POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Low Cost High Resolution Position Sensor with Sub-micron Accuracy

Supervisor
Prof. Claudio Passerone

Candidate
Luca Favario

Academic Year 2020-2021

Abstract

High accuracy position sensors are essential components of many precision machines. Currently available commercial sensors can achieve nanometre precision, but they are expensive because they require accurate machining and calibration. Such solutions include strain sensors, capacitive sensors, laser interferometers and linear encoders. A cost-effective sensor capable of sub-micron precision readings, besides being a competitive alternative to the existing solutions, can be used to add positioning feedback in inexpensive systems, improving machine performance and process quality. Therefore, this thesis proposes a technique to achieve sub-micron precision but using only low-cost components. The approach consists of imaging a strip composed of a sequence of bands and correlating the images taken at subsequent instants to determine the displacement. This thesis aims, first, at analysing the technique from the mathematical point of view, studying and optimizing the correlation computation. It then presents an implementation of the sensor written in C, which is executed on an ARM board running Linux and uses a USB microscope. Next, the performance of the implementation is compared on different devices and with various configurations. Eventually, the sensor is tested with the help of a 3D printer to verify the correctness of the readings.

Contents

Chapter 1

Introduction

High resolution position sensors are necessary in many precision machines which require accurate displacement measures or precise closed loop motion control. The applications include machining devices, medical instruments and scientific experiments. Since many available sensors are expensive, because of the required accurate machining or long and complex calibration, a cost-effective sensor with sub-micron accuracy can be used to improve the displacement resolution or to add positioning feedback to inexpensive systems, improving the quality of the process and the performance of the machine.

The commercially available sensors use different technologies to achieve resolutions in the nanometer range: strain gauge sensors derive the position from a change in resistance of a thin magnetic foil when it is stretched. Piezoresistive sensors convert mechanical movement into electric signals. Inductive sensors, also known as eddy-current sensors, exploit the electromagnetic induction to measure the distance between a coil and a conductive object. Capacitive sensors measure the distance between two plates as a change in capacitance. LVDTs (linear variale displacement transformers) measure the displacement of a ferromagnetic core as difference in the voltage induced in the surrounding coils. Linear encoders consist of a reference scale and a optical or magnetic read head and, finally, laser interferometers use expensive laser technology.

These solutions use special parts which require accurate machining (e.g. the film of the strain gauge sensors or the scale of the linear encoders) and expensive components (e.g. laser components). Moreover, the outputs are electric quantities which require to use appropriate conditioning circuits to convert the signal into the displacement reading using the calibration function.

This thesis proposes a displacement sensor that uses only commercially available low-cost components to achieve sub-micron accuracy and that provides the readings directly as digital values through serial interface. The solution is an incremental sensor and the measurement travel is potentially unlimited. The structure of the sensor is similar to a linear encoder, in fact it consists of a strip and a read head: the strip is composed of white and black vertical bands and it is applied to the moving object, which moves in the direction perpendicular to the bands, while the fixed read head is a camera device that captures the strip and it is connected to the processing device. The method consists of imaging the strip and correlating the images captured at subsequent instants to determine the displacement. This technique does not require specialized parts or components: the

strip can be printed on a laser printer and a common camera module can be used as a capture device. Moreover, the implementation developed in this thesis is a C program which runs in Linux, therefore the solution is extremely portable: the processing unit can be any computer or board running Linux and the camera module can be a USB device.

The operation used to correlate the captured frames is the cross-correlation, which is used in many applications to compute the translation between two images (e.g. optical mouse devices, image registration programs). What differentiates this solution are two main characteristics: first, since the displacement is only in the direction of the strip, the images are converted to vectors by computing the sum of the columns of the frames before applying the correlation, reducing the memory usage and computational cost with respect to the 2D operation. Second, the cross-correlation is interpolated exploiting the properties of the discrete Fourier transform, allowing the sensor to evaluate the displacement between the frames in sub-pixel units.

The thesis is divided as follows: Chapter 2 analyses the technique from a mathematical point of view, Chapter 3 presents in details the C implementation of the sensor, Chapter 4 shows the experiments used to test the correctness of the computed positions, Chapter 5 analyzes the performance of the C program on different devices and Chapter 6 concludes the thesis.

Chapter 2

Position sensor technique

In this chapter the technique proposed in [1] is illustrated from a mathematical point of view, highlighting the advantages and the limitations of the design. In Section 2.1 the required mathematical background is shown and in Section 2.2 a first solution using fast Fourier transform (FFT) and zero-padding is discussed. Because of high computational cost and memory usage of the FFT technique, the sensor implemented in Chapter 3 uses a more efficient approach to compute the sub-pixel displacement, which uses the definition and the properties of the discrete Fourier transform (DFT), as described in Section 2.3. The two discussed methods produce exactly the same results, but the optimized solution uses only a fraction of the resources.

2.1 Technical background

2.1.1 Cross correlation

The solution illustrated is based on the cross-correlation, which is a mathematical operation that allows measuring the similarity of two functions considering the displacement between them. The definition of the cross-correlation between two functions f and g is:

$$(f \star g)(\tau) = \int_{-\infty}^{+\infty} f^*(t)g(t+\tau)dt \tag{2.1}$$

Each point of the resulting function corresponds to the measure of similarity for a specific shift τ between the two inputs. In this application, where the f and g are two translated signals, the position of the maximum of the correlation with respect to the center represents the displacement between the functions.

While the definition in the time domain is rather expensive, it is possible to evaluate the cross-correlation in a less complex way using the properties of the Fourier transform:

$$(f \star g)(\tau) = \mathcal{F}^{-1}\left(\mathcal{F}^*(f(t)) \cdot \mathcal{F}(g(t))\right) \tag{2.2}$$

This approach allows computing the cross-correlation as the inverse Fourier transform of the product of the transforms of f and g, with one of them conjugated.

In the discrete domain, the circular cross-correlation of two discrete signals g and f can be expressed using the discrete Fourier transform:

$$(f \star g)[n] = \mathrm{DFT}^{-1}(\mathrm{DFT}^{*}(f) \cdot \mathrm{DFT}(g)) \tag{2.3}$$

The circular cross-correlation is different from the linear one because the function g is shifted circularly (rotated) when computing the integral. The linear cross-correlation can be easily obtained from the circular one by extending the discrete signals f and g before applying the Fourier transforms, but the circular correlation is used in this application because it is less expensive and produces similar results (Section 2.2.4).

2.1.2 Zero-padding and upsampling

The zero-padding, also known as frequency interpolation, is a property of the DFT which allows interpolating a signal while evaluating the Fourier transform. To apply the zero padding, the signal is extended with zeros by an upsampling factor U before applying the DFT (or inverse DFT). The resulting vector after the DFT is the transform of the signal upsampled with factor U. This property, applied to the cross-correlation in the frequency domain, allows evaluating the interpolated spatial cross-correlation, therefore the maximum can be identified with sub-pixel accuracy. Instead of applying the definition of DFT it is convenient to use FFT algorithms which have a much lower complexity. A solution using zero-padding and FFT is shown in Section 2.2. While the zero-padding technique is very powerful used together with FFT algorithms, it requires to compute the complete output vector and to store the padded complex vectors in memory, which can have a large impact on performance. A more efficient alternative is described in Section 2.3, where the upsampling is applied using the definition of DFT and only a neighborhood of the peak is computed.

2.1.3 Hermitian redundancy

A function f is a Hermitian function if:

$$f^{*}(x) = f(-x) \tag{2.4}$$

When the Hermitian redundancy is applied to a vector v of length N, with N even:

$$v^{*}[n] = v[N-n]\,,\ 1 \leq n < N-1 \tag{2.5}$$

In other words, the first half of the vector $v[1 : N/2-1]$ is equal to the complex conjugate of the mirrored second half $v[N/2+1 : N-1]$, excluding the elements $v[0]$ and $v[N/2]$ which do not have a matching item in the second half. This redundancy means that half of the vector is redundant, and we can avoid storing it, saving memory.

A property of the DFT states that:

- the DFT of a real vector is a complex vector which satisfies the Hermitian redundancy;
- the DFT of a complex vector which satisfies the Hermitian redundancy is a real vector.

This property is a direct consequence of the DFT definition. Moreover, by definition of the DFT, items $v[0]$ and $v[N/2]$ of the DFT of a real vector are purely real.

The Hermitian redundancy is exploited by FFTW (Section 3.5.2) and by the optimized inverse DFT (Section 2.3.3) to reduce memory and computational cost.

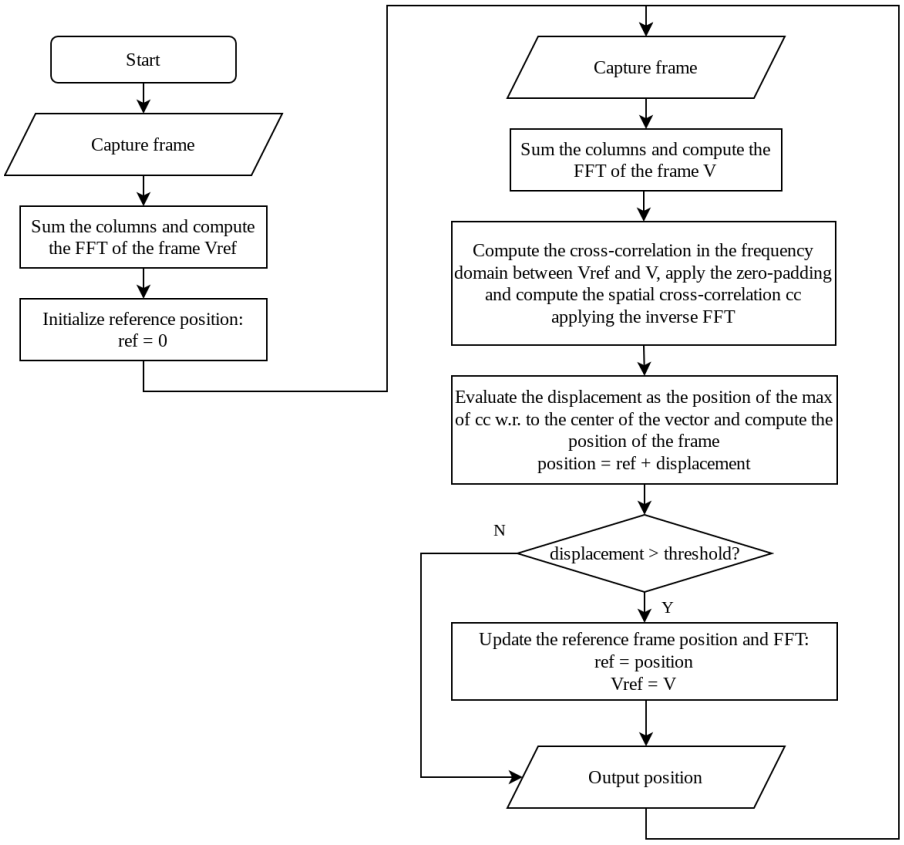

Figure 2.1: Flow chart FFT and zero-padding method.

2.2 Zero-padding and FFT solution

In this section it is discussed a solution using the zero-padding and FFT, which is different from the one implemented in the code, but it is worth mentioning because of its simplicity and because it shares many key points with the more complex one using the DFT. As shown in the flow chart in Figure 2.1, the solution consists of two main phases:

1. Set up phase: the first frame is captured, the sum of the columns and its Fourier transform is computed and the position is initialized to 0.

2. Main loop: a new frame is captured, the sum of the columns is evaluated and its transform is computed. Then, the cross-correlation between the first frame and the new one is computed in the frequency domain and the zero-padding is applied to it. The inverse FFT is applied to the padded vector and the upsampled displacement is evaluated as the position of the maximum with respect to the center of the vector. In the end the position is computed and, if the displacement is large enough, the reference frame transform and position are updated.

An example of the captured frames is shown in Figure 2.2, Figure 2.3 shows the vectors obtained as the sum of the columns of the frames and Figure 2.4 is the upsampled spatial

cross correlation.

The sensor based on the cross-correlation technique is an incremental position sensor, meaning that the position is not obtained as an absolute reading, but as the sum of the displacement of the previous frames.

The sensor should be able to evaluate the position of each frame produced by the camera device. To achieve this result, the time required to evaluate the position starting from the extracted frame has to be smaller than the frame time. If this constraint is not met, then some frames are skipped.

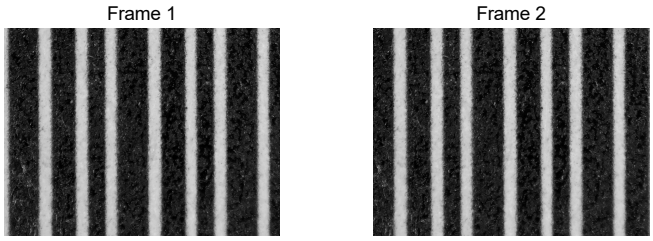

Figure 2.2: Example of captured frames with resolution 1600×1200 and shift 81.7 pixels

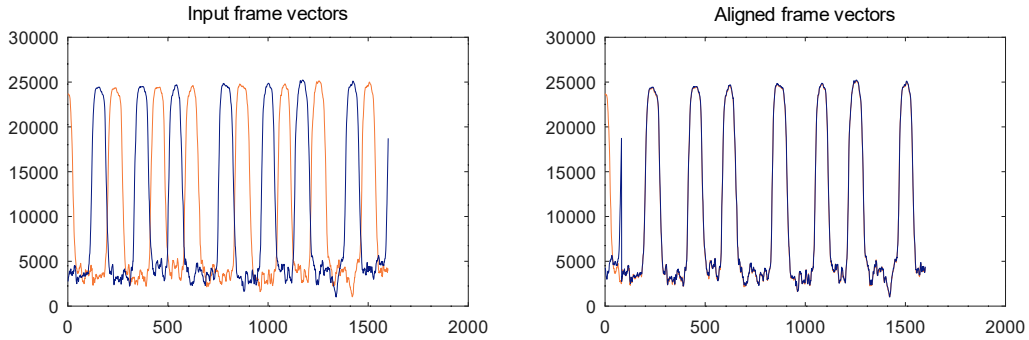

Figure 2.3: Vectors obtained as the sum of the columns of the frames. The left plot shows the original vectors, the right plot shows the vectors aligned in position corresponding to the maximum of the cross-correlation.

2.2.1 Tag design

The tag is a strip composed of black and white vertical bands which is applied to the moving objects and which is framed by the camera. The bands should be high enough to fill the field of view of the camera and the tag can have the desired length independent of camera horizontal field of view. In the vector produced by the sum of the columns of the frame the white bands corresponds to the peaks, while the black bands have a low value. The width of the bands is variable to allow the cross-correlation to recognize the pattern in the correlated frames, in particular the tag which has been used has a fixed

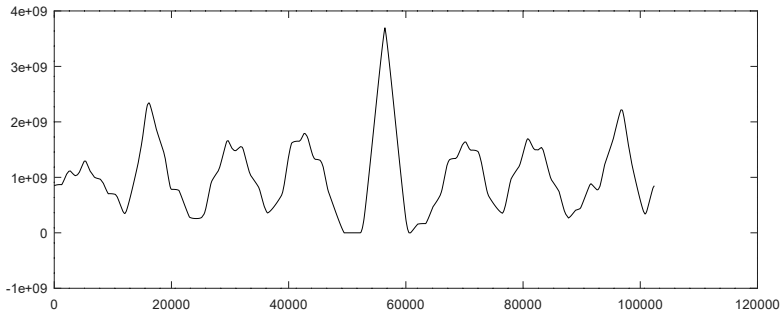

Figure 2.4: Upsampled spatial cross correlation. Horizontal resolution 1600 and upsampling 64.

width $2B$ for the white bands and a variable width ($2B$, $3B$ and $4B$) for the black bands, as shown in Figure 2.5. The actual size of the bands can be adjusted depending on the application, but it is important that the relative sizes of the bands are variable to recognize the position of the object. In fact, when evaluating the correlation between the frames with differently sized bands, there are many local peaks, but the one corresponding to the correct displacement has the highest value.

The implementation proposed in Chapter 3 converts the captured frames to grey scale, for this reason black and white bands are preferred. Moreover it is recommended that there are always at least 5 or 6 white bands in the frame to provide enough bands for the algorithm to recognise the movement correctly and because, if there are at least 4 white bands, they can be exploited to compute automatically the conversion factor from pixels to millimeters, as shown in Section 3.5.6.

The tag can be printed on a common laser printer and the defects in the print are filtered when computing the sum of the columns.

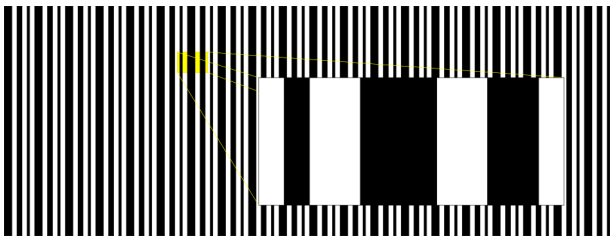

Figure 2.5: Tag design [1]

2.2.2 Sum of the columns of the frame

The first step, after having captured the frame, is computing the sum of the columns of the image into a vector v of length N, equal to the horizontal resolution of the frame. This step allows computing the rest of the algorithm working with 1D vectors instead of 2D frames, hugely improving the performance. The columns can be added without the loss of useful information because the framed strip has black and white vertical bands, which

means that the pixels of the captured frame are replicated in the columns. To be able to use this method some conditions have to be satisfied: first of all print quality of the strip and alignment of the bands in the frame should be sufficient to have multiple separated peaks as sharp as possible in the vector. Moreover, good lightning on the whole scene is necessary to reduce noise and produce peaks of the same height. An extra positive effect of the sum of the columns is the reduction of the noise with respect to the original frame.

The sum requires to compute NxM integer additions, where N is the horizontal resolution and M the vertical resolution. As a result, it can be quite expensive, even compared with the cost of the displacement computation. In fact, the complexity of the sum of the column can be approximated as $\mathcal{O}(N \cdot M)$, while the complexity of the displacement computation is $\mathcal{O}(NU \log(NU))$. The complexity of the sum is higher if $M \gg U$. Moreover, if the optimized inverse DFT is used (Section 2.3), the complexity of the displacement computation becomes $\mathcal{O}(2UN)$, which is lower than the complexity of the sum if $M > 2U$.

A possible approach to reduce the time required by the sum of the columns is to reduce the number of rows which are added. If only $R = M/4$ top rows are considered, then the complexity of the sum is reduced by a factor of 4. The choice of R should take into account both the performance gain and the loss in filtering capabilities of the sum. In fact, if less rows are considered, the black bands are noisier and the white bands peaks are less defined. A value which has produced good results with resolutions larger $\geq 640 \times 480$ during testing is $R = M/8$, which requires to sum only 1/8 of the rows. In case of smaller resolutions (e.g. 320×240), to obtain a sufficient filtering effect, it may be appropriate to increase the number of rows to $R = M/4$ or $R = M/2$. The comparison among different R values is shown in Figure 2.6

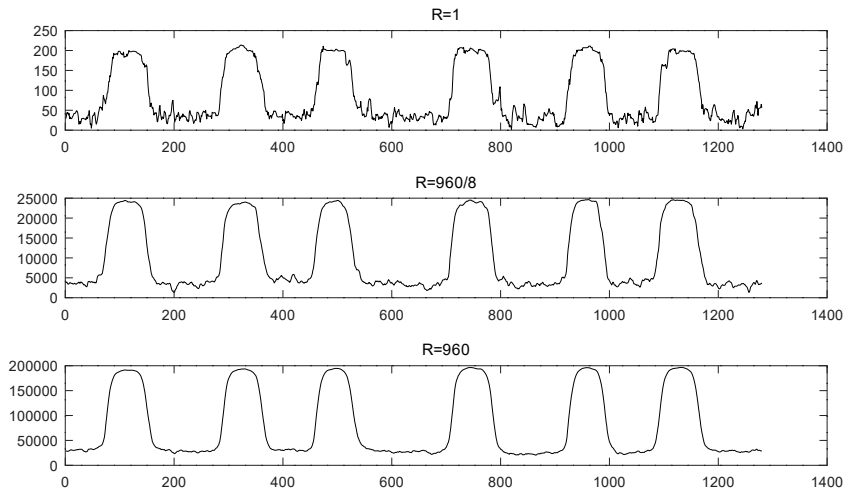

Figure 2.6: Frame vector computed with different R values. Resolution 1280×960.

The second advantage of considering less rows at the top of the image is the resilience to the tilted frames, which are captured by sensors with a rolling shutter when the object is moving. In fact, if a smaller set of rows is considered, the bands are approximately vertical and the sum of the column produces well defined peaks in the correct position, hence the

algorithm can be applied correctly. If all the rows are considered, the sum of columns flattens and the peaks are less sharp, hence a wrong displacement could be evaluated. The vectors computed in case of tilted bands and the used frame are shown in Figure 2.8 and Figure 2.7.

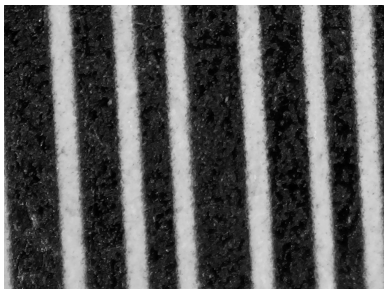

Figure 2.7: Frame with tilted bands due to the object moving and rolling shutter sensor

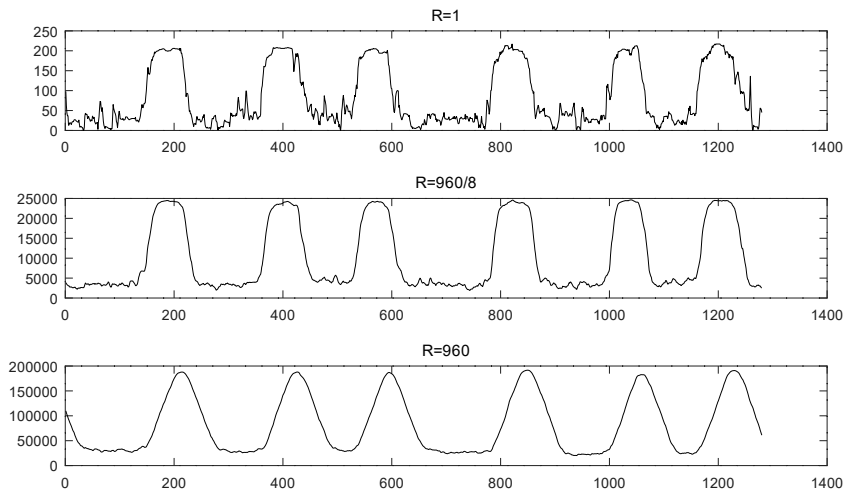

Figure 2.8: Frame vector of the tilted frame computed with different R values. Resolution 1280×960.

2.2.3 Displacement evaluation

When the vector computed as the sum of columns of the frame v is available, the FFT can be applied, producing the complex vector V in the frequency domain with length N. Once both the transform of the reference vector V_{ref} and of the new vector V are available, the complex cross-correlation in the frequency domain CC can be computed as the product element by element of the two vectors V_{ref} and V with one of them complemented. Now, to obtain a displacement in sub-pixel with upsampling factor U, the zero-padding is applied to CC before the inverse FFT: the vector CC with length N is extended with zeros to $N \cdot U$

elements in the vector CC'. After the zero padding, the inverse FFT is applied, obtaining the real spatial correlation cc' of length $N \cdot U$. Now the displacement in upsampled units between the two frames can be computed searching the position of the maximum with respect to the center of the vector.

Once the displacement is available, the position with respect to the first image captured can be computed as the sum of the position of the reference frame and the displacement. The position of the reference frame ref is initialized to 0 at startup and it is updated when a new reference is chosen. Moreover, when the reference frame is updated, the content of the vector V is moved to V_{ref}. In doing so, V_{ref} always stores the FFT of the last reference image, therefore there is no need to recompute it when new frames are available. The reference image is updated if the displacement is larger than a arbitrary threshold, which is chosen to ensure that correlation between the frames can always be computed correctly. The computed position is in sub-pixel units and it can be converted to pixels dividing the position by the upsampling U, obtaining a fractional result.

The advantages of the FFT method are its simplicity and the fact that it can be easily implemented and parallelized on different systems, hardware or software, thanks to the availability of FFT libraries and FPGA implementations. On the other hand one of the main disadvantages is the memory cost. In fact, the technique of the zero-padding requires to extend the vectors by a factor of U, which could be as large as tens or hundreds of times. Moreover, although FFT algorithms are fast, they require to compute the whole output vector which is not required by this technique and, in case of large U, can be extremely expensive in terms of computational complexity. One alternative approach to compute the upsampled DFT in the neighborhood of the peak, without evaluating the whole vector, is proposed in [2] and adapted to the system in Section 2.3.

2.2.4 Circular cross-correlation

GNU Octave, a high level language for numerical computations [3], has been used to decide whether in this application the circular cross-correlation can be used instead of the more expensive linear correlation. The program uses two frames, extracted from a high resolution image of the tag, shifted between each other by an arbitrary fractional amount. The algorithm is applied to the two frames, using both the linear and the circular correlations, and the results are compared with the expected value.

In Figure 2.9 an example is reported using resolution 1280×960 and upsampling 64. As we can observe in the figure, both the linear and the circular correlations do have an uncertainty in the order of few pixels in case of large displacements. The error depends on the resolution, the position of the bands in the frame and the noise in the black bands and, based on the combination of these factors, the circular correlation can produce better results than the linear one or vice versa.

Since the precisions obtained with the linear cross-correlation and the circular one are similar, then the circular correlation is used in the algorithm because its complexity is half the one of the linear one, both in terms of memory cost and computational complexity. Further considerations on the computed displacement accuracy are discussed in Section 2.2.5.

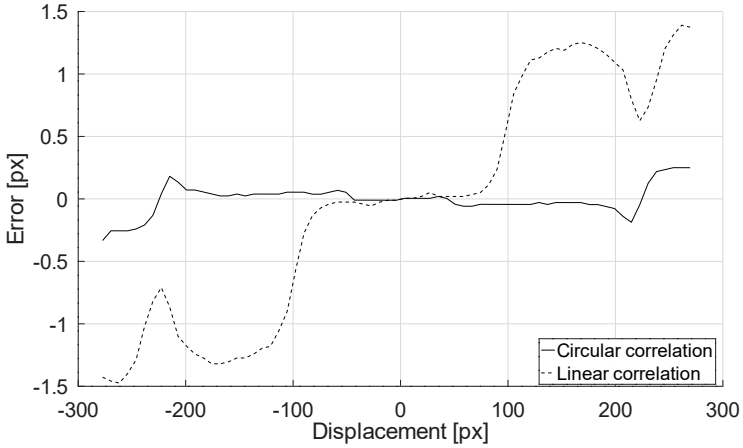

Figure 2.9: Error of the displacement computed with correlation with respect to the expected displacement. Resolution 1280×960 and upsampling 64.

2.2.5 Displacement and position error

While the cross-correlation in the ideal situation can provide infinite resolution and accuracy by increasing the upsampling U, this is not the case in real applications. In this section some considerations regarding the relation between the frames, the displacement and the displacement position are discussed. The considerations are based on the observation of the behavior of different configurations simulated in Octave.

First, the accuracy depends on the composition of the captured frames. In fact, both the number of peaks and their position have a large influence on the error of the displacement. Since the circular cross-correlation is used, when the frames are shifted one with respect to the other to compute the correlation between them, the section of the vector which is shifted out at the end is placed at the beginning. In the ideal case the frame has exactly the same width of one, or a multiple, of the set of bands of the strip (3 white and 3 black bands in this case) or captures precisely a group of bands (e.g. 4 white and 4 black) and the peaks of the rotated frame always has the same spacing of the strip. On the other hand, if the frame is smaller or larger than a set of bands, the peaks in the shifted vectors can be closer or further with respect to the bands on the strip, or even split in half. In this case the second frame is not a rotated copy of the first one and the circular cross-correlation may not provide the correct displacement value. This behavior has been verified in Octave using couples of frames extracted from an higher resolution image and shifted of an arbitrary amount between each other: the first couple of frames has a width similar to a set of bands, while the second has a larger width, as shown in Figure 2.10. The difference between the displacement, computed with upsampling 256, and the expected result, which is the shift applied to the images, is reported in Figure 2.11. The "ideal" couple has a maximum error of 0.005 pixels, while the other frames have a maximum error of 1.3 pixels. Since the captured bands combination largely affect the precision, exploring different bands widths and sets may lead to smaller displacement errors and larger displacement ranges.

A possible solution to solve this problem when working with small displacements is to

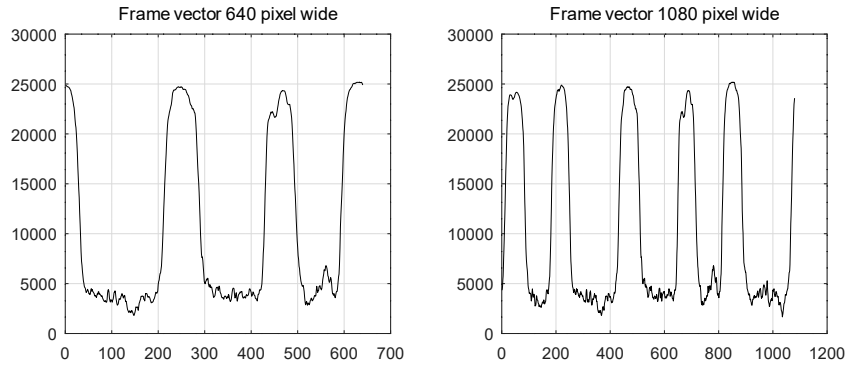

Figure 2.10: On the left the vector of the frame with the same width of a set of bands. On the right the vector with width larger different from an integer set of bands.

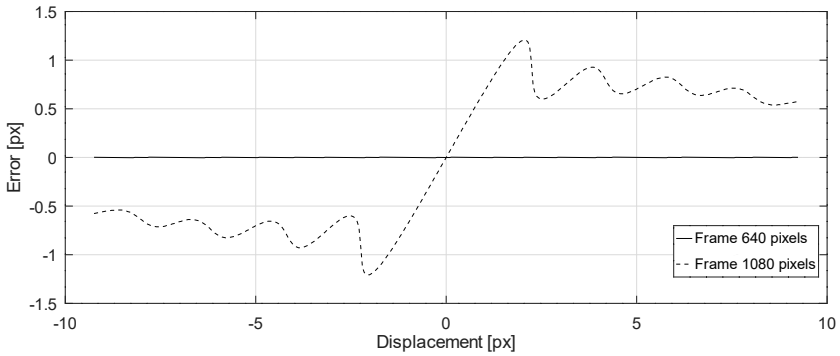

Figure 2.11: Difference between the computed displacement and the shift of the images. The continuous plot is computed using frames with the same width of a set of bands (640 pixels), while the dashed one uses frames with larger width (1080 pixels).

consider only a subset of columns of the frame corresponding to a set of bands (or an integer multiple), falling back to ideal conditions. Since a set of band is composed of 3 white bands and 3 black bands, its width can be measured on the first reference frame by calculating the number of pixel between the 1st and the 7th transitions. The computed value can be used to crop the width of all the following frames, since the distance between the camera and the strip is constant. This approach allows to largely reduce the error, for instance the maximum error of the previous non-ideal case is reduced from 1.21 to 0.009 pixels, which is similar to the "ideal" case, as shown in Figure 2.12. The advantage of this method is that it is not necessary to guarantee that the field of view corresponds exactly to the set of bands because the frame is cropped to the correct width in software. This solution is not implemented in the C code and can be object of future work.

The second parameter which affects the accuracy is the displacement itself. Using the same Octave program used above and increasing the maximum displacement (Figure 2.13), we can observe that the error increases largely with the displacement. The increase of the error is caused by the fact that, even if bands with the same width are captured, the images capture different positions on the tag, therefore the vectors are not identical. Moreover,

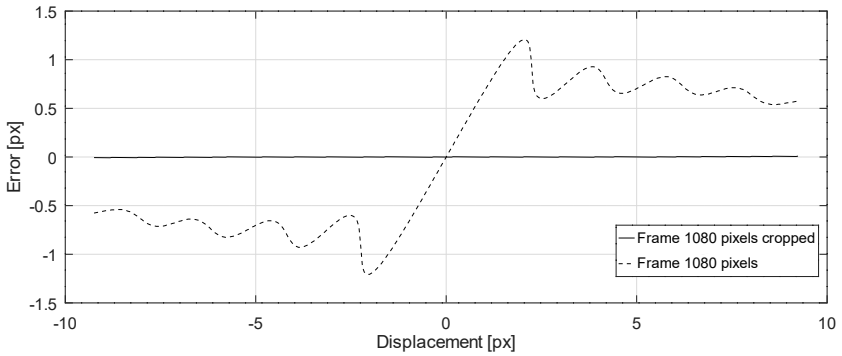

Figure 2.12: Difference between the computed displacement and the shift of the images. The continuous plot is computed using the 1080 pixel frames cropped to the same width of a set of bands, while the dashed one uses the full 1080 pixels frames.

if the frame is not ideal and the displacement is larger than a certain amount Δ, then the error jumps to unrealistic values because the maximum is obtained aligning the wrong peaks (Figure 2.14).

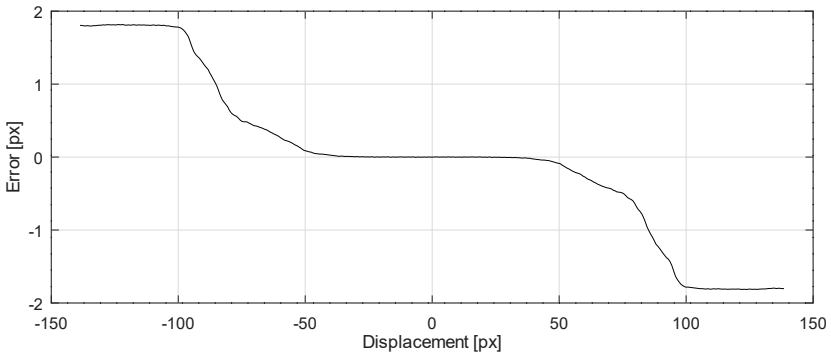

Figure 2.13: Difference between the computed displacement and the shift of the image. Frames with 640 horizontal resolution.

When working with large displacements the error can be in the order of pixels. It is possible to apply the previous solution to work with a more suitable frame but the error can still exceed the pixel. The error can be reduced with another approach, which is trying to remove the contribution of the black bands to the correlation by setting to zero the points of the vector of the frame which are below a threshold (e.g. 50% of the amplitude). Considering the previous frame with horizontal resolution 1080 and maximum displacement 230 pixels, the maximum error without any correction is 2.8 pixels. With the selection of one set of bands the error is reduced to 1.1 px and if the black bands are set to zero, without cutting the vector to the ideal length, the error is 0.6 pixels (Figure 2.15). While this solution reduces the absolute error in case of large displacements, it is not suitable for small displacements, where the cutting method produces better results.

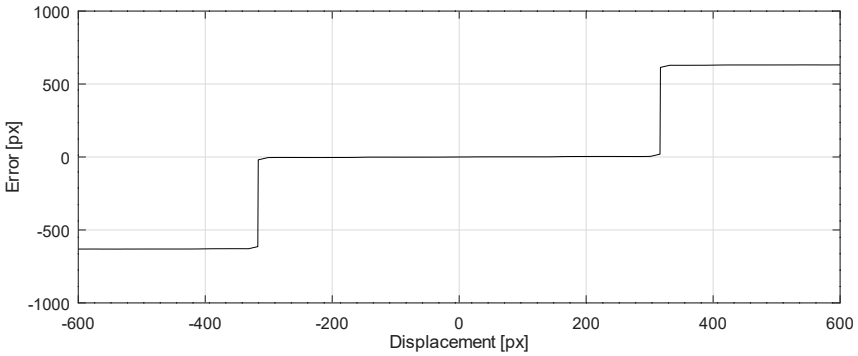

Figure 2.14: Difference between the computed displacement and the shift of the image. Frames with 1200 horizontal resolution.

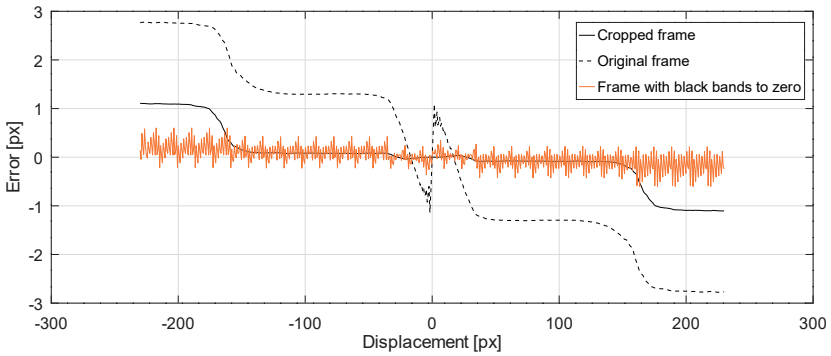

Figure 2.15: Difference between the computed displacement and the shift of the image. Frames with 1080 horizontal resolution.

As a conseguence, we can identify two operating modes. Given the maximum displacement Δ and the displacement τ:

- large displacement: if $\Delta/8 < \tau < \Delta$, the displacement can be determined with error in the order of 1/10 px by setting the black bands to zero;
- small displacement: if $\tau < \Delta/8$, then the error is in the order of 1/100 px if a vector with width equal to a set of vertical bands (3 black and 3 white bands) is considered.

While working in large displacement allows to increase the speed of the sensor, the accuracy of the displacement is decreased. On the other hand in small displacement conditions the accuracy is improved, but the speed is reduced. The speed and the maximum displacement are discussed in the following Section 2.2.6

The target sub-micron accuracy can be obtained with a more expensive high resolution camera sensor which supports large displacements or with a cheaper solution which supports only smaller displacements. In the first case, assuming a maximum error of 0.6 pixels and a desired uncertainty of 0.1 µm, each pixel should correspond to 0.16 µm. If the sensor has a pixel size of 1.1 µm, as it is common high resolution smartphone cameras (for

instance 13 MP 1/3" sensor), a magnification factor of 7 is still required. If the solution with small displacements is allowed and we assume a maximum error of 0.01 pixels and a desired uncertainty of 0.1 µm, then each pixel should correspond to a maximum of 10 µm. Assuming a magnification of 1x the sensor pixel size should be smaller than 10 µm, which is a suitable value for low cost camera sensors (e.g. the 1/4" OV5647 sensor used by the 5 MP Raspberry Pi camera).

In addition to the displacement uncertainty, the incremental behavior of the sensor is a source of error in the measurement of the position. In fact, the error accumulates for positions which require updates of the reference frame. In the worst case, the total error is the product between the number of reference frames updates and the maximum error of the displacement between two frames. As a result, a small number of updates can compromise the sub-micron accuracy of the sensor and requires using a sensor with smaller pixel size or higher magnification. Moreover, the measurement range of the sensor, although it is not limited by the technique, is limited by the required accuracy. For instance, assuming 5 µm pixel size and magnification 1, the error for small displacements is 0.05 µm. If the reference frame is updated when the displacement is larger than 20 pixels and the measurement range is 1 cm, the reference frame is updated 100 times and the maximum error of the position at full range is 5 µm. Instead, if the measurement range is 1 mm, the error is reduced to 0.5 µm. In this case, the position uncertainty can be defined as 0.05 %FS, with FS equal to the measurement range.

This problem can be mitigated with a voting or averaging system which reduces the displacement error: multiple different reference frames (e.g. 3) are stored and the displacement is computed with respect to each reference for every new frame. Then the displacements are compared with respect to each other and only the closer values are averaged and returned. This solution is almost three time more expensive in terms of computational cost and would require to further reduce the maximum speed.

The final position uncertainty and resolution can not be easily estimated with the available testing setup and should be evaluated using more accurate equipment and rigorous methodology. The analysis of the sensor accuracy and the implementation of the voting system, which are not developed in this thesis, can be object of future work.

2.2.6 Maximum displacement and speed

The maximum theoretical displacement between two frames is half of the horizontal resolution of the image $N/2$, which is the largest value that can be allowed while being able to distinguish the direction of the shift. If larger translations were allowed, then the camera could frame the same bands combination on the tag moving in both directions.

A stricter constraint when considering large displacements is due to the behavior of the circular cross-correlation. When the shifted vector wraps around then it can happen that the rotated peak is partially overlapping the peak of the reference image, as result it may produce a maximum value at the incorrect displacement τ'. When the frames includes more than 3 white and 3 black bands, a peak is split at the end of the frame and the displacement is large enough, it is common that the computed displacement is obtained rotating the image in the opposite direction and aligning the wrong peaks. For instance in Figure 2.16 the frames with horizontal resolution 1080 pixels are shifted by 250 pixels with respect to each other, which is smaller than the maximum value $N/2$. Instead, the

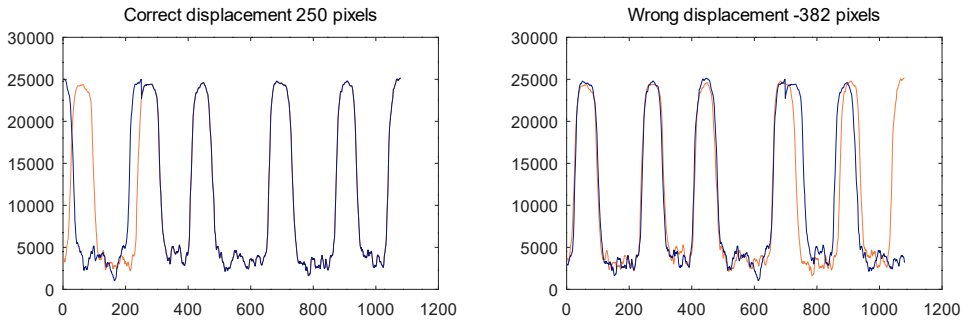

Figure 2.16: Frame vectors aligned in the expected position (left) and in the wrongly computed displacement (right)

computed displacement is -382 pixels because the wrong combination of peaks produces an higher peak value in the cross correlation (Figure 2.17). Since the bands captured in the frame affect the correlation results, exploring different tag designs may improve the displacement range.

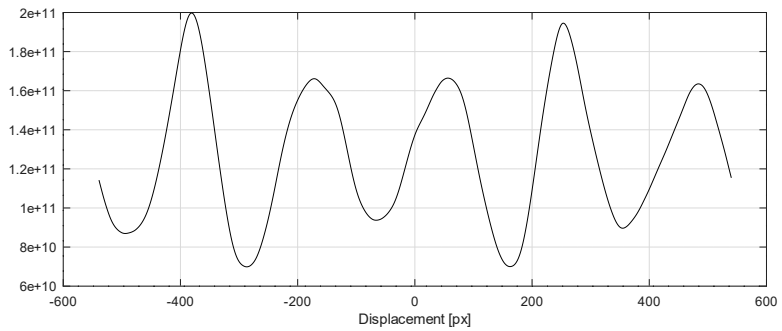

Figure 2.17: Spatial cross-correlation. Correct peak in 250, wrong peak in -382

To ensure that the correct band is considered, then the maximum displacement Δ has been limited to a value obtained experimentally in the testing phase, which is 1.3 times the average distance between two peaks d_{pp} in pixel units in the first captured frame. The reference frame is updated when the displacement is larger than a threshold γ which is equal to the average distance between the peaks d_{pp} multiplied by a factor of 0.8, to ensure that the reference frame is used at least two times at the maximum speed. While this approach has provided good results in terms of reliability, it has the drawback to limit the speed of the sensor. Considering the sampling rate of the sensor $f_s = 30\,\text{Hz}$, a frame with horizontal resolution $N = 800\,\text{px}$ and 6 peaks in the frame, then the maximum allowed speed is:

$$\begin{aligned} \gamma &= 0.8 \cdot d_{pp} \\ \Delta &= 1.3 \cdot d_{pp} \\ v_{\max_{px}} &= (1.3 - 0.8) d_{pp}\, f_s = 0.5\, \frac{800\,\text{px}}{6}\, 30\,\text{Hz} = 2000\,\text{px/s} \end{aligned} \tag{2.6}$$

The real speed is lower than the theoretical speed $v = 0.5\,N\,f_s = 6000\,\text{px/s}$. Assuming that one set of bands (3 white and 3 black) is 3.75 mm wide and it is represented on 600 pixel, then each pixel corresponds to $C = 6.25\,\mu\text{m}$ (Section 3.5.6) and the maximum speed is:

$$\begin{aligned} v_{\max} &= C\,v_{\max_{px}} = C\,(1.3 - 0.8)d_{pp}\,f_s = \\ &= 6.25\,\frac{\mu\text{m}}{\text{px}}\,2000\,\frac{\text{px}}{\text{s}} = 12.5\,\text{mm/s} = 0.75\,\text{m/min} \end{aligned} \tag{2.7}$$

If the required precision is in the order of $1/100\,\text{px}$, then the sensor has to work with small displacements: the maximum displacement is reduced to $\Delta' = \Delta/8$ and the threshold $\gamma' = \gamma/8$. As a result the speed is reduced by a factor of 8, too:

$$v'_{\max_{px}} = \frac{(1.3 - 0.8)d_{pp}\,f_s}{8} = 0.0625\,d_{pp}\,f_s = 250\,\text{px/s} \tag{2.8}$$

And the speed in millimeter units is:

$$v'_{\max} = C\,v'_{\max_{px}} = 1.6\,\text{mm/s} = 94\,\text{mm/min} \tag{2.9}$$

To increase the speed it is necessary increasing the sampling rate of the sensor f_s, improving both the camera frame rate and the processor computational capabilities, or increasing the horizontal resolution of the frame N, the distance between the camera and the tag and the size of the bands on the strip, which allows to keep the same C coefficient while increasing d_{pp}. If it is necessary to increase the speed without the possibility to upgrade the camera, then moving the camera further from the strip and increasing the size of the bands rises the coefficient C and the speed $v_{\max}$, but at the cost of a drop in the displacement resolution. Another possible solution to increase the speed is to reduce the threshold displacement γ, but the drawback is that the reference frame is updated more often, which can lead to an accumulation of the error and to a worse precision for the position.

In conclusion, increasing the displacement range allows to increase the speed but also increases the error, while reducing the speed allows to increase the precision (Section 2.2.5).

2.3 Inverse Fourier transform optimization using DFT

In this section it is presented an alternative approach to the upsampling obtained using FFT and zero-padding. This method is based on the upsampled image registration algorithm proposed in [2], which has then been adapted to the 1D case and optimized for this application. While the FFT approach can be very simple and flexible, it can be expensive in terms of memory cost and computational complexity because it requires computing and storing the whole upsampled vector of the spatial cross-correlation, although we are interested only in the upsampling of the region around the peak. The following approach allows to compute the upsampled spatial correlation only in an interval centered in the peak, reducing memory and computational cost.

As shown in Figure 2.18, the algorithm is divided in two parts: first the pixel displacement has to be computed, then the sub-pixel displacement is evaluated in a small window

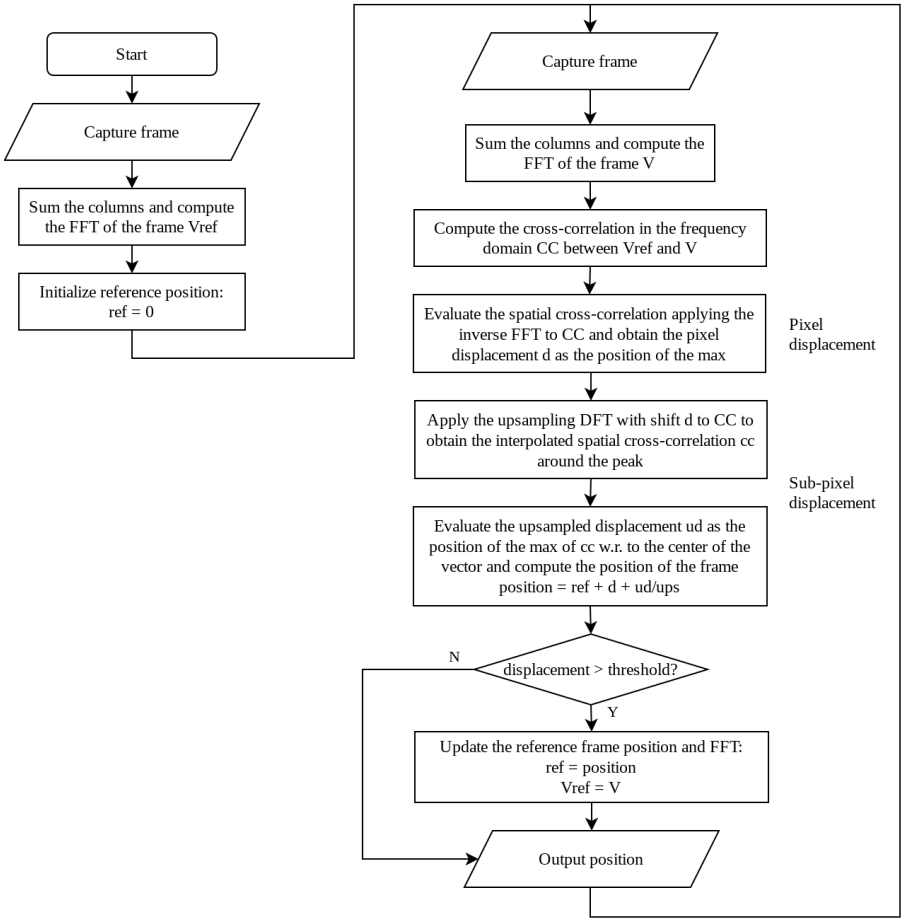

Figure 2.18: Flow chart inverse DFT upsampling.

of the upsampled spatial cross-correlation. The pixel displacement obtained in the first part is used by the DFT to center the upsampling interval in the peak.

To compute the pixel displacement it is sufficient to compute the correlation in the frequency domain and evaluate the inverse FFT, without zero-padding the vector or with a zero-padding factor of 2 (Section 2.3.6), to obtain the spatial cross-correlation. The position of the maximum with respect to the center is the required pixel displacement. For instance if the frames are shifted by 54.6 pixels, then the pixel displacement is 55 (Figure 2.19)..

Once the coarse displacement is available, the inverse DFT can be computed. The upsampled spatial correlation is computed only in a small window which is is centered in the position identified by the coarse displacement. The shift and the upsampling are applied while computing the inverse transform exploiting the properties of the DFT. The obtained spatial vector is centered in the coarse displacement, hence the upsampled displacement is computed with respect to the pixel displacement and the total shift is the sum of the sub-pixel displacement and the pixel one. Using the previous example, if the frames are shifted by 54.6 pixels and the computed pixel displacement is 55, then the expected displacement

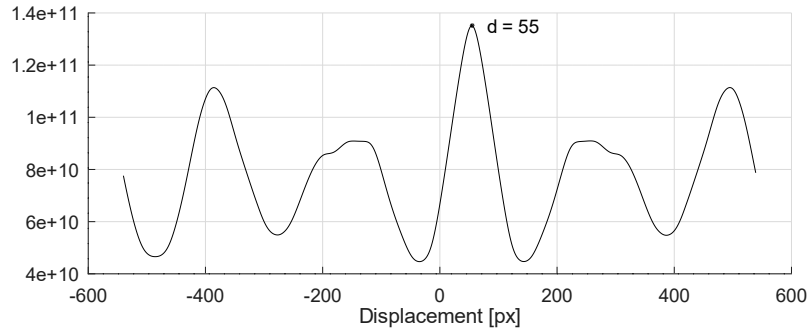

Figure 2.19: Non-upsampled spatial cross-correlation. The computed pixel displacement is 55 pixels.

is -0.4 pixels. Since the upsampling factor is 10, then $-0.4\,\text{px}$ correspond to -4 upsampled pixels (Figure 2.20).

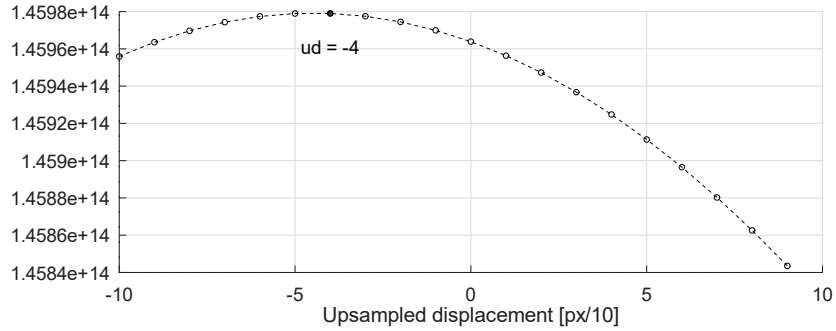

Figure 2.20: Upsampled spatial cross-correlation around the peak with upsampling factor 10. The additional upsampled displacement is -4/10 pixels.

Before showing the implemented algorithm, the necessary properties are recalled.

2.3.1 Shift theorem

The shift theorem for the DFT states that a displacement in the spacial domain δ corresponds to a linear phase term in frequency domain $e^{-j\omega_k\delta}$:

$$x(n-\delta) \longleftrightarrow e^{-j\omega_k\delta}X[k] \tag{2.10}$$

In this application the cross-correlation in the spacial domain has to be centered in its peak after the inverse transform. To obtain this result each element of the correlation has to be multiplied by the linear phase term corresponding to the pixel displacement before applying the transform:

$$CC'[k] = e^{-j\frac{2\pi k}{N}\delta}CC[k] \tag{2.11}$$

2.3.2 DFT upsampling

Another important observation is that, using the definition of DFT, we can obtain the same effect of the zero-padding technique without the need to apply the zero-padding and to evaluate the inverse transform on the whole vector. The result is similar to the scaling property of the Fourier transform. Consider the vector in the frequency domain CC with length N and the desired upsampling U. Assume that the frequency 0 is placed at the start of the vector. The resulting length of the vector after the application of the zero-padding CC' is $N \cdot U$, where only N elements are different from zero, and they are placed in the first $N/2$ and last $N/2$ positions. Now apply the inverse DFT to such vector:

$$cc'[m] = \sum_{k=0}^{NU-1} CC'[k] e^{j\frac{2\pi}{NU}mk}, \, 0 \leq m < NU - 1 \tag{2.12}$$

Due to the periodicity of the argument of the phase term with period $N \cdot U$, the inverse DFT can be written as:

$$cc'[m] = \sum_{k=0}^{NU/2-1} CC'[k] e^{j\frac{2\pi}{NU}mk} + \sum_{k=-NU/2}^{-1} CC'[k+NU] e^{j\frac{2\pi}{NU}mk} \tag{2.13}$$

Now, if the terms of CC' equal to zero are neglected:

$$cc'[m] = \sum_{k=0}^{N/2-1} CC'[k] e^{j\frac{2\pi}{NU}mk} + \sum_{k=-N/2}^{-1} CC'[k+NU] e^{j\frac{2\pi}{NU}mk} \tag{2.14}$$

In the first summation the first $N/2$ elements of CC' are equal to the first $N/2$ elements of CC. In the same way, in the second summation the last $N/2$ elements of CC' correspond to the last $N/2$ items of CC. So the expression can be written as:

$$cc'[m] = \sum_{k=0}^{N/2-1} CC[k] e^{j\frac{2\pi}{NU}mk} + \sum_{k=-N/2}^{-1} CC[k+N] e^{j\frac{2\pi}{NU}mk} \tag{2.15}$$

This shows that the same cc' upsampled vector obtained applying the inverse transform on the zero-padded vector can be computed without padding the starting vector and computing the sum of only N elements.

The cost of computing the whole cc' vector is $\mathcal{O}(N^2 \cdot U)$, which is larger than the FFT $\mathcal{O}(NU \cdot \log(NU))$ in this application where N, representing the horizontal resolution of the captured images, is usually larger than the upsampling U. But, as shown in the next section, if only a small section of cc' is computed, then this approach can be faster than the FFT.

2.3.3 Hermitian redundancy

In this application the images are saved in purely real vectors $v1$ and $v2$ and their DFT $V1$ and $V2$ are computed. Due to the Hermitian property the vectors $V1$ and $V2$ satisfy the Hermitian condition. The cross-correlation of the two vectors is computed as:

$$CC[n] = V1[n] \cdot V2^*[n] \tag{2.16}$$

Since both $V1$ and $V2$ satisfy the Hermitian redundancy, then the vector CC is Hermitian too. Then, the zero-padding is applied to CC and the resulting vector CC' still satisfies the Hermitian condition because the zeroes are added to the vector without changing the symmetry with respect to the center. As a last step, the inverse DFT is applied to the padded vector CC' and, since it satisfies the Hermitian redundancy, the obtained vector in the spacial domain cc' is real.

Since the vectors used in this application are Hermitian then some optimizations can be applied. The $V1$ and $V2$ are stored in complex vectors with length $N/2+1$, half their original length. Then the cross-correlation CC is computed only on half the vector, reducing the required memory and number of multiplication by a factor of two. The inverse DFT can be optimized too: since the spacial correlation is real, then only the real part of the vector cc' is computed. Moreover, only half of the products of the DFT definition are computed and added, then multiplied by two. These optimizations are explained more in detail when describing the C program in Chapter 3.

The FFT used in the program (FFTW) exploits the Hermitian redundancy to improve memory usage and speed by a factor of two [4].

2.3.4 Inverse DFT upsampling in the neighborhood of the peak

Combining the DFT shift property and the DFT upsampling shown in the previous sections, we can obtain the expression which allows shifting and upsampling while computing the inverse DFT. Given N the length of the vector CC, the upsampling factor U, the computed pixel displacement in the spacial domain d converted in upsampled units, and defining W the width in upsampled units of the neighborhood around the peak to be considered in the spatial domain:

$$\begin{aligned} cc'[m] = &\sum_{k=0}^{N/2-1} CC[k] e^{-j\frac{2\pi}{NU}(-d+W/2)k} e^{j\frac{2\pi}{NU}mk} + \\ &\sum_{k=-N/2}^{-1} CC[k+N] e^{-j\frac{2\pi}{NU}(-d+W/2)k} e^{j\frac{2\pi}{NU}mk} = \\ = &\sum_{k=0}^{N/2-1} CC[k] e^{j\frac{2\pi}{NU}(m+d-W/2)k} + \sum_{k=-N/2}^{-1} CC[k+N] e^{j\frac{2\pi}{NU}(m+d-W/2)k} \end{aligned} \tag{2.17}$$

The spatial cross-correlation obtained with the inverse transformation is rotated in the vector, in particular the position which should be the center is in position 0 at the beginning of the vector. Shifting the vector by the displacement d allows positioning the peak in position 0, while the shift term $-W/2$ centers the peak in the window of width W.

This expression can be simplified since, as shown in Section 2.3.3, CC satisfies the Hermitian redundancy. Moreover, the phase terms of the DFT satisfy the Hermitian redundancy too, in fact the arguments of the exponentials in first summation are:

$$\begin{aligned} &\alpha = \frac{2\pi}{NU}(m+d-W/2) \\ &0 \le k < N/2 \\ &\alpha k = 0, 1\alpha, 2\alpha \ldots (N/2-2)\alpha, (N/2-1)\alpha \end{aligned} \tag{2.18}$$

While the arguments of the phase term of the second summation are:

$$\begin{aligned} &0 < k \leq N/2 \\ &\alpha k = -1\alpha, -2\alpha \ldots - (N/2-1)\alpha, -(N/2)\alpha \end{aligned} \tag{2.19}$$

So, excluding the terms 0 and $-N/2$ which do not have a copy, the phase terms are redundant too. For this reason the product between CC and the phase term is Hermitian and the redundancy can be exploited to reduce by half the number of products and sum required. In the following expression we exploit the property that the inverse DFT of a Hermitian vector is purely real:

$$\begin{aligned} cc'[m] =& CC[0] + \\ &+ 2 \sum_{k=1}^{N/2-1} \mathrm{Re}\left(CC[k] e^{j\frac{2\pi}{NU}(m+d-W/2)k}\right) + \\ &+ \mathrm{Re}\left(CC[N/2] e^{j\frac{2\pi}{NU}(m+d-W/2)N/2}\right),\ 0 \leq m < W \end{aligned} \tag{2.20}$$

The window is centered in the position of the peak computed in pixel units and it should be wide enough to cover the interval between the pixels adjacent to the peak, which have a smaller value. For this reason the width of the window in pixel units is 2, which corresponds to $W = 2U$ in upsampled units.

In conclusion, in terms of memory the advantage of this approach compared to the padded FFT can be huge. While in the zero-padding implementation we need to store at least one complex vector of length NU (or $NU/2$ if it exploits the Hermitian redundancy), with the DFT method we need only half of the non-upsampled complex vector with length $N/2$. Considering the computational cost, the complexity of this method is $\mathcal{O}(W \cdot N)$, which is lower than the FFT complexity $\mathcal{O}(NU \log NU)$ if the width W of the neighborhood is low enough. A rough estimation of the maximum width that ensures that the DFT approach is less complex than the padded FFT is $W < U \log NU$, which is always satisfied with W equal to $2U$ (in this application $U \geq 1$ and N is the horizontal resolution of the images). The real world advantage depends on the hardware and the implementation: for instance, if SIMD is available and the FFT code exploits it, then there is a smaller advantage in using a DFT program which does not use SIMD.

2.3.5 Sine and cosine addition formulas

When computing the FFT, the result is available as a complex vector with real and imaginary part, for this reason in the expression 2.20 the multiplication between the cross-correlation CC, available as real and imaginary part, and the phase term $e^{j\alpha k}$ can not be easily computed using the phase. Moreover, it is convenient to work with real and imaginary parts when computing the summation. For these reasons, the phase term has to be converted to real and imaginary parts before computing the multiplication:

$$\begin{aligned} \alpha =& \frac{2\pi}{NU}(m + d - W/2) \\ \mathrm{Re}\left(CC[k] e^{j\alpha k}\right) =& \mathrm{Re}(CC[k])\mathrm{Re}(e^{j\alpha k}) - \mathrm{Im}(CC[k])\mathrm{Im}(e^{j\alpha k}) = \\ =& \mathrm{Re}(CC[K])\cos(\alpha k) - \mathrm{Im}(CC[k])\sin(\alpha k) \end{aligned} \tag{2.21}$$

The cosine and the sine of the phase element have to be computed for each element in the summation. But we can observe that the increment of the argument αk of the phase term is incremented by α at each iteration, so, instead of increment the argument and compute the real and imaginary part, it can be convenient to use the sine and cosine addition formulas:

$$\begin{aligned} \cos(\beta + \alpha) &= \cos(\beta)\cos(\alpha) - \sin(\beta)\sin(\alpha) \\ \sin(\beta + \alpha) &= \cos(\beta)\sin(\alpha) + \sin(\beta)\cos(\alpha) \end{aligned} \tag{2.22}$$

Since the increment α of the argument of the summation factor is constant for a certain item $cc[m]$, then the cosine and sine of the increment are constant too. The real and imaginary part of the phase terms can be expressed as:

$$\begin{aligned} \mathrm{Re}(e^{j\alpha 0}) &= \cos(0) = 1, \mathrm{Im}(e^{j\alpha 0}) = \sin(0) = 0\,,\ k = 0 \\ \mathrm{Re}(e^{j\alpha(k+1)}) &= \cos(\alpha k + \alpha) = \\ &= \cos(\alpha k)\cos(\alpha) - \sin(\alpha k)\sin(\alpha) = \\ &= \mathrm{Re}(e^{j\alpha k})\cos(\alpha) - \mathrm{Im}(e^{\alpha k})\sin(\alpha)\,,\ 0 \leq k < N/2 - 1 \\ \mathrm{Im}(e^{j\alpha(k+1)}) &= \sin(\alpha k + \alpha) = \\ &= \cos(\alpha k)\sin(\alpha) + \sin(\alpha k)\cos(\alpha) = \\ &= \mathrm{Re}(e^{\alpha k})\sin(\alpha) + \mathrm{Im}(e^{\alpha k})\cos(\alpha)\,,\ 0 \leq k < N/2 - 1 \end{aligned} \tag{2.23}$$

For each element $cc[m]$ the real and imaginary parts of the increment, $\cos(\alpha)$ and $\sin(\alpha)$, can be computed at the beginning of the loop, and, at each iteration the real and imaginary parts of the next phase factor can be computed with only two multiplications and one addition each.

One point which has to be considered when applying this approach is that the computation of the elements depends on the previous ones, which may lead to a loss of precision compared to using sine and cosine of the angle $k\alpha$ computed directly. In practice, it has been verified that, using single precision floating-point and with $N/2$ equal to 10000 (more than six times the horizontal resolution of a 4k video), the error is in the order of 10^{-5} compared to the double precision sine and cosine. The error is comparable to the one obtained when comparing single and double precision sine and cosine. Due to the previous considerations and since experimentally the results are the same when using cosine and sine and when the addition formulas are used, it has been decided to use the addition formulas in the code.

2.3.6 Upsampling the position of the window

In [2] the displacement used to shift the window, evaluated before applying the inverse DFT, is computed with an upsampling factor of 2 by applying the zero-padding technique. The idea behind this operation is that if the position of the window is known more precisely, then the width of the upsampling interval can be smaller, reducing the total cost of the algorithm. For instance, if a factor $P = 2$ is used, then the position of the peak is known with 0.5 px precision, the upsampling interval can be placed more precisely and the window can be only 1 px wide, which corresponds to 2 upsampled pixels. To be more precise, the width of the upsampling interval in upsampled units W can be defined as $W = 2U/P$, where U is the upsampling for the displacement.

This approach is a hybrid between the zero-padding and the DFT methods and it exploits the zero-padding with smaller upsampling factors, limiting the impact on memory and computational cost. The zero-padding is ignored when computing the DFT because the zeros do not have any effect on the result and would only increment the number of operations required.

Given the cross-correlation vector of length N, the cost of the FFT with zero-padding P is $\mathcal{O}(NP\log(NP))$ and the complexity of the inverse DFT with upsampling U in the neighborhood of the peak is $\mathcal{O}(2UN/P)$. While the complexity of the FFT increases with P, the cost of the DFT decreases. As a result, as shown in Figure 2.21, the sum of the two produces a curve with a minimum for $P \sim 4$ (depending on U and N) and the largest decrement in complexity is between $P = 1$ and $P = 2$. Moreover, the advantage of this method is greater in case of larger U. Therefore, this solution is used only with small values of P (2 or 4) and with $U \geq 32$. The performance of the real program will be discussed in Chapter 5.

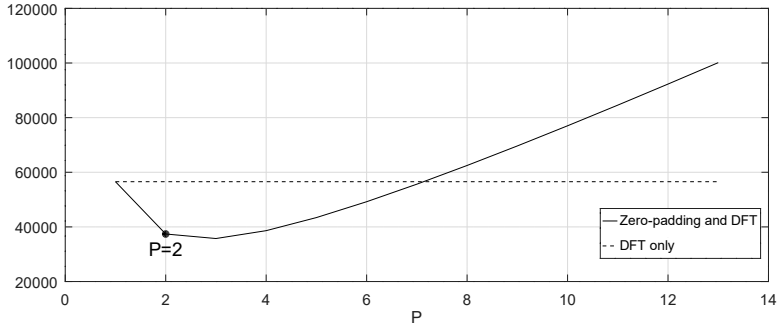

Figure 2.21: Comparison of the complexity of the zero-padding & DFT method and of the DFT approach. Resolution 800×600 and upsampling 32

Chapter 3

Implementation

In this section the C program implementing the algorithm will be discussed. The target device is any computer or board running Linux OS that supports USB webcams or other video devices (Section 3.6.2). The boards used for testing are the Terasic DE1-SoC board (ARM Cortex A9 dual core 800 MHz, NEON SIMD and 1 GB DDR3 RAM) and the Raspberry Pi 1 model B (ARM11 single core 700 MHz, VFP floating point unit and 512 MB RAM). The camera is the DBPOWER 5MP 300X USB Digital Microscope (YUYV422 and MJPEG formats, resolutions from 640x480 to 2048x1536 and frame rate up to 30 fps).

3.1 Overview

The code uses the solution proposed in Chapter 2 with the upsampling done in the neighborhood of the peak (Section 2.3.4), the sine and cosine addition formulas (Section 2.3.5) and the upsampling of the window position (Section 2.3.6).

The goal of the program is to continuously capture new frames, compute their position with respect to the first captured image and return it on stdout I/O stream or serial interface (Section 3.7). As shown in Figure 3.1, the program is divided in three main phases: set up, main loop and cleanup.

- In the set up phase the camera device is configured and the capture is started, the structures and the vectors used to compute the FFTs are initialized and the SIGINT handler used to stop the program is created. Once the system is ready, then the first reference frame is captured and it is converted into the vector `vect` computing the sum of the columns. The first 20-30 frames are discarded, allowing the camera to adjust its brightness to the current scene before saving the images. When the vector of the frame is available, it is processed to compute some parameters: `pix_to_mm` is the pixel to millimeter conversion factor (Section 3.5.6), `deltath` is the threshold which is compared with the displacement to choose whether the reference frame should be updated or not, and `deltarange` is the maximum allowed displacement (Section 2.2.6). In the end, the FFT of the reference vector `vect` is computed and saved in the complex vector pointed by `ref` and the position of the reference frame `ref_pos` is set to 0.

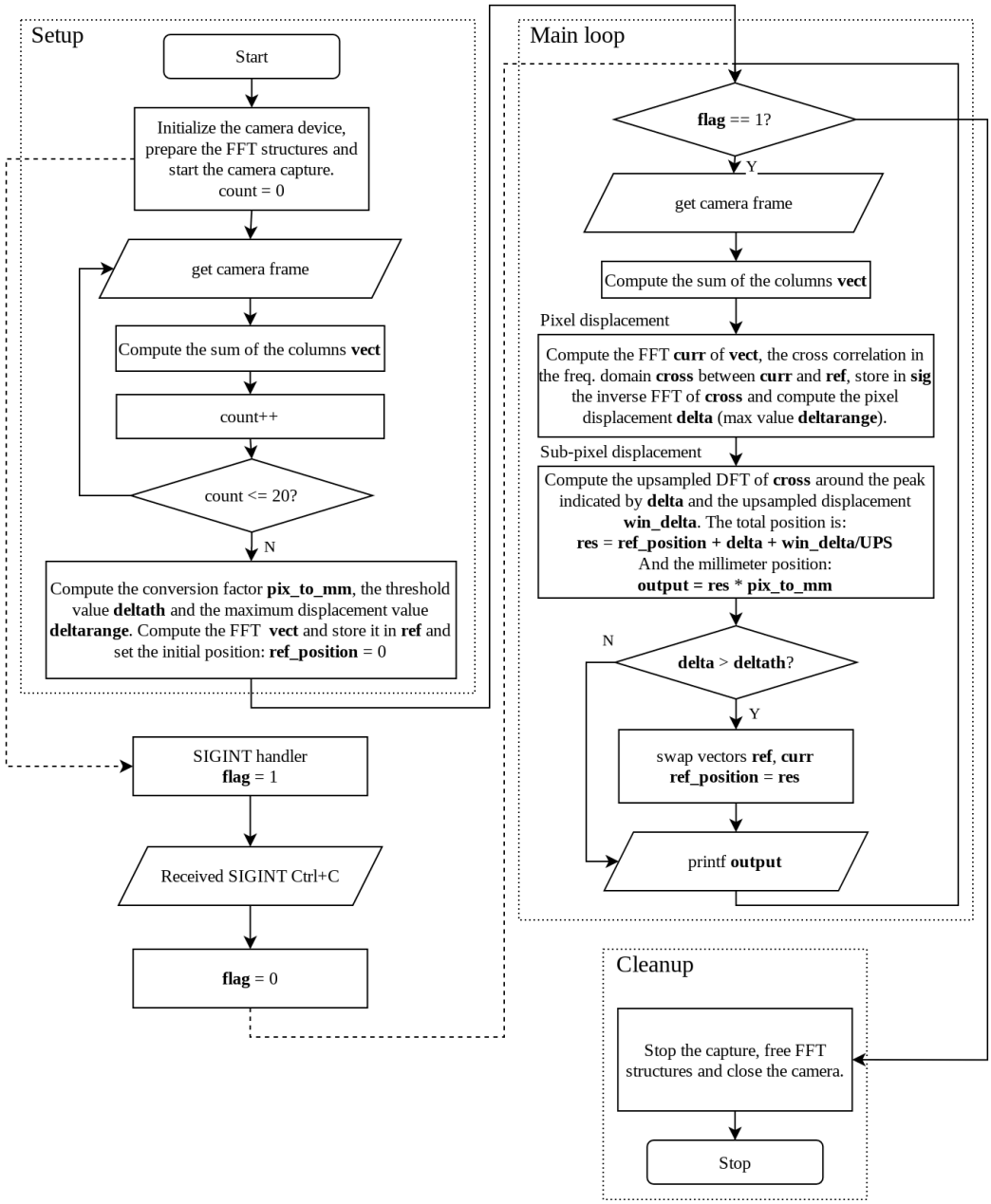

Figure 3.1: Flow chart of the main of the program

- The main loop is the core of the program: it is an endless loop in which a new frame is captured and its position is computed. First of all, a new frame is captured, it is converted to the vector `vect` summing the columns and the FFT of the vector is computed and stored in the complex vector pointed by `curr`. Then the complex cross-correlation in the frequency domain `cross` between the vector of the new frame `curr` and the reference vector `ref` is evaluated and the inverse FFT of the correlation

is evaluated an stored in the real vector `sig`. The position of the maximum of the spatial cross correlation `sig` with respect to the center is the displacement in pixel units `delta`. The maximum value of the displacement is `deltarange`, larger movements are not measured correctly. With the pixel displacement available, the upsampled DFT is computed in the neighborhood of the peak to obtain the interpolated displacement `delta_win`. The position `res` of the frame is the sum of the position of the reference vector `ref_pos`, the pixel displacement `delta` and the upsampled displacement converted to pixels `win_delta/UPS`, where `UPS` is the upsampling factor. The position is converted to millimeters as `output` = `res`$\cdot$`pix_to_mm`. Next, the coarse displacement `delta` is compared with the threshold `deltath` and, if the displacement is larger, the position of the reference frame `ref_pos` is updated to the position of the frame `ref` and the FFT vector is updated swapping the pointers `curr` and `ref`. In the end the position of the frame is written to stdout or to serial and a new loop is started.

- Lastly, the cleanup phase is reached when `SIGINT` is received (e.g. user input `Ctrl+C`). The signal handler sets the running flag to 0, stopping the execution of the main loop. The cleanup is in charge to stop the capture, close the camera device and free the allocated memory.

3.2 Linux requirements

The program requires the installation of several libraries:

- FFTW is a C library for computing the DFT of real and complex vectors of arbitrary length. [4]
- Video4Linux2 (V4L2) is a collection of drivers and API for supporting USB webcams on Linux systems. [5]
- libjpeg-turbo is a JPEG image codec, which is used to decompress the frames captured in MJPEG format. [6]

On Debian or Ubuntu systems, the required libraries and headers can be installed with the command:

```
# apt install libv4l-dev libfftw3-dev libjpeg-turbo8-dev
```

Depending on the Linux flavour, the user executing the program has to be in the `video` group to access the camera device. Moreover, if the camera is not working, verify that V4L2 devices are enabled in the kernel.

If the serial output is enabled in the program, make sure that the user running the program is in the `dialout` group and that the kernel and the bootloader are not using that port as a serial console.

3.3 Compilation

The code is compiled using `make` and the same makefile is used to compile the program on different architectures (x86, armv6, armv7), but it does not support cross compilation.

The workflow used when testing the code on the target device is to login in the target machine using `ssh`, copy the files with `scp` and compile the program with `make`.

The defined make targets are:

- `make sensor`: compile the main program without extra features. This is the default target.
- `make debug`: compile the program with extra debug messages and writing to file different useful vectors (e.g. the vector of the frames, the cross-correlation in spacial and frequency domains). It allows manipulating and visualizing the data with GNU Octave or other numerical computation languages.
- `make file`: build the program which reads the frames from a named pipe instead of capturing them with the camera. On the other side of the FIFO a Python program is in charge to read the frames of a video and to write them in the pipe. This method is useful for testing purposes and it is used when verifying the correct behavior with the 3D printer because it allows capturing a video once and then verify the displacements with different configurations using the same frames. The testing are discussed in Chapter 4.
- `make bench`: compile the program which is used to compare the performance with different configurations. Before exiting, the program prints to a file the timing information of different parts of the code in an analyzable and readable format. Moreover, in this mode some configuration parameters have to be defined as command line arguments at compilation time, which allows writing a script to build and run the program multiple times with different settings. This benchmarking program and results are be analyzed in Chapter 5.

The makefile defines some switches passed to `gcc` to optimize the compiled code:

```
CFLAGS = -O2 -ftree-vectorize -funroll-loops
```

In which `-ftree-vectorize` tells the compiler to exploit the SIMD hardware, if available, and `-funroll-loops` requires to unroll the loops. To correctly use the auto-vectorization function the hardware has to be defined. In many cases it is sufficient to add the flag `-march=native`, while in others the target has to be specified manually. The full set of arguments for the DE1-SoC board is:

```
CFLAGS = -O2 -ftree-vectorize -funroll-loops -mcpu=cortex-a9 \
       -mfloat-abi=hard -mfpu=neon
```

Where `-mcpu=cortex-a9` specifies the processor family, `-mfloat-abi=hard` tells the compiler to floating-point instructions and `-mfpu=neon` specifies the SIMD extension.

3.4 Files organization

The code is organized in different source files to improve code readability and reusability. The full source code is available as a compressed attachment. The main file (`sensor.c`) as well as the file containing the functions related to the Fourier transforms and to the

displacement computation (`fft.c`) are available in the appendix. This section wants to be a quick reference to the files and the functions constituting the code, while the implementation is discussed details in the following sections and commented in the code itself.

config.h User configuration file. Define and undefine macros to tune the behavior of the program (e.g. camera resolution and format). A list of the available macros follows:

- `UPS`: integer, upsampling factor.
- `WISDOM`: string, wisdom file location and file name. The suffix `_RESX_RESY_ZPUPS.txt`, with the actual values of the macros, will be appended to `WISDOM`.
- `DEVICE`: string, V4L2 camera device (e.g. `"/dev/video0"`).
- `FORMAT`: macro, format used for the camera capture. Supported formats are MJPEG and YUV422 (YUYV).
- `RESX`: integer, horizontal resolution of the camera capture.
- `RESY`: integer, vertical resolution of the camera capture.
- `STDOUT`: if defined the position value is printed to stdout.
- `SERIAL`: if defined write the position value to the UART interface.
- `SERIALDEV`: string, UART device name (e.g. `"/dev/ttyS0"`).
- `SERIALSPEED`: macro, UART interface baud rate (e.g. `B115200`). The allowed values are constants, refer to Section 3.7.
- `SERIALFORMAT`: macro, set the format of the transmitted data. The allowed values are `CHAR` to send the float data as a string or `RAW` to send the float value as binary data (4 bytes).
- `TRIPLETWIDTH`: float, width of one set of bands in millimeters (3 white and 3 blacks), used to compute the conversion factor from pixel to millimeter (Section 3.5.6).
- `PIXTOMMOUT`: if defined the position value is converted to millimeters using the computed conversion factor.
- `PIXTOMMCAL`: float, if defined its value is used as the conversion factor from pixel to millimeter. `PIXTOMMOUT` must be undefined.
- `THREAD`: if defined enable the thread that constantly updates the camera buffers (Section 3.6.4).

common.h Defines the macros used in different parts of the program. This file is not intended to be edited by the user.

- `ZPUPS`: zero-padding factor used to upsample the coarse-position, which is the position of the upsampling window. If the upsampling `UPS > 32` then $\texttt{ZPUPS} = 2$, otherwise $\texttt{ZPUPS} = 1$ (no interpolation).
- `WIN`: width of the inverse DFT upsampling window, defined in upsampled pixels. If $\texttt{ZPUPS} = 1$ then $\texttt{WIN} = 2 \cdot \texttt{UPS}$, if $\texttt{ZPUPS} = 2$ then $\texttt{WIN} = \texttt{UPS}$.
- `BUFN`: number of frame buffers used by the camera device driver V4L2 (Section 3.6.3).

- `SUMROW`: number of rows at the top of the image which are used to compute the sum of columns vector. It is defined as `RESY` $>> 3 =$ `RESY`$/8$.

sensor.c Contains the main function in which the three phases described in the previous Section 3.1 are implemented calling the functions defined in the other files. The flow chart of the main with the used function calls is represented in Figure 3.2.

- `int main()`: main function.

fft.c Defines the functions related to FFT, DFT, cross-correlation and displacement. The functions to set up and compute the FFT are wrappers around FFTW (Section 3.5.2).

- `float * fft_malloc_r(unsigned int n)`: allocate the memory for a real vector of `n` elements, making sure the data is properly aligned for FFTW to use the SIMD extension.
- `fftwf_complex * fft_malloc_c(unsigned int n)`: allocate the memory for a complex vector of `n` elements, making sure the data is properly aligned for FFTW to use the SIMD extension.
- `void fft_free(void ** v)`: deallocate previously allocated vector `v`.
- `fftwf_plan fft_init_fw(float *in, fftwf_complex *out)`:
  initialize the FFTW plan for the FFT of the vector of the frames. It requires the input vector `in` and the output vector `out` as parameters.
- `fftwf_plan fft_init_bw(fftwf_complex *in, float *out)`:
  initialize the FFTW plan for the inverse FFT to compute the spatial cross-correlation. It requires the input vector `in` and the output vector `out` as parameters.
- `void fft_execute_fw(fftwf_plan p, float *in, fftwf_complex *out)`:
  execute the FFTW plan `p` to compute the FFT of the frame vectors. It requires to specify the input vector `in` and the output vector `out`.
- `void fft_execute_bw(fftwf_plan p)`: execute the FFTW plan `p` of the inverse FFT to compute the spatial cross-correlation.
- `void fft_destroy(fftwf_plan *p)`: destroy the FFTW plan `p`.
- `int fft_read_wisdom()`: read the FFTW wisdom file, with filename defined by `WISDOM` in `config.h`.
- `int fft_save_wisdom()`: write the FFTW wisdom file, filename defined by `WISDOM` in `config.h`.
- `void fft_xcorr(fftwf_complex *v1, fftwf_complex *v2, fftwf_complex *out)`:
  compute the cross-correlation between the vectors `v1` and `v2` in the frequency domain. The correlation is written in the vector pointed by `out`.
- `int fft_displacement(float *in, unsigned int deltarange, float *index)`:
  compute the pixel displacement given the spatial cross-correlation `in` and the maximum allowed displacement `deltarange`. The displacement is written in the `index` and the functions returns 0 if successful and 1 in case of error.

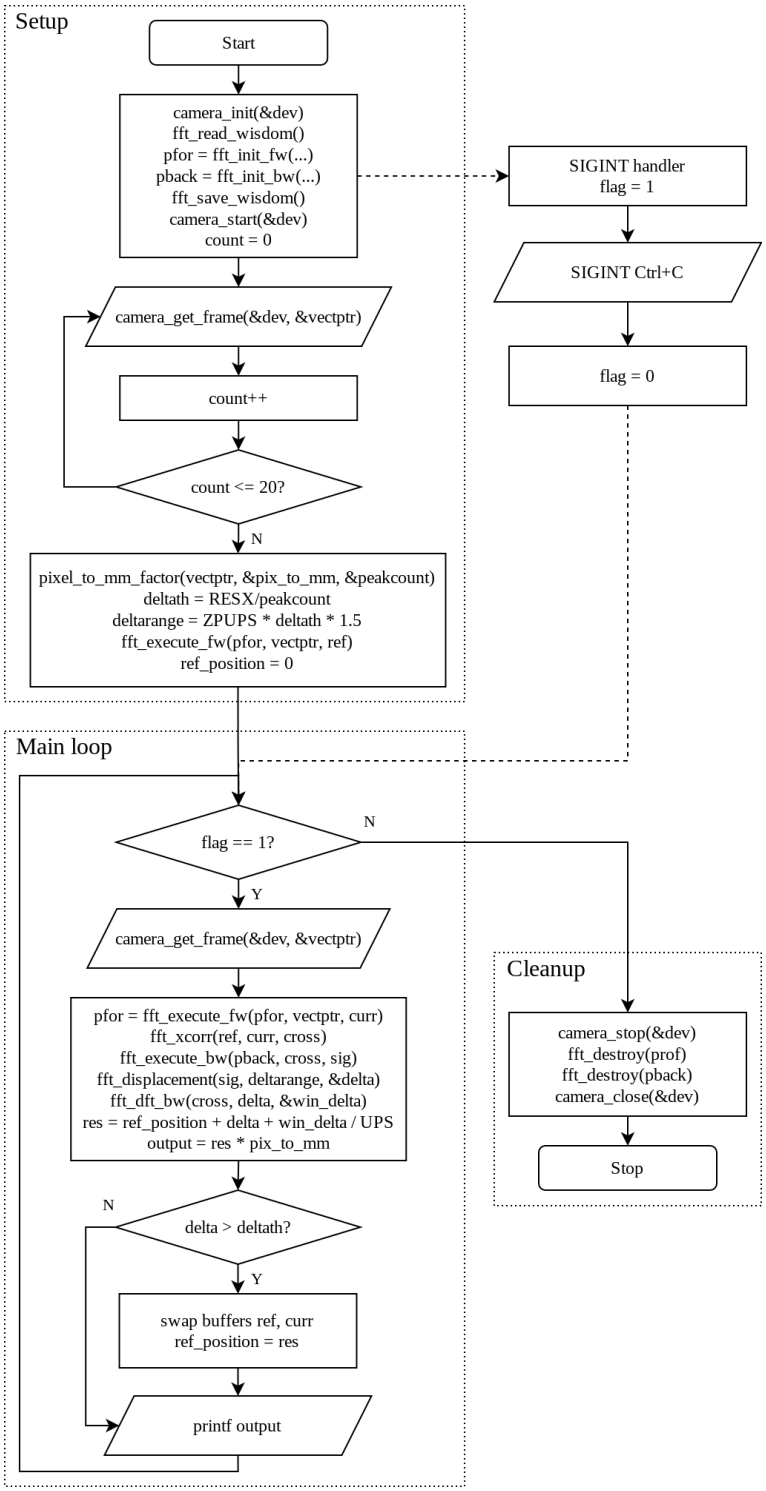

Figure 3.2: Flow chart of the main of the program and the function calls

- `int fft_dft_bw(fftwf_complex *cross, float delta, float *win_delta)`:
  compute the sub-pixel displacement `win_delta` using the upsampled inverse DFT of the cross-correlation in the frequency domain `cross`, computed around the peak indicated by the coarse displacement `cross`.

camera.c Contains the functions used to set up the V4L2 camera device, capture the frames, decompress them and compute the sum of the columns into a vector (Section 3.6).

- `struct camera_dev_t`: structure containing the file descriptor of the camera device (defined in `config.h`), information about the frame buffers used by V4L2 and other variables used by the threaded version of the program.
- `int camera_init(struct camera_dev_t *dev)`: open and configure the V4L2 camera device. It returns 0 if successful and 1 in case of error.
- `int camera_start(struct camera_dev_t *dev)`: start the capture of the frames. In the threaded version starts the thread in charge to update the buffers. It returns 0 if successful and 1 in case of error.
- `void camera_stop(struct camera_dev_t *dev)`: stop the capture. In the threaded version stop the thread, too.
- `void camera_close(struct camera_dev_t *dev)`: close the camera device.
- `int camera_get_frame(struct camera_dev_t *dev, float **vectptr)`: get the vector corresponding of the new frame. The sum of the columns is computed by this function, as well as the jpeg decompression in case of MJPEG capture format. The vector is returned via the parameter `vectptr`. The function returns 0 if successful and 1 in case of error.

serial.c Functions to set up and write to the UART serial output (Section 3.7). The device is configured using `termios.h`.

- `int serial_open(int *fd)`: open and configure the UART serial device. The device name is defined by `SERIALDEV` in `config.h`. The file descriptor is written in `fd` and the function returns 0 if successful or 1 in case of error.
- `void serial_close(int fd)`: close the file descriptor `fd` of the serial device.
- `int serial_write(int fd, float data)`: write the position `data` to the file descriptor `fd` of the serial device. The function returns 0 if successful or 1 in case of error.

misc.c Miscellaneous functions.

- `int pixel_to_mm_factor(float * vect, float * fact, unsigned int * peakcount)`
  compute the pixel to millimeter conversion factor `fact` (Section 3.5.6) and the number of peaks `peakcount` in the frame vector `vect`, used to compute the maximum displacement (Section 2.2.6). The function returns 0 if successful and 1 in case of error.

- `void threshold_apply(float * vect)`: Set the black bands to zero, as discussed in Section 2.2.5. The points of the input frame vector `vect` with value below 50% of the amplitude are set to zero (Section 3.6.5).

benchmark.c Defines the functions used to open and write the timings of different parts of the program, only when the macro `BENCHMARK` is set. This is used when comparing the performance with different settings and on different hardware (Chapter 5).

camera_file.c The frames are read from a named pipe instead of being captured with the webcam. It is used for testing purposes reading the frames from a video file, received from a Python program (Chapter 4). This file redefines the function of `camera.c` to maintain compatibility with the rest of the program.

3.5 Displacement computation

In this section the implementation of the upsampled cross-correlation is described. The code uses the solution proposed in Chapter 2 with the upsampling done in the neighborhood of the peak (Section 2.3.4), the sine and cosine addition formulas (Section 2.3.5) and the upsampling of the window position (Section 2.3.6).

3.5.1 Upsampled window position

The code implements the upsampling of the position of the interpolating window using the zero-padding technique, as discussed in Section 2.3.6. The zero-padding factor `ZPUPS` is defined as a constant at compile time comparing the value of the upsampling `UPS` with an arbitrary value:

Listing 3.1: `common.h`

```
154 #if UPS < 64
155 # define ZPUPS 1
156 #else
157 # define ZPUPS 2
158 #endif
```

When `ZPUPS` is 1, then the zero padding is not applied and the position of the window is defined in pixel units. If `ZPUPS` is 2 the padding is applied: the cross-correlation vector is extended to double the size and the position of the interval is computed with 0.5 px precision. As a result the complex arrays of the correlation in the frequency domain and the real spatial cross-correlation see their size doubled, the former from $\texttt{RESX}/2 + 1$ to $\texttt{RESX} \cdot \texttt{ZPUPS}/2 + 1$ and the latter from `RESX` to $\texttt{RESX} \cdot \texttt{ZPUPS}$. The zero-padding is applied by the function `fft_xcorr` after the cross-correlation is computed, as further discussed in Section 3.5.3.

The threshold to decide whether to apply the padding or not is set to 32 and it has been obtained observing the benchmark results, as shown in Section 5.3.2.

3.5.2 FFTW

To compute the FFT of the vectors, the program relies on FFTW, a C library for computing DFT of arbitrary input size of both real and complex data. The most interesting aspects of the library is that it works on arbitrary input size vectors with complexity $\mathcal{O}(N \log N)$, it supports SIMD extensions (SSE, Arm Neon), it provides optimized functions for real input or output data and it is free software. [4]

In this application it is used the complex type `fftwf_complex` defined by this library as float[2], composed of the real part (e.g. `val[0]`) and the imaginary part (e.g. `val[1]`).

FFTW does not use a fixed algorithm for computing the FFT, instead it tries to maximize the performance adapting the DFT algorithm to the hardware. For this reason the computation is divided in two phases: first the FFTW planner looks for the fastest way to compute the FFT on the device, which results in the creation of a plan data structure. Then the plan is executed to compute the FFT of the input as decided in the plan. The plan can be reused many times for different transforms of the same size, while a new plan is needed for FFT on vectors with different length. FFTW provides fast planners based on heuristics (`FFTW_ESTIMATE` when preparing the plan), which produces sub-optimal plans, and slow planners, that find an optimized plan computing several FFTs and measuring their execution time (`FFTW_MEASURE` or the more complete and slower `FFTW_PATIENT`). In the code the slow planner is used because only two different plans are needed and they are computed at set up, hence the plan creation does not affect the performance of the sensor. It should be considered that the plan creation may take up to several minutes in case of large plans on slow hardware.

Since the creation of new plans can be expensive, FFTW provides the wisdom mechanism, which is the ability to save and restore the plans which have been computed for FFTs of the same length. The wisdom file allows computing the plan once, store the obtained results and read them when the program is executed next. If the wisdom file does not exist or it is computed for the wrong array size, then the plans are evaluated normally. The wisdom file is defined as `WISDOM` in `config.h`

The functions used to save and restore the wisdom are wrappers around FFTW functions:

Listing 3.2: `config.h`

```
25 #define WISDOM "./wisdom"
```

Listing 3.3: `fft.c`

```
365 int fft_read_wisdom() {
366   return fftwf_import_wisdom_from_filename(WISDOM);
367 }

372 int fft_save_wisdom() {
373   return fftwf_export_wisdom_to_filename(WISDOM);
374 }
```

FFTW also provides the functions to create plans for real input or output data exploiting the Hermitian redundancy, which allows achieving a factor of two improvement in both speed and memory usage when executing the plan. These faster functions can be used to

compute both the forward FFT, which uses the real vector of the frame as input data, and the inverse FFT, since its input is Hermitian. Moreover, the output data of a FFT with a real input vector `RESX` long is a complex vector of length `RESX/2+1` because FFTW exploits the redundancy when storing the vectors. While the forward FFT is not affected by the factor `ZPUPS`, the size of the inverse FFT doubles when `ZPUPS` = 2, increasing the plan preparation time, doubling the memory required (`ZPUPS·RESX` vectors) and increasing the computational cost. The functions used to create the plans are:

Listing 3.4: `common.h`

```
254 #define ZPRU ((unsigned int)((unsigned int)RESX * (unsigned int)ZPUPS
        ))
```

Listing 3.5: `fft.c`

```
 41 fftwf_plan fft_init_fw(float *in, fftwf_complex *out) {
 42   return fftwf_plan_dft_r2c_1d(RESX, in, out, FFTW_PATIENT);
 43 }

 55 fftwf_plan fft_init_bw(fftwf_complex *in, float *out) {
 56   fftwf_plan plan;
 57   plan = fftwf_plan_dft_c2r_1d(ZPRU, in, out, FFTW_PATIENT |
          FFTW_PRESERVE_INPUT);

 68   return plan;
 69 }
```

The plans are executed with:

Listing 3.6: `fft.c`

```
214 void fft_execute_fw(fftwf_plan p, float *in, fftwf_complex *out) {
215   fftwf_execute_dft_r2c(p, in, out);
216 }

207 void fft_execute_bw(fftwf_plan p) {
208   fftwf_execute(p);
209 }
```

While `fft_execute_fw`, which computes the forward FFT, requires to specify the input and output vectors because it is applied to the reference frame and to the frames captured in the loop, the function `fft_execute_bw`, computing the inverse FFT, does not because it is always applied to the same vectors which are defined when the plan is created.

The library FFTW is available both in single and double precision versions. Since in this application it has been verified that `float` and `double` produce the same position results, then the single precision library has been used. To enable the single version of the library all the instances of `fftw_` have to be replaced with `fftwf_` and `double` has to be substituted with `float`.

Lastly, FFTW supports SIMD extensions, including ARM Neon used by the DE1-SoC board, which produces a speedup for most transforms. In order to use SIMD the

arrays must be specially aligned in memory. To guarantee proper alignment, both real and complex vectors must be allocated using `fftw_malloc`.

Listing 3.7: `fft.c`

```
15 float * fft_malloc_r(unsigned int n) {
16   return (float *) fftwf_malloc(sizeof(float) * n);
17 }

21 fftwf_complex * fft_malloc_c(unsigned int n) {
22   return (fftwf_complex *) fftwf_malloc(sizeof(fftwf_complex) * n);
23 }
```

3.5.3 Cross-correlation in the frequency domain and zero-padding

The cross-correlation in the frequency domain is computed as the product element by element of the transform of the vector of the two frames, with one of them conjugated. Since both the input vectors are Hermitian, then the cross-correlation is Hermitian too and, to reduce the operations required, only half of the vector is evaluated and stored.

After the correlation is stored, if `ZPUPS > 1`, the zero-padding is applied to the vector. The correlation vector is evaluated with the positive frequencies $[0; \texttt{RESX}/2-1]$ at the beginning of the vector and the frequency $-\texttt{RESX}/2$ in position $\texttt{RESX}/2$, at the end of the vector. Therefore, the padding is obtained moving the element $\texttt{RESX}/2$ to the end of the padded vector (position $\texttt{RESX} \cdot \texttt{ZPUPS}/2$) and writing zeros in positions $[\texttt{RESX}/2; \texttt{RESX} \cdot \texttt{ZPUPS}/2]$. Since the inverse FFT computed by FFTW preserves the input vector (`FFTW_PRESERVE_INPUT` flag is set when creating the plan) and writes only the first `RESX+1` positions, then the zeros can be written once when preparing the vectors during the set up phase, instead of writing them at each iteration. The zeros are written to the vector after the inverse FFT plan is created in `fft_init_bw`.

The function which executes the frequency correlation is `fft_xcorr` and takes as inputs the pointers to the FFT of the two frames and the pointer to the vector where the cross-correlation has to be stored. Defining $\texttt{RESX2} = \texttt{RESX}/2$ and $\texttt{ZPRU2} = \texttt{RESX} \cdot \texttt{ZPUPS}/2$, the correlation and the zero-padding are computed as follows:

Listing 3.8: `fft.c`

```
78 void fft_xcorr(fftwf_complex *v1, fftwf_complex *v2, fftwf_complex *
       out) {
79   unsigned int i;
80
81   // Compute the cross correlation of the first half of the vector
82   // The length of the vector is RESX/2+1
83   for (i = 0; i < RESX2 + 1; i++) {
84     // Real part
85     out[i][0] = v1[i][0] * v2[i][0] + v1[i][1] * v2[i][1];
86     // Imaginary part
87     out[i][1] = v1[i][1] * v2[i][0] - v1[i][0] * v2[i][1];
88   }
```

```
89
90  #if (ZPUPS > 1)
91    // Zero padding, the vector is preserved, no need to write all the
          zeros at
92    // each iteration
93    out[ZPRU2][0] = out[RESX2][0];
94    out[ZPRU2][1] = out[RESX2][1];
95    out[RESX2][0] = 0;
96    out[RESX2][1] = 0;
97  #endif
98  }
```

3.5.4 Coarse displacement

The coarse displacement is used to identify the position of the maximum with pixel accuracy, or half pixel if `ZPUPS` is 2, which is needed both to compute the displacement and to center the upsampling window in the peak. The inverse FFT is applied to the cross-correlation and the spatial vector is obtained. The pixel displacement is computed as the position of the maximum of the spatial vector with respect to the center. The function computing the pixel displacement is:

Listing 3.9: `fft.c`

```
118  int fft_displacement(float *in, unsigned int deltarange, float *index
         ) {
```

The function requires as input parameters the spatial vector `in` and the maximum displacement `deltarange`, and as output parameter the pointer where the computed displacement has to be saved `index`.

When searching the maximum value, it can happen that there are multiple points with the same peak value. In that case there are two possible scenarios: if there are only two peaks with the same value and they are adjacent, then the displacement is computed as the average of the two positions. If there are more than two equal peaks or they are far from each other, then the position can not be evaluated. The latter case usually occurs when the lighting of the tag is not adequate, the frame is not correctly focused or the movement of the tag is too fast for the camera shutter.

The maximum displacement is limited to the value defined by `deltarange` in the main, which corresponds to the width of the frame divided by the number of framed white bands and multiplied by 1.3, as discussed in Section 2.2.6, multiplied by the factor `ZPUPS`. Therefore, defining the length of the vector $\texttt{ZPRU} = \texttt{ZPUPS} \cdot \texttt{RESX}$, the maximum is only searched in the range $\texttt{in}[\texttt{ZPRU}/2 - \texttt{deltarange}, \texttt{ZPRU}/2 + \texttt{deltarange}]$, which corresponds to the displacements $[-\texttt{deltarange}, \texttt{deltarange}]$.

The spatial cross-correlation obtained with the inverse FFT has the first and the second half swapped: the element which should be in position 0 is in $\texttt{ZPRU}/2$ and the other way around. Instead of reordering the vector, which is expensive, it is sufficient to search for the position of the maximum iterating first the elements $[\texttt{ZPRU} - \texttt{deltarange}, \texttt{ZPRU} - 1]$, then $[0, \texttt{deltarange}]$.

If ZPUPS = 2 then the obtained displacement is converted from half pixels to pixels in the main before applying the inverse DFT:

Listing 3.10: sensor.c

```
312         delta = delta / (float) ZPUPS;
```

3.5.5 Inverse DFT interpolation

Once the pixel displacement is available, the upsampled inverse DFT around the peak can be computed (Section 2.3.4) using the trigonometric addition formulas (Section 2.3.5). The function computing the inverse DFT and sub-pixel displacement is:

Listing 3.11: fft.c

```
231  int fft_dft_bw(fftwf_complex *cross, float delta, float * win_delta)
         {
```

The input parameters are the pointer to the correlation in the frequency domain `cross` and the coarse displacement used to center the upsampling window in the peak `delta`, while the computed displacement is written in `win_delta`.

The upsampled spatial cross-correlation vector is evaluated using the Equation 2.20, which is implemented with two loops: the outer one iterates the elements of the output spatial cross-correlation, while the inner loop iterates the elements of the input correlation in the frequency domain (`cross` parameter) and the phase factors (`phase_term` variable), computing their product. Since in this case we are only interested in the displacement and not in saving the correlation vector, the position of the maximum value is searched while iterating the outer loop, without storing the computed elements in a vector.

The size of the interval where the upsampled cross-correlation is computed is defined by the macro `WIN` and corresponds to the number of iterations of the outer loop. The width `WIN` depends on the value of the zero-padding factor `ZPUPS` used to compute the position of the window. When `ZPUPS` is larger, then the upsampling interval can be smaller because the position of the window is known with better accuracy. The upsampling window covers the interval between the points adjacent to the peak, which corresponds to 2 pixels if ZPUPS = 1 or 1 pixel if ZPUPS = 2. Therefore `WIN` is defined as:

Listing 3.12: common.h

```
171  #if ZPUPS == 1
172  # define WIN ((int)(UPS*2))
173  #elif ZPUPS == 2
174  # define WIN ((int)(UPS*1))
175  #endif
```

When computing the sum of the product of each element of the spatial correlation, `ZPUPS` is ignored because the padding zeros do not change the result of the sum.

The key elements that allow applying the shift and the upsampling while computing the inverse DFT are the phase term, as shown in Section 2.3. For each element of the inner loop, the multiplication between the complex `cross`$[k]$ and the phase term `phase_term` is computed and added to the dot product `rc_product`. The real and imaginary parts of

the phase factor are evaluated using sine and cosine addition formulas, incrementing the angle by a fixed amount `arg_inc` for each iteration of the dot product loop. The real and imaginary parts of the increment of the phase term, stored in `phase_term_inc`, are the cosine and the sine of the increment of the argument `arg_inc`. In the following Listing it is reported the section of `fft_dft_bw` corresponding to the inner loop:

Listing 3.13: `fft.c`

```
253    for (m = 0; m < WIN; m++) {

277      for (k = 0; k < RESX2 - 1; k++) {
278
279        // Phase terms computed using sine and cosine addition formulas
               ,
280        // incremented each iteration by the same amount.
281        tmp = phase_term[0] * phase_term_inc[0] -
282          phase_term[1] * phase_term_inc[1];
283        phase_term[1] = phase_term[0] * phase_term_inc[1] +
284          phase_term[1] * phase_term_inc[0];
285        phase_term[0] = tmp;
286
287        // Sum of the product between the cross correlation and the
               phase
288        // term, DFT definition
289        rc_product += (*cc)[0] * phase_term[0] - (*cc)[1] * phase_term
               [1];
290
291        // Increment cross correlation pointer
292        cc++;

300      }
```

Since both the cross-correlation and the phase terms satisfy the Hermitian condition, then the complete dot product can be computed multiplying by 2 the sum of the products of the terms $[1, \texttt{RESX}/2 - 1]$ and then adding the missing terms (0 and $-\texttt{RESX}/2$):

Listing 3.14: `fft.c`

```
302      // Exploit hermitian redundancy to compute the complete row by
             column
303      // product.
304      rc_product *= 2;
305
306      // Add the missing terms which are not replicated in the vector.
307      // Phase term -RESX/2, which is the complex conjugate of RESX/2 (
             hence
308      // the sum instead of the subtraction).
309      rc_product += cross[ZPRU2][0] * (phase_term[0] * phase_term_inc
             [0] +
```

```
310         phase_term[1] * phase_term_inc[1]);
311
312       // The dc component phase term is 1
313       rc_product += cross[0][0];
```

Once the dot product is available, it is compared with the maximum value to find the position of the peak. If there are multiple adjacent points with the same max value, the positions of first and the last items with the same max value are saved and the displacement is computed considering using the center point of the interval.

Listing 3.15: `fft.c`

```
325       if (rc_product > max) {
326         max = rc_product;
327         id = m;
328         id_end = m;
329       } else if (rc_product == max) {
330         id_end = m;
331       }
332
333     }
```

```
341     *win_delta = (float)((int)(id + id_end) - (int)WIN) * 0.5;
```

The computed displacement is stored in `win_delta`, a float which represents the position in upsampled pixels of the interpolated peak with respect to the peak evaluated in pixel units `delta`. The total position with respect to the first frame is computed in the main as the sum of the position of the reference frame, the pixel displacement and the upsampled displacement around the peak converted to pixel units:

Listing 3.16: `sensor.c`

```
325         res = ref_position + delta + win_delta / (float) UPS;
```

3.5.6 Pixel to millimeter conversion

The tag used in the testing is composed of fixed white bands of width $2B$ and variable black bands of width $2B$, $3B$ and $4B$, as shown in Section 2.2.1. The width of the bands and the distance of the camera from the strip are chosen to have always at least 6 white bands in the frame, therefore there is always at least one full set of bands (3 black bands and 3 white ones) in the image. The total width of the periodic set is $L = 15B$ and it is defined when the tag is printed, for instance $L_{mm} = 15B = 3.75\,\text{mm}$. To compute the conversion factor from pixel to millimeter, we also need the width of a set in pixel units L_{px}, which is measured counting the number of pixels between the 1st and the 7th transitions in the vector of the sum of the columns. Assume that $L_{px} = 600\,\text{px}$, then the conversion factor is:

$$C = \frac{L_{mm}}{L_{px}} = \frac{3.75\,\text{mm}}{600\,\text{px}} = 6.25\,\mu\text{m/px} \tag{3.1}$$

The automatic conversion is enabled in the code defining the `PIXTOMMOUT` macro in `config.h` and requires to set the width of the set of bands to in millimeters using `TRIPLETWIDTH` in the same file.

This method to determine the conversion factor is suitable for high resolution images or if the maximum movement of the object is limited. In case of low resolution frames there can be a larger errors in the value of C because the distance between the transitions is less accurate. In fact, the sensitivity of the conversion factor with respect to the length in pixel L_{px} is:

$$\begin{aligned} L_{mm} &= 3.75\,\text{mm} \\ \left|\frac{\partial C}{\partial L_{px}}\right| &= \frac{L_{mm}}{L_{px}^2} \\ \left|\frac{\partial C}{\partial L_{px}}\right|_{L_{px}=320} &= \frac{3.75\,\text{mm}}{(320\,\text{px})^2} = 36 \cdot 10^{-6}\frac{\text{mm}}{\text{px}^2} \\ \left|\frac{\partial C}{\partial L_{px}}\right|_{L_{px}=1600} &= \frac{3.75\,\text{mm}}{(1600\,\text{px})^2} = 1 \cdot 10^{-6}\frac{\text{mm}}{\text{px}^2} \end{aligned} \tag{3.2}$$

The sensitivity is much larger for low resolution frames than for high resolution images, as a consequence the same absolute error in the evaluation of the distance between the transitions results in a higher error in the value of the conversion factor. Moreover, since C is multiplied by the pixel position to compute the millimeter position, the error of the conversion factor is a gain error and proportional to the error of the millimeter position. Therefore, the total error of the position is larger if the movement range is wider.

Because of the limitations of the conversion factor computed on the frame, it is recommended calibrating the sensor by defining manually C whenever possible, using the macro `PIXTOMMCAL` in `config.h`. The conversion factor can be evaluated by moving the object by a known quantity and dividing the shift by the pixel displacement returned by the sensor. The drawback of the manual calibration is that it requires a machine capable of precise movements or another calibrated measurement device. The calibration has to be done each time the distance between the camera and the band changes, while, if only the resolution is modified, it is sufficient to multiply the previously computed C by a conversion factor equal to the ratio of the horizontal resolutions.

3.6 Camera

In this section it is discussed the implementation of the capture of the frames. The camera device used for testing is the DBPOWER 5MP 300X USB Digital Microscope, with capture formats MJPEG and YUYV422, resolutions from 640x480 to 2048x1536 and frame rate up to 30 fps.

3.6.1 Supported capture formats

The camera capture formats supported by the program are YUV422 and MJPEG, which are also the formats available on the USB microscope used in the experiments. YUV422 is better for this application because it does not need to be decompressed, but there where

synchronization issues when using this format with the DE1-SoC board. For this reason MJPEG had to be used, even if it requires to decompress the frames, operation which is more expensive than the sum of the columns and the computation of the displacement, as shown in the benchmarks (Section 5.3.5).

The algorithm requires grey scale images, while both YUV422 and MJPEG include the color components. For this reason, they requires extra bandwidth to transfer the unused color information and they are not ideal for this application. When choosing the camera module for this sensor, a device supporting uncompressed grey scale (`V4L2_PIX_FMT_GREY` in V4L2) should be selected.

YUV422

YUV422 is a uncompressed data format which represents each couple of pixels on 4 bytes (2 bytes per pixel). There are three type of bytes: Y represents the brightness component (also colled luma), while U and V represent the chrominance components. The order of the bytes in each frame is:

$$U_0Y_0V_0Y_0 \quad U_1Y_1V_1Y_1 \quad U_2Y_2V_2Y_2 \dots$$

In this representation each pixel has its own brightness value Y, while the chrominance bytes are shared between the pixels. Since we are interested in the grey scale image, the bytes U and V are neglected. In the resulting image each pixel is represented by its brightness on one byte Y. The conversion to grey scale is obtained while computing the sum of the columns by considering only the columns of the YUV422 frame which contain the brightness components Y (one column every two).

MJPEG

In the MJPEG format each individual frame is compressed as a JPEG image. To decompress the frames the open source libjpeg-turbo library has been used, in particular its higher level API called TurboJPEG. The libjpeg-turbo codec is a JPEG image codec that uses SIMD instructions (MMX, SSE2, AVX2, Neon) to accelerate JPEG compression and decompression. The conversion to grey scale on 8 bits is done while decompressing the image, therefore each pixel of the decompressed frame is represented by one byte and the decompressed frame is saved in a char matrix with size equal to the resolution. The sum of the columns is computed on the decompressed grey scale frame and all the columns are considered.

3.6.2 Frames capture: V4L2

Video4Linux2 (V4L2) is a collection of device drivers and API that supports camera capture on Linux devices. The V4L2 framework is part of the Linux kernel, it supports a wide range of camera devices including USB webcams and MIPI CSI modules (used by the Raspberry Pi camera modules) as well as a huge list of formats. [5]

The V4L2 devices are programmed using `ioctl` system calls: different requests, defined in V4L2 header files, allow configuring and interacting with the camera, for example queuing or dequeuing the frame buffers. V4L2 defines several methods to read the frames

from the camera. This application uses the mmap method, in which only the pointers to buffers are exchanged between application and driver, as opposed to the read method which requires the CPU to copy the data.

Due to the behavior of the buffers in V4L2 (Section 3.6.3), two solutions to request the frames to the driver are implemented in the code: a single threaded version and a version using a dedicated thread to get the frames (Section 3.6.4).

In the code, the functions related to the camera configuration are contained in `camera.c` and the main steps to configure the device and start the capture are:

- Open and set up the video device:

  43 `int camera_init(struct camera_dev_t *dev) {`

  - initialize the structure `dev`, containing information about the open camera device and its buffers;
  - open the video device (`open` call);
  - set the video capture format (`ioctl` with request `VIDIOC_S_FMT`);
  - allocate the buffers (`ioctl` with request `VIDIOC_REQBUFS`);
  - require information about the buffer (`ioctl` with `VIDIOC_QUERYBUF`);
  - map the buffers into the application address space (`mmap`).

- Start the camera stream:

  109 `int camera_start(struct camera_dev_t *dev) {`

  - put all the buffers in the queue (`ioctl` with request `VIDIOC_QBUF`);
  - start the capture (`ioctl` with request `VIDIOC_STREAMON`).

- Read one frame:

  232 `int camera_get_frame(struct camera_dev_t *dev, float **vectptr) {`

  - Wait for a buffer to be ready, then remove it from the queue to allow reading (`ioctl` with request `VIDIOC_DQBUF`);
  - Process the frame: JPEG decompression and sum of the columns reading directly from the buffer;
  - Re-queue the buffer (`ioctl` with `VIDIOC_QBUF`).
  - Returns the computed vector in `vectptr`;

- Stop the camera stream:

  185 `void camera_stop(struct camera_dev_t *dev) {`

  - stop the capture (`ioctl` with request `VIDIOC_STREAMOFF`).

- Close the video device:

```
213  void camera_close(struct camera_dev_t *dev) {
```

  - unmap the buffers (`munmap` call);
  - close the device (`close` call).

3.6.3 Buffers management

V4L2 uses buffers to exchange data between the driver and the application. Using the mmap streaming method, only the pointers to the buffers are moved, while the data itself is not copied. The number of buffers can be 1 or larger and, in this application, it is 2 for the non-threaded sensor and 4 for the threaded sensor, as explained later in this section and in Section 3.6.4. The buffer count is defined by `BUFN` in `common.h`.

As shown in Section 3.6.2, a buffer can be read by the application only after it is removed from the queue. Then, when the processing of the frame is finished, the buffer has to be re-queued so that it can be written again. A buffer can be removed only when it has been written by the camera and when a buffer is removed from the queue the camera can not write to it. When all queued buffers have been written, the camera does not overwrite them, instead it waits for a buffer to be dequeued and queued again. The critical point is that, if there are multiple buffers ready, the oldest one is returned.

In the main loop of the sensor, first, a buffer is dequeued and the sum of the columns is computed reading the frame. Next, the frame buffer is re-queued and the displacement is computed using the vector of the sum of the columns. Once the position is available, the next iteration of the sensor starts and a new buffer is requested.

If the time required to compute the vector and the displacement are shorter than the time between the frames, then the program works as intended because there are never multiple written buffers in the queue and the displacement is always computed using the latest available image. If `BUFN` is 1, then it is always the same buffer that is dequeued and re-queued. Therefore, the buffer exchange overhead and the sum of the columns time is added to the frame time and the capture frame rate, as well as the sensor sample rate, can be lower than the expected camera speed. To ensure the highest possible camera frame rate multiple buffers should be used. For instance if `BUFN` is 2, while a buffer is dequeued, processed and queued again, another buffer is already in the queue ready to be written. Larger numbers of buffers do not provide a benefit when the computation time is shorter than the frame time because only two buffers are used at the same time, one for reading and one for writing. Figure 3.3 shows the difference in using 1 or 2 frame buffers when the computation time is shorter than the frame time. In the diagrams we assume that the buffers are written as soon as they are added to the queue, but, as shown in the benchmarks (Section 5.3.6), in the real case there is additional delay.

On the other hand, if the time between the frames is smaller than the computation times, then more buffers in the queue are written while the sum of the columns and the displacement are evaluated. As a result, since the oldest frames are dequeued first, the frame which is used to compute the displacement is an old image and some newer frames are ignored or delayed. The problem can be solved by setting the frame rate of the camera to a value which ensures that the computation is faster than the frame time or

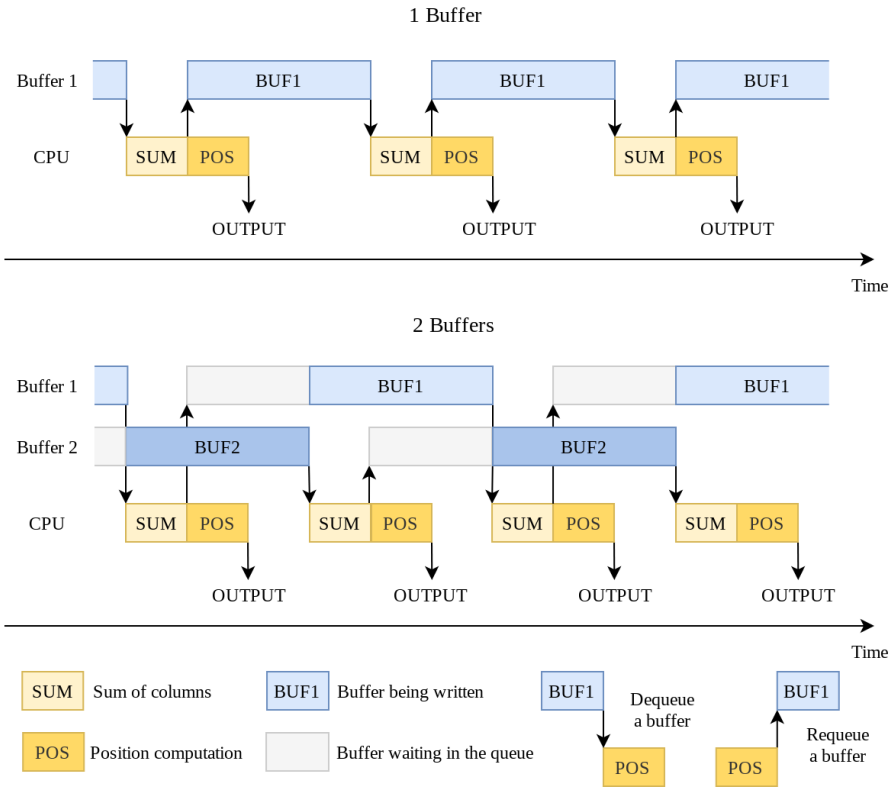

Figure 3.3: Flow diagram comparing the program behavior when using 1 frame buffer (top) or 2 frame buffers (bottom) when the computation time is shorter than the frame time

by constantly dequeuing and re-queuing the buffers. The first solution requires to set the frame rate of the camera to an arbitrary value which depends on the processor running the program. Although this solution can be not optimal, since usually only certain frame rates are allowed, it is the suggested one on low-performance single core systems because the required processing is not increased. Since it is not always possible to set the frame rate to an arbitrary value, then the solution which constantly refreshes the buffers can result in a higher sample rate. The continuous updating of the buffers can be implemented using a thread, as discussed in Section 3.6.4. This solution is suggested when more memory and cores are available since more buffers are required and there is an additional processing overhead due to the thread synchronization. Moreover, one core is constantly requesting new frames and the buffers are continuously written, even when they are not needed. As a result a larger power usage can be expected. The timing diagram of the different solutions is shown in Figure 3.4.

3.6.4 Buffers update thread

The code implements a thread which is in charge to continuously queue and de-queue the V4L2 buffers to ensure that the frames read by the main thread to compute the displacement are the most recent ones. The thread can be enabled or disabled using

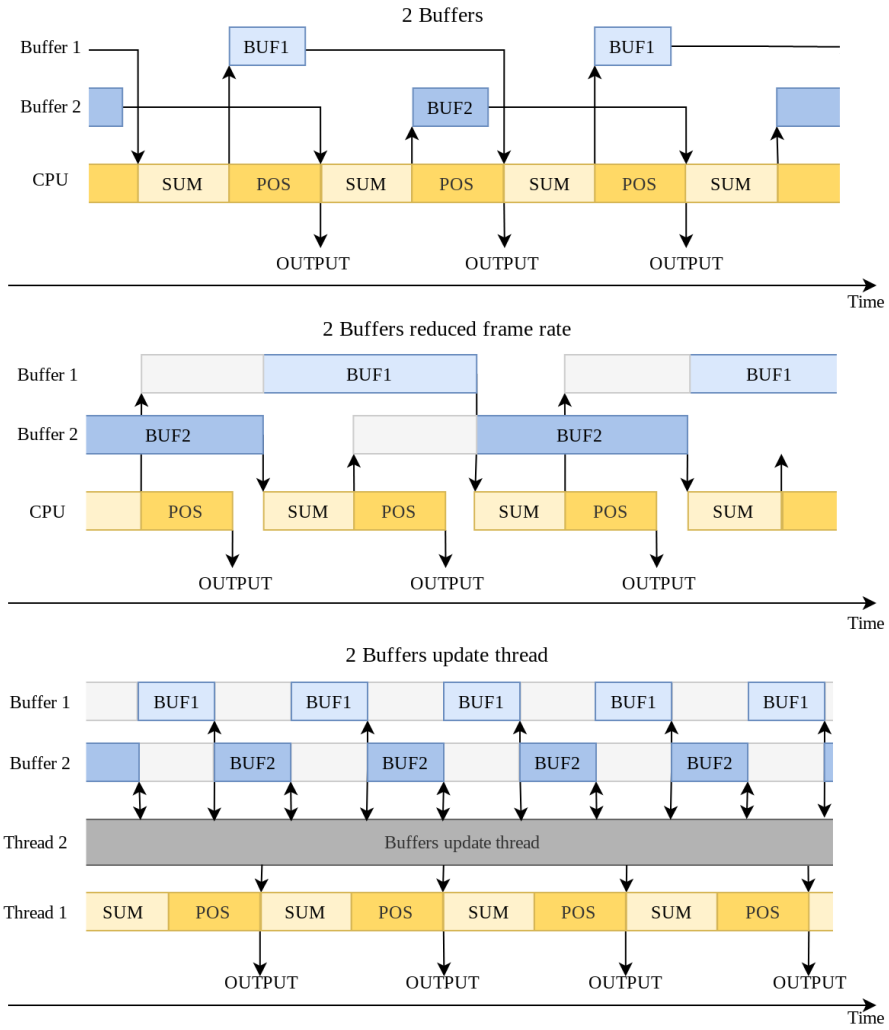

Figure 3.4: Flow diagram comparing the program behavior when the computation time is longer than the frame time using 2 frame buffer (top), 2 frame buffers and reducing the frame rate (middle) and 2 frame buffers with a thread constantly refreshing the frames (bottom).

the `THREAD` macro defined in `config.h` and should be enabled on multi-core processors if the processing time (sum of columns, displacement computation and, if required, frame decompression) is longer than the frame time. This is the case when using a combination of high resolutions, large upsampling and fast camera frame rate. On the contrary, when the processing is faster than the frame time, the thread can be safely disabled. The routine invoked by the buffer thread is:

Listing 3.17: `camera.c`

```
288  static void* camera_buffers_thread(void *devp) {
```

The goal of the thread is to make the most recent frame always available to the main thread.

This implementation is designed to work with 4 buffers in total, as defined by `BUFN`: two buffers are in the queue, one buffer stores the newest frame and it is shared between the main and the update threads, and one buffer stores the frame used by the main thread for the computation. When a new frame is available, the thread updates the shared pointer to the newest buffer. When the main thread requires a frame, the new updated pointer and the old one, which has already been used to compute the displacement, are swapped. If a frame is requested but a new one is not available yet, the main thread waits for a new buffer to be available. The ready buffer in the queue is immediately replaced by the update thread, therefore it is guaranteed that there is always a writable buffer in the queue.

The pseudocode of the routine `camera_buffer_thread`, called by the buffer updating thread, and the pseudocode of the function `camera_get_frame`, called by the main thread to request a new frame, are reported in the following listing:

Algorithm 1: `camera_buffer_thread`

```
1: while running do
2:     newbuf = dequeue buffer
3:     mutex lock (mutex)
4:     queue buffer (shbuf)
5:     swap pointers (newbuf, shbuf)
6:     mutex unlock (mutex_sync)
7:     mutex unlock (mutex)
8: end while
```

Algorithm 2: `camera_get_frame`

```
1: mutex lock (mutex_sync)
2: mutex lock (mutex)
3: swap pointers (currbuf, shbuf)
4: mutex unlock (mutex)
5: process frame ()
```

This implementation allows to update the newest frame and make it available to the main thread without moving the frames data, in fact only the pointers are copied. The buffer pointers are: `newbuf`, pointing to the de-queued frame, `currbuf`, which points to the buffer used by the main thread to compute the displacement, and `shbuf`, that is the pointer exchanged between the threads. The mutex `mutex` is used to lock the access to the shared pointer `shbuf`, while the mutex `mutex_sync` locks the main until a buffer is dequeued. The three buffer pointers, as well as the structs addressed by the pointers and the mutexes, are stored in the structure `camera_dev_t`.

Considering the benchmark results (Section 5.3.5), the implementation without the additional thread is suitable when the YUV format is used, in particular if the camera resolution and frame rate are low. For instance, on the Raspberry Pi 1 B using upsampling 256, resolution 640×480 and 30 fps, the time required by the sum of the columns and the displacement computation is 5 ms in total, which is much lower than the frame time of the camera (33 ms at 30 fps). On the other hand, if the MJPEG format is used, the decompression of the frame becomes the largest contribution to the computation timings. Using the same configuration as above, the time required by the MJPEG decompression is 59 ms and the total time required by the frame processing is 64 ms. Since the computation time is larger than the frame time 33 ms (at 30 fps), the sample rate of the sensor is lower of the frame rate of the camera. Moreover, if the implementation without the buffers updating thread is used, the program is not able to use the newest frames. Therefore, it is highly recommended choosing a camera device with a frame format that does not require decompression or, if it is not possible to use raw formats, using the buffers updating thread.

The only function of the additional thread is to update the V4L2 buffers as soon as a new one is available, while decompression, sum of the columns and displacement are computed in the main thread. As a result, the buffer thread spends most of its time waiting for a new buffer to be available, which is not optimal in multi core processor. On fast multi core processors, if YUV or another uncompressed format is used, the computation of the sum of the columns could be moved to the buffer thread and evaluated for each received frame, even if it is not used. In doing so, the sum of the columns of the newest frame and the displacement of the older one are computed in parallel, with a possible performance gain. The timing constraint to use this method is that the time required by the sum of the columns has to be shorter than the time between the frames, otherwise some frames are missed or delayed.

3.6.5 Sum of the columns

As explained in Section 2.2.2, the sum of the columns requires to compute `RESX·RESY` integer additions, which can be expensive compared to the displacement computation. The time required to sum of the columns is in almost any case larger than the time required by the displacement computation, as it can be observed in the benchmark results. For instance, the time required by the Raspberry Pi 1 B to compute the vector of a frame with resolution 1280×960 is 28 ms, while the displacement computation with upsampling 64 only requires 2.4 ms.

Decreasing the number of rows which are added allows reducing the time required by this step. In the code the number of rows can be configured using the macro `SUMROWN` in `common.h`. For instance, if `SUMROWN` = `RESY` >> 3, then 1/8 of the rows are added and, considering the same 1280×960 frame, only 120 rows are added. On the Raspberry Pi the time is reduced from 28 ms to 3.6 ms, which corresponds to a reduction of almost 8 times. The sum time, combined with the time required to compute the displacement 2.4 ms, results in a total time of 6 ms. The reduction in time allows stepping up from a 30 fps sensor to a 60 fps one using the same processor and without a reduction in precision.

If the algorithm is implemented on a processor using a lower level interface with the camera module, for instance a parallel interface for data transfer and a serial port to configure the device, then the sum of columns can be computed while the frame is read one row at a time. This approach reduces the cost of the sum of the columns, which is merged with the frame reading, and also the memory requirement, since instead of storing two full frame buffers it only requires to store few rows. The benefits in terms of performance and memory usage come at the cost of increased complexity of the system. First, the camera module has to be configured correctly, usually using a serial interface (e.g. I2C). Then, assuming DMA is used to write the frames in memory, the program has to read each buffer after it is written by the DMA but before it is overwritten by the next row.

In the end, as discussed in Section 2.2.5, the algorithm may require to set the black bands to zero to reduce the error in case of large displacement. In the code the bands are set to zero using the `threshold_apply` function after the frame vector is computed. Since `threshold_apply` iterates the frame vector two times, the first to compute the threshold value and the second to set to zero the values smaller than the threshold, a possible optimization which reduces the number of iterations is to evaluate and apply the threshold

while computing the sum of the columns. But, since the number of iterations to compute the sum of columns is equal to the number of rows to be added `RESY`/8 (e.g. 75 with resolution 800×600), the effect of this optimization is almost negligible.

3.7 Serial

The sensor can be configured to write the computed position to the terminal output (stdout) or to UART serial interface. While the serial output is useful if the program runs on a computer and the output position is logged or used by other programs (e.g. using shell pipe "|"), the serial interface allows reading the position from the program running on a standalone device, which is the case if it is used in a proper sensor. The Terasic DE1-SoC provides a serial port connected to a serial to USB converter and it can be connected directly to a PC using a mini USB B cable, while the Raspberry Pi 1 B uses two I/O pins as UART transmitter and receiver.

The UART serial interface can be used only if a peripheral is available and it is configured with the same parameter at both ends. Serial interfaces are supported by the Linux kernel and they are exposed as a device files (e.g. `/dev/ttyS0` or `/dev/serial0`). The interface can be configured using "termios.h" header file, which allows the program to set baud rate, character size, parity bit and stop bits, as well as more advanced processing at the input or the output.

The functions related to the UART serial interface are defined in `serial.c`. The port is open and set up by the function:

Listing 3.18: `serial.c`

```
6  int serial_open(int *fd) {
```

The transmitter is configured to have 8 bit characters, 1 stop bit and no parity bit (8N1). The special processing at the output and hardware flow control are disabled, in order to simplify the communication with all kinds of receiver. The baud rate and the device name are set by the user using the macros `SERIALSPEED` and `SERIALDEV` respectively, both defined in `config.h`. The value of the speed `SERIALSPEED` is set using the macros defined by `termios.h`: B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200, B230400.

Once the interface is configured, the program can write to the interface as it would do with a normal file using the `SERIALDEV` device file. The writing to serial is implemented in the function:

Listing 3.19: `serial.c`

```
59  int serial_write(int fd, float data) {
```

The UART output format is configured using `SERIALFORMAT` in `config.h`. Two alternatives are available: `CHAR` and `RAW`. If the format is set to `CHAR`, then the floating point output position is converted to a string and transmitted as a sequence of characters. This approach is very useful for logging purposes and when interfacing with high level programs (e.g. LabVIEW), but it requires to transmit a variable number of bits equal to the number of digits to be represented, plus the decimal point and the termination characters `"\n\r"`. For instance, 12345.678912 is transmitted as `"12345.67891\n\r"` and requires 13 bytes,

while 0.67891 needs only 9 bytes. If `SERIALFORMAT` is defined as `RAW`, then the position is transmitted as a float value on 4 bytes. To ensure the correct transmission the endianness of the data as well as the size of the floating point type must be the same, which was the case in the tested configuration using the Terasic DE1-SoC and a laptop (Intel i7 4th gen running Linux). If the transmitted and received formats are different, some extra steps are required at the receiver or the transmitter to reorder the bytes and cast the data to the appropriate type.

The baud rate of the interface should be set to ensure that the transmission of each position value is finished before the next data is computed. Assuming that the frame rate is 60 fps, the sensor sample rate is, in the best case, 60 Hz. If the output format is `CHAR`, the maximum displacement is 1000 mm and the value is printed with 5 decimal places, then the maximum size of each position is 12 bytes. Since each byte is transmitted in a packet of 10 bits including the start and stop bit, the minimum baud rate is 7200 bit/s and the standard value 9600 bit/s can be used.

In the end the serial device file descriptor is closed by:

Listing 3.20: `serial.c`

```
54  void serial_close(int fd) {
```

Chapter 4

Testing

In this section it is discussed the technique used to verify the sensor readings using the Creality Ender 3 3D printer.

4.1 Read the video from file

Before proceeding with the testing on the printer, the code has been updated to support reading the frames of a previously captured video file, instead of capturing the frames with the camera. Using a video allows capturing the tag movements on the 3D printer once and use the same file to test different configurations of the sensor, highlighting the differences in the results.

Given a video file, the frames are read in three steps:

1. Convert the video file to individual raw grey scale frames (pgm images). This step is implemented with a Bash shell script which uses ffmpeg, a open source command-line video and audio converter [7], to extract and convert the frames from a video. The script also allows capturing the video with the desired resolution and frame rate.

2. Then a Python program is in charge to read the extracted frames and send them to the sensor program using a fifo. The choice to implement this step with an external program is due to the fact that reading all the files contained in a given directory in Python is less complex than doing it in C. Moreover, in this context we are not interested in the performance of the sensor, therefore a slower, but easier to implement, solution is acceptable.

3. Lastly, the sensor program receives the frames via the named pipe, computes the sum vector and evaluates the displacement. The frames are read with the functions defined in the file `camera_file.c`, which reimplements the functions defined in `camera.c`. This solution allows switching between the webcam and the pipe input by changing the source files when compiling the program, without modifying the other files. The Makefile target `make file` compiles the sensor with the header to read the frames from the pipe. During the tests it is useful to log the position by writing the stdout output of the sensor to a file (e.g. `./sensor > /tmp/log`).

4.2 Setup

The USB microscope used in this experiment is the DBPOWER 5MP 300X USB Digital Microscope which supports YUYV422 and MJPEG formats, frame rate up to 30 fps and resolutions 640×480, 800×600, 1280×960, 1600×1200 and 2048×1536. When capturing the videos the MJPEG format is used because it supports higher frame rates than YUYV. The magnification ratio indicated by the manufacturer of the microscope is from 10 to 300, but the size of the sensor is unknown. It has been observed that some frames are duplicated in the captured video, probably because of the driver or the camera itself since other USB webcams work correctly.

The 3D printer used in the testing is the Creality Ender 3, a filament printer which uses three stepper motors to move the extruder along the X, Y and Z axes. The manufacturer specifies a printing precision of 0.1 mm. The Z-axis, used in the testing, uses a 1.8° stepper motor, with 16 micro-steps, connected to a 8 mm leadscrew, therefore the Z-axis micro-step is 2.5 µm. The movement of the printer is controlled by the gcode files, which contains a sequence of commands to set the position and the speed of the extruder on the three axes. The smallest displacement which can be requested in the gcode along the Z-axis is 20 µm, which corresponds to half of the full step (8 micro-steps).

The accuracy of the 3D printer is a limiting factor for the experiment. First, since stepper motors are used with micro-stepping technique, we can expect an uncertainty of the angle in the order of the micro-stepping angle 1.8°, which corresponds to a Z-axis movement of 2.5 µm. Moreover, it has been observed that if the gcode requires the Z-axis to move 1 mm down and 1 mm up, the axis does not return in the original position, as shown in Figure 4.1. This difference in the position is caused by a the stepper motor uncertainty combined with the load and the friction of the leadscrew. Since the manufacturer does not provide accurate specifications, the unknown uncertainty is a limitation when trying to verify the position of the sensor and, while we are able to observe the sub-pixel movements of the axis, we can not consider this experiment a metrological verification.

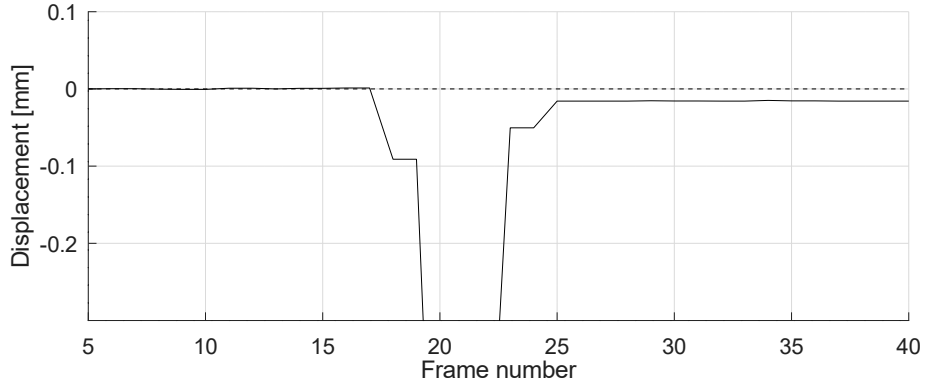

Figure 4.1: Position of the Z axis when moving down and up 1 mm. The position on the right half is −0.016 mm.

In this experiment the tag is applied to the extruder, which moves up and down along the Z-axis, while the camera is in a fixed position. Since the minimum gcode step is 20 µm,

it is apparent that sub-micron displacements can not be tested using this printer. However, it is possible to test sub-pixel displacements increasing the distance between the camera and the tag and, as a consequence, the field of view. If the field of view is 5.3 cm and the resolution is 640×480, each pixel corresponds to 83 µm. As a consequence, the 20 µm steps correspond to a displacement of 0.24 pixels. If the frame is scaled to half the resolution (320×240), each pixel corresponds to 166 µm and the step displacement is 0.12 pixels.

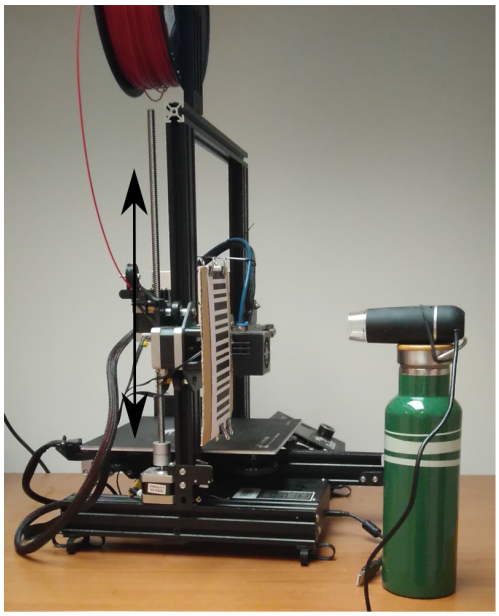

Figure 4.2: USB microscope and 3D printer setup during the sub-pixel testing

4.3 Results

The goal of the first test is to show the sub pixel displacement. In this case the total displacement is around 1 pixel, therefore the same reference frame is used for all the points. The gcode moves the printer extruder along the Z-axis in steps of 0.02 mm, with a delay between the movements. The camera is configured with a horizontal field of view of around 5.3 cm and the tag is printed with 3 cm wide set of bands. Different configurations of resolution and upsampling have been tested and the results are represented in Figures 4.4 and 4.3. We can observe that the steps do not have the same height with respect to each other. This difference is caused by the printer stepper: the 20 µm movements correspond to half of a full step of the stepper motor, therefore the displacements are alternated between full steps (16 micro-steps) and half steps (8 micro-steps). Since the fractional step and the full step have a different torque and the load of the leadscrew is applied to the motor, then the displacements of the full step and the half step are different.

Considering the resolution 640×480 and upsampling equal to 256, the maximum error with respect to the nominal value of the axis positions defined in the gcode is 4.8 µm, which corresponds to 0.06 px. Assuming that the uncertainty of the printer position is 2

micro-steps (5 µm) and that the uncertainty of the sensor reading is 0.01 pixels (0.8 µm), the sensor readings and the printer position are compatible.

In Figure 4.4 it is shown that the readings obtained scaling the frames to 320×240 are similar to the measures obtained with the 640×480 frames. The error of the measures with resolution 320×240 with respect to the nominal step is 5.6 µm (0.03 px), while the maximum difference with respect to the 640×480 readings is 1.6 µm (0.01 pixels). Assuming that the uncertaintiy of the readings is 0.01 pixels, which correspond to 0.8 µm for the 640×480 frames and to 1.6 µm for the 320×240 frames, the readings obtained with the two resolutions are compatible because their difference is smaller than the sum of their uncertainties.

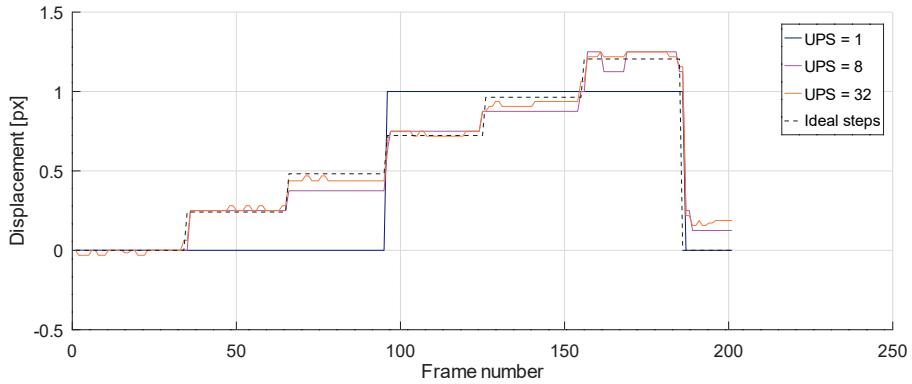

Figure 4.3: Sub-pixel test, displacement computed in pixels with resolution 640×480 and using different upsampling factors `UPS`

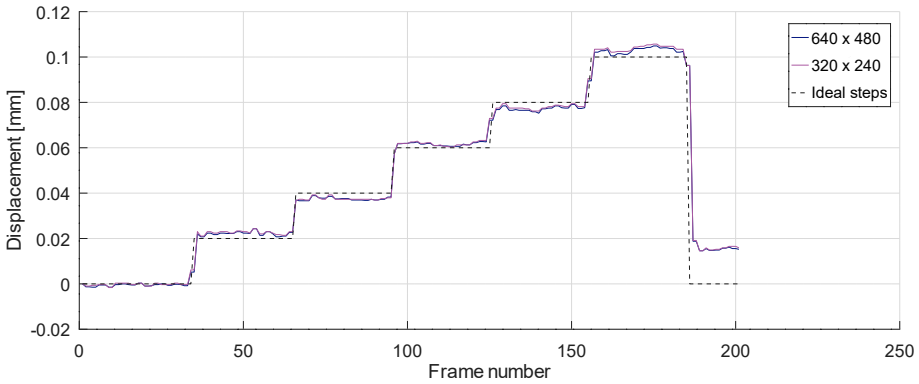

Figure 4.4: Sub-pixel test, displacement computed in millimeters with upsampling factor 256 and resolutions 640×480 and 320×240 (obtained downscaling 640×480)

The second test shows the behavior of the sensor using larger displacements and frequent reference frame updates. The error in the position of the printer in case of faster and larger movements is expected to be closer to the 0.1 mm declared by the manufacturer because of the acceleration and the vibration of the mechanical parts. Therefore, this test is more

qualitative than the previous one and its goal is only to show the ability of the sensor to track movements larger than the width of the frames. In this test the camera is placed close to the strip and the tag has narrow bands (a set of bands is 1.87 mm). The field of view of the microscope is 2.5 mm and, with resolution 640×480, each pixel corresponds to 3.9 µm. The gcode moves the extruder 3 mm up and down along the Z-axis at 36 mm/min, then moves it again 6 mm up and down on the Z-axis at 72 mm/min. The movements correspond to 769 pixel at 152 px/s and 1538 pixels at 304 px/s respectively. In this case the camera rolling shutter shows its limits, in fact the bands in the frames are tilted, in particular during the faster 6 mm movement.

The frame rate of the video is ~ 25 fps, therefore the displacement between the frames is 6 pixels during the slower 3 mm movement and 12 pixels during the faster 6 mm one. Since the frame contains 4 peaks, the reference frame is update with displacements larger than $640/4 \cdot 0.8 = 128$ pixels (Section 2.2.6). As a consequence, the reference is updated every 19 frames during the 3 mm movement and every 10 frames during the 6 mm displacement. Figure 4.5 shows the position measured during the experiment.

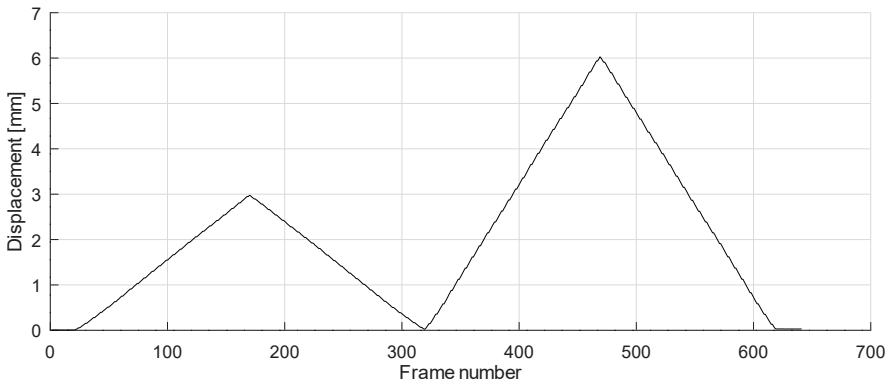

Figure 4.5: Large movements test, displacement computed in millimeters with upsampling factor 256 and resolution 640×480

Chapter 5

Benchmarks

In this chapter it is discussed the setup used to compare the performance of different sensor configurations. Then, the performance of different parts of the program is analyzed.

5.1 Automated benchmarks

The goal of the benchmark is to run the sensor program multiple times with different configurations and to save the timings in a file which can be easily read, parsed and plotted using Octave. The system is composed of three main parts:

1. The file `benchmark.c` defines the functions needed to compute and to write to file the timings of different parts of the program. The source code of the sensor has been modified adding calls to such functions, guarded by `#ifdef BENCHMARK`, to save the times required by the MJPEG decompression, the sum of columns and the actual displacement computation. The program saves the times required by each part at every iteration and, at the end, it writes the average timings and the sample rate. The output directory for the timings file is defined by `BENCHDIR` and the name of the file is `bench_RESX_RESY_ZPUPS_UPS`. In this mode the number of frames to be captured before closing the program is defined by `BENCHN` in `common.h`.

2. Makefile and `common.h` are modified to allow configuring and compiling the sensor with a single command. The make target to compile the benchmark is `bench`, which allows passing the sensor configuration parameters as command line variables and defines the macro `BENCHMARK`. The file `common.h`, if `BENCHMARK` is defined, overwrites the macros defined in `config.h` using the values received as command line variables. The parameters that can be overridden are `RESX`, `RESY`, `ZPUPS` and `UPS`. As an example, the following command compiles the sensor with 640×480 resolution, zero-padding disabled (`ZPUPS` $= 1$) and upsampling factor `UPS` equal to 32:

   ```
   $ RESX=640 RESY=480 ZPUPS=1 UPS=32 make bench
   ```

3. Lastly, the Bash script `runbench.sh` is in charge to compile the sensor program with different configurations, run it, read the produced logs and write the average times

in a file which can be easily parsed using Octave or similar programs. The script is executed on the target device and the files are copied on the computer manually. A possible improvement, which does not require to install gcc and the headers on the target device, is to cross-compile the code on the computer, copy the executable and run it on the target device (e.g. using a serial console or ssh) and send back to the computer the results (e.g. serial, ssh, sockets).

5.2 Benchmark setup

The two devices used for the benchmark are the Terasic DE1-SoC board (ARM Cortex A9 dual core 800 MHz, NEON SIMD and 1 GB DDR3 RAM) and the Raspberry Pi 1 model B (ARM11 single core 700 MHz, VFP floating point unit and 512 MB RAM). As it is shown in the following section, the DE1-SoC is capable to run higher frame rates and resolutions than the Raspberry Pi thanks to the higher frequency, the newer ARM architecture and the NEON SIMD unit.

In the benchmark results we are interested in comparing the timings of the displacement computation, which depends on the resolution, the zero-padding upscaling and the DFT upsampling. The displacement time starts at the computation of the FFT of the captured frame in the main loop and ends at the evaluation of the position. The other terms which are analysed are the time required to compute the vector of the sum of the columns of a frame, using different resolutions and changing the number of rows to be added, and the time required by libjpeg-turbo to decompress a JPEG frame. The maximum sample rate of the sensor, if a uncompressed frame format is used, is the inverse of the sum of the displacement computation time and of the time required by the sum of the columns. If the MJPEG format is used, the JPEG decompression time is added to the total time.

5.3 Results

5.3.1 FFT vs DFT

The goal of this section is to compare the performance of the implementation that uses the zero-padded inverse FFT (discussed in Section 2.2) and the performance of the solution using the definition of DFT to upsample the correlation only around the peak (Section 3.1). Both the solutions are implemented in C and use the FFTW library to compute the FFTs. The Raspberry Pi 1 B displacement times are shown in Figure 5.1 and the Terasic DE1-SoC times are reported in Figure 5.2.

The DFT approach is faster than the FFT method and the time difference between the two approaches is proportional to the upsampling and the resolution. Considering the Raspberry Pi results with resolution 1280×960 and upsampling 64, the time required by the displacement computation in the FFT case is 44 ms, while the DFT implementation takes only 3 ms. Assuming that in both cases the sum of the columns, considering only 1/8 of the rows, requires 2 ms and that the frames are uncompressed, then the FFT solution maximum sample rate is 21 Sa/s, while the DFT one can reach 200 Sa/s. This means that the sample rate is almost 10 times higher using the DFT method on the same hardware, or that a less capable processor can be used to obtain the same performance of the FFT

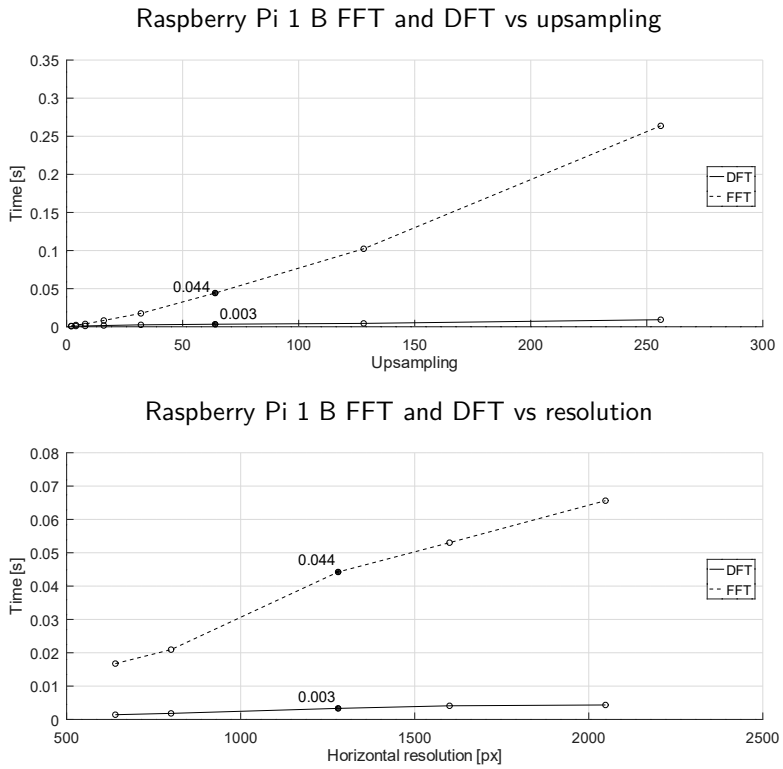

Figure 5.1: Raspberry Pi 1 B displacement computation time using FFT and DFT implementation. The marked points have resolution 1280×960 and upsampling factor 64.

solution. Moreover, the DFT solution computes the upsampled vector only around the peak and requires a fraction of the memory needed by the FFT implementation, that requires to store the complex zero-padded cross-correlation in the frequency domain and the whole upsampled spatial cross-correlation (Section 2.3.4).

The time required by FFTW to prepare the plans grows with the size of the input and output vectors. When using the FFT approach, if the exhaustive research `FFTW_PATIENT` is used (Section 3.5.2), the plan creation can require up to tens of minutes because of the large size of the padded FFT, corresponding to the horizontal resolution multiplied by the upsampling factor. For this reason, in the testing, the plans with upsampling factors larger than 32 are computed using the faster method which explores less solutions `FFTW_MEASURE` or the approximate method `FFTW_ESTIMATE`. The computed plans can be less optimized compared to the ones obtained with slower methods.

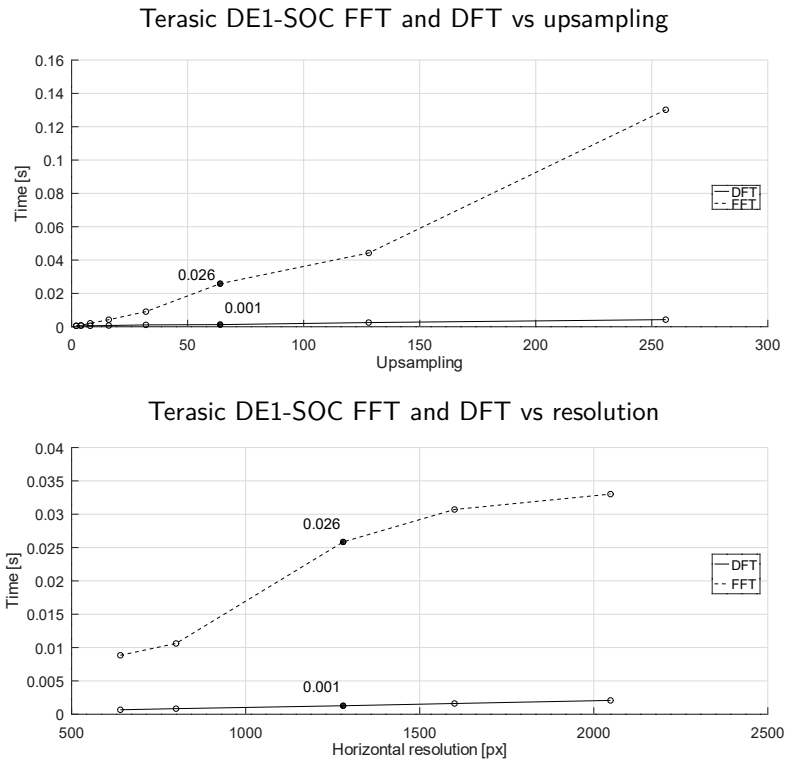

Figure 5.2: Terasic DE1-SoC displacement computation time using FFT and DFT implementation. The marked points have resolution 1280×960 and upsampling factor 64.

5.3.2 Window position upsampling

If the position of the upsampling window is determined with better precision using the zero padding technique, the width of the upsampling interval can be decreased and the total time of the displacement computation is reduced (Section 2.3.6).

In Figure 5.3 the displacement computation timings are shown without the zero-padding (`ZPUPS` $= 1$) and with zero-padding factor `ZPUPS` $= 2$ using the camera resolution 640×480. While in case of small upsampling factors there is a small difference in using the zero-padding, with higher upsampling values the time reduction is substantial.

For instance, considering the results obtained on the Raspberry Pi 1 B with resolution 640×480 and upsampling 128, the displacement computation of the solution without the zero-padding requires 3.8 ms, while the solution with `ZPUPS` $= 2$ takes 2.1 ms, which correspond to a reduction of 1.7 ms (45% reduction). Considering the upsampling factor 16, the reduction is 0.3 ms, from 1.2 ms to 0.9 ms (25% reduction). Since the time reduction is larger for greater upsampling factors and the zero-padding solution requires more memory to store the padded vectors, the solution with `ZPUPS` $= 2$ is used only for upsampling factors `UPS` > 32.

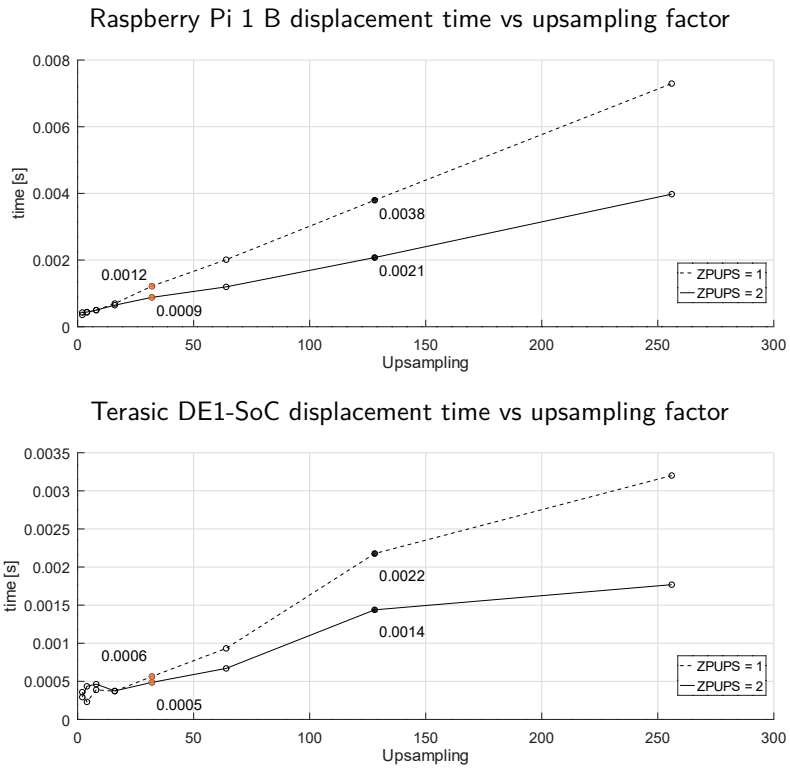

Figure 5.3: Displacement computation time with zero-padding factors `ZPUPS` 1 and 2. The resolution is 640×480, the marked points have upsampling 16 (orange) and 128 (black).

5.3.3 Sensor displacement computation

In this section the timings of the displacement computation using the DFT method with various upsampling factor and resolution combinations are shown. The sensor uses zero-padding (`ZPUPS` $= 2$) to compute the position of the upsampling interval only if the upsampling factor `UPS` > 32.

Figure 5.4 shows the displacement computation timings against the resolution and the upsampling on the Raspberry Pi 1 B. Figure 5.5 shows the same plots for the Terasic DE1-SoC.

The sample rate of the sensor is not determined only by the time of the displacement computation. In fact, the maximum sensor rate, in case of YUYV format, is the inverse of the total time required by the sum of the columns and the displacement (Section 5.3.4). If MJPEG is used, the decompression is the main contribution to the total time and the maximum sample rate drops drastically (Section 5.3.5). Moreover, the maximum value of the sample rate is limited by the frame rate of the camera device, therefore, it is not convenient to use a camera device with low resolution and low frame rate together with a high-performance processor because a part of the processor capabilities is unused (Section 5.3.6).

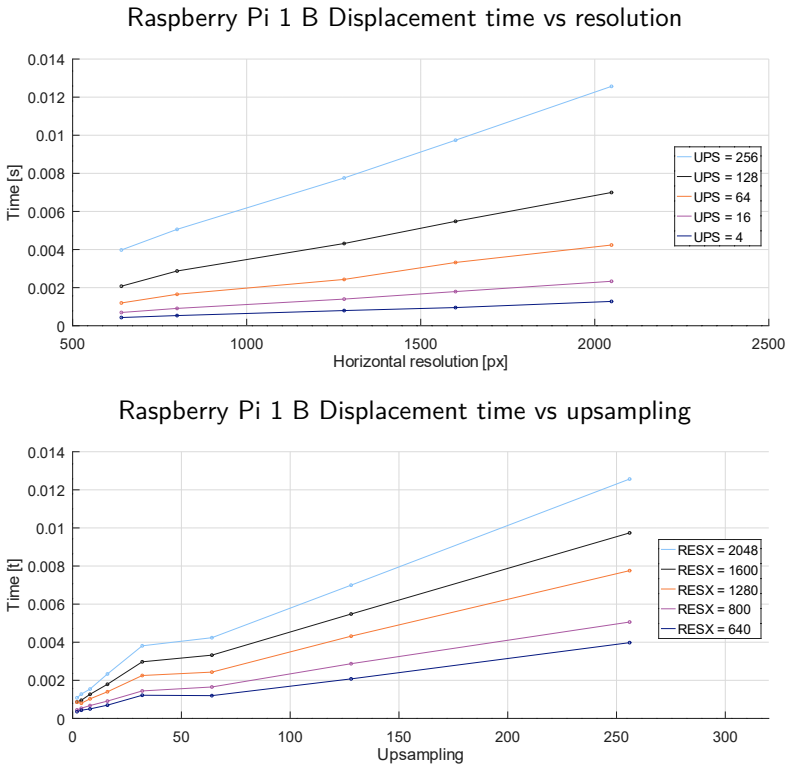

Figure 5.4: Raspberry Pi 1 B displacement computation time using different resolutions `RESX` and upsampling factors `UPS`

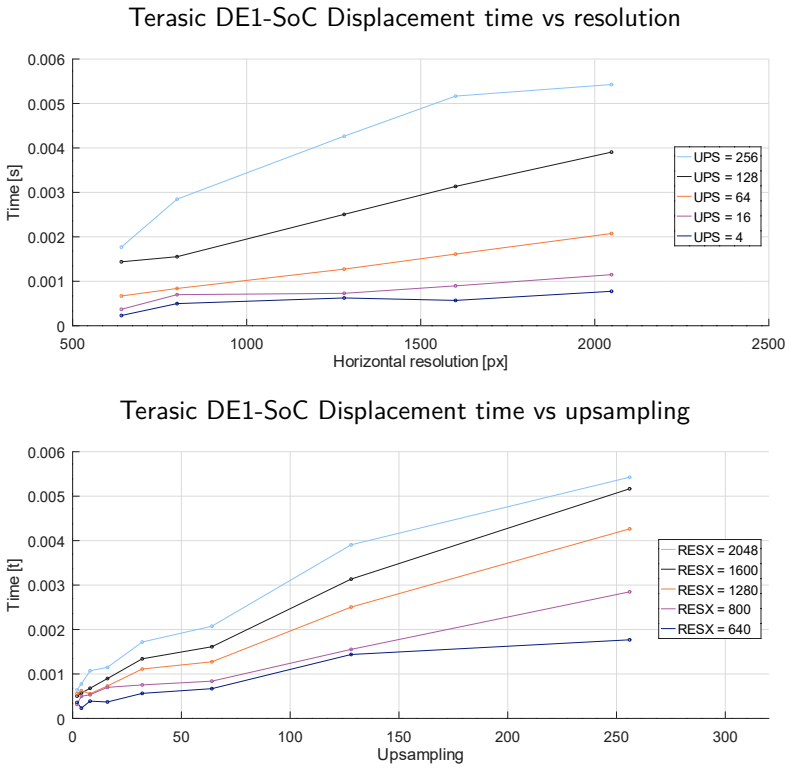

Figure 5.5: Terasic DE1-SoC displacement computation using different resolutions `RESX` and upsampling factors `UPS`

5.3.4 Sum of columns

As discussed in Section 2.2.2, the sum of the columns of a frame can be computed using all the rows or only a fraction of them. In Figure 5.6 the timings of different rows count are compared. The time required by the sum of the columns, even considering only 1/8 of the rows, is not negligible with respect with the displacement computation time. For instance, the following table shows the displacement time and the time required by the sum of the columns using 1/8 of the rows, evaluated on the Raspberry Pi using resolution 1600×1200 with upsampling 64 and resolution 1280×960 with upsampling 128.

Resolution	Upsampling	Displacement time	Sum of columns time	Total time
1600×1200	64	3.3 ms	5.6 ms	8.9 ms
1280×960	128	4.3 ms	3.6 ms	7.9 ms

Although the displacement time using the 1600×1200 image is smaller than the time using the 1280×960 frame because of the smaller upsampling factor, if the time of the sum of the columns is taken into account, the total time required by the 1600×1200 image is larger.

The total time required to compute the sum of the columns and evaluate the displacement, using different resolution and upsampling combinations, is shown against the resolution in Figure 5.7.

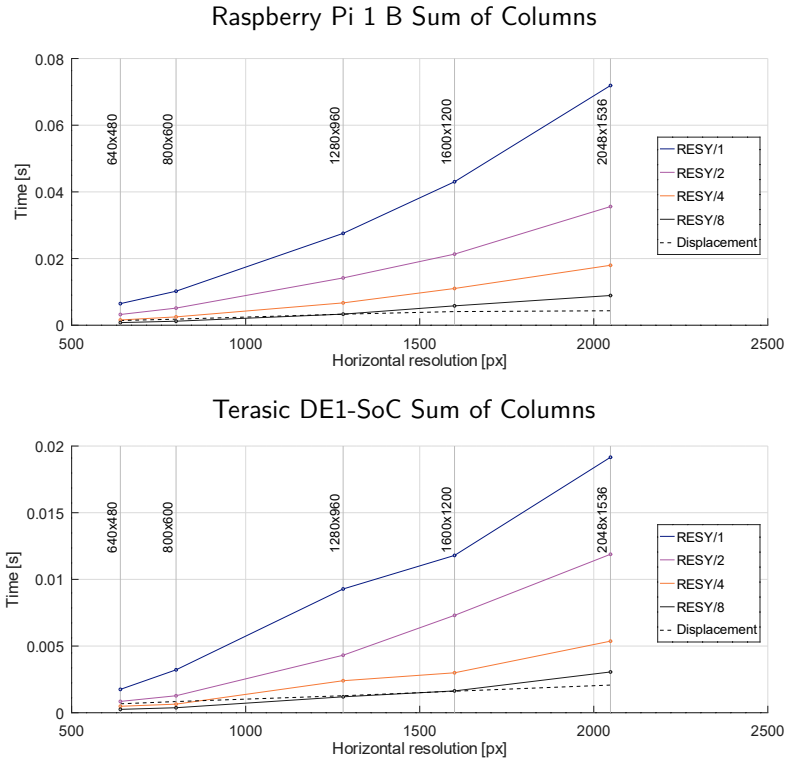

Figure 5.6: Sum of columns time against the upsampling factor using different resolutions. A variable number of rows is added, defined as fraction of the vertical resolution `RESY`. The displacement is evaluated with upsampling 64.

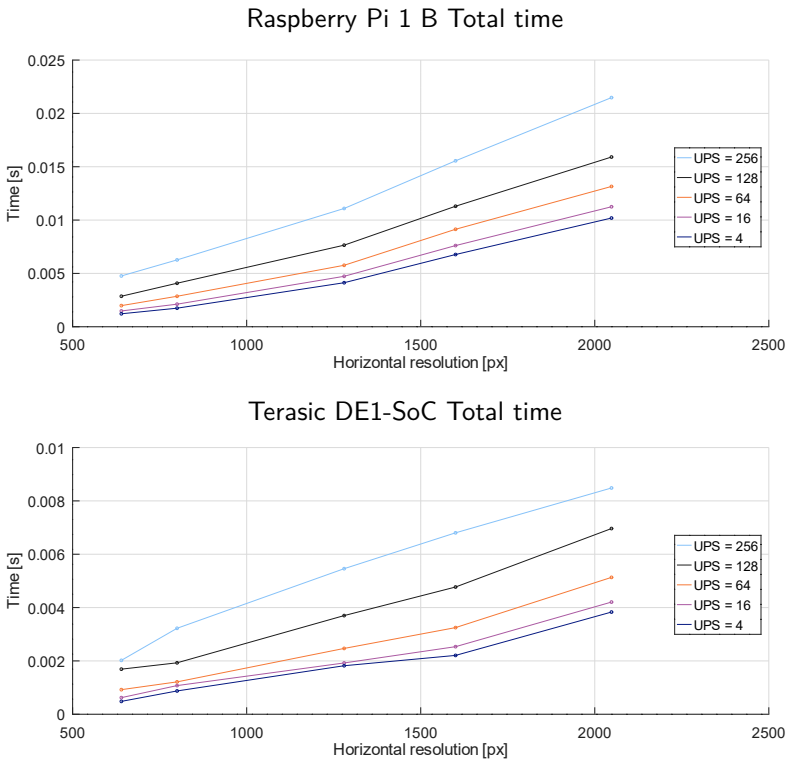

Figure 5.7: Total time (sum of the columns time plus displacement time) against the resolution for different upsampling factors `UPS`. The sum of the columns is computed using 1/8 of the rows.

5.3.5 JPEG decompression

Although uncompressed frame formats are recommended, the Terasic DE1-SoC board did not work correctly with YUYV because of synchronization issues, therefore MJPEG had to be used (Section 3.6.1). The results shown in this this section are evaluated using the DBPOWER USB microscope and they may vary using different camera devices because of the different JPEG quality. As shown in Figure 5.8, the time required by the JPEG decompression is much larger than time required by the sum of the column and the time of the displacement computation, therefore the total time required to process a new frame mostly depends on the JPEG decompression.

For instance, on the Raspberry Pi 1 B, considering the smallest resolution available 640×480 with an upsampling value of 256, the time required by the decompression is 61.4 ms, while the time to compute the displacement is 4.0 ms and the time to compute the sum of the columns is 0.8 ms. As a result, if MJPEG is used the total time is 66.2 ms and the maximum sample rate is 15 Sa/s, while if YUYV is used the total time is reduced to 4.8 ms and the maximum sample rate is increased to 208 Sa/s, which is a improvement by almost 14 times.

The impact of the JPEG decompression increases with the resolution: considering the resolution 2048×1536, the decompression takes 453 ms, while the sum of the columns and

the displacement computation with upsampling 256 take 22 ms. In this case the total time is reduced by a factor of 21 and the sample rate is increased from 2 Sa/s to 45 Sa/s.

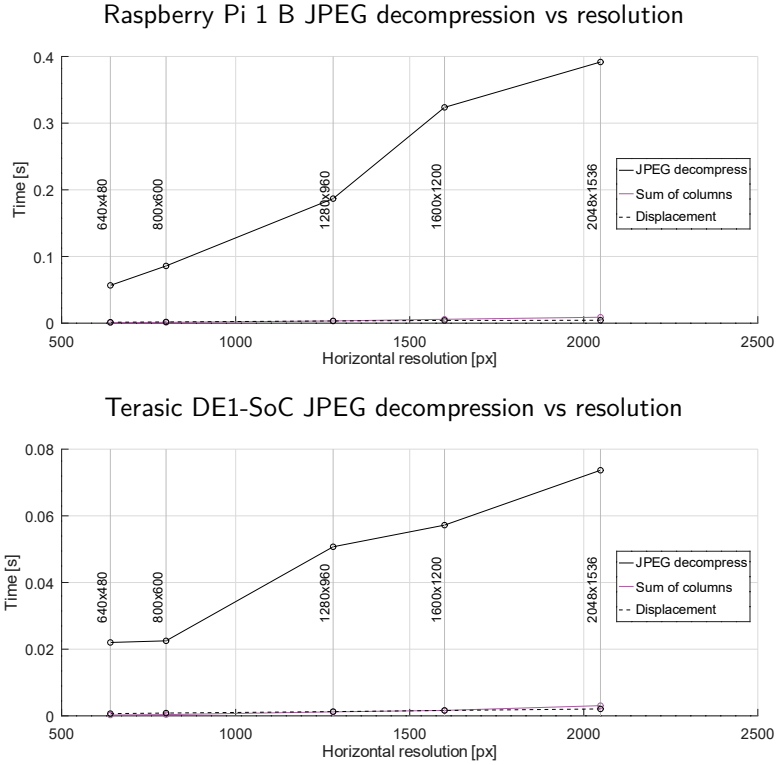

Figure 5.8: JPEG decompression time against the resolution. The sum of the columns using 1/8 of the rows and the displacement with upsampling 64 are shown for comparison.

5.3.6 V4L2 capture and buffers

In this section it is discussed the relation among the sensor sample rate, the V4L2 buffers configuration and the camera frame rate. The V4L2 buffer management directly affects the sensor latency and sample rate, as discussed in Section 3.6.3.

First of all, although the DBPOWER USB microscope is advertised with frame rate 30 fps, the maximum value which could be obtained using the smallest resolution 640×480 on a laptop with a Intel i7 4th gen processor is around 25 fps, both with MJPEG and YUYV. When using the boards the maximum frame rate is lower: around 20 fps on the Terasic DE1-SoC and 10 fps on the Raspberry Pi 1 B, both with MJPEG and YUYV. The frame rates are also measured using the Microsoft LifeCam HD-3000 USB webcam with resolution 640×480 and format MJPEG. In this case the frame rates are 30 fps on all the three devices. Since the frame rates of the microscope are lower using the same video configuration, in this section the Microsoft USB webcam is used. Another difference between the two devices is that the time required to decompress the JPEG frames captured using the Microsoft webcam is shorter than the time required to decompress the images produced by the microscope, probably due to a different image quality. For this reason the

JPEG decompression times used in section do not correspond with the results obtained in the previous sections.

Second, it is common that the camera devices supporting both compressed and uncompressed formats allow higher frame rates when using the compressed format compared to when the uncompressed format is used, primarily at higher resolutions, due to the lower transfer rate. For instance, the Microsoft webcam at 1280×720 resolution supports 30 fps if MJPEG is used, but only 10 fps when YUYV is used. In this section, where we are interested in comparing the V4L2 buffers behavior and not in the performance of the algorithm, the MJPEG format is used.

To compare different buffer configurations the Microsoft LifeCam HD-3000 USB webcam is configured with MJPEG format and resolution 640×480 at 30 fps, and the Raspberry Pi 1 B is used to run the program. Two cases are analyzed:

1. The processing time is smaller than the frame time: the sensor is configured to compute the sum of the columns using 1/8 of the rows and the upsampling factor is set to 1. The total time required by decompression, sum of the columns and displacement computation is 24 ms, which is smaller than the frame time 33 ms.

2. The processing time is larger than the frame time: the sensor computes the sum of the columns using all the rows and the upsampling factor is set to 1024. The configuration is chosen to increase the total time required to process the frame. The total time is 45 ms, which is larger than the frame time.

The configurations of the V4L2 buffers which are compared in this test are the solution without the buffer updating thread using 1, 2 and 4 buffers, and the solution with the updating thread, which uses 4 buffers in total but only 2 in the queue. In Figure 5.9 it is shown a chart with the obtained sample rates.

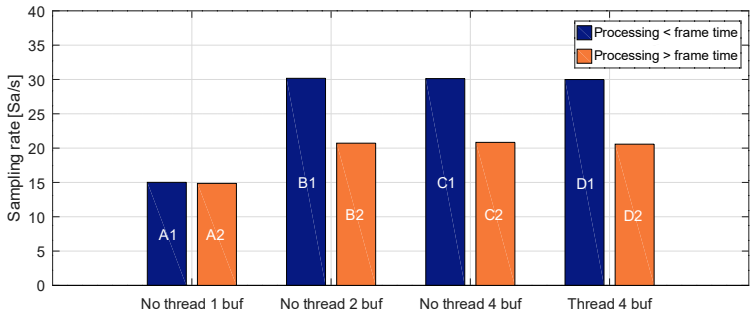

Figure 5.9: The blue columns (X1) show the sampling rate with the processing time is smaller than the frame time, the orange columns (X2) show the sampling rate with the processing time larger than the frame time.

The sample rate obtained using only 1 buffer (A) is lower than all the other solutions, even if the processing time is smaller than the frame rate (A1). Since the sample value is half of the expected one (15 vs 30 fps), we can assume that the frames are written to the buffers at intervals equal to the frame time. Therefore, a buffer which is pushed in the

empty queue has to wait for the camera to write the next frame and half of the frames are missed.

On the other hand, the solutions B, C and D provides the same sample rates: 30 fps when the processing time is lower than the frame time (B1, C1, D1) and $\sim$ 21 fps when the processing time is larger than the frame time (B2, C2, D2). In all the solutions that use multiple buffers it is guaranteed that there is always an empty buffer in the queue which can be written by the camera when it has finished writing the last frame. As a consequence, the sample rate is equal to the frame rate if the processing time is lower than the frame time, while it is equal to the inverse of the processing time (plus a little overhead due to the buffer handling) when the processing time is higher than the frame time.

Comparing sample rate of the solutions A and B we can observe that there is no advantage in using more than 2 buffers without the upsampling thread, therefore 2 threads should be used. On the contrary, although it appears that the threaded solution (D) does not provide any benefit considering the sample rate, the updating thread allows the sensor to use the most recent available frame when the processing is slower than the frame time, which is not always the case with the non-threaded solution, as discussed in Section 3.6.4.

Chapter 6

Conclusion

This chapter describes the obtained results and possible areas of future work.

6.1 Results

This thesis has described a incremental position sensor able to compute the sub-pixel displacement of the captured frames with accuracy up to 1/10 and 1/100 pixels, depending on the required movement speed and range. Sub-micron accuracy can be achieved using the appropriate camera sensor and optics combination, for instance using a camera sensor with pixel size $< 10\,\mu m$ and optics magnification 1.

The sensor achieves the best accuracy using small displacements between the frames, therefore low movement speed (Section 2.2.5). In fact the sensor is able to achieve 1/100 pixels accuracy if the circular cross-correlation works with an ideal set of bands. This result can be obtained either by cropping the frame vectors to the correct width, as analyzed in Octave, or by imaging exactly the ideal set of bands, which requires to manually configure the camera and the width of the bands on the printed strip. Since the C program does not implement the cropping, to achieve the best performance it is required the manual configuration. If the sensor has to track faster movements, then the solution using larger displacements between the frames can be used, but the accuracy is decreased (1/10 pixels range).

The measurement range of the sensor is virtually unlimited because of its incremental nature, but a wider range comes at the cost of a larger uncertainty derived by the accumulation of the error. In fact, each time the reference frame is updated, the uncertainty of its position is added to the error of the displacement of the next frames. For instance, if the maximum error of the displacement computed with the cross correlation is $0.1\,\mu m$ and the reference frame is updated 10 times, then the maximum error of the final position is $1.1\,\mu m$. To mitigate this effect, the reference position is not updated at each frame, but only if the displacement between two frames is larger than a certain threshold.

The sample rate of the sensor is limited by the camera frame rate and by the time required to compute the displacement. To ensure that all the captured frames are evaluated, which means that the sample rate of the sensor is equal to the frame rate of the camera, the processing time has to be lower than the frame time. If the processor is not

powerful enough, the processing time is larger than the frame time, therefore some frames are skipped and the sample rate is lower than the frame rate. Since most low-cost camera modules or USB devices have a maximum frame rate 60 or 90 fps, the sample rate is limited to such values. The sample rate of the proposed sensor is much lower than the bandwidth of commercial sensors, which exceeds 100 kHz using solutions such as capacitive and piezoresistive sensors. As a result, the proposed solution is not suitable to analyze vibrations and to control fast movements.

The maximum speed of the object is determined by the maximum allowed displacement between two frames (which depends on the number of white bands in the frames), the camera horizontal field of view and the sample rate of the sensor (Section 2.2.6). If the sample rate is limited to 60 or 90 Sa/s, the minimum number of white bands is ≥ 5 and the field of view is usually smaller than 10 mm, the maximum speed of the more accurate solution using small displacements is $v_{1_{\max}} < 11\,\mathrm{mm/s}$, while the maximum speed of the faster sensor that uses larger displacements is $v_{2_{\max}} < 90\,\mathrm{mm/s}$. The maximum speed of the proposed solution is much lower than the speed of commercial products. For instance, linear encoders support speeds up to several meters per second.

If the camera uses a rolling shutter, common on low-cost camera devices, the bands in the captured frame can appear tilted when the movement speed is large enough, which results in distorted peaks when the images are converted to vectors. For instance, during the testing, using the USB microscope with resolution 640×480 at 25 fps and field of view 2.5 mm, the impact is visible at speed 1.2 mm/s (300 px/s), while the maximum speed is 7.5 mm/s (Section 4.3). To reduce the distortion, the frame vectors are computed using only a fraction of the rows of the whole frame (e.g. 1/8 of the vertical resolution). The effect of the rolling shutter depends on the camera speed and it is not present using global shutter sensors.

The displacement results are available as digital float value and they are written to a serial interface. Since the sample rate is very low, the C implementation uses UART, but different interfaces can be used depending on the available peripherals. On the contrary, most of the commercial sensors have analog output (e.g. voltage, current, resistance, capacitance), therefore they require additional conditioning circuits, usually available from the sensor manufacturer, to obtain a reading.

The sensor uses a strip composed of black and white vertical bands printed using a common printer. If the print quality is too poor, the bands may not be well defined, affecting the precision of the sensor. Therefore, it is recommended a laser printer with high DPI (e.g 1200). Similarly, the lighting of the strip should be adequate and uniform on the whole area captured by the camera to ensure well defined peaks with equal height in the processed frame vectors. The sensor also requires that the distance between the tag and the camera is constant, that the camera is perpendicular to the strip and and that the vertical bands are as aligned as possible to the columns of the frame. These precautions ensure that the bands have the correct size in the captured frame at any point of the strip.

Since the strip and the camera are exposed to the environment, dust and debris can accumulate on the tag and on the camera lens. As a result, although the cross-correlation does not require extremely precise bands and the sum of the columns filters small defects in the frames, the readings can be affected in case of significant dirt on the strip or on the lens. Likewise, some types of commercials sensors, including optical linear encoders and laser interferometers, are not suitable to work in dirty environments. On the contrary,

solutions such as LVDTs and inductive sensors are resilient to contamination and they are optimized to work in demanding industrial environments.

Finally, the cost of the proposed sensor using off the shelf components can be lower than 100 €, depending on the selected hardware, camera device and lens. For instance, using the Raspberry Pi Zero W (9 €), which uses the same SoC of the Raspberry Pi 1 B used in testing, a compatible MIPI CSI camera module with 1/4" (3.2×2.4 mm) sensor with M12 lens mount (15–25 €) and a suitable lens to obtain magnification factor 1 (20–30 €), the total cost of the parts is around 65 €. Using the camera resolution 1280×720 at 60 fps and upsampling factor 256, the sensor can achieve 10 nm resolution, sub-micron accuracy and sample rate 60 Sa/s. On the other hand, the cost of commercially available sensors is in the order of a few hundreds to a few thousands of euros, depending on the technology, the features and the resolution. For instance, LVDTs with measurement range in the order of the centimeters and sub-micrometer resolution cost a few hundreds of euros, but, since they are used in industrial, aircraft or even nuclear applications, solutions designed for extreme environments are available at ten times the price. In case of linear encoders the resolution highly affect the price of the products: encoders with micrometer resolution cost a few hundreds of euros, while encoders with resolution equal to tens of nanometers cost thousands of euros. Therefore, the proposed sensor, that reaches sub-micron resolution, provides the position on a serial interface and costs less than 100 €, although it has limited sample rate and speed, is a compelling alternative for low-cost applications with slow movements.

6.2 Future work

The accuracy sensor has not been properly characterized, therefore an area of research is the rigorous evaluation the uncertainty and the resolution of the sensor to verify the obtained results and to better define the configuration required to obtain repeatable sub-micron readings. A definition of the characteristics of interest (e.g. non-linearity, resolution, accuracy), as well as a comparison of commercially available position sensors using the defined parameters, is available in [8].

The developed solution is a software implementation that uses floating point vectors and runs in a Linux distribution. As a result, the solution is extremely portable and can run on most of Linux computers or board, but at the cost of the high operating system overhead and the increased memory and processing power requirements. In order to develop a commercial product, the sensor has to be implemented in hardware using a FPGA or in software on a embedded processor or a DSP. The FPGA solution maximizes the performance of the sensor exploiting the parallelism of the FFT and is suitable for high frame rates and high frame resolution systems, but working with FPGAs increases the cost and the complexity. On the contrary, the processor (or DSP) implementation is recommended for low frame rates sensors because of the smaller cost and implementation complexity. In both cases the technique has to be implemented using fixed point notation to better utilize the hardware and, without the support of the Linux kernel, the frames capture has to be managed at a lower level.

Appendix A

C code

A.1 config.h

Version of the file `config.h` with comments removed.

```
1  #define UPS 256
2  #define WISDOM "./wisdom"
3  #define DEVICE "/dev/video0"
4  #define FORMAT MJPEG
5  #define RESX 640
6  #define RESY 480
7  //#define THREAD
8  #define STDOUT
9  //#define SERIAL
10 #define SERIALDEV "/dev/serial0"
11 #define SERIALSPEED B115200
12 #define SERIALFORMAT RAW
13 #define TRIPLETWIDTH 3.75*0.5
14 #define PIXTOMMOUT
15 //#define PIXTOMMCAL 0.08310249
```

A.2 common.h

Version of the file `common.h` with comments and unnecessary macros removed.

```
1  #include "config.h"
2
3  #if UPS < 64
4  # define ZPUPS 1
5  #else
6  # define ZPUPS 2
7  #endif
8
9  #if ZPUPS == 1
10 # define WIN ((int)(UPS*2))
11 #elif ZPUPS == 2
12 # define WIN ((int)(UPS*1))
13 #endif
14
15 #define DELTARANGEFACT 1.3
16 #define STRINGIFY(x) #x
17 #define STR(x) STRINGIFY(x)
18 #define WISDOMFILE WISDOM "_" STR(RESX) "_" STR(RESY) "_" STR(ZPUPS) ".txt"
19 //#define ZEROTH
20
```

```
#ifdef THREAD
# define BUFN 4
#else
# define BUFN 2
#endif

#define SUMROWN RESY >> 3
#define DELTATHFALLBACK (RESX/4)
#define DELTATHFACT 0.8
#define RESX2 ((unsigned int) RESX >> 1)
#define ZPRU ((unsigned int)((unsigned int)RESX * (unsigned int)ZPUPS))
#define ZPRU2 ((unsigned int)(ZPRU >> 1))
```

A.3 fft.c

```
#include "fft.h"

#ifdef DEBUG_FFT
static void print_vector(float *vect, char * filename, unsigned int length);
#endif

/**
 * DFT matrix exponents factor.
 * Definition of the common factor the arguments in the inverse DFT.
 */
#define ARG_CONST ((float)M_PI*2/UPS/RESX)

/* Allocate a real vector of length n and return its pointer.
 */
float * fft_malloc_r(unsigned int n) {
  return (float *) fftwf_malloc(sizeof(float) * n);
}

/* Allocate a complex vector of length n and return its pointer.
 */
fftwf_complex * fft_malloc_c(unsigned int n) {
  return (fftwf_complex *) fftwf_malloc(sizeof(fftwf_complex) * n);
}

/* Free an allocated vector.
 */
void fft_free(void ** v) {
  if (*v) {
    fftwf_free(*v);
    *v = NULL;
  }
}

/* Return the fftwf plan for the forward fft.
 * Wrapper of the fftwf function fftwf_plan_dft_r2c_1d.
 * Since the input is real the r2c plan is used because it is faster with
 * respect to the complex to complex transform.
 * Note that the input vector is not preserved.
 * The size of the vectors RESX is defined in common.h
 */
fftwf_plan fft_init_fw(float *in, fftwf_complex *out) {
  return fftwf_plan_dft_r2c_1d(RESX, in, out, FFTW_PATIENT);
}

/* Return the fftwf plan for the inverse fft.
 * Wrapper of the fftwf function fftwf_plan_dft_c2r_1d.
 * Since the input is real and the input complex vector in the frequency domain
 * satisfies the Hermitian condition, the r2c plan is used because it is faster
 * with respect to the complex to complex transform.
 * By default the inputs are not preserved, so FFTW_PRESERVE_INPUT is needed.
```

```
51   * Fill the input vector with zeros if ZPUPS > 1. Since the vector is
52   * preserved there is no need to write the zeros at each iteration.
53   * The size of the image vectors RESX and upscaling UPS are defined in common.h
54   */
55  fftwf_plan fft_init_bw(fftwf_complex *in, float *out) {
56    fftwf_plan plan;
57    plan = fftwf_plan_dft_c2r_1d(ZPRU, in, out, FFTW_PATIENT | FFTW_PRESERVE_INPUT);
58
59  #if ZPUPS > 1
60    // Not sure if the plan creation destroys the input vector, to be safe
61    // write the zero-padding zeros after plan creation.
62    for (unsigned int i = RESX+1; i < ZPRU2; i++) {
63      in[i][0] = 0;
64      in[i][1] = 0;
65    }
66  #endif
67
68    return plan;
69  }
70
71  /* Given the fourier transform of vectors of the two images compute the cross
72   * correlation in the frequency domain.
73   * Since the two image vectors satisfy the Hermitian condition, then the cross
74   * correlation satisfies it too. For this reason only half of the vector is
75   * computed.
76   * Apply the zero padding when needed.
77   */
78  void fft_xcorr(fftwf_complex *v1, fftwf_complex *v2, fftwf_complex *out) {
79    unsigned int i;
80
81    // Compute the cross correlation of the first half of the vector
82    // The length of the vector is RESX/2+1
83    for (i = 0; i < RESX2 + 1; i++) {
84      // Real part
85      out[i][0] = v1[i][0] * v2[i][0] + v1[i][1] * v2[i][1];
86      // Imaginary part
87      out[i][1] = v1[i][1] * v2[i][0] - v1[i][0] * v2[i][1];
88    }
89
90  #if (ZPUPS > 1)
91    // Zero padding, the vector is preserved, no need to write all the zeros at
92    // each iteration
93    out[ZPRU2][0] = out[RESX2][0];
94    out[ZPRU2][1] = out[RESX2][1];
95    out[RESX2][0] = 0;
96    out[RESX2][1] = 0;
97  #endif
98  }
99
100 /* Find the index of the maximum and compute the distance from the center of
101  * the image, which corresponds to the displacement.
102  * The cross correlation vector in space has the first half and the second half
103  * swapped because of the FFT and the cross correlation, this is taken into
104  * account computing the index (without reordering the vector).
105  * If there are 2 points with the same max value next to each other then
106  * average of their position is returned. If there are more than 2 points with
107  * the same max value or they are not adjacent, then the new position can not
108  * be computed. This condition is really rare and usually due to poor lighting
109  * or focus, or if the camera is moved too fast and the image is blurred or
110  * distorted.
111  * The range considered is ZPRU/2 - deltarange: ZPRU/2 + deltarange because
112  * considering the whole ZPRU range can lead to error due to the cyclic nature
113  * of the cross correlation (the wrong peak is returned).
114  * This also means that the maximum speed is reduced, in fact if the camera
115  * moves further than deltarange - deltath between two frames the obtained
116  * displacement is wrong and smaller than the correct one.
117  */
118 int fft_displacement(float *in, unsigned int deltarange, float *index) {
```

```
119
120   // First and last index of the max interval, all the points in the interval
121   // must have the same max value
122   unsigned int first_index, last_index;
123
124   // Max value and pointer to the input vector
125   float max, *inptr;
126
127   // Flag used to signal if there are two different max values
128   char many_peak;
129
130   // Initialize the pointer to the input vector, start at ZPRU - deltarange due to
131   // fftshift (negative positions are in the second half of the vector, while
132   // the positive positions are in the first half).
133   inptr = in + ZPRU - deltarange;
134
135 #ifdef DEBUG_FFT
136   volatile unsigned int indexes[3] = {0, 0, 0};
137 #endif
138
139   // Find index of the max in the vector
140   max = 0;
141   first_index = 0;
142   last_index = 0;
143   many_peak = 0;
144   for (int i = 0; i < (deltarange << 1); i++) {
145
146     // New max found
147     if (*inptr > max) {
148       max = *inptr;
149       first_index = i;
150       last_index = i;
151
152       // Clear the flag, the invalid points which were found were not
153       // peaks
154       many_peak= 0;
155
156     // Another element with the same max value found
157     } else if (*inptr == max) {
158
159       // Non-adjacent peak
160       if (i > first_index + 1) {
161         many_peak = 1;
162
163 #ifdef DEBUG_FFT
164         // Store the indexes
165         indexes[0] = first_index;
166         indexes[1] = last_index;
167         indexes[2] = i;
168 #endif
169
170       // Adjacent temporary max found
171       } else {
172         last_index = i;
173       }
174     }
175
176     // After deltarange items move the pointer at the beginning of the input
177     // vector, at the beginning of positive positions, else just increment
178     if (i == deltarange - 1)
179       inptr = in;
180     else
181       inptr++;
182   }
183
184   // If the flag is set after the loop terminated, then the couple invalid
185   // values found were in fact separated max values and the displacement can
186   // not be evaluated
```

```
187   if (many_peak) {
188
189 #ifdef DEBUG_FFT
190     // Print the max interval and the not allowed value
191     fprintf(stderr, "Multiple␣peak␣indexes:␣%d␣%d␣%d\n", indexes[0], indexes[1], indexes[2]);
192 #endif
193
194     return 1;
195   }
196
197   // Compute the effective index using the average between the first index
198   // and the last one. Then subtract deltarange * 2 to evaluate the displacement.
199   *index= (float)((int)(last_index + first_index) - (int)(deltarange << 1)) * 0.5;
200
201   return 0;
202 }
203
204 /* Execute the inverse fft using the given fftwf plan
205  * Wrapper of the fftwf function fftwf_execute
206  */
207 void fft_execute_bw(fftwf_plan p) {
208   fftwf_execute(p);
209 }
210
211 /* Execute the fft using the given fftwf plan
212  * Wrapper of the fftwf function fftwf_execute_dft
213  */
214 void fft_execute_fw(fftwf_plan p, float *in, fftwf_complex *out) {
215   fftwf_execute_dft_r2c(p, in, out);
216 }
217
218 /* Inverse DFT, compute the upsampled displacement in the window around the
219  * peak.
220  * The DFT is computed as a sum of products using its definition (or row by
221  * column product if we think in terms of DFT matrix).
222  * In the loop to compute one element of the DFT has only RESX/2 iterations
223  * instead of RESX because it exploits the hermitian condition which is
224  * satisfied both by the cross correlation in freq domain and by the row of the
225  * DFT matrix (the phase terms of the DFT definition).
226  * Another optimization is using the sine and cosine addition formulas to
227  * compute the real and imaginary parts of the phase terms. This is allowed by
228  * the fact that the argument is incremented between the elements by a fixed
229  * amount.
230  */
231 int fft_dft_bw(fftwf_complex *cross, float delta, float * win_delta) {
232
233   float tmp, arg_inc, shift_fix, rc_product, max;
234   float phase_term[2], phase_term_inc[2] ;
235   unsigned int m, k, id, id_end;
236   fftwf_complex *cc;
237
238 #ifdef DEBUG_FFT
239   float v[WIN];
240   volatile float zr[RESX], zi[RESX];
241 #endif
242
243   // Part of the argument common to all the elements
244   shift_fix = delta * UPS - WIN/2;
245
246   // Initialize the variables used to find the max of the spacial cross
247   // correlation
248   max = 0;
249   id = 0;
250   id_end = 0;
251
252   // Main loop, iterate between the elements of the output spacial cross correlation
253   for (m = 0; m < WIN; m++) {
254
```

```
255     // Initialize the row by column product (sum of products) used to
256     // compute the DFT.
257     // Notice that the result is real, so the imaginary part is not computed.
258     rc_product = 0;
259
260     // Argument increment between the phase terms of the DFT
261     arg_inc = ARG_CONST * (shift_fix + m);
262
263     // Argument increment written as real and imaginary part
264     phase_term_inc[0] = cos(arg_inc);
265     phase_term_inc[1] = sin(arg_inc);
266
267     // Initial phase term
268     phase_term[0] = 1;
269     phase_term[1] = 0;
270
271     // Pointer to the second cross correlation element (skip the dc
272     // component for now)
273     cc = cross + 1;
274
275     // Loop half of the elements of the DFT, compute the sum of products
276     // between the cross correlation and the phase terms
277     for (k = 0; k < RESX2 - 1; k++) {
278
279       // Phase terms computed using sine and cosine addition formulas,
280       // incremented each iteration by the same amount.
281       tmp = phase_term[0] * phase_term_inc[0] -
282         phase_term[1] * phase_term_inc[1];
283       phase_term[1] = phase_term[0] * phase_term_inc[1] +
284         phase_term[1] * phase_term_inc[0];
285       phase_term[0] = tmp;
286
287       // Sum of the product between the cross correlation and the phase
288       // term, DFT definition
289       rc_product += (*cc)[0] * phase_term[0] - (*cc)[1] * phase_term[1];
290
291       // Increment cross correlation pointer
292       cc++;
293
294 #ifdef DEBUG_FFT
295       // Save in a vector real and imaginary part of the computed phase
296       // terms (half DFT matrix)
297       zr[k] = phase_term[0];
298       zi[k] = phase_term[1];
299 #endif
300     }
301
302     // Exploit hermitian redundancy to compute the complete row by column
303     // product.
304     rc_product *= 2;
305
306     // Add the missing terms which are not replicated in the vector.
307     // Phase term -RESX/2, which is the complex conjugate of RESX/2 (hence
308     // the sum instead of the subtraction).
309     rc_product += cross[ZPRU2][0] * (phase_term[0] * phase_term_inc[0] +
310       phase_term[1] * phase_term_inc[1]);
311
312     // The dc component phase term is 1
313     rc_product += cross[0][0];
314
315 #ifdef DEBUG_FFT
316     // Save the computed spatial DFT element in a vector
317     v[m] = rc_product;
318 #endif
319
320     // Search for the peak while computing spacial cross correlation elements
321     // In this case it quite common to have multiple adjacent elements with
322     // the same max value, for this reason the position of the max is
```

```
323       // evaluated as the mean between the position of the first and the last
324       // max elements.
325       if (rc_product > max) {
326         max = rc_product;
327         id = m;
328         id_end = m;
329       } else if (rc_product == max) {
330         id_end = m;
331       }
332
333     }
334   #ifdef DEBUG_FFT
335     print_vector(v, "cross_win", WIN);
336   #endif
337
338     // Return displacement, which is the position of the peak, computed as the
339     // average between the position of the first and the last elements with max
340     // value, with respect to the center of the vector
341     *win_delta = (float)((int)(id + id_end) - (int)WIN) * 0.5;
342
343     // The peak is not centered in the window, it is at one of the extremes.
344     if ((id == 0 || id_end == WIN-1) && id == id_end) {
345       return 1;
346     }
347
348     return 0;
349   }
350
351
352   /* Destroy a fftwf plan
353    * Wrapper of the fftwf function fftwf_destroy_plan
354    */
355   void fft_destroy(fftwf_plan *p) {
356     if (p) {
357       fftwf_destroy_plan(*p);
358       *p = NULL;
359     }
360   }
361
362   /* Read fftwf wisdom, wrapper of fftwf function fftwf_import_wisdom_from_filename.
363    * Filename defined in config.h
364    */
365   int fft_read_wisdom() {
366     return fftwf_import_wisdom_from_filename(WISDOM);
367   }
368
369   /* Write fftwf wisdom, wrapper of fftwf function fftwf_export_wisdom_to_filename.
370    * Filename defined in config.h
371    */
372   int fft_save_wisdom() {
373     return fftwf_export_wisdom_to_filename(WISDOM);
374   }
375
376   #ifdef DEBUG_FFT
377   // Debug, save vector to file
378   static void print_vector(float *vect, char * filename, unsigned int length){
379     char str[64];
380     FILE *fp;
381     sprintf(str, "%s/%s", DEBUGDIR, filename);
382     fp = fopen(str, "w+");
383     for (int i = 0; i < length; i++) {
384       fprintf(fp, "%lf\n", vect[i]);
385     }
386     fclose(fp);
387   }
388   #endif
```

A.4 sensor.c

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <sys/time.h>
4  #include "fft.h"
5  #include "misc.h"
6
7  #ifdef FILECAMERA
8  #include "camera_file.h"
9  #else
10 #include "camera.h"
11 #endif
12
13 #ifdef SERIAL
14 #include "serial.h"
15 #endif
16
17 #ifdef BENCHMARK
18 #include "benchmark.h"
19 #endif
20
21 // Global variable used to stop the main loop when SIGINT is received
22 volatile int flag = 1;
23
24 void intHandler(int unused);
25
26 #ifdef DEBUG
27 static void print_vector(int cycle, float *vect, char * filename, unsigned int length);
28 #endif
29
30 int main() {
31
32   /*****************************
33    * Initialize system         *
34    *****************************/
35
36   // FFTW plans forward and inverse FFT
37   fftwf_plan pfor, pback;
38
39   // Reference frame, captured frame and cross-correlation in frequency
40   // domain. Length RESX/2+1 because Hermitian redundancy is exploited.
41   // f1_fft and f2_fft are used as double buffer to store the reference frame
42   // and the captured frames.
43   fftwf_complex *f1_fft = NULL, *f2_fft = NULL, *cross = NULL;
44
45   // Pointers for the double buffer, reference and current frame
46   fftwf_complex *ref, *curr, *tmp;
47
48   // Cross-correlation (not upsampled) in time domain
49   float *sig = NULL;
50
51   // Vector with the sum of the columns of the captured frame.
52   float *vect;
53
54   // Conversion factor from pixel to millimeters
55   float pix_to_mm;
56
57   // Number of peaks (white bands) in the first reference image. This is used
58   // to compute the threshold for the reference image.
59   unsigned int peakcount;
60
61   // Threshold used to decide when to update the reference image. If the
62   // displacement is larger than this threshold the image is updated.
63   float dpp, deltath;
64
65   // Maximum displacement allowed with respect to the reference frame.
66   unsigned int deltarange;
```

```
67
68    // Total displacement with respect to the first captured frame in pixel units
69    float res;
70
71    // Output displacement in pixel units or millimeters (the latter if
72    // PIXTOMMOUT is set)
73    float output;
74
75    // Position of the reference frame with respect to the first captured frame
76    float ref_position;
77
78    // Displacement in the upsampled neighborhood of the peak, the unit is
79    // upsampled pixel.
80    // Can be fractional: if there are multiple adjacent points with the same
81    // max value the returned position is the center of the max interval.
82    float win_delta;
83
84    // Displacement in pixel units, without upsampling, float because in case
85    // of multiple peaks the mean is returned
86    float delta;
87
88    // Struct used by camera.c to store informations about the device
89    struct camera_dev_t dev;
90
91    // Catch SIGINT
92    struct sigaction sa;
93
94    // Flag set when an error occurs computing the upsampled position
95    unsigned char do_not_update_ref = 0;
96
97    // Return value of the program 0 in case of success, 1 in case of error
98    unsigned int retval = 0;
99
100   // Variables used to evaluate computation time and numerate frames
101 #ifdef DEBUG
102   unsigned int count = 0;
103   struct timeval start, stop;
104 #endif
105
106 #ifdef SERIAL
107   int fd, val;
108
109   fprintf(stderr, "Opening␣and␣configuring␣UART...");
110   fflush(stderr);
111   if ((val = serial_open(&fd))) {
112     fprintf(stderr, "Fail\nFailed␣to␣open␣serial␣%s,␣check␣device␣name␣and␣permissions␣(%d)\n\n", SERIALDEV,
            val);
113     retval = 1;
114     goto cleanup_serial;
115   }
116   fprintf(stderr, "Done\n");
117 #endif
118
119   // Signal handler SIGINT
120   sa.sa_handler = intHandler;
121   sa.sa_flags = 0;
122   sigaction(SIGINT, &sa, NULL);
123
124   // Camera initialization
125   fprintf(stderr, "Preparing␣camera...");
126   fflush(stderr);
127   if (camera_init(&dev) < 0) {
128     fprintf(stderr, "Fail\nCan␣not␣open␣the␣specified␣device.\n\n");
129     retval = 1;
130     goto cleanup_camera_init;
131   }
132   fprintf(stderr, "Done\n");
133
```

```
134   // Fft initialization
135   fprintf(stderr, "Preparing␣plans...");
136   fflush(stderr);
137
138   // Allocate the vector to ensure SIMD compatible alignment.
139   vect = fft_malloc_r(RESX);
140
141   // Allocate the spatial cross correlation vector to be sure that the
142   // alignment is correct for FFTW SIMD.
143   sig = fft_malloc_r(ZPRU);
144
145   // Allocate the vectors allowing them to be aligned for SIMD in FFTW
146   f1_fft = fft_malloc_c(RESX2+1);
147   f2_fft = fft_malloc_c(RESX2+1);
148   cross = fft_malloc_c(ZPRU2+1);
149
150   // Update reference and current frame pointers
151   ref = f1_fft;
152   curr = f2_fft;
153
154   // Read wisdom file
155   fft_read_wisdom();
156
157   // Initialize plans
158   pfor = fft_init_fw(vect, f1_fft);
159   pback = fft_init_bw(cross, sig);
160
161   // Save wisdom
162   fft_save_wisdom();
163   fprintf(stderr, "Done\n");
164
165   // Setup the camera
166   fprintf(stderr, "Starting␣the␣camera␣capture...");
167   fflush(stderr);
168   if (camera_start(&dev) < 0) {
169     fprintf(stderr, "Fail\n");
170     retval = 1;
171     goto cleanup_camera_start;
172   }
173   fprintf(stderr, "Done\n");
174
175
176   /****************************
177    * Reference frame          *
178    ****************************/
179
180   fprintf(stderr, "Preparing␣reference␣vector...");
181   fflush(stderr);
182
183   // Get the vector of the referece frame.
184   // Throw away the first frames waiting for the camera to adjust after
185   // startup. Sometimes brightness changes in the first few frames captured.
186   for (int i = 0; i < 20; i++) {
187     if (camera_get_frame(&dev, &vect) < 0) {
188       fprintf(stderr, "Fail\nFailed␣to␣get␣frame\n");
189       retval = 1;
190       goto cleanup_camera_start;
191     }
192   }
193
194 #ifdef DEBUG
195   unsigned int refcount = 0;
196   print_vector(0, vect, "vector", RESX);
197 #endif
198
199 #ifdef PIXTOMMOUT
200   // Compute the conversion factor from pixels to millimeters and the number of peaks
201   if (pixel_to_mm_factor(vect, &pix_to_mm, &peakcount)) {
```

```
202       fprintf(stderr, "Fail");
203       fprintf(stderr, "\nFailed␣to␣evaluate␣conversion␣factor,␣check␣that␣at␣least␣one␣triplet␣is␣always␣
              present␣in␣the␣frame␣(narrow,␣medium␣and␣wide␣black␣bands)\n");
204       retval = 1;
205       goto cleanup_camera_start;
206     }
207   #else
208     // If PIXTOMMOUT is not set only the number of peaks is used
209     pixel_to_mm_factor(vect, &pix_to_mm, &peakcount);
210   #endif
211
212   #ifdef ZEROTH
213     threshold_apply(vect);
214   #endif
215
216     // Compute the displacement threshold for the new reference image
217     dpp = (peakcount) ? RESX / peakcount : 0;
218     deltath = DELTATHFACT * dpp;
219
220     // If deltath is too large set deltath to a known save value (if the number
221     // of white bands is too low).
222     if (deltath == 0 || deltath > DELTATHFALLBACK) deltath = DELTATHFALLBACK;
223
224     // Maximum allowed displacement with respect to the reference frame
225     deltarange = ZPUPS * (int)(dpp * DELTARANGEFACT);
226     if (deltarange > (ZPRU >> 1) || dpp == 0) deltarange = (ZPRU >> 1);
227
228     // Compute the Fourier transform of the reference frame
229     fft_execute_fw(pfor, vect, ref);
230     fprintf(stderr, "Done\n");
231     fflush(stderr);
232
233     // Initialize the position of the reference frame
234     ref_position = 0.0;
235
236     /*****************************
237      * Main loop        *
238      *****************************/
239
240     // Initialize the position
241     res = 0;
242     win_delta = 0;
243     output = 0;
244
245   #ifndef BENCHMARK
246
247     // Run until flag is true (until SIGINT is received)
248     while (flag) {
249   #else
250     if (bench_open()) {
251       fprintf(stderr, "Failed␣to␣open␣bench␣file\n");
252       flag = 0;
253     }
254     bench_start();
255     // Run BENCHN times or until flag is true
256     for (int i = 0; i < BENCHN && flag; i++) {
257   #endif
258
259   #ifdef DEBUG
260       // Debug mode, get a frame on key press
261       fprintf(stderr, "Press␣enter␣to␣continue\n");
262       getchar();
263   #endif
264
265       // Get the vector computed summing the columns the camera frame buffer
266       if (camera_get_frame(&dev, &vect) < 0) {
267         fprintf(stderr, "Failed␣to␣get␣frame\n");
268         retval = 1;
```

```
269       goto cleanup_camera_start;
270     }
271
272 #ifdef DEBUG
273     // In debug mode the conversion is recomputed each iteration to check
274     // the correct behavior, even when the flag is not set
275     pixel_to_mm_factor(vect, &pix_to_mm, &peakcount);
276
277     // Debug, save the captured frame
278     print_vector(count, vect, "vector", RESX);
279     gettimeofday(&start, NULL);
280
281 #elif defined BENCHMARK
282     bench_start_time();
283 #endif
284
285 #ifdef ZEROTH
286     threshold_apply(vect);
287 #endif
288
289     // Compute the FFT of the captured frame
290     fft_execute_fw(pfor, vect, curr);
291
292     // Cross correlation between reference frame and captured frame and
293     // apply the zero-padding if required
294     fft_xcorr(ref, curr, cross);
295
296     // Inverse FFT
297     fft_execute_bw(pback);
298
299 #ifdef DEBUG_FFT
300     print_vector(count, sig, "cross", ZPRU);
301 #endif
302
303     // Compute the displacement, position of peak with respect to the
304     // center.
305     if (fft_displacement(sig, deltarange, &delta)) {
306 #ifdef DEBUG
307       fprintf(stderr, "Warning:␣error␣computing␣the␣pixel␣displacement,␣multiple␣peaks␣found.␣The␣current␣
              frame␣is␣skipped␣and␣the␣previous␣position␣is␣returned.␣Please␣check␣camera␣lighting,␣speed,␣
              focus␣and␣bands␣verticality.␣The␣current␣frame␣will␣be␣skipped\n");
308 #endif
309       do_not_update_ref = 1;
310
311     } else {
312       // Compute the delta in pixel units
313       delta = delta / (float) ZPUPS;
314
315       // Inverse DFT, upsampled displacement in the neighborhood of the peak
316       // Returns 1 if the window is not centered correctly and in this
317       // case does not update the reference image.
318       if (fft_dft_bw(cross, delta, &win_delta)) {
319 #ifdef DEBUG
320         fprintf(stderr, "Warning:␣peak␣not␣centered␣in␣the␣window,␣the␣reference␣image␣won't␣be␣updated\n");
321 #endif
322         do_not_update_ref = 1;
323       }
324
325       // Compute the position with respect to the first captured frame
326       res = ref_position + delta + win_delta / (float) UPS;
327
328 #ifdef PIXTOMMOUT
329       // Output in mm evaluated using the factor computed using the
330       // transitions of the first frame
331       output = res * pix_to_mm;
332
333 #elif defined (PIXTOMMCAL)
334       // Output in mm using the manually provided factor
```

```
335       output = res * (float) PIXTOMMCAL;
336
337   #else
338       // Output in pixels
339       output = res;
340   #endif
341
342     }
343
344   #ifdef DEBUG
345     // Debug information
346     gettimeofday(&stop, NULL);
347     fprintf(stderr, "Computation time: %f\n", (stop.tv_sec - start.tv_sec) + (stop.tv_usec - start.tv_usec)
            / 1e6);
348     fprintf(stderr, "Delta: %f, Win delta: %f, Tot delta: %lf, Position in pixels: %lf\n", delta, win_delta,
             delta + win_delta/UPS, res);
349     fprintf(stderr, "Current frame: %u, Reference frame: %u\n", count, refcount);
350     fprintf(stderr, "Pixel to mm conversion factor: %f\n", pix_to_mm);
351   #elif defined BENCHMARK
352     bench_stop_time();
353     bench_write_computation();
354   #endif
355
356   #ifdef STDOUT
357     // Print the displacement with respect to the first frame to stdout
358     fprintf(stdout, "%f\n", output);
359   #endif
360
361   #ifdef SERIAL
362     // Write the displacement to serial
363     serial_write(fd, output);
364   #endif
365
366     // If the displacement in pixel units is larger than the defined
367     // threshold then update the reference vector in the frequency domain
368     // and the position of the reference frame.
369     if (!do_not_update_ref && ( delta > deltath || delta < -deltath)) {
370
371       // Update the reference vector in frequency domain using the buffers
372       tmp = ref;
373       ref = curr;
374       curr = tmp;
375
376       // Update the position of the reference frame with respect of the
377       // first frame
378       ref_position = res;
379
380   #ifdef DEBUG
381       // Debug info
382       fprintf(stderr, "Updated reference position: %lf, Count: %u\n", ref_position, count);
383       refcount = count;
384   #endif
385     }
386
387     // Reset flag
388     do_not_update_ref = 0;
389
390   #ifdef DEBUG
391     // Update counter for average time computation
392     count++;
393   #endif
394   }
395
396   #ifdef BENCHMARK
397     bench_stop();
398     bench_close();
399   #endif
400
```

```
401
402   /***************************
403    *  Uninitialize           *
404    ***************************/
405
406   // Camera stop capture
407 cleanup_camera_start:
408   fprintf(stderr, "\nStopping␣camera␣capture...");
409   fflush(stderr);
410   camera_stop(&dev);
411   fprintf(stderr, "Done\n");
412
413   // Destroy FFTW plans
414   fprintf(stderr, "Destroying␣FFTW␣plans...");
415   fflush(stderr);
416   fft_destroy(&pfor);
417   fft_destroy(&pback);
418
419   // Free allocated FFTW vectors
420   fft_free((void **)&f1_fft);
421   fft_free((void **)&f2_fft);
422   fft_free((void **)&cross);
423   fft_free((void **)&sig);
424   fft_free((void **)&vect);
425   fprintf(stderr, "Done\n");
426
427   // Camera uninitialize
428 cleanup_camera_init:
429   fprintf(stderr, "Closing␣camera...");
430   fflush(stderr);
431   camera_close(&dev);
432   fprintf(stderr, "Done\n");
433
434 #ifdef SERIAL
435 cleanup_serial:
436   fprintf(stderr, "Closing␣UART...");
437   fflush(stderr);
438   serial_close(fd);
439   fprintf(stderr, "Done\n");
440 #endif
441
442   return retval;
443 }
444
445 // SIGINT handler, set the flag to false to stop the loop
446 void intHandler(int unused) {
447   flag = 0;
448 }
449
450 #ifdef DEBUG
451 // Debug, save vector to file
452 static void print_vector(int cycle, float *vect, char * filename, unsigned int length){
453   char str[64];
454   FILE *fp;
455   sprintf(str, "%s/%s_%d", DEBUGDIR, filename, cycle);
456   fp = fopen(str, "w+");
457   if (fp != NULL) {
458     for (int i = 0; i < length; i++) {
459       fprintf(fp, "%lf\n", vect[i]);
460     }
461     fclose(fp);
462   }
463 }
464 #endif
```

Bibliography

[1] Claudio Passerone. «Low Cost High Resolution Position Sensor with Sub-micron Accuracy». In: *International Conference on Advanced Technology & Sciences* (Rome, Italy, Nov. 2016), pp. 182–188.

[2] M. Guizar-Sicarios and J.R. Fienup S.T. Thurman. «Efficient subpixel image registration algorithms». In: *Optics Letters* 33.2 (Jan. 2008).

[3] John W. Eaton et al. *GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations*. 2020. URL: `https://www.gnu.org/software/octave/doc/v5.2.0/`.

[4] Matteo Frigo and Steven G. Johnson. *FFTW version 3.3.10 manual*. 2021. URL: `https://www.fftw.org/#documentation`.

[5] *V4L2 API manual*. 2021. URL: `https://www.linuxtv.org/docs.php`.

[6] *Homepage of libjpeg-turbo project*. 2021. URL: `https://libjpeg-turbo.org/`.

[7] *Home page of FFMPEG project*. 2021. URL: `https://ffmpeg.org/`.

[8] A.J. Fleming. «A review of nanometer resolution position sensors: Operation and performance». In: *Sensors and Actuators A: Physical* 190 (Feb. 2013). Ed. by Elseview, pp. 106–126.