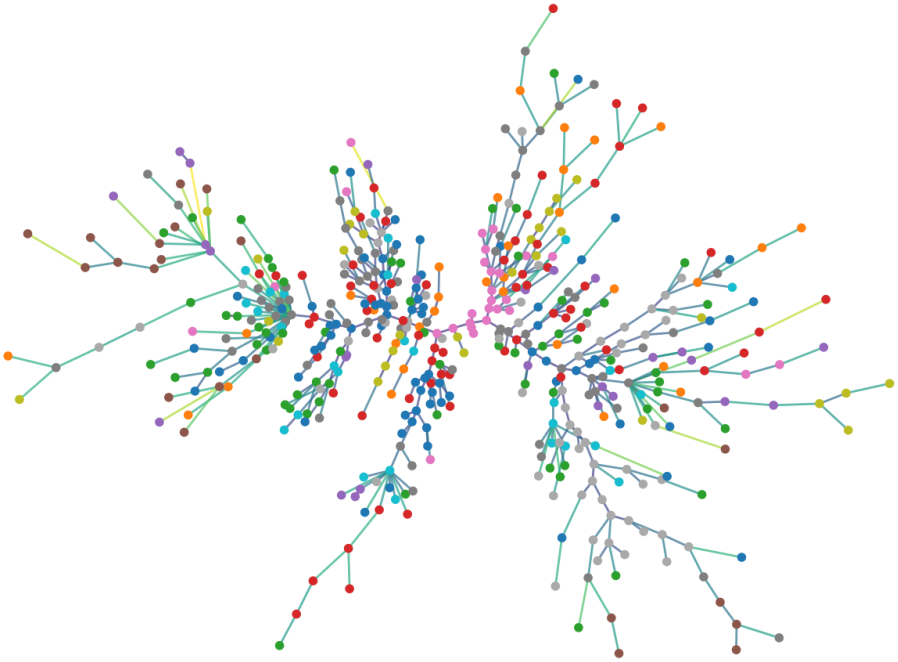
THE UNIVERSITY OF
MELBOURNE

Final Report

The Art of Scientific Computing: Applications of Cross-Correlation

Author:
John Fidel Kam

Subject Handbook Code:
COMP90072



Faculty of Science
The University of Melbourne

June 2021

Contents

Introduction

The concept of correlation pervades every scientific field. It is the measure of similarity or dependence between multivariate data, causal or not. In the following document, we introduce a type of correlation called cross-correlation, and explore its applications in the form of a programming project. The project will be implemented in Python version 3.8, where a Github repository containing all code, datasets, and documentation is found here: `https://github.com/jkfids/cross-correlation`.

The document is divided into three main sections. The first section, introduces the mathematics of cross-correlation, its basic applications, and programmatic implementation. The second section, is an investigation of stereo vision, which is the derivation of depth from multiple images. In this section, we will cover procedures including dot detection algorithms, calibration models, image comparison and 3D reconstruction. The final section explores an application in the field of finance, called a stock correlation network. In this section, we will implement algorithms to gather historical price data, calculate stock correlation matrices, and construct minimum spanning tree networks to represent stock correlation.

Potential learning outcomes include:

- A greater mastery of the Python programming language and object oriented programming in general
- Understanding data structures and compilation in relation to performance optimisation
- Developing programming style to maximise code readability and efficiency
- Learning to utilize core Python libraries (NumPy, SciPy, Pandas, etc...)
- Data visualisation and representation
- Dataset acquisition, cleaning and storage
- Version control through GitHub
- Report writing with LaTeX

1 Cross-Correlation

Cross-correlation is the measure of similarity between two series as a function of relative displacement. It is alternatively known as the sliding dot product or sliding inner product. A series may be a continuous function, a discrete vector, or an N-dimensional matrix. Cross-correlation has a multitude of applications across science, finance and technology, including pattern recognition, signal processing and portfolio analysis.

Section 1 introduces cross-correlation for 1D vectors, 2D matrices, image recognition, and cross-correlation via Fourier transforms. All code and data sets for this section are available on the GitHub repository: `https://github.com/jkfids/cross-correlation/tree/main/code`.

1.1 Cross-Correlation in 1D

We will first introduce cross-correlation for finite one-dimensional vectors. The nth element of the cross-correlation vector is given by

$$(f \star g)[n] = \sum_{m=1}^{N} \overline{f[m]} g[m-n+N], \qquad n \in [1, 2N-1] \tag{1.1}$$

where f and g are vectors of same length N, square brackets denote index, and overlines denote complex conjugates. It follows that the correlation vector has length $2N-1$. An animated visualisation of cross-correlation is linked in figure 1.1.

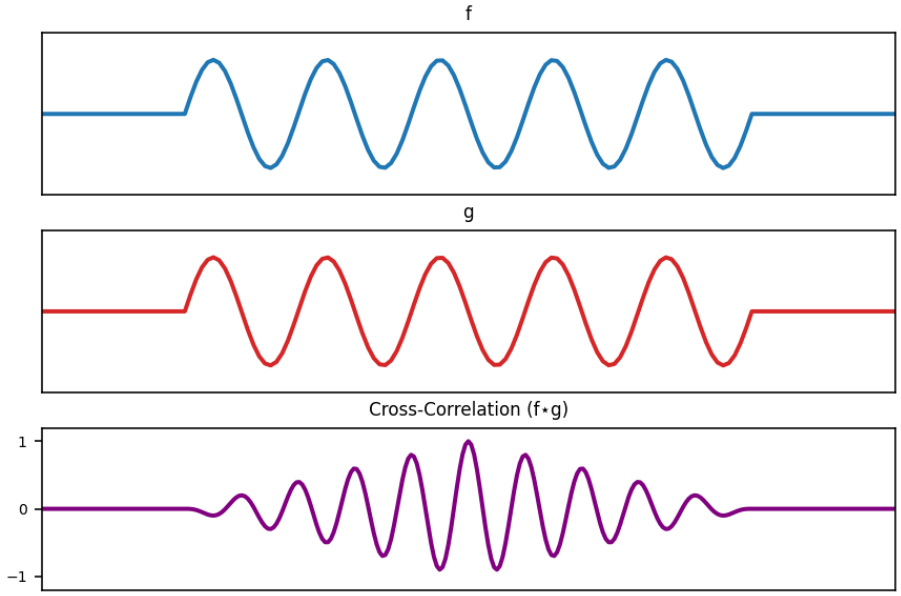


Figure 1.1: Cross-correlation of sine waves. See animated version: `https://github.com/jkfids/cross-correlation/blob/main/code/output/crosscorrelation.gif`. Vectors f and g are equivalent five-period sine waves. The cross-correlation function "slides" g across f, calculating the inner product at each step. The peaks in the resultant correlation vector correspond to when f and g are in phase, with the largest peak occurring when all five periods align i.e. zero displacement.

It is convenient to normalise the cross-correlation function so to provide a scale-free measure of similarity. The zero-normalised cross-correlation (ZNCC) function maps the elements of the cross-correlation vector to the interval $[-1, 1]$:

$$R(f \star g)[n] = \frac{1}{N} \sum_{m=1}^{N} \frac{\overline{(f[m] - \mu_f)}(g[m-n+N] - \mu_g)}{\sigma_f \sigma_g} \tag{1.2}$$

where μ_f, μ_g, σ_f and σ_g are the mean and standard deviations given by

$$\mu_v = \frac{1}{N}\sum_{i=1}^{N} v[i] \tag{1.3}$$

$$\sigma_v = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(v[i]-\mu_v)^2} \tag{1.4}$$

The unnormalised and zero-normalised cross-correlation functions are implemented in Python. These are shown in the following code snippets. For full code with documentation, view the `crosscorrelation.py` module.

```
@njit
def standev(f):
    # Standard deviation
    f = f - np.mean(f)
    return np.sqrt(np.dot(f, f)/f.size)

@njit
def crosscorr(f, g):
    # Unnormalised cross-correlation
    N = f.size
    r = np.zeros(2*N - 1, dtype=np.float64)
    r[N-1] = np.dot(f, g)
    for i in range(N-1):
        # Calculate elements of the cross-correlation vector by taking dot
        # products of (input) vector slices
        r[i] = np.dot(f[0:i+1], g[N-1-i:N])
        r[N+i] = np.dot(f[i+1:N], g[0:N-1-i])
    return r

@njit
def norm_crosscorr(f, g):
    # Zero-normalised cross-correlation
    stds = standev(f)*standev(g)
    f = f - np.mean(f)
    g = g - np.mean(g)
    return crosscorr(f, g)/(f.size*stds)
```

Notes:

- The NumPy library forms an integral component of the code. Functions such as `np.dot` and `np.mean` offer vectorised methods of processing numerical arrays. These methods are considerably faster than equivalent implementations utilizing explicit for-loops.
- Numba, a high performance Python compiler, is used to further boost execution speed. The `@njit` line calls Numba to translate a subset of Python and NumPy into fast machine code. Numba also offers parallelisation in the form CPU threading, SIMD vectorisation and GPU acceleration.
- The `crosscorr` function calculates elements of the correlation vector by taking variable length slices of the input vectors. This reduces the number of calculations in

comparison to taking full length dot products. Furthermore, it is important we predefine the correlation vector with length $2N - 1$. This is due to the faster speeds at which NumPy assigns elements in an existing array compared to appending them.

1.2 Finding Signal Offset

The first application of cross-correlation is finding the offset between two signals. Consider figure 1.2, where two sensors are placed at separate distances from a single source. Knowing the signal propagation speed and the sensor sampling rate, we can calculate the distance between the sensors with the following procedure:

1. Load the sensor signal data as numerical arrays
2. Calculate the cross-correlation vector between the signal 1 and signal 2 using `norm_crosscorr`
3. Find the index of the cross-correlation vector corresponding to the largest correlation
4. Perform calculations as necessary to extract the time offset and sensor distance

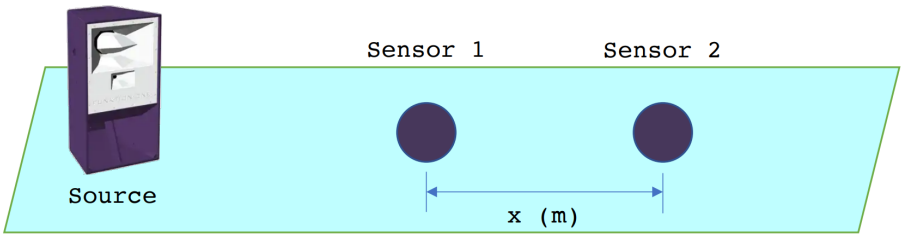


Figure 1.2: Source and sensor arrangement along x-axis.

The last step will depend on the exact correlation vector produced by the cross-correlation function. For the function defined in equation (1.1) and visualised in figure 1.1, a signal offset of zero will correspond to a maximum index located at the centre of the correlation vector. The signal time offset is then given by

$$\Delta t = \frac{l_f - i_{\max} - 1}{f_s} \tag{1.5}$$

where l_f is the length of the input vector, $i_{\max}$ is the maximum index, and f_s is the sampling rate. It follows that the distance $x = v\Delta t$, where v is the signal speed.

We apply this procedure for the sensor data shown in figure 1.3, where the cross-correlation vector is also shown. For $v = 333\text{ms}^{-1}$ and $f_s = 44\text{kHz}$, the time offset is found to be 0.4 seconds (sensor 2 lags sensor 1) and the distance to be 133.5 metres. The script used to run the experiment is found here: `https://github.com/jkfids/cross-correlation/blob/main/code/signaloffset.py`.

1.3 Cross-Correlation in 2D

We may expand cross-correlation to two dimensions. Consider template matrix t and search region matrix A. The 2D cross-correlation function translates t over A along both axis, calculating the cross-correlation at each step. Mathematically, the normalised 2D cross-correlation function is given by

$$R[i,j] = \frac{1}{w_t h_t} \sum_{y=1}^{h_t} \sum_{x=1}^{w_t} \frac{(A[x+i, y+j] - \mu_A(i,j))(t[x,y] - \mu_t)}{\sigma_A(i,j)\sigma_t} \tag{1.6}$$

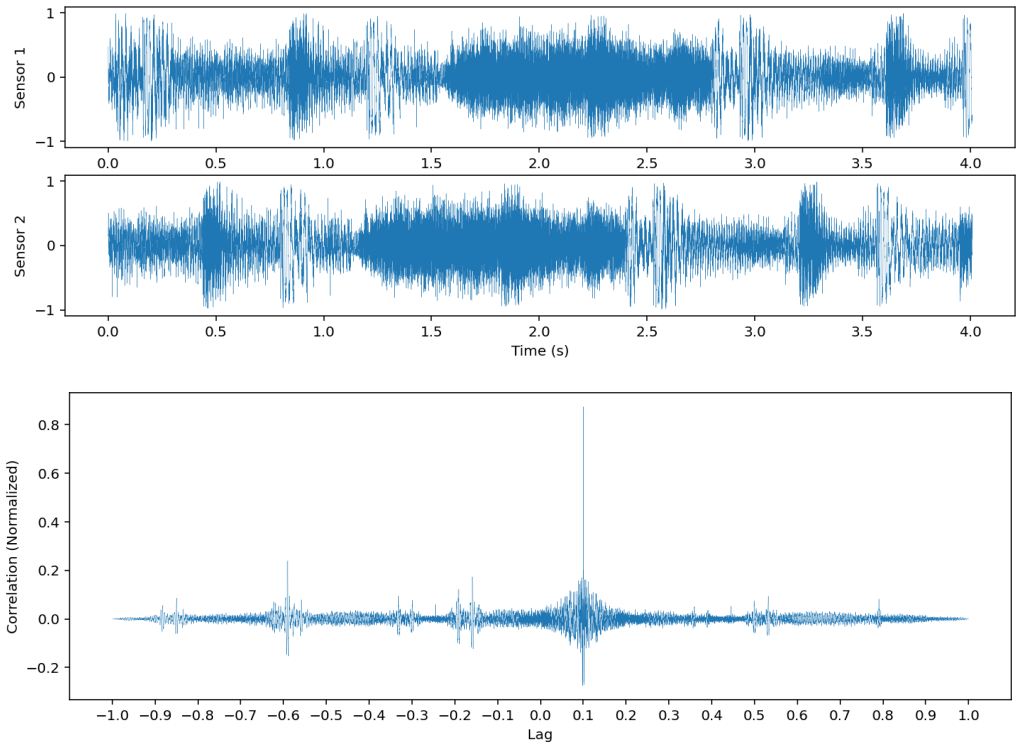


Figure 1.3: Sensor signal data (top) and the corresponding cross-correlation vector (bottom). The time offset and sensor distance calculations return 0.4s (sensor 2 lags sensor 1) and 133.5m.

where h_t and w_t are the height and width of the template matrix, and μ_t, $\mu_A(i,j)$, σ_t $\sigma_A(i,j)$ are the mean and standard deviations of t and A under region t respectively. The normalised 2D cross-correlation function is implemented as follows:

```
@njit
def norm_crosscorr2d(t, A):
    # Calculate the width and height of the cross-correlation matrix
    h_A, w_A = np.shape(A)
    h_t, w_t = np.shape(t)
    h_R = h_A - h_t + 1
    w_R = w_A - w_t + 1
    # Initialise R matrix
    R = np.zeros((h_R, w_R))
    t = t - np.mean(t)
    sigma_t = np.sqrt(np.sum(t*t))
    if sigma_t == 0:  # In case of division by 0 (sigma_t)
        return R
    # Slide t over A via nested for loops, normalising at each step
    for i in range(h_R):
        for j in range(w_R):
            A_subset = A[i:i+h_t, j:j+w_t]
            A_subset = A_subset - np.mean(A_subset)
            sigma_A = np.sqrt(np.sum(A_subset*A_subset))
            if sigma_A != 0:  # In case of division by 0 (sigma_A)
                R[i, j] = np.sum(A_subset*t)/(sigma_A*sigma_t)
    return R
```

Notes:

- To initialise the cross-correlation matrix, its dimensions are first calculated using

$$w_R = h_A - h_t + 1 \tag{1.7}$$
$$h_R = w_A - w_t + 1 \tag{1.8}$$

- The function calculates each element of the cross-correlation (R) matrix via a nested for-loop. Normalisation is performed at each step, where the $1/N = 1/w_t h_t$ terms in the standard deviation cancel with the $1/N$ term outside the sum in equation (1.6).
- In the scenario where the standard deviation evaluates to zero (e.g. searching over a dark background), the corresponding element is also set to zero. Therefore, we insert a pair of if statements to account for these cases (Noting that R is already initialised as a zero-array).
- Similar to previous functions, Numba's JIT compiler is called boost performance.

1.4 Image Recognition

2D cross-correlation serves as a convenient tool for image recognition. To demonstrate, we write a program to locate the template image shown in figure 1.4 in the search region shown in figure 1.5. The program works similarly to the procedure for finding signal offset:

1. Load the template and search region images (png files) and convert them into 2D NumPy arrays of grayscale values
2. Calculate the 2D cross-correlation matrix between the template and search region using `norm_crosscorr2d`
3. Find the index of the cross-correlation matrix corresponding to maximum correlation
4. Mark the location of the template in the search region, where the maximum index corresponds to the coordinate of the upper-left corner of the template

Figure 1.4: Rocket Man Template (111x123p)

The program utilizes Pillow, a comprehensive image processing library for Python, to load, convert, and mark the images. The grayscale matrices are calculated using a weighted average of RGB values. Code for this section is found here: `https://github.com/jkfids/cross-correlation/blob/main/code/findwally.py`.

The program was run for both Numba compiled and uncompiled versions of `norm_crosscorr2d`, where the output is shown in figure 1.6. The native Python implementation achieved a runtime of 397.5s on an Intel Core i5-8400 processor. In comparison, the Numba compiled code achieved a runtime time of 335.2s, approximately a one minute speedup.

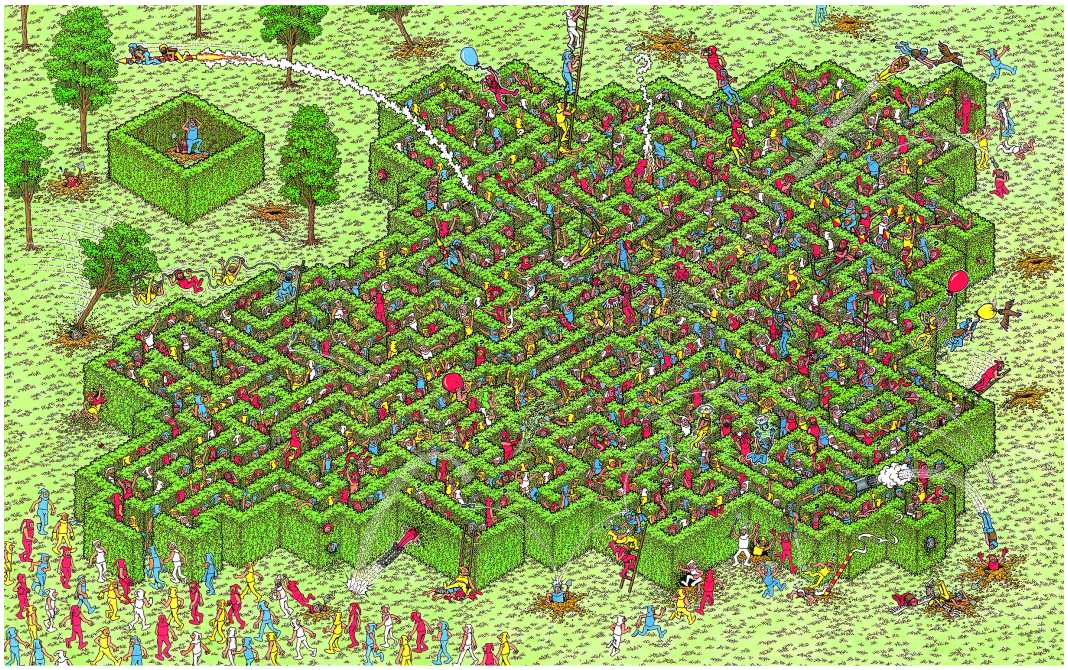


Figure 1.5: Maze search region (2446x1526p)

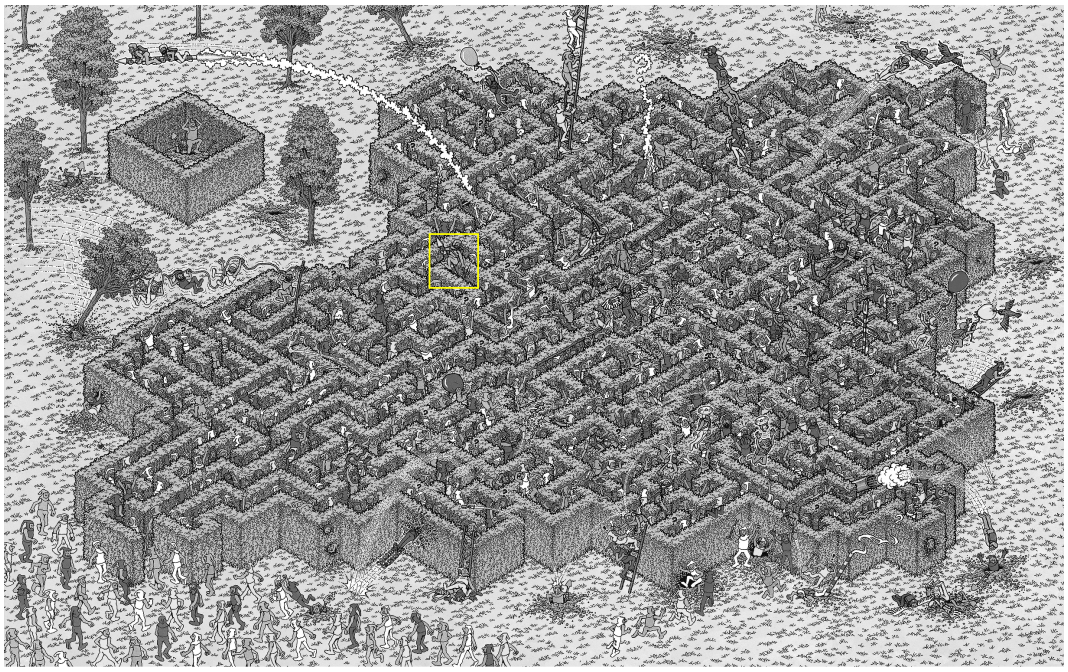


Figure 1.6: Solved wally puzzle (Runtime: 335.2s)

1.5 Cross-Correlation via FFTs

Cross-correlation can also be done in the spectral domain via Fourier transforms [5]. Analogous to the convolution theorem, the cross-correlation satisfies

$$\mathcal{F}\{f \star g\} = \overline{\mathcal{F}\{f\}} \odot \mathcal{F}\{g\} \tag{1.9}$$

where $\odot$ denotes the element-wise (Hadamard) product. Normalising, the spectral ZNCC is given by

$$(f \star g) = \frac{\mathcal{F}^{-1}\{\overline{\mathcal{F}\{f - \mu_f\}} \odot \mathcal{F}\{g - \mu_g\}\}}{N\sigma_f\sigma_g} \tag{1.10}$$

Coupled with fast Fourier transforms (ffts), equation (1.10) may be exploited for efficient numerical computation of the cross-correlation. While ffts are not covered explicitly, algorithms such as the Cooley-Turkey algorithm can compute the discrete Fourier transform in $O(N \log N)$ time. In our implementation of spectral cross-correlation, we utilize the fft package from SciPy, one of the most prevalent scientific libraries for Python.

```
from scipy.fft import fft, ifft

def spectral_crosscorr(f, g):
    stds = standev(f)*standev(g)
    f = f - mean(f)
    g = g - mean(g)
    return ifft(np.conjugate(fft(f))*fft(g)).real/(f.size*stds)
```

Notes:

- The spectral cross-correlation vector will be of different form from the spatial cross-correlation vector defined in equation (1.1). The spectral version of the cross-correlation vector for a five-period sine wave is shown in figure 1.7, where the spatial version was shown in figure 1.1.

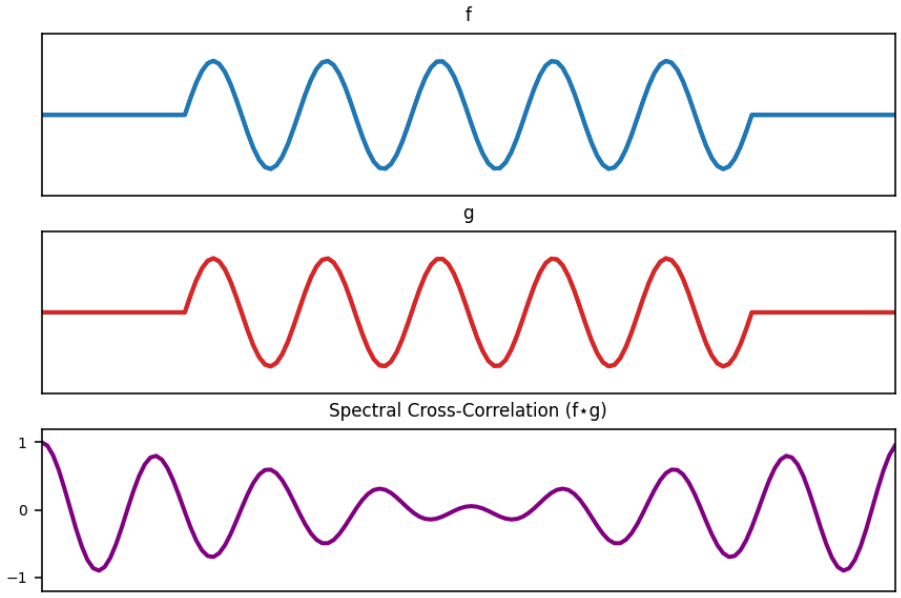


Figure 1.7: Spectral cross-correlation for five-period sine wave. The cross-correlation vector has the same length as the input vectors.

- The formula to extract the time offset for the spectral cross-correlation vector is then given by

$$\Delta t = \frac{i_{\max} - l_f}{f_s} \tag{1.11}$$

- Numba does not support JIT compilation of SciPy functions.

The signal offset problem in section 1.2 is repeated using the spectral cross-correlation function. A performance comparison is shown in table 1.

Cross-Correlation Method	Calculation Time
Spacial	4.46s ± 0.348s
Spacial (Numba)	3.03s ± 0.501s
Spacial (NumPy)	2.41s ± 0.512s
Spectral	109ms ± 4.83ms

Table 1: Performance comparison for calculation of the cross-correlation of sensor signal data. The Numba compiled cross-correlation function achieved a speed up of approximately 47% over the native Python implementation. On the other hand, NumPy's inbuilt cross-correlation function was even faster, achieving an 85% speed increase. All pale in comparison to the efficiency of spectral cross-correlation however, achieving speedups averaging 3992%.

2 Stereo Vision

Stereo vision is the extraction of depth from 2D images. By comparing images from relatively displaced vantage points, one can extract 3D information through cross-correlating similar features. Computer stereo vision is related to stereopsis in animals, which is the perception of depth derived from binocular vision. Stereo vision has a variety of modern applications including robotics, bio-engineering, virtual reality and urban planning.

Section two covers a programming implementation of stereo vision which may be broken down into five main components:

1. Feature detection on calibration plates
2. Calibration modeling
3. Image comparison
4. Optimisation strategies
5. 3D image reconstruction

2.1 Calibration

In order to produce 3D reconstructions from stereo images, it is necessary to calibrate the system to real space. This is typically accomplished by using calibration plates photographed at multiple known distances then fitting a model to map pixel coordinates to real coordinates. The calibration program for this section may be accessed here: `https://github.com/jkfids/cross-correlation/blob/main/code/calibration.py`

2.1.1 Dot Detection Algorithm

The first step in calibration is constructing a feature detection algorithm for the calibration plates. The calibration plate used for this project (figure 2.1) consists of evenly spaced bright spots. It is sufficient to model the spots as 2D Gaussian peaks and use cross-correlation to obtain the locations. The 1D and 2D Gaussian functions centred around the origin are given by

$$A \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{2.1}$$

$$A \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{2.2}$$

To construct an accurate as possible dot template, a close-up snapshot of a dot is taken from the calibration image (figure 2.1). Using SciPy's optimize package, we then fit a 1D Gaussian to a cross-section of the dot in order to obtain the function parameters. The template is then constructed by inputting these into equation (2.2) with appropriate coordinates. The Gaussian template is shown in figure 2.2.

The next step is relatively straightforward, and involves calculating the cross-correlation matrix between the template and the calibration image. Obtaining unique pixel coordinates for each dot however, requires a number of processes which are summarised below:

1. Obtain a coordinate list of all cross-correlation elements above a certain threshold. This is easily implemented with `np.argwhere(R > threshold)`
2. Select the first element from the coordinate list and initialise a coordinate summation

3. Iterate through every other coordinate in the list. If the coordinate is within a specified distance of the selected coordinate, add it to the summation then remove it from the list
4. Take the mean of the summation and append it to the list of unique pixel coordinates
5. Repeat 2 - 4 until the coordinate list is depleted
6. (Optional) Sort list of unique pixel coordinates from left to right, top to bottom
7. (Optional) Mark locations of dots in a scatter plot

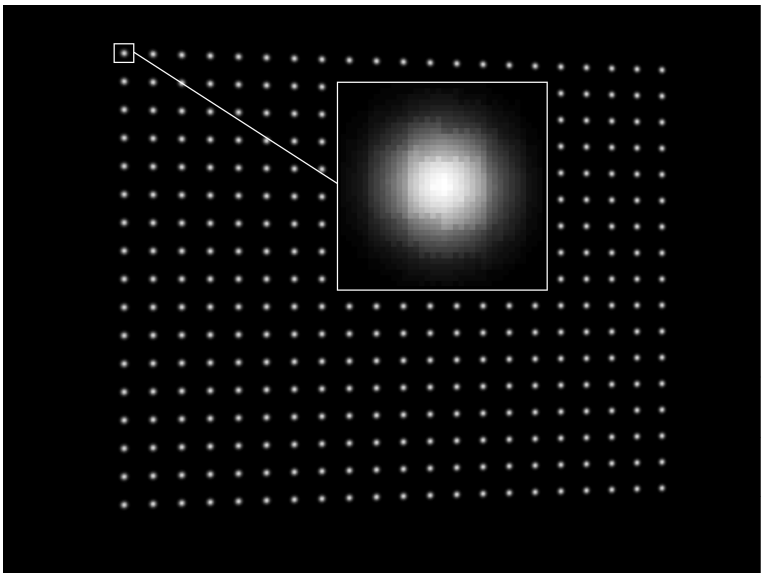


Figure 2.1: Calibration plate image (left camera). The plate is situated at a perpendicular distance of z = 2000mm.

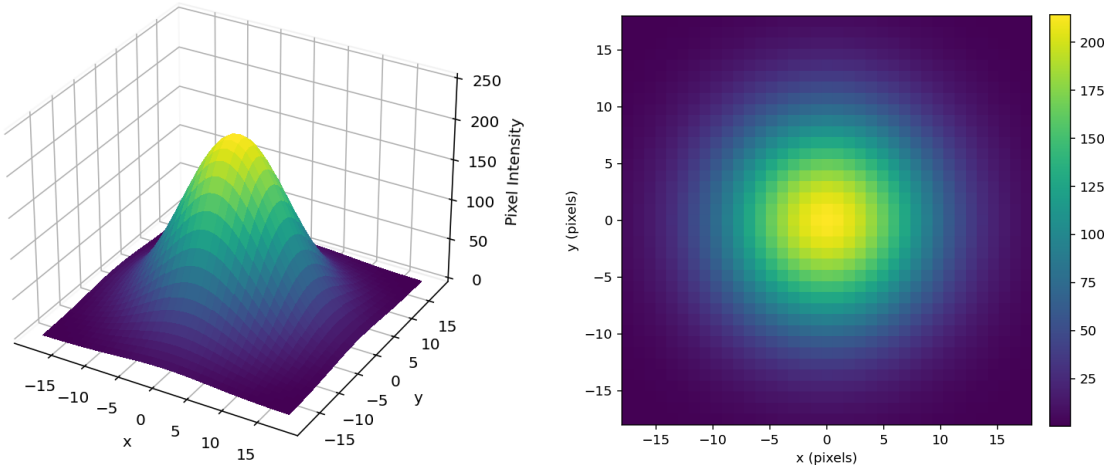


Figure 2.2: 3D surface plot (left) and heatmap (right) of Gaussian dot template.
Function parameters: $A = 214.45, \sigma = 6.934$

The previous steps are implemented in an algorithm under the `filter_coords` method. Note that cropping the template matrix (length L) optimises the computation time considerably, especially when combined with Numba compilation. Figure 2.3 shows the output obtained from applying the dot detection algorithm to the calibration image shown in figure 2.1. The script used to generate the outputs are found here: `https://github.com/jkfids/cross-correlation/blob/main/code/dotdetection.py`.

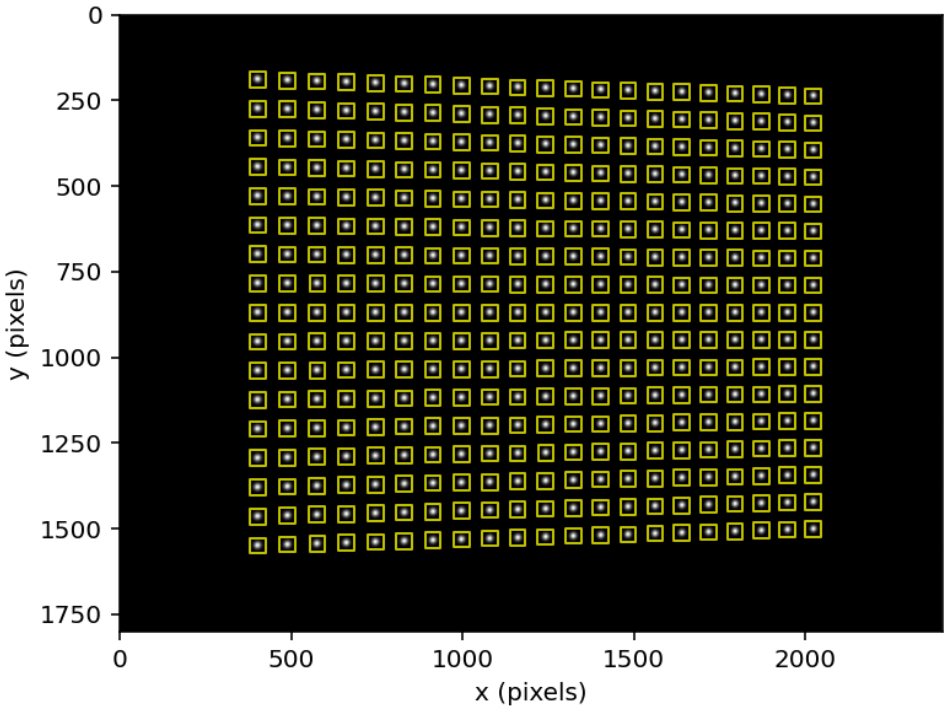


Figure 2.3: Demonstration of dot detection algorithm. Template length L = 8. Runtime (Numba): 8.23s, runtime (No Numba): 118.6s.

2.1.2 Calibration Model

The calibration model maps image space to real space for stereo images. Mathematically, we may represent the model as the following four-dimensional surface fit:

$$\begin{aligned} \boldsymbol{x} = \boldsymbol{c_0} + \boldsymbol{c_1} i_l + \boldsymbol{c_2} j_l + \boldsymbol{c_3} i_r + \boldsymbol{c_4} j_r + \boldsymbol{c_5} i_l j_l + \boldsymbol{c_6} i_l i_r + \boldsymbol{c_7} i_l j_r \\ + \boldsymbol{c_8} j_l i_r + \boldsymbol{c_9} j_l j_r + \boldsymbol{c_{10}} i_r j_r + \boldsymbol{c_{11}} i_l^2 + \boldsymbol{c_{12}} j_l^2 + \boldsymbol{c_{13}} i_r^2 + \boldsymbol{c_{14}} j_r^2 \end{aligned} \tag{2.3}$$

where $\boldsymbol{x} = (x, y, z)$ are the real space coordinates, i_l, j_l are the x, y pixel coordinates for the left image, i_r, j_r are the x, y pixel coordinates for the right image, and $\boldsymbol{c_n} = (c_{n,x}, c_{n,y}, c_{n,z})$ are constant coefficients.

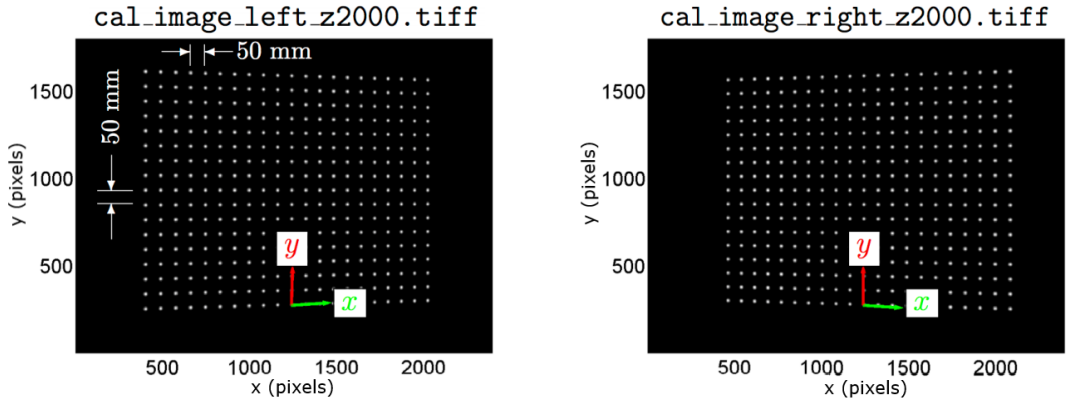


Figure 2.4: Calibration image pair. The plane of the calibration plate is situated at z = 2000mm. The dots are spaced apart in 50mm intervals, where the lower left dot has real coordinates $(x, y, z) = (-500, 0, 2000)$.

The pixel coordinates are obtained from the dot detection algorithm, and the real coordinates are predetermined when taking the calibration images. The goal of calibration is to

determine the coefficients in equation (2.3), such that pixel space can be mapped to real space for images taken with the same camera setup. There are several methods to implement this programmatically. A robust, well known method includes linear regression from the Scikit-learn library. The following code snippet demonstrates its ease of implementation.

```
from sklearn.linear_model import LinearRegression

# Call linear regression model and fit parameters
model = LinearRegression()
model.fit(variables, labels)

# Obtain coefficients
intercept = model.intercept_
coefficients = model.coef_
```

The dot detection algorithm is utilized to obtain the pixel coordinates for calibration images taken at $z = 1900, 1920, 1940, 1960, 1980$ and 2000. Then, using linear regression, we obtain the calibration coefficients for equation (2.3), shown in the following table.

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7
x	-689.969	0.616132	0.498246	-0.0608	-0.55942	-4.03E-05	-2.75E-05	6.47E-05
y	-130.137	0.505462	-0.4322	-0.50589	1.042382	2.27E-05	1.59E-05	-0.0006
z	1730.597	-1.72762	0.047452	2.164281	-0.05296	-9.04E-06	-0.00051	9.56E-06
	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	
x	6.50E-05	1.83E-05	-4.02E-05	-0.00012	-1.19E-06	0.00015	-1.71E-05	
y	0.000574	2.75E-05	7.45E-06	-7.89E-06	-1.38E-05	-7.91E-06	-1.36E-05	
z	1.04E-05	-5.00E-05	-6.43E-06	0.000165	2.51E-05	0.000171	2.48E-05	

Table 2: Calibration coefficients. Runtime (Numba): 40.4s

2.2 Image Comparison

The next component of the stereo vision framework is an image comparison program. In the previous section, comparing features was straightforward as the only features to compare were bright spots on a dark, homogeneous background. In order to apply it to real-world images however, we will need to develop more general strategies. One strategy, is to split up one image into a grid of windows, and cross-correlate each window with a search region in the other image. The image comparison program is outlined below.

1. Break up the left image into a grid of evenly sized windows. Pad along the bottom and right side of the image as necessary (figure 2.5)
2. Create a template from one window in the left image grid (figure 2.6 left)
3. Create a search region (larger than the template) in the right image centred around the same coordinates (figure 2.6 right)
4. Calculate the cross-correlation matrix between the template and the search region
5. Obtain the pixel shift in location (Δx and Δy) by finding the maximum index of the cross-correlation matrix
6. Repeat 2-5 for all windows, to obtain an array of Δx and Δy values

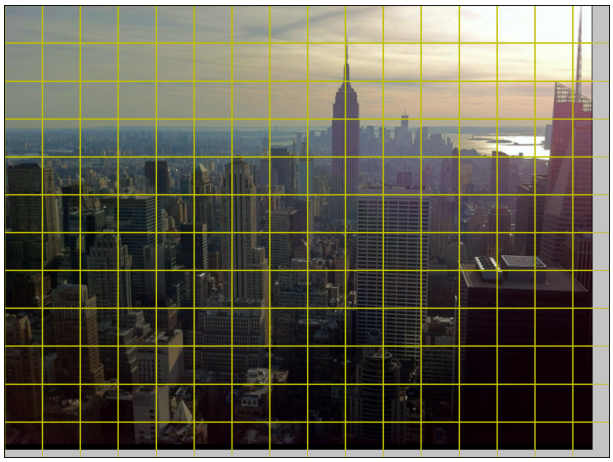


Figure 2.5: Left image is divided into a grid of evenly sized windows.

Figure 2.6: Template created in left image and search region created in right image.

The image comparison program is implemented as part of the `StereoVision` class in `stereovision.py`. There are two main methods which constitute the program:

1. `calc_shift(self, imarray1, imarray2, x, y, x0, y0)`

   The method comprises steps 2 to 5 of the outlined procedure. The inputs include the left and right image array (imarray1, imarray2), the centre pixel location of the template (x, y), and an initial guess of the pixel shift (x0, y0). The function returns the final pixel shift (dpx, dpy).

2. `calc_dparray(self, wsize, ssize, overlap, multipass_level)`

   The `calc_dparray` function executes the whole procedure by calling `calc_shift` for each window. The input parameters include `wsize`, which is the window size in pixels, and additional optimisation parameters which are elaborated on in the next section. The function outputs the array of pixel shift values (dparray) along with an array of the centre pixel locations (coord_grid).

The program was run for the stereo image pair shown in figure 2.7, where heatmaps of the pixel shift arrays are shown in figure 2.8. It is observed that for further distances, the pixel shift values increase due to the camera angles. Furthermore, there exists spurious vectors, which are artificialy high dpx/dpy values caused by errors in the cross-correlation. One method to mitigate spurious vectors is to set the dpx or dpy values beyond a specified threshold (e.g. multiple standard deviations) to the mean value. This is implemented in `filter_dparray`.

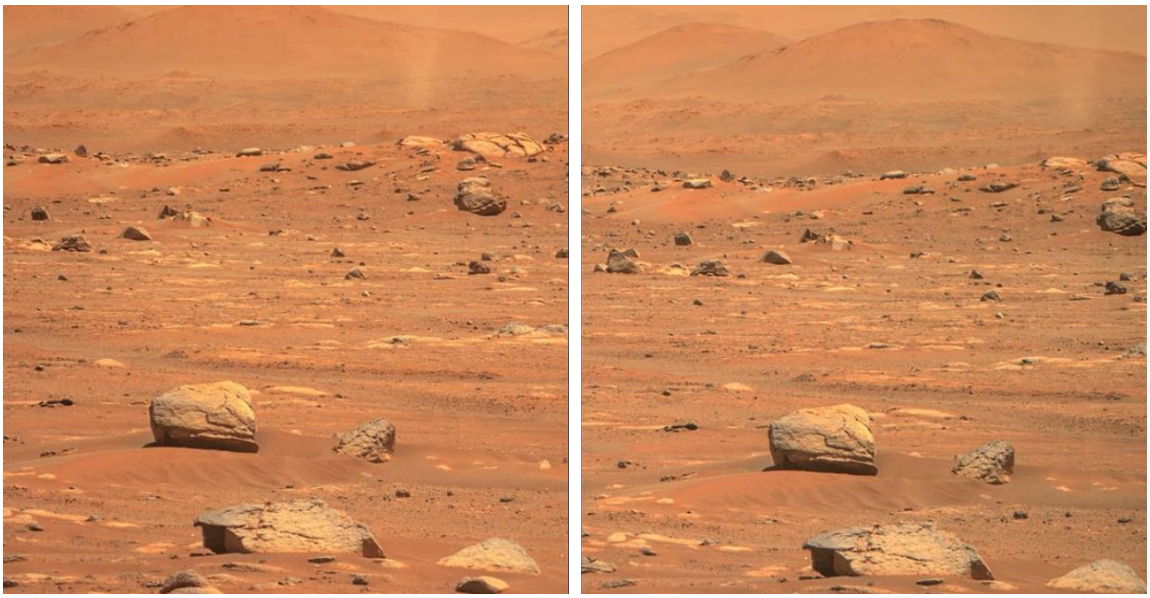


Figure 2.7: Desert stereo image pair (613x639p)

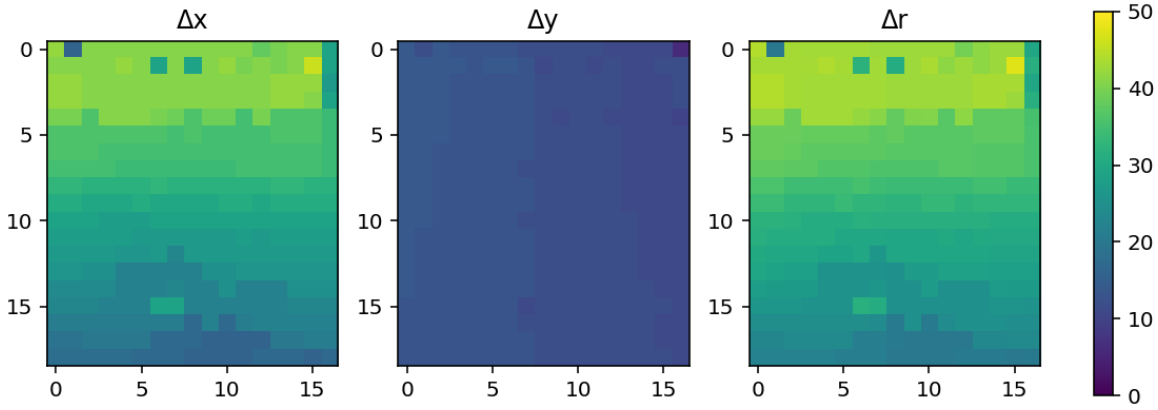


Figure 2.8: Pixel shift heatmaps of desert stereo images (Runtime: 3.80s). Δx and Δy are the pixel shift values along their corresponding axis, and $\Delta r = \sqrt{\Delta x^2 + \Delta y^2}$ is the total pixel shift magnitude.

2.3 Optimisation Strategies

We will now explore three optimisation strategies to improve the accuracy and the computational speed of the image comparison system. These are variable search geometry, variable window overlap, and multi-pass cross-correlation.

1. **Variable Search Geometry**

   The first and most straightforward optimisation technique is to vary the search region geometry. Setting too large of a search region will increase the computation time and the rate of spurious vectors. On the other hand, too small of a search region may cause the program to fail in finding correlated regions in the first place.

   Additionally, the search region need not be a square. A rectangular search region may be more optimal if the average shift in pixel location is skewed towards a certain axis.

   As part of the `calc_shift` function, search region geometry may be varied by adjusting the `ssize=(w,h)` input parameter. Where w and h are the width and height of the search region (as multiples of `wsize`).

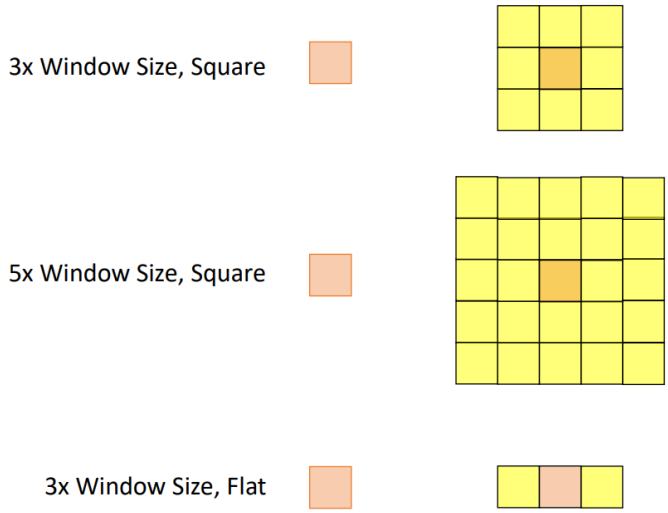


Figure 2.9: Variable search regions

2. **Variable Window Overlap**

   In figure 2.5, it is assumed that the image is divided into a grid with zero overlap. This has the downside where features in between windows may not be properly accounted for. To compensate, we allow for a specified percentage of overlap between windows (see figure 2.10).

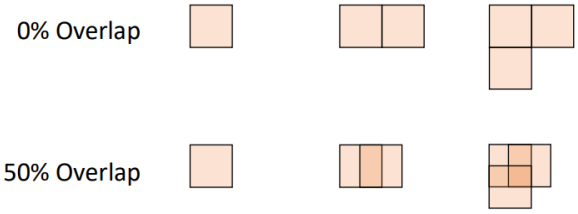


Figure 2.10: Variable window overlap

The percentage overlap may be adjusted by varying the `overlap` input parameter between zero and (almost) one. A comparison of Δr heatmaps for the desert images is shown in figure 2.11. Evidently, increasing the overlap improves the resolution and detail of the heatmap (at the expense of longer runtimes).

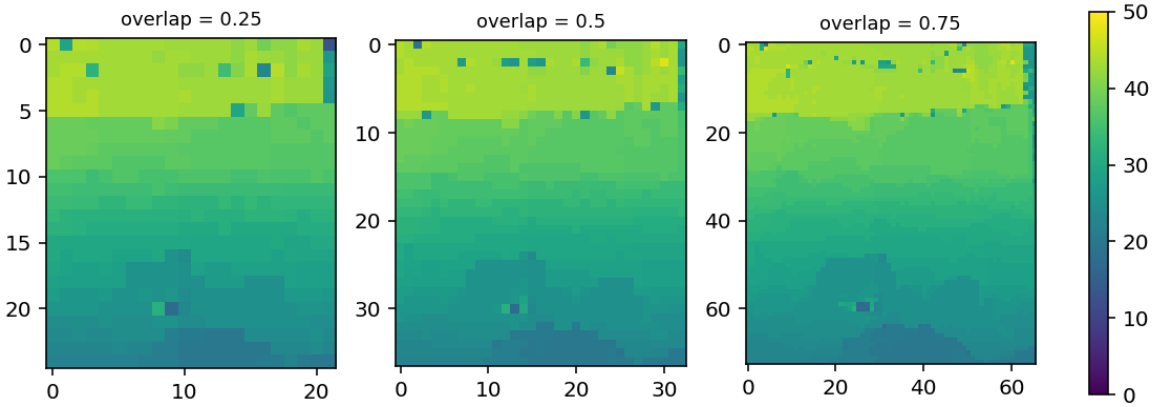


Figure 2.11: Δr heatmap comparison for different overlap percentages. Runtimes (left to right): 5.076s, 11.292s, 44.017s.

3. **Multi-Pass Cross-Correlation**

The final optimisation strategy involves doing a course-to-fine multi-pass cross-correlation. Consider the 2-pass correlation shown in figure 2.12. For the initial pass, the image comparison program is executed for a window length of 64 pixels. An array of dpx and dpy values associated with each window are obtained as usual. In the second pass, the window length is halved, and the previous dpx and dpy values are used as an initial guess when searching for the correlated regions. In other words, the first pass obtains a broad guess of where the object has moved, and the second pass uses this information to provide finer detail. The process may be generalised for more than two passes.

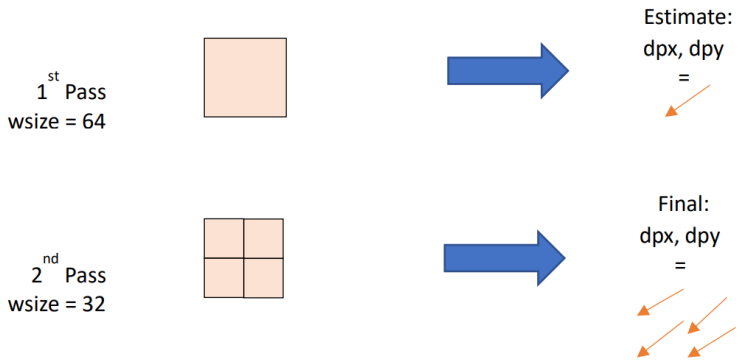


Figure 2.12: Caption

In `calc_dparray`, the number of passes may be adjusted with the `multipass` input parameter. Each subsequent pass halves the window length, where the search region shrinks as a fixed multiple of `wsize`. Multi-pass cross-correlation is demonstrated in figure 2.13. It is shown that in some scenarios, higher level passes not only improve the optimisation time, but also reduce the rate of spurious vectors.

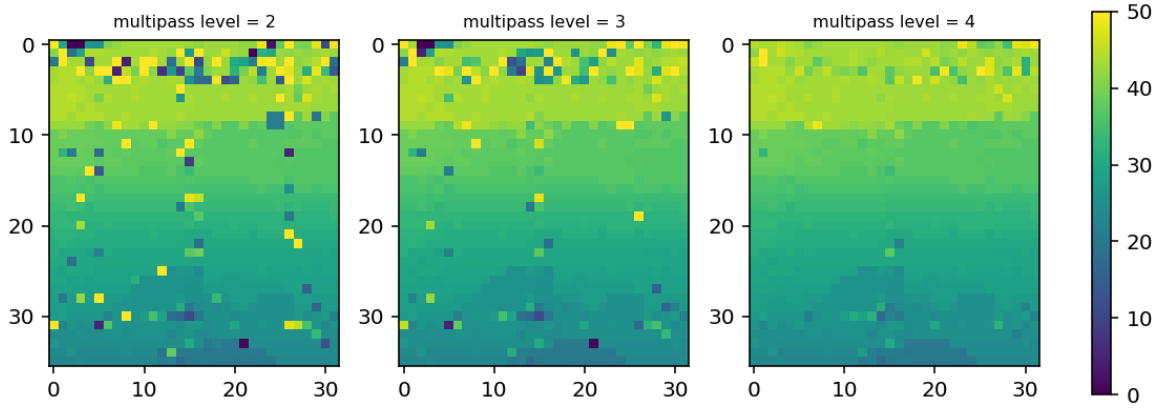


Figure 2.13: Multi-pass cross-correlation. The program was set up so that the initial search region sizes (in pixels) and the final window sizes are the same. Runtimes (left to right): 26.488s, 15.863s, 7.49s.

2.4 Test Scans

2.4.1 Additional Image Comparisons

Shown below are additional image comparisons for pairs of stereo images. The script used to run the comparisons is accessible here: `https://github.com/jkfids/cross-correlation/blob/main/code/imagecomp.py`.

Figure 2.14: Portal stereo image pair (1920x1080p)

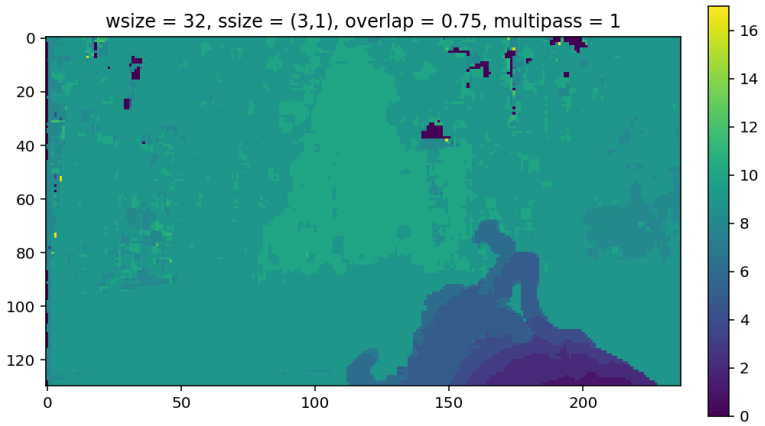


Figure 2.15: Δr heatmap for portal stereo images.

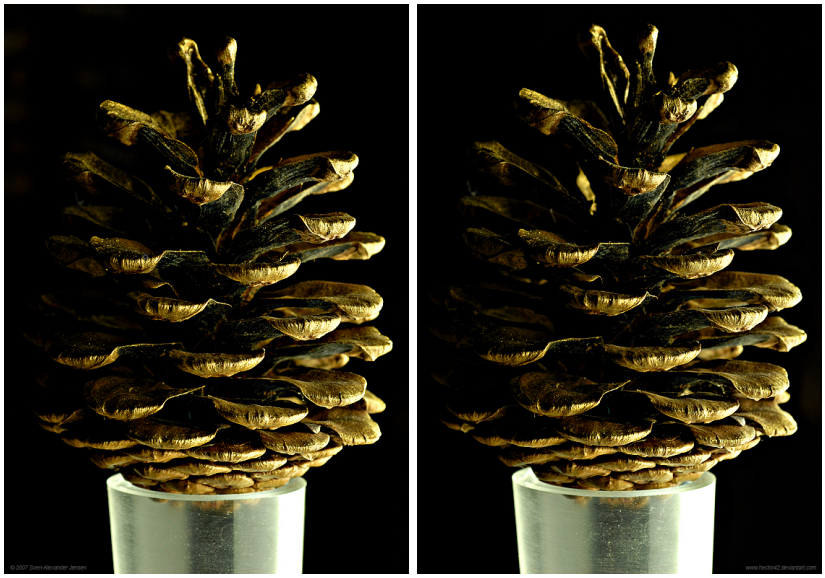


Figure 2.16: Conifer cone stereo image pair (560x790p)

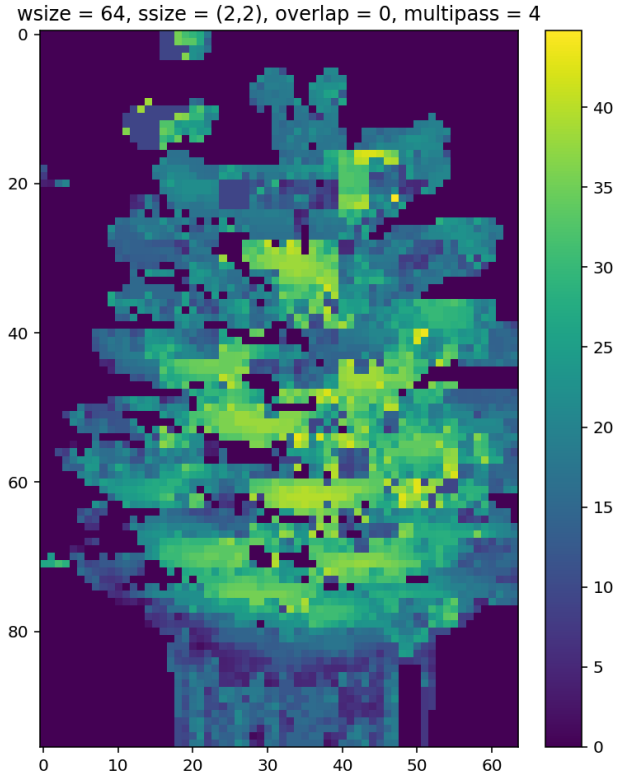


Figure 2.17: Δr heatmap for cone stereo images.

2.4.2 3D Image Reconstruction

Using the image comparison program, it is possible to derive 3D reconstructions for calibrated images. The left image coordinates are comprised of the centre locations of the window templates. To obtain the right image coordinates, we need only add the dpx and dpy values to the corresponding left image coordinates. Calculating the real space coordinates is then as simple as calculating the RHS of equation (2.3), where the $\boldsymbol{c_n}$ coefficients are obtained from calibration. 3D reconstructions of several test image pairs (using the calibration in table 2) are shown below. Code for this section is found here: `https://github.com/jkfids/cross-correlation/blob/main/code/testscan.py`.

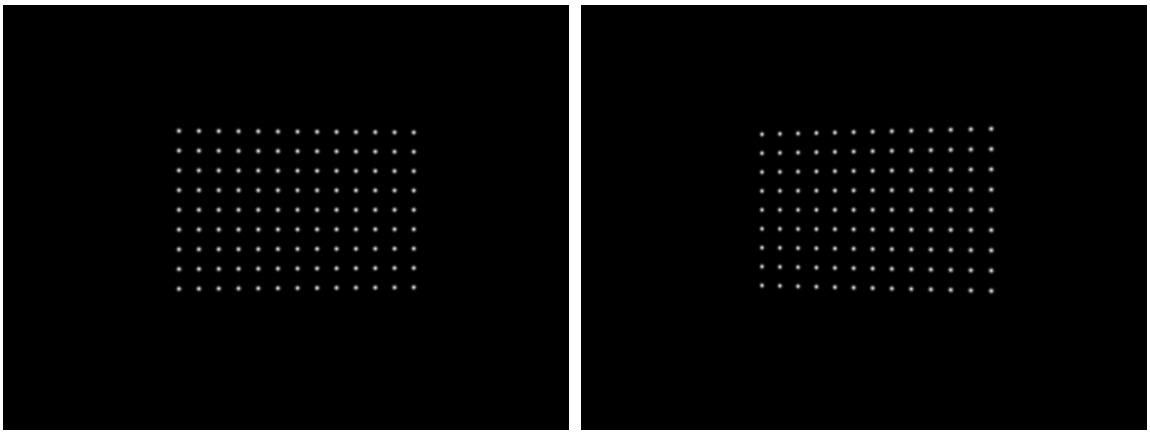


Figure 2.18: Test pair 1 (Calibration dots). Used to test calibration.

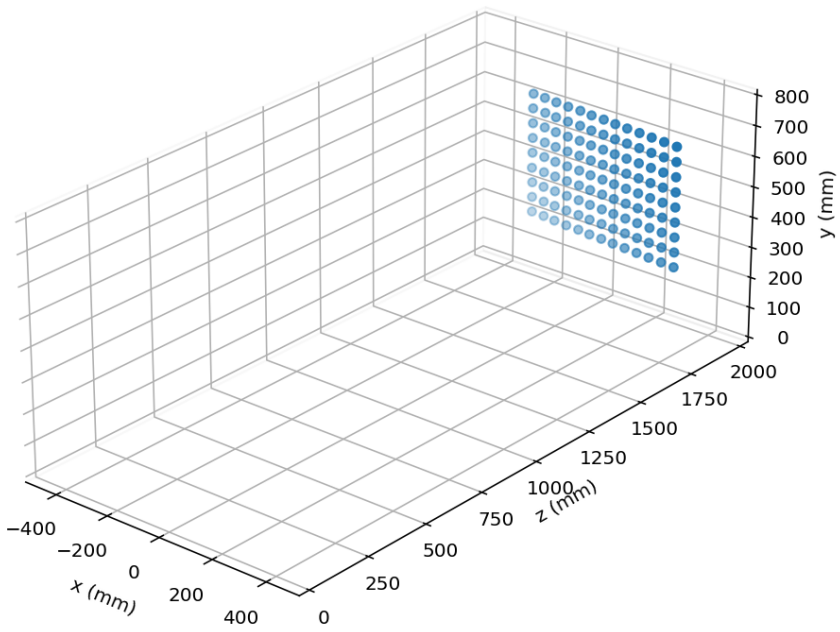


Figure 2.19: 3D reconstruction of first test pair (Runtime: 7.121s). The dot detection algorithm is used instead of the image comparison program to obtain the pixel coordinates.

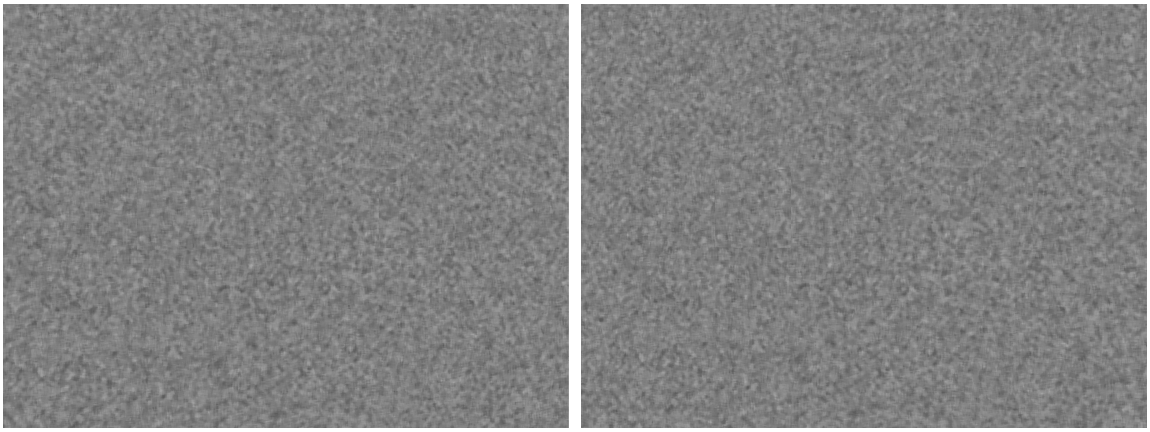


Figure 2.20: Test Pair 3 (2401x1801p)

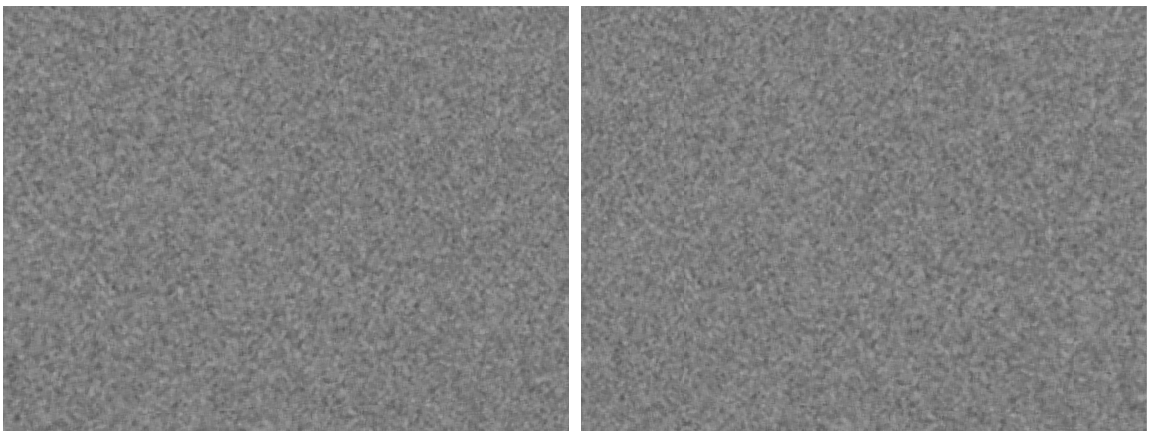


Figure 2.21: Test Pair 2 (2401x1801p). For images such as these, it is difficult to distinguish depth with the human eye. Cross-correlation on the other hand may extract 3D structures not visible at first glance.

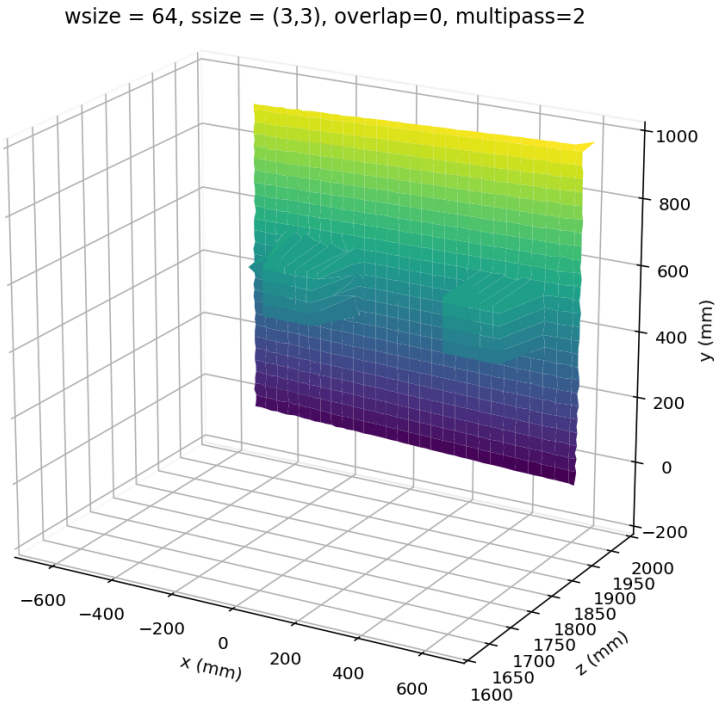


Figure 2.22: 3D reconstruction of second test pair (Runtime: 688.091s). Spurious vectors are removed by setting threshold values along the z-axis, where values beyond the threshold are set to z = 2000mm. The 3D reconstruction reveals two protrusions from the x-y plane.

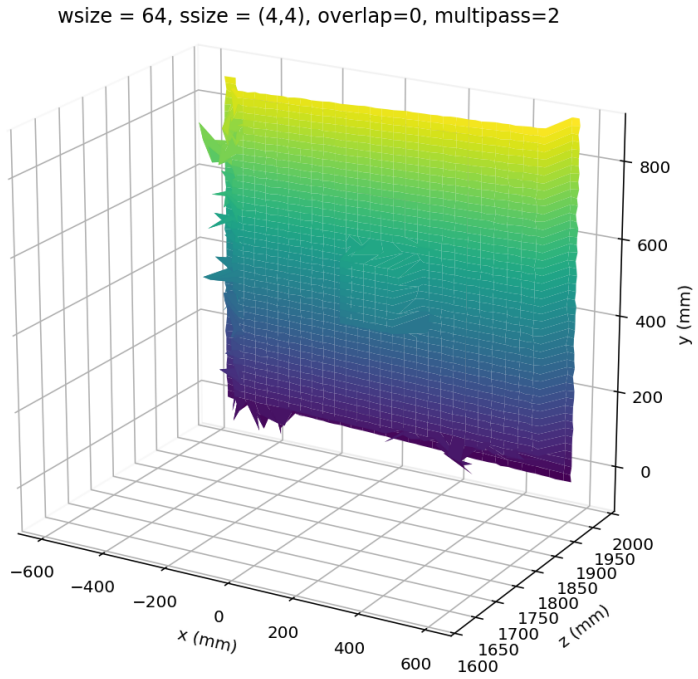


Figure 2.23: 3D reconstruction of third test pair (Runtime: 1556.588s). Values beyond the threshold are set to z = 1960mm. The 3D reconstruction shows a protrusion in the centre of the image concentric to a wider elevation.

A number of optimisation parameters were varied to obtain the best results. It seems that for the simple 3D structures revealed in test pairs 2 and 3, increasing overlap does not substantially increase the quality of the image (but does significantly increase computation time). Multi-pass correlation can be used to speed up the program, though it should be considered that spurious vectors appearing in earlier passes are propagated in subsequent passes.

2.5 Limitations

It is worthwhile to note a few limitations of the stereo vision program.

- Computation times to calculate the dparray can be exceedingly long for high resolution images such as in figure 2.23. This can be mitigated in the current system by reducing the resolution of the images, or increasing the window size and number of correlation passes. On the other hand, the ideal strategy may be to adapt the spectral cross-correlation function to two dimensions in order to find the pixel shifts. Libraries such as SciPy contain packages that can compute 2D ffts.
- Cross-correlation can only correlate images if they are nearly identical. It is unable to correlate objects under rotation or scale change. In order to account for these, a more sophisticated stereo vision system, such as neural networks, may need to be implemented.
- Only the bare minimum amount of 3D information can be extracted from a pair of images. Constructing full 3D models from 2D images, requires many more photographs from various of angles. Some 3D imaging systems utilize a single camera which is rotated around the object, which is beyond the scope of this project.

3 Stock Correlation Network

A stock correlation network is a type of dynamic network based on the cross-correlation of historical stock data. It is also known as a dynamic asset tree, and in recent years, has been increasingly used to study and predict financial markets. Stock correlation networks have been shown to be a strong indicator of market turbulence, as shown for the 1987 Black Monday crash by Chakrabortia and Onella [4]. Furthermore, some networks have been shown to exhibit scale free behaviour, where certain system parameters may be modeled after power law distributions [3].

The basic procedure to construct a stock correlation network is outlined as follows:

1. Obtain time series data for the desired stocks. The historical data may be anything from daily closing prices, opening prices, or price returns.
2. Calculate the cross-correlation matrix of all the stocks, where their time series data is free of time delays.
3. Construct the stock network as a minimum spanning tree using the cross-correlation matrix.

There exist alternative network configurations that may be used in place of the minimum spanning tree in step 3. These include the planar maximally filtered graph, or the winner-take-all model.

In this final section of the report, we will implement the outlined steps to construct a stock correlation network for the Standard and Poor's 500 (or S&P500) stock market index.

3.1 Gathering Historical Data

The first step in constructing the correlation network is to obtain the list of companies in the S&P500 index. The pandas library, one of the most popular data management frameworks in Python, offers a method to obtain S&P500 company information by scraping HTML data off Wikipedia. This is executed in a few of lines of code:

```
import pandas as pd

table = pd.read_html(
    "https://en.wikipedia.org/wiki/List_of_S\%26P_500_companies")
```

Moving on, we will need to obtain the historical price data of stocks. There are multiple ways to accomplish this, but a simple method would be to employ the `Ticker` module from the yfinance library. For example, if we wanted to collect the stock history of Google (GOOGL) over the past year, we would write

```
from yfinance import Ticker

tick = Ticker('GOOGL')
history = tick.history(period='1y')
```

A script is written to obtain the time series data of S&P500 stocks over the past 10 years. The data is then stored as csv files. Link: `https://github.com/jkfids/cross-correlation/blob/main/code/genstockdata.py`. For the purposes of this project, we will only only consider the daily closing prices.

3.2 Correlation Matrix

The next step involves calculating the correlation matrix for every stock. It is important to note that the correlation matrix does not refer to the 2D cross-correlation matrix defined in equation (1.6). Rather, it is the cross-correlation of historical prices between every combination of stocks, free from time delays. Mathematically, the normalised correlation matrix is given by:

$$C_{ij} = \frac{1}{N} \frac{(\boldsymbol{f} - \mu_f) \cdot (\boldsymbol{g} - \mu_g)}{\sigma_f \sigma_g} \tag{3.1}$$

where i, j denotes index, f and g are vectors of length N, and μ, σ are the mean and standard deviation.

In the `stockcorrelation.py` module, an algorithm to calculate the correlation matrix is implemented under the `calc_corrmatrix` method. In the case vectors v_f and v_g are not the same length, the larger vector is cropped to match the size of the smaller vector. The program is run for all S&P500 stocks using the last 10 years of historical data. The resultant matrix is visualised in figure 3.1.

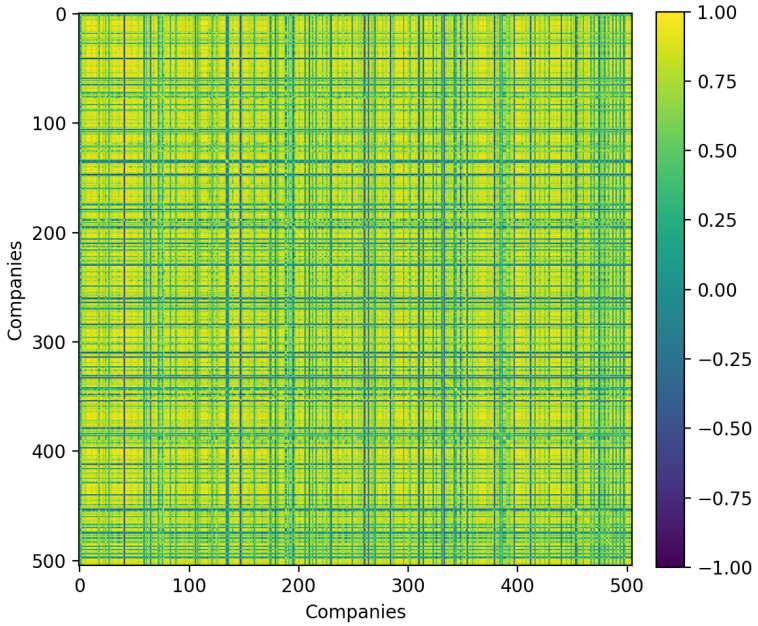


Figure 3.1: Heatmap of S&P500 stock correlation matrix (10 years). The companies are ordered in alphabetical order. For the most part, stocks are highly correlated, although some anti-correlated stocks exist.

As a bonus, we also calculate the correlation matrix between category averages (figure 3.2).

3.3 Minimum Spanning Tree

It is now possible to construct the minimum spanning tree of stocks from the correlation matrix. The minimum spanning tree is a type of weighted, undirected graph that connects all nodes together, without any cycles and with the minimum total edge weight. For the stock network, nodes are companies, and edge weights are given by the formula

$$W_{ij} = \sqrt{2(1 - C_{ij})} \tag{3.2}$$

We utilize NetworkX to construct the graph network. NetworkX is a comprehensive Python library for the creation, manipulation and visualisation of complex networks. An algorithm

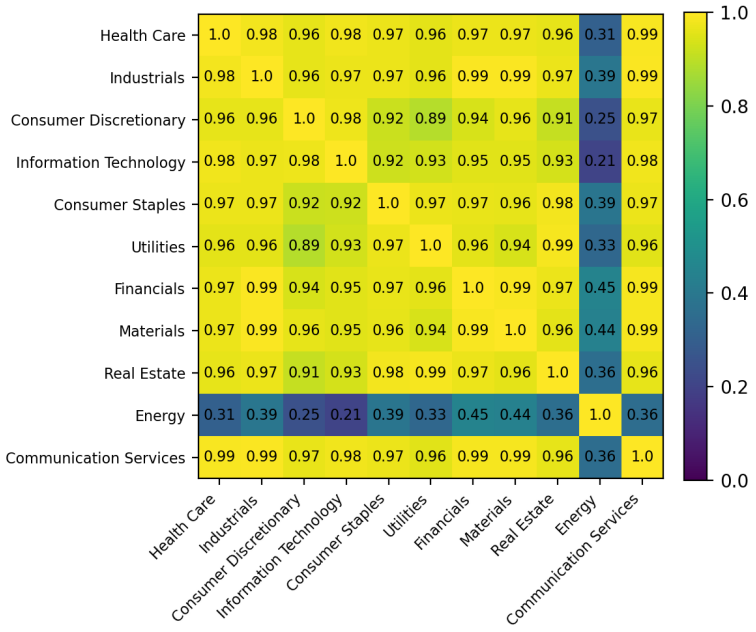


Figure 3.2: Heatmap of S&P500 categorical correlation matrix (10 years). Interestingly, all sectors are highly correlated with the notable exception of the energy sector. The energy sector is often cited to be the worst performing sector in the last decade.

to generate the tree can be divided into two main components:

1. Iterate over every unique stock combination, constructing a graph where each stock is a node, and edges are given by equation (3.2)
2. Construct the minimum spanning tree from the graph in step 1, using any number of algorithms

The algorithm is implemented under the `gen_stocknetwork` method in `stockcorrelation.py`. The first step is accomplished via a nested for-loop, and the second step is easily implemented by NetworkX. NetworkX generates the minimum spanning tree using Kruskal's algorithm. A flow chart of the algorithm is shown in figure 3.3.

The program is run for the correlation matrix shown in figure 3.1, where the generated network is shown in figure 3.4. We also introduce a metric called the normalised tree length, which is simply the mean of all edge weights. Finally, the process is repeated using historical data over the past year, shown in figure 3.5.

It appears nodes in the ten year network are more uniformly distributed than in the one year network. In the one year network, nodes are concentrated around a central region, which is extended by branches of highly anti-correlated stocks. The larger normalised tree length for the one year network indicates a more divergent market. This may be explained by the disproportionate effect of the COVID-19 pandemic on different sectors.

In Onnela et al. (2003), it is shown that the convergence of normalised tree length as a function of window time could have predicted the Black Monday stock market crash [4]. During a market crash, the minimum spanning tree experiences a rapid reconfiguration. Furthermore, Bonanno et al. (2003) showed that the network dynamics of real markets could not be reproduced by the random market model and by the one-factor model [1].

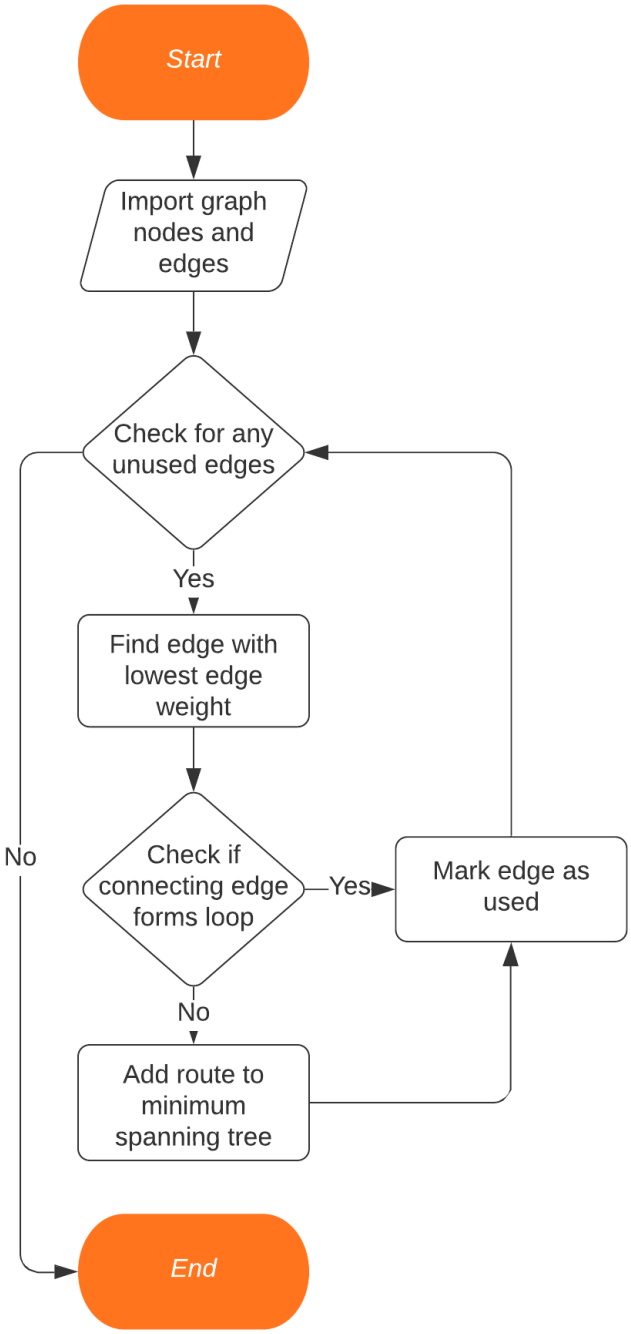


Figure 3.3: Flow chart for Kruskal's algorithm

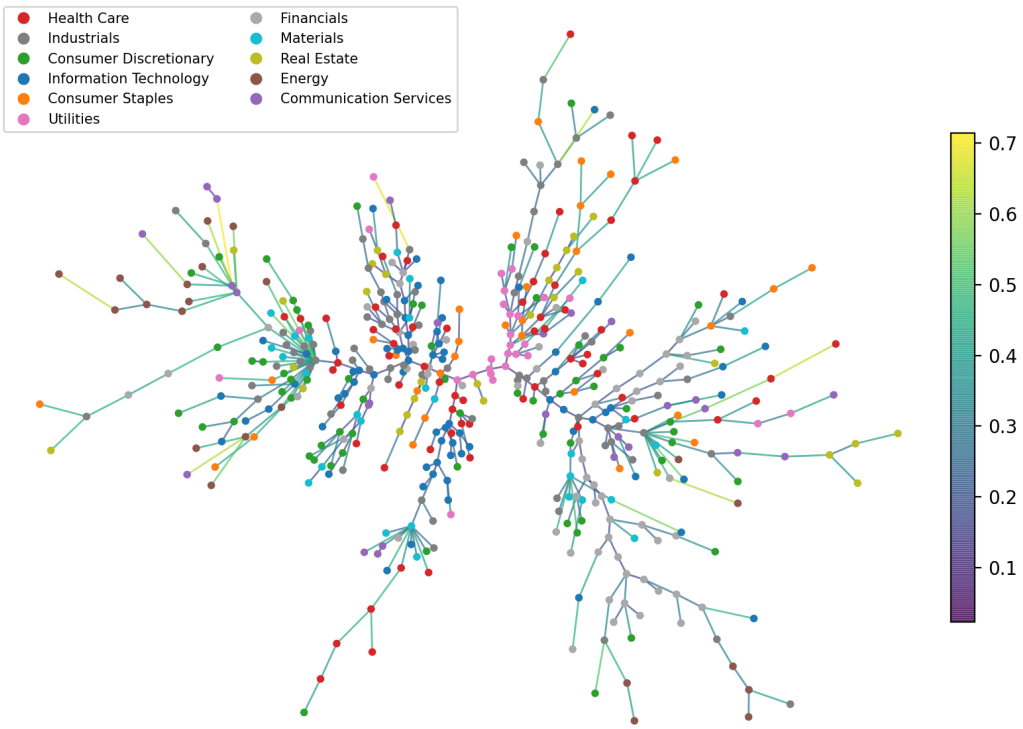


Figure 3.4: S&P500 stock correlation network (10 years). The color bar maps the edge weights. Normalised Tree Length: 0.2674

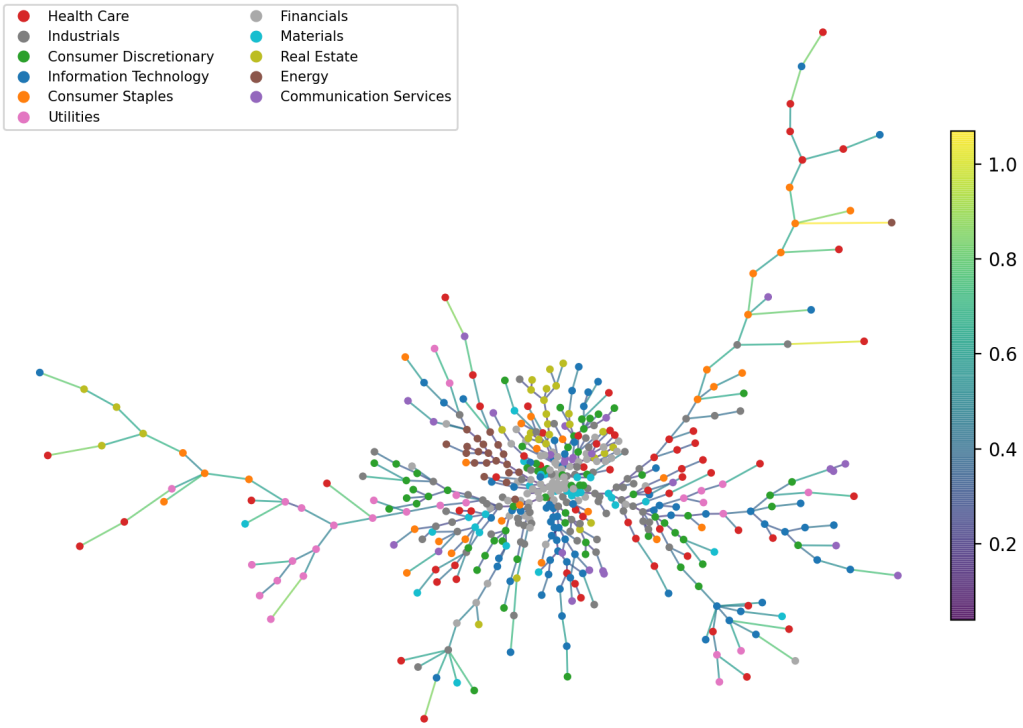


Figure 3.5: S&P500 stock correlation network (1 year). Normalised Tree Length: 0.3303

Conclusion

The project has been an in-depth exploration of cross-correlation and two of its major applications. In section one, we introduced cross-correlation and its computational implementation using popular Python libraries such as NumPy. We explored its applications for 1D signals and 2D images, and developed an efficient computational method to calculate cross-correlation via ffts.

In section two, we investigated how cross-correlation can be utilized to extract 3D information from 2D images. We successfully constructed a calibration model using a dot detection algorithm, and an image comparison algorithm for stereo image pairs. By combining these elements, we were then able to derive 3D reconstructions from digital images. We also discussed the limitations of our program, and suggested alternative strategies including neural networks.

In the section three, we explored the stock correlation network. We developed a program that could download and store historical price data, and constructed a correlation matrix for the S&P500 stock market index. Using the correlation matrix, we were able to construct a stock correlation network in the form of a minimum spanning tree, and discussed its significance and utility as a tool to study financial markets.

In conclusion, the project has been a fruitful endeavour in developing the personal artistry that is scientific computation.

References

[1] Giovanni Bonanno et al. "Topology of correlation-based minimal spanning trees in real and model markets". In: *Physical Review E* 68.4 (Oct. 2003). ISSN: 1095-3787. DOI: 10.1103/physreve.68.046130. URL: http://dx.doi.org/10.1103/PhysRevE.68.046130.

[2] Paul Bourke. "Cross correlation". In: *Cross Correlation", Auto Correlation—2D Pattern Identification* (1996).

[3] K Tse Chi, Jing Liu, and Francis CM Lau. "A network perspective of the stock market". In: *Journal of Empirical Finance* 17.4 (2010), pp. 659–667.

[4] J.-P. Onnela et al. "Dynamic asset trees and Black Monday". In: *Physica A: Statistical Mechanics and its Applications* 324.1-2 (June 2003), pp. 247–252. ISSN: 0378-4371. DOI: 10.1016/s0378-4371(02)01882-4. URL: http://dx.doi.org/10.1016/S0378-4371(02)01882-4.

[5] Eric Weisstein. *Cross-Correlation.* URL: https://mathworld.wolfram.com/Cross-Correlation.html.