 0$, $x_2 = 1.66$, $x_3 = 2$, $x_4 = 2.34$, $x_5 = 10$. With these nodes we can compute the partition and the inverse F-Transform as shown in *Fig. 2.4*, which gives a better approximation of the function $f(x)$ than *Fig. 2.2* even with the same number of nodes.

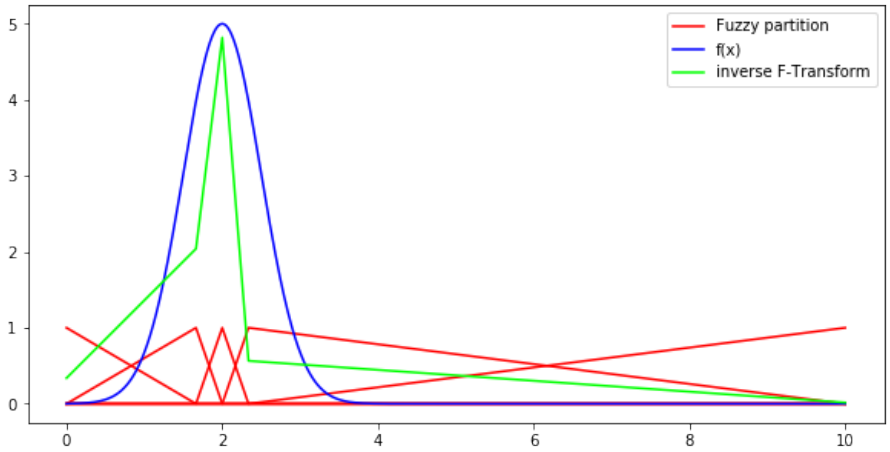


Fig. 2.4 Inverse F-Transform of $f(x)$ with a triangular fuzzy partition of adjusted nodes with $n = 5$.

This method, however, only works when the subinterval where most of the information is, is the same as where the function reaches its highest values. If we define a new function $f_2(x) = 5(1 - e^{-2(x-8)^2})$, as seen in *Fig. 2.5* we want the fuzzy partition to have most of its nodes around the interval $[6, 10]$, where the function changes the most.

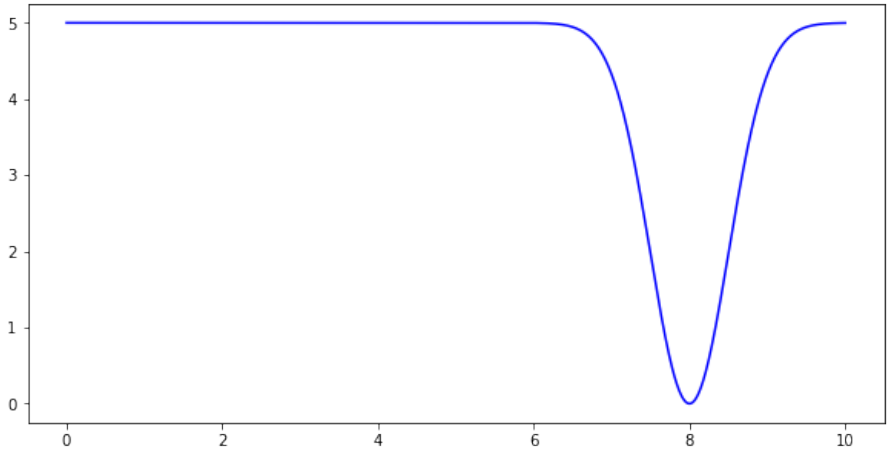


Fig. 2.5 $f_2(x) = 5(1 - e^{-2(x-8)^2})$ defined on the interval $[0, 10]$.

Nonetheless, we can see in *Fig. 2.6* that, when computing the cumulative density function, in this case, the partition does not coincide with the values we expected to have and, actually, the nodes $x_2 = 2.19$, $x_3 = 4.37$ and $x_4 = 6.56$ are exactly in the opposite interval we want them to be as we expected to have them in $[6, 10]$. And thus, the inverse F-Transform of $f_2(x)$ (*Fig. 2.7*) is not as good as for $f(x)$ with this same method.

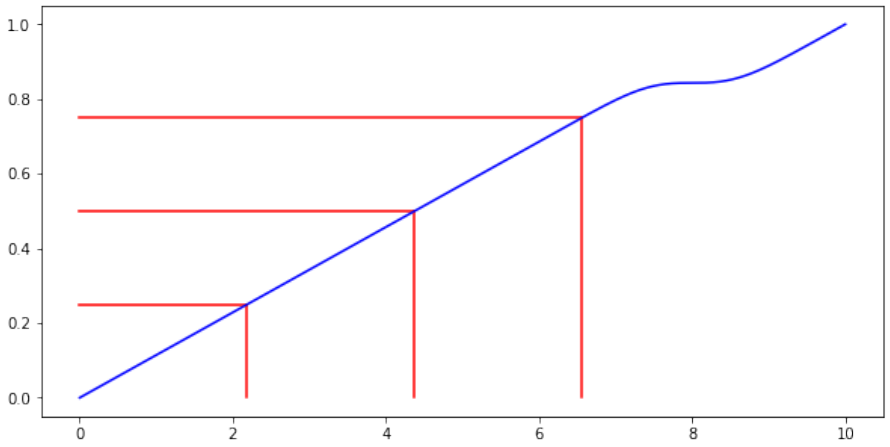


Fig. 2.6 Fuzzy partition of the interval $[0, 10]$ based on the cumulative density function of $f_2(x) = 5(1 - e^{-2(x-8)^2})$.

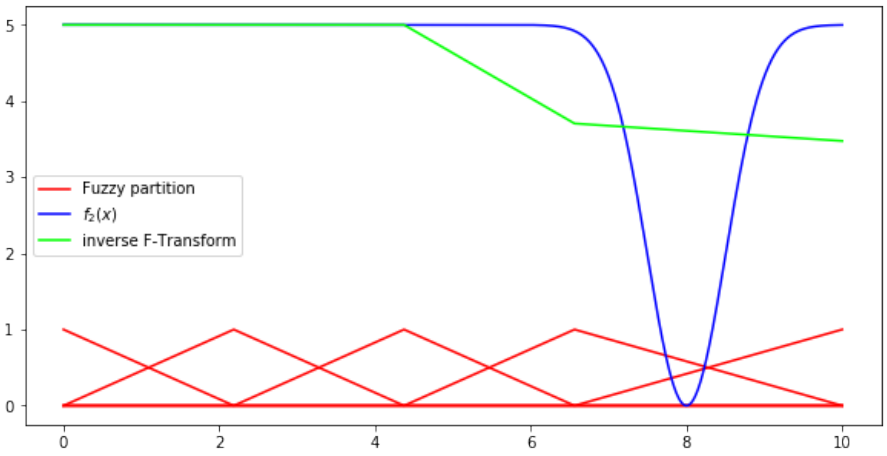


Fig. 2.7 Inverse F-Transform of $f_2(x)$ with a uniform triangular fuzzy partition of adjusted nodes with $n = 5$.

2.1 Optimized partition using the first derivative

To solve the arisen problem, we propose to use the absolute value of the first derivative of the function $f(x)$ rather than the function itself, since the absolute value of the derivative is higher where the original function changes.

Given a non-negative differentiable function $f : [a, b] \to \mathbb{R}^+$, the methodology follows these steps:

1. Compute the density function given by the absolute value of the first derivative of the function, i.e., $d : [a, b] \to \mathbb{R}^+$ given by $d(x) = \frac{|f'(x)|}{\int_a^b |f'(x)|dx}$, $\forall x \in [a, b]$.
2. Compute the cumulative distribution function of d, $D : [a, b] \to \mathbb{R}^+$ given by $D(x) = \int_a^x d(t)dt$, $\forall x \in [a, b]$.
3. Divide the codomain of D $[0, 1]$ into n equally distributed nodes: $x'_1, \ldots, x'_n$, where $x'_i = \frac{i-1}{n-1}$

As D might not be an invertible function, we will approximate the inverse $D^{-1}(x)$ in the points $x'_2, \ldots, x'_{n-1}$ as:

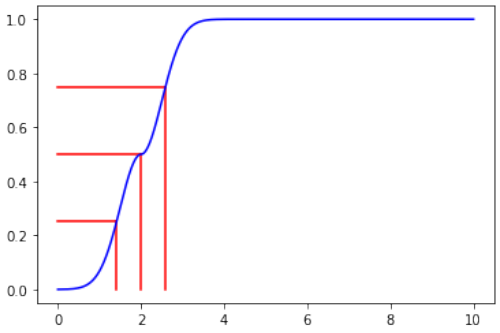
$$D^{-1}(x'_i) = \inf\{x \mid D(x) \geq x'_i\} \tag{2.1}$$

In order to have the fuzzy partition well defined, we determine that $x_1 = a$ and $x_n = b$ and thus we get the fuzzy partition as:

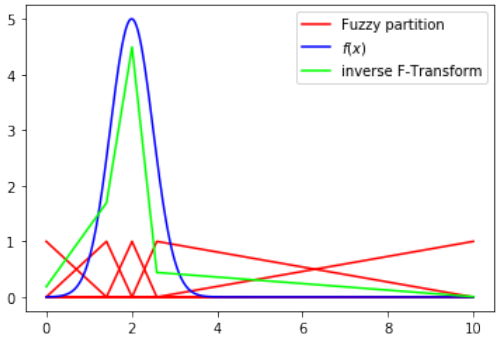
$$x_1 = a, x_2 = D^{-1}(x'_2), \ldots, x_{n-1} = D^{-1}(x'_{n-1}), x_n = b$$

If we retrieve the previously defined functions $f(x)$ and $f_2(x)$, we can now apply this algorithm to compute the inverse F-Transform given by the optimized partitions. In figures *2.8* and *2.9* we can see the result of applying the method to both of the functions. As it can be seen in *Fig. 2.8(a)*, the nodes obtained for $f(x)$ are more or

less the same as with the previous method and the inverse F-Transform is the same as in *Fig. 2.4*. However, in *Fig. 2.9* we see that, in this case, the cumulative density function of the absolute value of the first derivative of $f_2(x)$, showed in *Fig. 2.9a*, gives a more appropriated partition of the interval $[0, 10]$ for the function $f_2(x)$, and the inverse F-Transform in *Fig. 2.9(b)* is a better approximation of the function $f_2(x)$ than the one in *Fig. 2.7*.

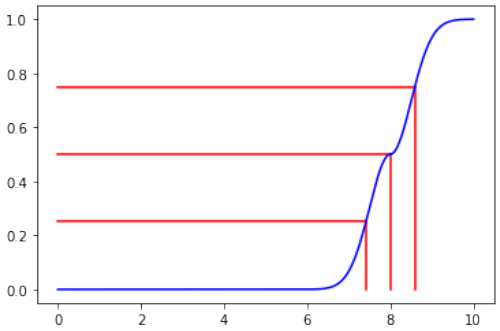


(a) Partition of the interval $[0, 10]$ computed with $F(x)$ given by eq. (2.1).

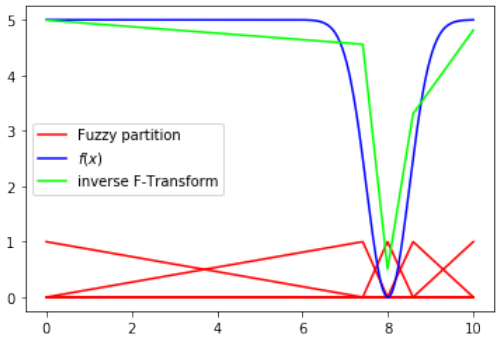


(b) Inverse F-Transform of $f(x)$ using an optimized partition.

Fig. 2.8 Optimized inverse F-Transform applied to $f(x) = 5e^{-2(x-2)^2)}$



(a) Partition of the interval $[0, 10]$ computed with $F_2(x)$ given by eq. (2.1).



(b) Inverse F-Transform of $f_2(x)$ using an optimized partition.

Fig. 2.9 Optimized inverse F-Transform applied to $f_2(x) = 5(1 - e^{-2(x-8)^2)})$

2.2 Optimized partition for image compression

An $N \times M$ grayscale image can also be seen as a bivariate discrete function $I : \{0, \ldots, N-1\} \times \{0, \ldots, M-1\} \to \{0, \ldots, 255\}$. As seen in subsection 1.1.2, the F-Transform can be applied to discrete bivariate functions, therefore, we can adapt the algorithm described in section 2.1 to optimize the partitions for images. Given an image I of $N \times M$ pixels, the objective is to find a fuzzy partition of $[0, N-1] \times [0, M-1]$ based on the first derivative of the image.

While adapting the algorithm we find two obstacles. First, approximating the

derivative of a discrete function. Second, performing this optimization on two-dimensional functions.

The idea of the first derivative consists in calculating the maximum difference of intensities between the pixels around each pixel. Let $I(x, y)$ with $(x, y) \in \{0, \ldots, N-1\} \times \{0, \ldots, M-1\}$ be the intensity of the pixel on the x-th row and y-th column and let $\mathcal{N}_t(x, y) = \{(i, j) \mid |i - x| \leq t \text{ and } |j - y| \leq t\}$ be the set of indices of the neighbouring pixels of $I(x, y)$ with distance less than or equal to t. Then, the approximation of the magnitude of the first derivative in a given pixel $I(x, y)$ is calculated as:

$$d(x, y) = \max_{(i,j) \in \mathcal{N}_t(x,y)} I(i, j) - \min_{(i,j) \in \mathcal{N}_t(x,y)} I(i, j) \tag{2.2}$$

Evidently the approximation of the magnitude directly depends on t, the bigger t, the greater the magnitude of the derivative. We usually consider $t = 1$ or $t = 2$, which respectively form a 3×3 or 5×5 window around the pixel.

By applying eq. (2.2), we obtain a new image d with an approximation of the magnitude of the first derivative in each pixel. d is obviously a discrete two-dimensional function and the second problem still remains unsolved. We are going to transform d into two one-dimensional functions d_0 and d_1 respectively defined on $[0, N-1]$ and $[0, M-1]$. These will be the result of accumulating the values of d along the corresponding axis:

$$d_0(x) = \frac{\sum_{j=0}^{M-1} d(x, j)}{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} d(i, j)}, \ x \in [0, N-1] \tag{2.3}$$

$$d_1(y) = \frac{\sum_{i=0}^{N-1} d(i, y)}{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} d(i, j)}, \ y \in [0, M-1] \tag{2.4}$$

Now that we have d_0 and d_1 one-dimensional functions that represent the amount of changes in rows and columns, we can calculate the cumulative functions to apply the transform to them. The cumulative functions D_0 and D_1 are defined as:

$$D_0(x) = \sum_{i=0}^{x} d_0(i), \ x \in [0, N-1] \tag{2.5}$$

$$D_1(y) = \sum_{j=0}^{y} d_1(j), \ y \in [0, M-1] \tag{2.6}$$

Following the steps described in section 2.1, we divide the interval $[0, 1]$ into n and m equally distributed nodes and by approximating the inverse of D_0 and D_1 with eq. (2.1), we obtain the nodes of the partitions $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ in the intervals $[0, N-1]$ and $[0, M-1]$. Algorithm 2 represents the pseudocode for this algorithm.

Algorithm 2 Optimized fuzzy partition

Input: I: image of $N \times M$ pixels; $n < N$, $m < M$: Number of nodes in the fuzzy partition; t: size of the neighbourhood.

Output: Optimized fuzzy partition $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ of $[0, N-1]$ and $[0, M-1]$, respectively.

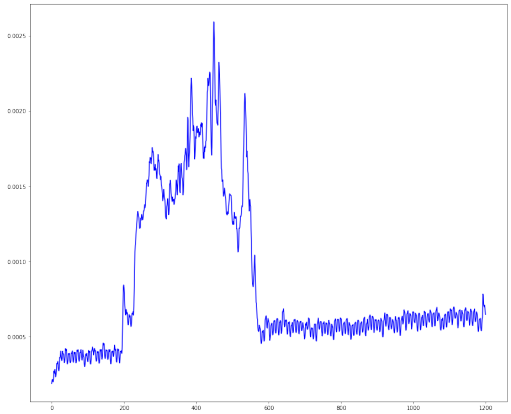
1: **for** each pixel (x, y) of I **do**
2: Apply eq. (2.2) to calculate d(x,y)
3: **end for**
4: Apply eq. (2.3) and eq. (2.4) to calculate d_0 and d_1 respectively.
5: Apply eq. (2.5) and eq. (2.6) to calculate D_0 and D_1 respectively.
6: **for** each $i \in \{1, \ldots, n-2\}$ **do**
7: $x'_i := \frac{i-1}{n-1}$
8: $x_i := \inf\{x \mid D_0(x) \geq x'_i\}$
9: **end for**
10: **for** each $j \in \{1, \ldots, m-2\}$ **do**
11: $y'_j := \frac{j-1}{m-1}$
12: $y_j := \inf\{y \mid D_1(y) \geq y'_j\}$
13: **end for**

Here is an example with the step-by-step results of applying this algorithm to an image. First, we apply eq. (2.2) to obtain the image with the magnitude of the first derivative (*Fig. 2.9(b)*). By applying eq. (2.3) and eq. (2.4) we respectively obtain d_0 an d_1 shown in *Fig. 2.11(a)* and *Fig. 2.11(b)* the approximations of the derivatives on each axis. With eq. (2.5) and eq. (2.6), we obtain the cumulatives of the derivatives D_0 and D_1 respectively. We divide the interval $[0, 1]$ into n and m equally distributed nodes and, by applying 2.1, we obtain the partitions shown in *Fig. 2.11(c)* and *Fig. 2.11(d)*. *Fig. 2.12* shows the grid created by the uniform partition and our optimized version over the image. In *Fig. 2.13* we see the result of applying the F-Transform and its inverse with the created partitions. *Fig. 2.13(a)* has a MSE of 24.595, whereas *Fig. 2.13(b)* a MSE of 9.362. This difference is obvious just by looking at the bird in the picture and the difference in the sharpness of both images.

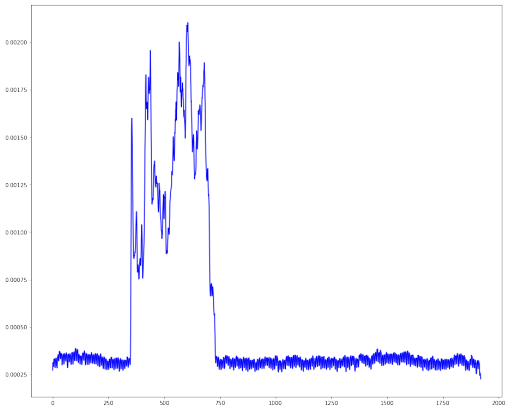
(a) Grayscale version of *Fig. 2.*

(b) Magnitude of the first derivative.

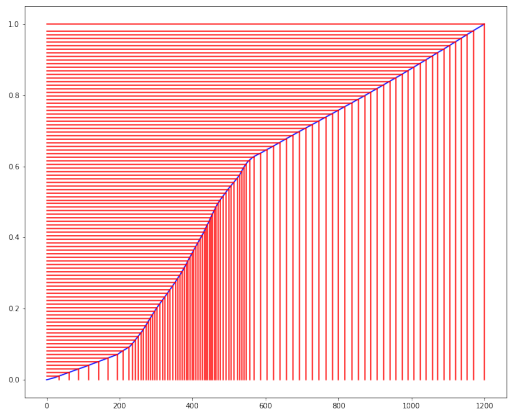
Fig. 2.10 A 1200×1920 image and the magnitude of its first derivative given by eq. (2.2) with $t = 2$.



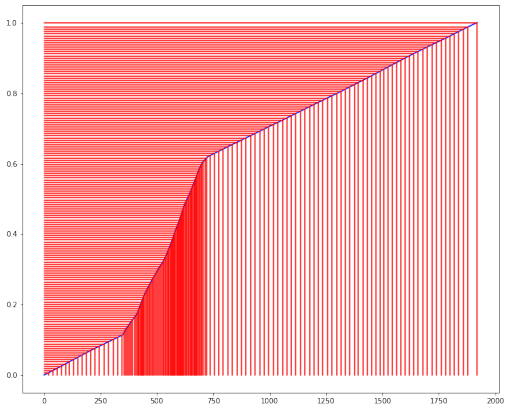
(a) d_0 approximation of the derivative (eq. (2.3)).



(b) d_1 approximation of the derivative (eq. (2.4)).

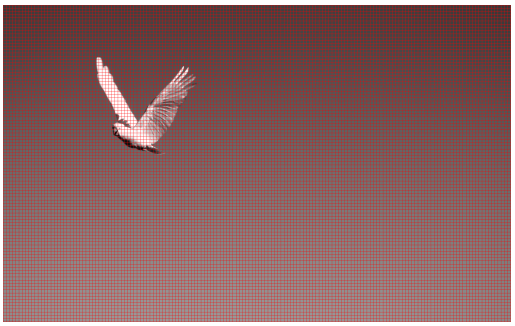


(c) D_0 cumulative of the derivative (eq. (2.5)) and distribution of the nodes in the x axis.



(d) D_1 cumulative of the derivative (eq. (2.6)) and distribution of the nodes in the y axis.

Fig. 2.11 First derivatives and its cumulatives with the distribution of the nodes.



(a) Uniform distribution of the nodes.

(b) Distribution of the nodes based on the first derivative.

Fig. 2.12 Grids created by the uniform and the optimized partitions.

(a) Inverse uniform F-Transform with a uniform triangular partition. $MSE = 24.595$

(b) Optimized F-Transform with partition created with Algorithm 2. $MSE = 9.362$

Fig. 2.13 Comparison between the uniform and the optimized version of the F-Transform after reducing *Fig. 2.9(a)* with a $144 : 1$ compression ratio from dimension 1200×1920 to 100×160.

Chapter 3

Key point detection

Now that we have seen a method that improves the fuzzy partition, we can try to improve the way in which we perform the F-Transform to the image. We want to focus the F-Transform in the object of the image, thus, we are going to perform two different F-Transforms and see if the results are better. The first one will be the F-Transform of the object itself and the second one will have the information of the background.

To do this, there are some things we need to take into account. First, we obviously need to find a way to separate the object and the background. Second, once we get the background we need to have a whole image with the empty background. And last, if we have an $N \times M$ image we want to reduce to $n \times m$ with n and m the number of nodes used for each of the dimension, we need to define how many of the nodes will be used to do the F-Transform of the object and how many for the background.

3.1 Separating the object and the background

In this section we are first going to define some methods to find where the object is so we can then perform both of the transforms. Before we apply each of the algorithms to obtain the key points, in order to avoid cases were there is some noise in the background that can then give us false key points, we are going to apply a Gaussian filter to the image. Once we have these points, we can then get the area that contains all of them and separate the object from the background.

3.1.1 Based on the magnitude of the first derivative

For this method, we are going to take advantage of the image d we computed by applying eq. (2.2). We know the object will be located where the first derivative reaches the highest magnitudes, therefore, we can fix a percentage that, based on the highest value of the first derivative, will give us the pixels where the image changes the most.

Since the background of the images we are using is more or less constant, all this points will be the ones of the object, as seen in *Fig. 2.10(b)*.

3.1.2 OneClassSVM

The sklearn library [3] in python has the class OneClassSVM which is an unsupervised outlier detector. This class receives a set of points to be classified. The SVM tries to find a frontier that fits to the given points and then the points that do not fit inside the created frontier are classified as outliers. We can use this outlier detector by passing to it the points of the magnitude and then it will give us back the points that it considers that are not outliers so we can then use them to find the object.

3.1.3 Feature detection with OpenCV

The OpenCV library [4] has some methods that are used to find the key points of an image and, for this purpose, we are going to use two of them, Features from Accelerated Segment Test, and Speeded-Up Robust Features.

Features from Accelerated Segment Test

The Features from Accelerated Segment Test or FAST method implemented in the OpenCV library [5] performs a key-point detection based on corner detection based on the algorithm proposed by E. Rosten [6]. With a fixed threshold t, this method goes through the image and, for each pixel with intensity I_p. Considering a circle of 16 pixels around it (*Fig. 3.1*), the pixel contains a corner if there exists a set of $n = 12$ pixels around it that are brighter than $I_p + t$ or darker than $I_p - t$.

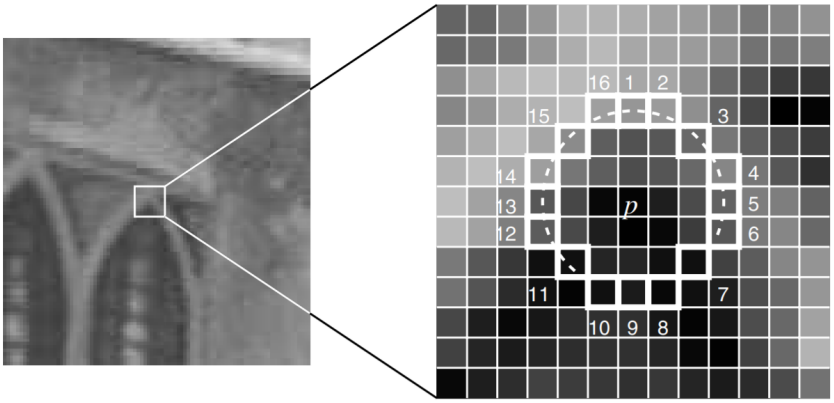
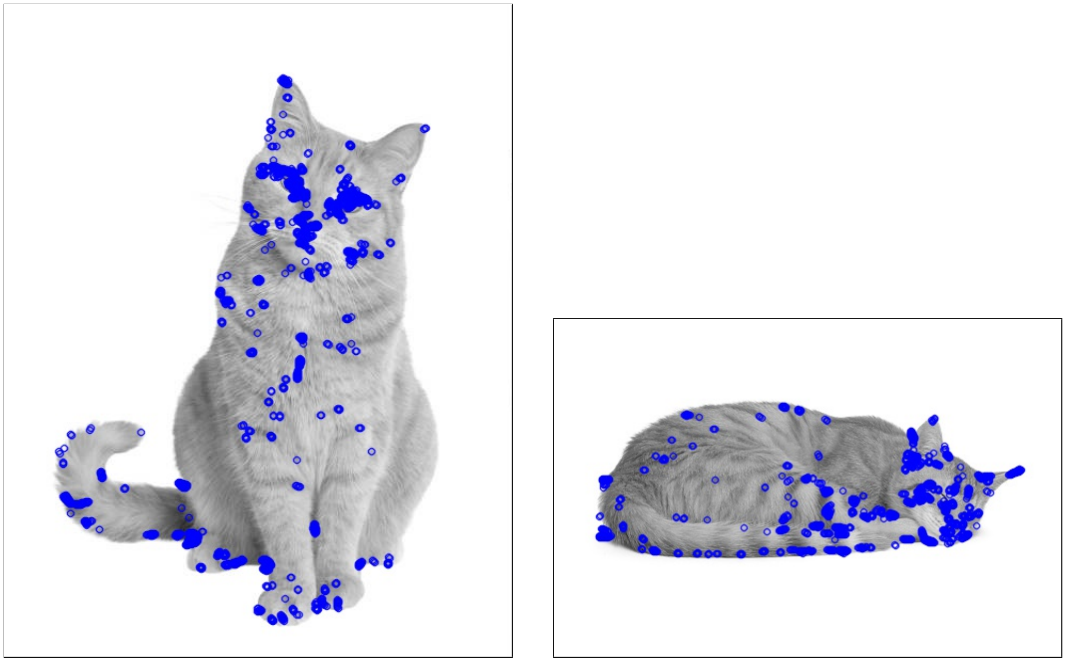


Fig. 3.1 FAST algorithm taken from [6].

To make the algorithm faster, a high-speed test is performed by first analyzing pixels 1 and 9 to check if any of them fulfills the criterion. If any of them does, it checks 5 and 13. Since $n = 12$, at least 3 of these must be darker or brighter in order to consider the pixel a corner. If the pixel passes the test, then the full circle will be examined.



(a) *(b)*

Fig. 3.2 Example of applying the FAST method to two images.

Speeded-Up Robust Features

When detecting corners, the algorithm must obviously be rotation-invariant, which means that even if the image is rotated, the detected corners will be the same. There is, though, another possibility of not detecting a corner, this is when the image is scaled and the algorithm does not detect the corner because it is too big when analyzing pixel-to-pixel. An algorithm that detects the corners even when it the image is scaled is called scale-invariant.

D. G. Lowe proposed the Scale-Invariant Feature Transform or SIFT algorithm [7] to solve this. The method is based on the Laplacian of Gaussian or LoG where the Laplacian is found for different values of σ. Since the LoG has a higher computational cost, an approximation is calculated with the Difference of Gaussians or DoG. The DoG is computed as the difference of the result of applying a Gaussian filter with different σ.

This algorithm is quite slow and for this problem, H. Bay, T. Tuytelaars, and L. Van Gool proposed the Speeded-Up Robust Features or SURF method [8]. Based on D. G. Lowe's proposal, SURF is, as its name suggests, the speeded-up version of the SIFT method.

The SURF goes a little further with the approximation of the LoG and instead of using the DoG, it makes an approximation by means of a the discretized hessian matrix of the Gaussian filter and applies different size of box filters to the image. We can see in *Fig. 3.3* the 9×9 box filters for the y^2 and xy second order derivatives. As the image shows, the filters only need to sum the white areas and subtract the black ones. This way, the filter can be easily applied in a fast way with $\mathcal{O}(1)$ by use of integral images making SURF much faster than SIFT with only having to access

to 4 values of the integral image for each region to be added or subtracted.

We can also find the SURF implemented in the OpenCV library [9].

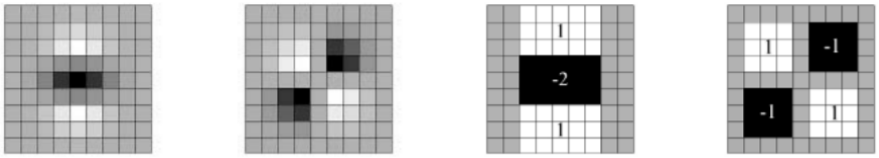
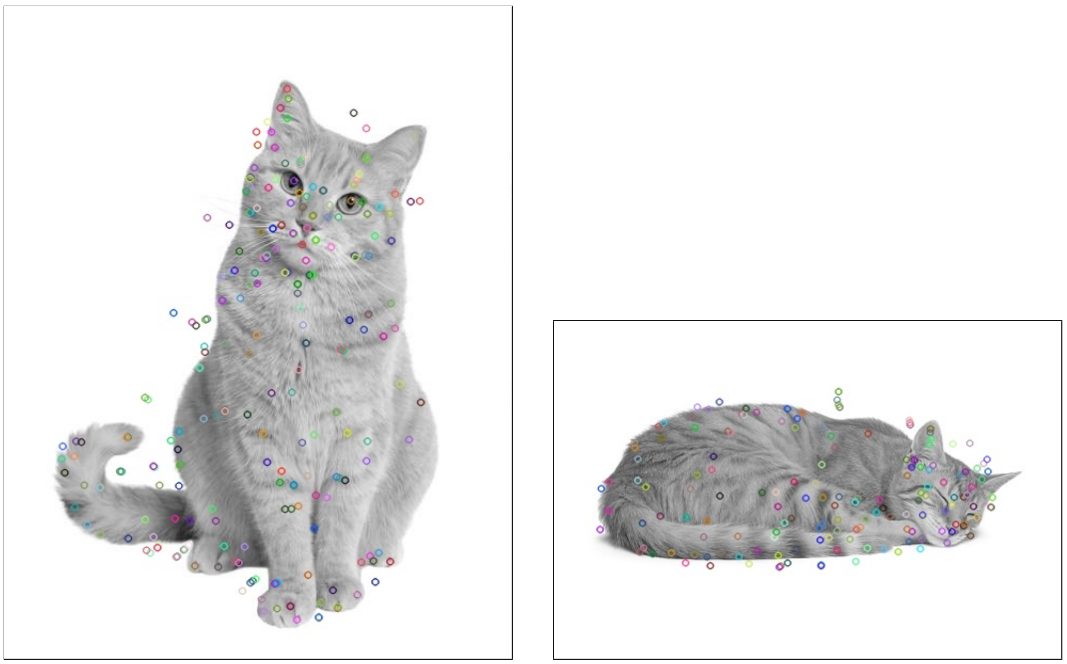


Fig. 3.3 Approximation of the discretized Gaussian second order derivatives in y-direction and xy-direction used for box filters. Taken from [8].



(a) *(b)*

Fig. 3.4 Result of applying the SURF method to two images.

3.2 Inpainting

Once we have found the area where the object is located with any of the previous methods, there is still one problem for the compression of the background, and that is, that, even if we can select the object from the image, the background still contains the object on it.

Inpainting is a field of image processing that helps reconstruct images that have been damaged and have some parts where the image is incomplete. This method can also be used to remove parts of the image we do not want to have on it. This is exactly what we want and, therefore, we are going to apply an inpainting algorithm to our image in order to remove the object from it and get an approximation of what the background would look like without the object.

In order to apply an inpainting algorithm, a mask containing the parts were the image is damaged or of the pixels we want to remove is needed. In our case, this is quite simple since we already have the location of the object so we only need to use the object itself as a mask.

The inpainting method we will be using is the one implemented in the OpenCV library [4] for python with the `cv.INPAINTING_TELEA` flag. The algorithm follows the method proposed by A. Telea [10] based on the Fast Marching Method, which is a faster method of inpainting that gives almost the same results as more complex algorithms.

This method starts from the boundary of the area to be inpainted and gradualy goes inside by filling the boundary first. A pixel to be inpainted is taken and it is replaced by a normalized weighted sum of the known pixels in a small neighbourhood around it. The nearer the pixel to the point, the greater weight it has.

Fig. 3.5 Inpainted image with the mask of the object found based on subsection 3.1.1.

3.3 Divide the number of nodes

For an $N \times M$ image to be reduced to $n \times m$, we will be using n nodes for one dimension and m nodes for the second one. Now that we have located the object and separated it from the background, we need to think of a way to divide n and m between both the object and the background.

We propose two different methods for this purpose.

3.3.1 First method

An easy and fast way to divide the nodes is to determine what percentage of the changes happen in the area of the object and outside of it.

The method is simple, we only need to use the matrix with the magnitudes of

the first derivative d computed with eq. (2.2). We first normalize the matrix and calculate the sum of the magnitudes inside the area of the object. This gives us the percentage to use for the object and by subtracting it to 1, we will get the percentage for the background.

3.3.2 Second method

It could be that the image had some noise in the background and if the background is much bigger than the object, the sum of the magnitudes were similar to the sum of the magnitudes of the object. To avoid this, we are going to first sum the magnitudes and then divide it with the area in number of pixels. Once we have the indexes for both of the areas, we need to normalize them in order to get the percentages.

Chapter 4

Results

In this chapter we are going to see the results of the developed algorithm and the proposed methods for different compression ratios. Taking into account all the explained methods, the optimized F-Transform and the uniform F-Transform, there are 10 different transformations we can do to an image. Inasmuch as 10 results for every image and every compression ratio we want to use is too much, first of all, we are going to apply all the previous methods over a single image with different compression ratios and compare the results. Then, we will find out which method is the best one and we will compare the results given by that one method, the Optimized and the Uniform F-Transform over a set of images and the behaviour with a larger number of higher compression ratios. To conclude, we are going to try the algorithm with a different kind of images to see how good the results are.

4.1 Best method

We are going to use the image from *Fig. 2* and apply to it all the methods for the compression ratios (4:1, 8:1 and 64:1). Figures 4.12 - 4.20 at the end of this chapter suggest that the best configuration might be method 3.1.1 for finding the object based on the magnitude of the first derivative with the first method to divide the number of nodes. If we see *Table 4.1*, where the same calculations have been done for a larger set of images (shown in *Fig. 4.5*), we can confirm that the best configuration is the one just mentioned.

Let us compare the different algorithms to see if we have improved the use of the F-Transform for image compressing.

Fig. 4.1 shows the Mean Squared Error of the three algorithms for compression ratios x^2 with $x \in \{2, \ldots, 16\}$ for *Fig. 2*. As we can see, the results we can obtain by applying the developed algorithms are so much better than the original method with a uniform partition. For instance, we can achieve $\frac{2}{3}$ less of error with our new methods even with a compression ratio of 256:1.

Let us see what happens if we use a bigger set of images and take a look at the mean MSE.

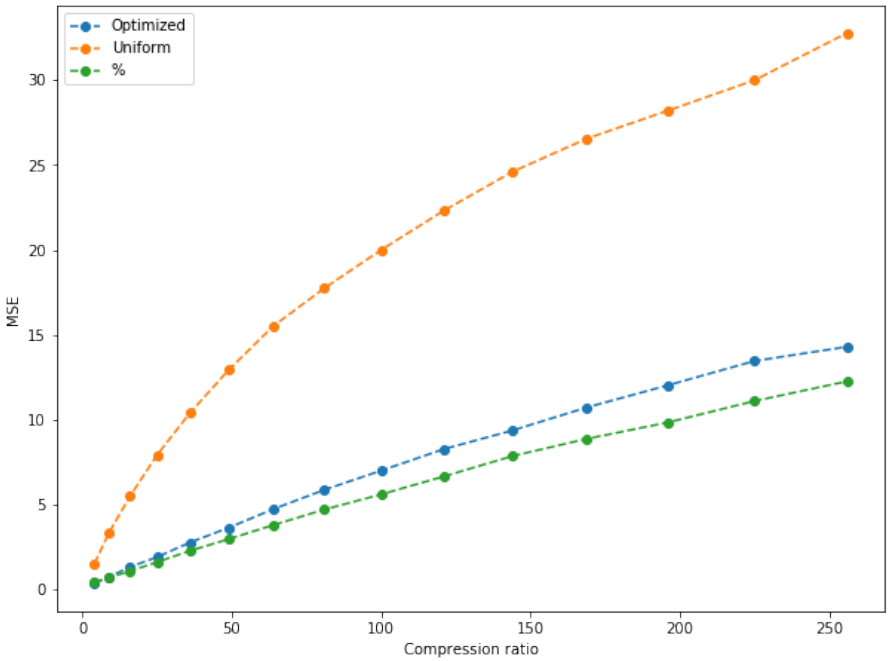


Fig. 4.1 Mean MSE over the set of images for the 3 main methods and different compression ratios.

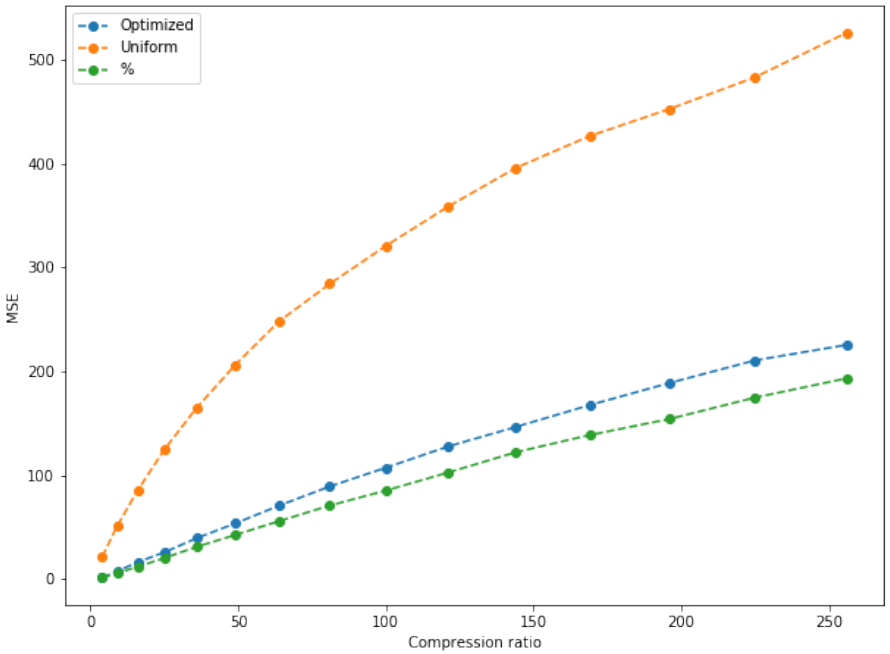


Fig. 4.2 Mean MSE over the object of interest of the set of images for the 3 main methods and different compression ratios.

4.2 General results

Let us see what happens to the set of images with higher compression ratios. *Table 4.2* shows that for a 256:1 compression ratio, the algorithms do not behave as expected. In *Fig. 4.3* we see that for compression ratios higher than 100:1, the optimized version gets worse results than the uniform one, while method 3.1.1 still gets better results.

Nevertheless, if we take a look at *Fig. 4.4* we can observe that, even if the global error of the inverse transform is worse with the optimized method and the difference between the uniform and method 3.1.1 is not that big, the MSE over the object is about $\frac{1}{3}$ less and, as expected, we . Since we are studying images where the object is the thing we are interested in, we can conclude that, if we want to preserve the maximum information as possible of it, both algorithms are better even if we lose more information from the background.

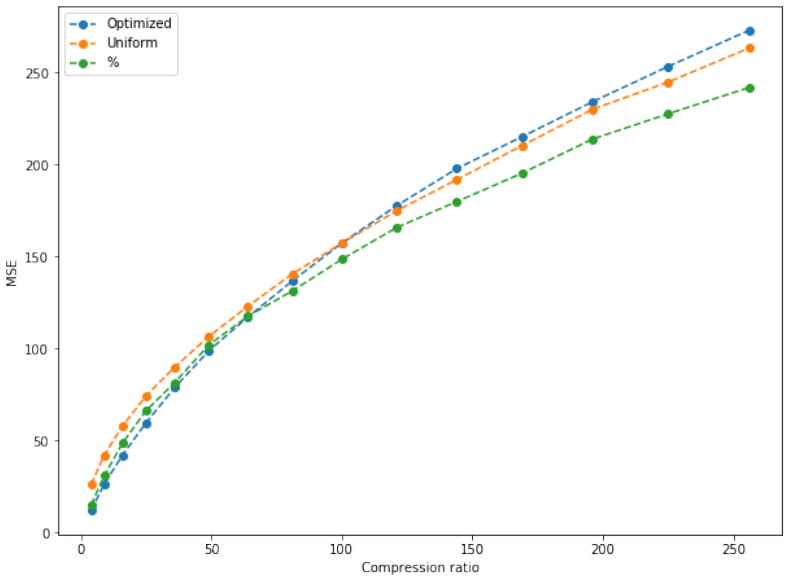


Fig. 4.3 Mean MSE over the set of images for the 3 main methods and different compression ratios.

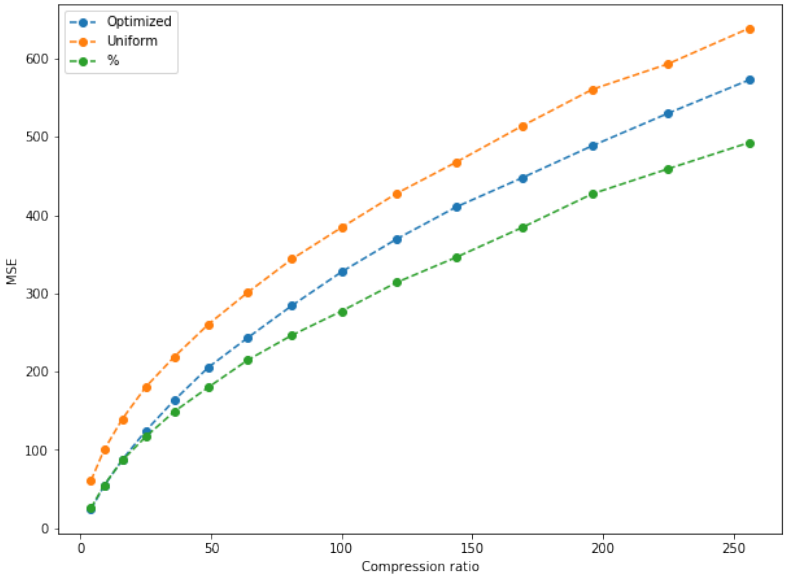


Fig. 4.4 Mean MSE over the object of interest of the set of images for the 3 main methods and different compression ratios.

Fig. 4.5 Used set of images.

%	Partition	Key points	Object		Optimized		Uniform	
			Object	Complete	Object	Complete	Object	Complete
4 : 1	First	%	26.0	15.15	24.08	12.11	60.82	26.22
		svm	80.0	21.82	30.81		102.86	
		Feature FAST	26.0	22.94	23.79		66.68	
		Feature SURF	27.0	24.32	22.64		71.27	
	Second	%	24.26	15.64	24.08		60.82	
		svm	11.0	55.76	30.81		102.86	
		Feature FAST	23.0	30.16	23.79		66.68	
		Feature SURF	22.0	42.75	22.64		71.27	
16 : 1	First	%	87.0	48.69	87.46	42.18	139.51	58.01
		svm	309.0	61.46	135.49		248.67	
		Feature FAST	92.0	62.72	89.98		153.06	
		Feature SURF	96.0	67.68	89.22		160.54	
	Second	%	84.44	48.74	87.46		139.51	
		svm	74.0	140.34	135.49		248.67	
		Feature FAST	82.0	85.53	89.98		153.06	
		Feature SURF	84.0	89.76	89.22		160.54	
64 : 1	First	%	215.0	117.64	243.26	117.19	301.17	123.1
		svm	746.0	148.37	397.51		566.32	
		Feature FAST	225.0	149.85	252.21		330.07	
		Feature SURF	236.0	141.27	242.69		342.5	
	Second	%	204.94	111.0	243.26		301.17	
		svm	215.0	357.11	397.51		566.32	
		Feature FAST	208.0	188.09	252.21		330.07	
		Feature SURF	211.0	192.37	242.69		342.5	

Table 4.1 Mean MSE for over the set of images and the objects of interest for each of the methods explained before with different compression ratios.

%	Partition	Key points	Object		Optimized		Uniform	
			Object	Complete	Object	Complete	Object	Complete
256 : 1	First	%	493.0	242.02	572.65	273.11	638.65	263.4
		svm	1294.0	362.86	976.19		1191.26	
		Feature FAST	526.0	312.35	610.45		696.59	
		Feature SURF	573.0	325.25	621.91		723.79	
	Second	%	476.39	267.77	572.65		638.65	
		svm	552.0	675.69	976.19		1191.26	
		Feature FAST	485.0	349.25	610.45		696.59	
		Feature SURF	499.0	386.93	621.91		723.79	

Table 4.2 Mean MSE for over the set of images and the objects of interest for each of the methods explained before with a 64:1 compression ratio.

4.3 Other types of images

It is obvious that applying any of the algorithms explained in Chapter 3 to any other kind of images would not make sense. These methods try to look for an object in the image and extract it to compress the background and the object separately. However, we have seen in *Fig. 2.12* that Algorithm 2 can find a good partition that makes the F-Transform focus on the areas with more information and achieves a better reconstruction than the uniform one. We are going to apply this algorithm to images that do not fulfil the criteria of having a single object and a constant background to see if the results are better.

For instance, we are going to try with two sets of three images. The first set is of images that have more than one object and the second one of images that do not contain any defined object.

4.3.1 More than one object

If we apply the optimized F-Transform to images that have more than one object, we can see in *Fig. 4.6* that the result is much better than with the uniform transform. As we see in figures *4.7-4.9*, since the cumulative distribution of the first derivative will have the information of both of the objects, the nodes are distributed over the objects therefore making the f-transform focus on both of the objects.

There is however a disadvantage. As we see in *Fig. 4.9(b)*, because of how the objects are located, the nodes create two extra interest areas where there is no object.

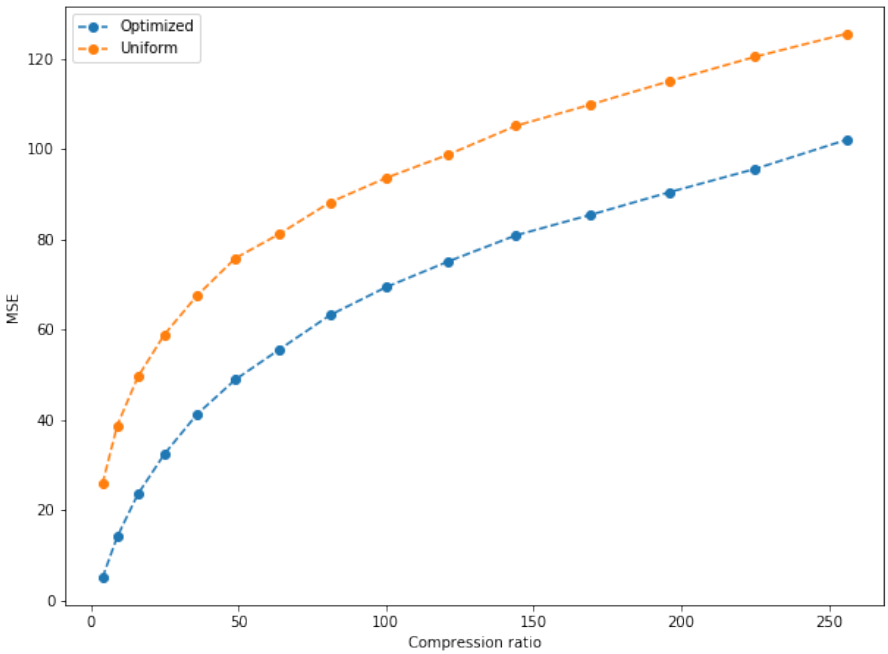
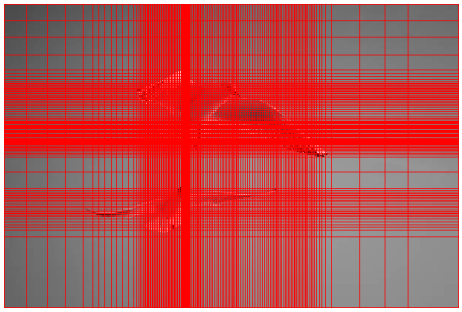


Fig. 4.6 Mean MSE over the set of images with two objects for the two methods and different compression ratios.

(a) Image with 2 objects



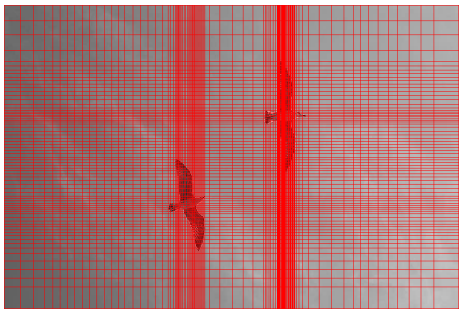
(b) Distribution of the nodes

(c) Optimized F-Transform with uniform F-Transform. $MSE = 66.95$

(d) Optimized F-Transform with optimized F-Transform. $MSE = 29.17$

Fig. 4.7 Comparison between the uniform and optimized F-Transform for image with two objects with a compression ratio of $8:1$

(a) Image with 2 objects

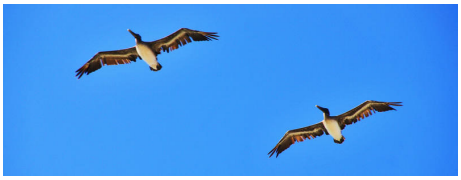


(b) Distribution of the nodes

(c) Optimized F-Transform with uniform F-Transform. $MSE = 32.46$

(d) Optimized F-Transform with optimized F-Transform. $MSE = 18.78$

Fig. 4.8 Comparison between the uniform and optimized F-Transform for image with two objects with a compression ratio of $64:1$



(a) Image with 2 objects

(b) Distribution of the nodes

(c) Optimized F-Transform with uniform F-Transform. $MSE = 59.77$

(d) Optimized F-Transform with optimized F-Transform. $MSE = 31.45$

Fig. 4.9 Comparison between the uniform and optimized F-Transform for image with two objects with a compression ratio of $8:1$

4.3.2 No defined object

Let us test the algorithms with a set of images (*Fig. 4.11*) that do not have any defined object. As we see (*Fig. 4.10*), for smaller compression ratios, there is some difference on the MSE. Since the cumulative of the derivative does not find any object, the nodes are distributed almost uniformly creating a F-Transform very close to the uniform version. Therefore, we can say that for this type of images, the use of the optimized method is not that worth it because the difference in error with respect to the uniform F-Transform is not that big and there is no object where more information is preserved as it happened in section 4.2.

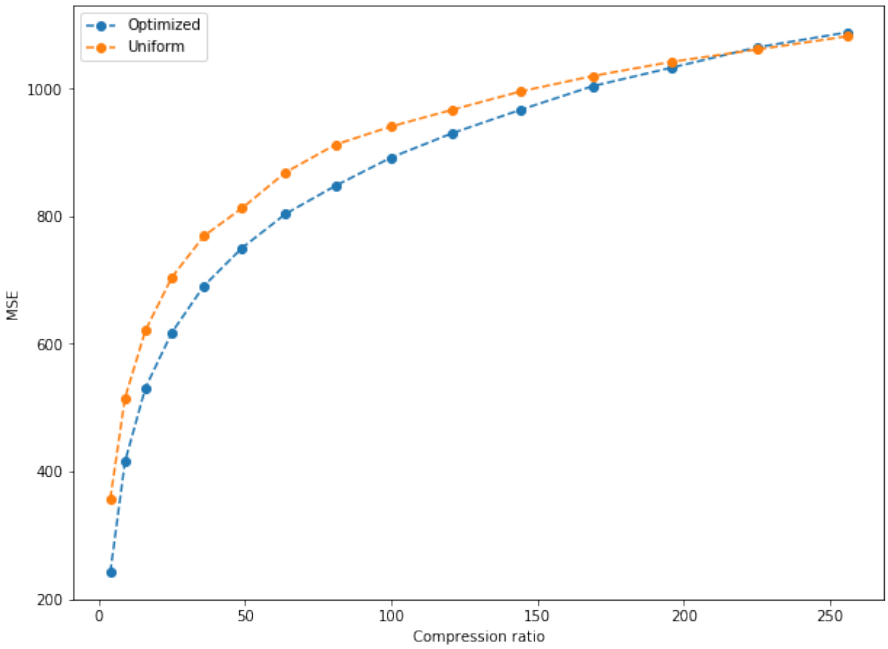
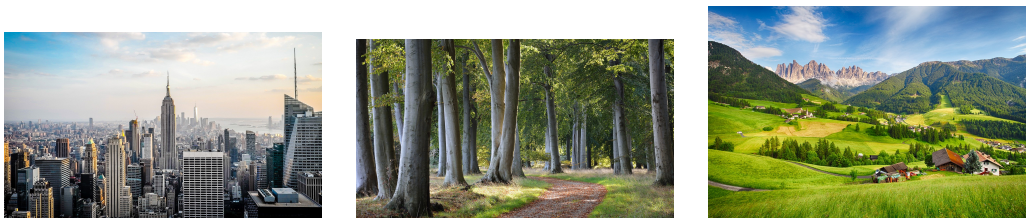


Fig. 4.10 Mean MSE over the set of images without objects for the two methods and different compression ratios.



(a) *(b)* *(c)*

Fig. 4.11 Set of images without a defined object.

4:1 reduction

(a) Uniform F-Transform with compression ratio of $4:1$. $MSE = 1.51$

(b) Optimized F-Transform with compression ratio of $4:1$. $MSE = 0.33$

Fig. 4.12

(a) Optimized F-Transform with method 2.1. $MSE = 0.41$

(b) Optimized F-Transform with OneClassSVM. $MSE = 0.51$

(c) Optimized F-Transform with FAST feature detection. $MSE = 0.46$

(d) Optimized F-Transform with SURF feature detection. $MSE = 0.43$

Fig. 4.13 Different methods of finding an object on the image with method 3.3.1 and a compression ratio of $4:1$

(a) Optimized F-Transform with method 2.1. $MSE = 0.62$

(b) Optimized F-Transform with OneClassSVM. $MSE = 2.8$

(c) Optimized F-Transform with FAST feature detection. $MSE = 2.53$

(d) Optimized F-Transform with SURF feature detection. $MSE = 2.44$

Fig. 4.14 Different methods of finding an object on the image with method 3.3.2 and a compression ratio of $4:1$

8:1 reduction

(a) Uniform F-Transform with compression ratio of $8:1$. $MSE = 5.5$

(b) Optimized F-Transform with compression ratio of $8:1$. $MSE = 1.32$

Fig. 4.15

(a) Optimized F-Transform with method 2.1. $MSE = 1.08$

(b) Optimized F-Transform with OneClassSVM. $MSE = 1.47$

(c) Optimized F-Transform with FAST feature detection. $MSE = 1.28$

(d) Optimized F-Transform with SURF feature detection. $MSE = 1.21$

Fig. 4.16 Different methods of finding an object on the image with method 3.3.1 and a compression ratio of 4 : 1

(a) Optimized F-Transform with method 2.1. $MSE = 2.5$

(b) Optimized F-Transform with OneClassSVM. $MSE = 5.24$

(c) Optimized F-Transform with FAST feature detection. $MSE = 5.13$

(d) Optimized F-Transform with SURF feature detection. $MSE = 4.19$

Fig. 4.17 Different methods of finding an object on the image with method 3.3.2 and a compression ratio of 4 : 1

64:1 reduction

(a) Uniform F-Transform with compression ratio of 64 : 1. $MSE = 15.52$

(b) Optimized F-Transform with compression ratio of 64 : 1. $MSE = 4.73$

Fig. 4.18

(a) Optimized F-Transform with method 2.1. $MSE = 3.79$

(b) Optimized F-Transform with OneClassSVM. $MSE = 4.48$

(c) Optimized F-Transform with FAST feature detection. $MSE = 4.36$

(d) Optimized F-Transform with SURF feature detection. $MSE = 4.36$

Fig. 4.19 Different methods of finding an object on the image with method 3.3.1 and a compression ratio of 64 : 1

(a) Optimized F-Transform with method 2.1. $MSE = 17.68$

(b) Optimized F-Transform with OneClassSVM. $MSE = 21.92$

(c) Optimized F-Transform with FAST feature detection. $MSE = 21.3$

(d) Optimized F-Transform with SURF feature detection. $MSE = 19.8$

Fig. 4.20 Different methods of finding an object on the image with method 3.3.2 and a compression ratio of 64 : 1

Chapter 5

Conclusions and future lines

5.1 Conclusions

Throughout the project, we have developed several methods to improve the F-Transform that lead to some different conclusions we are now going to discuss.

First of all, in chapter 1 we have proposed different methods of accelerating the computation of both the F-Transform and its inverse. As a result, we have seen that the complexity has been reduced to the extent that the computational time has been scaled down several orders of magnitude.

In addition to this, the use of the approximation of the magnitude of the first derivative of the image to improve the distribution of the nodes also leads to better results compared to the uniform fuzzy partition.

The proposal in chapter number 3 of separating the object and the background can as well give better results than both the optimized and the uniform F-Transforms and specially if we want to get better results on the object and do not need to have good results on the background. From this section we can also conclude that out of the proposed methods, the one using the derivative as a reference to find the object and the first method to divide the number of nodes for the object and the background are the best ones.

To conclude, from the results of applying the F-Transform to other types of images, we can say that, as long as the background is constant and there are some clearly defined objects in the image, we can still use the optimized version of the fuzzy partition and get better results compared to the uniform partition. However, we can not say that this can be applied to any kind of image where no defined object can be found since the difference with respect to the uniform one is not that big and the computational cost of finding a better partition is much higher.

5.2 Future lines

After this project, there are many possibilities to improve the algorithm that go beyond the scope of this thesis. New methods to find the object in an image or different ways to approach the proposed ones can be studied to find a more efficient or better resulting way of separating it from the background. A further step to

follow is applying methods of detecting multiple objects over the image and then dividing the number of nodes so that the transform can be applied to images with several objects without creating false interest areas.

Bibliography

[1] L. Zadeh, "Fuzzy logic," *Computer*, vol. 21, no. 4, pp. 83–93, April 1988. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/53

[2] I. Perfilieva, "Fuzzy transforms: Theory and applications," *Fuzzy Sets and Systems*, vol. 157, no. 8, pp. 993–1023, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0165011405005804

[3] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[4] *The OpenCV Reference Manual*, 4th ed., OpenCV, October 2020.

[5] *The OpenCV Reference Manual*, 4th ed., OpenCV, April 2021. [Online]. Available: https://docs.opencv.org/4.5.2/df/d0c/tutorial_py_fast.html

[6] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443.

[7] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[8] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417.

[9] *The OpenCV Reference Manual*, 4th ed., OpenCV, October 2020. [Online]. Available: https://docs.opencv.org/4.5.0/df/dd2/tutorial_py_surf_intro.html

[10] A. Telea, "An image inpainting technique based on the fast marching method," *Journal of Graphics Tools*, vol. 9, no. 1, pp. 23–34, 2004. [Online]. Available: https://doi.org/10.1080/10867651.2004.10487596