Depth maps and image detection

November 2020

Abstract

The efficient construction of depth maps is a key part of 3D reconstruction methods. Perceiving depth is something most humans do unconsciously, however for computer vision applications, like in autonomous vehicles, constructing depths maps requires a high degree of accuracy, scalability and real-time capability.

To this end we construct depth maps from scratch and explore the foundations upon which they depend: convolution and cross-correlation. By exploring these two methods we see the importance of employing efficient computation techniques, and appreciate the challenge of creating a real-time capability. Although Python libraries exist to perform most of the tasks in this report, we start from the basics to build foundational knowledge.

Finally we apply one of the developed methods (2D convolution) to create animated depth maps of autonomous driving scenarios and compare the performance to a developed Python image library.

Contents

1 Cross-correlation

The cross-correlation is used in signal processing and template matching. The cross-correlation is a measure of similarity between two vectors (1D), matrices (2D) or tensors for higher dimensional applications and is essentially the dot product of two tensors.

The cross-correlation has applications in many research fields where two signals need to be compared. In astronomy unresolved objects in space produce *background radiation*. The cross-correlation has been proposed as a method to probe this background radiation by cross correlating different energy 'maps' of the sky to draw conclusions about the source of signals from outer space. Recently Ammazzalorso et al. (2020) used the cross-correlation to probe the signal from unresolved gamma rays to determine their source. It was hypothesised these gamma rays are likely to be blazars, which are galaxies that shoot out *really, really fast* particles. Here on Earth cross-correlation methods are used to count fish stock by placing sensors underwater and passively recording 'fish sounds', Hossain and Hossen (2020) demonstrated how to use three unequally spaced sensors and the cross-correlation to calculate fish populations. Another interesting application is for image detection, in medical imaging computer-aided detection systems (CAD) are used to help specialists in the search and identification of objects of interest. Fan et al. (2002) proposed using cross-correlation to identify lung nodules by cross correlating an image of a lung with a template of a cancerous nodules.

The above examples use the 1D or 2D cross-correlation to detect fish (1D), explore signals from deep space (2D) or help physicians find suspicious lung nodules (2D). In the next two sections the 1D and 2D cross-correlation are introduced, including algorithm flow charts to demonstrate possible implementations, followed by two simple applications to demonstrate how cross-correlation works.

1.1 Introduction to the 1D cross-correlation

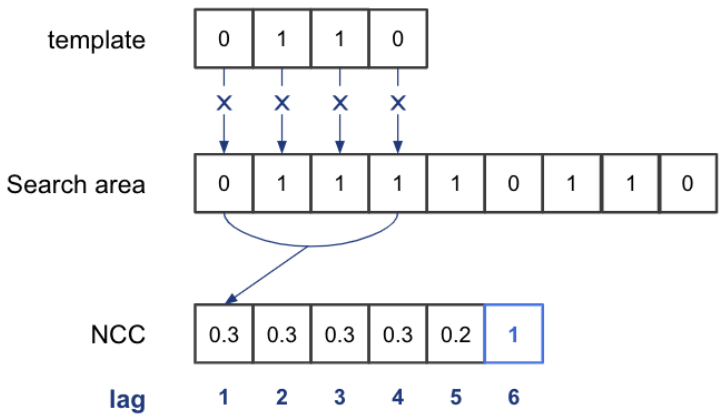


Figure 1.1: *Calculating the NCC at different positions (lags) for two vectors: pattern and template. Highlighted is the NCC value when the template aligns with the first four points of the search area (when* lag = 1*). Sliding the template along the search area is how the NCC is calculated for lags* 1, 2, 3, 4, 5 and 6*. The NCC is maximal where the template lines up with the pattern (at* lag = 6*).*

The 1D normalised cross-correlation (NCC) is simply the normalised element-wise multiplication of two vectors at different locations. Expressed mathematically for two 1D vectors F and G, the 1D NCC is:

$$(\mathrm{F} \star \mathrm{G}) = \frac{\sum_{i=0}^{k} F_i.G_i}{\sqrt{\sum_{i=0}^{k} F_i^2 . \sum_{i=0}^{n} G_i^2}}, \tag{1.1}$$

where $F_i \epsilon \mathrm{F} \forall\, i \epsilon \{0, 1, \ldots k\}$. Here k is the length of F. Similarly $G_i \epsilon \mathrm{G} \forall\, i \epsilon \{0, 1, \ldots k\}$ and k is the length of G. Eqn. 1.1 requires F and G be the same length. If F and G are not the same length, the shorter vector can padded with zeros, or the longer vector can be trimmed down to the same length.

Figure 1.1 visualises the calculation of the NCC for two vectors. The pattern 'slides' across the template and the normalised dot product is calculated at every location or *lag*. In Fig. 1.1 the NCC for each lag is calculated for locations 1 - 6; for example at lag location 1, the NCC is $(0.1 + 1.1 + 1.1 + 0.1)/6$. We see the maximum value for NCC is 1 at position 6, which corresponds to the lag between the pattern and the template. Thus the template lags six positions behind the search area.

When F = G the NCC is maximal, and reduces to 1:

$$\begin{aligned}(\mathrm{F} \star \mathrm{F}) &= \frac{\sum_{i=0}^{k} F_i^2}{\sqrt{\left(\sum_{i=0}^{k} F_i^2\right)^2}} \\ &= 1.\end{aligned} \tag{1.2}$$

This property is useful to find lag between two signals which are *exactly* the same.

One implementation of Eqn. 1.1 is shown in the algorithm flow chart in Fig. 1.2. This approach involves calculating the NCC of sampled vector segments from F and G. The sampling ensures the NCC doesn't need to be calculated for the whole length of either F or G. This implementation is used in the next section, for which more details can be found in Sec. A.

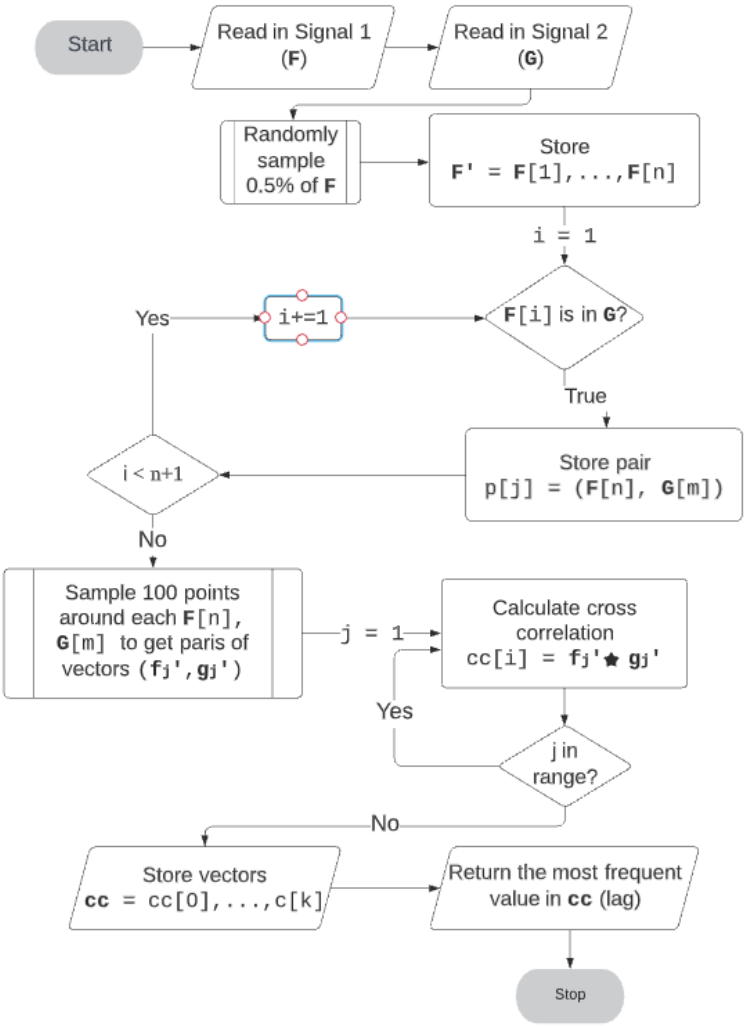
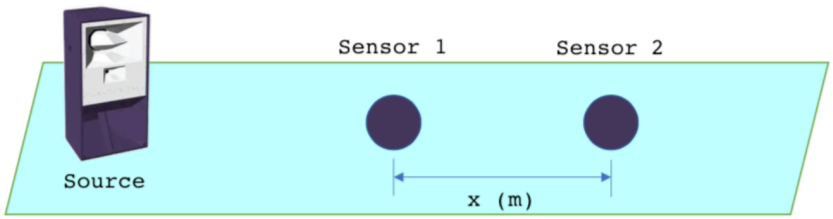
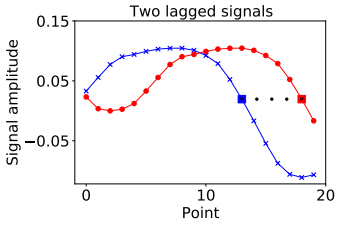


Figure 1.2: *Algorithm flow chart of a method used to implement Eqn. 1.1.*



(a) Schematic of the experimental system used to collect an audio signal generated by the Source at a frequency of 44.1 kHz *which propagates at* 333 $\mathrm{m.s^{-1}}$. *The audio signal is collected at Sensor 1 and Sensor 2 located* x m *apart which causes a time lag* Δt *between the recorded signals* `s1` *and* `s2`.



(b) A plot showing one signal with blue crosses lagging behind another signal with red dots. The squares show two points which line up in both signals. The black dots highlight the lag $\Delta p = 4$.

Figure 1.3: *Schematic of the experimental set up used to record a signal with two sensors (left), these sensors record two signals which lead/lag each other.*

```
INFO on line 103: The lag between signals is:
       50082.0 points, which
       corresponds to 378.17 m
INFO on line 104: sample of points used to
       confirm where s1=s2: 0.36 %
       (631 points)
INFO on line 105: time taken: 2.07 s
```

Listing 1.1: *Output for the code used to implement the 1D NCC.*

1.1.1 Application of the 1D normalised cross-correlation: finding the distance between two sensors

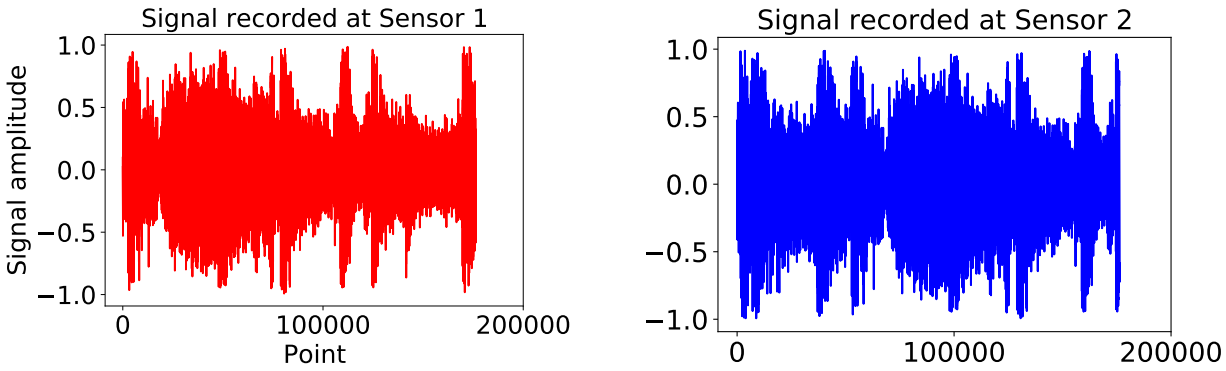
Similar to the method used by Hossain and Hossen (2020) to detect fish populations, we use a similar, but simplified experimental set-up and use the NCC to find the time delay and thus distance between between two sensors. Out of the water this type of analysis has applications in auditory scene analysis where it can be used for speech recognition and sound detection (Beutler (2005)).

Figure. 1.3a is a schematic of the experimental set-up: two sensors, Sensor 1 and Sensor 2 are set a distance x apart and record a globally non-repeating source signal, called Signal 1 and Signal 2 respectively. When the outputs of the two sensors are plotted together we expect to see something similar to Fig. 1.3b: the signals are offset by a fixed amount (or lag). Here the signal at Sensor 1 is recorded first (shown as a red line with circle markers) then the signal is then recorded at Sensor 2 (the blue line with cross markers). The big black squares show where the two signals are aligned and the black dots show how many points the blue signal lags behind the red signal. We use the NCC to detect this lag.

The source generates a signal at 44.1 kHz which propagates at 333 $\mathrm{m.s^{-1}}$. The outputs from Sensor 1 and Sensor 2 are plotted in Fig. 1.4a and Fig. 1.4b respectively. Using the the NCC, the lag between the signal from Sensor 1 and Sensor 2 is determined to be $50,082$ points - the output is shown in Code Section 1.1. The lag is visualised in Fig. 1.5a by plotting the data from from Sensor 1 lagged by $50,082$ points in blue and the data from Sensor 2 (unchanged) in red on the same plot - the purple region is where the signals from Sensor 1 and Sensor 2 align perfectly. To further demonstrate the lag, we zoom into the first 200 points of Signal 1 in Fig. 1.5b and plot the data from Signal 2 lagged by $50,082$ points in blue. Clearly the data points from Signal 1 and Signal 2 align visually confirming the lag between Signal 1 and Signal 2 is $50,082$ points.

A lag of $50,082$ points places the two sensors a distance $x = 378.19$ m apart: the time between each point is 22.7 μs (the inverse of the signal frequency) and the signal propagates at 333 $\mathrm{m.s^{-1}}$ giving distance $50,082 \times 22.7\,\mu\mathrm{s} \times 333\,\mathrm{m.s^{-1}} = 378.19$ m

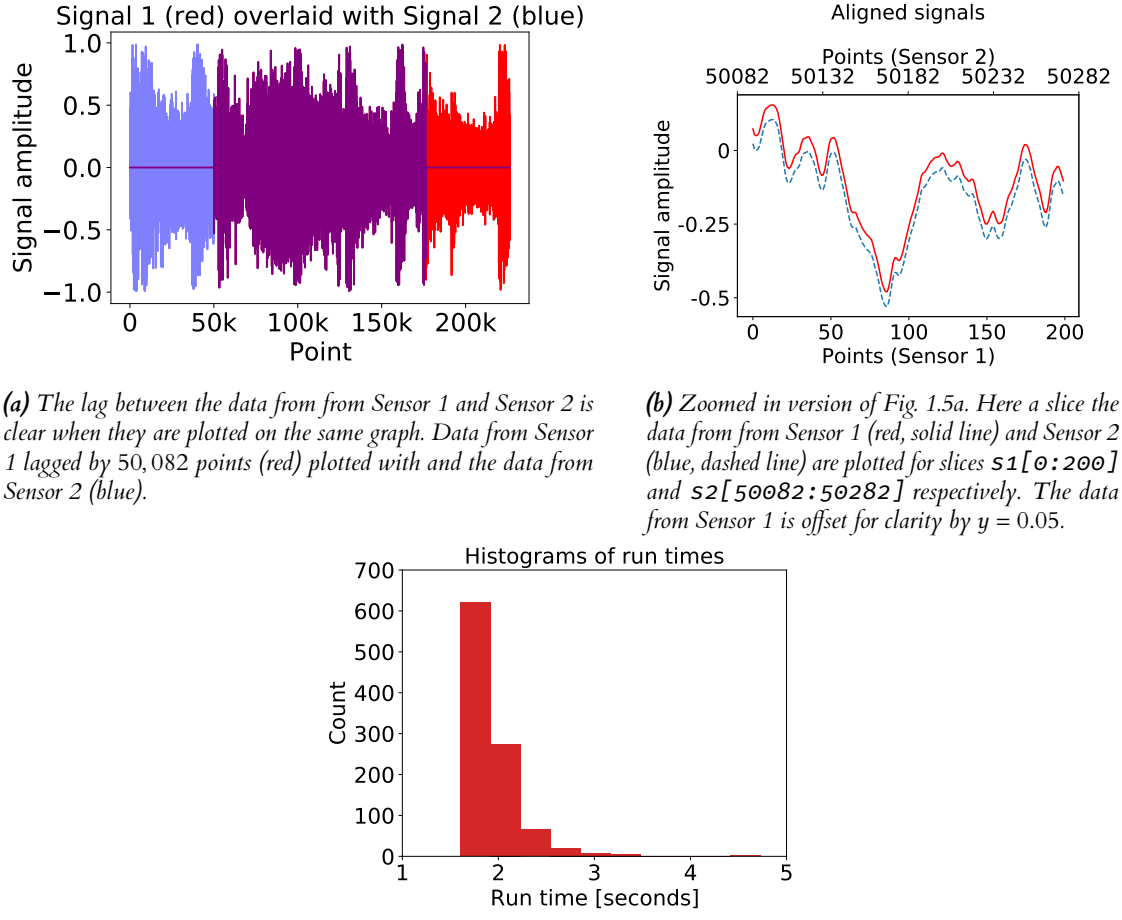
The run time of the code used to calculate the 1D NCC is around ~ 2 s. Fig. 1.5c is a histogram of a thousand runs



(a) A plot of the audio signal collected by Sensor 1.

(b) A plot of the audio signal collected by Sensor 2.

Figure 1.4: *Two plots of the signal recorded at Sensors 1 and 2.*



(a) The lag between the data from from Sensor 1 and Sensor 2 is clear when they are plotted on the same graph. Data from Sensor 1 lagged by 50,082 *points (red) plotted with and the data from Sensor 2 (blue).*

(b) Zoomed in version of Fig. 1.5a. Here a slice the data from from Sensor 1 (red, solid line) and Sensor 2 (blue, dashed line) are plotted for slices `s1[0:200]` *and* `s2[50082:50282]` *respectively. The data from Sensor 1 is offset for clarity by* $y = 0.05$.

(c) A histogram of a thousand runs showing a range of $\sim 1.5 - 3.5$ s.

Figure 1.5: *Signal alignment and run time when applying the NCC to signal detection.*

showing a range of ~ 1.5 – 3.5 s. The range of run-times is due to the nature of the code taking varying sample sizes to calculate the 1D NCC, for more detail see Appendix A.

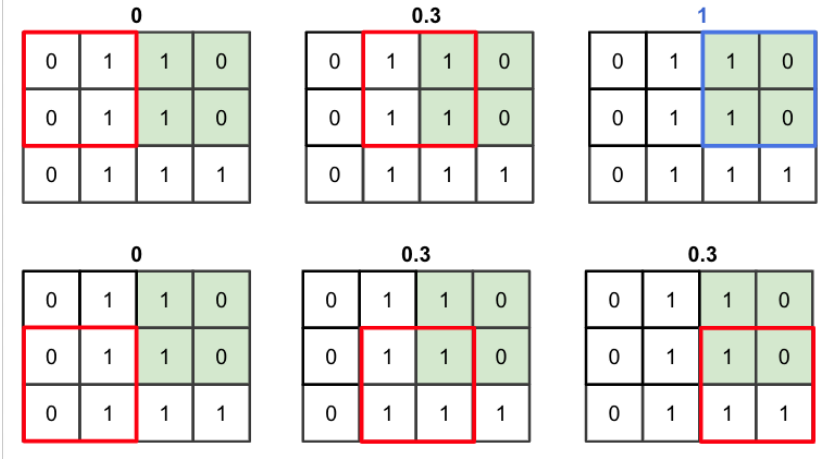
HANDLING LOCAL REPETITION

Looking at the signals in Figs. 1.4a and 1.4b it doesn't look like the signals repeat. However nearly two million points are recorded by Sensor 1 and Sensor 2 making it difficult to visually inspect for any local repetition. To mitigate the rare chance Sensor 1 and Sensor 2 have locally repeating signals, the NCC is calculated for many samples along each signal. See Appendix A for details.

1.2 Introduction to the 2D NCC

The cross-correlation in 2D, is similar to the 1D cross-correlation except it involves taking the sum of the point-wise multiplication of matrix elements from a 2D matrix instead of a 1D vector. The normalised cross-correlation is visualised in Fig. 1.6a, the matrix 'template' is coloured in green and occurs in the top right-hand corner of the search area. The NCC can be expressed as a matrix, as shown in Fig. 1.6a where the maximal value indicates the location of the template in the search area (top, right corner). Figure 1.6b shows the template moving to each position in the search area and the NCC value, reported in the cross-correlation matrix.

(a) NCC of two matrices.



(b) Step-by-step demonstration of the calculated NCC of the template in the search area. Notice when the NCC is maximal when the template is found in the NCC.

Figure 1.6: *The NCC.*

Expressing the above mathematically, the NCC of two matrices F and G with matrix elements $F_{ij} \in$ F and $G_{ij} \in$ G is:

$$(\mathrm{F} \star \mathrm{G}) = \frac{\sum_{i=0}^{k} F_{ij}.G_{ij}}{\sqrt{\sum_{i=0}^{k} F_{ij}^2 . \sum_{i=0}^{n} G_{ij}^2}}, \tag{1.3}$$

where i, j are the number of rows and columns in F (and G) respectively.

Similar for the 1D case the maximum value for the NCC indicates where F = G.

$$(\mathrm{F} \star \mathrm{G}) = \frac{\sum_{i=0}^{k} F_{ij}^2}{\sqrt{\left(\sum_{i=0}^{k} F_{ij}^2\right)^2}} = 1. \tag{1.4}$$

```
import numpy as np
from PIL import Image

my_image = Image.open(image_path)
my_image_as_an_array = np.asarray(my_image)
```

Listing 1.2: *Converting an image at location* `image_path` *to a NumPy array.*

1.2.1 Applying the 2D normalised cross-correlation to images

Converting an image to a matrix enables us to apply the NCC and use its properties for template matching. Python image processing libraries exist to convert images to matrices, Code Section 1.2 is an example of an image at location `image_path` being opened using the Python Image Library (PIL) and converted to a matrix (or tensor) using NumPy. The image type determines the dimensions of the NumPy array, for example a greyscale image produces a 2D array, where each element of the array maps to the greyscale value. If the image is an RGBA image, where RGBA stands for the (Red, Green, Blue, Alpha) values of the image (alpha is the transparency), then the dimension of an n px $\times$ m px image will result in a $n \times m \times 4$ tensor, where the last dimension of length 4 encodes the RGBA values. When an RGBA image is converted to a greyscale image the $n \times m \times 4$ tensor will reduce to a $n \times m$ matrix. In this work we use both RGBA and greyscale images.

Once an image is converted to a NumPy array it's possible to use the NCC to find the lag. An implementation of Eqn. 1.3 given two RGBA images is shown in the algorithm flow chart in Fig. 1.7. This approach is similar to the implementation in Fig. 1.2 in that it samples regions of the template and uses these to reduce the search space in the pattern. More details on this implementation are found in Sec. B.

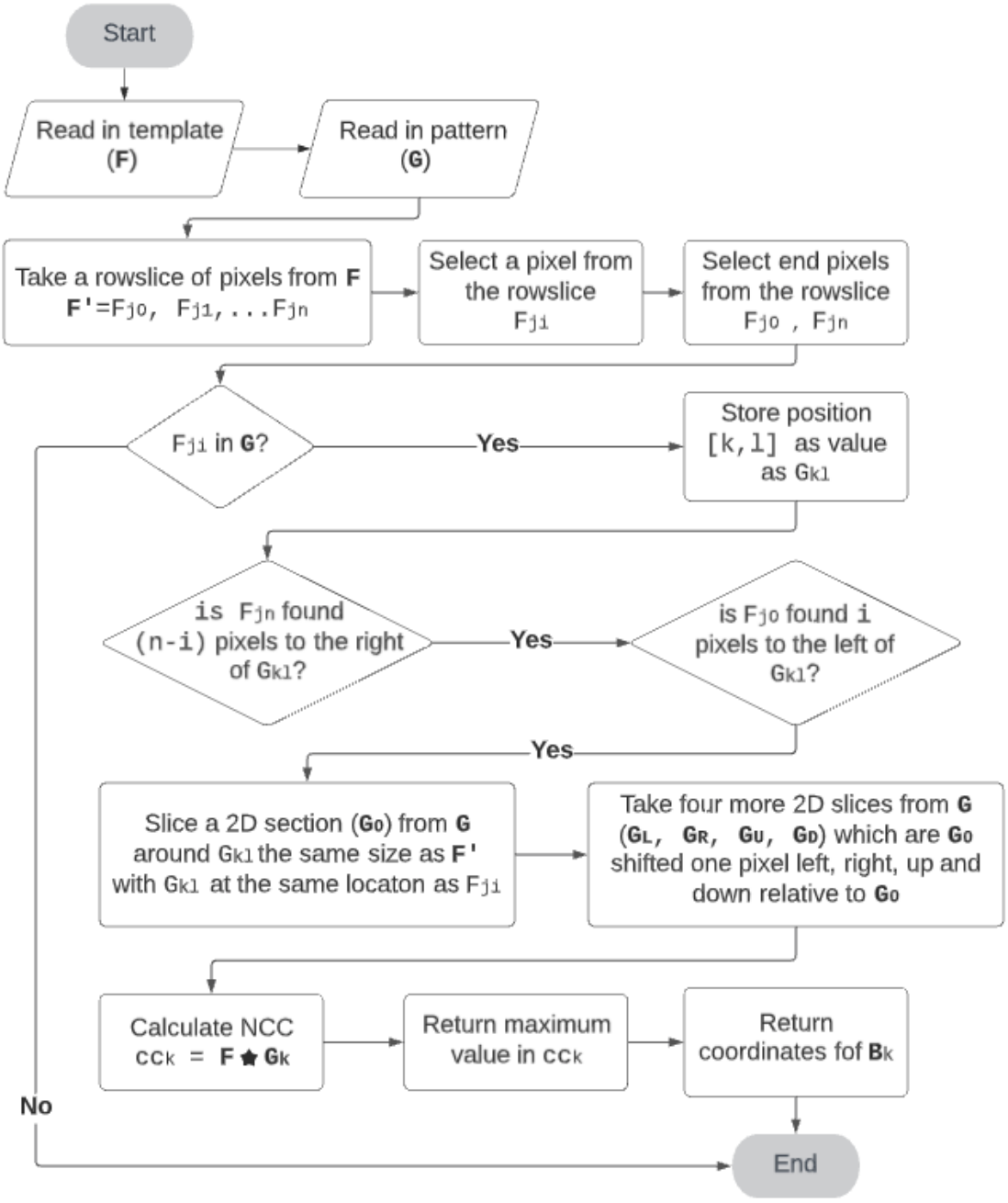


Figure 1.7: *Algorithm flow chart of one method used to implement Eqn. 1.3.*

(a) Rocketman. *(b)* Rocketman can be found in this visually 'busy' maze.

Figure 1.8: *Rocketman and the maze in which he's found.*

```
INFO on line 142: Rocketman found at lag (651, 982).
             bottom row: 528, top row: 651, col left: 982, col right: 1093
INFO on line 150: Time taken: 0.4926331043243408s
```

Listing 1.3: *Output for the code used to implement the 2D NCC and find Rocketman in the maze.*

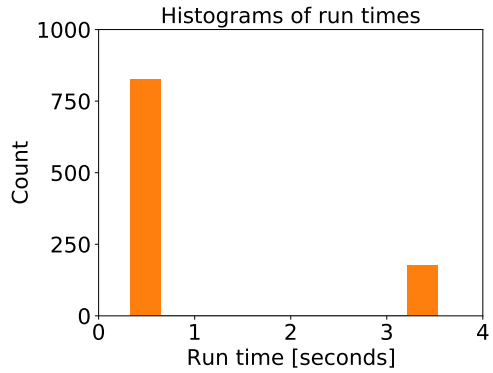
1.2.2 Application of the 2D normalised cross-correlation: template matching

Template matching involves looking for a smaller image in a bigger image, such as computer aided lung-cancer detection system previously discussed. Instead of looking for suspicious lung nodules in lung images like Fan et al. (2002), we'll perform a much simpler operation trying to find 'Rocketman' shown in Fig. 1.8a in the maze in Fig. 1.8b.

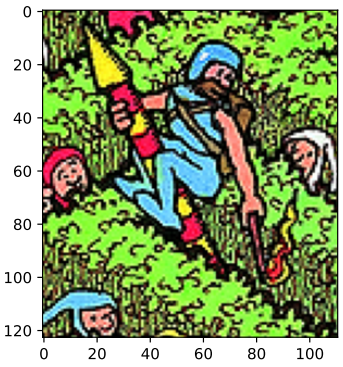
To start Rocketman and the maze were converted into three-dimensional tensors and the NCC was calculated. For a 3D tensor of dimension $n \times m \times 4$ the normalised cross correlation is simply calculating the dot product of the RGBA values in the template and pattern, i.e. $(R_T, G_T, B_T, A_T) \cdot (R_P, G_P, B_P, A_P)$ at each location $[n, m]$. Using this approach the maximum NCC value found the location of Rocketman: the top, left corner of Rocketman can be found at (row, column)= $(651, 982)$ - the output is shown in Code Section 1.3.

Rather than implementing a brute-force approach and calculating the NCC for almost all rows and columns in the maze the search space is reduce from 3,732,596 (the number of pixels in the maze) to at most ~ 800 locations. This is achieved by selecting a random pixel in Rocketman and finding pixels with the same RGBA values inside the maze. A 5×5 grid is then constructed at these locations and the NCC is calculated to find Rocketman.

While the current approach is fast, it works using colour pixels with an RGBA value, this means if the template or pattern is changed, e.g. the colours are dulled, the script won't work. Operating in grey-scale will correct these limitations, however this would rob the author of the opportunity to play with higher dimensional NumPy arrays. More information on the approach is detailed further in Appendix B.



(a) *A histogram of 1000 runs demonstrating it usually takes under a second to find Rocketman. The variable nature of reducing the search-space results in longer times.*



(b) *Using the results in Code Section 1.3 to slice the maze (*`answer=template_array[528:651,981:1092,:]`*) Rocketman is found.*

Figure 1.9: *Results from taking the 2D NCC.*

2 Introduction to the 1D and 2D convolution

In the previous section I avoided calculating the NCC over whole 1D or 2D regions of the patterns (the signal and maze) by creating a search space much smaller than than the original inputs. Generally though, the NCC is slow and it may not be possible to speed up the computation for all applications. Thus for time sensitive applications another approach is required, such as using the convolution.

The computational efficiency of the convolution has been noted in applications from building an automated fabric detection system (AS et al. (2013)) to visual tracking (Ward et al. (2013)). Both of these applications are time sensitive - ideally a fabric inspection system would work 'on-loom' to inspect the whole width of a fabric *in-situ*; such a system can't wait minutes for the NCC to return results. Similarly applications for visual tracking systems can't wait minutes if they are tracking a moving object.

Finding the lag between two 1D or 2D inputs using the convolution is fast and easy, given two signals f and $\mathbf{g}$ the lag is calculated using the steps below:

1. Take the FFT of $f(t)$ and $\mathbf{g}(t)$ to create the transformed vectors $\hat{f}(\omega)$ and $\hat{g}(\omega)$ for frequencies ω.
2. Take the conjugate of one of the signals, say $\hat{g}(\omega) \rightarrow \overline{\hat{g}(\omega)}$.
3. Point-wise multiply $\hat{f}(\omega)$ and $\overline{\hat{g}(\omega)}$: $\hat{f}(\omega).\overline{\hat{g}(\omega)}$.
4. Take the inverse FFT to get the lag.

The success of this method is due to the fact the convolution does not have to be calculated discretely at many points in a search area. The success of this method hinges on the convolution theorem, which states (under the right conditions) the Fourier transform of two signals is the element-wise product of their individual Fourier transforms. This enables us to point-wise multiply $\hat{f}(\omega)$ and $\overline{\hat{g}(\omega)}$, and take the inverse FFT of their product to get the lag. For more detail see Appendix C.

WHY DOES THE CONVOLUTION WORK?

It's not clear why taking two signals and applying a combination of steps involving the FFT , complex conjugate, a multiplication and the inverse FFT finally results in the lag between these two signals. To find out why and how these steps work see Appendix C for detail.

In the next two sections the 1D and 2D convolution are introduced, and are applied to the same examples discussed in Chapter 1. Given much of the material was introduced in Chapter 1 these sections are relatively brief and are contained to presenting the results and demonstrating the speed of this approach.

2.1 Application of the 1D convolution: finding the distance between two sensors

The convolution is used to calculate the lag and thus the distance between the same two sensors discussed in Section 1.1. See Section 1.1 for the experimental set up and general approach.

As is the case in Section 1.1 using the convolution yields the same results: the sensors were calculated to be 378.17 m apart. Typical output is shown in Code Section 2.1:

The run time for one run using the convolution is 1.71 s, this lies in the range as the 1D NCC (1.5 – 3.5 s). While the run time is comparable between the NCC and convolution, the convolution involved no effort to reduce the search space, thus the implementation is simpler saving a lot of coding time and making bug fixes easier due to its simplicity.

```
INFO on line 62: when first signal is at i=0 value is: 0.023158,
second signal is ahead by 50082 with value 0.023158

The distance between points is: 50082,
difference in time is: 1.1356462585034013s,
and the distance apart is: 378.1702040816326m
```

Listing 2.1: *Output for the code used to implement the 1D convolution to find the lag between two 1D signals.*

```
the lag is: (528, 982)
run time is 2.802893877029419s
```

Listing 2.2: *Output for the code used to implement the 2D convolution to find Rocketman in the maze.*

2.2 Application of the 2D convolution: template matching

The task here is the same as in Section 1.2: finding Rocketman (Fig. 1.8a) in a busy maze (Fig. 1.8). See Section 1.2 for an introduction to Rocketman and the maze. One difference in the approach here is the RGBA image was first converted to a greyscale image - meaning this approach is more robust to colour changes in the image, e.g. any changes in transparency should not impact calculating the lag.

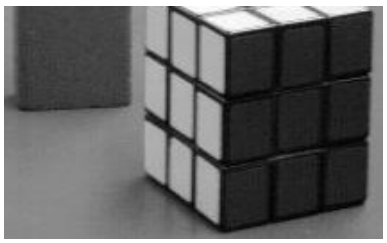
As is the case in Section 1.2 using the convolution yields the same results: the top-left corner of Rocketman can be found at row, column: (651,982). Typical output is shown in Code Section 2.2:

The run time for one run using the convolution is $\sim 3\,\mathrm{s}$ in the above example. This run time is in the range of $0.5 - 3.5\,\mathrm{s}$ obtained using the 1D NCC. However similar to the previous section, implementing the 2D convolution involved no effort in reducing the search space and resulted in simpler code.

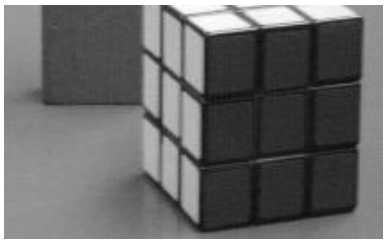
3 Depth maps

Depth perception, or rather, being able to see how far away objects are, is crucial to be able to move around in our world. One way the the brain perceives depth is through stereo vision: essentially when we look at an object a slightly different image is projected onto each retina. These two images are triangulated by the brain to give the perception of depth.

Depth perception works because if an object is further away the image it projects onto both retinas barely changes, however if an object is close the image it projects onto each retina is noticeably different. This phenomena can be visualised by inspecting the figures in Fig. 3.1: the left and right image are taken from cameras slightly separated to mimic the human eye. In the background is a block and in the foreground is a Rubik's cube. Notice block in the background appears to shift more from left to right, than the Rubik's cube in the foreground.



(a) *A Rubik's cube viewed through a left lens .*



(b) *A Rubik's cube viewed through a right lens.*

Figure 3.1: *Two images taken from a slightly different perspective, not too dissimilar to how the same scene has a slightly different projection into each eye. Notice the block in the foreground appears to change less in comparison to the Rubik's cube in the foreground.*

This shift in background objects relative to foreground objects is how we perceive depth. Computationally this is achieved by dividing one image (say the left image) into a grid as shown in Fig. 3.2 (left box). From the left image, one grid is chosen (called the template) and a region in the right image (called the search area) is searched until the best match for the template in the search area. This *(*best match) is found using the 2D convolution, and it returns the location of the template in the search area. If the location of the template is at location $[i, j]$ and is found at $[k, l]$ in the search area, then how much the template as moved from between the left and right image is the euclidean distance $lag = \sqrt{(i-k)^2 + (j-l)^2}$. Images close to the camera will have a small lag, and images far away will have a larger lag, thus if the lag is projected onto a 2D surface and coloured by value images far away (with a small lag) can be distinguished from close images (with a large lag) by their colours.

Different search parameters need to be optimised to produce a depth map: the dimensions of the window (or grid) size can be changed, the search region's geometry or size can be changed as shown in Fig. 3.2 (middle box) and the window overlap can be adjusted to cover more area as shown in Fig. 3.2 (right box). Modifying these parameters can change the look of the depth map dramatically, and are discussed in the next section.

3.1 Creating depth maps with stereo vision image pairs

Figure 3.3 is a depth maps of a Rubik's cube constructed from Figs. 3.1a and 3.1b and shows the result of varying the window size, overlap and search areas. From top to bottom, each row has a different parameter varied: window size, overlap and search area. The depth maps outlined in red all have the same parameters for comparison: a window size of 20 px × 20 px, an overlap of 0.9 and a search area of of 20 px × 20 px. From the depth maps in Fig. 3.3 a clearer depth map is obtained with a high overlap, a comparable search area to the window size and a windows size around

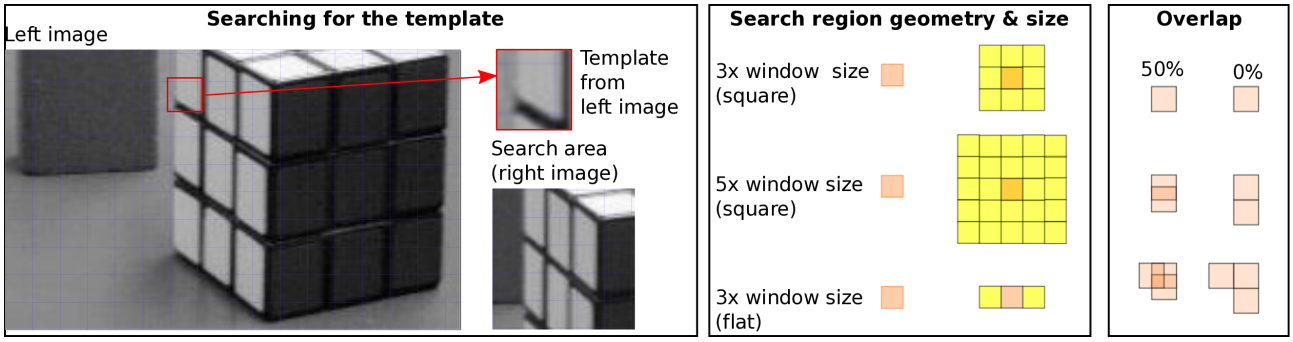


Figure 3.2: *Depth maps are created by looking for small sections of the left image, in the right image. To save searching the whole right image, only a small region is searched as shown here.*

20 px × 20 px. Focusing on the depth map outlined in red, the corner of the Rubik's cube is green at the corners (nearer to the lens) but fades to yellow as the face recedes from the lens. The background is difficult to resolve, likely because the background is very bland so most of the background looks the same in the left and the right image giving the illusion of no lag and thus no depth. Figure 3.4a compares features in the depth map with features in the right

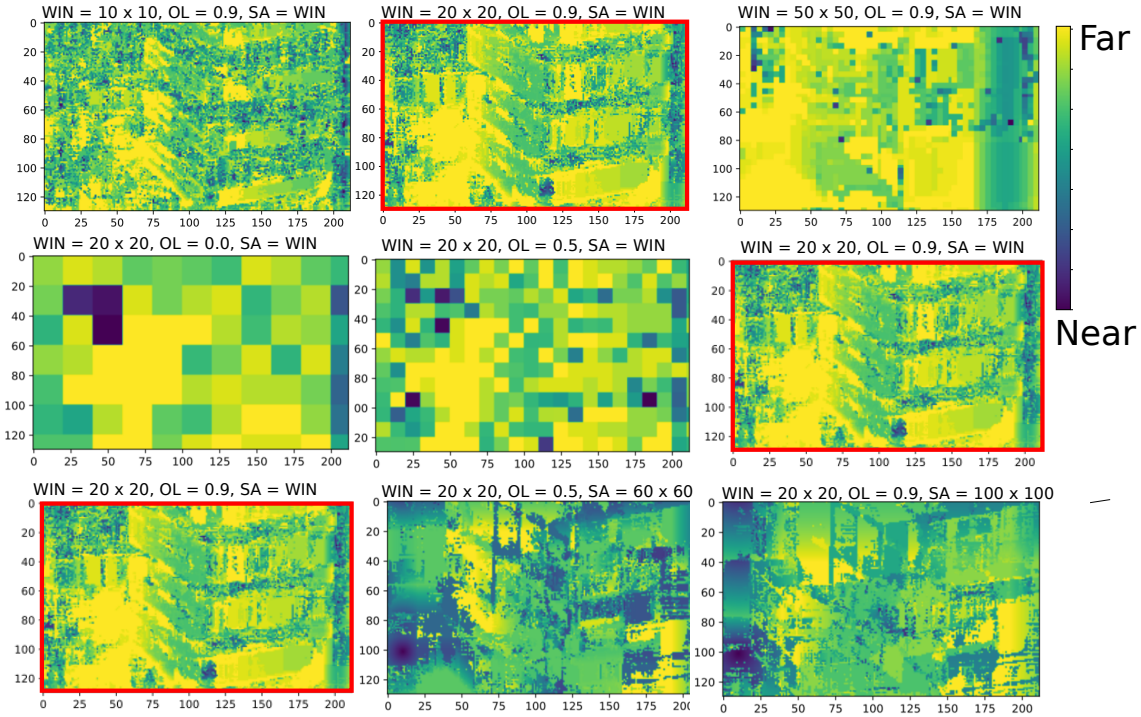


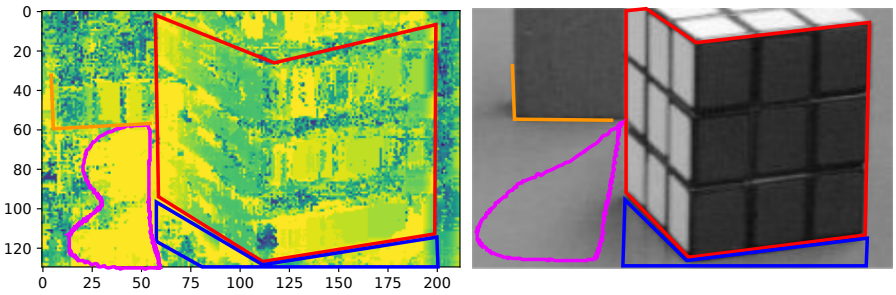
Figure 3.3: *Depth maps of a Rubik's cube constructed from Figs. 3.1a and 3.1b. Each plot is for a different window size (WIN) given in pixels, overlap (OL) and square search area (SA) given in pixels. The red outlines are for the clearest depth maps with the same parameters.*

image. The Rubik's cubes surfaces are resolved (outlined in red), and some structure underneath the cube (the shadow) appears to be captured to a small degree (outlined in blue). The shadows and light change in each image, which appears to be yellow (outlined in pink). Interesting the light region (outlined in pink) is all shown as "far" (yellow), this could be due to shadows causing colour gradients which are hard to match thus the convolution detecting a large lag. Some structure in the background looks like the block, however because the block has a uniform surface grid segments from the left and right image may look similar resulting in a small lag and thus having a dark green to purple colour coding it as being 'near'.

Depth maps are also constructed for two more image pairs: giraffes and a QR code. The left, right image pairs of giraffes are shown in Figs. 3.4b and 3.4c respectively. The Depth map in Fig. 3.5a picks up a giraffe in the foreground (blue), a second giraffe behind it (in green) as well background structures (in yellow) and finally a tree (light green)

coloured appropriately for the depth. The fence structure is picked up but the depth is not resolved, likely due to the pattern repeating and thus a clear match was not found when performing the template match.

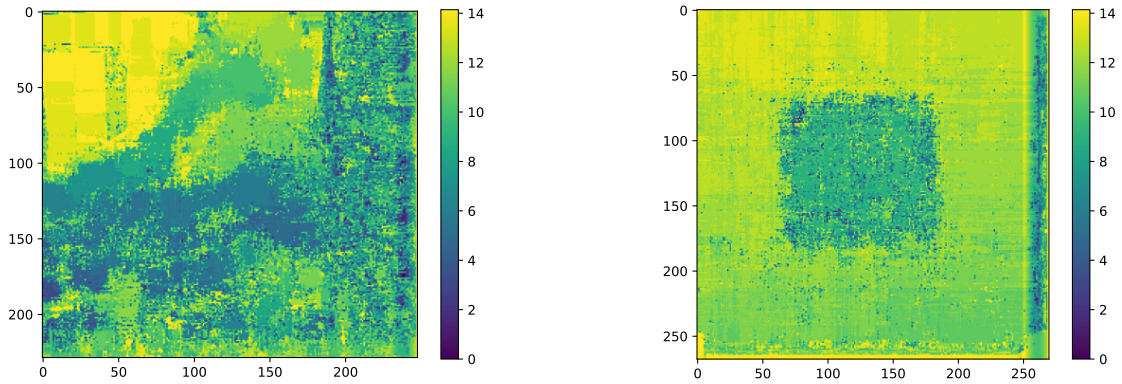
The left and right perspective of the QR code is shown in Figs. 3.4d and 3.4e respectively. A raised square in the centre is revealed by the depth map. Interestingly the background varies from green to yellow (bottom to top). This is only a difference of 2 uncalibrated units however a more uniform look may be achieved by using an adaptive windowing approach, perhaps using larger windows for the background and smaller windows closer to the centre of the QR code.



(a) *A depth map of a Rubik's cube compared to the right image in Fig. 3.1b.*

(b) *Giraffes (left lens).* ***(c)*** *Giraffes (right lens).* ***(d)*** *QR code (left lens).* ***(e)*** *QR code (right lens)*

Figure 3.4: *Two paired images used to construct depth maps.*



(a) *A depth map of giraffes constructed from Figs. 3.4b and 3.4c.* ***(b)*** *A depth map of a QR code constructed from Figs. 3.4d and 3.4e.*

Figure 3.5: *Depth maps for paired images of giraffes and a QR code. Both images have a with a window size of* 20 px × 20 px *and overlap* = 0.90 *and the search region isn't padded.*

4 Exploring stereo vision in autonomous driving scenarios

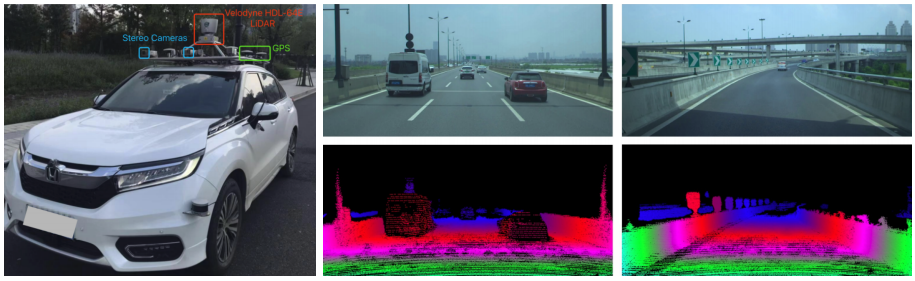


Figure 4.1: *A sample of the data (top, middle and right) from the open source data set DrivingStereo: A Large-Scale Dataset for Stereo Matching in Autonomous Driving Scenarios created by Yang et al. (2019). Their depth maps are shown bottom, middle and right. The SUV used for image collection is shown left.*

When driving we are able to recognise objects in our 3D environment and perform a high-level analysis of our surroundings before executing a number of essential tasks: planning our path, avoiding collisions, and sticking to the speed limit. An independent, autonomous vehicle must be able to perform the same sequence of events requiring at least two problems to be addressed: depth perception and object identification. In this chapter we explore depth perception using the work from Sec. 3.

To this end we build a proof-of-concept by applying the convolution method developed in Sec. 2.2 to a part of the publicly available dataset *DrivingStereo: A Large-Scale Dataset for Stereo Matching in Autonomous Driving Scenarios* created by Yang et al. (2019).

Figure 4.1 shows a sample of the image data available (top, middle and right) recorded by the *DrivingStereo* data acquisition vehicle (left). Depths maps produced by the authors Yang et al. (2019) are shown bottom, middle and right. Image collection is done by an SUV with mounted colour cameras on top (Basler ACA1920-40GC). Two of those cameras are chosen to be stereo pairs, the top centre and top right, they are 54 cm apart with a field of view of 50 °. Images are captured at a rate of 10 Hz. More details about the image capture are found in the supplementary materials provided by Yang et al. (2019).

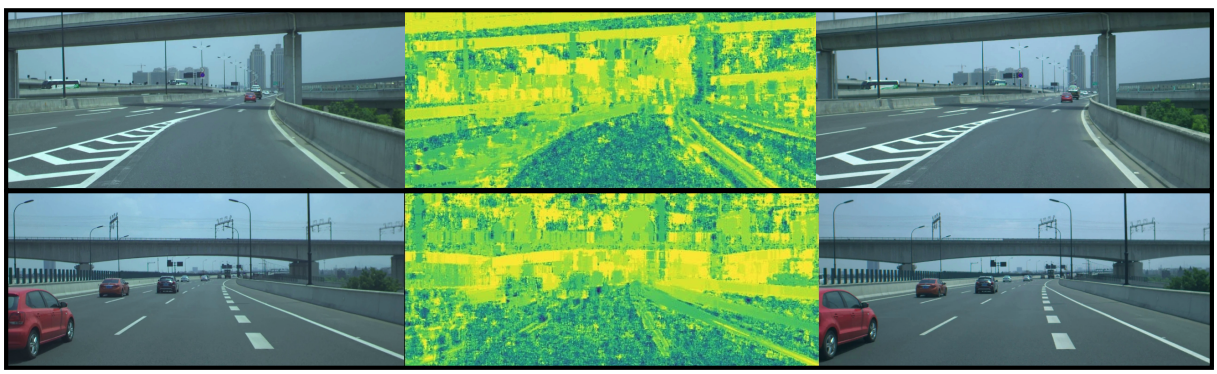


Figure 4.2: *Stereo image pairs (left and right) with uncalibrated depth maps in between (purple/green is close, yellow is far). The depth maps were created with a window size of* 20 px × 20 px*, an overlap of* 0.95 *with the search region unpadded.*

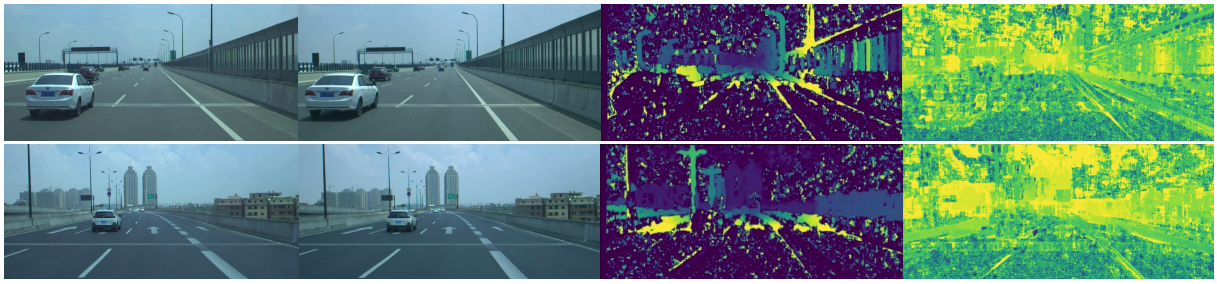


Figure 4.3: *From left to right: Stereo image pairs left and right, an uncalibrated depth map produced using Open CV and an uncalibrated depth map produced using the convolution method. The Open CV depth map used the function* `StereoBM_create` *and has inputs* `numDisparities=16` *and* `blockSize=11`*, where* `blockSize` *is equivalent to the window size and* `numDisparities` *is the number of 'depth layers' or levels the algorithm will attempt to return.)*

Figure 4.4: *Left and middle images are produced by Menze and Geiger (2015). Top, left image highlights moving objects. Top, middle image captures the optical flow of the scene (a colour map of the relative motion between the driver and the road environment). Bottom, left is a depth map with the moving objects highlighted. Bottom, middle is the 'optical flow ground truth' (actual velocities and distances). Right: an example of scene segmentation using Facebook's Detectron suite (Girshick et al. (2018)).*

Figure 4.2 shows the left stereo image, depth map produced by the convolution method (overlap = 0.95 or ~ 19 px with square size = 20 px) and right stereo image. This approach identifies large features from the environment such as roads, some road markings, barriers and bridges. Smaller features such as cars are obscured, though they can be seen when the images are stitched together to form a video. Each image took on average 111 s to produce, which is far too slow for any practical usage on the road. To illustrate this fact a study of 100 drivers by Dingus et al. (2006) reported crashes took place over time frames of 1.1 s to 6.7 s, thus an autonomous vehicle will have to process data and make a decision in sub-second windows, rendering this approach an academic exercise..

In comparison the depth maps produced by the Python *Open Computer Vision Library*(*Open CV*) developed by Bradski (2000) are computed ~ 1000 times faster at 0.28 s per image pair. Fig. 4.3 compares the depth maps from *Open CV* and the convolution method: from left to right is the left stereo image, right stereo image, depth map produced by *Open CV* and the depth map produced by the convolution method. Both depth maps use the same colouring scheme, though as they are uncalibrated the colours do not translate to a meaningful or consistent depth in real space. From close to far the colour range is purple → green → yellow. The depth maps produced by *Open CV*, loosely speaking appear 'less blocky', the scene is somewhat decomposed into objects (lights, barriers, cars) and there are rows of colour somewhat distinguishing the foreground (purple) from the background (yellow and green layers). While cars can be seen by their contrast to the background, they are coloured the same colour as the road, highlighting that other techniques are required to detect objects on the road.

Clearly depth maps do not distinguish very different objects at a similar depth, like roads, road markings and cars. This means autonomous vehicles need more than just depth maps for navigation - they need to perform object detection (e.g. detect parked cars, signage), moving object detection (e.g. detect driving cars, cyclists), perform object tracking over time, and detect roads and lanes. These components fall under Scene Flow Estimation (SFE): the task of understanding "the geometric layout of a scene as well as its decomposition into individually moving objects." As an example of what is possible, Fig. 4.4 shows an example of SFE by Menze and Geiger (2015) and Facebook's object detection product *Detectron* (Girshick et al. (2018)). The left and middle images in Fig. 4.4 contain *optical flow information*, which is used to deduce the relative motion between an object and the observer (i.e. the autonomous car and the environment) - notice the cars are coloured different colours, which encodes their relative motion. The output of Facebook's Detectron is shown in Fig. 4.4 (right) where people and bikes are identified. The type of information

encoded in the images in Fig. 4.4 demonstrate the type of information required to make driving decisions: what objects are moving and in what direction, how fast are objects moving and an awareness of the different kinds of objects in an environment.

4.1 Conclusion

As an exploratory method the convolution method is capable of demonstrating depth maps can be used to identify important features in autonomous driving scenarios. However when compared to the depth maps produced by *Open CV*, the depth maps produced by *Open CV* were produced faster and the image appeared smoother with a foreground, background and improved object definition. We found depth maps, even those produced by *Open CV* did not clearly identify objects like cars, which is to be expected. Applying convolution method implemented in this project to driving data highlighted the importance of real-time computation for autonomous driving, it took ~ 40 hour to compute ~ 1500 frame, or a few minutes of driving time.

5 Summary

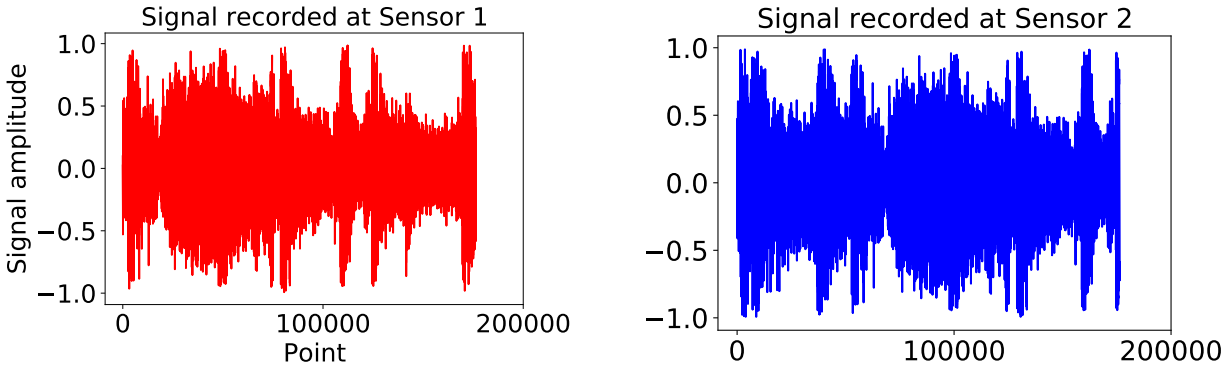
In this work we introduced the 1D and 2D cross-correlation and convolution and demonstrated their application to signal and image detection. The 2D convolution was then used to produce depth maps and eventually applied to the publicly available dataset *DrivingStereo* by Yang et al. (2019).

Stereo image pairs in the *DrivingStereo* dataset were collected at a rate of 10 kHz, yet our method took ~ 110 s to process *one* image pair roughly translating to ~ 40 hours of computation for ~ 1500 frames recorded in a time much, much shorter than 40 hours. We compared this processing time to the same task implemented using the Python library *Open CV* which performed about 1000 times faster.

From the depth maps created by using both the *Open CV* library, as well as the convolution method, objects like cars were not resolved. This highlights why other techniques such as object detection and knowing their relative motion are also required in autonomous driving applications.

A | Implementing the 1D cross-correlation

In this appendix we'll present how the cross-correlation is calculated for two, long, 1D signals. Recall the two signals to be compared are shown in Fig. A.1a and Fig. A.1b; each signal is 176,401 points long.



(a) A plot of the audio signal collected by Sensor 1.

(b) A plot of the audio signal collected by Sensor 2.

Figure A.1: *Two plots of the signal recorded at Sensors 1 and 2.*

Using the approach introduced in Section 1.1, the lag between the signal recorded and Sensor 1 and Sensor 2 will be found by taking a slice of the signal from one sensor to create a template. This template then slides along the full length of the other signal (called the *search area*) and the NCC is calculated. A schematic of this approach is shown in Fig. A.2, the template is given a different colour outline for each new position it takes.

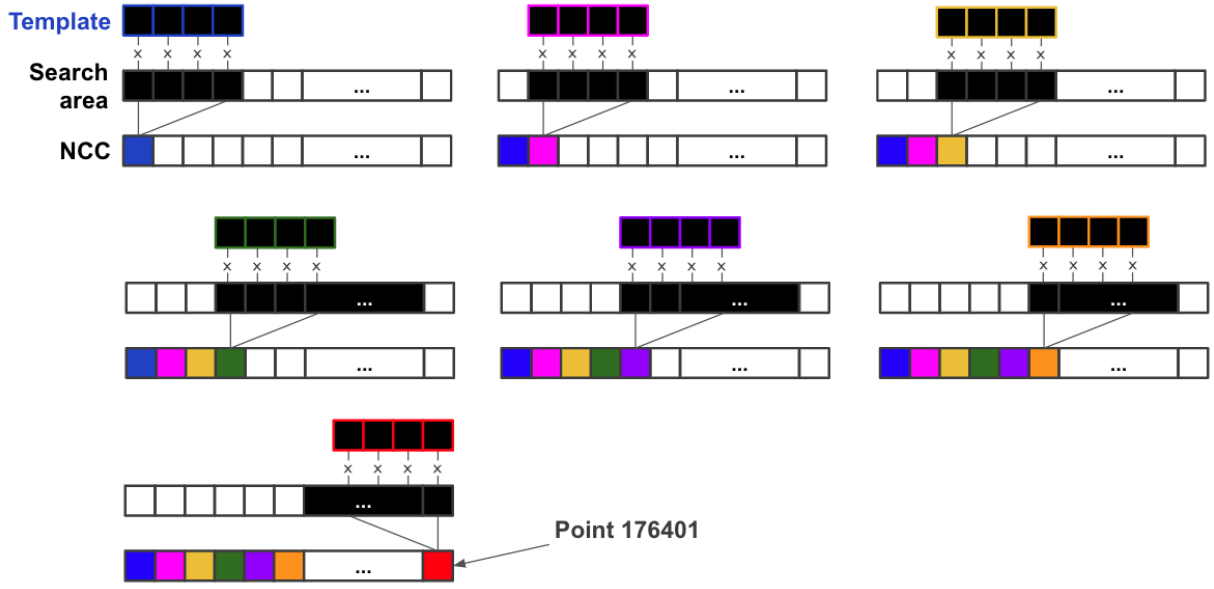


Figure A.2: *For two signals, such as those represented in Fig. A.1a and Fig. A.1b, the lag between to two is traditionally calculated by constructing a template from one signal (outlined in colour and filled in black) and calculating the NCC along the full length of the other signal (filled and outlined in black).*

Using the approach depicted in Fig. A.2 takes a long time because the NCC is calculated with the template at almost ~ 176k different positions. Alternatively, this compute time can be reduced if the NCC is calculated for a sample of relevant regions in the *Search area*. The approach implemented in this work, is shown in Fig. A.3 and is as follows:

1. Randomly sample (without replacement) a fraction of the points in recorded by Signal 1, let's call this set F = $\{f_1, f_2, ..., f_i\}$.
2. Look for these points in Signal 2 and record their location in Signal 2. Let's call the set of matches in Signal 2 G which are found at locations j = $\{j_1, j_2, ..., j_k\}$. Note, because the signals are lagged it's possible some of points in Signal 1, won't be found in Signal 2.
3. Sample 100 points in the vicinity of the matched pairs in Signal 1 and Signal 2: sample 100 points around f_i in Signal 1, and sample 100 points around each matched location in Signal 2. We can write each matched pair (g′, f′) such that f′ = $\{f_l, f_{l+1}, ..., f_{l+99}\}$ and g′ = $\{g_l, g_{l+1}, ..., g_{l+99}\}$.
4. Calculate the cross-correlation for each (f′ ⋆ g′).
5. Find the lag by finding where (f′ ⋆ g′) is equal to one (or maximal).

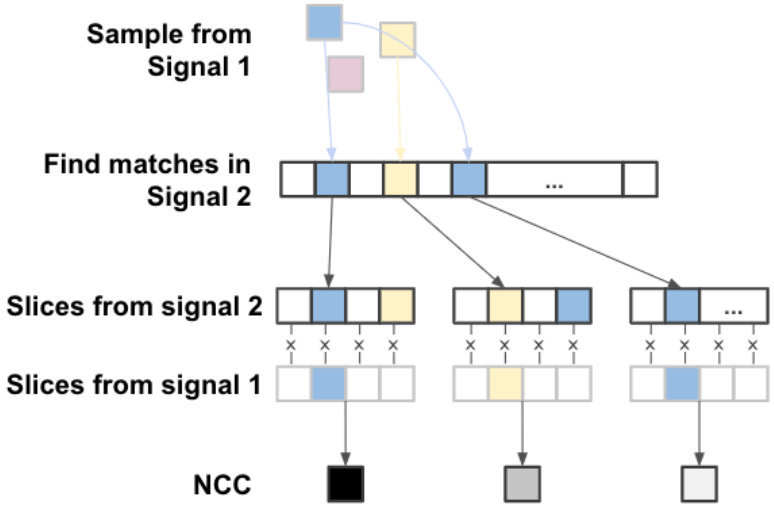


Figure A.3: *An alternative method for calculating the lag, essentially random templates are constructed from Signal 1 and 'strategic' slices are taken from Signal 2 as a collection of search areas. The templates and matching slices are used to calculate the NCC, and the lag is returned when for all templates/search areas for which the NCC*= 1.

The benefit of this approach is, not only is it faster, but the lag is calculated at a sample of regions along the data from Sensors 1 and 2 - if there's any local repetition likely to fool the NCC, the most frequently calculated lag can be returned.

B | Implementing the 2D cross-correlation for template detection

(a) Rocketman. *(b)* Rocketman can be found in this visually 'busy' maze.

Figure B.1: *Rocketman and the maze in which he's found.*

As in the 1D case, to find Rocketman in Fig.B.1a the 2D NCC is calculated using the NumPy array representation of both images. The maze contains 3,732,596 pixels, thus moving through the maze pixel by pixel as depicted in Fig. B.1, places an upper limit in the order of many millions for the number of times the 2D NCC needs to be calculated. This could take minutes.

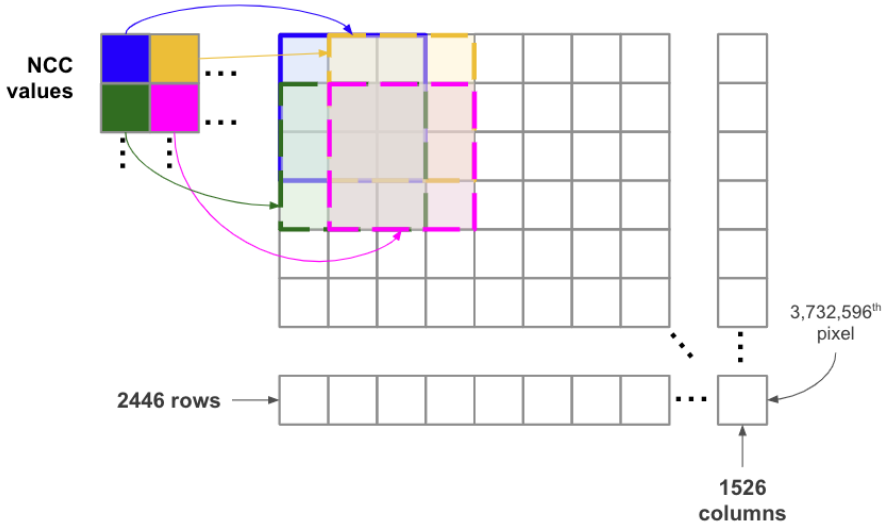
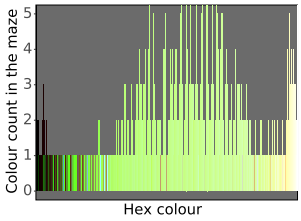


Figure B.2: *Schematic of the the brute force calculation of the 2D NCC. The template of Rocketman is shown as a* 3×3 *grid (not to scale), different colours are used to show the position of the Rocketman array covering the region of the maze array in one-pixel steps.*

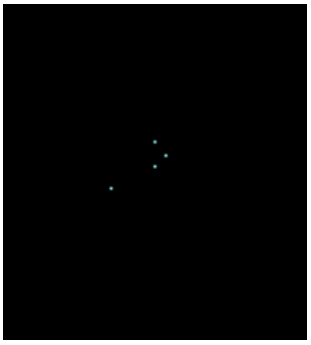
In this work we instead find Rockman quite quickly by narrowing down the search area from 3,732,596 down to tens of points. The approach taken to reduce the search space is:

1. Take a non-transparent slice of pixels from Rocketman as shown in Fig. B.2 (left) say along row j, and choose one pixel $P_{i,j}$ from this slice.
2. In the maze M I look for all instances of the pixel $P_{i,j}$ such that $M_{k,l} = P_{i,j}$.
   Fig. B.2 (middle) shows how matching coloured pixels reduces the search space considerably. Fig. B.2 is a histogram showing the number of times each colour appears in the maze (each bar is the same colour it counts). With the exception of black pixels (which occur at 141 locations in the maze), all other colours occur 13 times or less in the maze.
   Fig. B.2 (right) illustrates how the search space is reduced to a smaller area, in this case where $P_{i,j}$ is a blue pixel.
3. The search area is further reduced by using a method inspired by the Boyer–Moore string-search algorithm: I look at the last non-transparent pixels on either side of the Rocketman-slice $P_{i-n,j}$, $P_{i+m,j}$, and look for the same pixels, at the same location with respect to $M_{k,l}$ in the maze. This approach is illustrated in Fig. B.4.
4. With the search locations narrowed down to a few locations, the 2D cross-correlation can be used to confirm the location of Rocketman, by finding where $(\mathrm{F} \star \mathrm{G})_{i,j}$ is maximal.

(a) *A ten pixel thick slice of the Rocketman for illustrative purposes (I only use a one-pixel slice in my approach).*



(b) *A histogram of each RGB colour in the maze, each bar is coloured the pixel it counts. The y-axis is truncated at $y = 5$.*



(c) *An example of finding one type of pixels (blue) in a portion of the maze. All pixels different to blue are coloured black for contrast.*

Figure B.3

Because the template of Rocketman is not exactly the same as the 2D slice taken from the maze (Rocketman is surrounded by transparent pixels) the normalised cross-correlation will not return 1, so instead I search for the maximum cross-correlation in the search region.

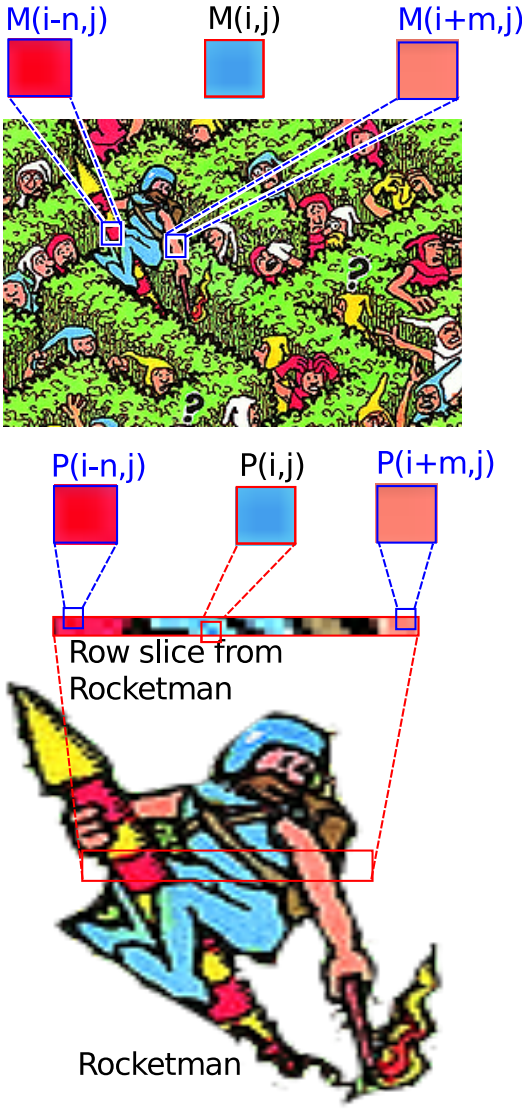


Figure B.4: *A row of pixels is chosen from Rocketman, from which three pixels are chosen: two end pixels shown here in shades of red with the original blue pixel in between. Next we find all blue pixels in the maze (see also Fig. B.3c) and look along the horizontal plane for the two red pixels to further narrow down the search. In these narrowed down regions we then perform the cross-correlation in a* 5×5 *grid.*

C | Deriving the lag from the convolution

To show why the above steps work in deriving the lag between two identical 1D signals I'll first derive the time-shift property of the Fourier Transform (FT) and then show how the lag is obtained. Although this project works with discrete points, I'll assume the results derived below for continuous signals holds for discrete signals.

C.1 Time-shift property of the Fourier transform

The FT acts on a signal $f(t)$ via the operator $\mathcal{F}$ such that $f(t)$ is mapped onto frequency space $\hat{\chi}(\omega)$ via Eqn. C.1:

$$\begin{aligned}\mathcal{F}\{f(t)\} &= \int_{-\infty}^{\infty} f(t)e^{-i\omega t}\,dt \\ &= \hat{\chi}(\omega).\end{aligned} \tag{C.1}$$

Now introduce a time-shift $t - t_0$ to Eqn. C.1 and use a change of variables to obtain the time-shift property:

$$\begin{aligned}\mathcal{F}\{f(t-t_0)\} &= \int_{-\infty}^{\infty} f(t-t_0)e^{-i\omega t}\,dt \\ &= \int_{-\infty}^{\infty} f(u)e^{-i\omega(u+t_0)}\,du \\ &\text{where } u = t - t_0 \\ &= e^{-i\omega t_0}\int_{-\infty}^{\infty} f(u)e^{-i\omega u}\,du \\ &= e^{-i\omega t_0}\hat{\chi}(\omega).\end{aligned} \tag{C.2}$$

C.2 Obtaining the lag between two identical 1D signals

Using Eqn. C.2 and following the steps outlined above we start with two signals $f_1(t)$ and $f_2(t)$ where $f_2(t) = f_1(t - t_0)$.

Step 1: Transform $f_1(t)$ and $f_2(t)$ into frequency space:

$$\mathcal{F}\{f_1(t)\} = \hat{\chi}_1(\omega), \tag{C.3}$$

and

$$\begin{aligned}\mathcal{F}\{f_2(t)\} &= \hat{\chi}_2(\omega) \\ &= e^{-i\omega t_0}\hat{\chi}_1(\omega).\end{aligned} \tag{C.4}$$

Where Eqn. C.4 is due to the time-shift property of the Fourier transform.

Step 2: Next take negative complex conjugate of one of the signals (say $\hat{\chi}_1(\omega)$) in Eqn. C.3 and multiply by Eqn. C.4:

$$\begin{aligned}\hat{\chi}_2(\omega)\hat{\chi}_1^*(\omega) &= e^{-i\omega t_0}\hat{\chi}_1(\omega)\hat{\chi}_1^*(\omega)\\ &\text{(time – shift property)}\\ &= e^{-i\omega t_0}|\hat{\chi}_1(\omega)|^2\\ &\text{(modulus of a complex function)}\\ &= e^{-i\omega t_0}E_1(\omega)\\ &\text{(definition of energy spectral density)}\\ &= \int_{\infty}^{\infty} e^{-i\omega t_0}|f_1(t)|^2\,dt, \qquad \text{(C.5)}\\ &\text{(definition of energy spectral density)}\end{aligned}$$

where $E_1(\omega)$ is the energy spectral density of $\hat{\chi}(\omega)$ and is equal to $E_1(t)$.

Step 3: Now applying the inverse FT operator $\mathcal{F}^{-1}$, we have:

$$\mathcal{F}^{-1}\left\{\int_{\infty}^{\infty} e^{-i\omega t_0} f_1(t)\,dt\right\} = \qquad \text{(C.6)}$$

$$\int_{-\infty}^{\infty}\int_{\infty}^{\infty} e^{-i\omega t_0}|f_1(t)|^2\,dt\; e^{i\omega t'}\,d\omega. \qquad \text{(C.7)}$$

$$\text{(C.8)}$$

Separating out the integrals:

$$\int_{-\infty}^{\infty}|f_1(t)|^2\,dt\;\int_{-\infty}^{\infty} e^{-i\omega(t'-t_0)}d\omega. \qquad \text{(C.9)}$$

$$= E_1(t)\,\delta(t-t_0), \qquad \text{(C.10)}$$

where:

$$\int_{-\infty}^{\infty} e^{-i\omega(t'-t_0)}d\omega = \delta(t-t_0). \qquad \text{(C.11)}$$

The δ-function in Eqn. C.10 means the result is zero everywhere except when $t' = t_0$ and thus the maximum (or first non-zero) result gives us the lag.

C.3 Implementation of the 1D cross-correlation

The 1D signals are loaded using the function `LoadSensorData` and I make sure they are the same size with the function `MakeSameSize`, and the 1D lag is returned using the function `Get1DLag` which acts on two input signals.

Looking at `Get1DLag` in greater detail, this function takes four inputs:

- `signal_1` - a 1D signal,
- `signal_2` - an identical 1D signal, which lags behind/ahead of `signal_1`,
- `sampling_rate=44100` - the sampling rate given in Hz
- `speed_sound=333` - the propagation speed of the signal, here given as m.s^{-2}.

The function returns three outputs in a tuple:

- The number of samples between `signal_1` and `signal_2` (`s2_point_lag`).
- The time between `signal_1` and `signal_2` (`s2_time_lag`).
- The distance between the two sensors recording `signal_1` and `signal_2` (`distance`).

In order to return `s2_point_lag`, `s2_time_lag` and `distance`, the following steps are taken:

1. The two input signals `signal_1` and `signal_2` are transformed into the frequency domain using the function `Get1DFFT`.
2. One of the signals is conjugated, here it's `s1`: `s1_conj= s1.conjugate()`
3. Next I calculate the lag by multiplying `s1_conj` and `s2`, then take the inverse FT (`np.fft.ifft`) and find the maximum:
   `s2_point_lag=np.argmax(np.abs(np.fft.ifft(s1_conj*s2)))`.
4. The lag given is either how much the `s1` is ahead *or* behind `s2`. To tell the difference I check if the first/last point of `signal_1` matches the lag in `signal_2`, which is done by checking `signal_1[0]==signal_2[s2_point_lag]` and
   `signal_1[-1:]==signal_2[s2_point_lag-1:s2_point_lag]` respectively. Once this is clarified the lag is returned.

Bibliography

Ammazzalorso, S., Gruen, D., Regis, M., Camera, S., Ando, S., Fornengo, N., Bechtol, K., Bridle, S. L., Choi, A., Eifler, T. F., Gatti, M., MacCrann, N., Omori, Y., Samuroff, S., Sheldon, E., Troxel, M. A., Zuntz, J., Carrasco Kind, M., Annis, J., Avila, S., Bertin, E., Brooks, D., Burke, D. L., Carnero Rosell, A., Carretero, J., Castander, F. J., Costanzi, M., da Costa, L. N., De Vicente, J., Desai, S., Diehl, H. T., Dietrich, J. P., Doel, P., Everett, S., Flaugher, B., Fosalba, P., García-Bellido, J., Gaztanaga, E., Gerdes, D. W., Giannantonio, T., Goldstein, D. A., Gruendl, R. A., Gutierrez, G., Hollowood, D. L., Honscheid, K., James, D. J., Jarvis, M., Jeltema, T., Kent, S., Kuropatkin, N., Lahav, O., Li, T. S., Lima, M., Maia, M. A. G., Marshall, J. L., Melchior, P., Menanteau, F., Miquel, R., Ogando, R. L. C., Palmese, A., Plazas, A. A., Romer, A. K., Roodman, A., Rykoff, E. S., Sánchez, C., Sanchez, E., Scarpine, V., Serrano, S., Sevilla-Noarbe, I., Smith, M., Soares-Santos, M., Sobreira, F., Suchyta, E., Swanson, M. E. C., Tarle, G., Thomas, D., Vikram, V. and Zhang, Y. (2020), 'Detection of cross-correlation between gravitational lensing and γ rays', *Phys. Rev. Lett.* **124**, 101102.
URL: *https://link.aps.org/doi/10.1103/PhysRevLett.124.101102*

AS, M., J-Y, D., L, B. and J-F, O. (2013), 'Optimization of automated online fabric inspection by fast fourier transform (fft) and cross-correlation.', *Textile Research Journal* **83**, 256–268.

Beutler, E. (2005), *Separation of Speech by Computational Auditory Scene Analysis*, Springer, New York, chapter 16, pp. 371–402.

Bradski, G. (2000), 'The OpenCV Library', *Dr. Dobb's Journal of Software Tools* .

Dingus, T. A., Klauer, S. G., Neale, V. L., Petersen, A., Lee, S. E., Sudweeks, J., Perez, M. A., Hankey, J., Gupta, D. R. S., Bucher, C., Jermeland, Z. R. D. J. and Knipling, R. R. (2006), The hundred-car naturalistic driving study. phase 2, results of the hundred-car field experiment, Technical report, National Highway Traffic Safety Administration, Department of Transportation, United States. Contract No. DTNH22-00-C-07007. Report identifiers: DOT-HS-810-593 and NTIS-PB2006110756.

Fan, L., Qian, J., Odry, B. L., Shen, H., M.D., D. N., Kohl, G. and Klotz, E. (2002), 'Automatic segmentation of pulmonary nodules by using dynamic 3d cross-correlation for interactive cad systems', *SPIE, International Society for Optics and Photonics* **4684**, 1362.

Girshick, R., Radosavovic, I., Gkioxari, G., Dollár, P. and He, K. (2018), 'Detectron', `https://github.com/facebookresearch/detectron`.

Hossain, S. and Hossen, M. (2020), 'Impact of unequal distances among acoustic sensors on cross-correlation based fisheries stock assessment technique', *Nature* **10**, 15911.

Menze, M. and Geiger, A. (2015), Object scene flow for autonomous vehicles, *in* '2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)', pp. 3061–3070.

Ward, D. A., Lee, I., Kearney, D. A. and Wong, S. C. (2013), Chaining convolution and correlation in practice: A case study in visual tracking, *in* '2013 International Conference on Digital Image Computing: Techniques and Applications (DICTA)', pp. 1–8.

Yang, G., Song, X., Huang, C., Deng, Z., Shi, J. and Zhou, B. (2019), Drivingstereo: A large-scale dataset for stereo matching in autonomous driving scenarios, *in* 'IEEE Conference on Computer Vision and Pattern Recognition (CVPR)'.