Universidad de La Laguna

Escuela de Doctorado y Estudios de Posgrado

TÍTULO DE LA TESIS DOCTORAL

Consideraciones acerca de la viabilidad de un sensor plenóptico en dispositivos de consumo

---

AUTOR/A

Óscar	Gómez	Cárdenes

DIRECTOR/A

José Manuel	Rodríguez	Ramos

CODIRECTOR/A

José Gil	Marichal	Hernández

DEPARTAMENTO O INSTITUTO UNIVERSITARIO

FECHA DE LECTURA

05/11/2019

Universidad de La Laguna

Tésis

Consideraciones acerca de la viabilidad de un sensor plenóptico en dispositivos de consumo

Autor:
Óscar Gómez Cárdenes

Director:
Dr. José Manuel Rodríguez Ramos

Codirector:
Dr. José Gil Marichal Hernández

11 de septiembre de 2019

D. José Manuel Rodríguez Ramos, Doctor en Física, Profesor Titular del área de Tecnología Electrónica, adscrito al Departamento de Ingeniería Industrial de la Universidad de La Laguna, como Director,

AUTORIZA: a D. Óscar Gómez Cardenes, del programa de Doctorado en Ingeniería Industrial, a presentar esta Tesis Doctoral.

Lo que firmo, en La Laguna a 12 de Septiembre de 2019.

Dr. José Manuel Rodríguez Ramos
Dpto. Ingeniería Industrial
Universidad de La Laguna.

D. José Gil Marichal Hernández, Doctor en Informática, Profesor Contratado Doctor del área de Teoría de la Señal y las Comunicaciones, adscrito al Departamento de Ingeniería Industrial de la Universidad de La Laguna, como Codirector,

AUTORIZA: a D. Óscar Gómez Cardenes, del programa de Doctorado en Ingeniería Industrial, a presentar esta Tesis Doctoral.

Lo que firmo, en La Laguna a 12 de Septiembre de 2019.

Dr. José Gil Marichal Hernández
Dpto. Ingeniería Industrial
Universidad de La Laguna.

Dedico este trabajo a
mi mujer Nathália

II

Agradecimientos

Agradezco enormemente a mi director, el doctor José Manuel Rodríguez Ramos por permitirme realizar este doctorado.

Agradecer, especialmente a mi codirector, José, por compartir su brillante trabajo conmigo, así como por haber dedicado su esfuerzo y muchísimas horas de su tiempo.

A Nathália que ha estado apoyandome, aún en los momentos más difíciles.

Agredezco, por último, a mi familia, en especial, a mis padres, Vicente y Cristina y a mi hermano Sergio, por haber sido mi soporte sentimental y económico, y ya que sin ellos nunca hubiera sido posible.

Óscar Gómez Cárdenes

Contents

Agradecimientos **III**

1 Introduction **1**
1.1 Goals . . . 1
1.2 Background . . . 3
1.2.1 Active sensing . . . 3
1.2.2 Measuring depth passively . . . 4
1.2.3 Depth from stereo . . . 4
1.2.4 Depth from coded aperture . . . 9
1.2.5 Plenoptic lenses on smart-phone cameras . . . 9
1.2.6 Integral imaging mobile prototype modifying an attachable lens-style camera . . . 13

2 Shape From Focus **17**
2.1 Focal Stack . . . 18
2.2 General Algorithm . . . 19
2.3 Pre-processing . . . 20
2.3.1 Optical distortion correction . . . 22
2.3.2 Image registration . . . 23
2.4 Focus operators . . . 26
2.4.1 Laplacian operators . . . 28
2.4.2 Wavelets . . . 28
2.4.3 Ridgelets and Curvelets . . . 30
2.5 Sparse DMAP calculation: Maximum search . . . 34
2.6 Dense depth-map computation . . . 37
2.6.1 Optimization algorithms . . . 38
2.6.2 Guided Filter . . . 39
2.6.3 GDT: Geodesic Distance Transform . . . 46

2.6.4 Algorithms comparison 48
2.6.5 Solving artificial edges 48
2.7 All-In-Focus computation 50

3 Discrete Radon Transform extensions for our Curvelet transform 57
3.1 Curvelets and DRT 57
3.2 Conventional 2D discrete Radon transform 59
3.2.1 Forward multi-scale discrete Radon transform 62
3.2.2 Radon ad-joint transform 66
3.2.3 Radon inverse transform 67
3.3 DRT variations based on pruning 69
3.3.1 Central DRT 70
3.3.2 Periodic DRT 76
3.3.3 Comparison of Central and Periodic DRT 78
3.4 Simultaneous computation of DRT quadrants for its efficient implementation 81
3.4.1 Founding idea on Perimeter DRT 84
3.4.2 Data structure 85
3.4.3 Algorithm 88

4 DRT accelerated on smart-phones 93
4.1 Introduction 93
4.2 Parallelism, multithreading, autotuning 93
4.2.1 Coarse grained parallelism 93
4.2.2 Vectorization and SIMD 94
4.2.3 OpenCL 96
4.2.4 Halide 98
4.2.5 Conventional DRT, Central DRT, Periodic DRT and Perimeter DRT implementations in Halide 100
4.2.6 GPU 103
4.3 Time measurements 110
4.3.1 Exploiting parallelism on the CPU 111
4.3.2 Exploiting parallelism on the GPU 111
4.4 DRT applications 113
4.4.1 SFF - Curvelet with Periodic DRT 113
4.4.2 Bar-code detection with Central DRT 117

5 Conclusions **131**

Índice de tablas

2.1 Measured times on an Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz processor for a focal-stack of size $N \times W \times H = 9 \times 1024 \times 1024$. 37
2.2 Measured times on an Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz processor for input *sDMAP* and *guide* images of size $W \times H = 1024 \times 1024$. All times are calculated using an average of 30 executions. 48

List of Figures

1.1 Simple stereo optical arrangement. 5
1.2 Triangulation. 6
1.3 Search lines. 7
1.4 Example taken from the bench-marking website [1]. 8
1.5 Camera lens with occluder. 9
1.6 Photographic camera example. 10
1.7 Standard and plenoptic arrangements. 11
1.8 Detail of a plenoptic image. 11
1.9 The Lytro Illum. 12
1.10 Two camera array modules. 12
1.11 Working prototype. 13
1.12 Hardware units of the prototype. 14
1.13 Sensor module. 14
1.14 Processed real scene. 16

2.1 Focal stack representation. 19
2.2 SFF algorithm. 21
2.3 Calibration printed pattern. 23
2.4 Automatic distortion correction on a focal-stack. 24
2.5 Focal stack with motion. 27
2.6 Laplacian operator example. 29
2.7 Wavelet focus operator example. 30
2.8 Wavelet as a 1D filter. 31
2.9 Comparison of hard thresholding on different transforms. . . . 32
2.10 Subbands of middle and high frequency. 33
2.11 Detail of the eye. 33
2.12 Slicing on a focal-stack. 34
2.15 Computing the maximum without a support window. 35

2.18 Computation of the maximum using a support window. 36
2.21 Wave-front on a depth-map. 39
2.22 The weights are modified by the guide. 40
2.23 Using guided filter to obtain a dense depth-map 42
2.24 Result of executing the algorithm in 1 44
2.25 Execution of the colorization using Guided Filter algorithm. . 45
2.26 Dense depth-map computation using *GDT*. 48
2.27 Execution of the algorithm 3. 50
2.28 Computation of an all-in-focus using equation 2.12. 50
2.13 The measurements on a textureless zone are predominated by noise. 51
2.14 An edge location will give a good focus measurement. 52
2.16 A depth-map is calculated using different support window sizes. 53
2.17 Threshold applied to computation of depth-maps 54
2.19 Comparison of several focus operators. 55
2.20 An execution of the colorization algorithm. 56

3.1 DRT computation. 60
3.2 Number of pixels summed on a DRT. 61
3.3 Whole set of discrete lines computed by conventional DRT. . . 64
3.4 Behavior of the 4 quadrants of a DRT. 65
3.5 The ad-joint discrete Radon transform. 68
3.6 First 5 iterations of recursive multi-grid inverse DRT. 68
3.7 Central DRT computation. 69
3.8 Periodic DRT computation. 70
3.9 Whole set of discrete lines computed by Central DRT. 71
3.10 Memory patterns on a Central DRT. 71
3.11 Relations of data in the computation of two columns. 73
3.12 N-length vectors to be added at each stage. 74
3.13 Whole set of discrete lines computed by Periodic DRT. 77
3.14 Comparison of conventional, Central and Periodic DRT. . . . 79
3.15 Ad-joint operators applied to conventional, Central and Periodic DRTs. 80
3.16 Forward and back-projected conventional, Central and Periodic DRT. 82
3.17 Quality of the inversion for conventional, Periodic and Central DRTs. 83

3.18 Conventional and dyadic squares based lines at several scales for the first quadrant. 84
3.19 Enumeration of vertexes at the perimeter of a square and triangular shape of the structure containing its Perimeter DRT. . 86
3.20 Intersections for every displacement for minimum, intermediate and maximum slope for first, second, third and fourth quadrant of a 8×8 problem. 88
3.21 Cuts at dyadic boundaries in a 8×8 problem. 90
3.22 Sums carried out to compute 3 line segments. 90
3.23 Perimeter DRT and rearrangement as a conventional DRT of an input image. 91

4.1 Two access patterns. 103
4.2 Measured times of the executions of several algorithms without parallelism. 111
4.3 Measured times of our algorithms using auto-scheduling on an Intel i5. 112
4.4 Measured times of our algorithms using auto-scheduling on an Intel i9. 112
4.5 Measured times of our algorithms using auto-scheduling on a Qualcomm Snapdragon 845. 113
4.6 Measured times of our algorithms using GLSL on a GeForce 1070. 114
4.7 Measured times of our algorithms using GLSL on an Adreno 540. 114
4.8 An image and its approximate DRT. 116
4.9 An image and its decomposition in 3 sub-bands using Wavelets. 117
4.10 Recovered depth-map of a synthetic cone using Wavelets. . . . 118
4.11 Recovered depth-map of a synthetic cone presenting discontinuity using Wavelets. 119
4.12 Shape-from-focus using aDCT focus estimator of a real scene. 120
4.13 Shape-from-focus using aDCT focus estimator of a real scene containing discontinuities. 120
4.14 Comparison of continuous transform, classical DRT and proposed DRT. 126
4.15 Examples of bar-code localization. 127
4.16 More examples of bar-code localization. 128
4.17 Steps 1-4 of the method described in section 4.4.2 129

4.18 Steps 5-10 of the method described in section 4.4.2 130

Resumen

La medida pasiva de distancia a los objetos en una imagen da lugar a interesantes aplicaciones con capacidad para revolucionar la fotografía. En esta tesis se creó y estudió un prototipo de cámara plenóptica para dispositivos móviles. Esta técnica presenta dos inconvenientes: la necesidad de modificar el módulo de cámara y la pérdida de resolución. Por ello, el prototipo fue descartado para utilizar otra técnica: la profundidad a partir del desenfoque.

En esta técnica el método de captura consiste en tomar varias imagenes variando la distancia de enfoque. El conjunto de imágenes se denomina focal-stack. Se estudian distintos operadores de desenfoque, que dan una medida de desenfoque por pixel y por plano del focal-stack. Siendo elegido como óptimo el operador de desenfoque curvelet, que es computacionalmente más intensivo que otros operadores pero es capaz de descomponer imagenes naturales utilizando muy pocos coeficientes.

Para hacer posible su uso en dispositivos móviles se construye una nueva transformada curvelet basada en la transformada discreta de Radon. La transformada discreta de Radon tiene complejidad linearítmica, no utiliza la transformada de Fourier y usa sólo sumas de enteros.

Por último, se analizan distintas versiones de la transformada de Radon con el objetivo de conseguir una transformada aún más rápida y se implementan para ser ejecutadas en dispositivos móviles.

Además se presenta una aplicación de la transformada de Radon consistente en la detección de códigos de barras con cualquier orientación en una imagen.

Abstract

Passive distance measurement of the objects in an image gives place to interesting applications that have the potential to revolutionize the field of photography. In this thesis a prototype of plenoptic camera for mobile devices was created and studied. This technique has two main disadvantages: the need for modifying the camera module and the loss of resolution. Because of this, the prototype was discarded in order to utilize another technique: depth from focus.

In this technique the capture method consists in taking several images while varying the focus distance. The set of images is called focal-stack. Different focus operators are studied, which give a measure of defocus per pixel and plane of the focal-stack.

The curvelet based focus operator is chosen as the most adequate. It is computationally more intensive than other operators but it is capable of decomposing natural images using few coefficients.

In order to make viable its usage in mobile devices a new curvelet transform based on the discrete Radon transform is built. The discrete Radon transform has logarithmic complexity, does not use the Fourier transform and uses only integer sums.

Lastly, different versions of the Radon transform are analyzed with the goal of achieving an even faster transform. These transforms are implemented to be executed on mobile devices.

Additionally, an application of the Radon transform is presented. It consists in the detection of bar-codes that have any orientation in an image.

Chapter 1

Introduction

1.1 Goals

The photographic camera has been around for almost two centuries already, with the first permanent photography dating back from 1826 and taken by Joseph Nicéphore Niépce. The inner workings of this marvelous contraption were kept untouched, having a physical element that reacted to the light passing through an aperture or a lens and producing as a result a rendition of the scene, where rays of light were interacting with the objects present and finally entering the camera. But then, computers came along and made things pragmatically different for everyone. Photography was then disturbed by the unstoppable digital revolution.

The digital camera was born and film was replaced by digital memory and digital sensors. Still, the principles of photography as an art remained similar, in the sense that, in order to take a picture, we have to take into consideration aperture, exposition time, sensitivity of the sensor or the film, field of view, etc. But what if the next incarnation of photography changed that?

Computational photography enables an array of applications and new possibilities by adding a processing step in between of exposure and 'developing'. We have already seen the first examples on the new smart-phones, those carrying both a camera and a 'computer' closely attached. Things like portrait or bokeh mode are examples of synthetic aperture. High Dynamic Range, HDR, algorithms vary the exposition time and/or the sensitivity (ISO) in order to compose an image that has far more dynamic range

than what would have been possible with a single exposure. It is also possible to improve the color sensitivity and signal to noise ratio in low-light environments by taking a burst of images and combining them to increase the signal to noise ratio. This kind of software 'tricks' have been seen on the Google Pixel smart-phones [2].

But then, what if photography can be 'salted' with a conventionally absent information about objects in a scene? Their distance to the camera, also called depth. Depth sensing is an active field of development and research. It is widely used in art, self-driving cars, consumer electronics and industrial applications. There are two kinds of depth sensors depending the on principle they operate: the active depth sensors and the passive depth sensors.

The goal of this thesis is to analyze the viability of measuring distance on mobile devices passively. Measuring depth means that with a shot a depth map is obtained in addition to the image. This is also called 2.5 D. It enables an array of applications:

- Refocus at will. Images can be refocused after being taken.
- Synthetic aperture. Different aperture combinations can be generated after taking the image. An example of this is the bokeh effect where the background is blurred in order to isolate the subject from the rest of the scene. An effect that in conventional photography requires a very bright, and expensive, objective lens.
- Chroma key. By knowing the depth of the pixels of the scene, the background can be removed or substituted, similar to what a chroma key achieves in cinema special effects, but without the need for recording against a green background.
- Point of view generation. Versions of the image as if taken from points of view slightly different to the actually used to capture it, can be generated synthetically.

At the end of this chapter, after introducing background information about depth extraction from images, a plenoptic camera prototype for mobile phones is introduced. This approach was later discarded since the method incurred in some drawbacks, the loss of resolution being the most compelling.

The second chapter takes a different approach, shape from focus. Those techniques use a single lens and infer depth by varying the lens focus distance. In that chapter, the general algorithm is described and a set of focus operators are analyzed. One particular focus operator is chosen: the curvelet transform. The rest of the thesis is devoted to develop a curvelet transform fast enough to run on a mobile device. The Discrete Radon transform is chosen to be the basis on which the curvelet transform is built, but even then, it must be extensively modified and optimized. That task will be the subject of the third chapter, in which different versions of the Radon transform are described. In the fourth chapter the chosen Radon transform is implemented and accelerated. As well a collateral application, to locate bar-codes in images, is shown.

1.2 Background

1.2.1 Active sensing

The active sensors emit a signal to the media in order to measure how is the reflection once it has interacted with the scene and is brought back to the sensor. Using these kind of sensors has several advantages compared to using passive sensors. The most important one is quality: it is easier to measure the differences on the signal if you know it *a-priori*. Albeit being more reliable they also come usually with the drawback of being notably more expensive.

The *LiDAR* is an example of active sensor. It illuminates the scene by pulsating laser light and then measures the pulses with a sensor. It allows for high-resolution maps. The *Velodyne LiDAR* [3] costs about $4k$ U.S. dollars. It seems overkill to include a technology like this on a smart-phone, so many efforts have been put into using different kinds of sensors.

Microsoft Kinect is a module intended for the Xbox, a gaming console. It packs an active depth sensor consisting of a monochrome CMOS sensor and an infra-red light projector that emits pseudo-random IR patterns. It was very popular not only among the *gamers* but also among the *makers* and researchers, as it was reverse-engineered and used for more general purposes.

After Microsoft's footsteps, Apple introduced in some of its iPhones a dot-projector and a infra-red camera [4]. Qualcomm has announced the *Second Generation Spectra Module Program* [5], an image signal processor, ISP, that will allow smart-phones carrying a Qualcomm's System on Chip, SoC, to

pack active IR depth sensors. It still contemplates that most smart-phones will not rely on expensive sensors but those on the cheaper side: of the time-of-flight kind. Algorithms to process depth passively from stereo will still be present giving rise to hybrid passive-active sensing.

1.2.2 Measuring depth passively

There are many techniques to achieve depth maps from passive systems of capture and most of them need some sort of modification of the camera. The most natural approach is to imitate the human vision. Nature gave us what we call binocular vision. This type of vision uses a pair of eyes and a brain to process the difference, the binocular disparity.

By using two cameras we can artificially recreate the binocular capture system and by using a processing unit running an algorithm on these pairs of images we can imitate the brain understanding the depth of the environments. Before having enough computational power to do this, the binocular camera already existed if only for the sake of capture, storage and transmission, with no depth inference until now.

1.2.3 Depth from stereo

Algorithms for calculating depth from stereo use two images taken at the same time from slightly different points of view. Objects closest to the capture system will have more *disparity* than those further away. Disparity is the difference of the position of corresponding 3D point projected on pixels to both images. From the disparity, a depth map can be calculated by taking into consideration the characteristics of the capture system.

The main difficulty that depth from stereo algorithms attempt to solve is the unequivocal identification of the pixels on both images where a scene point P projects. Success in this task is what basically determines the quality of the results for different algorithms. In order to find the depth of the points shown in one of the two images, the template image, we take each projected point P projected onto the left image at $X_l = (x_l, y_l)$) and we search for the corresponding projected point on the candidate (right) image ($X_r = (x_r, y_r)$). Also, there could be occlusions so the success of the search is not guaranteed. In order to accelerate the process we can use the epipolar geometry constraint. In that case the space of search on candidate image reduces to a line.

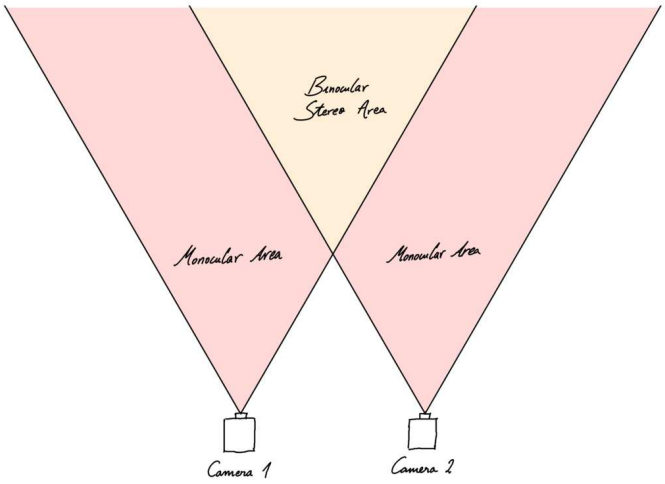


Figure 1.1: Simple stereo optical arrangement. Depth can only be measured at the binocular stereo area.

The simplest optical arrangement capture model is to use two cameras for which the optical axes are parallel (see figure 1.1). Then, depth can be estimated by using a triangulation (see figure 1.2). The X_l and X_r points are located by looking for the point along an epipolar plane i.e. the plane that contains the points X_l, X_r and P. This yields a scan line that is parallel for both images (left and right).

A generalized optical arrangement is also valid. This model is useful because it maximizes the binocular stereo area. In this case the scan lines of both images are not parallel.

So, once known the parameters of the camera, all boils down to the correspondence problem. For each pixel, we need to find where is its new position along the epipolar line. One way to do this is to use a general correspondence algorithm and restrict its search space to the epipolar line. The other way consists on rectifying (or *warping*) the images so their epipolar lines are corresponding horizontal lines. This approach can lead to more efficient algorithms.

The first stereo matching algorithms used to look for suitable features that were easily identifiable on the search space. Modern algorithms produce dense depth maps. Texture-less areas have a high degree of uncertainty as they do not have unequivocally identifiable features, so these algorithms have

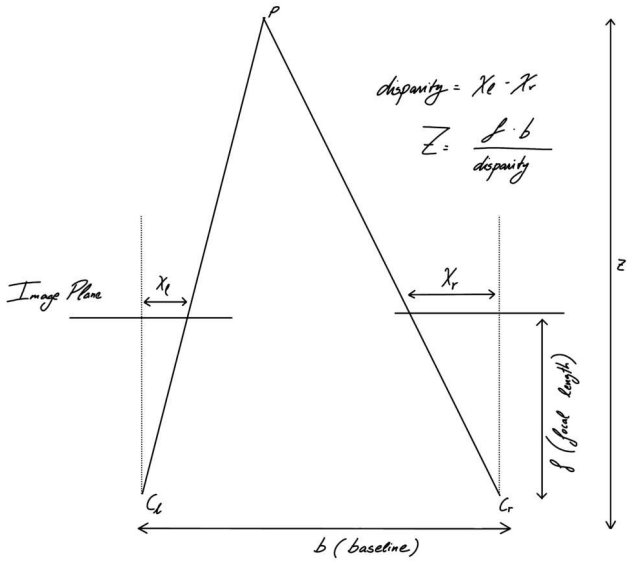


Figure 1.2: Triangulation. Baseline and focal distance must be known beforehand.

to make assumptions on how should they fill the unknown areas.

Local matching strategies try to match a template area of pixels with a candidate area. Using only one pixel is not robust against noise so a support region must be used. The support region can be defined using different techniques having different computational impacts. The simplest methods are using a box shaped window around the pixel or a gaussian window. Segmentations, graph-cuts and geodesic support weights are more advanced techniques and have also been used widely. The size of the window affects the precision of the results. A bigger size will improve robustness to the noise but will decrease the precision of the smaller details.

After the sparse map of disparities is computed, a global optimization is performed in order to *fill* the *gaps*. Simulated annealing, graph-cuts, dynamic programming and segmentation are only some examples of techniques that have been used to solve this problem. A complete comparison of these methods is described in [6].

Lastly, in the last 5 years, many algorithms based on neural networks have been developed and, in many cases, achieving better results than those of more classical approaches, for example [7].

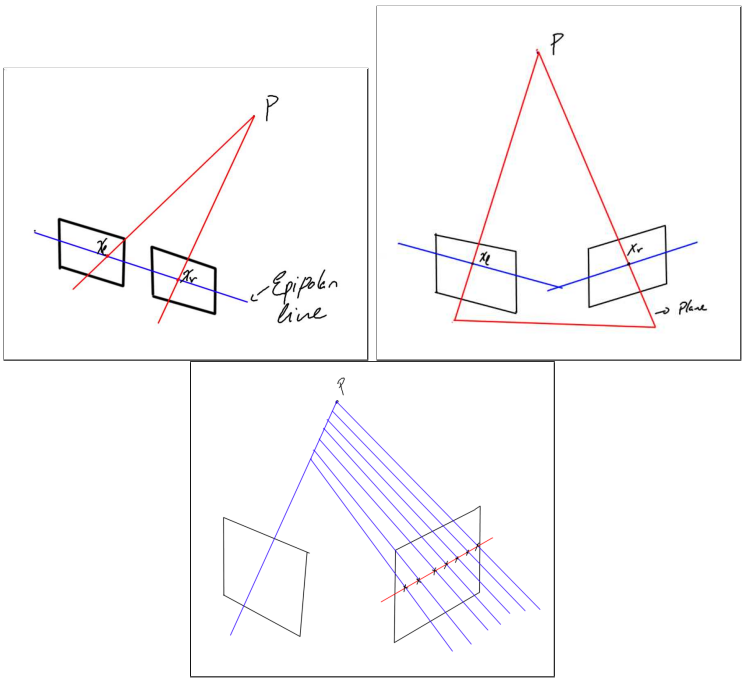
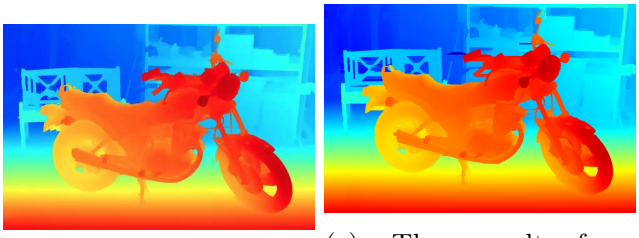


Figure 1.3: Search lines. On the left, the simple version, epipolar lines are parallel; right, epipolar lines are not parallel but the model is still valid.

(a) Left input image. (b) Right input image. (c) Ground truth.



(d) The result from a method that uses a convolutional neural network [8], taking 925 ms.

(e) The result from a "classical" method, based on super-pixel segmentation [9], taking 635 ms.

Figure 1.4: Example taken from the bench-marking website [1].

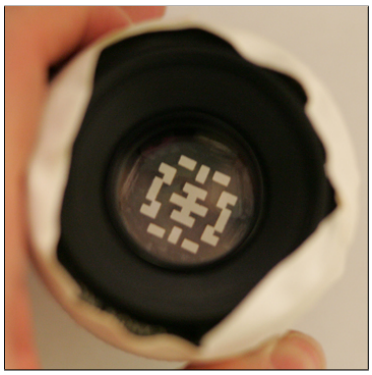


Figure 1.5: Camera lens with occluder, adding a *coded aperture*. Taken from [10].

1.2.4 Depth from coded aperture

Another possibility is to add a *coding* to the aperture. This entails a simple modification to the lens that consists on adding an occluder with a particular shape. The result of this modification is that objects that are out of focus appear blurred with a scaled version of the aperture shape. This scale is directly proportional to the depth of the object. While a pentagonal disk shape, typical of inexpensive camera diaphragms, provides some depth cues, they are difficult to exploit, so a specific aperture is designed in order to be easily measured in size i.e. with less uncertainty.

The work presented in [10] is able to get an all-in-focus image along with a coarse depth map from a single image.

1.2.5 Plenoptic lenses on smart-phone cameras

The plenoptic function

Physical space is filled with light rays that travel in every direction. These rays depend on how the light is emitted, reflected and refracted by the surfaces of the 3D objects located on the space, and on the physical medium of propagation. The *plenoptic function*, also called *light-field*, represents the amount of light traveling in each direction in each point of the space [11]. This function has 7 dimensions. Three of them define the point in 3D space, other two define the 2D angular direction of observation and the remaining two define the time and wave-length. Therefore, the function is defined as

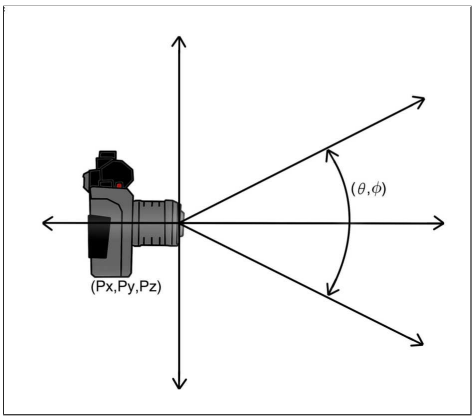


Figure 1.6: Example with a photographic camera. The dimensions p_x, p_y and p_z are fixed on a point. The angular dimensions θ and ϕ are restricted to a range

follows:

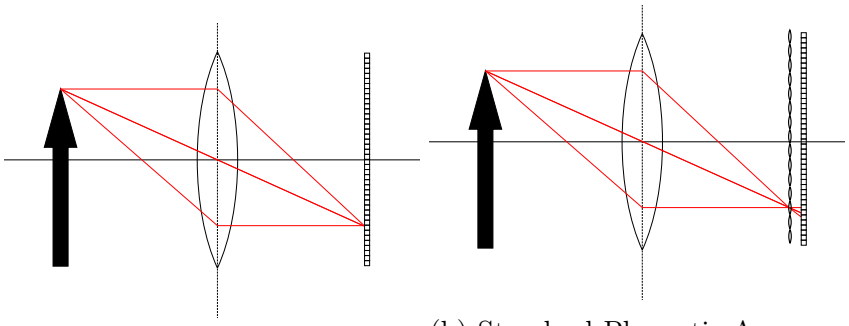
$$P(p_x, p_y, p_z, \theta, \phi, \lambda, t). \tag{1.1}$$

In a standard camera, each element of the captured image is the result of integration of rays that hit a particular sensor location (at p_x, p_y, for a fixed p_z), coming from the range of angles covered by the lens (θ and ϕ dimensions). As a result, we lose this information and a 2D image is *projected*.

If we want to capture angular information about the incoming rays, along with spatial information, we must use a different optical arrangement. Stereo photography allows to get 'full' angular information, but restricted to two spatial locations, so that is not the way to go. Instead a plenoptic camera must be used to capture a range within the angular dimensions. A plenoptic camera can be implemented by adding a *micro-lens array* in front of the sensor 1.7.

Existing technologies

The Lytro company, founded by Dr. Ren Ng, has commercialized two plenoptic photographic cameras. The first one was the first widely available to the general public. It was capable of *a-posteriori* refocusing and point of view generation and had a final resolution of 1080×1080, taking 8 seconds in doing so. It used to cost around 400 $. The second one was called Lytro



(a) Conventional arrangement. (b) Standard Plenoptic Arrangement.

Figure 1.7: By adding a micro-lenses array rays with different angles are redirected to different pixels, allowing angular information to be registered.

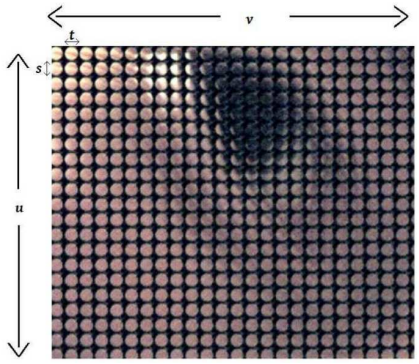
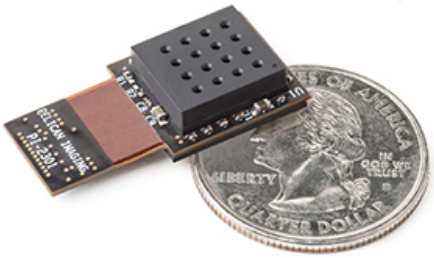


Figure 1.8: Detail of a plenoptic image. s and t are the dimensions corresponding with the pixels below each micro-lens and u and v correspond with the macro-pixel, i.e. the micro-lens within the image.

Figure 1.9: The Lytro Illum.



(a) The PiCam made by Pelican.

(b) The LinX camera modules.

Figure 1.10: Two camera array modules.

Illum (see 1.9), and costed 1500 $. It was powered by a snapdragon 801, which is the SoC that high-end smartphones were packing at the moment of release. This product failed, as not much people bought it. The image quality was probably not on par with DSLR cameras and the resolution still low (2450×1634). A 700 Megapixel plenoptic camera for cinema was being developed when the company was declared defunct on March, 2018.

The company Pelican Imaging Corporation designed a camera array for smartphones called *PiCam*. Similarly to the Lytro cameras, light-field can be captured and then perform many applications such as refocus, super-resolution and depth map generation. The capture system has not been adopted by the industry. A similar technology has been developed by the company LinX.

Figure 1.11: Working prototype. The modified Sony QX-100 is attached to a Samsung Galaxy S5 running Android 5.0.

1.2.6 Integral imaging mobile prototype modifying an attachable lens-style camera

In 2014 we implemented our own version of a plenoptic camera using smartphones. It uses a modified Sony QX-100 camera that can be attached to any smartphone running our software on Android.

Previous to this work, a prototype was built. CAFADIS, the plenoptic camera patented by the University of La Laguna [12]. It consisted in building a camera to measure wave-front phases and distances and was applicable to multiple scenarios (small and large scales) and involved the use of GPUs and FPGAs in order to work in real-time. This prototype was successful and led to this present work.

A new prototype was then developed (see fig. 1.11) to provide a good user experience and is, at the same time, inexpensive. Our goal was to build a handheld device that showcases an array of applications for integral imaging that may be present in a future generation of smartphones. Some of these applications are: to refocus after capture, synthetic aperture, depth map generation, and 3D displaying (both stereoscopic and integral).

The chosen camera for the prototype was the Sony QX-100. It makes possible to have a lens style module with good specifications while being totally independent to the unit that processes the images. The camera is controlled by an Android smartphone or Tablet via Wi-Fi. Once the image is ready, it will be transferred using the Wi-Fi connection to the Android device. The camera was modified to fit a micro-lens array close to the sensor. The application that deals with the user interaction runs on the Android device.

Figure 1.12: Basic diagram showing the hardware units of the prototype.

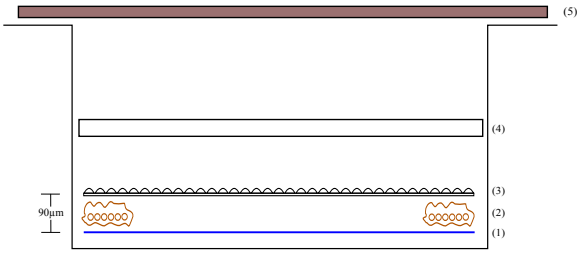


Figure 1.13: Representation of the module that hosts the sensor. Elements 1, 4 and 5 were present in the original module while 2 and 3 were added. (1) Sensor; (2) Liquid with micro-spheres; (3) MLA; (4) Clear pane; (5) UV/IR cutoff filter.

The flow of the process starts after the captured image is loaded. The first step is the image correction. After the image is corrected, the refocusings are calculated and shown. Lastly, a depth map is calculated and a 3d effect is shown. A stereo pair and an all-in-focus image are also generated.

The standard plenoptic camera model [13] was used in order to build this prototype. It has micro-lenses placed one focal length away from the image plane of the sensor. If we compare this arrangement with a normal camera, one can note that the only new item it has is the array of micro-lenses (see fig 1.7). This arrangement trades spatial resolution off in order to achieve angular resolution. The camera has enough room to trade some of the spatial resolution off and still get acceptable images and hence it is adequate to be used for the prototype.

The applied modification consists of adding a micro lens array just in front of the sensor and before the main lens (see fig 1.13). The sensor is covered with two protective layers that have to be removed before proceeding to place the micro-lens array (*MLA*).

The addition of the *MLA* and the sole nature of the computational photography makes necessary some process in order to determine which zone of the sensor is creating image by each micro-lens, how many pixels it extends to and what coordinates it maps to. This is, in other words, to measure the individual PSFs (Point Spread Functions) of each micro-lens.

A previous offline process of calibration has been made in order to measure these data. This process must be executed for each zoom distance f. Also, each time an image is taken in the prototype, a correction is needed in order to have a correct distribution of micro-lenses along the vertical and horizontal axis of the image. In our case, this is crucial because the *Sony QX-100* adds an automatic step of pincushion distortion correction which plays against the placement of the micro-lenses. Also, inside each micro-lens, we need to correct the distortion that the optical arrangement creates. This inner distortion (inside each micro-lens) depends also on the position of the micro-lens relative to the sensor in the x and y axis.

Refocus One interesting application of the plenoptic camera is to be able to synthesize a new image that is focused to a certain plane. In the article, [14], a method for refocusing plenoptic images was described in detail, the 4D:3D Radon Transform. The method is able to refocus images in a linear time and does not require complex number transforms, trigonometric calculus, multiplications or float numbers. It was also accelerated to be able to run 16 megapixels images within seconds, taking advantage of multi-threading in native code and NEON SIMD instructions.

Since the usable set of pixels of each micro-lens for the prototype is 8×8, the number of planes that the image can be refocused to is 15.

Distance map In order to obtain a depth estimations the light-field data is used directly without any intermediate computation step, except calibration as shown in figure 1.12. The method is presented in [15] was used in this prototype. This method was then discarded when it was decided to use focal-stacks instead of plenoptic images and it is explained in chapter 2.

Once disparities are obtained, it is straightforward to render all-in-focus images or stereo pairs using the disparity map and the color data recorded by the plenoptic camera.

One of the consequences of using *MLA*s is that the spatial resolution of the sensor is traded off in order to get angular information. This means

Figure 1.14: Real scene taken and processed in our prototype. On the left, depth-map; right, all-in-focus image.

that the refocusings, all-in-focus or 3D representations will have a resolution that is lowered by a factor compared to the whole sensor resolution. This is why we decided to taken another approach, since this disadvantage makes it impossible to make it to market.

In the case of our prototype, from the 20 Mpx SONY QX-100 sensor (5472×3648 pixels), we only use 2070×1355 pixels, then we start from a 2.8 Mpx sensor and the computed images that it produces are only 414×271 pixels. By using a super-resolution algorithm [16], the final resolution can be increased.

Finally, our algorithms can be highly parallelized by using vectorization, thread-level parallelization or GPUs. The only optimized algorithm in the current implementation of the prototype was the refocus algorithm. For example, the Samsung Galaxy Note 10.1 was able to compute the refocusings of focal stack in 0.2 seconds. The same tablet produced a depth map in 23.73 seconds for a input resolution of 2.6 Mpixel.

Chapter 2

Shape From Focus

Using a micro-lens array, MLA, has two main drawbacks. In order to use them you need to modify your capture system. In this case, the camera must include a *MLA* that needs to be inserted in front of the sensor. This is not likely to happen in a small camera module from the kind used for smartphones. The other inconvenient is that we are sacrificing spatial resolution in order to gain angular resolution, effectively dividing the sensor resolution and resulting on images that are usually about only the 30% of the whole resolution.

If we are aiming at a smartphone implementation, we want to think of a solution that does not impose these kind of restrictions. So, our solution should only use the widely available hardware, already present on smartphones. From the standpoint of the scope of this work, it makes sense to invest time on researching a closely related group of algorithms called *shape from focus*.

Shape-from-focus, SFF, pertains to the category of passive methods, and consists on measuring the 3D geometry of the scene by photographing it through a wide open aperture lens from the same point of view, while varying the focal distance so that a set of images whose depth of field sweep the distances in the scene is acquired and used as input to a focus estimator. The arguments of maxima on each pixel along the focal stack constitutes a map of depths, describing the shape of the scene. A seminal work where the term shape-from-focus was coined is due to Nayar and Nakagawa[17]. Recently some authors even propose SFF methods that infer all the range of depths using only one image[18].

The cue in this type of passive method comes from the fact that the

photographic images of objects outside of the depth of field region will not exhibit high frequencies. As images are taken to adequately cover a range of distances of interest in the scene with non overlapped depth of fields, there will be some plane where the rays pertaining to that object are sharper than in any other plane. Obviously if the objects are texture-less to start with, this cue is insufficient. Depth maps reconstructed with this method usually do not supply a solution per pixel –rays impinging centered at the same point on the surface of an object, to be precise– because simply there are features that will never get sharp, and measuring this way will never tell when they get into focus. If a dense map is required, the solution is somehow extended to "guess" those uncertain pixels and then is regularized attending to other cues. Another problem arising with this method is that isolated pixels are insufficient to measure focus. Instead, a region of support around the rays will be needed, but then the frontiers between objects at different depths will become problematic.

Apart of the problems concerning existence and uniqueness of the inversion, another set of problems affect to the acquisition process solely. In order to obtain a complete in range, but non overlapped, sweep of depth of field imply changing the focusing distance of the lens whilst the rest of photographic parameters remain fixed. But these changes of focus distance will normally imply, as collateral effect, different magnification and radial distortions. Thus, an aligned version of this focal stack, where a feature in the scene can be located precisely at the same pair of pixel coordinates through all planes is necessary to determine depths accurately, but it is far from being trivial to obtain.

Also these focal stacks are normally obtained through time multiplexing, and therefore are only applicable to static scenes.

2.1 Focal Stack

These kind of algorithms are based on estimating locally the depth of a point, and then combining the measured information in order to create a depth map. The algorithm used to measure the focus level for every image pixel can be referred as *focus measure* operator. These operators can be used for both auto-focus and SFF applications.

For our algorithm the input will be a focal stack of N images. The characteristics of the camera will determine the accuracy and the range of

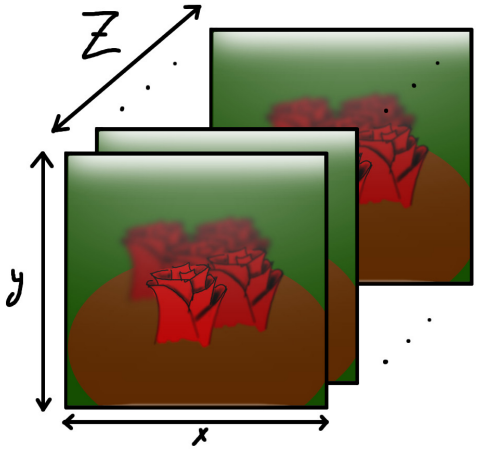


Figure 2.1: Representation of a focal stack of $N \times W \times H$ pixels. The dimensions that represent the 2D spatial coordinate are named x, y and the position that corresponds with its focal distance is named z.

depth that it can measure. Given that the algorithm measures blur, its size must be measurable, so at least the radius of the blur has to be greater than one pixel. Moreover, the smallest measurable defocus is limited by the Airy disk of the capturing device.

The focal stack consists of N images and therefore will have 3 spatial coordinates (see 2.1). We define the 3D focal stack image $W \times H \times N$ where W is number of pixels in width, H is number of pixels in height, and N number of pixels in depth, i.e. the number of images taken in the focal stack as the set:

$$F = \{F(x, y, z) : 0 \leq x < W, 0 \leq y < H, 0 \leq z < N\} \tag{2.1}$$

2.2 General Algorithm

In order to get a depth map from a focal stack there are three main steps. First the amount of focus of each pixel of each image of the focal stack is calculated. This step has as input each image of the focal stack and gives as output a *processed focal* stack where each image is a map of intensities where

each pixel contains a measure of how focused a pixel is.

The second step is to build a *sparse depth map*. In order to do so, the Z-axis of the *processed focal stack* is analyzed. The algorithm will measure the amount of focus for each point along the focal stack in order to find or interpolate the maximum value. This focal plane corresponds to the best focused version of the point and therefore the focal distance is a measure of the distance of the point on the 3D space. If the noise is so big that the measured maximum is not the right one, a false positive will appear on the results. The effect of the noise on the measured amount of focus depend on the focus operator.

Depending on the amount of measurements several strategies can be chosen in order to find the maximum measured focus: find the maximum directly from measured points, fit a curve from measured points and calculate its maximum, or low pass filter the measurements and then find the maximum. If the algorithm does not have enough certainty of the measured distance, the value remains null.

This step takes as an input the *processed focal stack* and outputs a *sparse depth-map*, that is, a map containing depths values only for those on which the algorithm has had enough certainty. Usually the lack of certainty is caused by a lack of textures or edges; or caused by a low signal to noise ratio.

The final step is to estimate a value for those null values remaining after step 2. This can be done following many different approaches where an algorithm has to "guess" the distances and usually will use as input the sparse depth map and the color information in the focal stack, because it will infer that similar colors mean pertaining to the same object and so being at the same distance.

The whole process is represented in figure 2.2.

2.3 Pre-processing

In order to execute the algorithm described in 2.2, the pixels in different z values for corresponding (x, y) coordinates should belong to the same 3D point.

A telecentric lens that has its entrance pupil at infinity produces orthographic projections of the scene. In this case, image magnification is independent of the object's distance or position in the field of view. Usually, when changing the focal distance of the camera, the magnification will change, as

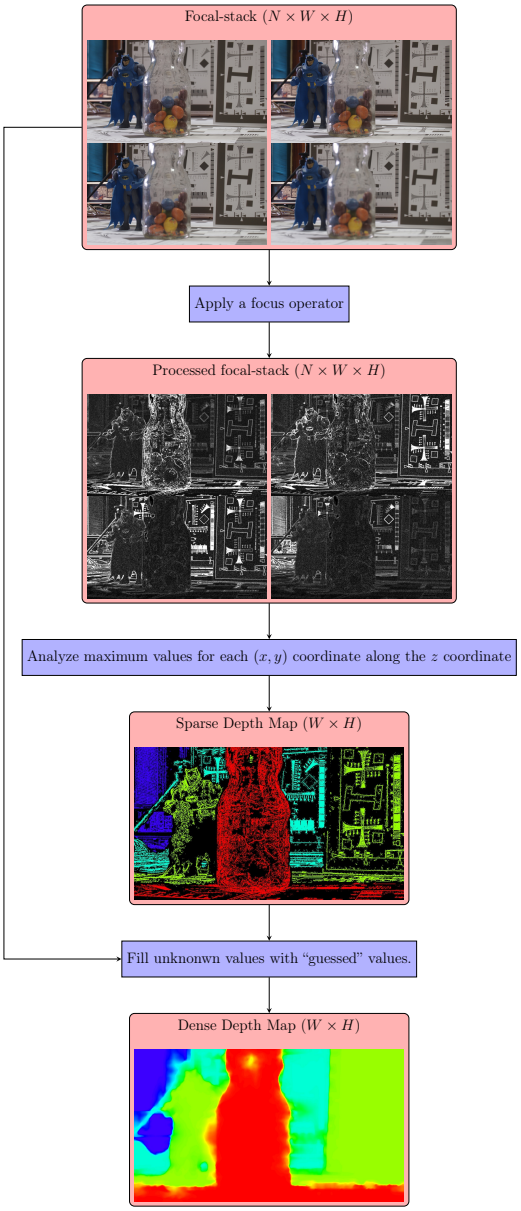


Figure 2.2: Shape from focus generic algorithm. Processes are shown in blue and data is shown in red.

what happens in reality is that the distance between the sensor and the lens varies. So in this case, this magnification can be easily corrected since a magnification amount exists for each focal distance and can be calculated or measured.

In the more general case of a non-telecentric lens,when changing focal distance, in addition to the magnification, another effect will take place, the change in *perspective*, produced by the variation of focal length. This effect is small or non-existent in expensive complex lenses. For our purposes, we are considering this perspective shift non-existent, or negligible and treated as noise.

The other effect happening when focal distance is changed is a variation in lens distortion, in the form of a barrel or pincushion distortion. This effect is very noticeable and can ruin the results if not corrected, especially on smart-phones.

2.3.1 Optical distortion correction

By knowing the *transfer function* of the capture system for each focal setting, it is possible to remap each image in order to achieve a focal stack where each (x, y) location back-projects to the same 3D ray in the scene. In fact it is enough finding the *transfer function* that translates one central image of the focal stack to the rest of them.

Camera calibration is usually carried out by using an object that contains a special pattern that allows for an algorithm to calculate the camera parameters. This procedure is called *photogrammetric calibration*, and usually a 2D printed checkerboard pattern 2.3 is used [19]. If we do not use a calibration object and rely on the rigidity of the scene in order to calculate the parameters the procedure is called *self-calibration*.

Calibration with defocused images

Since by definition of our solution we will always have blur in our images, most method based on photogrammetry will fail, as they look for patterns that disappear or become unreliable when they are blurred. A camera self-calibration algorithm is used instead. Our calibration algorithm will use as an input a pair of images with different focal distance and will find the difference in camera parameters. Because of this our algorithm will be simpler than those found in the literature, for example [19, 20].

Figure 2.3: Image that contains a printed pattern and can be used to calibrate the camera.

Our experiments showed that a search through the spaces of magnification and radial distortions is enough to find a transformation that warps one image in order to be similar to the other. In order to evaluate the search, the absolute sum of the difference of the images is valid. In order to decrease the number of evaluations, a heuristic can be used. This procedure has to be repeated for each pair conformed by the central image and another image of the focal stack.

Once found a set of parameters for the whole focal stack they can be used to remap or *warp* the images and always get a corrected focal-stack. So for our case, each camera needs to be calibrated at least once. This calibration can be automatic using our algorithm and could be executed directly in the smartphone for mobile applications.

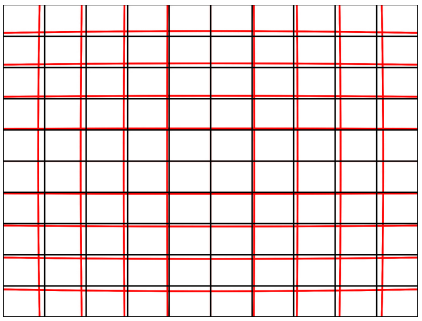
2.3.2 Image registration

There are, though, two other problems that are harder to solve. The first one is that the camera can be moving. This is the case for hand-held cameras, that are not mounted on a tripod and rely on the steadiness of the hands of the user in order to capture the focal stack. Notice that this problem is also present in standard cameras. In a general case, if the shutter speed is slower than about $1/100th$ of a second, motion blur will be noticeable in hand-held taken photographies.

For the case of focal stack, we need to take a number of images. If we are

(a) Template image, focused at a close object.

(b) Candidate image, focused at a far object.



(c) Best automatic distortion correction found by the algorithm. In red the corrected distortion over the black, straight lines.

(d) Correction applied to the candidate image.

Figure 2.4: The objects represented in the template image have different corresponding (x, y) in the candidate image. They have equal corresponding coordinates in the corrected candidate image.

able to take images at $1/500th$ of a second, the images will not present motion blur but the position of the camera can be too different between shots, since the total amount of time that the capture took was greater. For example, for $N = 5$ planes, the total capture time will be $1/500 * 5 = 1/100$ seconds. This time is enough to have noticeable camera motion that is unsolvable as noise by the algorithm. In a standard camera module as the sony imx377 [21], used for example in the Google's Nexus 6p smart-phone takes about 0.25 seconds on average in moving the lens reliably to a new position and capturing a new image. Considering this fact, the capture of a focal-stack will always be too long, and unless using a tripod, we need a solution.

Moving camera

If the difference of time between the capture of the first image of the focal stack and the last one is for example 1 second, the position of the camera can be very different. If the photographer tried to be still, it might be a solvable problem, by applying a registration algorithm on the images of the focal stack. This algorithm can be guided by the sensors of the smart-phone such as giroscopes and accelerometers. This approach has been used by Google in order to stabilize the recording of videos with great results and it is used widely for VR applications.

There are two kinds of registration methods: global methods and local methods. A global method is only able to correct linear transformations. This only works when the scene is located at a infinity distance. Real location changes of the camera produce local transformations, so a non-rigid transformation model must be used. Sticking to just correcting the global transformation of the images of the focal-stack could still make sense from a performance standpoint as correcting that will fix part of the problem.

The correction of local transformations is to take place after the correction of the optical distortion. [22] reviews the state of the art algorithms for image registration, at that time (1992). It also analyzes the principal components that define a registration algorithm. The most important concept is the *feature*. A feature defines information present in the images that can be used for *matching*. Features can be selected by their degree of suitability for being correctly identified on the candidate image. Matching is carried out by searching the feature, found in the template image, through the search space, in the candidate image. The search space can be simple, for example, a 2D translation, or more complex, as a perspective transformation. Lastly these

algorithms have a metric of similarity that allows for comparing how good a transformation is. An example of that is the sum of the absolute difference of the pixels of the transformed template feature and candidate feature.

Having understood how to make a good registration method that can cope with the normal shaking of a hand-held camera, there's another particularity of our problem: images will present different amounts of defocus on different 2D points, depending on their position within the focal-stack. Therefore an evaluation on how the template matching process is affected by blur is necessary. There's a modern registration algorithm described in [2] that is probably suitable to our endeavors. This algorithm was used to align a raw burst of fast, underexposed photos in order to reduce noise and produce a tone-mapped HDR image. The technique was used initially exclusively on Google Pixel smart-phones, but since then the Google's camera application has been ported by users to other smartphones. Adapting the algorithm to be able to align our focal stacks will remain as an open line of this thesis.

Moving objects

Another, even harder problem to solve is the presence of moving objects in the scene. In some video stabilization algorithms, the movement that moving objects produce is distinguished from the movement produced by the hand-held camera. Once this problem is solved, we need to decide what to do with these objects. They could be entirely suppressed or kept in a single location (the one of most focus), for the depth-map and all-in-focus images. If we do that, we also need to keep track of the occlusions that the movement produced. In the work [23], they refer at these kinds of focal-stacks as space-time focal volume and they take into consideration that objects are focused at a distance and can be moving.

2.4 Focus operators

The first part of the general SFF (shape-from-focus) algorithm consists on the obtainment of a measure of the amount of focus for each pixel ((x, y, z) in figure 2.1). These algorithms usually use a support local window in (x, y). Some methods as the one in [23] also take into consideration the z coordinate in order to track moving objects.

In [24] a broad selection of focus operators is evaluated. All focus opera-

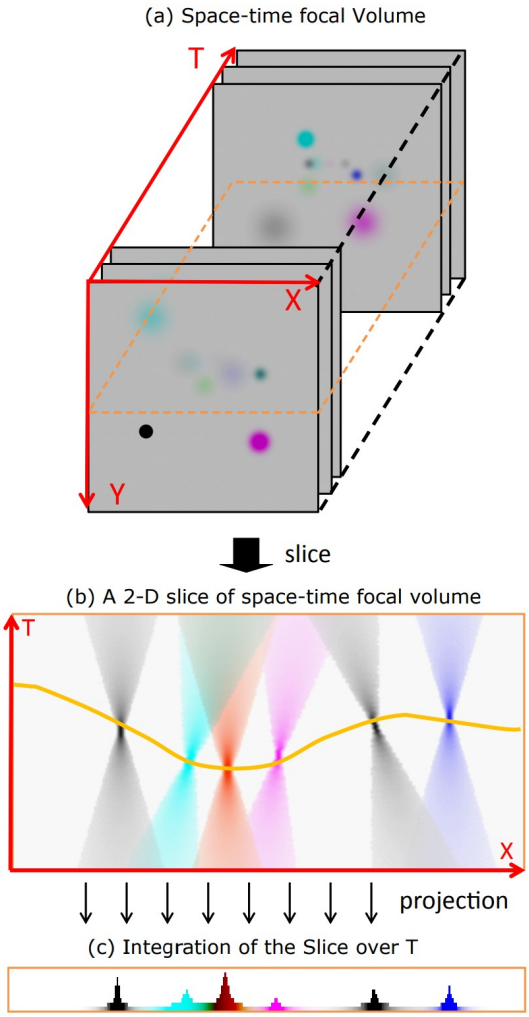


Figure 2.5: Figure taken from [23]. In (a), a representation of a focal stack with moving balls. In (b) the 2D slice show focusing cones with movement (cones are slanted). In (c), the maximum peaks indicate where the points of maximum focus are.

tors need at least to set a windows size parameter. Selecting the correct size is a problem. By using a smaller window, a finer precision is possible but at the same time the sensitivity to the noise increases and the problem of occlusion appears [25].

2.4.1 Laplacian operators

For this work we have selected a custom Laplacian kernel H (equation 2.2). It works well with images coming from a modern smartphone camera working with a capture resolution of FullHD (1920×1080). The proposed kernel remains small and can be expressed as a combination of two kernels that in turn are separable (H_0 and H_1, see 2.3, 2.4, 2.5). The absolute value of the result of the convoluted gray-scale image is then used for the subsequent steps.

$$H = \begin{bmatrix} 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{2.2}$$

$$H_0 = \begin{bmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \tag{2.3}$$

$$H_1 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \end{bmatrix} * \begin{bmatrix} -1 \\ 1 \end{bmatrix} \tag{2.4}$$

$$H = H_0 \circledast H_1 \tag{2.5}$$

2.4.2 Wavelets

Wavelets have been used as a tool for multi-resolution analysis for maybe 100 years, with its today widespread discrete biorthogonal version initiating around 1990. Since then they have shown their advantages in denoising, compression, signal analysis, deconvolution or contrast enhancement, to name a few. They have been also applied as a focus estimator recently [26, 27]. Indeed, several focus measurement operators based on Wavelets have been described and they outperform or match the performance of operators based on other local operators like gradient or Laplacian based [24].

(a) Original input image

(b) Absolute of the convolution on the gray-scale image (black is more intensity)

Figure 2.6: On the left: original image, where the subject is in focus, the beer bottle on the left is closer to the camera and defocused and the background is also defocused. Note that strong textures such as the ones present in the dress and hair make the filter respond with more intensity as well as sharp edges. In the background, bokeh also produces sharp edges.

Taking for example the method in [26], similarly to the Laplacian method exposed in 2.4.1, the image is high-pass filtered in order to find high frequencies, but this time, the summation of four bands is calculated in order to compose a more robust focus operator (see equation 2.6).

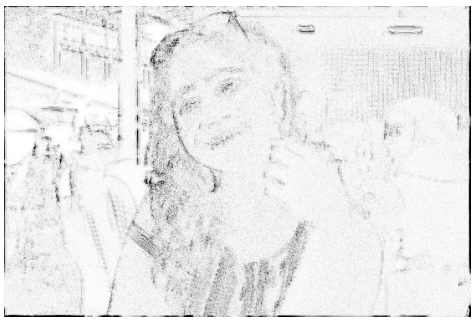
$$\phi = \sum_{(i,j)\in\omega_D} |W_{LH1}(i,j)| + |W_{HL1}(i,j)| + |W_{HH1}(i,j)| \tag{2.6}$$

While Wavelets applied to focus measurement provide good results, they are far from optimal since they decompose the image following just three isotropic directions: horizontal, vertical and diagonal.

The discrete Wavelet transform can be applied to digital images in order to decompose them in a multi-scale manner. Our primary effort is to explore the use of this and alternative transforms to decompose and represent an image in such a way that it can be separated in additive terms of a coarse model of the signal and another layer of texture and, probably, noise. See figure 2.8, where an example of how Wavelets can separate a synthetic 1D signal is depicted for two cases. In the top row there are not discontinuities, whereas in the bottom there is a discontinuity.

On each case the coarse structure is represented in red, while noise and

(a) Wavelet decomposition for two levels dauchebies-6.



(b) Absolute value of the recursive summation of the coefficients as stated in the equation 2.6 (black means more intensity).

Figure 2.7: Example of execution of the focus operator described in [26] that uses wavelets.

detail in are represented in orange. The left column contains the synthetic input, meanwhile the orange and red signals will be unknowns in the right column and are isolated analyzing the signal with filtering in Wavelet domain. In both depicted cases the separation of layers at both ends of the domain is slightly erroneous, but in presence of a discontinuity Wavelets fail to adequately detect the gap as part of the coarse signal, transferring energy to both sides of the edge.

2.4.3 Ridgelets and Curvelets

Wavelets allow to analyze an image in terms of sparse coefficients at multiple resolutions. This transform performs admirably well, –obtains the highest sparsity–, when applied to 2D images that can be described as a polynomial surface, indeed, the behavior will be optimal when the vanishing moments of the Wavelet surpasses the degree of the polynomial.

But the underlying model of natural images at lower frequencies is a rather unnatural one, as can be seen in the top row of figure 2.9. Wavelets adapt poorly to images containing shapes with sharp boundaries as they take a large number of coefficients to be described, which is not desired for the task at hand.

Ridgelets are the first step to overcome this undesired effect. They will

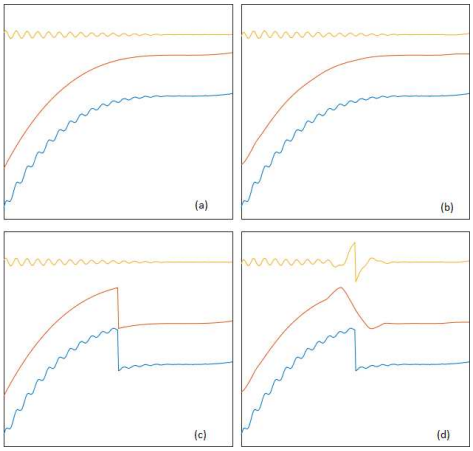


Figure 2.8: (a) A 1D signal expressed as the sum of a coarse continuous signal and a noise + texture layer; (b) the coarse and detail + noise layers isolated with filtering in Wavelet domain; (c) and (d) same process than (a) and (b) but with a piece-wise continuous signal as coarse layer.

perform better than Wavelets when describing images that contain linear radial structures, but then, Ridgelets will show their limitation when dealing with edges that are not only straight, but curved as well, or short lines that do not extend to the whole domain. This is why we will only use Ridgelet as a necessary step for arriving to Curvelets.

Curvelets will better explain images that have linear and curved radial components, because they will need less coefficients to describe natural images and hence they will be able to differentiate better the coarse geometry of the image from the texture and the noise, which is needed in order to measure defocus.

The idea after first generation Curvelets is to describe curved lines as superposition of several short straight lines on several scales. Therefore Ridgelet are not directly applied to the whole image, instead, the image is first decomposed in several subbands of frequency, then all subbands except the one with the lowest frequencies are decomposed in multiple overlapping blocks, and then each block is independently Ridgelet transformed.

The lower subband is not described in terms of lines. The medium and high frequency subbands, liberated from describing the coarse surface covered by the lowest band, has now a content more adequate to be described as sums of lines.

Figure 2.9: Effect of reducing, by hard thresholding, the energy carried by weaker coefficients in different domains, from top row to bottom: Wavelets, Ridgelets and Curvelets. From left column to right: threshold fixed to let pass the 99, 90, 70, 50 and 25 % of accumulated energy.

Figure 2.10 shows the medium and high frequency subbands of the example image. The filtering has been made with 2D Wavelets, but this is not mandatory. Additionally, the image of medium subband shown to the left, contains 8×8 blocks of size 64×64, and, to the right, the high frequency subband contains 16×16 blocks of 32×32 size. Each block overlaps completely with their 4-connected neighbors.

These blocks are the objects that will be independently passed through Ridgelet transform. In reconstruction, a windowed fusion will remove those overlapped areas whilst artifacts among blocks pass unnoticed, at the cost of increasing computations and the number of Curvelet coefficients.

The bottom row of figure 2.9 shows the partial reconstructions from Curvelet coefficients, and demonstrates that aDRT can be used as the foundation block of Curvelets. In spite of removing, by hard thresholding, some of the coefficients, the inverse still gives a reasonable result. As a proof of concept, an ill conditioned inverse would have succumbed to the lost of

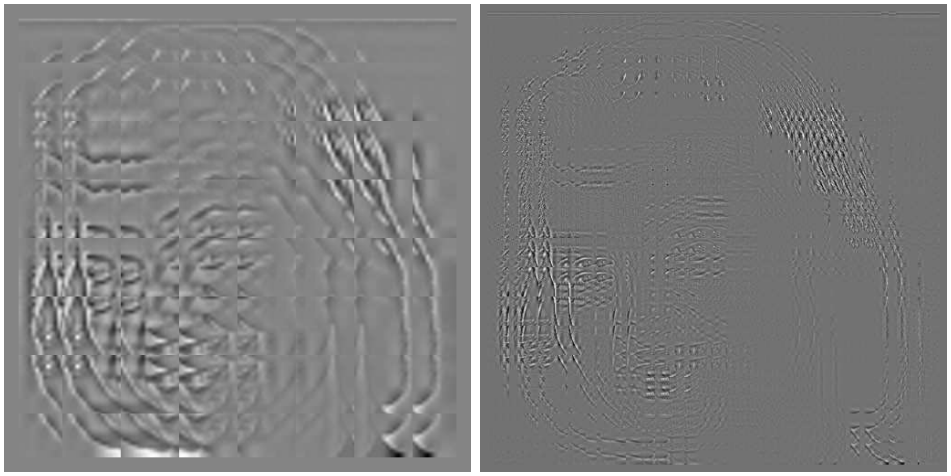


Figure 2.10: Subbands of middle and high frequency of the example image described by overlapping blocks where Ridgelet transform will be applied

symmetry due to the non-linear coefficient removal.

Figure 2.11: From left to right: detail of the eye in the image of example; original detail with simulated defocusing; inverse of Curvelets coefficients carrying 75% of the energy; inverse of Wavelets coefficients carrying 75% of the energy.

Moreover, partial Curvelets inversion adjust better to defocused version of the images. See figure 2.11 where a zone containing an eye of the image of example has been enlarged. From left to right it is shown the original eye, a simulated defocused version, and then the reconstruction from Curvelets and then Wavelets hard thresholded at 75% of the energy. Notice, for example, how much better the Curvelet version is, compared to Wavelets, when describing the crease around the lower eyelid. This partial reconstruction subtracted from different focused and defocused pictures of that eye, will allow to detect and measure focus locally with more precision than previous methods based on Wavelets.

2.5 Sparse DMAP calculation: Maximum search

Having calculated all the focus estimations for all pixels of all images of the focal-stack $\phi(x, y, z)$, the next step is to decide for every 2D coordinate what is the focal distance for which the corresponding 3D point belongs to.

$$sparse(x, y) = F(x, y, z), \forall z \in (0, N]$$

In order to decide what is the corresponding distance we take a slice of the processed focal stack that contains the result of the focus operator, being the *slice* of size $1 \times 1 \times N$) (see figure 2.12).

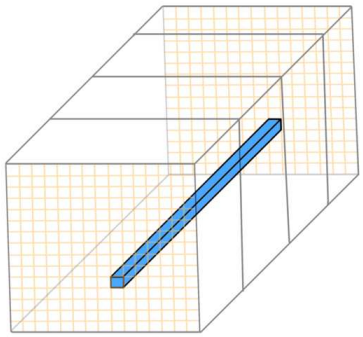


Figure 2.12: A slice is taken in order to determine its z distance.

If we analyze a slice, like the one en 2.12, we see that it resembles correctly the amount of focus, and it will suffice with finding the argument of the maximum in order to find its distance. This is specially true on hard edges, but it is not accurate where noise presence is strong or there is no texture. Therefore, many points will yield an erroneous depth estimation, as can be seen in figures 2.13, 2.14 and 2.15.

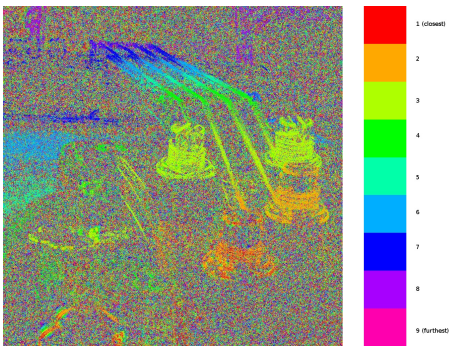


Figure 2.15: Computing the maximum without a support window. Only the points near the edges are correct.

This problem can be solved by using a support window. In order to do this a vicinity of pixels of the processed focal stack is averaged. If we apply a threshold to remove these values that did not achieve a minimum level of energy throughout the processed focal stack, it can be assumed that there will be no way of knowing if the energy belongs to a truthful measurement due to the presence of noise. Observe, in figures 2.16 and 2.17, that for big kernel sizes we have more "valid" information, and in the smaller kernel sizes we have less, but we do not lose precision.

From the observation of the results of varying the size of the support window, makes sense finding a way of maintaining the fine detail present at lower sizes while discarding errors as they appear at bigger kernel sizes. One way to do this is to discard those values that are different in, at least, depth-maps calculated from two different window sizes, see equation 2.7, where $S(x, y)$ is the refined sparse depth map at a given coordinate and $D_k(x, y)$ is the calculation of the non-refined depth-map using a support window of size k.

$$S(x,y) = \begin{cases} -1; & \bigcup_{k=0}^{K-2} D_i \neq D_{k+1} \\ D_0(x,y) & \end{cases} \tag{2.7}$$

Moreover, given that for both qualities –detail, and robustness against noise– the most restrictive D instances are always respectively D_0 and D_{K-1}, the calculation of $D_k, \forall k \neq 0 \cap k \neq K-1$ can be omitted, therefore resulting on the updated equation 2.8, and a much lighter computational effort.

$$s(x,y) = \begin{cases} -1; & d_0 \neq d_{k-1} \\ d_0(x,y) \end{cases} \tag{2.8}$$

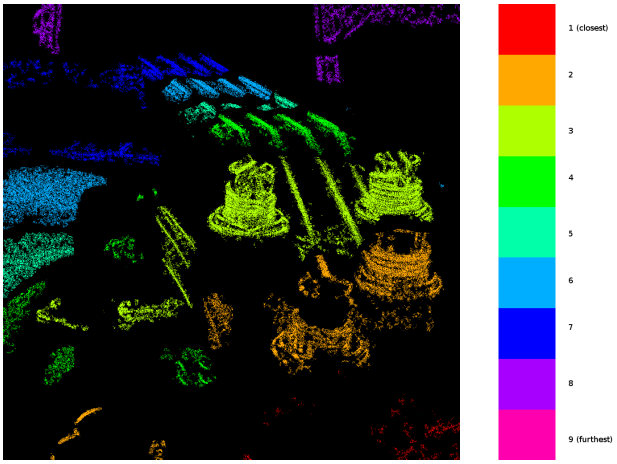


Figure 2.18: Result of the application of the formula in 2.7. Note that no false positives remain on the result and points match the exact location that produced the measurement.

In the examples shown in figures 2.16, 2.17 and 2.18, a box filter – an average – of the $k \times k : k \in 2 \times \mathbb{Z} + 1$ surrounding pixels was used. Other implementations are also possible, for example, a Gaussian filter can be used instead, of more complex approaches such as a guided filter [28], using as the guide, a color cue taken directly from the focal stack. Another possibility is also, using the color cue, apply a technique of vicinity based on *geodesic support weights* as this article [29] is applying to a depth-from-stereo algorithm. Segmentations such as super-pixel based are also possible.

In order to speed up computations, we can replace the Gaussian filter with a pyramid down-sampling such as the gaussian pyramid in [30]. A box filter can be applied instead of a gaussian filter and there is no need for going back to the full scale image.

Now that the method for obtaining a sparse depth-map has been described, we can test some focus operators. Let us look to the figure 2.19 for a visual interpretation. The quality of the method depends not only on the amount of true and false positives, but on how precise the measured point

Focus Operator	**Time (ms)**
Laplacian	436.67
Wavelets	360.32
Laplacian + Guided	715.71
Laplacian + Geodesic Distance	1742.36
Wavelets + Geodesic Distance	1671.36

Table 2.1: Measured times on an Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz processor for a focal-stack of size $N \times W \times H = 9 \times 1024 \times 1024$.

is as well. For example, wavelets seem to perform very well, but it is actually very imprecise compared to the Laplacian operator. A combination of wavelets and geodesic distance yield very good results. Since the wavelets have a limitation because of its locality we will build a focus operator based on curvelets. Again, the computational cost of curvelet, is a major handicap to its adoption as focus estimator, and we will center our efforts in its acceleration.

Computational times are shown in table 2.1. Note that it corresponds to a naive implementation made on python, but it is enough to give an idea of how complex each method is.

2.6 Dense depth-map computation

In the previous section, the mission was to accomplish a precise and as complete as possible sparse depth-map (*sDMAP*). For some locations it will be always impossible to recover depth information, due to many factors, among them: noise, under/over exposure, motion-blur, lack of textures, etc. In the computation of the dense depth-map (*DMAP*) we will assume that the *sDMAP* is precise, meaning that the non-null points in the map are only true positives.

The *DMAP* will be built using the *sDMAP* and the color focal-stack as inputs. The same problem is also present in shape from plenoptic or stereo algorithms as they often calculate a *sDMAP* first.

2.6.1 Optimization algorithms

Guessing the missing information can be formulated as an optimization problem. Given the ill-posed nature of the problem, it must be solved using a regularization technique. In order to solve an optimization problem, the constraints of the problem must be declared first.

In the problem at hand, the *sDMAP* must transformed into a *DMAP* by adding values of depth to the places where a null value was stored – those represented as black in the figures 2.17, 2.18 and 2.19–. As additional information we will use the color information in the focal stack. For the moment, a 2D version I of the focal stack will suffice:

$$I(x,y) = \sum_{i=0}^{N-1} F(x,y,i)$$

The premise in our problem is that if we have similar colors in I, they probably belong to the same plane and so they should have similar depth. By following this idea, a *cost function* can be defined as the *distance* between each two colors. Therefore, our cost function C can be defined as follows. Coordinate variables r and s are two-dimensional coordinates:

$$C = \sum_{r} \left(\text{sDMAP}(r) - \sum_{s \in N(r)} W(r,s)\text{sDMAP}(s) \right)^2$$

$N(x,y)$ denotes a set containing the *neighboring* pixels of r. $W(r,s)$ is the distance in color of the two pixels and can be defined as follows:

$$W(r,s) = (I(r) - I(s))^2$$

Other definitions can be used in order to take into account the local contrast such as [31]:

$$W(r,s) = e^{\frac{-(I(r)-I(s))^2}{2\sigma_r^2}}$$

and:

$$W(r,s) = 1 + \frac{1}{\sigma_r^2}(I(r) - \mu_r)(I(s) - \mu_r)$$

being σ_r and μ_r are the mean and variance of I in a window around r.

Using this equations we can build a linear equation system in order to feed a minimization algorithm, such as Gaussian elimination.

This problem has a very close resemblance to the colorization problem. In the work [31] a minimization on an inversion of an equation system takes place –using Matlab ® least-squares solver– in order to colorize an image sequence by providing the algorithm color hints as sketches. Using this approach we can trick this algorithm in order to get our problem solved. If we think of the *sDMAP* as color hints, we can ask the algorithm to "colorize" the image I and then remove the intensity component to get our result (see fig 2.20). An improvement of the method, also suggested in the same article, consists of using a multi-grid solver. It took 41.6 seconds on running the whole algorithm by using the least-squares solver and 20.0 seconds using the multi-grid solver.

While the results are good, it is not fast enough for our purposes. We need something that runs in less than two seconds.

2.6.2 Guided Filter

Another approach is to, iteratively, fill the gaps. In order to do that, we can imagine that each known point of the *sDMAP* acts like a source, from where a wave-front will travel in every direction (see 2.21). Also we need a way of restricting how much the wave-front can advance, again, using a metric based on color distance and variance on a window.

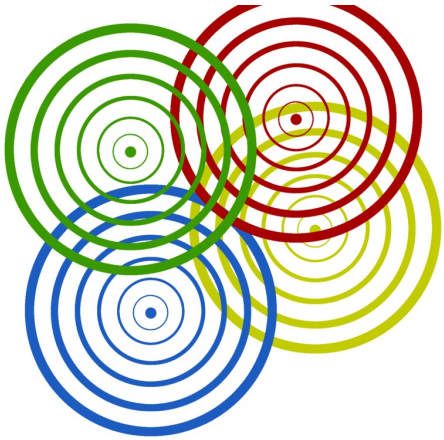


Figure 2.21: Each known point produces a wave-front that travels in every direction, being mixed with other points in the process.

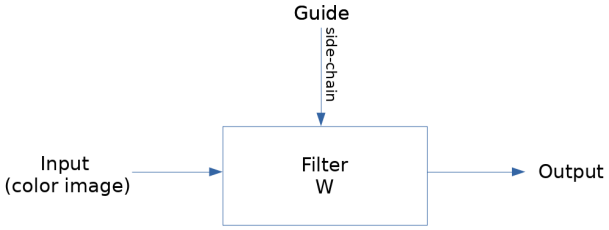


Figure 2.22: The weights are modified by the guide.

The first thing can be achieved by applying several times a low-pass filter such as a Gaussian filter or a box filter, or even a circular filter, whatever is cheaper on the architecture we are using. The second thing can be achieved by modifying the way the filter works, by adding additional weights on top of the "natural" weights the filter has. This can also be called *side-chaining* or *joint n-lateral filtering*, see figure 2.22 .

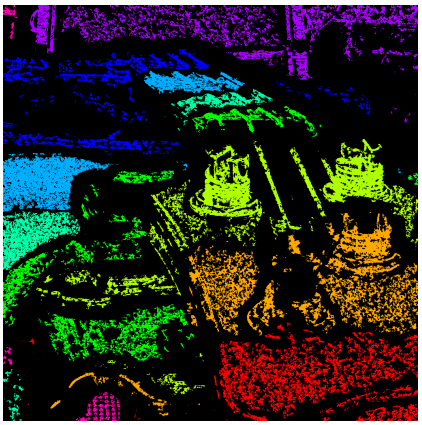
It turns out that we have already discussed a filter that does that. The *Guided filter* in its fast implementation [32] is ideal for the task. So with that tool, let us define the algorithm in 1. The $sDMAP$ input image must be built from the sparse depth-map obtained from the methods described in previous sections in the following manner. We will use the HSV colorspace in order to encode separately the information about the depth in H and the amount of certainty on the V. Implementation of the HSV colorspace conversions can be found on the book in [33]. The transformation consists on mapping each distance into the H, leaving the S and V at 255 when the sparse depth-map has a value and 0 otherwise. Assume $H \in [0, 360), S \in [0, 256), V \in [0, 256)$.

$$sDMAP(x,y) = \begin{cases} (0,0,0) & sDMAP(x,y) = Null \\ ((sDMAP(x,y) * 180/N) + 90, & \\ 255, 255) & sDMAP(x,y) \neq Null \end{cases} \tag{2.9}$$

Finally the sDMAP is converted back into the RGB color-space and the algorithm can start. The transform is similar to the one that has been used to show sparse depth-maps in the figure 2.19.

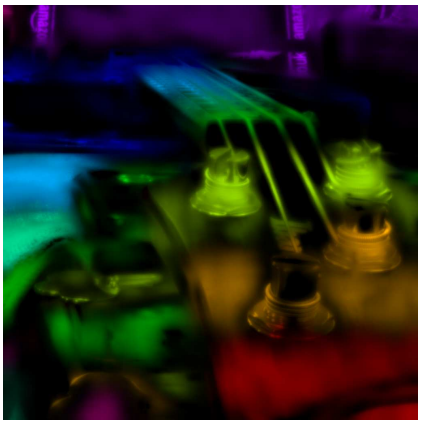
The important key that makes this algorithm work is that when a pixel that is colored "mixes" with an absolutely black pixel the three components – r, g and b – decrease proportionally, which mean, in HSV color-space, that

the H is exactly the same as the pixel that had value and is not modified by the black pixel – a pixel without a value –. So, depending on the type of pixel it will be mixed with, two behaviors can happen, a change of *Hue* and/or a change of intensity – *value* –. This effect can be better appreciated with an example, see figure 2.23.

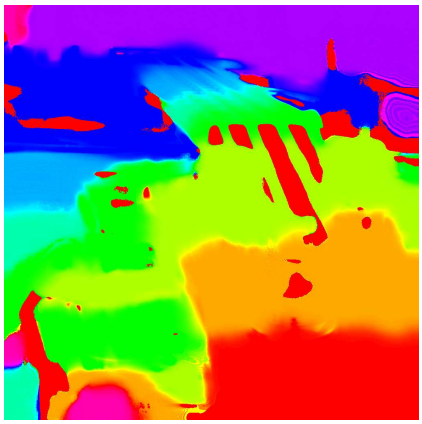


(a) Original depth map coded as hue (eq 2.9)

(b) Guide image.



(c) Result of the application of the guided filter.



(d) The result was converted to HSV color-space, set V and S to 255 and back to RGB color-space.

Figure 2.23: A filter of window-size 25×25 is applied to (a) using as guide (b). Note that in (c), while colors present variations in intensity and saturation, they have the hue that (a) provided. In (d) the only hues that do not come from the sparse depth-map are those red artifacts, which come from the undetermined values of the completely black pixels of (c).

Once the algorithm has been executed, the output must be converted back to HSV colorspace in order to extract the H component and remap it

Algorithm 1 Create a dense depth map using a guided filter.

Input: Image $sDMAP$ of size $W \times H$
Input: Image Set (the focal-stack) F of size $W \times H \times N$
Output: Image $DMAP$ of size $W \times H$

$th \leftarrow 102$ ▷ Upper threshold
$th2 \leftarrow 51$ ▷ Lower threshold
$ws \leftarrow 3$ ▷ Window Size
$si \leftarrow 0.8$ ▷ Sigma value
$finish \leftarrow False$
$DMAP \leftarrow sDMAP$
$img\_guide = \sum_{i=0}^{N} F_i$
while not finish **do**
    $sDMAP \leftarrow GuidedFilter_{ws,si}(sDMAP, img\_guide)$ ▷ Thresholding
    $cnt \leftarrow 0$
    for $i = 0$ to H **do**
        for $j = 0$ to W **do**
            if $DMAP(i,j) \neq (0,0,0)$ **then** ▷ If this value has not been already set
                $color = sDMAP(i,j)$
                $color_{h,s,v} = rgb2hsv(color)$ ▷ Transform to another colorspace
                if $color_v \geq th2$ **then**
                    if $color_v \geq th$ **then**
                        $DMAP(i,j) \leftarrow hsv2rgb(color_h, 255, 255)$
                    else
                        $DMAP(i,j) \leftarrow sDMAP(i,j)$
                    end if
                    $cnt \leftarrow cnt + 1$
                end if
            end if
        end for
    end for
    if $cnt = 0$ **then**
        $finish \leftarrow True$
    end if
    $ws \leftarrow ws + 2$
return $DMAP$
end while

to our depth-space, effectively reverting the behavior of the formula 2.9.

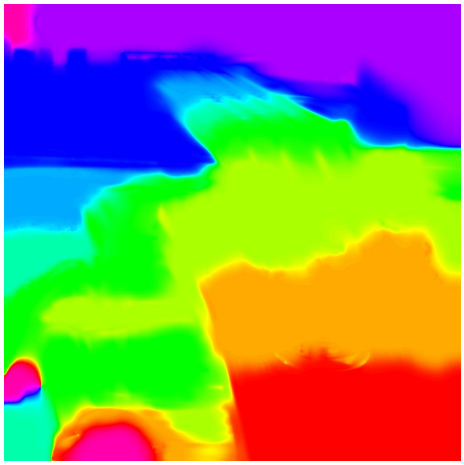
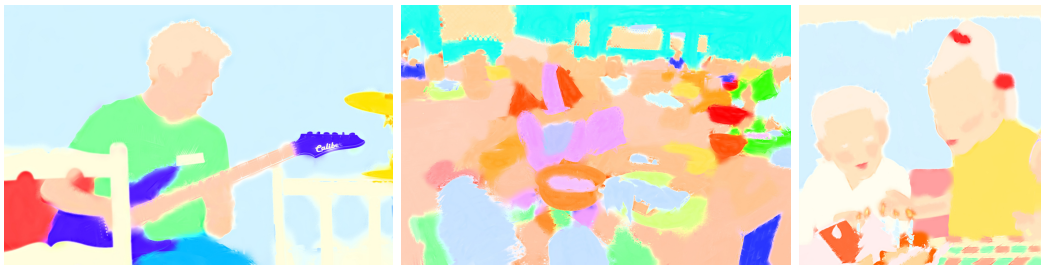


Figure 2.24: Result of executing the algorithm in 1

Since this result is very similar to the colorization algorithm of [31] we can re-purpose this faster algorithm for the same original task of coloring a black and white image or video. This time, the input of the algorithm is a gray-scale image with scribbles on it made by an user. These scribbles contain color and saturation information (h and s in hsv color-space). See an example in figure 2.25.

(a) Original gray-scale image.

(b) Annotations of color made by the user.



(c) Output of the algorithm (just the *hue* and *saturation*)

(d) *Hue* and *saturation* come from the output and the *intensity* values come from the gray-scale image.

Figure 2.25: Execution of the colorization using Guided Filter algorithm.

Since the algorithm increments the size with each iteration in order to fill larger areas and the finer details are set at the first iterations, we can

simulate the change of size of the filter by scaling down the images with each iteration. For this purpose we use a pyramid as we did on section 2.5. This modification results in a constant filter size, and a very low and fixed number of iterations, making it feasible for real time implementation, since the computational cost is about 1/5 of that of a guided filter. The guided filter has been already implemented in real time giving place to algorithms based on this technique such as [34], so our algorithm can have a real-time implementation as well.

2.6.3 GDT: Geodesic Distance Transform

In section 2.5, *geodesic support weights* where used in order to provide a support window with weights depending its "distance". This measure depended both on the euclidean distance and color distance. The underlying mathematical apparatus that was used for the *geodesic support weights* is the *Geodesic Distance Transform* (GDT). In the GDT, the distance is computed as the length of the shortest path from every pixel in the image to the set of "source" pixels, also called *reference set*. Although several implementations have been proposed in the literature, such as [35], [29] and [36], we will use the algorithm 2. For the costs image we can use the absolute gaussian gradient, among others and apart from the $BREAK$ variable we could set a fixed number of iterations that is enough for our resolution and purpose.

In order to build a depth map, an image in transformed domain will be calculated for each plane $p \in [0, N)$, setting to zero where the distance is equal to the plane in the $sDMAP$ and the rest to infinity on the initial A:

$$A_p^0(x,y) = \begin{cases} 0; & sDMAP(x,y) = p \\ \infty \end{cases} \tag{2.10}$$

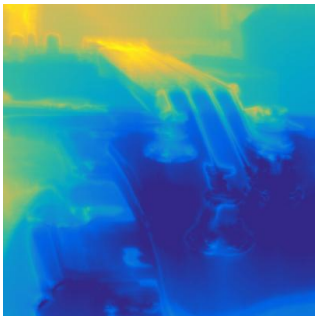
Once obtained the N transforms by calling to the algorithm 2, they will be mixed in order to compose the $DMAP$. The constant α is used to control how much the planes can mix.

$$DMAP(x,y) = \sum_{p=0}^{N} p \cdot \frac{1}{A_p^\alpha} \tag{2.11}$$

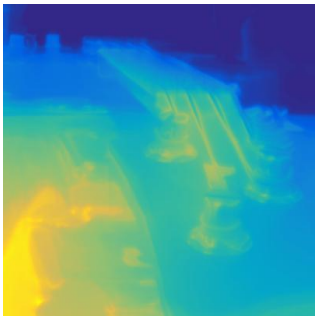
Algorithm 2 Compute the Geodesic Distance Transform.

Input: Image reference set A of size $W \times H$ of 16 bit integers
Input: Costs image C of size $W \times H$ of 16 bit integers

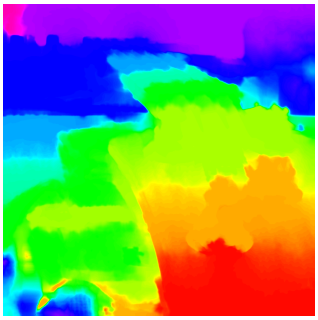
BREAK $\leftarrow$ 9000
finish $\leftarrow$ False
while not finish **do**
    $A \leftarrow$ rotate_90_degrees(A)
    $C \leftarrow$ rotate_90_degrees(C)
    max_diff $\leftarrow 0$
    for $i = 1$ to H **do**
        for $j = 1$ to W **do**
            $t_1 \leftarrow A_{i,j-1}$
            $t_2 \leftarrow A_{i-1,j}$
            if $|t1 - t2| > C_{i,j}$ **then**
                $t_0 \leftarrow min(t_1, t_2) + C_{i,j}$
            else
                $t_0 \leftarrow (t_1 + t_2 + (abs(2 * C_{i,j}) - abs(t1 - t2)))/2$
            end if
            max_diff $\leftarrow max($max_diff$, A_{i,j} - t_0)$
            $A_{i,j} \leftarrow min(A_{i,j}, t_0)$
            if (max_diff $<$ BREAK) *and* orientation_is_correct(A) **then**
                finish $\leftarrow$ True
            end if
        end for
    end for
end while
return A



(a) Algorithm 2 applied on a plane of closer focus.



(b) Algorithm 2 applied on a plane of farther focus.



(c) Result of mixing all the planes using equation 2.11

Figure 2.26: Example of execution of the depth map computation using *GDT*.

2.6.4 Algorithms comparison

The three algorithms to compute a dense depth-map have been executed on an Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz processor using a python – or Matlab ®, in the case of colorization – implementation that does not have any hardware optimizations. See table 2.2.

Algorithm	**Time (s)**
Colorization using Matlab's ® solver	41.6
Colorization using multi-grid solver	20.0
Dense *DMAP* using Guided Filter	3.8662
Dense *DMAP* using Guided Filter, pyramid-based implementation	0.698
Dense *DMAP* computation using Geodesic Distance Transform	1.0356

Table 2.2: Measured times on an Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz processor for input *sDMAP* and *guide* images of size $W \times H = 1024 \times 1024$. All times are calculated using an average of 30 executions.

2.6.5 Solving artificial edges

Given the nature of the *sDMAP* computation algorithm, those shapes that swipe a long set of continuous depth in the focal stack is going to be segmented into pieces, depending on the amount of texture and the actual focal

distances of the planes of the focal stack. In figure 2.24, the bass-guitar has been segmented into sections, where it should be a continuous shape.

A simple algorithm that can correct this effect is 3. See figure 2.27. This algorithm can be improved by using the guide and use an edge-preserving filter in order to decide if colors should be blended instead of just the color distance. And the size of the window needs to be adjusted depending on how far the planes are, and therefore how big they are.

Algorithm 3 Compute a "corrected" *sDMAP*.

Input: sDMAP of size $W \times H$ of 16 bit integers
Output: OUT corrected *sDMAP*
Parameter: wsize $\equiv$ Window size of the filter
Parameter: $\alpha \equiv$ Maximum distance in color

```
1:  for i ∈ [0, W) do
2:      for j ∈ [0, W) do
3:          color = 0
4:          if sDMAP(i, j) = −1 then
5:              OUT(i, j) = −1
6:              continue
7:          end if
8:          for k ∈ [max(i − wssize, 0), min(i + wssize), h) do
9:              for l ∈ [max(j − wssize, 0), min(j + wssize), w) do
10:                 if sDMAP(k, l) = −1 then
11:                     continue
12:                 end if
13:                 if |sDMAP(k, l) − sDMAP(i, j)| < α then
14:                     inc(count)
15:                     color = color + sDMAP(k, l)
16:                 end if
17:             end for
18:         end for
19:         OUT(i, j) = color/count
20:     end for
21: end for
```

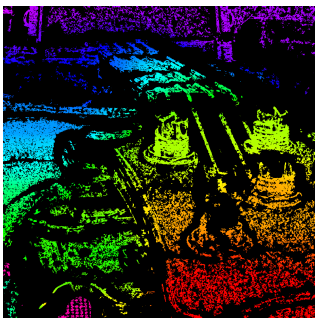


Figure 2.27: Execution of the algorithm 3.

2.7 All-In-Focus computation

In order to obtain an all-in-focus image from the focal-stack, we can use the $DMAP$ using any of the algorithm described on previous sections. Since our $DMAP$ gives place to non-integer values – $DMAP(x, y) \in \mathbb{R}$ – an interpolation is necessary and so we will use a linear interpolation. Let us call the nth image of the focal-stack at the location x, y $FS_n(x, y)$.

$$\begin{aligned} AIF(x,y) = \mathrm{FS}_{\lceil \mathrm{DMAP}(x,y) \rceil}(x,y) \cdot \{DMAP(x,y)\} + \\ FS_{\lfloor DMAP(x,y) \rfloor}(x,y) \cdot (1 - \{DMAP(x,y)\}) \end{aligned} \tag{2.12}$$

Figure 2.28: Computation of an all-in-focus using equation 2.12.

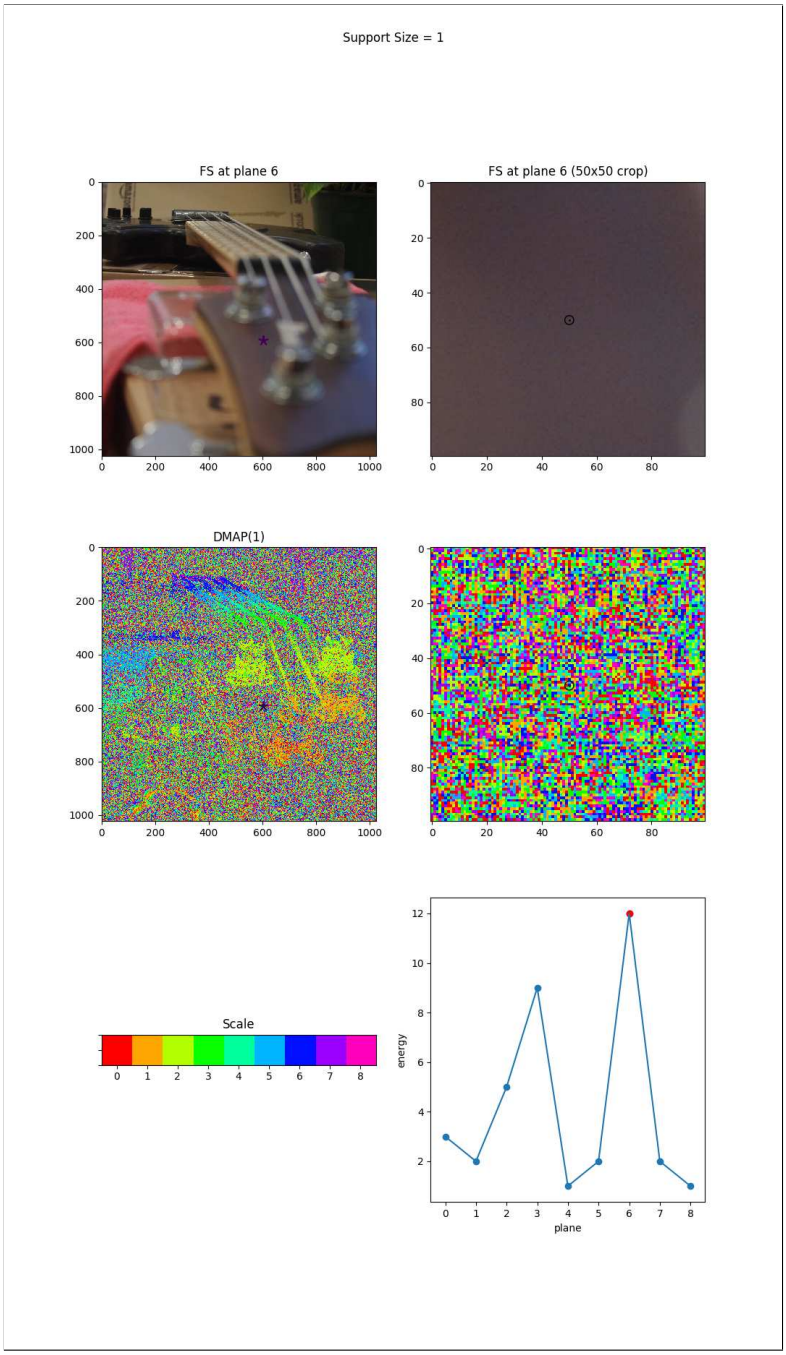


Figure 2.13: The measurements on a textureless zone are predominated by noise.

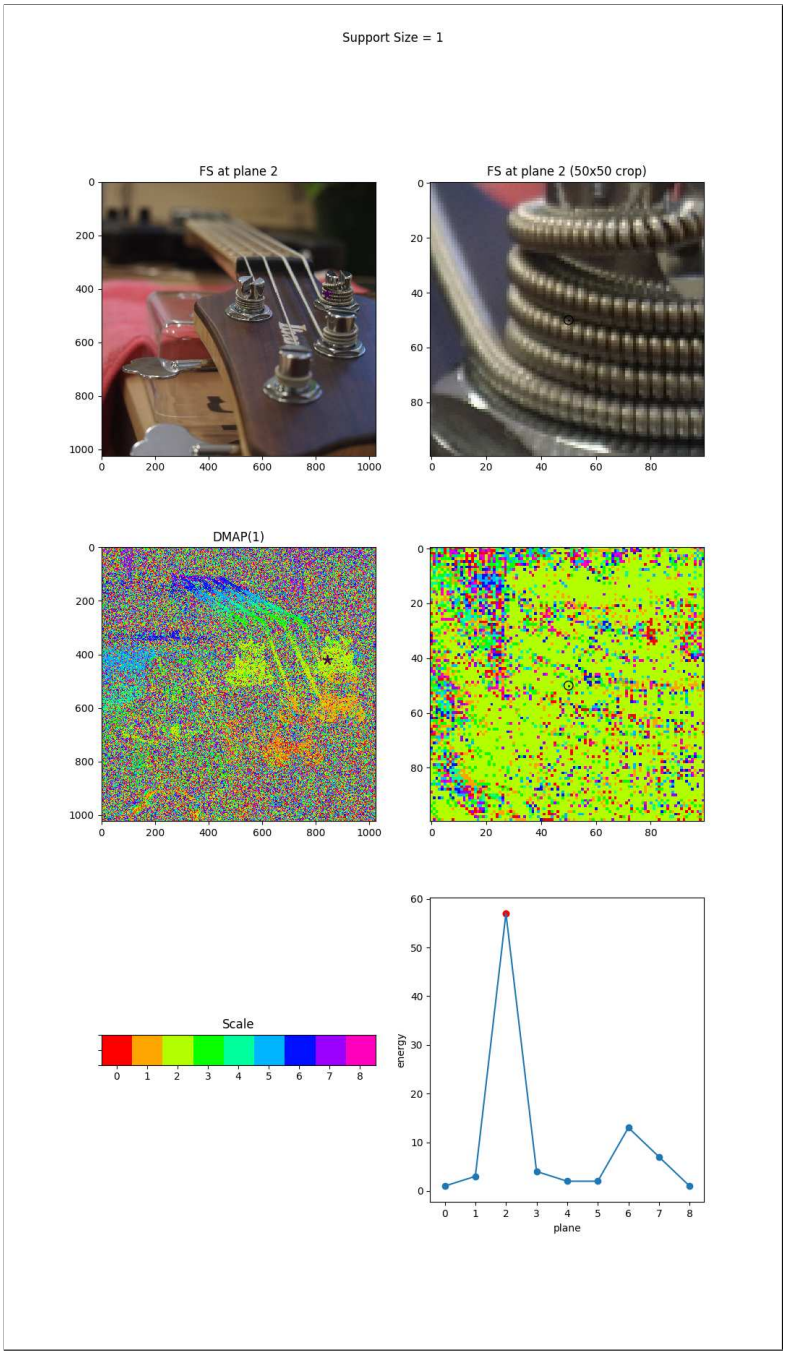
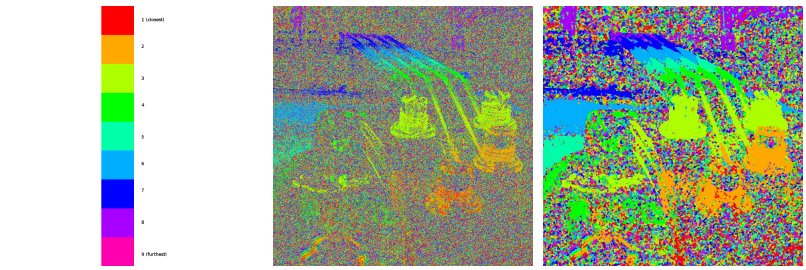
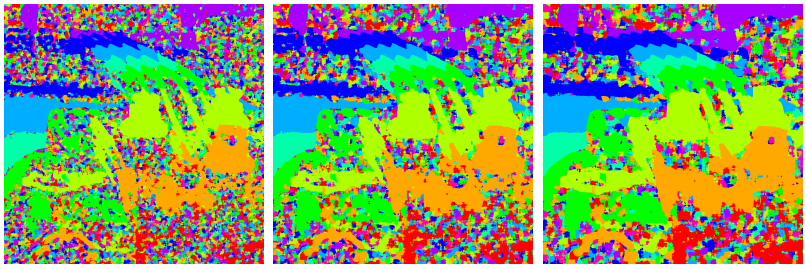


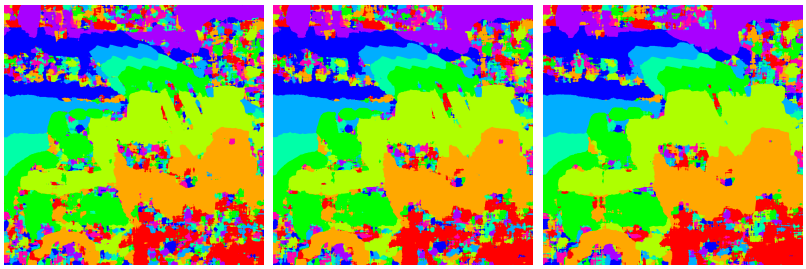
Figure 2.14: An edge location will give a good focus measurement. The value of the maximum is very different from the rest of measurements.



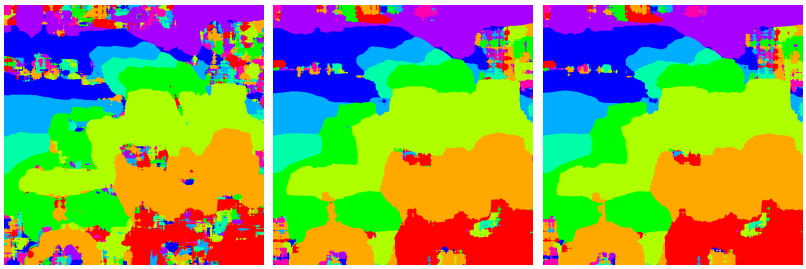
(a) Scale (b) $k = 1$ (c) $k = 1$



(d) $k = 13$ (e) $k = 19$ (f) $k = 25$

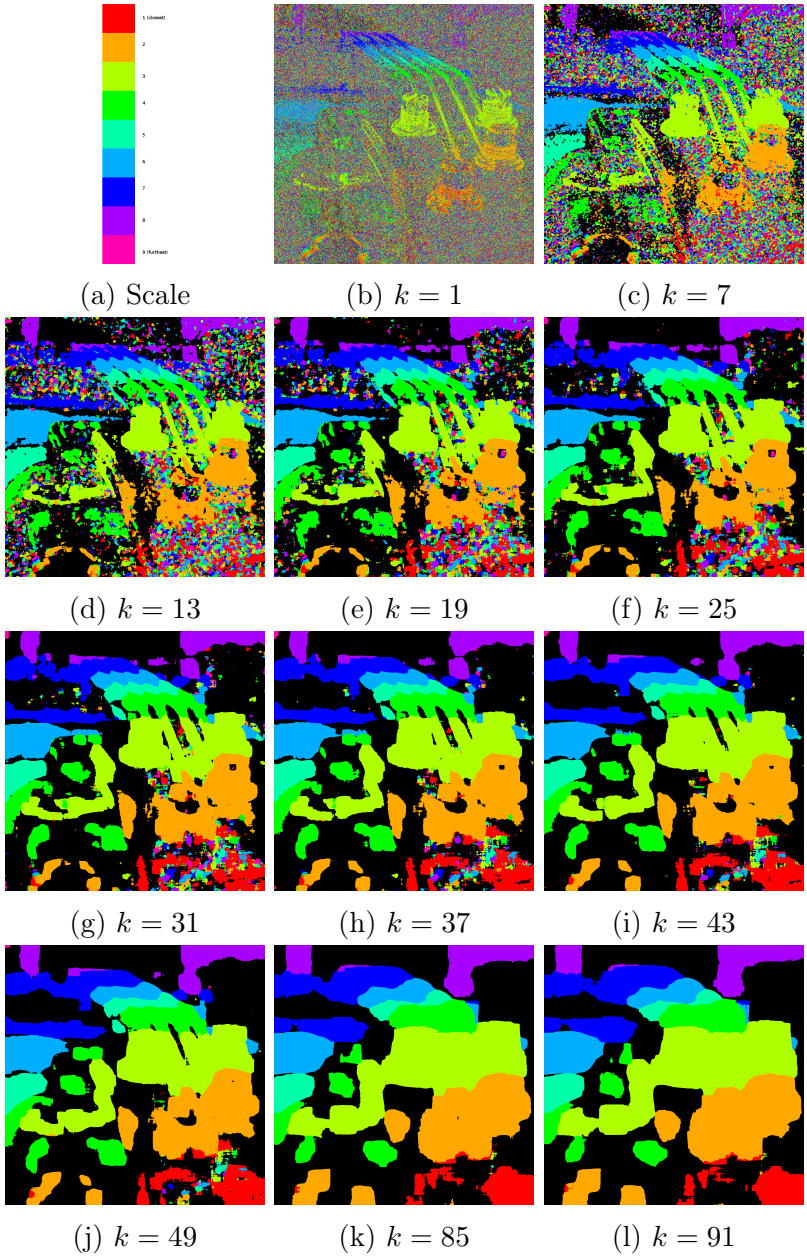


(g) $k = 31$ (h) $k = 37$ (i) $k = 43$



(j) $k = 49$ (k) $k = 85$ (l) $k = 91$

Figure 2.16: A depth-map is calculated using different support window sizes.



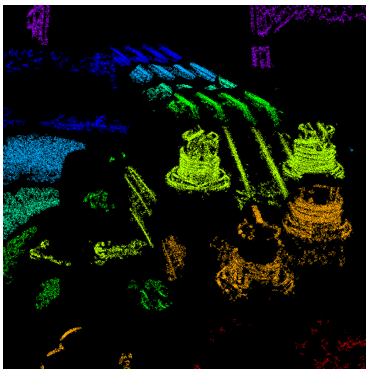
(a) Scale (b) $k = 1$ (c) $k = 7$

(d) $k = 13$ (e) $k = 19$ (f) $k = 25$

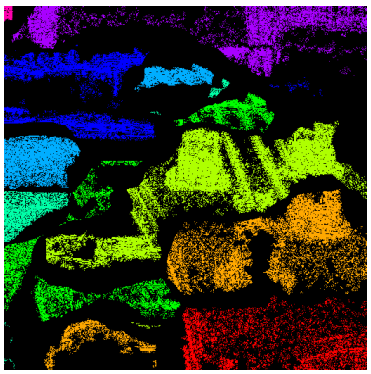
(g) $k = 31$ (h) $k = 37$ (i) $k = 43$

(j) $k = 49$ (k) $k = 85$ (l) $k = 91$

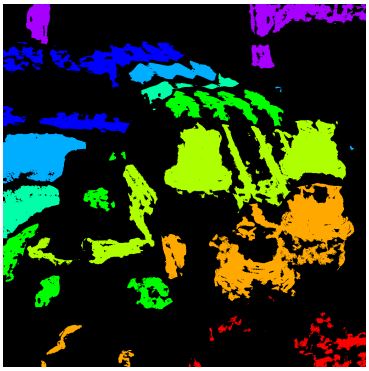
Figure 2.17: A threshold has been applied on the calculation of depth-maps. Note that bigger kernel sizes allow the noise to be neutralized at the cost of losing spatial resolution



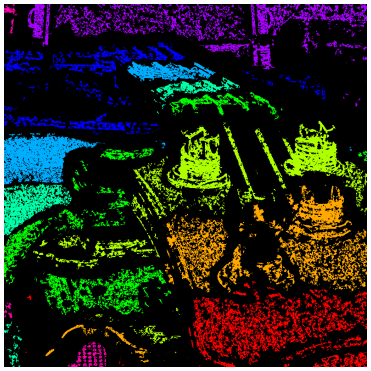
(a) Laplacian operator



(b) Wavelets operator (see equation 2.6)



(c) Guided filter + Laplacian operator.

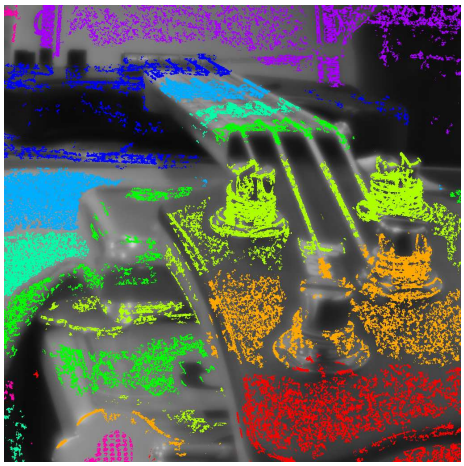


(d) Geodesic distance + Laplacian operator

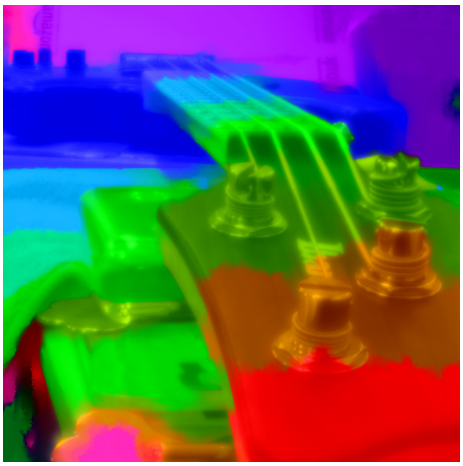
(e) Geodesic distance + Wavelets operator

Figure 2.19: Comparison of several focus operators after applying the algorithm in 2.8.

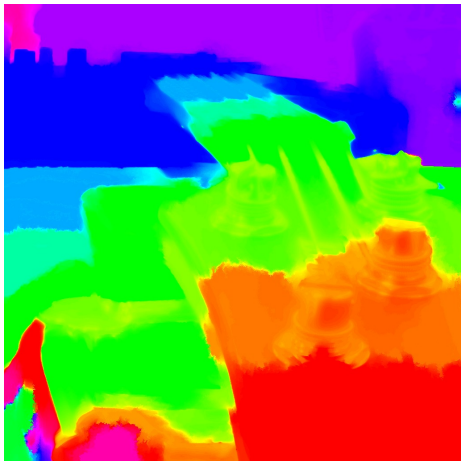
(a) Image I of intensities.



(b) Intensity image "scribbled" with the *sDMAP*.



(c) Colorized image.



(d) Only the color component, the depth-map.

Figure 2.20: An execution of the colorization algorithm with an adaptation that fits our endeavors.

Chapter 3

Discrete Radon Transform extensions for our Curvelet transform

3.1 Curvelets and DRT

The inadequacy of the Wavelets as a focus estimator was introduced in section 2.4.3. While Wavelets applied to focus measurement provide good results, they are far from optimal since they decompose the image following just three isotropic directions: horizontal, vertical and diagonal. Ridgelets [37, 38] are more suitable to our purposes since they decompose the image with a higher degree of directionality. Ridgelets detect very well linear radial structures, but they result insufficient to describe curved ones.

The Curvelet transform is an extension of the Ridgelet transform that overcomes this restriction[39]. It allows to decompose the image not in linear radial structures but in curved ones. So that they explain better "natural" images, in the sense that with fewer coefficients –with more sparsity– they can cope with the majority of the energy of a piece-wise continuous 2D signal. Other approaches explain even better natural images: Grouplets[40], Wedgelets[41], Contourlets[42]... but attending to the compromise between the expected quality and the computational effort involved, we chose the Curvelet transform as our tool to determine focused regions in an image.

There is a precedent in using a Curvelet transform as focus operator[43], in their implementation the authors make use of second generation Curvelets

and obtain very satisfying results at the cost of a heavy computational load. First generation Curvelet transforms[44] require to have pairs of Ridgelet, Wavelet and Radon transform computed at intermediate steps and at multiple scales. On the other hand second generation Curvelets[45] do not need to apply Radon transforms explicitly, but instead they rely heavily on FFT to accelerate computations.

In order to reduce computational requirements we will avoid the use of FFT as a constructing block, with the long term aim of porting our results to multi-threaded SIMD integer fixed point arithmetic architectures. This constraint discards second generation Curvelets completely and most implementations of the first generation ones. Avoiding the use of FFT is specially hard to achieve at Radon level, as most of available Radon fast algorithms rely on the Fourier slice theorem to reduce the otherwise heavy, $O(N^2)$, load.

With that in mind the number of choices reduce enormously.

Few fast Radon transforms refuse to be built on anything different than Fourier slice theorem[46]. One of those rare exceptions is the Finite Radon Transform[47] on which a Finite Ridgelet Transform [48] can be built upon, but again it fails when, in order to accelerate interpolation to non integer pixel positions it appeals to FFT. Similar reasoning exclude the Fast Slant Stack[49].

After having discarded those, we would have selected the Mojette transform but the only existing algorithm that computes its inverse transform that is simultaneously fast and well conditioned works in Fourier domain[50].

So we will propose a novel approach to calculate the Curvelet transform, in both its direct and inverse form: to use as underlying Radon algorithm the approximate Discrete Radon Transform, aDRT[51, 52]. This is now possible because it has been demonstrated recently [53] that there exists an algorithm to compute its inverse which is fast, well conditioned, and, in spite of being iterative, is also exact and does not rely on Fourier transforms.

This approximate Discrete Curvelet Transform, aDCT, will be constructed in the same fashion that first generation Curvelets, that is: with a Radon and a Wavelet transform we will define a Ridgelet transform, and that Ridgelet applied at multiple scales will become a Curvelet transform.

3.2 Conventional 2D discrete Radon transform

Discrete Radon transform, DRT, with a multi-scale approach, dates back to the late 1990s [51, 52, 54], and was originally designed to compute all of the integrals –sums, indeed– of pixels located on a discrete line, –actually a 1 pixel wide, N pixels long, stripe with approximately constant slope–, that touches at least one point on an image of size $N \times N$ whilst projecting in a semi circumference around it. There exists an algorithm that solves with linearithmic complexity, $O(N^2 \log(N))$, this problem for a quadrant covering from 0 to 45 degrees. And by joining together 4 runs of the algorithm on 4 mirrored versions of the input a sort of Möbius band is obtained that comprise the whole set of line integrals that arise when projecting on 180 degrees around the image: a discrete version of the *sinogram* in the continuum devised by Johann Radon[55, 56].

Suppose an image of size $N \times N$, N results are obtained by adding together the values in the same rows. And other N results are obtained by summing values lying in the same columns. These two sets of summations correspond with projections at 0 and 90 degrees respectively. Each summation adds N values and each set of summations comprise N sums corresponding to different intercepts in the projection axis. 2 N^2 sums have been computed in order to obtain $2N$ *line integrals*. With a little more computational effort the multi-scale DRT algorithm provides N sets of line integrals to different angles between 0 and 45 degrees, and for each angle, its whole set of intercepts.

Taking into account all the discrete lines that touch an image for a given projection angle is a non-trivial task. In order not to discard any line, even if it touches the image in just a single pixel, more intercepts must be considered for angles close to odd multiples of 45°, and less for those close to 0°, 90° and so forth. This explains why conventional DRT can have between N and $2N - 1$ number of intercepts depending on the angle being calculated. Conventional DRT computes them all, and as a result, the total output size for an $N \times N$ input occupies $3N \times 4N$, where only half of the results sum other than zero.

In this work two alternatives are presented. They generate an output of size N along the intercept dimension independently of the angle being considered, that is, the output size for a single quadrant will be the same than the input. This will alleviate the computations and the memory footprint of DRT almost by half.

Considering the figure 3.1, the first three sub-figures from left to right

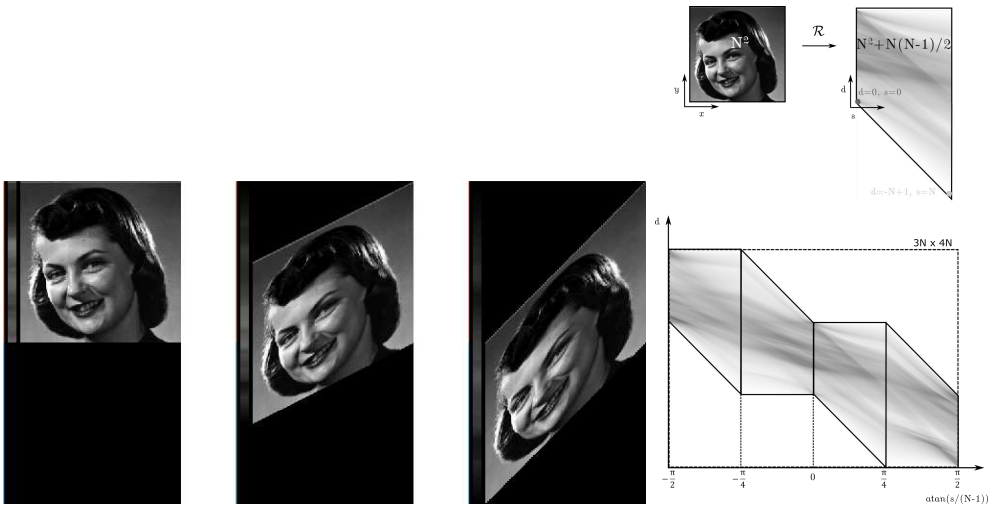


Figure 3.1: First three sub-figures: from left to right an image sheared to accommodate in a row the discrete lines of conventional DRT for minimal, intermediate and maximal slope of the 0 to 45 degrees quadrant. Rightmost, a depiction of input and output of conventional DRT for a single quadrant. And shape of output of 4 quadrants, joined together within a $3N \times 4N$ rectangular memory footprint where half of the space is wasted.

show the same image but sheared so that when summed along rows they give as result the conventional DRT for slopes 0, $\frac{1}{2}$ and 1. There is a column at the left of each sub-figure that is precisely the DRT at each intercept for that slope. It can be appreciated that the number of intercepts to be calculated can be as much as twice the vertical size of the input in order to accommodate the most extreme intercepts when slope is maximal. This is what gives rise to the particular shape of the conventional DRT output as shown to the rightmost in figure 3.1.

The algorithm itself does not shear the images. Instead it can be thought that a plurality of discrete lines will be traversed summing the pixels that lie on them. For the 0 to 45 degrees quadrant, the relation between the pixels belonging to the same discrete line is formulated based on the ascents on the vertical dimension while the horizontal dimension is traversed. A discrete line will visit exactly one pixel on each column, and no interpolation will be required. In this basic quadrant the lines to be considered are the discrete versions of lines of the form $y = s \cdot x + d$, where s denotes the *s*lope or ascent; and d denotes the *d*isplacement or intercept. The variety of

names is due to the adoption of algebraic slope-intercept formulation of lines and simultaneously the adherence to previous authors' notation. Actually, in the context of discrete lines, it is more precise to denominate ascent the parameter normally denoted as s. Anyway, from that s parameter the slope could be calculated as $\frac{s}{N-1}$ and the angle of projection relative to the positive direction of x-axis as $\tan^{-1}(\frac{s}{N-1})$. From now on the parameters on Radon domain will be denoted as s and d, and be called slope and displacement.

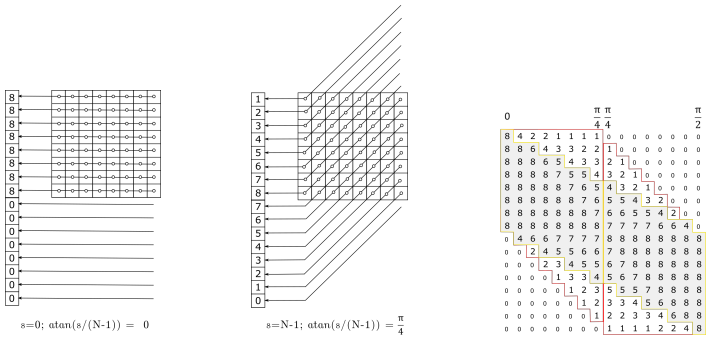


Figure 3.2: Number of pixels to be summed together at different displacements when calculating DRT for $s = 0$ (left) and $s = N - 1$ (middle). The number of pixels evaluated for the rest of slopes and displacements of two quadrants of a 8×8 image are shown on the right.

In the continuum the 2D Radon transform allows to describe a 2D signal, $f(\cdot)$, in terms of the integrals along lines parameterized by an angle and a displacement, (θ, ρ), instead of single values accessed by their horizontal and vertical pair of cartesian coordinates, (x, y). This is,

$$\mathfrak{R}f(\theta, \rho) = \iint f(x, y)\ \delta(x\ cos\ \theta + y\ sin\ \theta - \rho)\ \mathrm{d}x\ \mathrm{d}y, \tag{3.1}$$

or, equivalently, using the absolute $s$lope and $d$isplacement form, with $|s| < 1$,

$$\begin{aligned} \mathfrak{R}_{|\theta| \leq \frac{\pi}{2}} f(s, d) &= \int f(u,\ u\ s + d)\ \mathrm{d}u, \\ \mathfrak{R}_{|\theta| \geq \frac{\pi}{2}} f(s, d) &= \int f(u\ s + d,\ u)\ \mathrm{d}u. \end{aligned} \tag{3.2}$$

The calculation of the Radon transform on a computer has to deal with the discretization of data. Basically, for regularly spaced discrete data, they

will be available only for a finite number of samples normally accessed with integer indexes. And the problem arises when the continuous definition of line integral makes necessary to evaluate the function at positions where there is no sample available and interpolation is required.

However using the pseudo-polar Fourier transform[57] –a variation of FFT that operates on a grid of concentric squares–, a discrete Radon transform can be designed that is algebraically exact, invertible, fast[58], and can be generalized to 3D[49, 59]. But it is based on Fourier transforms which is something that we want to avoid. Moreover, if, as it is the case in our application domains, the problem to solve is purely discrete the multi-scale Radon transform performs better than any other discrete Radon transform[46], including those based on pseudo-polar FFT.

3.2.1 Forward multi-scale discrete Radon transform

Götz & Druckmüller [51], Brady [52] and Brandt & Dym [54], almost simultaneously, proposed a divide and conquer approach reminiscent to FFT, in the sense that it works solving the problem at smaller scales and then combines those solutions to solve at greater scales, but with no multiplications nor complex *twiddle* factors involved, relying exclusively on integer arithmetics to achieve its goal. By working at multiple scales, and due to the symmetry of the problem, intermediate computations can be reused preventing any sum to be computed twice and so reducing the computational load from $\mathrm{O}(N^3)$ to $\mathrm{O}(N^2 \log N)$.

To make this possible, the key is to define a loose discrete line that traverses the domain visiting only integer positions, and therefore not exactly in a straight way. Ascensions are defined recursively, making lines composed of two halves, which in turn come each from other two line segments of half their size and so on until line segments that join only two points are reached, and the problem can not be further reduced.

DRT authors eluded to establish any sort of trigonometric relation between x and y variable, instead they decomposed u and s variables of equation 3.2 into binary indexes and mixed them at binary level, one index of u and one index of s at a time, this way avoiding direct multiplication of u by a slope which would have produced non-integer indexes.

On a previous work[60], the formulation of DRT was redefined so that the extension to more dimensions became feasible. This work adheres to that notation. The details on the formulation of the discrete Radon transform

can be found there. The resulting discrete Radon transform, DRT, or more specifically, the definition of discrete lines, eq. (3.3); the definition of partial transform until stage m, eq. (3.4); and the mapping between two stages, eq. (3.5), are:

$$l_s^n(u_0,\ldots,u_{n-1}) = l_{\lfloor s/2\rfloor}^{n-1}(u_0,\ldots,u_{n-2}) + u_{n-1}\left\lfloor\frac{s+1}{2}\right\rfloor = \sum_{i=0}^{n-1} u_{n-1-k}\cdot\left\lfloor\frac{\frac{s}{2^i}+1}{2}\right\rfloor \quad (3.3)$$

$$\widetilde{f}^{\,m}(\,\overbrace{s_{n-m},\overbrace{s_{n-m+1},\ldots,s_{n-1}}^{\boldsymbol{\sigma}}}^{\boldsymbol{s}}\,|\,\overbrace{v_m,\ldots,v_{n-1}}^{\mathbf{v}}\,|\,d) = \sum_{\mathbf{u}\in\mathbb{Z}_2^m} f(\lambda(\mathbf{u},\mathbf{v})|l_{\lambda(\boldsymbol{s})}^m(\mathbf{u})+d) \quad (3.4)$$

$$\widetilde{f}^{\,m+1}(\underbrace{\overbrace{s_{n-m-1},\overbrace{s_{n-m},\ldots,s_{n-1}}^{\boldsymbol{\sigma}:\,m\ bits}}^{\boldsymbol{s}:\,m+1\ bits}\,|\,\overbrace{v_{m+1},\ldots,v_{n-1}}^{\mathbf{v}:\,n-m-1\ bits}}_{n\ bits}\,|\,d) = \widetilde{f}^{\,m}(\boldsymbol{\sigma}|\,0,\mathbf{v}|\,d) + \widetilde{f}^{\,m}(\boldsymbol{\sigma}|\,1,\mathbf{v}|\,d+s_{n-m-1}+\lambda(\boldsymbol{\sigma})) \quad (3.5)$$

with $\lambda(u_0,\ldots,u_{n-1}) = \sum_{i=0}^{n-1} u_i\cdot 2^i$, i.e., the function that converts from binary multidimensional indexes, to decimal unidimensional index. Notice that single comma (,) is used to separate binary indexes, and vertical bar (|) is used to separate different parameters.

Notice also that the number of bits in partial stages is varying and depends on m, the current stage. When $m=0$, the array $\widetilde{f}^{\,0}(s|v|d)$ is really bidimensional, as variable s is still empty, so $\widetilde{f}^{\,0}(-|v|d)$ maps directly to $f(x|y)$. And when $m=n$, the last stage, variable v will be emptied, and so $\widetilde{f}^{\,n}(s|-|d)$ is the desired result $\mathfrak{R}f(s|d)$. That can also be appreciated by evaluating the definition of partial transform, eq. 3.4, for stage $m=n$: $\mathfrak{R}f(s|-|d) = \widetilde{f}^{\,n}(\,s_0,\ldots,s_{n-1}\,|-|\,d) = \sum_{u=0}^{N-1} f(\,u\,|\,l_{\lambda(\boldsymbol{s})}^n(u)+d)$. This last

equation is no other than the discrete version of Radon transform as expressed in eq. (3.2) with multiplication with the slope substituted by discrete line interaction between u and s at binary level.

Discrete lines in conventional DRT

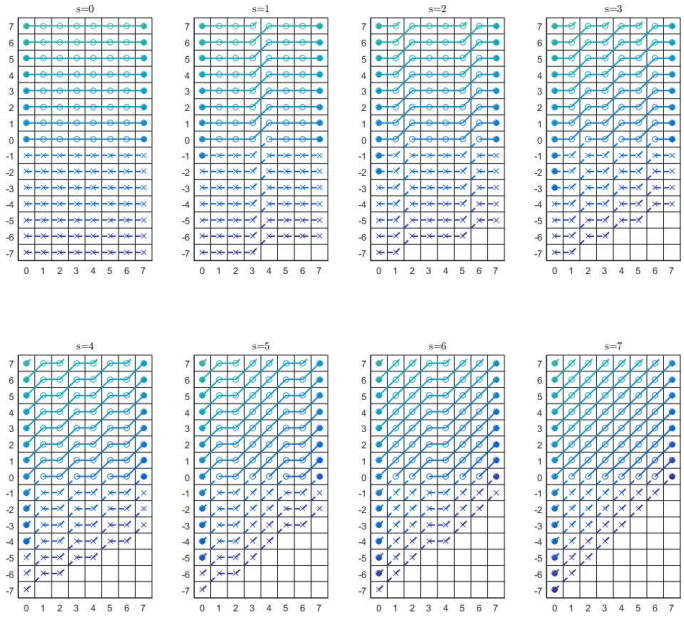


Figure 3.3: A depiction of the whole set of discrete lines covering the displacements and slopes as computed by conventional DRT.

As said when analyzing figure 3.1 it is not the image itself that is sheared, but the interns of the algorithm that traverse it adding pixels lying on a plurality of lines. The lines that are traversed for a 8×8 image –with another 8×8 region filled with zeros padded below– are depicted in figure 3.3. Each rectangle depicts the $2N - 1$ lines starting at different displacements for a certain slope. There are 8 rectangles corresponding to the N slopes, $\frac{s}{7}$, with s varying from 0 to 7.

Notice that each line joins a point with coordinates $\{x = 0, y = d\}$ with the point at $\{N - 1, d + s\}$. The inner points of those lines having integer coordinates $\{u,\, l_s^3(\boldsymbol{u}) + d\}$, that is $\{\lambda(u_0, u_1, u_2),\, l^3_{\lambda(s_0,s_1,s_2)}(u_0, u_1, u_2)\} = \{\lambda(u_0, u_1, u_2),\, u_0 \cdot s_2 + u_1 \cdot (s_2 + s_1) + u_2 \cdot (2 \cdot s_2 + s_1 + s_0) + d\}$. The crosses depict values previously filled with zeroes. The circles are the input data.

When there is a cross on the left, column 0, it means that the discrete line with that slope and displacement will never touch the image; a circle on the left, even if with negative displacement, meaning that it will sum different than zero (for a non null image).

The algorithm exhibits linearithmic complexity because any two-points line segment, for example the one that joins the points $\{2, 0\}$ and $\{3, 0\}$, will be computed only once and then reused when needed. In this case when computing results with parameters $\{s = 0, d = 0\}$, $\{0, 1\}$, $\{2, -1\}$ and $\{3, -1\}$. This holds true for every segment computed on any scale.

The zeroes padded to the input at the first stage make the input double in size, and therefore doubling the number of operations at each stage, just to accomplish the completeness in slopes and intercepts, even if a lot of partial sums on this region will never be used: for example the sums of pixel at $\{6, -1\}$ with $\{7, -1\}$ or pixel at $\{4, -4\}$ with $\{5, -3\}$.

Full sinogram construction

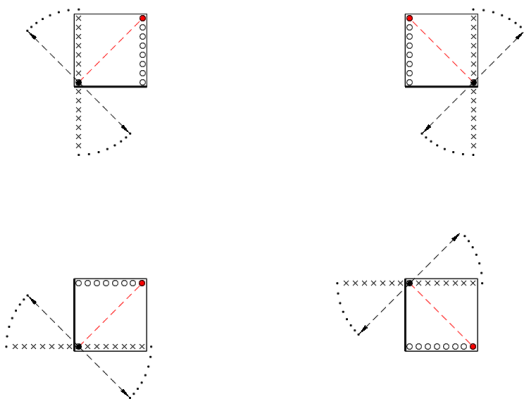


Figure 3.4: Schemes of behavior of the 4 quadrants of a full DRT. From left to right, and top to bottom: $0° : 45°$ quadrant; $135° : 180° \equiv 0° : -45°$ quadrant; $90° : 45°$ quadrant and $-90° : -45° \equiv 90° : 135°$ quadrant.

Equations (3.3), (3.4) and (3.5) constitute the core of the DRT algorithm for the basic quadrant with angles between 0 and 45 degrees. A discrete version of the full sinogram can be accomplished, reusing the algorithm that solves a quadrant, by applying it another thrice to versions of the input where axis are swapped and/or flipped conveniently and their partial outputs

merged in a four times bigger global result. A typical output was shown to the rightmost on fig. 3.1.

Figure 3.4 depicts the behavior of the basic quadrant algorithm in the top left sub-figure. The other sub-figures explain how the rest of angles could be computed.

The schemes on the sub-figures follow these conventions:

- the black circle will be considered our $\{0, 0\}$ reference point.
- the reference point will be joined by discrete lines with the circles on the opposite extreme, so those circles depict the plurality of slopes to be considered.
- the reference and the slopes are at the extremes of the axis marked with a coarse black line. This axis is coincident with the discrete line with null displacement and slope.
- for the maximal slope, the reference point will be joined with the red circle. This line of maximal slope is one of the diagonals of the plane and has as normal the vectors depicted with dashed arrows. Those can be interpreted as positives or negatives.
- the rest of discrete lines, those corresponding to the ascents from 1 to $N - 2$ have as finishing vertex of their normals the black dots that describe arches close to the axis where crosses are depicted.
- those crosses represent the displacements that must be considered.

3.2.2 Radon ad-joint transform

To achieve an inverse Radon transform it is necessary to define previously the ad-joint transform.

It can be seen in the next terms: the forward transform computes, from an input image, the summation of pixel values through a set of lines, which is complete in slopes and displacements.

The ad-joint transform is such that for every point in the Radon domain, corresponding to a certain slope and displacement, it redistributes uniformly that summation value back to the image domain assigning the same quantity

Algorithm 4 Compute the DRT of a quadrant

Input: Image f(x, y) consisting of $N \times N$ data
Output: Radon transform of f, $\Re f$(s, d) consisting of $N \times 2N$ data

```
1: n ← log2(N)
2: f^m ← zeros(N, 2N)
3: f^{m+1} ← zeros(N, 2N)
4: f^m(:, N : 2N) ← f                                   ▷ Fill the upper part of the output
5: for m = 0 to n − 1 do
6:     for d = 0 to 2 · N − 1 do
7:         for v = 0 to (1 << (n − m − 1)) − 1 do
8:             for σ = 0 to (1 << m) − 1 do             ▷ Outside mem accesses return zero.
9:                 f_0 ← f^m(σ + (v << (m + 1)), d)
10:                f^{m+1}((σ << 1) + (v << (m + 1)), d) = f_0 + ...        ▷ Case s_{n−m−1} = 0
11:                    f^m(σ + (1 << m) + (v << (m + 1)), d + sigma)
12:                f^{m+1}(1 + (σ << 1) + (v << (m + 1)), d) = f_0 + ...    ▷ Case s_{n−m−1} = 1
13:                    f^m(σ + (1 << m) + (v << (m + 1)), d + 1 + sigma)
14:            end for
15:        end for
16:    end for
17:    f^m ← f^{m+1}
18:    f^{m+1} ← zeros(N, 2N)
19: end for
20: ℜf ← f^m
21: return ℜf
```

to every pixel traversed by the line. That obviously is not the inverse transform, but –and that is the key of the contribution from William H. Press[53]– it is a sufficiently good start point to induce an iterative refinement process.

The ad-joint discrete Radon transform, is defined by the inverse mapping of equation (3.5):

$$\begin{aligned} \tilde{f}^{\,m}(\boldsymbol{\sigma}|\,0,\mathbf{v}|\,d) &+= \tilde{f}^{\,m+1}(\,s_{n-m-1},\boldsymbol{\sigma}\,|\,\mathbf{v}|\,d) \\ \tilde{f}^{\,m}(\boldsymbol{\sigma}|\,1,\mathbf{v}|\,d+s_{n-m-1}+\lambda(\boldsymbol{\sigma})) &+= \tilde{f}^{\,m+1}(\,s_{n-m-1},\boldsymbol{\sigma}\,|\,\mathbf{v}|\,d) \end{aligned} \tag{3.6}$$

Figure 3.5 shows the ad-joint operator working independently on the four quadrants of a forward DRT, as well as the sum of all of them.

3.2.3 Radon inverse transform

Now the inverse, or backward, Radon transform algorithm can be built upon this ad-joint transform. The ad-joint transform will be used as an approximate inverse operator of the forward transform in the theory of iterative

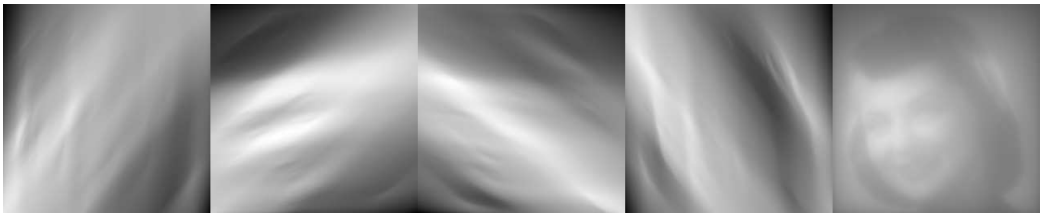


Figure 3.5: The ad-joint discrete Radon transform, as defined by eq. 3.6, applied on each quadrant of the forward transform of the image used as example through this paper. The image on the right corresponds to the ad-joint of the whole DRT, and is equivalent to the sum of the four images in the left. It is reminiscent to the unfiltered back-projection method used in medical imaging.

improvement of a solution to linear equations, described in chapter 2.5 of the numerical recipes book from, again, Press[61].

Figure 3.6: The first 5 iterations of recursive multi-grid inverse Radon transform of a DRT.

This method by itself is extremely slow to converge. In order to accelerate this inversion so that it becomes fast, exact and practical Press suggests to use a multi-grid approach, described in the same book[61] chapter 19.6. In figure 3.6, the images depict the convergence after just 5 iterations. Press demonstrated that, in spite of being iterative, it can be properly called a fast inverse, because convergence up to the resolution of the machine never takes more than a few iterations. Additionally it is well conditioned, so that noise do not preclude the inversion.

3.3 DRT variations based on pruning

The slopes to be evaluated are those that generate ascents varying from 0 to $N-1$ pixels on the vertical axis whilst traversing N pixels on the horizontal axis. For each slope there will be as much as $2N-1$ displacements to be calculated. In case of the slope being maximum all the $2N-1$ displacements are different than zero, whilst when slope is minimal only N displacements will really touch the image. This can be better appreciated in figure 3.2.

These new algorithms have as goal to suppress the computations for the output at marginal displacements and by doing so the shape of the new DRTs output for the 4 quadrants will become a $N \times 4N$ fully populated rectangle. In other words, the number of displacements considered for a single quadrant will be always N independently of the slope.

Two alternative algorithms with different uses and properties that overcome aforementioned problem have been developed, while still remain of linearithmic complexity.

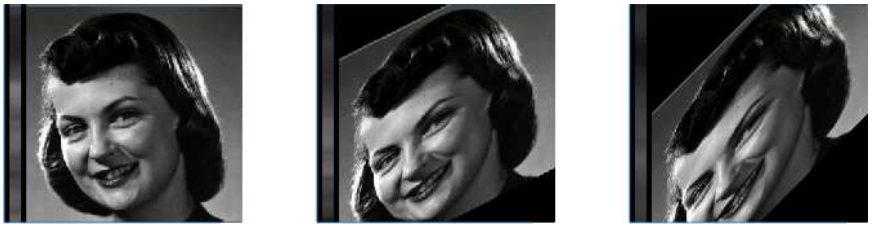


Figure 3.7: From left to right an image sheared to accommodate in a row the discrete lines of Central DRT for minimal, intermediate and maximal slope of the 0 to 45 degrees quadrant.

The equivalent shearing for different slopes shown in figure 3.1 for conventional DRT, is now shown for Central DRT in figure 3.7. The idea behind Central DRT is to compute exclusively the N displacements around the projection of the center of the image. Therefore the output of Central DRT should be coincident with a portion, –the central band on a light gray color on figure 3.2 surrounded on yellow–, of the output of the conventional DRT –surrounded on red–. It can be seen that most of the input, when sheared and truncated to the central N displacements, will still be considered. On fig. 3.2 the values inside the red area but outside the yellow area, those that are going to be discarded, are less than a 15% of the total. It will be demonstrated that this can be achieved with a saving of almost half of the computations

and in certain scenarios, this central band can substitute the whole output of the conventional DRT. The proposed transform will save computations in the forward transform, but although the backward transform exists and converges, it will be much slower than the backward transform of conventional DRT, becoming practically unusable.

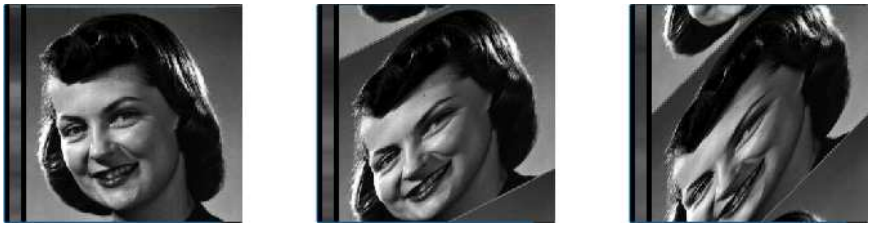


Figure 3.8: From left to right an image sheared to accommodate in a row the discrete lines of Periodic DRT for minimal, intermediate and maximal slope of the 0 to 45 degrees quadrant.

The second alternative transform that will be proposed will also have a reduced output size and a lesser number of computations compared to conventional DRT. But now there will also exist a fast backward transform. However its output will, mostly, not be coincident with any portion of the conventional DRT. As can be noticed in figure 3.8 the Periodic DRT will operate as if the input image has been extended by periodicity. It will be coincident with conventional (and central) DRT for certain slopes and displacements, but in most cases it will add replicas of the input where the other DRTs would add zeroes.

3.3.1 Central DRT

Discrete lines in Central DRT

Figure 3.9 shows, similarly to fig. 3.3 for conventional DRT, how the discrete lines of Central DRT operate on a 8×8 image. Input data must still be padded in the displacement dimension, but an increase of $N/2$ rows is sufficient. This scheme shows an increase of $N-1$ rows for the sake of comparison with fig. 3.3.

The circles in the first column of each rectangle are the displacements that the central DRT will compute for that slope. Now they will always be N independently of the slope. Their position, in order to maintain centered the

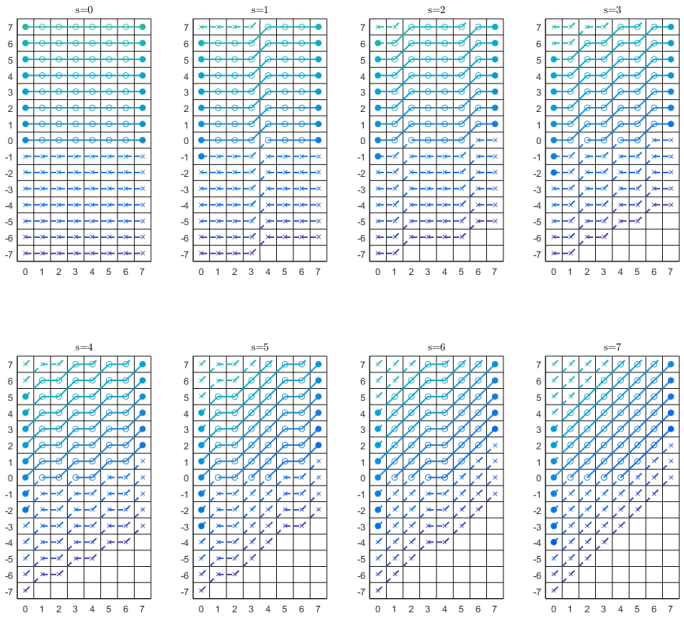


Figure 3.9: A depiction of the whole set of discrete lines covering the displacements and slopes as computed by Central DRT.

line that crosses the image through its center, descends as s increase. Lines whose displacements verify $d > N-1-\lfloor (s+1)/2 \rfloor$, or $d < -\lfloor (s+1)/2 \rfloor$ will be simply ignored.

Memory footprint at partial transforms

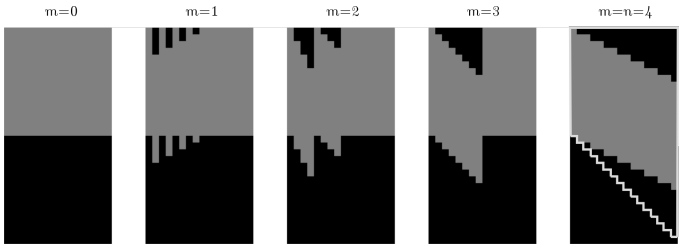


Figure 3.10: Memory patterns of data required for computing Central DRT on a 16×16 image.

The proposal of Central DRT arises from the finding that if we prune the computations of $\mathfrak{R}f\{s,d\}$ values outside this central band, as a collateral

effect half of the values at partial transforms do not need to be evaluated. Remember that the discarded output values represent less than 15% of Radon summations on input data: this is because the zones to be discarded basically operate on the zones padded with zeroes, and so, for certain applications, can be safely ignored.

Pruning techniques have a very reduced effect in Fast Fourier transform[62]: from $O(N \log_2 N)$ to $O(N \log_2 K)$ when only $K < N$ coefficients are required. This is, to achieve the same reduction by half of computations on a 256 1D FFT, the output size should be limited to just 16 coefficients. This reduction by half of computations when halving required coefficients, in an already linearithmic transform, was unexpected.

Figure 3.10 shows the memory patterns of data required for the computation of a quadrant of Central DRT for an image of size 16×16, that is, $N = 16, n = 4$. The rectangles represent visually if a value at partial stages $\widetilde{f}^{\,m}(\underbrace{s \,|\, v}_{horizontal} \;|\; \underbrace{d}_{vertical})$ as defined by eq. (3.4) is needed in order to compute the Central DRT, in that case it is shown in gray, otherwise in black.

This depiction reveals that at each partial stage now only N adjacent values must be computed per column.

The other finding is that the pattern observed here for $N = 16$, –that sort of saw-tooth on the first horizontal half of each stage– follows a formula that will be later described.

Central DRT algorithm

Unraveling this access pattern is in itself the algorithm that is being looked for, since it will be enough to operate exactly as in conventional DRT but only on the highlighted zones. The algorithm described however will add another improvement: the DRT can be formulated with point-wise sums of N-length vectors lying on different columns. This way paving the road to an efficient implementation on SIMD arithmetic units.

So, instead of offering a formula for each value d, it is possible to operate on N consecutive values of two columns on a previous stage to achieve another N-length vector that comprise the required data at a column on a later stage. The indications of where each N-length vector of interest starts for a given stage, size and slope is provided next.

It can be demonstrated that the computation relationship between columns shown in figure 3.10 is in accordance with the diagram shown in

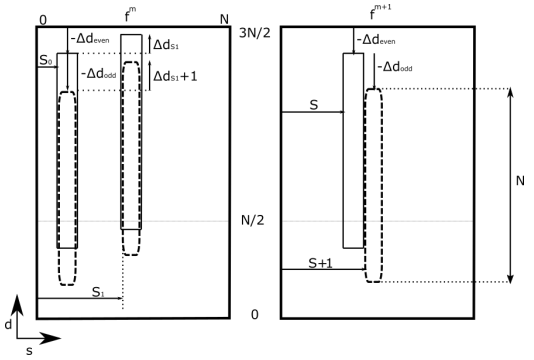


Figure 3.11: Data related in the computation of two consecutive columns, with indexes S and $S+1$, at stage $m+1$, from two columns of the previous stage, with indexes S_0 and S_1.

figure 3.11.

Let consider the data at a certain stage $m+1$ which will be calculated from data from the previous stage m. The index m, denoting the stage, will range from 0 to $n-1$, with $n = \log_2(N)$. Let S be an index at an even column, followed at an odd position by index $S+1$, both belonging to stage $m+1$. In order to calculate the required N values of each of these columns, it is necessary to add together two vectors of length N coming from columns with indexes S_0 and S_1. The starting point, over the d axis, of the vectors to add will be expressed according to the schema and will depend on values $\Delta_{d_{even}}$, $\Delta_{d_{odd}}$ and $\Delta_{d_{S_1}}$.

The index S will be subdivided according to its binary expression as follows:

$$S = [0, \underbrace{\sigma}_{m \text{ bits}}, \underbrace{v}_{n-m-1 \text{ bits}}] = 0 + \sigma << 1 + v << (m+1).$$

Then, indexes of stage m will be expressed as

$$S_0 = \sigma + v << (m+1) \qquad \text{and} \qquad S_1 = S_0 + 1 << m.$$

To calculate the starting point of the vector on the other axis, d, we have to define

$$\Phi = \frac{\left(2^{m+1} - (m < (n-1))\right) \cdot \left(2^{n-m-2} - v\right)}{2^{m+1} - 1} \cdot (v < 2^{n-m-2}).$$

With this value we can calculate: $\Delta_{d_{even}} = \lfloor 2 \cdot \sigma \cdot \Phi \rfloor$, $\Delta_{d_{odd}} = \lceil \Phi \rceil$ and $\Delta_{d_{S_1}} = \sigma$.

In these formulas, $\lceil \cdot \rceil$ y $\lfloor \cdot \rfloor$ are, respectively, the rounding operators to the next larger and smaller integer, the $<<$ symbol refers to the binary shift and $<$ is the *lower than* comparison evaluated to 0 or 1.

Those values are applied following the schema on fig. 3.11:

- N data in column S descending from row index $top - \Delta_{d_{even}}$, are the result of adding N data in column $S0$ descending from row index $top - \Delta_{d_{even}}$, and N data in column $S1$ descending from row index $top - \Delta_{d_{even}} + \Delta_{S_1}$.
- N data in column $S+1$ descending from row index $top - \Delta_{d_{even}} - \Delta_{d_{odd}}$, are the result of adding N data in column $S0$ descending from row index $top - \Delta_{d_{even}} - \Delta_{d_{odd}}$, and N data in column $S1$ descending from row index $top - \Delta_{d_{even}} - \Delta_{d_{odd}} + \Delta_{S_1} + 1$.

The resulting algorithm can be found in pseudo-code as Algorithm 5. The last **for** loop rectifies the central band so that the output is a $N \times N$ square.

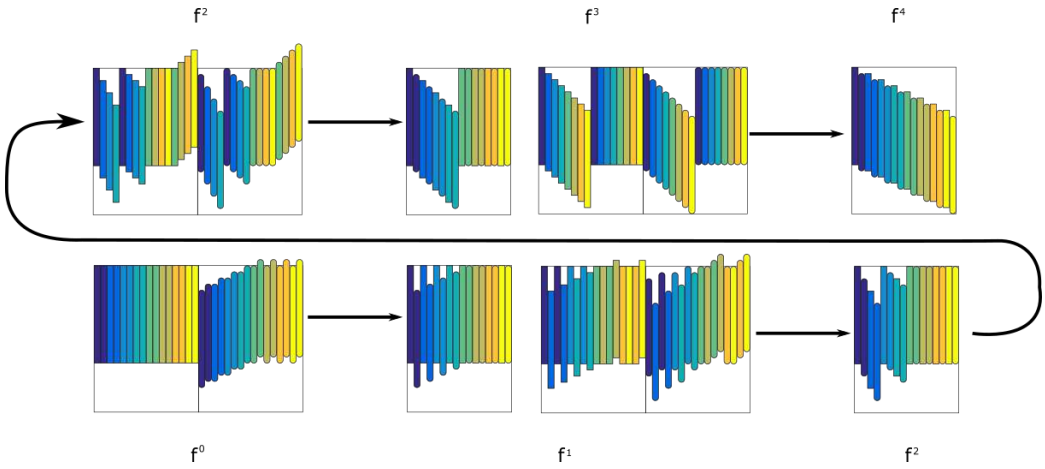


Figure 3.12: N-length vectors lying on different columns to be added at each stage of the algorithm for a 16×16 image.

Figure 3.12 shows the vectors to be added at each stage of the algorithm for a 16×16 image. From left to right and from bottom to top: initial stage, partial stages and final stage of the transform. Each stage, labeled as

Algorithm 5 Compute the Central DRT of a quadrant

Input: Image f(x, y) consisting of $N \times N$ data
Output: Central Radon transform of f, $\Re f$(s, d) consisting of $N \times N$ data

$n \leftarrow \mathbf{log2}(N)$
$f^m \leftarrow \mathbf{zeros}(N,\ 3 \cdot N/2)$
$f^{m+1} \leftarrow \mathbf{zeros}(N,\ 3 \cdot N/2)$
$f^m(0 : N-1,\ N/2 : 3 \cdot N/2 - 1) \leftarrow f(0 : N-1,\ 0 : N-1)$
$top \leftarrow 3 \cdot N/2 - 1$
for $m = 0$ to $n - 1$ **do**
    for $s = 0$ to $N/2 - 1$ **do**
        $\sigma \leftarrow s\ \&\ ((1 << m) - 1)$
        $\boldsymbol{v} \leftarrow s >> m$
        $S_0 \leftarrow (\boldsymbol{v} << (m+1)) + \sigma$
        $S_1 \leftarrow S_0 + (1 << m)$
        $S \leftarrow s << 1$
        $\Phi \leftarrow ((1 << (m+1)) - (m < (n-1))) \cdot ((1 << (n-m-2)) - \boldsymbol{v})$
          $/\ ((1 << (m+1)) - 1) \cdot (\boldsymbol{v} < (1 << (n-m-2)))$
        $\Delta_{d_{even}} \leftarrow \mathbf{floor}(2 \cdot \sigma \cdot \Phi)$
        $\Delta_{d_{odd}} \leftarrow \mathbf{ceil}(\Phi)$
        $\Delta_{d_{S_1}} \leftarrow \sigma$
        $f^{m+1}(S,\ top - \Delta_{d_{even}} - N : top - \Delta_{d_{even}}) \leftarrow$ ▷ Pointwise sum on Column S
    $sum($ $f^m(S_0,\ top - \Delta_{d_{even}} - N + 1 : top - \Delta_{d_{even}})$,
          $f^m(S_1,\ top - \Delta_{d_{even}} + \Delta_{d_{S_1}} - N + 1 : top - \Delta_{d_{even}} + \Delta_{d_{S_1}})$ $)$
        $f^{m+1}(S+1,\ top - \Delta_{d_{even}} - \Delta_{d_{odd}} - N + 1 : top - \Delta_{d_{even}} - \Delta_{d_{odd}}) \leftarrow$ ▷ Pointwise sum on Column $S+1$
    sum($f^m(S_0,\ top - \Delta_{d_{even}} - \Delta_{d_{odd}} - N + 1 : top - \Delta_{d_{even}} - \Delta_{d_{odd}})$,
          $f^m(S_1,\ top - \Delta_{d_{even}} - \Delta_{d_{odd}} + \Delta_{d_{S_1}} + 1 - N + 1 :$
                $top - \Delta_{d_{even}} - \Delta_{d_{odd}} + \Delta_{d_{S_1}} + 1)$ $)$
    end for
    $f^m \leftarrow f^{m+1}$
    $f^{m+1} \leftarrow \mathbf{zeros}(N,\ 3 \cdot N/2)$
end for
for $s = 0$ to $N - 1$ **do**
    $\Re f(s,\ 0 : N-1) \leftarrow f^m(s,\ \mathbf{floor}(top - (s+1)/2) - N :\ \mathbf{floor}(top - (s+1)/2))$
end forreturn $\Re f$

f^m, with $m = 0..4$, shows data positions that will be accessed to calculate the next stage. Accesses for even column computations are shown with flat corners whilst odd columns accesses are shown with rounded corners.

At each stage of the computation there will be two memory buffers storing the data at stage m and the data at stage $m + 1$. But here, for the sake of depiction, accesses from odd and even columns are shown isolated, or otherwise they would overlap, but they access to the same columns for a pair of S and $S+1$ computation, only that with different starting row index. The result of computation at a stage, is shown again as input for the next step of the transformation.

It can be seen in the schema how in each stage only the data required by a later stage are computed. During the transformation, positions outside the $0 : 3N/2$ zone are sometimes accessed. These accesses are simply ignored because they would add zeros.

An example of output of Central DRT will be shown later, after exposing the Periodic DRT algorithm.

Additionally, a coupled pair of variable changes can make the fetching from fm and storing to $fmp1$ be constrained to the N size in the y axis instead of the otherwise needed $3N/2$. This is interesting when facing a glsl implementation, where the output and input sizes are directly affecting performance. This will be further studied on chapter 4.

3.3.2 Periodic DRT

Equations (3.3), (3.4) and (3.5), remained untouched when defining Central DRT, because it computes the same than conventional DRT, only that for fewer values. The algorithm had to be modified to operate on half of the values, but the operations carried out on those values did not change. Now it will be the opposite, the algorithm remains mostly untouched from conventional DRT, but there will be a slight change in eq. (3.4).

It is desirable, in the case of Periodic DRT, that discrete lines do not abandon the square $N \times N$ region where the input is defined, instead, if for a certain slope and displacement a discrete line is going to cross the upper limit through position $\{x, N - 1\}$, it will be forced to reenter at position $\{x + 1, 0\}$.

This *line wraparound effect* is normally undesired, because, such definition of lines turns the transform useless for tomographic inversion. In Fourier based transforms this effect is counteracted by additional padding.

But some applications can benefit from a collateral effect of this redefinition of discrete lines: the forward transform that emerges has a fast inversion algorithm, something that Central DRT has not.

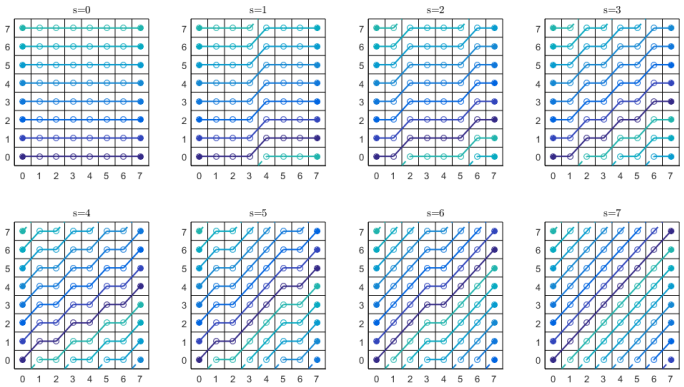


Figure 3.13: A depiction of the whole set of discrete lines covering the displacements and slopes as computed by Periodic DRT.

In figure 3.13 it is depicted the expected behavior for discrete lines in Periodic DRT. Lines with parameters $\{s, d\}$ will finish now on positions $\{N-1,\ (d+s) \mod N\}$. To accomplish such behavior, the partial transform equation must be redefined:

$$\widetilde{f}^{\,m}(\overbrace{s_{n-m},\ \boldsymbol{\sigma}}^{\boldsymbol{s}} \,|\, \mathbf{v} \,|\, d) = \sum_{\mathbf{u}\in\mathbb{Z}_2^m} f\left(\lambda(\mathbf{u},\mathbf{v}) \,|\, \left(l^m_{\lambda(\boldsymbol{s})}(\mathbf{u}) + d\right) \mod N\right) \tag{3.7}$$

That change translates directly into the mapping equation:

$$\widetilde{f}^{\,m+1}(\overbrace{s_{n-m-1},\ \boldsymbol{\sigma}}^{\boldsymbol{s}} |\mathbf{v}| \, d) = \widetilde{f}^{\,m}(\boldsymbol{\sigma}|\, 0, \mathbf{v}|\, d) + \widetilde{f}^{\,m}\left(\boldsymbol{\sigma}|\, 1, \mathbf{v}|\, \left(d+\lambda(\boldsymbol{s})\right) \mod N\right) \tag{3.8}$$

The input image must not be padded now. The algorithm for a quadrant is given in Algorithm 6. This algorithm is a more direct translation of the equations defining the transform than before, on Central DRT, because operations are performed again at datum level, not on N-length vectors.

The output can be rectified applying a circular shift of magnitude $\lfloor \frac{s+1}{2} \rfloor$ to each column of the 4 quadrants output, $\mathfrak{R}f(s,\cdot)$.

Algorithm 6 Compute the Periodic DRT of a quadrant

Input: Image f(x, y) consisting of $N \times N$ data
Output: Periodic Radon transform of f, $\mathfrak{R}f$(s, d) consisting of $N \times N$ data

1: $n \leftarrow \mathbf{log2}(N)$
2: $f^m \leftarrow f$
3: $f^{m+1} \leftarrow \mathbf{zeros}(N,\ N)$
4: **for** $m = 0$ to $n - 1$ **do**
5: **for** $d = 0$ to $N - 1$ **do**
6: **for** $v = 0$ to $(1 << (n - m - 1)) - 1$ **do**
7: **for** $\sigma = 0$ to $(1 << m) - 1$ **do**
8: $f_0 \leftarrow f^m(\sigma + (v << (m + 1)),\ d)$
9: $f^{m+1}((\sigma << 1) + (v << (m + 1)),\ d) = f_0 + \ldots$ ▷ Case $s_{n-m-1} = 0$
10: $f^m(\sigma + (1 << m) + (v << (m + 1)),\ (d + sigma) \mod N)$
11: $f^{m+1}(1 + (\sigma << 1) + (v << (m + 1)),\ d) = f_0 + \ldots$ ▷ Case $s_{n-m-1} = 1$
12: $f^m(\sigma + (1 << m) + (v << (m + 1)),\ (d + 1 + sigma) \mod N)$
13: **end for**
14: **end for**
15: **end for**
16: $f^m \leftarrow f^{m+1}$
17: $f^{m+1} \leftarrow \mathbf{zeros}(N, N)$
18: **end for**
19: $\mathfrak{R}f \leftarrow f^m$ **return** $\mathfrak{R}f$

3.3.3 Comparison of Central and Periodic DRT

Figure 3.14 shows the output of conventional, Central and Periodic DRTs when fed with the image containing two black circles on a white background shown on a corner of the conventional DRT output. Those two circles have diameters 240px and 120px in a $N = 256$ square image. Additionally 4 pixels in the center of the image are black.

Those images reveal the internal functioning of each DRT. It can be noticed the particular shape of conventional DRT in order to contemplate every possible displacement and slope; how Central DRT contains the central band of conventional DRT rectified and with the center of the image projecting always in the central displacement of the DRT, at index $d = \frac{N}{2}$; and the Periodic DRT can be considered as a mixture of both, in the sense that the displacement dimension now remains of size N independently of the slope, but those projections not considered in Central DRT, are now again present only that added to a position already occupied by another displacement of the central band. The value occupying position $\mathfrak{R}f(s, d_c)$ in conventional DRT separated δd values with respect to the displacement of the center of the image, $d_o = N/2 + \lfloor \frac{s+1}{2} \rfloor$, for a certain slope s, i.e. $\delta d = d_c - d_o$, $|\delta d| > \frac{N}{2}$ will be added to the value $\mathfrak{R}_P f(s, (N/2 + \delta d + N) \mod N)$ in Periodic DRT.

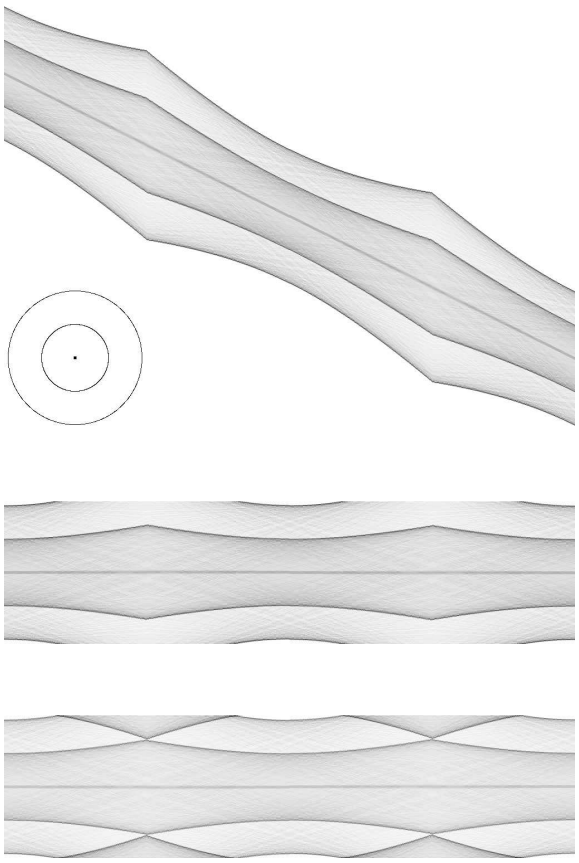


Figure 3.14: From top to bottom: conventional DRT, Central DRT and Periodic DRT of a 256×256 image of two concentric circles, shown bottom left of first sub-figure.

Ad-joint operators of Central and Periodic DRT

Both proposed transforms have an ad-joint, also called back-projection, operator. In the case of Periodic DRT it will be based on the inverse mapping of eq. (3.8):

$$\begin{aligned} \widetilde{f}^{\,m}(\boldsymbol{\sigma}|\,0,\mathbf{v}|\,d) \mathrel{+}= \widetilde{f}^{\,m+1}(s_{n-m-1},\,\boldsymbol{\sigma}|\mathbf{v}|\,d) \\ \widetilde{f}^{\,m}\Big(\boldsymbol{\sigma}|\,1,\mathbf{v}|\,\big(d+s_{n-m-1}+\lambda(\boldsymbol{\sigma})\big) \mod N\Big) \mathrel{+}= \widetilde{f}^{\,m+1}(s_{n-m-1},\,\boldsymbol{\sigma}|\mathbf{v}|\,d) \end{aligned} \tag{3.9}$$

In the case of Central DRT, the equation (3.6) remains valid. Apart from that the ad-joint operator has to repeat every step on the forward transform but on the reverse order, with outer loop of algorithm for a quadrant descending from $m = n - 1$ to $m = 0$.

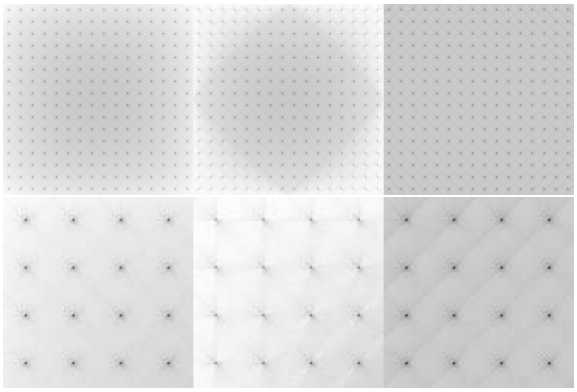


Figure 3.15: Top row, from left to right, the respective ad-joint operators applied to the conventional, Central and Periodic forward DRTs of an input constituted by 16×16 equispaced black pixels. Bottom row, details of top left corners of back-projections in the same order.

Figure 3.15 shows the result of applying the respective ad-joint operators on a previously forward transformed image containing equispaced *deltas*, for each type of transform: conventional, Central and Periodic, in that order from left to right.

There is a sort of *vignetting* in conventional DRT due to the different length of discrete lines that are considered in this method: every line traverses N values, but depending on the slope and displacement, more or less of those values correspond to real pixels on the input image or correspond to zeroes in the padded zone.

The back-projected result corresponding to Central DRT is equivalent to that of conventional DRT but only inside of the rhomboid with vertexes $\{0, N/2\}, \{N/2, N-1\}, \{N-1, N/2\}$ and $\{N/2, 0\}$. The *vignetting* effect is very notorious outside that zone. Moreover the shapes around each originally black dot are not symmetric: those shapes are lacking the discrete lines discarded by this method.

The back-projection corresponding to Periodic DRT is the more symmetric of the three results and it is not affected by the *vignetting* effect, simply

all N values considered for each value of the output have traversed effectively N pixels of the input.

Convergence of inversion algorithm

With the aforementioned ad-joint operators the same technique described by Press[53] for conventional DRT has been applied to Central and Periodic DRT.

Convergence of inversion of Periodic DRT is as fast as that of conventional DRT.

In the case of Central DRT, even if the eigenvalues of the multiplication of the ad-joint and forward operators expressed as matrices, are lower than one, the convergence of its backward transform is fast only in the previously mentioned rhomboid central zone. Outside of that zone, it converges so slowly that it ruins the speedup gained on the forward step.

The inversion method proposed by Press reduces the error on image domain, based on observations of the error on Radon domain, where the trace of cornered line integrals are basically absent.

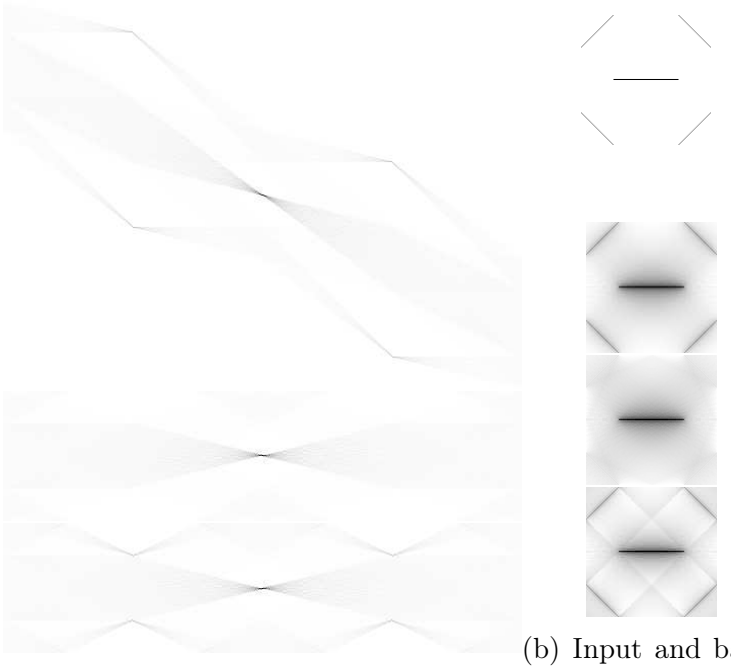
Figure 3.16 shows the behavior of combined forward and back-projection transforms for each DRT type, operating on an input with an horizontal line inside the rhombus inscribed in a square zone described for Central DRT, and then additional smaller lines slanted and positioned closer to each corner.

The cornered lines are part of those not considered by Central DRT and so they leave no trace nor in the forward transform, nor in the back-projection. On the other hand, the back-projection in the case of Periodic DRT is good enough for the method to converge.

Figure 3.17 shows the quality of inversion achieved for each type of DRT after 3 and 5 iterations. Surprisingly Periodic DRT performs even better than conventional. Central DRT inversion is only valid in the central rhomboid zone.

3.4 Simultaneous computation of DRT quadrants for its efficient implementation

In the work [63] a new algorithm was proposed. One of the most prominent drawbacks that the previous described algorithms is that the algorithm con-



(a) Forward Transforms

(b) Input and back-projections

Figure 3.16: Forward and back-projected transforms of an input consisting of one horizontal line in the center, and four lines closer to the corners. From top to bottom, conventional, Central and Periodic DRT.

(a) Inversion after 3 iterations

(b) Inversion after 5 iterations

Figure 3.17: From top to bottom, quality of the inversion for conventional, Periodic and Central DRTs.

sists of the computation of four independent tasks. Alternatives that avoid processing the four quadrants independently were explored.

There is a preceding work from Donoho and Huo [64] that gives a deep insight on multi-scale line analysis of images, but then, it simultaneously exceeds and ignores the purpose of the present work. As stated by the authors, on section 8.1, "*we are specifically* not *interested in fast approximate calculation of the multi-scale Radon transform*". Instead they generate a corpus of scale/location/orientation invariant analysis based on dyadically-organized line segments. Our intention is, indeed, the opposite of that expressed in the quoted paragraph: we want to distillate all that corpus to extract just a fast and efficient algorithm for computing the approximate Radon transform. We will name it *Perimeter Discrete Radon Transform*, PDRT. The data structure and algorithm we will derive is completely original and it is not –implicitly nor explicitly– given in that work.

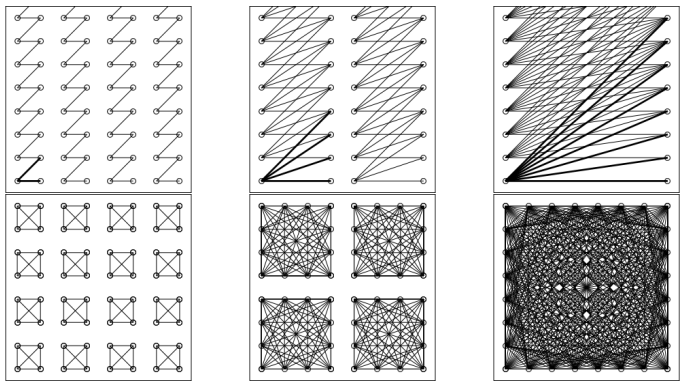


Figure 3.18: On top, depiction of lines being considered at several scales of conventional DRT for the first quadrant ($y = x \cdot slope + d$). On bottom, same scheme for lines within dyadic squares at several scales.

3.4.1 Founding idea on Perimeter DRT

The formulae of multi-scale DRT are cause and consequence of this particular strategy for transforming data through increasingly wider columns. The partial transform definition is the key to understand this fact. But the depiction on top of figure 3.18 is enough to understand what is preventing a single formula from solving as elegantly as before the DRT of all quadrants simultaneously: it is possible to grow transforming columns and taking into

account every possible vertical displacement and still work in-place reusing the same memory footprint that stores the input image, as long as the lines to be solved are of the form $y = x \cdot slope + d$; but it is not possible to grow transforming simultaneously on vertical and horizontal, and it is neither possible to consider simultaneously ascents and descents.

Therefore, an alternative strategy must be considered. It is depicted on bottom of figure 3.18. Instead of partially transforming columns or rows independently, both must be tackled simultaneously and so the sub-regions to be partially transformed must be sub-squares of the total squared domain. And this is basically the only link to the work on *beamlets*[64]: our partial stages of transformation remind what Donoho and Huo denominate *complete Recursive Dyadic Partition*. The solution will cover the whole $N \times N$ domain, and will be constructed from the combination of DRTs from its four dyadic sub-squares of size $\frac{N}{2} \times \frac{N}{2}$; and those from the 16 previous stage DRTs of size $\frac{N}{4} \times \frac{N}{4}$... and so forth.

Or, if we reverse the order of explanation, and build from bottom to top: at the lowest scale for a given $N \times N$ size, we will have $\frac{N}{2} \times \frac{N}{2}$ sub-squares of size 2×2, which will evolve to $\frac{N}{4} \times \frac{N}{4}$ sub-squares of size 4×4, ...

We will name the resulting algorithm as 'Perimeter DRT', PDRT, because we must work on an alternative data structure that is capable of storing all computations that will be needed at each stage. And this new structure instead of mimicking the cartesian grid of input images, where x and y are gradually replaced by v, σ and d parameters, will now parameterize DRTs with the pair of end-points at the *perimeter* of each sub-square.

3.4.2 Data structure

We are interested on computing the sum of pixels that traverse the input image starting from and arriving at any pixel of the perimeter of any dyadic sub-square at any scale. This set of line segments is complete in terms of slopes and displacements in the sense that it was defined in conventional DRT and includes all the data, partial and final, that were computed by conventional DRT and were part of its discrete output sinogram.

We will next describe a data structure to hold all these data, which are the result of partially transforming input data up to a certain stage. The size of this data structure will vary with stage, thus preventing in-place computation.

As well as before, the input image can be considered as a special case of

the data structure for stage 0 ($N/1 \times N/1$ sub-squares of size 1×1) and the output result can be considered an instantiation of the data structure for the stage n.

At each stage m of partial transform of an input image of size $N \times N$, with $N = 2^n$ and $M = 2^m$, there will be $N/M \times N/M$ sub-squares, each one of size $M \times M$. There will be $4M - 4$ pixels surrounding the perimeter of each sub-square. In order to store the PDRT starting from each of those perimeter pixels and arriving to any other perimeter pixel (including itself) there must be $(4M - 4 + 1) \times (4M - 4)/2$, this *triangular shape* emanates of the fact that the PDRT from pixel i to j is equal to that from pixel j to i.

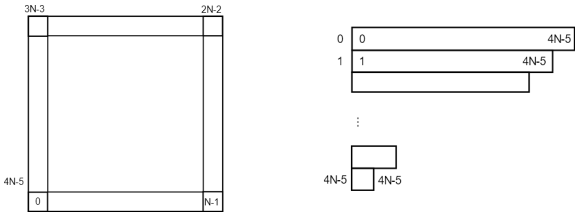


Figure 3.19: Enumeration of vertexes at the perimeter of a square; and triangular shape of the structure containing its Perimeter DRT. Diagrams are not to same scale, notice that the rows are 4 times wider on the Perimeter DRT structure.

Figure 3.19 shows the enumeration of vertices at the perimeter of a square of size $N \times N$. There will be $4N - 4$, ranging from 0 (bottom left vertex) and increasing counterclockwise up to $4N - 5$. On the right it is depicted the contents of the PDRT, right column indicating the index where each line starts and inside the rectangular arrays the index of pixel of arrival. Note that first datum at each *row* holds the value itself of a perimeter pixel.

We can index into this data structure making use of the function listed on algorithm 7.

It is possible to rearrange this structure to convert it into a DRT sinogram by translating slope, displacement coordinates into intersections with perimeter pixels, as depicted in figure 3.20. Of course, applying similar reasoning you can rearrange a sinogram as an PDRT.

Algorithm 7 Returns a linear memory address in a PDRT structure from a set of parameters describing a line

Input: Parameters *indIn*, *indOut*, *sx*, *sy*, m, n
Output: Memory address of line segment starting in pixel *indIn*, arriving at pixel *indOut*, within subsquare of coordinates (*sx, sy*), at partial stage m, in a problem of size n.

if $indIn > indOut$ **then**
    swap$(indIn, indOut)$
end if
$N \leftarrow 2^n$
$M \leftarrow 2^m$
$nSquares1D \leftarrow N/M$
$nVertex \leftarrow 4 \cdot (M-1)$
if m $==$ 0 **then**
    $sizeRadonSquare \leftarrow 1$
else
    $sizeRadonSquare \leftarrow (nVertex+1) \cdot nVertex/2$
end if
$n \leftarrow in$
$a1 \leftarrow nVertex$
$d \leftarrow -1$
$an \leftarrow a1 + (n-1) \cdot d$
$sizePreviousIn = (a1 + an) \cdot n/2$ ▷ The *triangular shape* gives rise to an arithmetic serie
$memIndex \leftarrow (ySquare \cdot nSquares1D + xSquare) \cdot sizeRadonSquare + ...$ ▷ We skip previous subsquares...
        $sizePreviousIn + ...$ ▷ then skip the previous *rows*...
        $indOut - indIn$ ▷ and lastly skip the previous *columns*
return $memIndex$

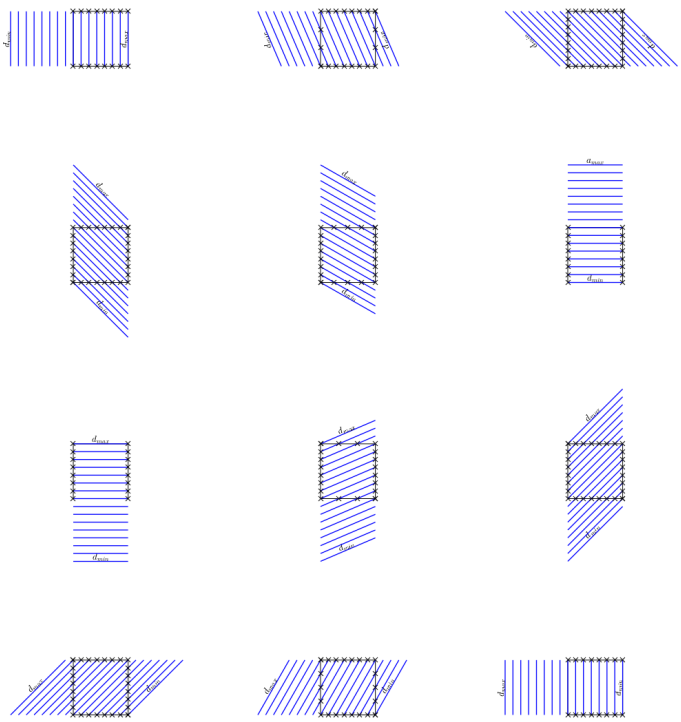


Figure 3.20: Intersections for every displacement for minimum, intermediate and maximum slope (from left to right) for first, second, third and fourth quadrant (from top to bottom) of a 8×8 problem.

3.4.3 Algorithm

The algorithm listed on 8 computes the partial transform of each sub-square at each partial stage. The order of computation of sub-squares and pair of vertices within a sub-square is irrelevant, but stages should be computed in order from 1 to n, as each stage is built upon data from previous partial transform stage. At last stage, when $m = n$, there will be just one square to be considered and its PDRT is the solution.

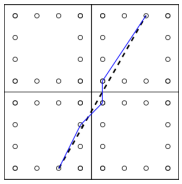
In that algorithm the transform between input data and stage 1 is treated as a special case. Data within each sub-square of size 2×2 are named as A, B, C and D, and the PDRT consisting of the sum of A with each vertex (including itself alone); then B alone, and summed with C and D; then C alone and C with D, and D alone are directly computed and stored.

The rest of stages compute every combination of input and output vertex at perimeter ($indIn$ and $indOut$) within each sub-square (of coordinates

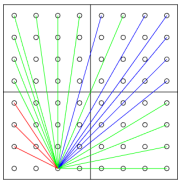
Algorithm 8 Computes the PDRT of an image

Input: image f of size $N \times N$, with N a power of two
Output: $PDRT$ of input

$N \leftarrow$ **sizeof**(f)
$n \leftarrow$ **log2**(N)
$maxSize \leftarrow (4 \cdot N - 3) \cdot (4 \cdot N - 4)/2$ ▷ Last stage will hold the maximum size of PDRT
$f^m \leftarrow$ **zeros**$(1, maxSize)$ ▷ We allocate two buffers of that maximum size
$f^{m+1} \leftarrow$ **zeros**$(1, maxSize)$ ▷ One buffer for current stage and one for next stage, m and $m+1$
$f^m(0 : N \cdot N - 1) \leftarrow f$ ▷ We copy the input to partial transform at current stage
▷ We compute PDRT of subsquares of size 2x2 as a special case
for $squareY \leftarrow 0$ **to** $N/2 - 1$ **do**
  for $squareX \leftarrow 0$ **to** $N/2 - 1$ **do**
    $A \leftarrow f^m(squareY \cdot 2 \cdot N + squareX \cdot 2)$
    $B \leftarrow f^m(squareY \cdot 2 \cdot N + squareX \cdot 2 + 1)$
    $C \leftarrow f^m((squareY \cdot 2 + 1) \cdot N + squareX \cdot 2)$
    $D \leftarrow f^m((squareY \cdot 2 + 1) \cdot N + squareX \cdot 2 + 1)$
    $f^{m+1}((SquareY \cdot N/2 + SquareX) \cdot 10 + [0 : 9]) = ...$
      $[A, A + B, A + C, A + D, B, B + C, B + D, C, C + D, D]$
  end for
end for
▷ The rest of stages starting from stage 2 are computed next
for $m \leftarrow 1$ **to** $n - 1$ **do** ▷ Will compute stage m+1 from data at stage m
  $MP1 \leftarrow 2^{m+1}$
  $nVertex \leftarrow 4 \cdot MP1 - 4$
  $nSquares \leftarrow N/MP1$
  for $squareY \leftarrow 0$ **to** $nSquares - 1$ **do**
    for $squareX \leftarrow 0$ **to** $nSquares - 1$ **do**
      for $indIn \leftarrow 0$ **to** $nVertex - 1$ **do**
        for $indOut \leftarrow indIn$ **to** $nVertex - 1$ **do**
          $memIndexWrite \leftarrow$ **linearIndex**$(indIn, indOut, squareX, squareY, m + 1, n)$ ▷ Writing address at stage m+1
          $f^{m+1}(memIndexWrite) \leftarrow 0$
          $cuts \leftarrow$ **cutsAtSubsquare**$(indIn, indOut, m + 1)$ ▷ Compute line segments cuts at dyadic boundaries
          while not isEmpty$(cuts)$ **do**
            $c \leftarrow$ **pop**$(cuts)$
            $memIndexRead \leftarrow$ **linearIndex**$(c.indIn, c.indOut, c.squareX, c.squareY, m, n)$ ▷ Reading address at stage m
            $f^{m+1}(memIndexWrite) \leftarrow f^{m+1}(memIndexWrite) + f^m(memIndexRead)$ ▷ Accumulate
          end while
        end for
      end for
    end for
  end for
  swapBuffers(f^m, f^{m+1})
end for
return f^{m+1}



(a) A line in a square of any given scale can be computed by reusing at most three line segments of its 4 children dyadic sub-squares. The discrete lines are not strictly collinear.



(b) Number of line segments from 2^{nd} perimeter vertex to each other are represented by a color. Red, green and blue meaning 1, 2 and 3 line segments, respectively.

Figure 3.21: Cuts at dyadic boundaries in a 8×8 problem.

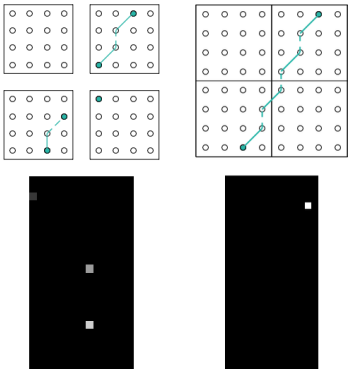


Figure 3.22: A depiction of the sums carried out to compute the 3 line segments shown in figure 3.21a

$squareX$ and $squareY$) making use of two functions: **linearIndex** which was listed in algorithm 7; and **cutsAtSubsquare**, which will be described below, making use of figure 3.21.

In the sub-figure 3.21a, it is depicted the desired behavior of **cutsAtSubsquare**$(2, 15, 0, 0, 3, 3)$, that is, which are the line segments that arise when joining vertex 2 and 15 at perimeter of the unique sub-square at last stage of a $N = 2^3$ problem? And the answer should be $\{\{indIn = 2, indOut = 5, squareX = 0, squareY = 0\}, \{9, 9, 1, 0\}, \{0, 7, 1, 1\}\}$.

The algorithm making use of that structure should fetches the appropriate 3 data from previous stage (at positions 27, 150, 241) and write their sum in the appropriate place (at position 68) in the stage being computed, as seen on figure 3.22.

In order to find the segments to be added in each case we propose to take advantage of a fact exemplified on sub-figure 3.21b for a initial vertex ranging from 0 to $M/2-1$: there will be an extra line segment to be considered when arrival index ranges from $M/2$ to $M/2+nVertex/2-1$; and another extra line segment when arrival index ranges from $M+M/2$ to $M+M/2+nVertex/2-1$, that is, when arrival is behind of the vertical or horizontal lines that cut by half the parent square. With an exception on the diagonal, when arrival vertex is $nVertex/2$ indices apart from initial, in that case there will be 2 instead of 3 line segments.

Line segments should be described in the *cuts* structure by means of its entry and exit pixel to its child sub-square. This will be computed using a rule of three restricted to integer indices.

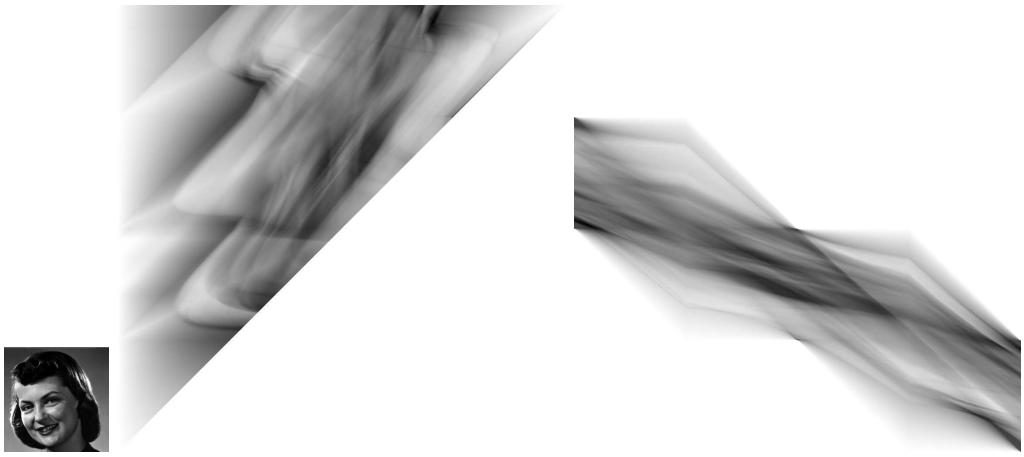


Figure 3.23: From left to right, an input image, its Perimeter DRT, and its rearrangement as a conventional DRT sinogram. Colors in transformations are inverted, and hence white represents value zero.

s

Chapter 4

DRT accelerated on smart-phones

4.1 Introduction

Curvelets are a great tool in general, and, specifically, for the purpose of focus estimation. The computational power required to run curvelets is the factor that holds back its widespread usage. Therefore, a very fast and efficient implementation of curvelets will be proposed in an attempt to facilitate its usage and apply it to our shape-from-focus problem.

Because the majority of the computational time is spent on the Radon transform, it makes sense to approach this problem first. As it was introduced on the last chapter, the DRT will be used to build a Curvelet transform. This argument was discussed in 3.1. This chapter will focus only on the DRT implementation on different architectures.

4.2 Parallelism, multithreading, autotuning

The conventional, Central and Periodic transforms described in sections 3.2, 3.3.1 and 3.3.2 have a set of common characteristics that allow us to easily apply coarse grained parallelism.

4.2.1 Coarse grained parallelism

The algorithms can be rewritten in order to have the variables d, v and σ be a function of generic x, y variables that travel the $fmp1$ memory. d can be put as function of y and v, while $sigma$ can be function of x. This is

important because it allows us to have two linear variables in which we can apply parallelism by dividing its computation among threads. These two variables will have constant size across all steps.

4.2.2 Vectorization and SIMD

In order to achieve a more fine grained parallelism, we need to write custom code that is architecture dependent. An optimization that includes fine-grained vectorization has been previously done for the 4D:3D DRT [14] with the goal of speeding the refocusing of plenoptic images on a smart-phone, and after applying coarse-grained parallelization.

For example, on Intel®processors, if we were write something like:

Listing 4.1: Example code, expressed in plain C++.

```
const int n = 512;
for (int x = 0; x < w; x++) {
   for (int y = 0; y < h; y++) {
      for (int i = 0; i < n; i++) {
         buffer3[y * w + x + i] += buffer1[y * w + x + i] + buffer2[i];
      }
   }
}
```

we could take advantage of one of the supplementary instruction sets that Intel processors have. For example the SSE2 SIMD instruction set, designed to accelerate video streaming. The code can be accelerated by using intrinsics and it translates to:

Listing 4.2: Example code, with vectorization, tailored to a Intel®microprocessor.

```
const int n = 512;
const int vectorsize = 128;
__m128d b1, b2, b3;
for (int x = 0; x < w; x++) {
   for (int y = 0; y < h; y++) {
      for (int i = 0; i < n / vectorsize; i++) {
         /* load 128 values in parallel from the array */
         b1 = _mm_load_pd(buffer1 + y * w + x + i);
         b2 = _mm_load_pd(buffer2 + i);
         b3 = _mm_add_pd(b1, b2);
         /* store 8 values in parallel to the array */
         __mm_store_pd(buffer3 + y * w + x + i, b3);
      }
   }
}
```

Note that the number of iterations has been reduced by the size of the vector (128). We could theoretically calculate the number of clocks needed per iteration and compare it to the original code. The number of clocks is also dependent on the architecture, so, for a Skylake, the number of clocks per instruction (CPI) of the load, add and store operations are 0.33, 0.5 and

0.33, respectively. Multiplying this with the number of iterations would give the theoretical number of clocks needed to compute our code, if there would not be pipelining.

The declaration and assignments statements in the code using the variable of type `__m128d` translate directly into the usage of available registers, so, unless there is not enough registers, it will not add additional time. If there are not enough registers, the compiler will decide to use main memory. This is called "register spilling".

Also, the compiler can be aware of these optional instruction sets and vectorize the code for us (`gcc` with the option `-O3`). One can expect to beat the optimization made by the compiler by carefully designing the usage of the available instruction sets and writing directly low level code, as exemplified in the listing 4.2.

If the code must run in ARM processors as well, another codification is yet needed in order to tailor the desired vectorization to a different processor that has different architecture. ARM is RISC based architecture, very common in smart-phones. The NEON SIMD extension is available in ARMv7 based architectures, which is the vast majority of smart-phones, including Android and Apple ones.

Listing 4.3: Example code, with vectorization, tailored to a ARMv7 microprocessor.

```
1  const int n = 512;
2  const int vectorsize = 8;
3  uint16x8_t b1, b2, b3;
4  for (int x = 0; x < w; x++) {
5     for (int y = 0; y < h; y++) {
6        for (int i = 0; i < n / vectorsize; i++) {
7           /* load 8 values in parallel from the array */
8           b1 = vld1q_s16(buffer1 + y * w + x + i);
9           b2 = vld1q_s16(buffer2 + i);
10          b3 = vaddq_u16(b1, b2);
11          /* store 8 values in parallel to the array */
12          vst1q_u16(buffer3 + y * w + x + i, b3);
13       }
14    }
15 }
```

The listing 4.3 shows an example of how the example code could be parallelized. Note that the vector sizes in NEON are much smaller than the ones present in Intel architectures for PCs. In order to successfully make these optimizations and achieve a notable speedup, one must carefully study the targeted architecture, so the acceleration exploits as much as possible the peculiarities of the design of the processing unit. That means completely knowing the available instruction set and how many CPIs each one takes,

the ALUs that execute them, register sizes, how pipelining works in order to avoid stalling, number of register, size of registers, how fast is the cache, what is the cost of a cache miss, etc.

Sometimes after analyzing a particular architecture, one can reach to the conclusion that we must redesign the algorithm at a higher, mathematical level in order to further exploit the architecture. The process of writing highly architecture dependent, optimized code requires a great degree of knowledge, time and ability.

In the further sections we analyze solutions that make implementations independent from underlying architecture, in order to achieve great results on a broad range of devices without having to spend limitless amount of effort.

4.2.3 OpenCL

Modern Android smart-phones have GPUs. The most common GPUs included in the SoCs are ARM®Mali™ and Adreno (Qualcomm). Apple iPhone models come with an SDK (Metal) with the purpose of leveraging computation chunks to the GPU. This has not been the case with Android. While the Android SDK provides an API called RenderScript that could be the equivalent to Metal in Android, RenderScript does not allows to specify where your code execution lives. The fine-grained control needed in some high-performance use-cases is very limited or non-existing.

Both Metal and RenderScript are specific to the Operating System, iOS and Android respectively, meaning that it will never be possible to write RenderScript code that runs *natively* on an iPhone or write Metal code that runs on an Android device.

There is a third disadvantage of writing for these two APIs: we are not able to test an execution on our code on a development machine (for example a PC with an Intel or AMD microprocessor). This is often very desirable for speeding up development times, since it becomes impossible to have many testing devices in some cases, and depends on the specific workflow that a developer or team has chosen. Also can be the case, that code targeting a NVIDIA CUDA and Intel HD GPU platforms are required alongside with Metal and RenderScript code.

To solve the mentioned problems we need a framework that is able to compile highly optimized code for all the targeted architectures. The first one that should come to mind is OpenGL with kernels written on GLSL.

This solution works for all cases, but because it was designed with graphical pipelines in mind, it is not quite adequate for a general-purpose programming use-case. Also, different versions of GLSL support different number of features and have a number of restrictions. For example, the GLSL version is more restrictive on smart-phones than it is on PC. For example smart-phones' GLSL imposes restrictions on the type of data allowed as an input/output buffer, which is not the case for PCs' GLSL. Furthermore, operations such as reductions become very difficult to express. In a GLSL kernel, reading for arbitrary memory positions is allowed (gathering) but writing to arbitrary memory positions (scattering) is not.

A much more flexible framework is OpenCL and it is usually the go-to for writing GPU powered applications on Android and iOS devices. It allows the execution of kernels in CPUs, GPUs, FPGAs and DSPs. It allows a very flexible workflow where you can execute both on the development machine and the target machine. Scattering is also allowed, since the framework has been designed as a general-purpose programming tool.

The listing 4.4 shows the kernel that computes the central DRT of one quadrant. It does not contain the needed host code that compiles the kernel and runs it. OpenCL kernels are intended to compile at run-time, so portability is maintained across devices. OpenCL provides wrappers for Python, Java and defines native APIs for C and C++. If protecting the code from inspection is important, the intermediate code SPIR is available as a target.

Listing 4.4: Kernel that computes a quadrant of the Central Radon transform on OpenCL.

```
__kernel void rd_qRadon(__read_only image2d_t fm, __write_only image2d_t fmp1,
                        short N, short m, short n, float powA, float maxDispM,
                        float stepDispM, short deltaS_S1_M, ushort quarter) {
   // Slopes (angles)
   const ushort s = get_global_id(0);
   // Displacements
   const ushort d = get_global_id(1);
   short deltaD_S1, startD, dispDST, size, RANGE_A, RANGE_B, DST_A;
   short start, end, src_S0_V, v, src_S0, src_S1, sigma, s0;
   half stepDispV, nonIntPwr, maxDispV;
   unsigned int IntPwr;
   v = s >> (m + 1);
   if (v < powA)
      maxDispV = stepDispM * (powA - v);
   else
      maxDispV = 0;
   // is non-integer when m==n
   IntPwr = 1 << m + 1;
   stepDispV = maxDispV / (IntPwr - 1);
   src_S0_V = v * IntPwr;
   sigma = (s >> 1) & (convert_int(native_exp2(convert_float(m))) - 1);
   src_S0 = src_S0_V + sigma;
   s0 = s & 1;
   // s = v * IntPwr + sigma * 2 + s0;
   dispDST = round((sigma * 2 + s0) * stepDispV);
```

```
26      src_S1 = src_S0 + deltaS_S1_M;
27      deltaD_S1 = sigma + s0;
28      startD = 0;
29      RANGE_A = 3 * N / 2 - dispDST - N;
30      RANGE_B = 3 * N / 2 - dispDST;
31      if (m == (n - 1)) {
32         DST_A = N / 2;
33      } else {
34         DST_A = RANGE_A;
35      }
36      ushort px0 = read_imageui(fm, radonSampler, (int2)(src_S0, RANGE_A + d)).x;
37      ushort px1 =
38         read_imageui(fm, radonSampler, (int2)(src_S1, RANGE_A + d + deltaD_S1)).x;
39      if (m >= 8) {
40         px0 = px0 / 2;
41         px1 = px1 / 2;
42      }
43      ushort result = px0 + px1;
44      write_imageui(fmp1, (int2)(s, DST_A + d), result);
45  }
```

4.2.4 Halide

When writing accelerated image processing pipelines, there is a recurrent problem. If we want the program to run as efficient as possible, we need to write an expression of our pipeline that is specific to the hardware it will be run on.

Take, for example, the difference between a typical GPU and a CPU. Because these two architectures are drastically different, it will not make sense that they execute the same code but to have two different versions, each one exploiting different characteristics. The GPU will have a degree of parallelism that is greater than the CPU, while the CPU will have more clock speed and more general-purpose instructions.

At the end of the day, the GPU code is quite different from the CPU code. A smart-phone GPU and a PC GPU are very different as well. Moreover, using languages such as OpenGL (more available across platforms) compared to using OpenCL (less restrictive) also imposes changes to the code. As a result, programmers are often faced with the dilemma of writing modular, simple and portable code or have high efficiency.

Also, pipelines composed by highly efficient kernels lead to inefficient code. This is because the full pipeline must be taken into consideration when optimizing the code. In this case, modularity breaks efficiency because there is no fusion.

Halide solves all these problems with elegance [65, 66, 67]. The language lets the programmer define the algorithm in an almost mathematical dialect. This definition will be valid and unchanged across all platforms and pipelines. Take the example in 4.5. It constitutes the "definition" of a 3×3 Sobel filter.

It is not dependent on hardware, and the order of execution, the locality or storage in memory are simply let undefined.

Listing 4.5: Sobel 3x3 Halide example definition.

```
void sobel3() {
   Func sobel_x_avg(x, y, c) = in(x - 1, y, c) + 2 * in(x, y, c) + in(x + 1, y, c);
   Func sobel_x(x, y, c) = abs(sobel_x_avg(x, y - 1, c), sobel_x_avg(x, y + 1, c));
   Func sobel_y_avg(x, y, c) = in(x, y - 1, c) + 2 * in(x, y, c) + in(x, y + 1, c);
   Func sobel_y(x, y, c) = abs(sobel_y_avg(x - 1, y, c), sobel_y_avg(x + 1, y, c));
   out_sobel(x, y, c) = sobel_x(x, y, c) + sobel_y(x, y, c);
}
```

Now that we have our definition, we can decide how will execute, depending on the architecture 4.6. A parallelization is applied in the x variable and the function "out_sobel" is asked to be computed. That means that the intermediate functions ("sobel_x" and "sobel_y") are not explicitly set to be fully computed as a stage and are only calculated by demand, so they may or may not have a full representation on memory at run time. Halide will generate code for us. This code will be different for the target architecture. For example, for ARM microprocessors, it may add NEON SIMD instructions in order to achieve the 8 element vector size parallelism. In the case of Intel microprocessors, it may add SSE SIMD instructions. If we need to we can easily tweak each scheduling for each platform we want our code to run in.

Listing 4.6: Sobel 3x3 Halide example CPU scheduling.

```
void sched_cpu() {
   out_sobel.vectorize(x, 8); // Parallelize the x var
   out_sobel.compute_root();  // Ask Halide to compute the out_sobel function
}
```

If we want our code to run in GPU, Halide is also capable to generate kernels for CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal and Microsoft Direct X 12. In the example 4.7, the same code is scheduled for a GLSL (OpenGL) kernel.

Listing 4.7: Sobel 3x3 Halide example GPU scheduling.

```
void sched_gpu() {
   out_sobel.bound(x, 0, W);
   out_sobel.bound(y, 0, H);
   out_sobel.bound(c, 0, 4);
   out_sobel.unroll(c);
   out_sobel.compute_root();
   out_sobel.glsl(x, y, c);
}
```

In the case of CPU scheduling, Halide provides a utility called auto-scheduling [68]. It will output a reasonable scheduling for the targeted CPU architecture, simplifying, in many cases, the task of the programmer greatly. GPU auto-scheduling is also developed and probably to be included as a feature soon [69].

If we wanted a bigger processing pipeline, we could combine several definition of algorithm into a pipeline by defining a single schedule for the whole pipeline. In the previous example, the function "out_sobel" could be inline (as "sobel_x" and "sobel_y"). The computation of "out_sobel" is then triggered by the computation of another function that needs values from "out_sobel". By defining the pipeline this way, Halide achives modularity and efficiency at the same time.

The usage of the programming-language Halide has been part of stories of success within modern, real-life computational photography projects such as Google's camera, that includes the HDR+ algorithm [70], used now in many smart-phones. Halide is currently being developed and in constant change. Its adoption seems to only increase overtime and will probably become a *de facto* standard, at least, within the field of computational photography.

4.2.5 Conventional DRT, Central DRT, Periodic DRT and Perimeter DRT implementations in Halide

The algorithms are written differently in Halide. Structure of data, order of execution and barriers are often not explicited by the programmer but inferred by the generator. In Halide, the flow of compilation works as follows: the C++ source code is compiled to produce a generator binary. This generator is executed to produce a pipeline binary. Therefore we can talk about *compilation time*, *generation time*, and *execution time*.

The programmer has to define what are the results of the operations without needing to explicit the data structure holding the data. The programmer also writes the *scheduling* for each desired target. Scheduling effectively separates algorithm code from the target-dependent code. Once this is done, the C++ code is compiled in order to produce a generator. This generator program can generate code for any platform and at this point flow, data structure and barriers are not defined yet. It is at the generation stage that these are defined automatically.

Once the programmer executes the generator specifying the target plat-

form, the executable code for the algorithms (*a pipeline*) is produced. The available platforms at the time of writing are: X86, ARM, MIPS, Hexagon, PowerPC for CPU and CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal and Microsoft Direct X 12 [71].

Given the Android mobile-phone related nature of this thesis, we are specially interested in ARM, Hexagon, OpenCL and OpenGL. Hexagon is a DSP that is designed specifically for handling multimedia processing in smart-phones. In fact it is used as part of the camera pipeline processing in Qualcomm SoCs and is the "secret hardware sauce" in the Google's HDR+ algorithm present in the Google Camera that Pixels phones have [70]. Hexagon would be perfect for our DRT algorithms, but it is not available for user applications as camera software running on it is considered firmware, and a Qualcomm development board is needed in order to develop on the DSP.

Therefore our efforts are mostly centered on ARM and OpenGL.

CPU

In order to implement the DRT on CPU we part from the algorithm defined in 4. In order to write it in Halide language, we just have to define the variables over which the threads or compute units will iterate. In our case the variables will be named x and y, as we have only two dimensions. The rest of variables, the ones originally used to iterate, will be expressed as functions of these two. Also, since we will use data with 8 bits length, we decided to arithmetically shift left additions beyond stage iteration 7 to avoid overflow.

The Halide code embedded in C++ is defined in 4.8 and its autoscheduling is defined in 4.9:

Listing 4.8: Halide code defines define the DRT of a quadrant.

```
Halide::Func fm[M];
Halide::Var x, y;
Halide::Expr m, d, sigma, v, s0, dir_fm, dir_fmp1, f0, f1;
fm[0](x, y) = Halide::cast<uint16_t>(in(x, y - N));
for (int m = 0; m < M - 1; m++) {
   d = y;
   v = x >> (m + 1);
   sigma = (x & ((1 << (m + 1)) - 1)) >> 1;
   s0 = x & 1;
   dir_fm = sigma + (v << (m + 1));
   f0 = fm[m](dir_fm, d);
   f1 = select(d + sigma + s0 >= N * 2, 0, fm[m](dir_fm + (1 << m), d + sigma + s0));
   fm[m + 1](x, d) = Halide::select(m > 7, Halide::cast<uint16_t>(f0 / 2 + f1 / 2), Halide::cast<uint16_t>(f0 + f1));
}
out(x, y) = fm[M - 1](x, y);
```

Listing 4.9: Schedule Halide code for the DRT of a quadrant.

```
in.dim(0).set_bounds_estimate(0, N);
in.dim(1).set_bounds_estimate(0, N);
out.estimate(x, 0, N).estimate(y, 0, N);
```

Note that the for loop only exists at compile time. Once the generator is produced, there will be an unrolled set of statements instead of the loop. Also note that there is no explicit definition of buffers or the size of them. This is decided by the generator automatically and sizes will be inferred from the input buffer (in) and the expressions used to index the stores.

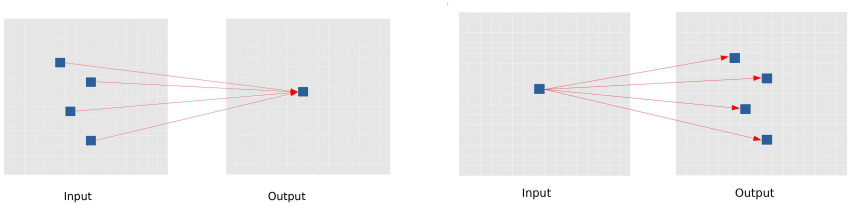
The code expressed above only computes a quadrant. In order to compute the four quadrants we need to call the code four times, each with different orientations of the inputs. The main advantage of using Halide is the following: the generator will generate an optimum implementation for the new pipeline which, in this case consists on producing four quadrants, not only one, and therefore, the strategies used to parallelize can be completely different as the quadrants do not have to be computed sequentially, but using as many computing units as available.

We could write an entirely different scheduling or use the auto-scheduling feature. All of this while maintaining the original code of 4.8. If we wanted to add more processing steps such as a gradient magnitude filter to the input, the whole scheduling can be changed to accommodate it without having to add more buffers. This is very difficult in, for example, OpenCl, where, in order to maintain modularity, an intermediate buffer must be used.

See the code that calls the $qDRT$ in 4.10 and its auto-scheduling in 4.11:

Listing 4.10: Schedule Halide code for the DRT of four quadrants.

```
qDRTf q[4]; // qDRTf class contains the code that defines the DRT of a quadrant
Func q_out[4];
Func clamped_in;
Func pre_out;
Input<Buffer<uint8_t>> in{"in", 2};
Output<Buffer<uint16_t>> out0{"out0", 2};
Output<Buffer<uint16_t>> out1{"out1", 2};
Output<Buffer<uint16_t>> out2{"out2", 2};
Output<Buffer<uint16_t>> out3{"out3", 2};
const int Nm1 = N - 1;
pre_out(x, y) = cast<uint16_t>(0); // Initialize with zeros
clamped_in = Halide::BoundaryConditions::constant_exterior(in, 0);
Func input[4];
input[0](x, y) = clamped_in(y, Nm1 - x);
q[0].qdrtf(input[0], q_out[0]);
input[1](x, y) = clamped_in(x, Nm1 - y);
q[1].qdrtf(input[1], q_out[1]);
input[2](x, y) = clamped_in(x, y);
q[2].qdrtf(input[2], q_out[2]);
input[3](x, y) = clamped_in(y, x);
q[3].qdrtf(input[3], q_out[3]);
out0(x, y) = q_out[0](x, 2 * N - y);
out1(x, y) = q_out[1](Nm1 - x, y);
out2(x, y) = q_out[2](x, 2 * N - y);
```



(a) Gathering. (b) Spreading.

Figure 4.1: Two access patterns.

```
25  out3(x, y) = q_out[3](Nm1 - x, y);
```

Listing 4.11: Schedule Halide code for the DRT of four quadrants.

```
1  in.dim(0).set_bounds_estimate(0, N);
2  in.dim(1).set_bounds_estimate(0, N);
3  out0.estimate(x, 0, N).estimate(y, 0, N * 2);
4  out1.estimate(x, 0, N).estimate(y, 0, N * 2);
5  out2.estimate(x, 0, N).estimate(y, 0, N * 2);
6  out3.estimate(x, 0, N).estimate(y, 0, N * 2);
```

4.2.6 GPU

When faced with a Halide based GPU implementation on a Qualcomm SoC there are two options: OpenCL and OpenGL. The difference lies mainly in the restrictions. While OpenCL has almost the same restrictions as CPU, meaning that the previous algorithm defined in 4.8 is directly compatible with OpenCL, the OpenGL restricts the type of data and size of the inputs and and outputs. They have to be floats and the size must be $N \times M \times 4$. Also OpenGL can deal with gathering, i.e. *fetching* any number of pixels from the input buffer, but cannot deal with spreading, i.e. *storing* in several positions of the output buffer, see figure 4.1.

Since our problem consists on computing four quadrants of the $qDRT$, it translates pretty well with this kind of architecture so we chose OpenGL in order to have a more optimal implementation. With that in mind, let us rewrite the above code in order to use four channels and iterate only on a gathering fashion. The resulting code (4.12 will work well in CPU as well since the restrictions of CPU are only a subset that the OpenGL implementation has.

Listing 4.12: Halide code for the definition DRT of four quadrants on GPU.

```
fm[0](x, y, c) = in(x, y - N, c);
for (dir_type m = 0; m < M - 1; m++) {
   Halide::Expr sigma{"sigma"};
   Halide::Expr s0{"s0"};
   Halide::Expr dir_f0{"dir_f0"};
   Halide::Expr dir_f1{"dir_f1"};
   Halide::Expr f0{"f0"};
   Halide::Expr f1{"f1"};
   Halide::Expr aux{"aux"};
   sigma = (Halide::cast<dir_type>(x) & ((1 << (m + 1)) - 1)) >> 1;
   aux = Halide::cast<dir_type>(x) >> (m + 1) << (m + 1);
   // This clamp will determine the size of fm[i]
   dir_f0 =
      Halide::clamp(Halide::cast<uint16_t>(sigma + aux), 0, N - 1);
   f0 = fm[m](dir_f0, y, c);
   dir_f1 = dir_f0 + (1 << m);
   dir_f1 = Halide::clamp(dir_f1, 0, N - 1);
   s0 = Halide::cast<dir_type>(x) & 1;
   f1 = Halide::select(
      (y + sigma + s0) < (N * 2),
      fm[m](dir_f1, Halide::clamp(y + sigma + s0, 0, N * 2 - 1), c),
      0);
   fm[m + 1](x, y, c) = Halide::cast<uint8_t>(f0 * 0.5f + f1 * 0.5f);
}
out(x, y, c) = Halide::cast<uint8_t>(fm[M - 1](x, y, c));
```

Note that each store (in $fm[m]$) is now indexed with the tuple (x, y, c), therefore gathering access is used only and the expressions are now written following variable changes. See also the code that schedules in 4.13.

Listing 4.13: Scheduling for GPU.

```
for (int i = 0; i < MAX_IT; i++) {
   fm[i]
      .bound(x, 0, VAL_N)
      .bound(y, 0, VAL_N * 2)
      .bound(c, 0, 4)
      .unroll(c)
      .compute_root()
      .glsl(x, y, c);
}
```

Again in order to prepare the inputs to occupy each of the four channels that the third dimension have, we do not need to add any buffer. See code 4.14.

Listing 4.14: Input preparation for DRT in 4 channels.

```
cin = Halide::BoundaryConditions::constant_exterior(in, 0);
int Nm1 = N - 1;
clamped_in(x, y, c) = Halide::select(
      c == 0, cin(y, Nm1 - x, c),
      c == 1, cin(x, Nm1 - y, c),
      c == 2, cin(x, y, c),
      c == 3, cin(y, x, c), 0
      );
q.drtf(clamped_in, q_out, x, y, c);
out(x, y, c) =  Halide::select(
      c == 0, q_out(x, (2 * N - 1) - y, c),
      c == 1, q_out(Nm1 - x, y, c),
      c == 2, q_out(x, (2*  N - 1) - y, c),
      c == 3, q_out(Nm1 - x, y, c), 0
      );
```

Since this implementation works well both in GPU and in CPU, we will stick to it. The implementations for the Periodic DRT and the Central DRT have been implemented in the same way and are very similar.

There are two additional things to have into consideration when working with these two platforms. Sometimes the internal data type does not match, meaning that some operations will yield different results. That is why some Halide casts have been added so it works the same way in CPU and GPU. Also, when the variables that access the input buffers are not directly variables, but expressions, a clamp is needed in order to tell Halide what is the extent of these buffers.

Perimeter DRT on CPU and GPU

The case of the Perimeter DRT is not that straightforward. If we take a look at the algorithm 8, in section 3.4.3, there is a stack of *cuts*. Implementing this on Halide (or any other parallel language) is difficult, so a refactorization was carried out, so that stacks were avoided, see algorithm 9 and 10.

Next, the algorithm has to be compatible with the restrictions that we have in OpenGL (so its implementation's time measures are comparable with the others). In this case this means that we have to take the indexing that is lineal and make it work for a size of $N \times N \times 4$. Also, to turn it into a gathering scheme of accesses, the for loops must iterate over x, y, and c, variables whose bounds coincide with the size of the buffers in $N \times N \times 4$. Therefore the variables $squareY$, $squareX$, $indIn$ and $indOut$ have to be calculated from the x, y and c and the algorithm will iterate over them instead. These changes have been performed from the code in 11 and 12.

Algorithm 9 Computes the PDRT of an image (refactorization part1)

Input: image f of size $N \times N$, with N a power of two
Output: $PDRT$ of input

```
1: N ← sizeof(f)
2: n ← log2(N)
3: maxSize ← (4 · N − 3) · (4 · N − 4)/2                ▷ Last stage will hold the maximum size of PDRT
4: f^m ← zeros(1, maxSize)                               ▷ We allocate two buffers of that maximum size
5: f^{m+1} ← zeros(1, maxSize)                           ▷ One buffer for current stage and one for next stage, m and m + 1
6: f^m(0 : N · N − 1) ← f                                ▷ We copy the input to partial transform at current stage
7:                                                       ▷ We compute PDRT of subsquares of size 2x2 as a special case
8: for squareY ← 0 to N/2 − 1 do
9:     for squareX ← 0 to N/2 − 1 do
10:        A ← f^m(squareY · 2 · N + squareX · 2)
11:        B ← f^m(squareY · 2 · N + squareX · 2 + 1)
12:        C ← f^m((squareY · 2 + 1) · N + squareX · 2)
13:        D ← f^m((squareY · 2 + 1) · N + squareX · 2 + 1)
14:        f^{m+1}((SquareY · N/2 + SquareX) · 10 + [0 : 9]) = ...
15:                [A, A + B, A + C, A + D, B, B + C, B + D, C, C + D, D]
16:    end for
17: end for
18:                                                      ▷ The rest of stages starting from stage 2 are computed next
19: for m ← 1 to n − 1 do                                ▷ Will compute stage m+1 from data at stage m
20:    MP1 ← 2^{m+1}
21:    nVertex ← 4 · MP1 − 4
22:    nSquares ← N/MP1
23:    for squareY ← 0 to nSquares − 1 do
24:        for squareX ← 0 to nSquares − 1 do
25:            for indIn ← 0 to nVertex − 1 do
26:                for indOut ← indIn to nVertex − 1 do
27:                    memIndexWrite ← linearIndex(indIn, indOut, squareX, squareY, m + 1, n)
28:                    x0, y0 ← ind2xy(indIn, Mp1)
29:                    x1, y1 ← ind2xy(indOut, Mp1)
30:                    qx0 ← x0 >> m
31:                    qx1 ← x1 >> m
32:                    qy0 ← y0 >> m
33:                    qy0 ← y1 >> m
34:                    slope ← 0
35:                    horizontal ← abs(x1 − x0) > abs(x1 − y0)
36:                    if x1 > x0 then
37:                        dirX ← 0.5
38:                    else
39:                        dirX ← −0.5
40:                    end if
41:                    if y1 > y0 then
42:                        dirY ← 0.5
43:                    else
44:                        dirY ← −0.5
45:                    end if
46:                    incX ← x1 − x0
47:                    incY ← y1 − y0
48:                    if x1 ≠ x0 then
49:                        slope ← incY/incX
50:                    end if
51:                    if ((qx0 = qy0) ∧ (abs(indIn − indOut) > 2 · Mp1 − 2)) ∨ ((qx0 = qy0)∧!(abs(indIn − indOut) >
   2 · Mp1 − 2)) then
52:                        hxmid ← −1
53:                        hymid ← −1
54:                        if (qy0 ≠ qy1) then
55:                            hymid ← Mp1/2 − 0.5
56:                            hxmid ← x0
57:                            if slope ≠ 0 then
58:                                hxmid ← ((hymid − y0)/slope) + x0
59:                            end if
60:                            p1, p2 ← getPointsOnIntersection(hxmid, hymid, dirX, dirY, horizontal, incX, incY)
61:                            val ← val + getVal(x0, y0, p1, n, m, xSquareMp1, ySquareMp1)
62:                        else
63:                            p2 ← (x0, y0)
64:                        end if
65:                        if qx0 ≠ qx1 then
66:                            vxmid ← Mp1/2 − 0.5
67:                            vymid ← y0 + slope · (vxmid − x0)
68:                            if vxmid ≠ hxmid ∧ vymid ≠ hymid then
69:                                p3, p4 ← getPointsOnIntersection(vxmid, vymid, dirX, dirY, horizontal, incX, incY)
70:                                val ← val + getVal(p4, x0, y0, n, m, xSquareMp1, ySquareMp1)
71:                            else
72:                                p4 ← p2
73:                            end if
74:                        else
75:                            p4 ← p2
76:                        end if
```

Algorithm 10 Computes the PDRT of an image (refactorization part2)

```
77:                     else
78:                         vxmid ← −1
79:                         vymid ← −1
80:                         if qx0 ≠ qx1 then
81:                             vxmid ← Mp1/2 − 0.5
82:                             vymid ← y0 + slope · (vxmid − x0)
83:                             p1, p2 ← getPointsOnIntersection(vxmid, vymid, dirX, dirY, horizontal, incX, incY)
84:                             val ← val + getVal(x0, y0, p1, n, m, xSquareMp1, ySquareMp1)
85:                         else
86:                             p2 ← (x0, y0)
87:                         end if
88:                         if qy0 ≠ qy1 then
89:                             hymid ← Mp1/2 − 0.5
90:                             hxmid ← x0
91:                             if slope ≠ 0 then
92:                                 hxmid ← ((hymid − y0)/slope) + x0
93:                             end if
94:                             if (vxmid ≠ hxmid ∧ vymid ≠ hymid then
95:                                 p3, p4 ← getPointsOnIntersection(hxmid, hymid, dirX, dirY, horizontal, incX, incY)
96:                                 val ← val + getVal(p2, p3, n, m, xSquareMp1, ySquareMp1)
97:                             else
98:                                 p4 ← p2
99:                             end if
100:                        else
101:                            p4 ← p2
102:                        end if
103:                        val ← val + getVal(p2, p3, n, m, xSquareMp1, ySquareMp1)
104:                    end if
105:                    fm^{m+1}(indEscritura) ← val
106:                end for
107:            end for
108:        end for
109:    end for
110: end for
```

Having completed these steps, the effort yields a pseudo-code (it actually was written in python) that is prepared for its implementation on Halide. See it in the code 4.15.

Listing 4.15: Halide code for the description of the Perimeter DRT.

```
dir_type m = 0;
Expr Mp1 = 1 << (m + 1);
Expr M = 1 << m;
Expr nSquaresMp1 = VAL_N / Mp1;
Expr nVertexMp1 = 4 * (Mp1 - 1);
Expr sizeSquareMp1 = (nVertexMp1 + 1) * nVertexMp1 / 2;
Expr idx = nn4_to_linearMemIndex(x, y, c, m + 1);
Expr linearsize = get_linear_size(m + 1);
Expr offset = idx % sizeSquareMp1;
Expr base = idx / sizeSquareMp1;
Expr ySquareMp1 = cast<uint16_t>(base / nSquaresMp1);
Expr xSquareMp1 = cast<uint16_t>(base % nSquaresMp1);
Expr A = cast<uint16_t>(in(clamp(ySquareMp1 * 2, 0, VAL_N - 1),
                           clamp(xSquareMp1 * 2, 0, VAL_N - 1), 0));
Expr B = cast<uint16_t>(in(clamp(ySquareMp1 * 2, 0, VAL_N - 1),
                           clamp(xSquareMp1 * 2 + 1, 0, VAL_N - 1), 0));
Expr C = cast<uint16_t>(in(clamp(ySquareMp1 * 2 + 1, 0, VAL_N - 1),
                           clamp(xSquareMp1 * 2 + 1, 0, VAL_N - 1), 0));
Expr D = cast<uint16_t>(in(clamp(ySquareMp1 * 2 + 1, 0, VAL_N - 1),
                           clamp(xSquareMp1 * 2, 0, VAL_N - 1), 0));
// clang-format off
fm[0](x,y,c) = select(idx < 0 || idx >= linearsize, 0,
                      (offset == 0), A,
                      (offset == 1), A + B,
                      (offset == 2), A + C,
                      (offset == 3), A + D,
                      (offset == 4), B,
                      (offset == 5), B + C,
                      (offset == 6), B + D,
                      (offset == 7), C,
                      (offset == 8), C + D,
                      (offset == 9), D,
                      0);
// clang-format on
for (dir_type m = 1; m < MAX_IT - 1; m++) {
   Expr Mp1 = 1 << (m + 1);
   Expr M = 1 << (m);
   Expr nSquaresMp1 = VAL_N / Mp1;
   Expr nVertexMp1 = 4 * (Mp1 - 1);
   Expr sizeSquareMp1 = (nVertexMp1 + 1) * nVertexMp1 / 2;
   Expr indEscritura = nn4_to_linearMemIndex(x, y, c, m + 1);
   Expr linearsize = get_linear_size(m + 1);
   // Calculate xSquareMp1 and ySquareMp1
   Expr offset = indEscritura % sizeSquareMp1;
   Expr xSquareMp1 = (indEscritura / sizeSquareMp1) % nSquaresMp1;
   Expr ySquareMp1 = (indEscritura / sizeSquareMp1) / nSquaresMp1;
   // Calculate indIn and indOut
   Expr a1 = nVertexMp1;
   Expr d = -1.f;
   Expr b = -(2.f * a1 + 1);
   Expr _c = 2.f * offset + 2.f;
   Expr u = sqrt(b * b - 4 * _c);
   Expr _x0 = (-b + u) * 0.5f;
   Expr _x = select(_x0 < nVertexMp1, _x0, (-b - u) * 0.5f);
   _x = select((_x == floor(_x)), _x - 1, _x);
   Expr _in = cast<dir_type>(floor(_x));
   Expr Sin = cast<dir_type>(0.5f * (2.0f * a1 + d * (_in - 1)) * _in);
   Expr indIn = _in;
   Expr indOut = _in + offset - Sin;
   // Find the crossing parts
   Expr x0, x1, y0, y1;
   ind2xy(indIn, Mp1, x0, y0);
   ind2xy(indOut, Mp1, x1, y1);
   Expr qx0 = x0 >> m;
   Expr qx1 = x1 >> m;
   Expr qy0 = y0 >> m;
   Expr qy1 = y1 >> m;
   Expr horizontal = abs(x1 - x0) > abs(y1 - y0);
   Expr dirX = select(x1 > x0, 0.5f, -0.5f);
   Expr dirY = select(y1 > y0, 0.5f, -0.5f);
   Expr incX = cast<float>(x1) - x0;
   Expr incY = cast<float>(y1) - y0;
```

```
    Expr slope = select(x1 != x0, incY / incX, 0);
    // Let us analyze the cases
    Expr COND0 = ((qx0 == qy0) && (abs(indIn - indOut) > 2 * Mp1 - 2)) ||
                 ((qx0 != qy0) && !(abs(indIn - indOut) > 2 * Mp1 - 2));
    Expr COND1 = qy0 != qy1;
    Expr COND2 = qx0 != qx1;
    Expr COND4 = abs(indIn - indOut) == (2 * Mp1 - 2);
    Expr COND5 = (qy0 != qy1) and not((qx0 != qx1) and COND4);
    Expr COND6 = (qx0 != qx1) and not((qy0 != qy1) and COND4);
    // clang-format off
    Expr vxmid = select(COND0 && COND2 ||
                        !COND0 && COND6,
                        Mp1 / 2 - 0.5f,
                        -1.f);
    Expr vymid = select(COND0 && COND2 ||
                        !COND0 && COND6,
                        y0 + slope * (vxmid - x0),
                        -1.f);
    Expr hymid = select((COND0 && COND5 || !COND0 && COND1),
                        Mp1 / 2 - 0.5f,
                        -1.f);
    Expr hxmid = select((COND0 && COND5 || !COND0 && COND1) && slope == 0,
                        cast<float>(x0),
                        (COND0 && COND5 || !COND0 && COND1) && slope != 0,
                        ((hymid - y0) / slope) + x0, -1.f);
    // clang-format on
    Expr COND3 = (vxmid != hxmid) && (vymid != hymid);
    // For cond0
    Expr pxa, pya;
    Expr pxb, pyb;
    Expr pxc, pyc;
    Expr pxd, pyd;
    getPointsOnIntersection(hxmid, hymid, dirX, dirY, horizontal, incX,
                            incY, pxa, pya, pxb, pyb);
    getPointsOnIntersection(vxmid, vymid, dirX, dirY, horizontal, incX,
                            incY, pxc, pyc, pxd, pyd);
    pxa = cast<uint16_t>(pxa);
    pxb = cast<uint16_t>(pxb);
    pxc = cast<uint16_t>(pxc);
    pxd = cast<uint16_t>(pxd);
    pya = cast<uint16_t>(pya);
    pyb = cast<uint16_t>(pyb);
    pyc = cast<uint16_t>(pyc);
    pyd = cast<uint16_t>(pyd);
    x0 = cast<uint16_t>(x0);
    y0 = cast<uint16_t>(y0);
    // clang-format off
    Expr p1x = select(COND0,
                      pxa,
                      pxc);
    Expr p1y = select(COND0,
                      pya,
                      pyc);
    // All cases are covered, so the ", 0"s are just syntax-needed
    Expr p2x = select(COND0 && COND5,
                      pxb,
                      COND0 && !COND5,
                      x0,
                      !COND0 && COND6,
                      pxd,
                      !COND0 && !COND6,
                      x0, 0);
    Expr p2y = select(COND0 && COND5,
                      pyb,
                      COND0 && !COND5,
                      y0,
                      !COND0 && COND6,
                      pyd,
                      !COND0 && !COND6,
                      y0, 0);
    Expr p3x = select(COND0 && COND2 && COND3,
                      pxc,
                      !COND0 && COND1 && COND3,
                      pxa, 0);
    Expr p3y = select(COND0 && COND2 && COND3,
                      pyc,
                      !COND0 && COND1 && COND3,
```

```
                    pya, 0);
   Expr p4x = select(CONDO && COND2 && COND3,
                    pxd,
                    CONDO && COND2 && !COND3,
                    p2x,
                    CONDO && !COND2,
                    p2x,
                    !CONDO && COND1 && COND3,
                    pxb,
                    !CONDO && COND1 && !COND3,
                    p2x,
                    !CONDO && !COND1,
                    p2x, 0);
   Expr p4y = select(CONDO && COND2 && COND3,
                    pyd,
                    CONDO && COND2 && !COND3,
                    p2y,
                    CONDO && !COND2,
                    p2y,
                    !CONDO && COND1 && COND3,
                    pyb,
                    !CONDO && COND1 && !COND3,
                    p2y,
                    !CONDO && !COND1,
                    p2y, 0);
   // clang-format on
   Expr D0 =
      select(CONDO && COND5 || !CONDO && COND6,
             getVal(x0, y0, p1x, p1y, m, xSquareMp1, ySquareMp1), 0);
   Expr D1 =
      select(CONDO && COND2 && COND3 || !CONDO && COND1 && COND3,
             getVal(p2x, p2y, p3x, p3y, m, xSquareMp1, ySquareMp1), 0);
   Expr D2 = getVal(p4x, p4y, x1, y1, m, xSquareMp1, ySquareMp1);
   Expr COND_WRITE = indEscritura >= 0 && indEscritura < linearsize;
   fm[m](x, y, c) = select(
      COND_WRITE && m < 5, cast<uint16_t>(D0 + D1 + D2),
      COND_WRITE && m >= 5, cast<uint16_t>((D0 + D1 + D2) * .5f), 0);
}
out(x, y, c) = cast<uint16_t>(fm[MAX_IT - 2](x, y, c));
```

4.3 Time measurements

The transforms described above have been compared to state-of-the art algorithms that compute a Radon transform. In order to make the comparison fair, our algorithms have been run without any coarse or fine grained parallelization. The Mojette transform has been chosen [72], using the code available in [73]. The Pseudo Polar Radon Transform [57] was chosen as well and its implementation is available in [74].

Both transform can calculate the same number of angles that our transforms do and all test have been measured using the same number of angles ($N * 4$). In figure 4.2 we can see that our algorithms are faster than those of the peers except for the Perimeter DRT which its advantage can only be seen when taking advantage of high degree of parallelism.

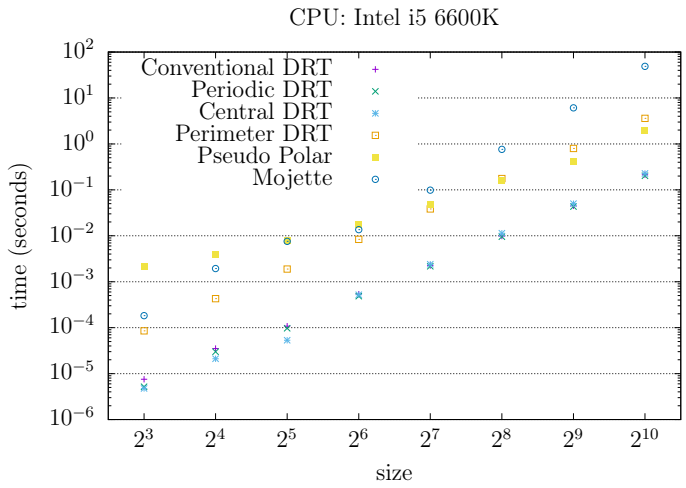


Figure 4.2: Mean of the times of 20 executions of several algorithms. All measurements were made on an Intel i5 6600K and without exploiting any parallelism.

4.3.1 Exploiting parallelism on the CPU

Our algorithms can really benefit from parallelism. In order to do that we will use the Halide implementation, that uses coarse and fine-grained parallelism. In order to measure execution times on CPU platforms we will use the auto-scheduling feature that Halide provides. While it will not give the best scheduling possible, it should give a good one.

The measured times are exposed in 4.3, 4.4 and 4.5. The Intel i5 6600K has four cores @ 3.50 GHz, the Intel i9 9900K has 8 cores running 16 threads @ 3.60 GHz and the Qualcomm Snapdragon 845 has eight cores @ 2.8 GHz. The speed of the Conventional DRT is usually improved by the Central and Periodic versions.

4.3.2 Exploiting parallelism on the GPU

Because all algorithms were programmed in a way that was compatible with the OpenGL GLSL kernels, that is what we are going to use for these tests. Except for the Perimeter transform which has too many branches and therefore is not easily implemented on GPU and because of the branches, the parallelism is not going to be fully exploited.

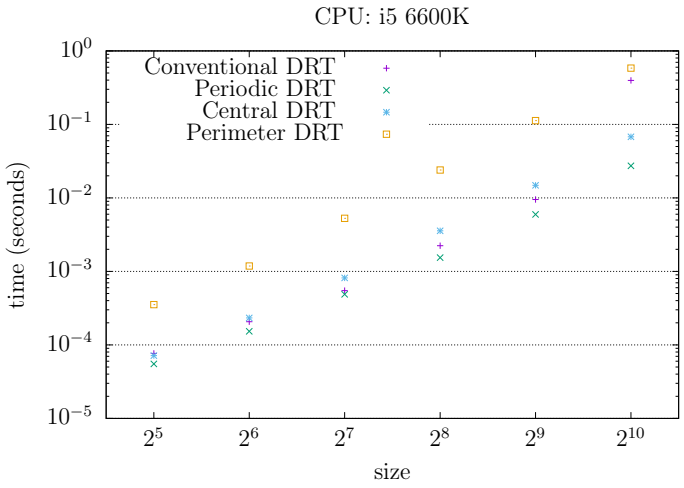


Figure 4.3: Mean of the times of 20 executions of our algorithms on an Intel i5 6600K using the auto-scheduling feature to exploit parallelism.

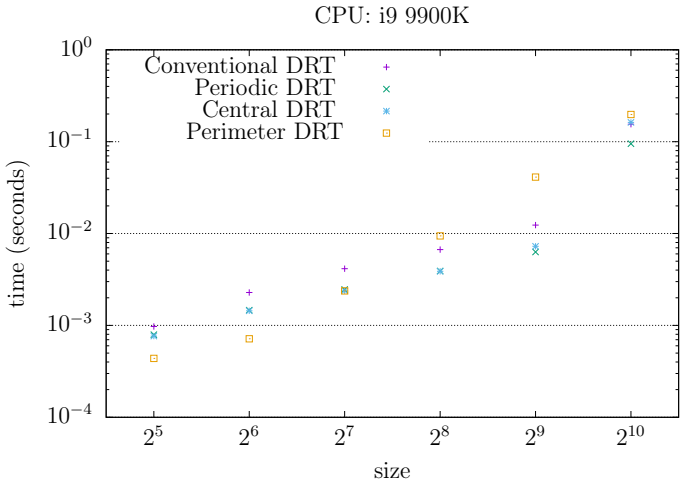


Figure 4.4: Mean of the times of 20 executions of our algorithms on an Intel i9 9900K using the auto-scheduling feature to exploit parallelism.

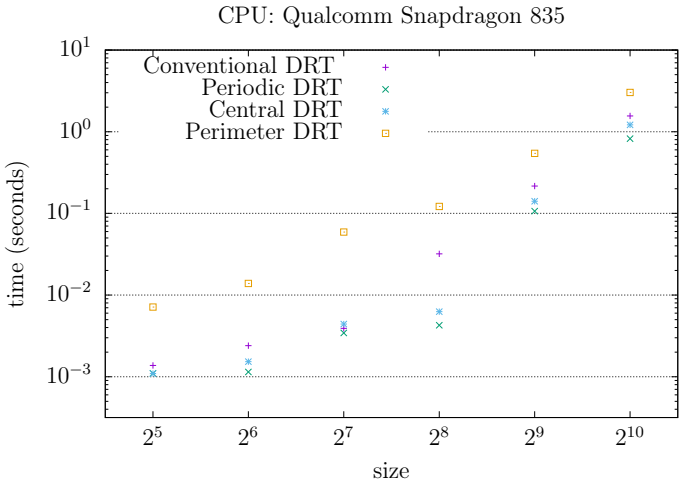


Figure 4.5: Mean of the times of 20 executions of our algorithms on Qualcomm Snapdragon 845, using the auto-scheduling feature to exploit parallelism.

We have tested our algorithms on two modern GPUs. One PC platform GPU: the NVIDIA GeForce 1070 and one mobile platform GPU: the Adreno 540 present on the Qualcomm Snapdragon. These machines are very different. The GeForce 1070 has 1920 CUDA cores @ 1506 MHz while the Adreno 540 has 256 pipelines @ 710 MHz. The GeForce 1070 has a theoretic max of 6.463 GFlops and the Adreno 540 only 567 GFlops (both for FP32).

By taking a look at figures 4.6 and 4.7, we can see that parallelism is exploited, obtaining very good execution times. Given that the Adreno has 256 pipelines (2^8), it behaves very well for sizes of $N \leq 2^8$.

4.4 DRT applications

4.4.1 SFF - Curvelet with Periodic DRT

The shape-from-focus algorithm was introduced in chapter 2.

In this section we will try to demonstrate that Curvelet-based, –more specifically, an operator based on the approximate Discrete Curvelet Transform, that uses discrete Radon transform underneath–, can successfully replace and beat Wavelet-based operators.

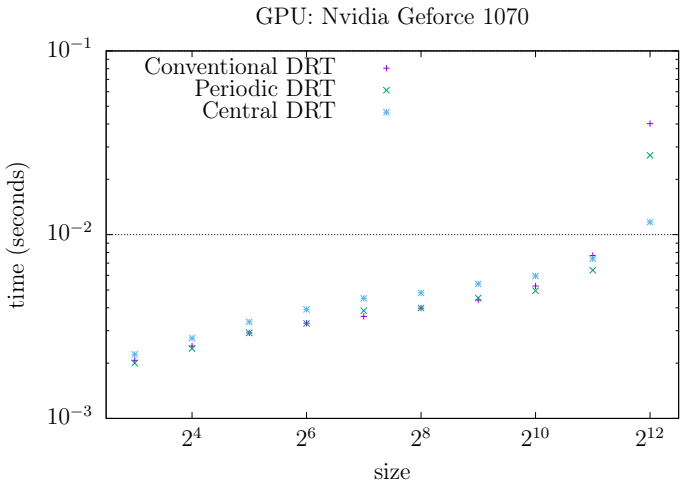


Figure 4.6: Mean of the times of 20 executions of our algorithms on a NVIDIA GeForce 1070, using GLSL kernels.

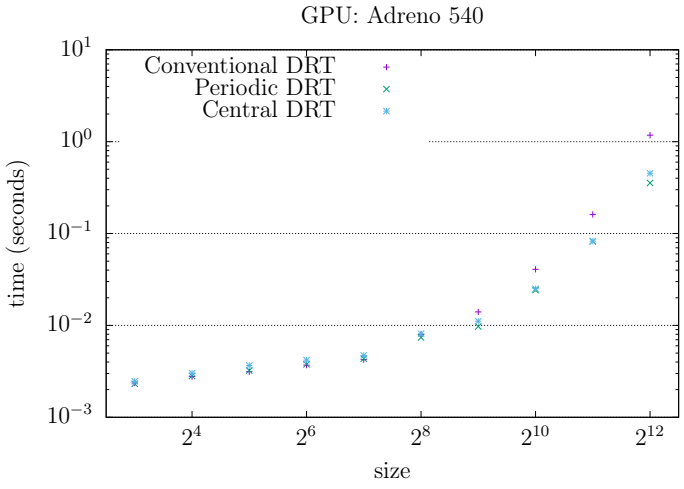


Figure 4.7: Mean of the times of 20 executions of our algorithms on Qualcomm Snapdragon 845 Adreno 540, using GLSL kernels.

To measure focus within a Wavelet context previous works have managed to identify in which localities of the original domain the transformed coefficients predict that there is information that could not be there unless the object is inside the depth of field. In other words, to locate focused features, the coefficients that affect that locality are brought together, then their energy is compared at different bands, and attending to some criteria the zone covered under those relative Wavelet coefficients is determined to be in or out of focus. But Wavelet-based methods do not explicitly perform a direct transform, then filter, then do inverse transform and compare in image domain, as it is time consuming. In the case of discrete decimated Wavelets with dyadic partition, this relationship between transformed coefficients and original values can be trivially established, only that the focus estimation is normally not given per pixel, but for aggregates of 2×2 pixels.

On the other hand, if it is possible to detect focus by comparison between original and filtered versions of the images, then there is a chance that the same discrimination can be done without explicit direct transform + filter + inverse transform, but implicitly in the transformed domain.

At this stage in our research we have not yet established this relationship in part because the discrete Radon transform in use has the intricate shape shown in figure 4.8.

An example of discrimination directly in transformed domain and another of discrimination through filtering and inversion is shown in middle and bottom row, respectively, in figure 4.9.

For our Curvelet based operator the aim, therefore, is to design a filter in transformed domain, that separates coarse from texture layer. Several strategies were tested and we found that the best performing one is to attenuate only the highest frequency Wavelet band inside Ridgelet description of blocks. By attenuating linearly those coefficients we are able to detect even zones of barely noticeable defocus.

Several scenes, synthetic and real, have been processed using the described procedure.

Our first synthetic test consists of creating a synthetic cone, covered with a radial texture, and simulating defocus to create a focal stack. Then this focal stack has been processed with the SFF method based on the Wavelet operator described in Xie *et. al.*[27] and the Curvelet one proposed by us. Defocusing has been simulated following [75]. In this case the performance of Wavelets and Curvelets based operators is on par, and using Curvelets does not suppose a clear advantage.

Figure 4.8: A 256×256 gray image, shown on the lower left corner, and its approximate Discrete Radon Transform. Note the increase in size: $6 \cdot N^2$ non zero coefficients, in a 3×4 shape the original size.

In the figure 4.11 the same test is repeated using a similar scene, but with different shape that includes a curved discontinuity. The robustness of the Curvelet-based operator outperforms that of Wavelets around the discontinuity.

In the figure 4.12, a real scene has been captured using a conventional pocket camera and a depth map has been computed using SFF aDCT-based approach. The maxima are represented only when a threshold is exceeded, attending to the lack of textures in several zones. The result is almost free of false positives, and the borders of objects are well defined even when pixel support is a simple 5×5 window.

Figure 4.13 shows a test conducted with a focal stack of a scene that has been captured using a mirror-less camera with a macro lens. The synthetic extended aperture image shown as result, all-in-focus image, demonstrates that our SFF operator can be used with almost no artifacts even in the presence of occlusions and borders. In fact, the only noticeable error is due

Figure 4.9: A $256x256$ image in gray tones on the top left, alongside its decomposition in 3 sub-bands using Wavelets. Middle row, the inversion of exclusively middle and higher Wavelet sub-bands coefficients, and its ratio. Bottom row, the original image and a version of itself where the weakest coefficients on Wavelet domain has been thresholded. On the right it is shown the difference.

to the lack of texture in the background.

4.4.2 Bar-code detection with Central DRT

Another usage that collaterally appeared while researching alternatives for a faster discrete Radon transform belongs to a –apparently unrelated– field, that of detecting bar-codes in an image. It is of course not that dissimilar as Radon transform is the tool to use when analyzing an image in search of linear structures, as those that constitute marks and spaces in linear, onedimensional, 'zebra-like' bar-codes.

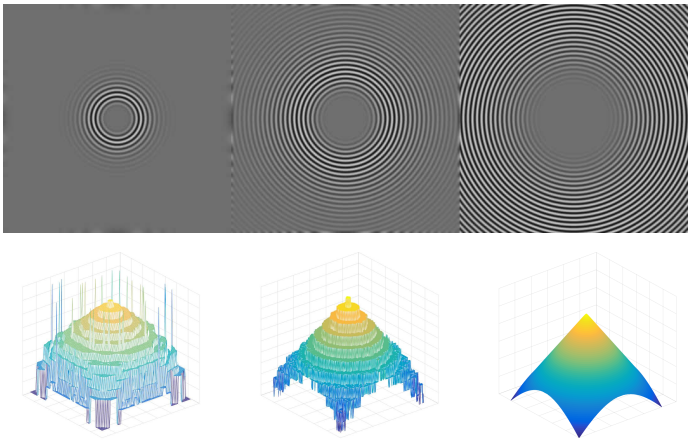


Figure 4.10: On the top, a selection of three images of a focal stack of a synthetic cone is shown. On the bottom: the recovered depth-map using a Wavelet-based operator on the left, and using Curvelets at the center. Ground truth is shown on the right.

Background

Typically, 1D bar-codes, defined in the ISO15417, ISO15420, ISO15424, ISO15426-1, and ISO16390 standards [76, 77, 78, 79, 80], are read by laser scanners, which require pointing directly at the bar-code (*data carrier*, in the terminology of the standards) with an appropriate orientation and distance between the scanner and the code.

Some bar-code readers implemented on mobile devices emulate this restrictive behavior of laser scanners, and only decode the codes if a raster line or region of interest on the camera is totally aligned with the bar-code. That is, they decode but do not locate automatically. Automatically localization of bar-codes means being able to detect them in an image even when they may appear rotated, translated and/or scaled.

There are methods for performing this automatic localization on mobile devices using different computer vision techniques, but many impose tight requirements regarding the position between the camera and the bar-code, in order to alleviate the computational cost of localization.

Because that cost will be normally much more higher than decoding itself. It can be achieved by filtering, detecting and/or clustering image characteristics that give cues of the presence of bar-codes: such as concentration of

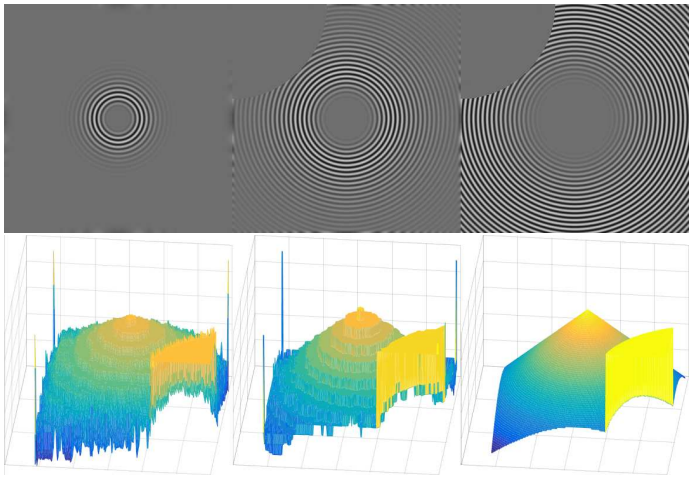


Figure 4.11: Same order of illustrations than in fig. 4.10, but now the synthesized shape includes a curved discontinuity.

corners, relative magnitude of gradients in two orthogonal directions, preeminence of bi-level features, ...

Detecting those characteristics normally impose tight restrictions on what inclination the bar-code and the camera can have or otherwise a bar-code can pass inadvertent, because a small number of directional filters is used to relieve computation.

Some authors have preconized the analysis of Radon transform –or its equivalent Hough transform– of the pictures, instead of the pictures themselves to locate the bar-code, as Radon transform achieves invariance to angle. But until now it was not possible to apply it in mobile devices because of the computational complexity of the transform itself [81, 82].

DRT applied to bar-code reading

The two upper sub-figures in figure 4.14 show the aspect of the Radon transform of an image composed of more or less thick dots and line segments. The highest value of the transform is reached in a dot whose coordinates correspond to the angle and displacement of the longest line. Meanwhile, the dots in image create a sinusoidal trace in the transform, which is why this output is also known as the "sinogram". The angles occupy the horizontal axis and the displacements occupy the vertical axis.

Although the DRT computation is faster than the computation of Radon

Figure 4.12: Shape-from-focus using aDCT focus estimator applied to a real scene. On the left are shown the four images that constitute the focal stack. On the right, the obtained sparse depth map.

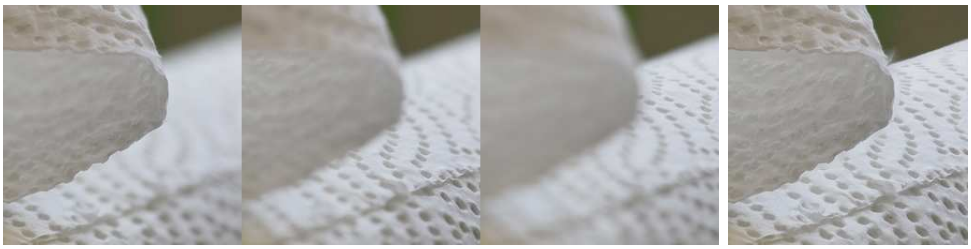


Figure 4.13: Real scene that contains discontinuities. Three images from a focal stack of ten are shown on the left and an all-in-focus image is shown on the right. The all-in-focus image was generated using the shape that was recovered using our aDCT method.

transform using Fourier Slice theorem and FFTs, it still devotes half of the computations to values that actually contain little "energy". Since in any projection the displacements around the center of the image will receive the most contributions, while those displacements around the corners receive the least.

Only when projections are in multiples of 90° the sums are different than zero for exactly N displacements, but the odd multiples of 45° project on $2N-1$ displacements, the centers receiving contributions from N pixels and the extremes adding decreasingly less pixels.

Proposed improvements to DRT as bar-code detector

The improvements we are proposing focus on alleviating those deficits that we have identified:

- To optimize the number of computed data: calculate only the N central displacements for each projection, reducing displacement computations from $2N - 1$ to N for each intermediate stage, thus also lowering the algorithm memory requirements.
- To optimize the computation of those data: by exposing the algorithm between data stripes, instead of between individual data, to better exploit the parallelism of the problem.
- To eliminate conditional accesses outside memory limits to make better use of parallelism.
- To make sure that the central point of the image always projects in the central displacement for any angle in the transform.

Therefore, and because we do not care for the inversion in this application, we use the Central DRT defined in 3.3.1.

Bar-code reading application

With a smart mobile device and making use of the modified DRT algorithm previously defined, it is possible to compute in milliseconds the Radon transform of the gradients of the image being captured.

When locating a bar-code in an image, and therefore be able to decode it by analyzing the intensity profile of one or several lines that cross through the center of the code in the location and orientation estimated by the automatic locator, we propose to analyze the central Radon transform, that computing just the N central displacements, applied to the image gradient. As exemplified in the figures 4.15 and 4.16.

The presence of a bar-code in an image generates in the Central DRT transform of its gradients a rhombus-shaped zone, which stands out as having a greater value at the same time that a relative variance smaller than the rest of the zones.

Normally, a simple local detector based on these two characteristics - mean value and relative variance - is sufficient, especially with isolated bar-codes such as those shown in the figures 4.15 and 4.16. When areas with

high frequencies are present next to the bar-code, it is convenient that the rhombus be located by a neural network trained for this purpose. In any case, the rhombus boundary vertexes, shown with white stars in the transforms on the left of the figure, are converted in the case of top and bottom vertexes (3 pointed stars) into solid lines superimposed on the original images that reveal the upper and lower limits of the code in image space, and on the other hand, the left and right vertexes (4 pointed stars) become the diagonal dashed lines that cross the code cutting approximately through its center. With this information, the intensity profile would be analyzed in one or more lines close to this center and parallel to the horizontal limits.

To convert from Radon d and s parameter space to lines in image space, basically we apply that, for first quadrant, the line corresponding to d and s is: $y = d + s/(N-1) \cdot x$, and similarly for the other quadrants.

Step by step bar-code localization

Our proposed method for automatically localize bar-codes present within an image consists of the following steps:

1. Capture an image with the smart mobile device camera, optionally a square region in the center or an image with any aspect ratio

2. If necessary resize the image so that it becomes a square image with size 512x512 or 1024x1024 (depending on device computation capacity)

3. Compute the module of the gradients of the luminance channel of the square image

4. Compute the four central DRT quadrants of the gradient. Attach a vertically and horizontally mirrored version of the first quadrant to the right of the 4th quadrant

5. Locate the most prominent rhombus in the 5 quadrants version of the central DRT, based on its higher mean and smaller relative variance. To do this we propose two alternatives:

    - Analyze locally the mean and variance of non-overlapping patches of the 5 quadrants version of the central DRT. Order the patches by that measure and take only the greater ones, try to group them if adjacent and extract the vertexes of the resulting rhombus

- Pass the 5 quadrant version of the central DRT to a Neural Network which has been previously trained to recognize rhombuses vertexes in thousands of exemplary pairs of images containing bar-codes as input and location of rhombuses in their gradient's DRTN as outputs

6. Average the s index of top and bottom vertexes of the rhombus: that is the slope of the bar-code in the square image space

7. Take the difference in d axis between top and bottom vertexes: that is an indicator of the height of the bar-code in the square image space

8. Convert the coordinates of left and right vertexes of the rhombus into line parameters on square image space. Calculate the crossing point between those two lines: that is the approximate center of the bar-code in the square image space

9. If the image from the camera has been resized and/or stretched in the optional steps 1 and 2, then convert all the previous indicators from steps 7,8 and 9 that are currently referred to square image coordinates into original image coordinates

10. The height, slope and center coordinates of the bar-code in original image coordinates can be used by a bar-code decoder that analyzes a single-line through the center of the bar-code; or a mixture of several lines that traverse longitudinally the bar-code around its center

Algorithm 11 Computes the PDRT of an image (refactorization2 part1)

Input: image f of size $N \times N$, with N a power of two
Output: $PDRT$ of input

$N \leftarrow$ **sizeof**(f)
$n \leftarrow$ **log2**(N)
$maxSize \leftarrow (4 \cdot N - 3) \cdot (4 \cdot N - 4)/2$ ▷ Last stage will hold the maximum size of PDRT
$f^m \leftarrow$ **zeros**$(1, maxSize)$ ▷ We allocate two buffers of that maximum size
$f^{m+1} \leftarrow$ **zeros**$(1, maxSize)$ ▷ One buffer for current stage and one for next stage, m and $m+1$
$f^m(0 : N \cdot N - 1) \leftarrow f$ ▷ We copy the input to partial transform at current stage
▷ We compute PDRT of subsquares of size 2x2 as a special case
for $x \leftarrow 0$ **to** $size_{m+1}$ **do**
  for $y \leftarrow 0$ **to** $size_{m+1}$ **do**
    for $c \leftarrow 0$ **to** 4 **do**
      $linearWriteDir \leftarrow nn4ToLinearMemIndex(x, y, c, m+1)$
      $offset \leftarrow linearWriteDir\%sizeSquareMp1$
      $squareX \leftarrow linearWriteDir/sizeSquare_1)\%nSquaresMp1_1$
      $squareY \leftarrow linearWriteDir/sizeSquare_1)/nSquaresMp1_1$
      $A \leftarrow fm^0(squareY \cdot 2 \cdot N + squareX \cdot 2)$
      $B \leftarrow fm^0(squareY \cdot 2 \cdot N + squareX \cdot 2 + 1)$
      $C \leftarrow fm^0((squareY \cdot 2 + 1) \cdot N + squareX \cdot 2)$
      $D \leftarrow fm^0((squareY \cdot 2 + 1) \cdot N + squareX \cdot 2 + 1)$
      if $offset = 0$ **then**
        $val \leftarrow A$
      else if $offset = 1$ **then**
        $val \leftarrow A + B$
      else if $offset = 2$ **then**
        $val \leftarrow A + C$
      else if $offset = 3$ **then**
        $val \leftarrow A + D$
      else if $offset = 4$ **then**
        $val \leftarrow B$
      else if $offset = 6$ **then**
        $val \leftarrow B + C$
      else if $offset = 7$ **then**
        $val \leftarrow C$
      else if $offset = 8$ **then**
        $val \leftarrow C + D$
      else if $offset = 9$ **then**
        $val \leftarrow D$
      end if
      $fm^1(x, y, c) \leftarrow val$
    end for
  end for
end for
▷ The rest of stages starting from stage 2 are computed next
for $m \leftarrow 1$ **to** $n - 1$ **do** ▷ Will compute stage m+1 from data at stage m
  $MP1 \leftarrow 2^{m+1}$
  $nVertex \leftarrow 4 \cdot MP1 - 4$
  $nSquares \leftarrow N/MP1$
  for $x \leftarrow 0$ **to** $size_{m+1}$ **do**
    for $y \leftarrow 0$ **to** $size_{m+1}$ **do**
      for $c \leftarrow 0$ **to** 4 **do**
        $val \leftarrow 0$
        $memIndexWrite \leftarrow$ **nn4ToLinearMemIndex**$(x, y, c, m+1)$
        if $memIndexWrite \geq sizeSquareMp1 \cdot nSquaresMp1 \cdot nSquaresMp1$ **then**
          continue
        end if
        $offset \leftarrow indEscritura\%sizeSquareMp1$
        $xSquareMp1 \leftarrow (indEscritura/sizeSquareMp1)\%nSquaresMp1$
        $ySquareMp1 \leftarrow (indEscritura/sizeSquareMp1)/nSquaresMp1$
        $inIn, indOut \leftarrow linear2inOut(offset, m+1)$
        $x0, y0 \leftarrow$ **ind2xy**$(indIn, Mp1)$
        $x1, y1 \leftarrow$ **ind2xy**$(indOut, Mp1)$
        $qx0 \leftarrow x0 >> m$
        $qx1 \leftarrow x1 >> m$
        $qy0 \leftarrow y0 >> m$
        $qy0 \leftarrow y1 >> m$
        $slope \leftarrow 0$
        $horizontal \leftarrow$ **abs**$(x1 - x0) >$ **abs**$(x1 - y0)$
        if $x1 > x0$ **then**
          $dirX \leftarrow 0.5$
        else
          $dirX \leftarrow -0.5$
        end if
        if $y1 > y0$ **then**
          $dirY \leftarrow 0.5$
        else
          $dirY \leftarrow -0.5$
        end if
        $incX \leftarrow x1 - x0$
        $incY \leftarrow y1 - y0$
        if $x1 \neq x0$ **then**
          $slope \leftarrow incY/incX$
        end if

Algorithm 12 Computes the PDRT of an image (refactorization2 part2)

```
82:                 if ((qx0 = qy0) ∧ (abs(indIn − indOut) > 2 · Mp1 − 2)) ∨ ((qx0 = qy0)∧!(abs(indIn − indOut) >
    2 · Mp1 − 2)) then
83:                     hxmid ← −1
84:                     hymid ← −1
85:                     if (qy0 ≠ qy1) then
86:                         hymid ← Mp1/2 − 0.5
87:                         hxmid ← x0
88:                         if slope ≠ 0 then
89:                             hxmid ← ((hymid − y0)/slope) + x0
90:                         end if
91:                         p1, p2 ← getPointsOnIntersection(hxmid, hymid, dirX, dirY, horizontal, incX, incY)
92:                         val ← val + getVal(x0, y0, p1, n, m, xSquareMp1, ySquareMp1)
93:                     else
94:                         p2 ← (x0, y0)
95:                     end if
96:                     if qx0 ≠ qx1 then
97:                         vxmid ← Mp1/2 − 0.5
98:                         vymid ← y0 + slope · (vxmid − x0)
99:                         if vxmid ≠ hxmid ∧ vymid ≠ hymid then
100:                            p3, p4 ← getPointsOnIntersection(vxmid, vymid, dirX, dirY, horizontal, incX, incY)
101:                            val ← val + getVal(p4, x0, y0, n, m, xSquareMp1, ySquareMp1)
102:                        else
103:                            p4 ← p2
104:                        end if
105:                    else
106:                        p4 ← p2
107:                    end if
108:                else
109:                    vxmid ← −1
110:                    vymid ← −1
111:                    if qx0 ≠ qx1 then
112:                        vxmid ← Mp1/2 − 0.5
113:                        vymid ← y0 + slope · (vxmid − x0)
114:                        p1, p2 ← getPointsOnIntersection(vxmid, vymid, dirX, dirY, horizontal, incX, incY)
115:                        val ← val + getVal(x0, y0, p1, n, m, xSquareMp1, ySquareMp1)
116:                    else
117:                        p2 ← (x0, y0)
118:                    end if
119:                    if qy0 ≠ qy1 then
120:                        hymid ← Mp1/2 − 0.5
121:                        hxmid ← x0
122:                        if slope ≠ 0 then
123:                            hxmid ← ((hymid − y0)/slope) + x0
124:                        end if
125:                        if vxmid ≠ hxmid ∧ vymid ≠ hymid then
126:                            p3, p4 ← getPointsOnIntersection(hxmid, hymid, dirX, dirY, horizontal, incX, incY)
127:                            val ← val + getVal(p2, p3, n, m, xSquareMp1, ySquareMp1)
128:                        else
129:                            p4 ← p2
130:                        end if
131:                    else
132:                        p4 ← p2
133:                    end if
134:                    val ← val + getVal(p2, p3, n, m, xSquareMp1, ySquareMp1)
135:                end if
136:                fmm + 1(x, y, c) ← val
137:            end for
138:        end for
139:    end for
140: end for
```

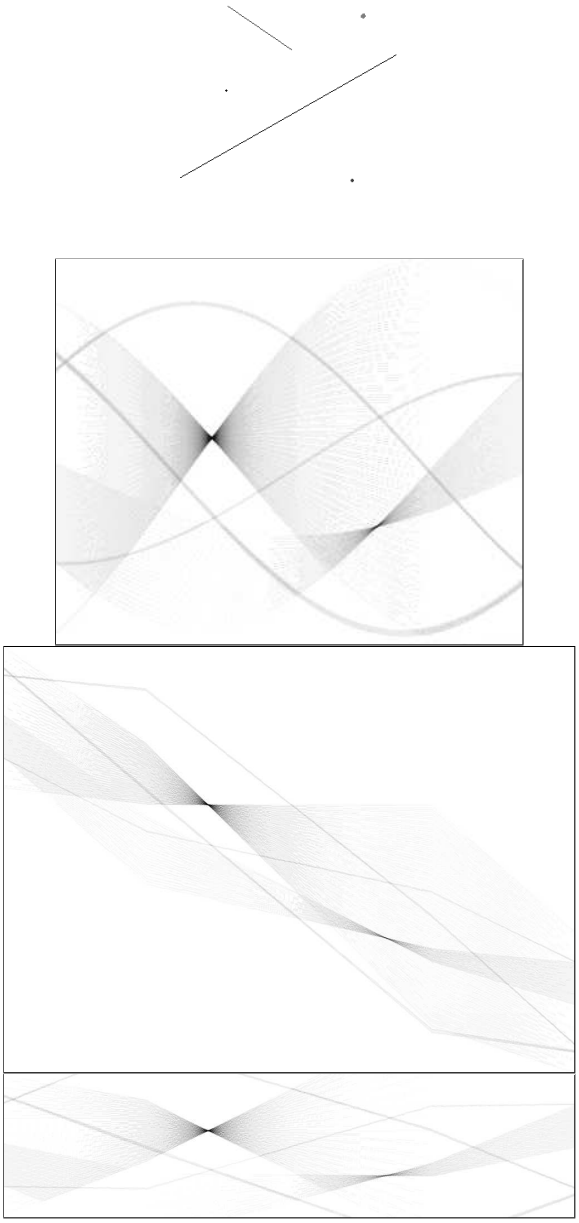


Figure 4.14: From top to bottom: An image composed of points and line segments; its continuous transform; classical DRT transform and proposed DRT transform. Black color indicate high values and white null value.

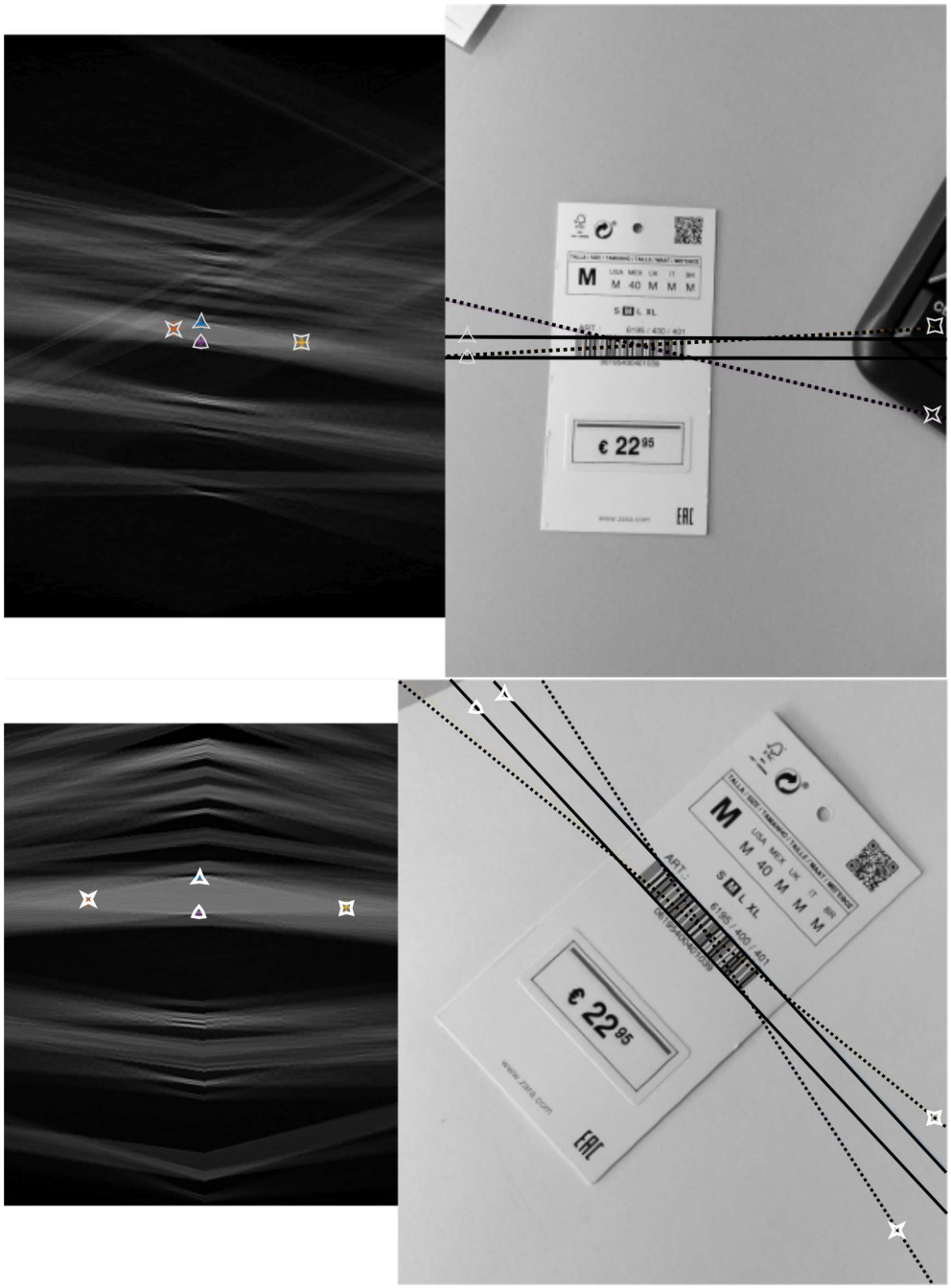


Figure 4.15: Examples of automatic bar-code localization by analyzing the Radon DRTN transform of the gradient. Bar-codes are transformed into rhombuses with higher energy and less variance than the rest of the transform. The top and bottom vertex points of the rhombus become the lines that horizontally limit the code while the left and right vertexes become diagonal lines whose cut indicates approximately the center of the code. The rhomboid shape in the transform does not vary with the bar-code rotation, it only moves on the s axis.

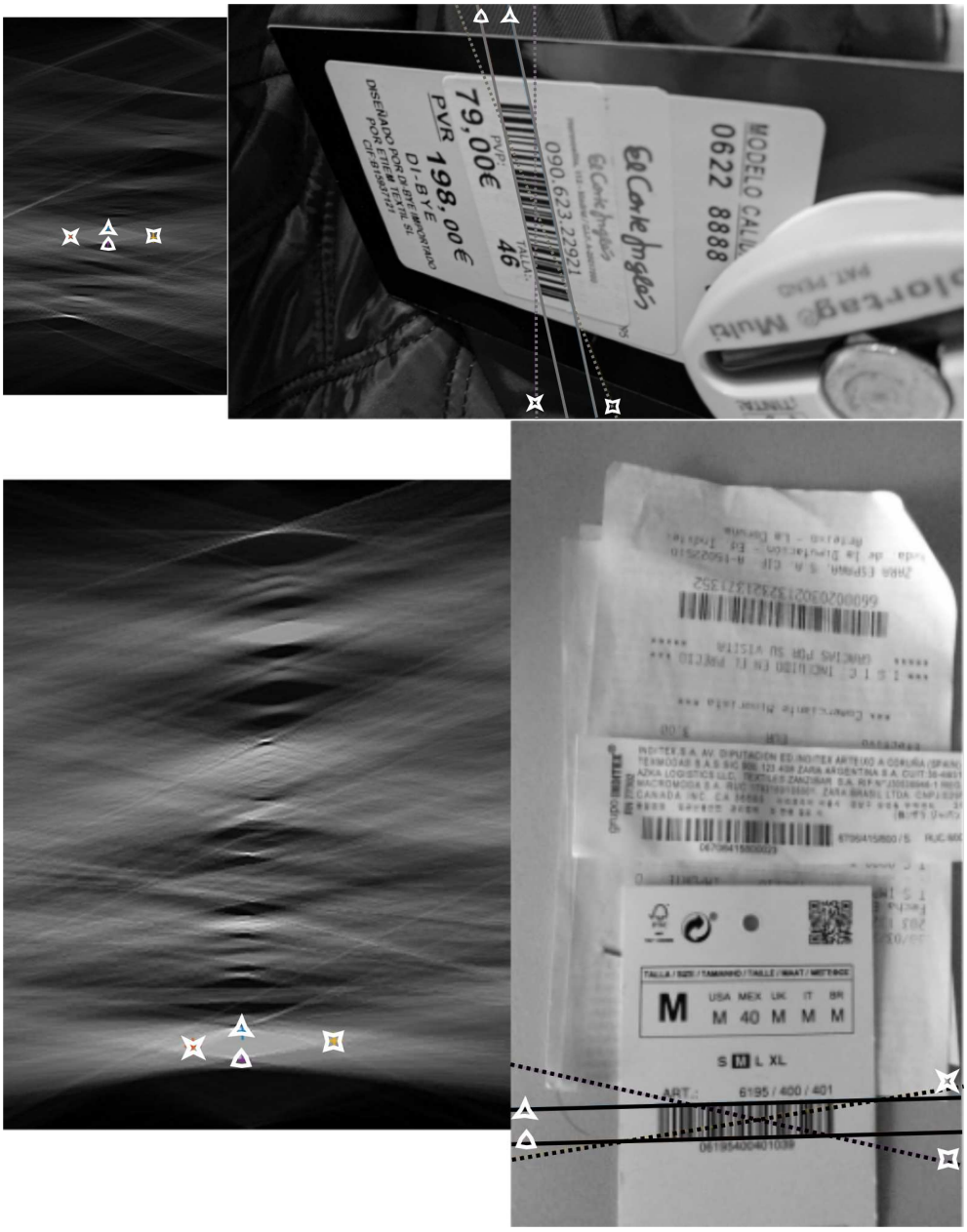


Figure 4.16: More examples of automatic bar-code localization by analyzing the Radon DRTN transform of the gradient.

Figure 4.17: Steps 1-4 of the method described in section 4.4.2

Figure 4.18: Steps 5-10 of the method described in section 4.4.2

Chapter 5

Conclusions

In this work, we introduced the approximate Discrete Curvelet Transform, aDCT, which is based on the approximate Discrete Radon Transform, aDRT. The new transform inherits the advantageous properties of aDRT of being fast and having a well conditioned inversion. It allows to avoid Fourier transforms, so that it can be implemented using fixed integer arithmetic, without requiring complex numbers, trigonometric functions, nor even multiplications.

To our knowledge is the first DCT that can be accomplished exclusively by integer arithmetics whose inverse is exact and well conditioned, so that reconstruction after filtering, including thresholding techniques, is possible.

Another goal of this work is to evaluate the applicability of this new transform as a focus estimator in order to use it for the shape-from-focus problem. Our preliminary results give support to our assumptions:

- The aDCT can be used as a focus estimator at least as robust as those Wavelet-based.
- The aDCT will be advantageous in situations where Wavelet based operators behave worse: in presence of discontinuities in depth.

Curvelets are in disadvantage compared to Wavelets and other focus measurement operators due to its more intensive computationally load. We have studied how a parallelized implementation using multithreading and SIMD lowers the gap between execution times in order to make Curvelets a real alternative to less sophisticated methods even when being executed in handheld devices. The selection of aDRT as basis Radon method makes this path viable.

But even then, this study constitutes a first approach to shape-from-focus using the aDCT. While the results of this work are promising, it is just a starting point and more work will follow along this line. The focus operator has to be further refined in order to provide the best results and we need to further analyze its advantages and limitations.

Some of the relevant aspects that should be addressed to enhance this first approach are:

- To deepen in which is the best strategy to detect focus in Curvelet transform.
- That study should probably include establishing the relation between coefficients at different scales, that may overlap completely or partially in the original domain.
- To relate energy in Curvelet domain with localities in the original domain so that focus detection can be done directly in the transformed domain.
- To study the applicability of shrinkage methods in order to make a better filtering through different bands and attenuate the effect of noise.

A disadvantage of shape-from-focus method is how it handles objects in movement. Given that the focal stack is composed of several shots and these happen at different times, the movement of the subjects and the camera shake hurts the precision of the algorithm. There are two solutions to this problem. The first one is using a robust registration algorithm that ensures that the positions in the x, y coordinates of the focal stack have pixels pertaining to the same object through the z axis. The second one is to use a camera that focus really fast. This kind of camera exist and uses a liquid lens in order to focus. By using a liquid lens that is capable of capturing a whole focal stack in 0.03 seconds real-time is achieved and therefore video using is shape from focus is possible.

Bibliography

[1] http://vision.middlebury.edu/stereo/

[2] Samuel W Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T Barron, Florian Kainz, Jiawen Chen and Marc Levoy. "Burst photography for high dynamic range and low-light imaging on mobile cameras". *ACM Transactions on Graphics (TOG)*, **35**, 6 (2016) 192.

[3] "VLP-16". http://velodynelidar.com/vlp-16.html

[4] "iPhone X". https://www.apple.com/lae/iphone-x/

[5] "Spectra Press Kit" (2017). https://www.qualcomm.com/news/media-center/press-kits/spectra

[6] Richard Szeliski. "Computer vision: algorithms and applications". Springer Science & Business Media (2010).

[7] Zhengfa Liang, Yiliu Feng, Yulan Guo Hengzhu Liu Wei Chen and Linbo Qiao Li Zhou Jianfeng Zhang. "Learning for Disparity Estimation through Feature Constancy". In "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition", pages 2811–2820 (2018).

[8] Tatsunori Taniai, Yasuyuki Matsushita, Yoichi Sato and Takeshi Naemura. "Continuous 3D label stereo matching using local expansion moves". *arXiv preprint arXiv:1603.08328*.

[9] Lincheng Li, Shunli Zhang, Xin Yu and Li Zhang. "Pmsc: Patchmatch-based superpixel cut for accurate stereo matching". *IEEE Transactions on Circuits and Systems for Video Technology*.

[10] Anat Levin, Rob Fergus, Frédo Durand and William T Freeman. "Image and depth from a conventional camera with a coded aperture". *ACM transactions on graphics (TOG)*, **26**, 3 (2007) 70.

[11] Edward H Adelson, James R Bergen et al. "The plenoptic function and the elements of early vision".

[12] JM Rodríguez-Ramos, B Femenía Castellá, F Pérez Nava and S Fumero. "Wavefront and distance measurement using the CAFADIS camera". In "Adaptive Optics Systems", volume 7015, page 70155Q. International Society for Optics and Photonics (2008).

[13] Ren Ng, Marc Levoy, Mathieu Brédif, Gene Duval, Mark Horowitz and Pat Hanrahan. "Light Field Photography with a Hand-Held Plenoptic Camera". Technical report (2005). `http://graphics.stanford.edu/papers/lfcamera/`

[14] Óscar Gómez-Cárdenes, José G. Marichal-Hernández, Fernando Rosa, Jonas P. Lüke, Juan J. Fernández-Valdivia and José M. Rodríguez-Ramos. "Refocusing from a plenoptic camera within seconds on a mobile phone". volume 9139, pages 9139 – 9139 – 10 (2014). `https://doi.org/10.1117/12.2053186`

[15] J.P. Lüke, F. Rosa, J.G. Marichal-Hernández, J.C. Sanluís, C. Dominguez Conde and J.M. Rodríguez-Ramos. "Depth From Light Fields Analyzing 4D Local Structure". *Display Technology, Journal of*, **11**, 11 (2015) 900–907. ISSN 1551-319X.

[16] J. P. Lüke, F. Rosa, J. G. Marichal-Hernández, J. C. Sanluís, C. Domínguez Conde and J. M. Rodríguez-Ramos. "Depth From Light Fields Analyzing 4D Local Structure". *Journal of Display Technology*, **11**, 11 (2015) 900–907. ISSN 1551-319X.

[17] Shree K Nayar and Yasuo Nakagawa. "Shape from focus". *IEEE Transactions on Pattern analysis and machine intelligence*, **16**, 8 (1994) 824–831.

[18] Jianping Shi, Xin Tao, Li Xu and Jiaya Jia. "Break ames room illusion: depth from general single images". *ACM Transactions on Graphics (TOG)*, **34**, 6 (2015) 225.

[19] Zhengyou Zhang. "A flexible new technique for camera calibration". *IEEE Transactions on pattern analysis and machine intelligence*, **22**.

[20] Olivier D Faugeras, Q-T Luong and Stephen J Maybank. "Camera self-calibration: Theory and experiments". In "European conference on computer vision", pages 321–334. Springer (1992).

[21] "IMX377 Image Sensor for Camera". `https://www.sony-semicon.co.jp/products_en/IS/sensor2/products/imx377.html`

[22] Lisa Gottesfeld Brown. "A survey of image registration techniques". *ACM computing surveys (CSUR)*, **24**, 4 (1992) 325–376.

[23] Changyin Zhou, Daniel Miau and Shree K Nayar. "Focal sweep camera for space-time refocusing". *Technical Report, Department of Computer Science*.

[24] Said Pertuz, Domenec Puig and Miguel Angel Garcia. "Analysis of focus measure operators for shape-from-focus". *Pattern Recognition*, **46**, 5 (2013) 1415–1432.

[25] Jonathan A Marshall, Christina A Burbeck, Dan Ariely, Jannick P Rolland and Kevin E Martin. "Occlusion edge blur: a cue to relative visual depth". *JOSA A*, **13**, 4 (1996) 681–688.

[26] Ge Yang and Bradley J Nelson. "Wavelet-based autofocusing and unsupervised segmentation of microscopic images". In "Intelligent Robots and Systems, 2002.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on", volume 3, pages 2143–2148. IEEE (2003).

[27] Hui Xie, Weibin Rong and Lining Sun. "Wavelet-based focus measure and 3-d surface reconstruction method for microscopy images". In "Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on", pages 229–234. IEEE (2006).

[28] Kaiming He, Jian Sun and Xiaoou Tang. "Guided image filtering". In "European conference on computer vision", pages 1–14. Springer (2010).

[29] Asmaa Hosni, Michael Bleyer, Margrit Gelautz and Christoph Rhemann. "Local stereo matching using geodesic support weights." In "ICIP", volume 9, pages 2093–2096. Citeseer (2009).

[30] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt and Joan M Ogden. "Pyramid methods in image processing". *RCA engineer*, **29**, 6 (1984) 33–41.

[31] Anat Levin, Dani Lischinski and Yair Weiss. "Colorization using optimization". In "ACM transactions on graphics (tog)", volume 23, pages 689–694. ACM (2004).

[32] Kaiming He and Jian Sun. "Fast Guided Filter". *CoRR*, **abs/1505.00996**. http://arxiv.org/abs/1505.00996

[33] John F. Hughes, Andries van Dam, Morgan McGuire, David Sklar, James D. Foley, Steven Keith Feiner and Kurt Akeley. "Computer graphics: principles and practice". Addison-Wesley (2014).

[34] Asmaa Hosni, Michael Bleyer, Christoph Rhemann, Margrit Gelautz and Carsten Rother. "Real-time local stereo matching using guided image filtering". In "Multimedia and Expo (ICME), 2011 IEEE International Conference on", pages 1–6. IEEE (2011).

[35] Pekka J Toivanen. "New geodesic distance transforms for gray-scale images". *Pattern Recognition Letters*, **17**, 5 (1996) 437–450.

[36] Rubén Cárdenes, Carlos Alberola-López and Juan Ruiz-Alzola. "Fast and accurate geodesic distance transform by ordered propagation". *Image and Vision Computing*, **28**, 3 (2010) 307–316.

[37] Emmanuel Jean Candes. "Ridgelets: theory and applications". Ph.D. thesis, Stanford University (1998).

[38] David L Donoho. "Ridge functions and orthonormal ridgelets". *Journal of Approximation Theory*, **111**, 2 (2001) 143–179.

[39] Jalal Fadili and Jean-Luc Starck. "Curvelets and ridgelets". In "Encyclopedia of Complexity and Systems Science", pages 1718–1738. Springer (2009).

[40] Stéphane Mallat. "Geometrical grouplets". *Applied and Computational Harmonic Analysis*, **26**, 2 (2009) 161 – 180. ISSN 1063-5203. http://www.sciencedirect.com/science/article/pii/S1063520308000444

[41] Justin K. Romberg, Michael B. Wakin and Richard G. Baraniuk. "Multiscale geometric image processing". In "VCIP", (2003).

[42] Minh N Do and Martin Vetterli. "The contourlet transform: an efficient directional multiresolution image representation". *IEEE Transactions on image processing*, **14**, 12 (2005) 2091–2106.

[43] Rashid Minhas, Abdul Adeel Mohammed and QM Jonathan Wu. "Shape from focus using fast discrete curvelet transform". *Pattern Recognition*, **44**, 4 (2011) 839–853.

[44] David L. Donoho and Mark R. Duncan. "Digital Curvelet Transform: Strategy, Implementation and Experiments". In "in Proc. Aerosense 2000, Wavelet Applications VII", pages 12–29. SPIE (1999).

[45] Emmanuel Candes, Laurent Demanet, David Donoho and Lexing Ying. "Fast discrete curvelet transforms". *Multiscale Modeling & Simulation*, **5**, 3 (2006) 861–899.

[46] Andrew Kingston, Imants Svalbe and Jean-Pierre Guédon. "The discrete Radon transform: a more efficient approach to image reconstruction?" In "Proc.SPIE", volume 7078, pages 7078 – 7078 – 10 (2008).

[47] ETHAN D Bolker. "The finite Radon transform". *Contemp. Math*, **63** (1987) 27–50.

[48] Minh N Do and Martin Vetterli. "The finite ridgelet transform for image representation". *IEEE Transactions on image Processing*, **12**, 1 (2003) 16–28.

[49] A Averbuch, RR Coifman, DL Donoho, M Israeli and J Waldén. "Fast slant stack: A notion of Radon transform for data in a Cartesian grid which is rapidly computable, algebraically exact, geometrically faithful and invertible". In "SIAM J. Sci. Comput", (2001).

[50] Andrew Kingston, Heyang Li, Nicolas Normand and Imants Svalbe. "Fourier Inversion of the Mojette Transform", pages 275–284. Springer International Publishing, Cham (2014). ISBN 978-3-319-09955-2. `http://dx.doi.org/10.1007/978-3-319-09955-2_23`

[51] WA Götz and HJ Druckmüller. "A fast digital Radon transform—An efficient means for evaluating the Hough transform". *Pattern Recognition*, **29**, 4 (1996) 711–718.

[52] Martin L Brady. "A fast discrete approximation algorithm for the Radon transform". *SIAM Journal on Computing*, **27**, 1 (1998) 107–119.

[53] William H Press. "Discrete Radon transform has an exact, fast inverse and generalizes to operations other than sums along lines". *Proceedings of the National Academy of Sciences*, **103**, 51 (2006) 19249–19254.

[54] Achi Brandt and Jonathan Dym. "Fast calculation of multiple line integrals". *SIAM Journal on Scientific Computing*, **20**, 4 (1999) 1417–1429.

[55] J. Radon. "Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten". *Akad. Wiss.*, **69** (1917) 262–277.

[56] A. Kak and M. Slaney. "Principles of Computerized Tomographic Imaging". Society for Industrial and Applied Mathematics (2001).

[57] Amir Averbuch, Ronald R. Coifman, David L. Donoho, Moshe Israeli and Yoel Shkolnisky. "A Framework for Discrete Integral Transformations I-The Pseudopolar Fourier Transform". *SIAM J. Scientific Computing*, **30**, 2 (2008) 764–784.

[58] Amir Averbuch, Ronald R. Coifman, David L. Donoho, Moshe Israeli, Yoel Shkolnisky and Ilya Sedelnikov. "A Framework for Discrete Integral Transformations II-The 2D Discrete Radon Transform". *SIAM J. Scientific Computing*, **30** (2008) 785–803. `https://sites.google.com/site/yoelshkolnisky/software`

[59] Amir Averbuch and Yoel Shkolnisky. "3D Fourier based discrete Radon transform". *Applied and Computational Harmonic Analysis*, **15**, 1 (2003) 33 – 69. ISSN 1063-5203. `https://mathworks.com/matlabcentral/fileexchange/61815`

[60] Jose G Marichal-Hernandez, Jonas P Luke, Fernando L Rosa and Jose M Rodriguez-Ramos. "Fast approximate 4-D/3-D discrete Radon transform for lightfield refocusing". *Journal of Electronic Imaging*, **21**, 2 (2012) 023026–1.

[61] William H Press, Saul A Teukolsky, William T Vetterling and Brian P Flannery. "Numerical recipes in C", volume 2. Cambridge Univ Press (1982).

[62] Henrik V. Sorensen and Charles Burrus. "Efficient Computation of the DFT with Only a Subset of Input or Output Points". *Signal Processing, IEEE Transactions on*, **41** (1993) 1184 – 1200.

[63] Ricardo Oliva-García, Óscar Gómez-Cárdenes, David Carmona-Ballester, José G Marichal-Hernández and José M Rodríguez-Ramos. In "Real-Time Image Processing and Deep Learning 2019", International Society for Optics and Photonics (2019).

[64] David L Donoho and Xiaoming Huo. "Beamlets and multiscale image analysis". In "Multiscale and multiresolution methods", pages 149–196. Springer (2002).

[65] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe and Frédo Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines".

[66] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". *ACM SIGPLAN Notices*, **48**, 6 (2013) 519–530.

[67] Jonathan Millard Ragan-Kelley. "Decoupling algorithms from the organization of computation for high performance image processing". Ph.D. thesis, Massachusetts Institute of Technology (2014).

[68] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley and Kayvon Fatahalian. "Automatically scheduling halide image processing pipelines". *ACM Transactions on Graphics (TOG)*, **35**, 4 (2016) 83.

[69] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand and Jonathan Ragan-Kelley. "Differentiable programming for image processing and deep learning in halide". *ACM Transactions on Graphics (TOG)*, **37**, 4 (2018) 139.

[70] Introducing Dataset. "Introducing the HDR+ Burst Photography Dataset" (2019). https://ai.googleblog.com/2018/02/introducing-hdr-burst-photography.html

[71] Jonathan Ragan-Kelley. "Halide" (2019). https://halide-lang.org/

[72] S. S. Chandra, N. Normand, A. Kingston, J. Guédon and I. Svalbe. "Robust Digital Image Reconstruction via the Discrete Fourier Slice Theorem". *IEEE Signal Processing Letters*, **21**, 6 (2014) 682–686. ISSN 1070-9908.

[73] (2019). https://github.com/shakes76/finite-transform-library

[74] Yoel Software. "Yoel Shkolnisky - Software" (2019). https://sites.google.com/site/yoelshkolnisky/software

[75] Said Pertuz. "Defocus Simulation". Matlab file exchange. https://es.mathworks.com/matlabcentral/fileexchange/55095-defocus-simulation.

[76] "Information technology – Automatic identification and data capture techniques – Code 128 bar code symbology specification". Standard, International Organization for Standardization (2007).

[77] "Information technology – Automatic identification and data capture techniques – EAN/UPC bar code symbology specification". Standard, International Organization for Standardization (2009).

[78] "Information technology – Automatic identification and data capture techniques – Data Carrier Identifiers (including Symbology Identifiers)". Standard, International Organization for Standardization (2008).

[79] "Information technology – Automatic identification and data capture techniques – Bar code verifier conformance specification – Part 1: Linear symbols". Standard, International Organization for Standardization (2006).

[80] "Information technology – Automatic identification and data capture techniques – Interleaved 2 of 5 bar code symbology specification". Standard, International Organization for Standardization (2007).

[81] R. Muniz, L. Junco and A. Otero. "A robust software barcode reader using the Hough transform". In "Proceedings 1999 International Conference on Information Intelligence and Systems (Cat. No.PR00446)", pages 313–319 (1999).

[82] Alessandro Zamberletti, Ignazio Gallo, Simone Albertini and Lucia Noce. "Neural 1D Barcode Detection Using the Hough Transform". *IPSJ Trans. Computer Vision and Applications*, **7** (2015) 1–9.