THE UNIVERSITY *of* EDINBURGH

MEng Civil Engineering

Thesis

Towards the conversion of CoastSnap to an open source citizen science tool for coastal monitoring.

Nicholas Heaney

Supervised by:

Dr Encarni Medina-Lopez

29th April 2021

Word count: 15 998

Abstract

The coastline is experiencing unprecedented wave and sea level conditions due to climate change. The shoreline is naturally adapting to these new conditions through morphological changes and for nearby communities to respond appropriately it is essential to monitor these. Coastal monitoring techniques have typically been expensive, however with increasing demand, cheaper, more efficient approaches are being developed. One of these is CoastSnap, a beach monitoring tool for measuring shoreline change. CoastSnap uses citizen science to take advantage of the large source of high quality images provided by the public. The software used to extract shorelines from these crowd sourced images runs in MATLAB, a proprietary platform. MATLAB licensing is a financial barrier for many beach locations looking to adopt CoastSnap. However, this problem can be overcome by converting CoastSnap to the alternative programming language, Python. The conversion process is initiated here by the identification of the critical functions required for CoastSnap to be operated. Equivalent functions in Python are proposed with an analysis of their performance in comparison to the original MATLAB functions. Generally, very well performing equivalent functions have been found. Some Python functions were not able to return exactly the same results as the MATLAB equivalents particularly when the organisation of parameters inside the MATLAB function could not be determined.

Statement of Covid-19 impact

The uncertainty created by the Covid-19 pandemic has resulted in high levels of anxiety during certain periods during which this thesis was being written.

It has been my intention to return to my home country, Australia on completion of my studies, however, due to border restrictions it has been very difficult to plan this.

As a result, for several weeks during February I was under high levels of anxiety and I was distracted from work.

Acknowledgements

I would like to thank Dr Encarni Medina-Lopez for her supervision throughout this work. Without her guidance I would not have known where to begin with this task.

Thank you to the CEReS research group. Their ideas and suggestions have been very helpful.

Declaration of own work

All of the material provided here is original unless explicitly stated.
This is with the exception of the literature review section. Much of the material here was presented as part of the of the submission for the Research Methods 5 course.

Contents

Figures

Introduction

Coastal monitoring first became prominent during the 1970s as a relatively small community of coastal scientists began to formulate the morphodynamic principles governing our shorelines (Short and Jackson, 2013). These monitoring techniques were labour intensive and even after technological advances such as coastal imaging reduced the level of human resource required (Holman and Stanley, 2007), the cost of implementing these systems has typically kept the overall costs high.

Today, interest in coastal monitoring has expanded from specialised research groups to coastal communities all over the world. This is due to climate change altering the conditions encountered by coastal regions. The coastal environment is experiencing unprecedented impacts due to more severe wave conditions and higher sea levels (Masselink, Scott, *et al.*, 2016). Consistent monitoring is key to enable research and understanding of the behaviour of the coast under these new conditions. With this, coastal regions can be better managed, and appropriate strategies implemented to protect populations vulnerable to natural coastal processes (Cooper and McKenna, 2008).

As interest in coastal monitoring has increased, so too has the demand for more affordable monitoring systems (Ton *et al.*, 2020). This is to enable poorer regions to manage their coastlines as well as to enable expansive regions of coastline to be economically monitored, not just individual beach sites.

One emerging approach, citizen science, takes advantage of a synergy between the public and researchers. This is achieved by the ability of the public to collect large

volumes of data combined with the expertise of researchers to analyse and interpret this data and extract valuable conclusions.

CoastSnap is a recent creation in this field. It extracts data about the location of shorelines from crowd sourced images to record shoreline changes over continuous, long periods (Harley *et al.*, 2019). This approach provides a good level of accuracy and temporal resolution at a more affordable cost than most alternative methods. In addition, the CoastSnap analysis software code has been made freely available for anyone to obtain. As a result, this citizen science monitoring approach has been adopted by locations throughout the world (NSW Department of Planning Industry and Environment, 2020).

However, whilst CoastSnap has been successfully adopted on hundreds of beaches, the code enabling the image analysis requires the proprietary platform MATLAB for its operation. As a result expensive MATLAB licensing fees as well as required skills for its operation represent a continued barrier for communities with limited resources. For example, one environmental management programme in the Caribbean has described the investment required for MATLAB as a barrier for human resources and the budgets of small Caribbean departments of government or NGOs (H Saheley 2021, personal communication, 10 February). This barrier could be minimised as free alternatives to MATLAB exist. Python is the most suitable alternative to MATLAB given the increasing adoption of this programming language within the coastal science and engineering community including on projects similar to CoastSnap (Conlin *et al.*, 2020).

If a Python version of CoastSnap can be created, its software will become completely free. This may enable regions, previously restricted by the cost of MATLAB licensing,

to implement the monitoring tool. Furthermore, CoastSnap will be less susceptible to project abandonment related to funding cuts (Conrad and Hilchey, 2011). It will therefore provide a greater opportunity for indefinite, continuous beach monitoring at a large scale. The results from a code conversion of the software may also be particularly useful for similar projects which aim to convert image processing functionality from MATLAB to Python.

Aim

To investigate and initiate the conversion of CoastSnap from MATLAB to Python.

Objectives

1. Understand the operation of CoastSnap.
2. Identify the critical functions in CoastSnap's MATLAB code.
3. Identify equivalent functions in Python and compare their performance to the MATLAB functions.
4. Begin production of the Python code.

These objectives are addressed in the following sections. The methodology provides an explanation of the of the processes enabling CoastSnap's operation. The critical functions which perform these processes are presented in the results and discussion section beside equivalent functions in Python. The performance of these Python functions is compared to the original functions in MATLAB. Some sections of the code have been produced in Python and can be run using supporting Jupyter Notebooks (details provided at the beginning of the results and discussion section).

For each of these objectives, this thesis focuses on the 'image georectification' and 'map shoreline' sections of the CoastSnap code. These sections comprise the image processing steps used in CoastSnap and are dependent on the statistics and image processing toolboxes in MATLAB, both of which require additional licensing fees. It was identified during discussions with stakeholders that the dependency on these toolboxes could be eliminated. This would require completing the 'image georectification' and 'map shoreline' code sections first, enabling the possibility of a hybrid MATLAB-Python version of CoastSnap while the remainder of the Python code is generated. These two sections are therefore the focus of this study.

Literature Review

Beach morphodynamics

Beaches are highly dynamic systems. Together with their dunes they provide a buffer for the coastline as sediment transport dissipates incident wave energy (Anthony, 2019). This is made particularly apparent during storm events when significant morphological changes can occur in a period of hours due to aggravated sediment transport. The effects can be influenced by a number of factors including wave conditions, tide level, and beach orientation (de Vries, Wengrove and Bosboom, 2020) and as a result, the world's beaches exhibit a variety of morphodynamic behaviour on both a temporal and spatial scale. A length of embayed coastline covering ten kilometres can host a range of morphological responses on its beaches at one time, including accretion, erosion, rotation and headland bypassing (Masselink, Scott, *et al.*, 2016). Whilst beaches can experience significant changes during these storm events, they demonstrate resilience during the ensuing period of recovery when lower energy wave conditions enable beach sediment to be redistributed (Castelle and Harley, 2020).

Sea level rise and more powerful storm wave conditions due to climate change are resulting in unprecedented impacts on the coastline (Masselink, Castelle, *et al.*, 2016). Beach dune systems can become imbalanced if the recovery period is unable to fully reverse the impacts of storm events. In such situations, beach migration can occur in the landward or seaward direction depending on the overall sediment transport balance and other factors such as the existence of non-erodible surroundings (Cooper *et al.*, 2020). Beach systems have evolved in such a way throughout history and many beaches exist as they are today due to landward migration across millennia as a result of postglacial sea level rise (Carter and Woodroffe, 1994). However, during the last century, global mean sea level rise has accelerated from a rate of 1.4mm per year (between 1901 – 1990) to 3.6mm per year (between 2006 – 2015) and it is predicted to

continue to accelerate (Oppenheimer *et al.*, 2019). At the same time, the global wave climate is being altered considerably (Hemer *et al.*, 2013) and as both of these influences on beach morphodynamics adjust to climate change, so too do the morphologic responses of the beaches themselves.

Relevance of coastal monitoring

These unprecedented impacts to beaches represent a large risk for communities and their activities in coastal regions. Coastlines provide opportunities for a range of economic activities including trade, tourism and real estate development and they are often highly desirable locations in which to live, due to their natural beauty. As a result, these regions are experiencing sustained population growth with human activities encroaching into dynamic beach dune environments, leaving themselves vulnerable to coastal processes (Neumann *et al.*, 2015). In locations where landward beach migration may occur, assets beyond the edge of the dune system are also at risk if migration is extensive.

In order to understand the risks presented by a beach system and implement appropriate management strategies, it is necessary to monitor beach behaviour.
Due to the disparity of beach behaviour in time and space, it is important to monitor beaches individually. Neighbouring beaches may experience significantly different morphodynamic behaviour, resulting in different or even opposing levels of risk to nearby property. With time, these risks will likely change with the conditions governing morphodynamic behaviour.

Coastal monitoring and the analysis of historical monitoring data is also critical to assess the potential consequences of shoreline management options. Hard engineering defences such as seawalls, groynes and breakwaters offer a defence for properties and

infrastructure directly at threat from coastal impacts. Historically these options have been popular and often admired by societies which have viewed the ocean as something to be defeated (Thom, 2020). However, these physical defences interrupt the sediment balance provided by natural beach evolution and are generally observed to have detrimental effects at larger spatial and temporal scales (Cooper and McKenna, 2008). For example, sediment supplies can be depleted for beaches which neighbour defence structure locations (Fitton, Hansom and Rennie, 2016) and entire beaches can be eroded from the foot of seawalls leading to disastrous environmental and social implications (Thom, 2020). In addition, such defences can incentivise development behind them, further increasing the dependency on them, however, the cost of maintenance and enhancements will continue to increase with more severe future conditions. This is a cost for future generations which could be avoided with alternative management strategies (Cooper and McKenna, 2008). Alternative management options such as relocating communities and restricting development in at risk locations offer long term advantages but rely on monitoring to determine which areas will be at risk in the future.

Preceding coastal monitoring methods

The first coastal monitoring surveys to analyse morphodynamic behaviour depended on in-situ methods, such as the Emery method, requiring a pole and a measuring tape to record cross-shore widths of beaches (Turner *et al.*, 2016). These surveys were carried out by a relatively small group of coastal researchers investigating the processes occurring at the shoreline, becoming particularly eminent during the 1970s.(Short and Jackson, 2013).

With advances in technology, coastal monitoring methods have developed since the 1970s. The in-situ methods now have remote sensing alternatives which eliminate

much of the time and effort required to carry out regular surveys. For example, the pioneering coastal imaging system Argus is able to continuously extract data using fixed cameras overlooking beaches (Holman and Stanley, 2007). More recently, the feasibility of satellite data to supply global shoreline information has improved significantly due to enhancements in its accessibility and resolution combined with access to exceptional processing capabilities through platforms such as Google Earth Engine (Luijendijk and de Vries, 2020). Similarly, airborne and terrestrial LiDAR systems are able to frequently and rapidly scan expansive beaches areas to provide high quality topographic data for analysis (Le Mauff *et al.*, 2018).

Limitations of current methods

Whilst these remote sensing methods improve the efficiency of data collection, their implementation is often constrained by high costs (Harley *et al.*, 2019), limiting the ability for coastal monitoring to be adopted at a large scale and in areas with insufficient access to funds. Satellite data continues to be constrained by low temporal and spatial resolution to be useful for the analysis of small scale or short term beach processes such as storm recovery. Constraints such as these have led researchers to the consensus that professional science alone is unable to provide the resolution and scale of data necessary to properly understand environmental change (Roger *et al.*, 2020). This is an issue in the context of using coastal monitoring to investigate the implications of climate change for coastal regions. For this coastal monitoring must provide data at a large spatial scale, in large volumes, across a long-term period.

Requirement of coastal monitoring to provide data at a large spatial scale

There is minimal observation data for the majority of the world's coastlines which makes it difficult to fully understand and adapt to their behaviour (Luijendijk *et al.*, 2018; Harley *et al.*, 2019). For example, few beaches have been the subject of a

continuous, multi decadal monitoring programme like that at Narrabeen-Collaroy beach (Splinter, Harley and Turner, 2018). As a result the beach's comprehensive database has been used to develop a deep understanding of its response to varying conditions (Splinter, Harley and Turner, 2018). However, beach characteristics vary greatly in different regions and it is not appropriate to apply the principles developed from Narrabeen-Collaroy to other locations. It is therefore important to extensively study beaches with varying characteristics, such as beach slope, sediment type, and wave conditions (Eichentopf, Karunarathna and Alsina, 2019). This can only be achieved by monitoring beaches across all regions.

Analysing data at a large spatial scale can also enable processes occurring across expansive regions to be revealed. For example, data collected from locations around an ocean basin may be used in the future to recognise patterns in climatic impacts on coastlines (Splinter, Harley and Turner, 2018).

Requirement of coastal monitoring to provide large volumes of data

Large volumes of data, concentrated at particular times and locations are valuable to improve the understanding of processes of interest. One example is the analysis of beach recovery after storm surges. This process is poorly understood and often overlooked in storm sequence investigations (Eichentopf, Karunarathna and Alsina, 2019). The collection of large data volumes of the beach profile before and during the recovery period of the storm is important to enable a high temporal resolution of the process for analysis (Eichentopf, Karunarathna and Alsina, 2019). As climate change drives more powerful surges against beaches in inhabited regions, the analysis and understanding of beach recovery will become increasingly critical. Many nearshore beach processes, such as swash dynamics, require a similarly concentrated spatial and temporal resolution of data to be collected (Splinter, Harley and Turner, 2018).

For gradually occurring processes such as beach rotation, it is also important to collect large volumes of data. However instead of being concentrated over a short duration, it is more important that data samples are frequently collected over a long period. This makes it possible to differentiate between permanent morphological changes and fluctuating coastal processes (Harley *et al.*, 2019). For example if data is collected yearly from a beach, trends may indicate accretion of the beach. However if more frequent data can be analysed, it may become clear that the beach fluctuates between accretion and erosion due to seasonal variations in wave processes.

Requirement of coastal monitoring to provide long-term, continuous data

Long-term and continuous coastal monitoring data are important in the creation of coastline forecasting models. Empirical shoreline models, for example, are calibrated using shoreline data provided by various monitoring techniques (Splinter, Turner and Davidson, 2013). Typical models require at least monthly data samples over a two year period before they can be calibrated. Longer forecasting models require calibration data samples over longer periods (Splinter, Turner and Davidson, 2013). The ability to predict and prepare for coastal impacts thus depends on the access of models to long term, uninterrupted data samples.

Citizen Science

Cheaper, approaches to coastal monitoring are therefore required to enable more extensive coverage and delivery of these demands required of coastal monitoring. One approach is the use of citizen science. Citizen science is defined as, the involvement of the public in scientific research, whether community driven research or global investigations (Citizen Science Association, 2020). With the public's increasing access to high quality equipment (found in smart phones and laptops, for example), it is

becoming an increasingly popular method for performing scientific research around the world, particularly in disciplines associated with the natural environment (Roger *et al.*, 2020). For example, the Great Reef Census is a citizen science project to survey the Great Barrier Reef (only 5-10% of which is regularly surveyed) by mobilising members of the public to provide and help analyse images (Citizens of the Great Barrier Reef, 2020). This approach enables a synergy due to the magnitude of resources provided by the public combined with the expert knowledge of researchers. In the realm of beach monitoring, a project in Victoria, Australia determined that citizen scientists using drones to collect topographic data across beaches generated data as unbiased and accurate as professional researchers (Pucino *et al.*, 2021). This exemplifies the ability of the public to generate professional standard data.

Whilst citizen science provides an effective means to generate large volumes of high quality data, it is also widely acknowledged as a means to educate the public about the topic under investigation (Roche *et al.*, 2020). With the appropriate implementation of a citizen science project members of the public can be made aware of, be educated on and even be engaged by issues they were previously unaware of. This is particularly valuable for the issue of climate change impacts on coastal erosion which the public is not sufficiently informed about (Committee on Climate Change, 2018).

CoastSnap

CoastSnap is another beach monitoring approach which seeks to take advantage of these opportunities provided by the principle of citizen science(Harley *et al.*, 2019). The public are instructed to send researchers beach images captured from cradles located at fixed locations. Dedicated software, using a functionality similar to those in established coastal imaging systems such as Argus, then extracts quantitative data of the shoreline location. The cradles are often positioned at locations experiencing a

regular flow of passing people who provide a long term regular source of high quality images. The accuracy of the data obtained by CoastSnap during its development in South-East Australia is similar to conventional coastal imaging approaches (Harley *et al.*, 2019).

The creators of CoastSnap have enabled the project to be especially influential through the concept of open science. This concept is explained by Buckheit and Donoho who clarify that the impact of research is not provided by the results alone, but by the full software environment, code and data used to undertake the research (Buckheit and Donoho, 1995). This is particularly applicable for CoastSnap as it is used to monitor individual beach sites. Therefore to enable CoastSnap beach monitoring on as many beaches as possible, it is necessary to provide researchers at other beach locations with the same tools to monitor their own location. As a result the software code for CoastSnap is open source and has been adopted in locations around the world (NSW Department of Planning, Industry and Environment, 2020).

Financial limitation of CoastSnap

While the CoastSnap code is open source, it is not completely free to operate. This is because the code runs using the MATLAB platform. MATLAB is proprietary, requiring a license purchase and this cost could prohibit the ability to adopt the CoastSnap method of coastal monitoring. Alternative free and open source (FOS) programming languages exist which would enable CoastSnap to be adopted without any licensing costs, however, the code translation is expected to be a significant task. Such a task is justified, since CoastSnap as an FOS software is likely to bring significant advantages for its applicability.

Effect of FOS software on practicability of CoastSnap in developing regions

The majority of citizen science projects are found in the developed, western world, particularly across North America, Europe and Australia with comparatively few found in Africa, Asia and Central and South America (Chandler *et al.*, 2017). There are many barriers in the developing world which contribute to the absence of citizen science, including limited awareness of the opportunities, limited organisational capacity and lack of familiarity with citizen science (Pocock *et al.*, 2019).

While many barriers are difficult for project designers to address, the software used can help to address some barriers, including limited access to technology and inadequate funding (Pocock *et al.*, 2019). FOS software is being increasingly adopted in developing countries due to the ability to be free from proprietary licensing (Chen *et al.*, 2010). FOS software can therefore be beneficial if the CoastSnap adopter does not need to depend on having proprietary software which may be unaffordable.

However, FOS software may not always be favourable in developing countries if someone wishing to adopt a project is unfamiliar with the FOS software required. A study on academic researchers based in Nigeria, Ghana and Bangladesh identified that proprietary software was overwhelmingly relied upon to conduct their research (Vermeir *et al.*, 2018). FOS software alternatives to the proprietary programs used were available, however a lack of awareness was identified as a major reason for them not being adopted, with 75.6% of respondents indicating that they did not know that FOS software was available. This indicates that if CoastSnap is being designed with the intention for it to be implemented by academic researchers in developing countries, FOS software may not be beneficial at the present time.

While this is presently the case, it is likely to change as knowledge of FOS software changes. The same study by Vermeir et al. established that 87.1% of respondents either agreed or strongly agreed that they were interested in FOS software. This indicates that the desire exists in academic researchers in developing countries to widely adopt FOS software. This also identifies an opportunity for CoastSnap to be means to disseminating awareness of an FOS programming language such as Python in order for FOS software using this language to become accessible.

Regarding wider society outside of academic research, the potential of FOS software adoption is being realised in many developing countries. GitHub, a platform enabling collaboration on open source projects (including CoastSnap), identified Nigeria as the country experiencing the largest growth in contributions (65.9%), with Bangladesh in fourth place and the top ten being held largely by developing countries (Forsgren *et al.*, 2020). This indicates that software skills in developing countries are becoming increasingly oriented around FOS software, especially those using the Python programming language (Forsgren *et al.*, 2020). FOS software will therefore hold an increasingly high potential to enable the practicability of CoastSnap in developing countries in the future.

Effect of FOS software on different parts of society to adopt CoastSnap

The impact of climate change is having disastrous implications for coastal communities in developing countries (Dasgupta *et al.*, 2009). These are the regions which have the most limited resources in order to monitor and adapt to coastal change. Such injustices can be found in the occurrence of many environmental problems where impacts are over proportionally felt by poorer communities (Li *et al.*, 2018).

The locations of coastal investigations have traditionally been determined by professional science, however, with citizen science and FOS software, regular members of any community can access the resources required to investigate environmental impacts themselves.

For example, one initiative known as HabitatMap is enabling city inhabitants, who have no previous experience of conducting scientific research, to investigate levels of particulate matter in their area (Lim *et al.*, 2019). The concept enables citizen scientists to install low cost sensors and record particulate levels using FOS software provided by HabitatMap (HabitatMap, 2020). This software can be modified to suit the needs of their own monitoring project (such as enabling higher frequency recordings). This enables members of the community to undertake environmental monitoring by minimising the cost of the monitoring software.

The principles of HabitatMap in coastal monitoring projects such as CoastSnap can empower coastal communities to investigate and evidence coastal changes themselves. This eliminates a community's dependency on professional science and will provide a greater opportunity for the community to foresee and adapt to changes.

Effect of FOS software on the ability to maintain a CoastSnap project

The issue of long term funding is another barrier for citizen science which can be alleviated using FOS software. In many cases, projects receive funding contributions from outside parties such as government and industry (Roger *et al.*, 2020). Projects can be dependent on these contributions to cover recurring costs including software licencing fees. This poses a problem if funding is withdrawn as the project no longer has access to the required resources (Conrad and Hilchey, 2011). Even projects which are recognised to provide valuable data can lose funding contributions (MacPhail and

Colla, 2020) so it is important to minimise long term funding dependencies. The best way of ensuring this is to minimise recurring costs. It is therefore desirable, for CoastSnap to be an FOS software. This improves the potential to maintain coastal monitoring at a location indefinitely.

Suitability of Python as the FOS programming language to be adopted by CoastSnap

Python has been determined to be the most suitable FOS programming language to convert CoastSnap's original MATLAB code to. Python is becoming increasingly popular both amongst the general population, especially in developing regions (Forsgren *et al.*, 2020), and amongst the scientific community (Magaña *et al.*, 2020). Furthermore, Python syntax is similar to that of MATLAB meaning that existing CoastSnap users who can use MATLAB will be able to adapt to Python straightforwardly. Python's suitability to undertake the operations required in CoastSnap is exemplified by the program SurfRCaT (Conlin *et al.*, 2020) in which surf camera images are georectified, a major step in CoastSnap's operation. The following sections will investigate the initiating steps of the conversion of CoastSnap from MATLAB to Python.

Methodology

Note: much of the theory explained in this methodology has been presented by Harley et al. in the original publication on CoastSnap (Harley *et al.*, 2019).

Image Georectification

Georectification is the process in which an image's pixels with coordinates U and V are translated into a world coordinate system, x, y and z. This converts the original, oblique form of an image to a plan view. In CoastSnap, this is necessary to standardise the shore line position results for comparison by locating them in world coordinates. Figure 1 illustrates an oblique image and the result after it is georectified.

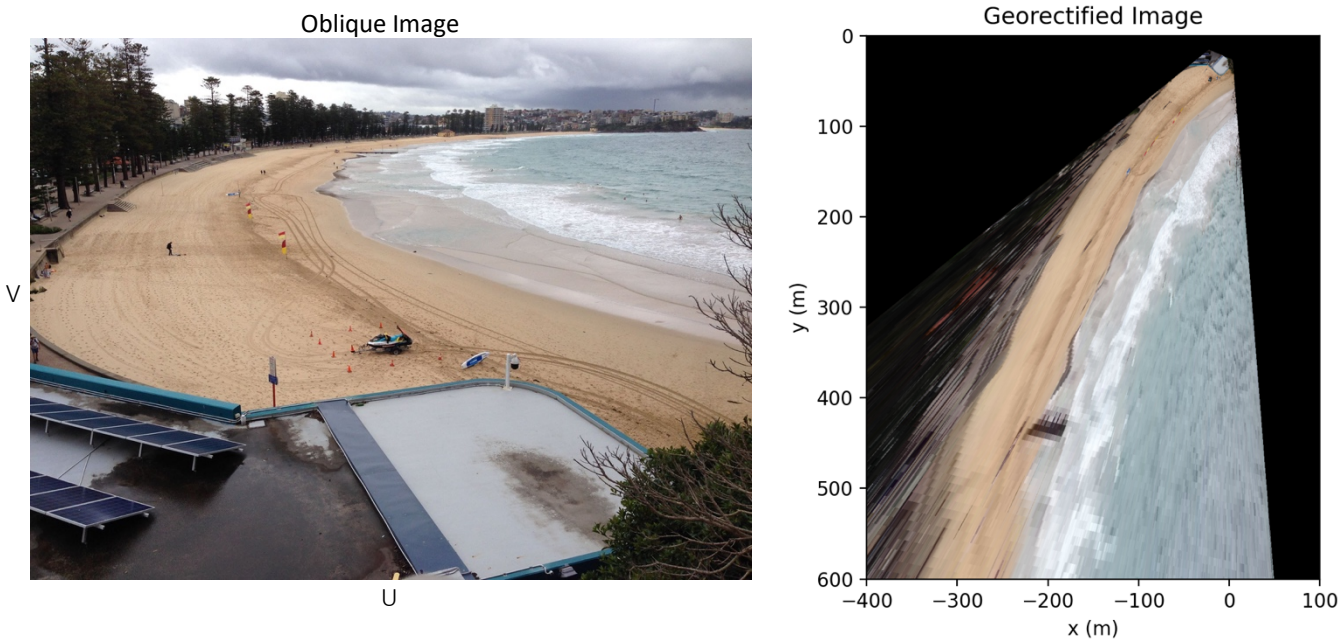


Figure 1: Image before and after georectification

Upon an elementary investigation into the operation of CoastSnap's MATLAB code, the basic operational structure was identified and can be found in Fig. 2. The operational steps which are circled by an orange dashed line have been investigated further in the results and discussion section. The focus has been applied to these sections as they have been anticipated to pose some challenges when converting the code from MATLAB to Python. This is due to the specialised nature of each of these

tasks which are likely to require functions that are not commonly used in other programs. The steps are taken in order to georectify the image by applying the pinhole camera model. This model will be explained now to give context to these steps.

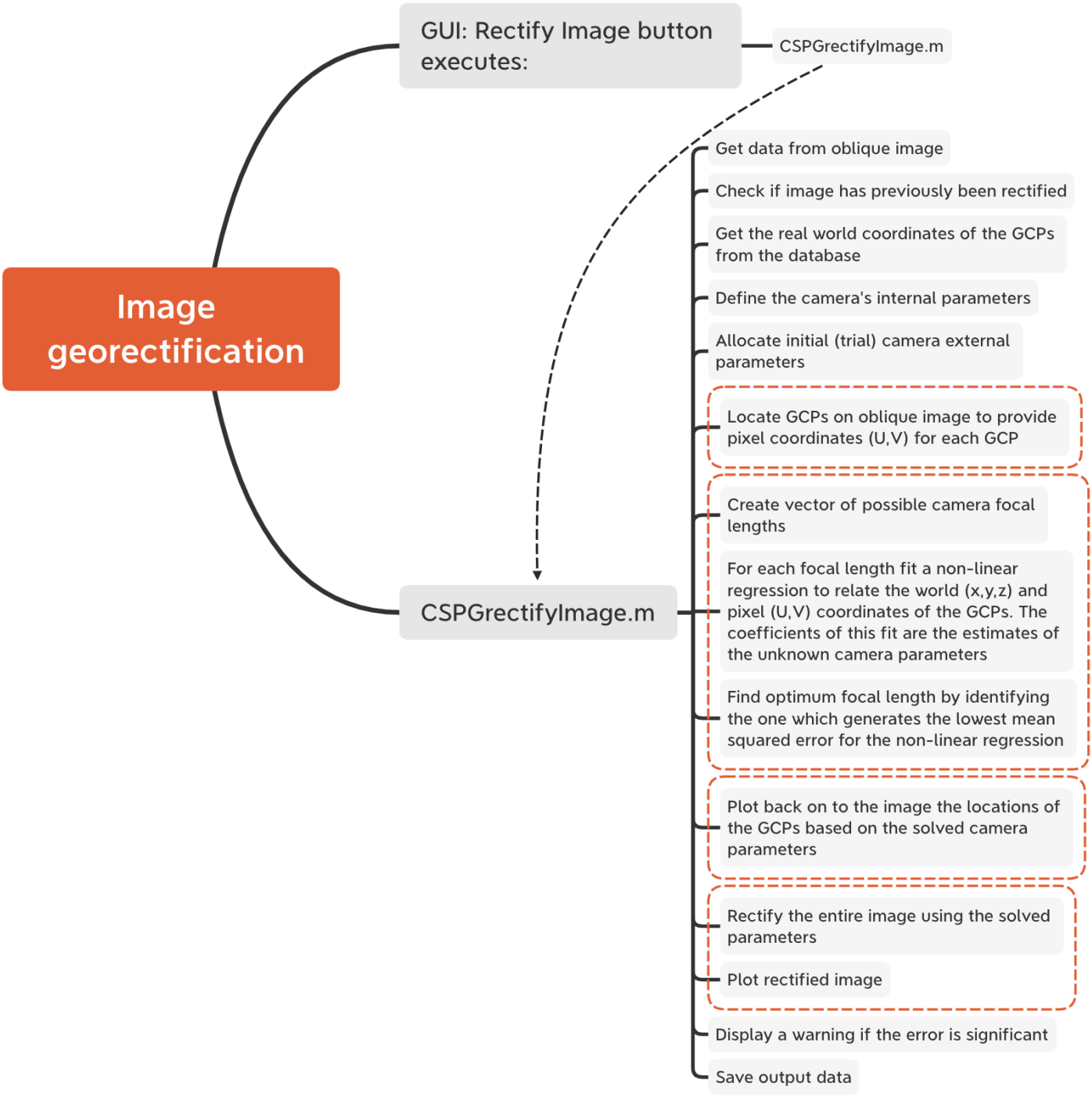


Figure 2: Basic operational structure for image georectification in CoastSnap

The pinhole camera model (Hartley and Zisserman, 2011)

The pinhole camera model converts an image's pixel coordinates (U,V) to the world coordinate system (x,y,z) using the camera's intrinsic and extrinsic parameters in the following transformation:

$$\begin{bmatrix} U \\ V \\ 1 \end{bmatrix} = P \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Before defining the projection matrix, **P**, it is necessary to list each of the extrinsic and intrinsic parameters required:

- **Intrinsic camera parameters** (determined by the camera's configuration):
  - Focal length, **f.**
  - Skew, **s.**
  - Pixel coordinate of the principle point in the U axis, **c0U.**
  - Pixel coordinate of the principle point in the V axis, **c0V.**
  - Pixel aspect ratio γ.

- **Extrinsic camera parameters** (determined by the spatial position of the camera):
  - Camera x position, **x.**
  - Camera y position, **y.**
  - Camera z position, **z.**
  - Azimuth, α.
  - Tilt, **t.**
  - Roll, **r.**

The extrinsic parameters, azimuth, tilt and roll, are illustrated in the diagram in Fig.3.

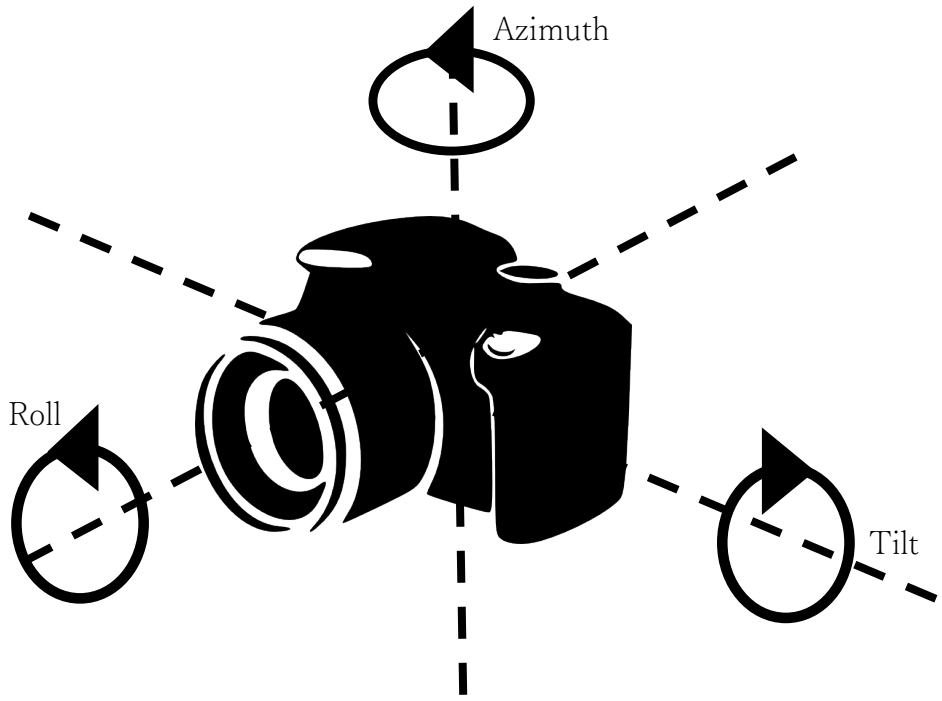


Figure 3: Visual representation of azimuth, tilt and roll

Now the projection matrix P and its constituents can be presented:

$$P = KR[\, I \mid -C \,]$$

Where:

- $K = \begin{bmatrix} f & s & c0U \\ 0 & \gamma f & c0V \\ 0 & 0 & 1 \end{bmatrix}$
- R is the rotation matrix (3x3) defined by azimuth (α), tilt (t) and roll (r). Appendix I displays the formulation of each element;
- I is the identity matrix: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ and;
- C is a vector of the camera location: $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$.

In CoastSnap the following **assumptions** are made for matrix *K* to simplify the pinhole camera model's application (Harley *et al.*, 2019):

- Skew, **s** is 0.
- The pixel coordinates of the principle points, **c0U** and **c0V** are at the image centre.
- The pixels are square so the pixel aspect ratio, γ is 1.

Following these assumptions, the projection matrix, *P*, contains 4 **unknowns**:

- Focal length, **f**.
- Azimuth, α.
- Tilt, **t**.
- Roll, **r**.

Thus, in order to apply the pinhole camera model and perform the conversion of the image coordinates, these 4 unknowns must first be found.

The steps required to resolve these unknowns are analysed in the results and discussion section along with the proposed equivalent Python code. In addition, the code which applies this model and generates the plan image is analysed thereafter. The principles used in CoastSnap for each step will now be explained along with the requirements to be checked to ensure that the Python recommendations will perform satisfactorily.

a. Identification of ground control points.

In CoastSnap the fundamental requirement for resolving the unknown parameters is the identification of the location of ground control points (GCPs) within the oblique image by the user. The GCPs are prespecified, fixed locations for which the world coordinates (x,y,z) are known. In order for the user to identify the location (U,V) of the GCPs, a user interface is required which enables the locations to be selected by the click of a mouse. The interface provided by CoastSnap is presented in Fig. 4.

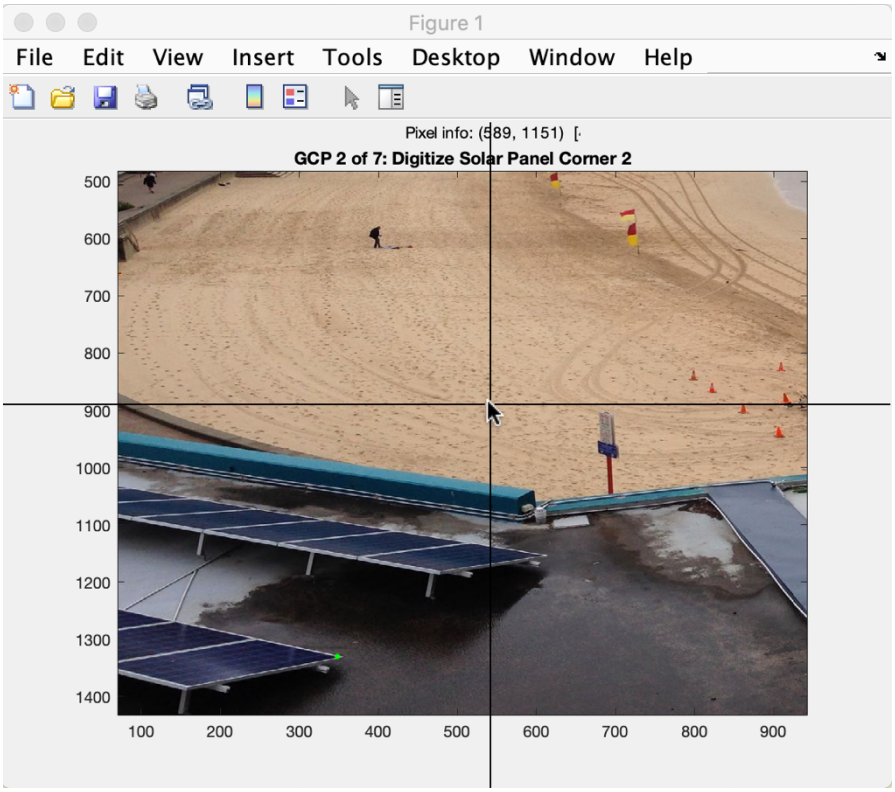


Figure 4: CoastSnap user interface to locate GCPs

To ensure a satisfactory equivalent in Python a user interface must be generated which can operate in a very similar manner to that in MATLAB. The functions used must also enable the extraction of the (U,V) coordinates which to be contained within a variable for manipulation.

b. Resolving of the unknown camera parameters.

Now that the GCPs have been assigned (U,V) and (x,y,z) coordinates, this sample can be used to infer the unknown parameters which will enable the pinhole camera model to generate the (x,y,z) from the given (U,V) coordinates. Figure 5 presents a visual representation of this concept.

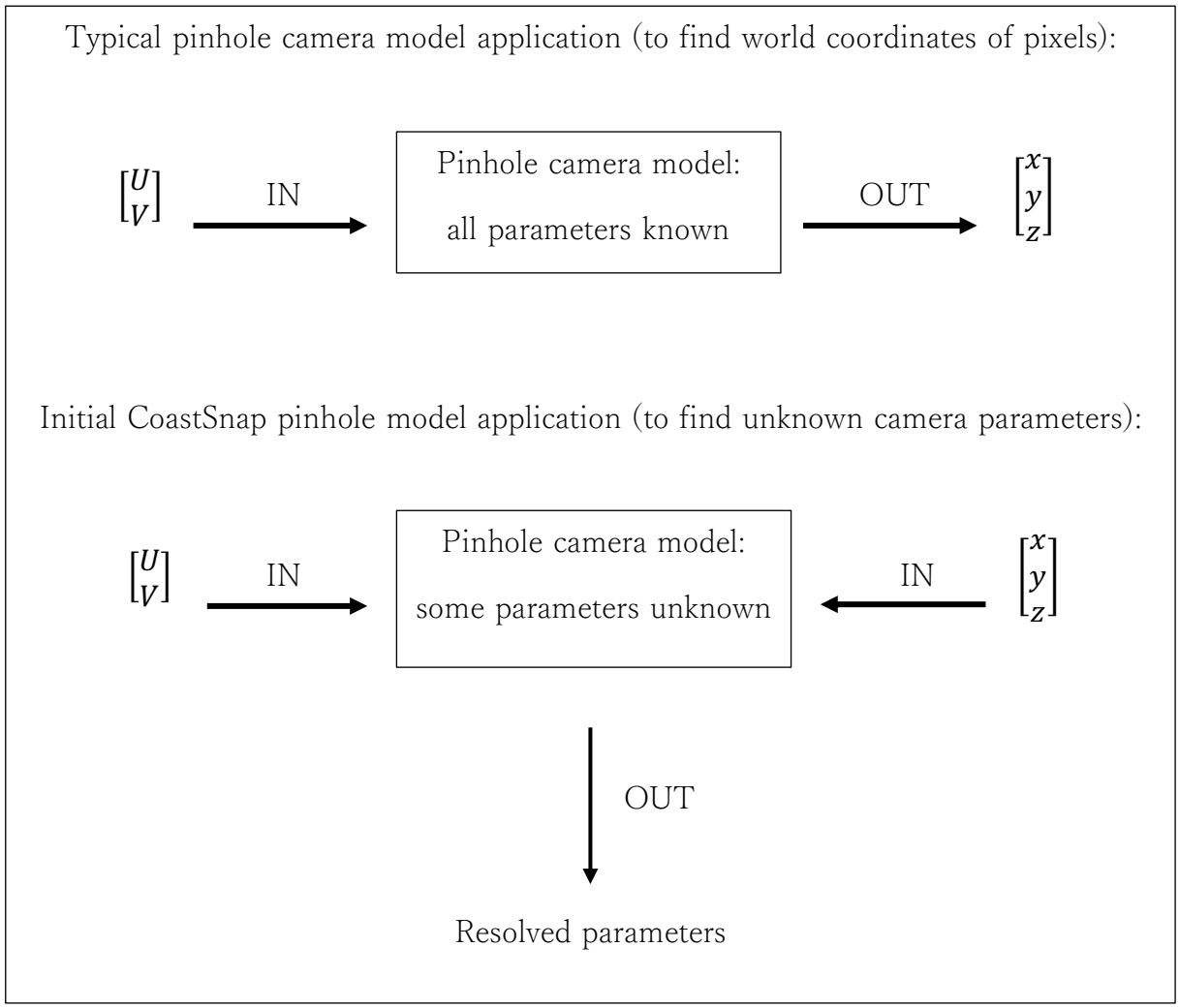


Figure 5: Explanation of the pinhole model's application to resolve unknown camera parameters in CoastSnap

In order to infer the unknown camera parameters in the pinhole camera model, it is necessary to fit a non-linear regression of the model for the situation described by provided (U,V) and (x,y,z) coordinates. The function which generates this fit must

return the resolved model parameters for this situation. These then become the estimates for the unknown camera parameters. Therefore, the Python function recommended for the fitting of the non-linear regression must be shown to return the same parameters as the MATLAB equivalent.

It must be noted that in CoastSnap, this method only identifies 3 of the 4 unknown parameters: azimuth, tilt and roll. The fourth unknown, focal length is resolved by repeating the non-linear regression fit for a range of possible focal lengths. The mean squared error value (mse) of each of these fits is analysed and the focal length responsible for the minimum mse is taken as the resolved focal length. The Python function should be tested to ensure that it provides the same mse values as those generated by its equivalent in MATLAB.

c. Plot the GCP locations generated by the resolved camera parameters onto the oblique image.

With all parameters of the pinhole camera model resolved, the model can be applied in reverse to obtain pixel coordinates (U,V) from world coordinates (x,y,z) as is shown in Fig. 6.

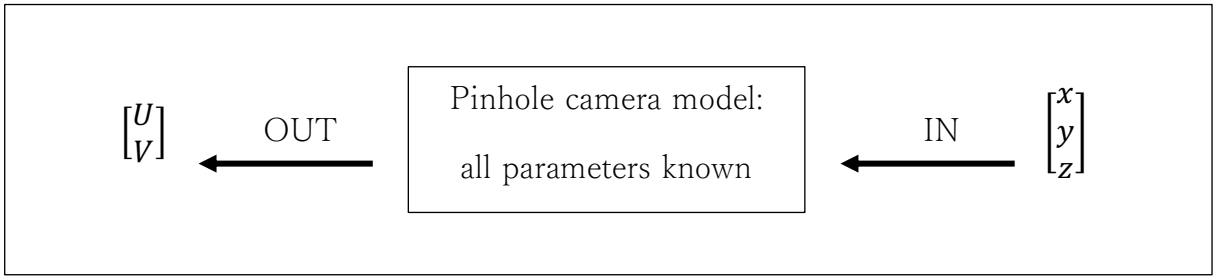


Figure 6: Reversal of the pinhole camera model

Applying the model in this way using the resolved camera parameters enables the known GCP world locations to be converted to (U,V) coordinates and plotted back on to the oblique image. This enables the locations of GCP points within the oblique image, identified by the user, to be compared with the corresponding locations generated by the resolved parameters in pinhole camera model. This provides a visual representation of the accuracy of the camera parameter resolving process.

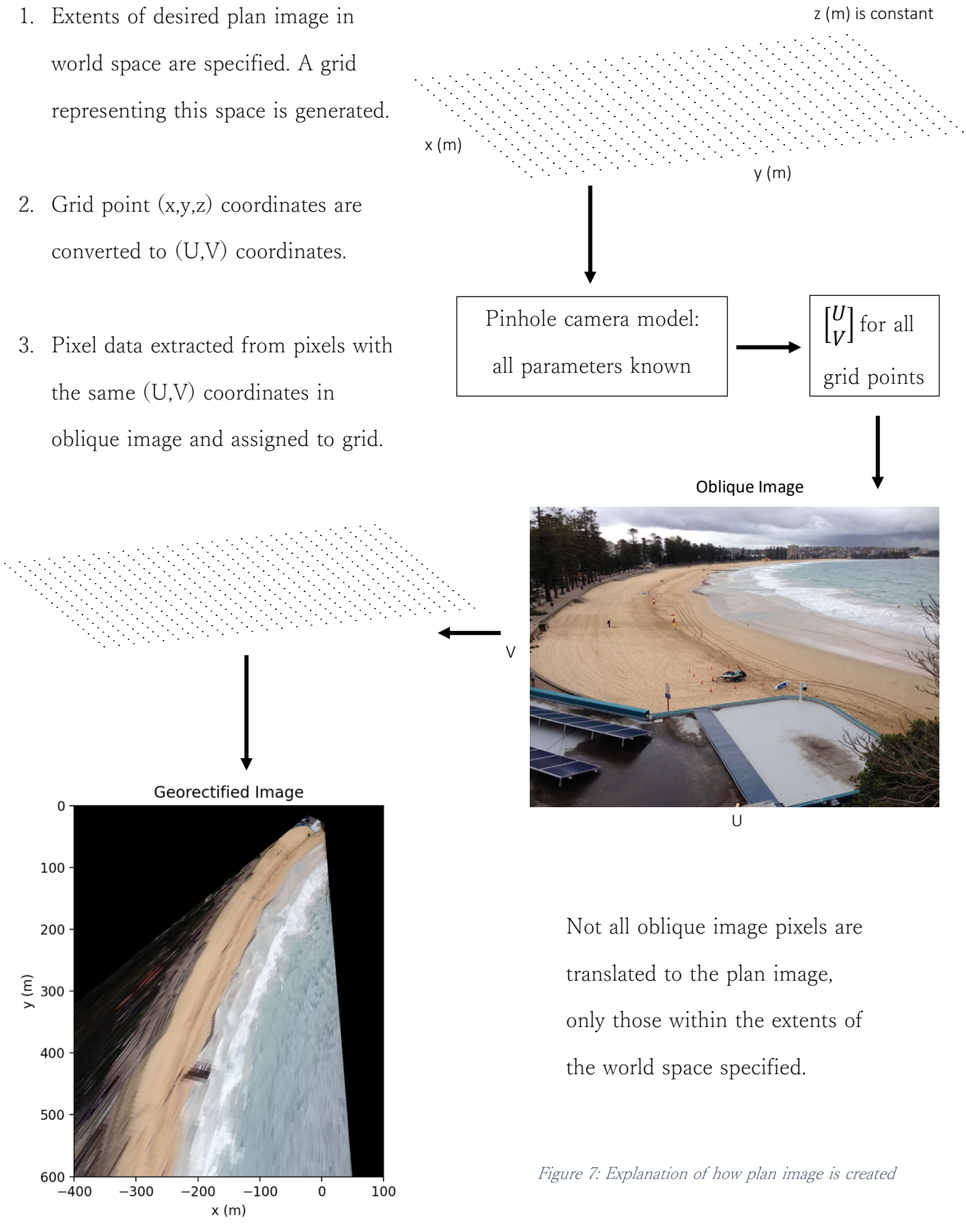
d. Create the plan image in the world coordinate system.

Similarly to the regeneration of the GCPs, the plan image is created by again applying the pinhole camera model in reverse. The user first defines the desired x and y extents of the plan image (in the CoastSnap database) from which a grid of points is created that will hold the plan image pixel data. Each point in this grid is assigned world coordinates. Thereafter, the (x,y,z) coordinates of these points (note, z for each grid point is the height of the shoreline) are applied to the reversed pinhole camera model to determine their corresponding coordinates in the (U,V) system of the oblique image. In short, instead of transferring every pixel from the oblique image to the (x,y,z) world coordinate system, CoastSnap defines a grid which will extract the corresponding pixel data from the oblique image to create a plan image. This concept is presented visually in Fig. 7. Note: by using this method, the points of the world grid don't always have associated (U,V) coordinates lying within the oblique image and therefore do not have pixel data assigned. These points are represented by the black regions in the georectified image in Fig.7. The Python code must be tested to ensure it returns the same world grid containing the pixel data as returned using CoastSnap in MATLAB.

1. Extents of desired plan image in world space are specified. A grid representing this space is generated.

2. Grid point (x,y,z) coordinates are converted to (U,V) coordinates.

3. Pixel data extracted from pixels with the same (U,V) coordinates in oblique image and assigned to grid.



Not all oblique image pixels are translated to the plan image, only those within the extents of the world space specified.

Figure 7: Explanation of how plan image is created

Map Shoreline

After the image has been georectified, CoastSnap next undertakes a number of processes to identify the location of the shoreline. The basic operational structure of CoastSnap in MATLAB has again been presented in Fig. 8, this time for the processes required to map the shoreline.

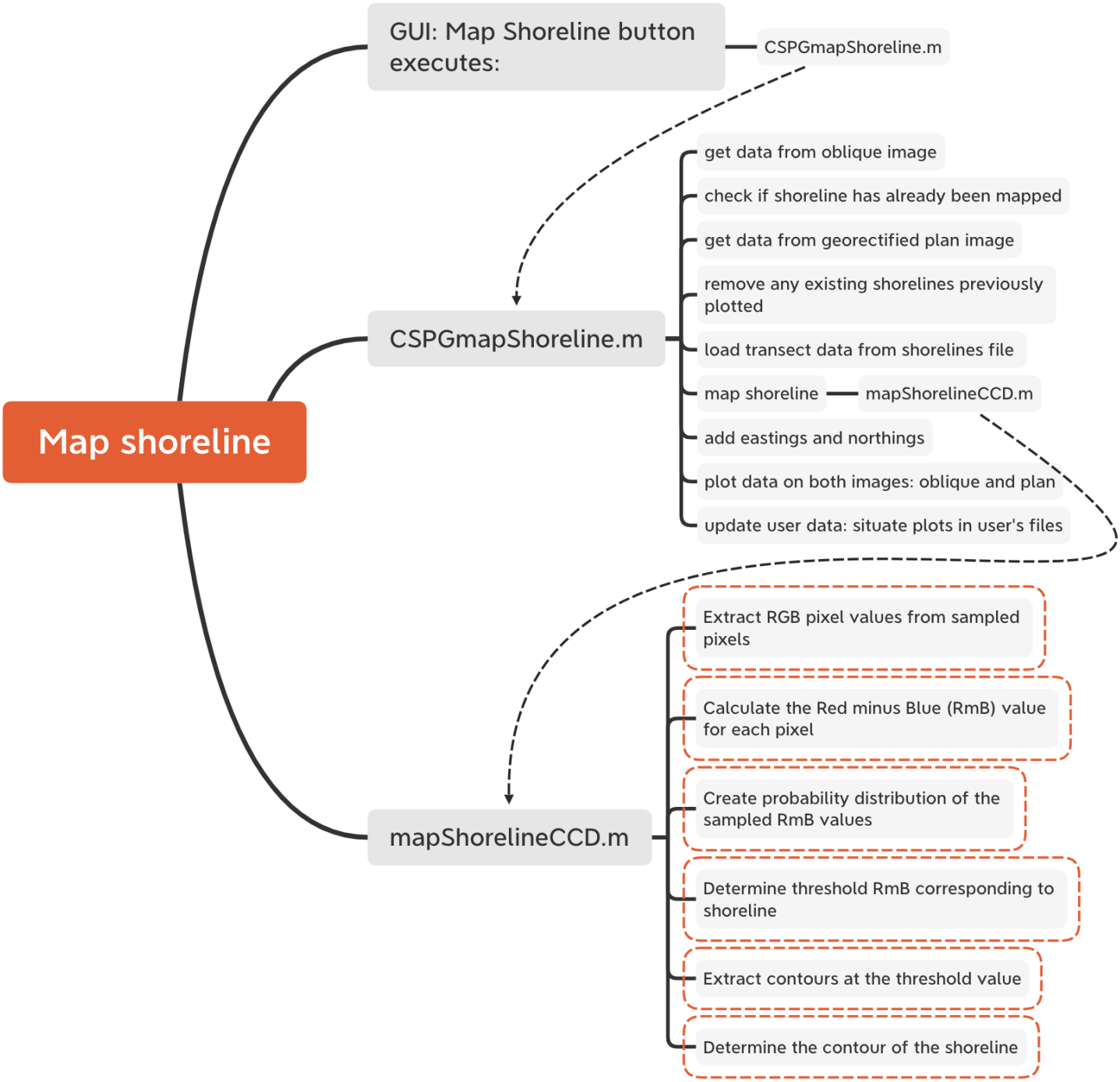


Figure 8: Basic operational structure for shoreline mapping in CoastSnap

The operational steps which are circled by an orange dashed line in Fig. 8 have been investigated further in the results and discussion section. It can be seen that these processes operate inside the function *mapShorelineCCD.m*. The focus has been applied to these sections as they have been anticipated to pose some challenges when converting the code from MATLAB to Python. This is due to the specialised nature of each of these tasks which are likely to require functions that are not commonly used in other programs. The steps outside of those circled in orange are not expected to require as much attention and are therefore not covered in this thesis.

The principles used in CoastSnap for each step involved in the mapping of the shoreline will now be explained. The requirements to be checked to ensure that the Python code recommendations will perform satisfactorily are presented.

a. Extract the red, green and blue (RGB) pixel values from sampled points.

For each CoastSnap location a set of transects are defined across the region of the beach. These run perpendicular to the shoreline beginning in the sandy region, crossing the shoreline and ending in the water region. Figure 9 presents an example of the transects used for images taken at the Manly Beach CoastSnap location in South East Australia. The transects are used as guides along which pixels in the plan image are sampled.

Each pixel contains a red, green and blue (RGB) pixel value, each between 0 and 255. The combination of these pixel values is responsible for the final colour visible in the image. CoastSnap extracts each of the RGB values for a set of sample points which occur along the transects at uniform intervals. For example, the system of transects in Fig. 9 generates approximately 60 000 sample sets (each containing all RGB values).

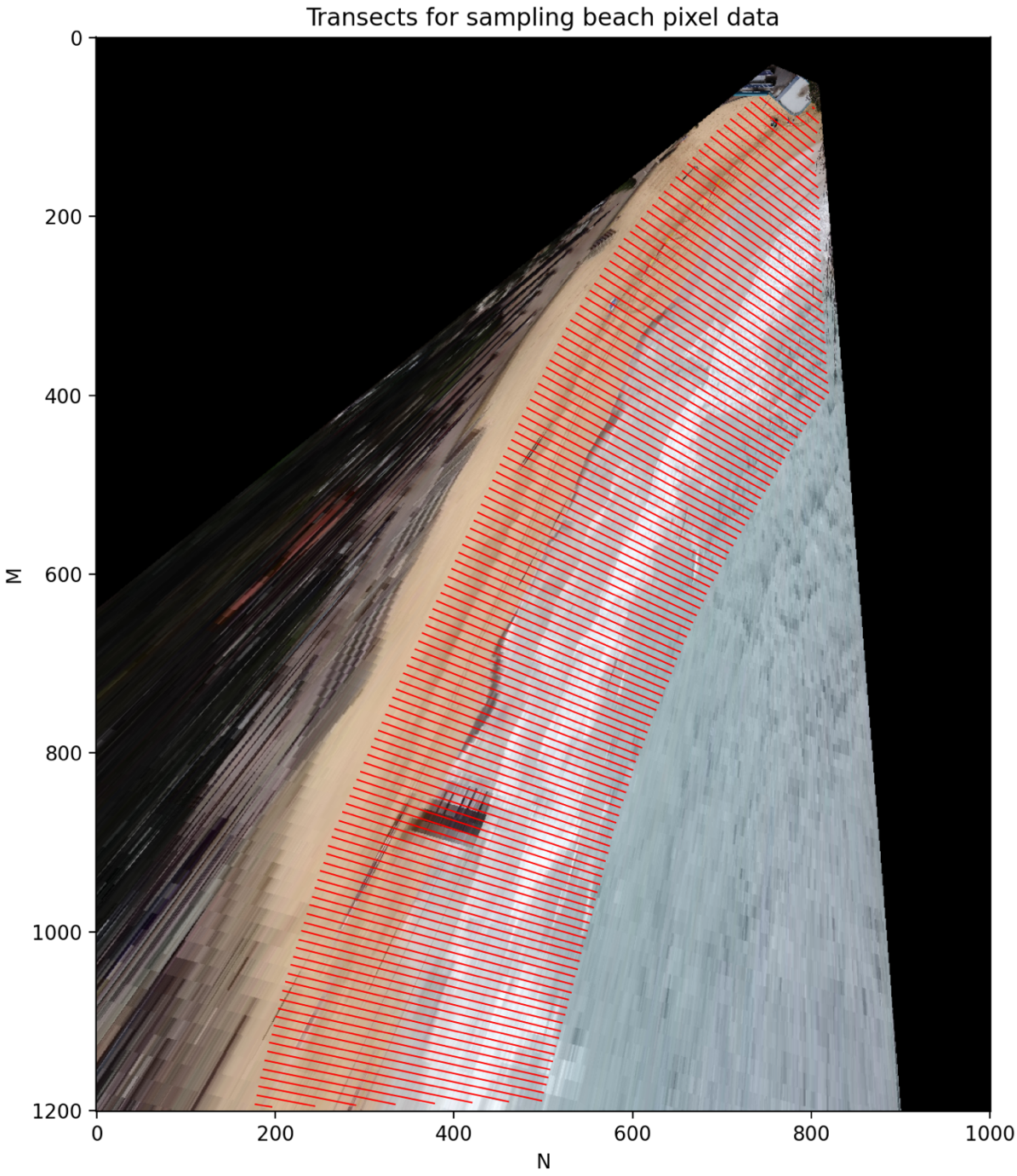


Figure 9: Example transect system for collecting pixel samples

For the Python code to perform satisfactorily, it should be tested that a similar number of sample points are generated across the same transect system as that used in CoastSnap using MATLAB. The similarity of the data sets generated must be analysed and they will ideally return identical results.

b. Calculate the red minus blue (RmB) value for each pixel.

Only the red and blue pixel values are of interest to CoastSnap. This is due to the typically large contrast between red and blue values of the pixels representing sand and the pixels representing water (sand pixels contain a high red pixel value and a low blue pixel value. Water pixels contain a high blue pixel value and a low red pixel value). To make this contrast especially clear, CoastSnap subtracts the blue value from the red value to generate the red minus blue (RmB) value for each pixel (sand pixels have high RmB value, water pixels have a low RmB value). With the intensified contrast between sand pixels and water pixels using RmB values, CoastSnap is able to extract the shoreline in the following steps.

c. Create a probability distribution of the sampled RmB values.

The RmB values from all of the samples are used to generate a probability distribution. The probability of each RmB value occurring is assigned a value. Figure 10 provides an example of a probability distribution which could be expected.

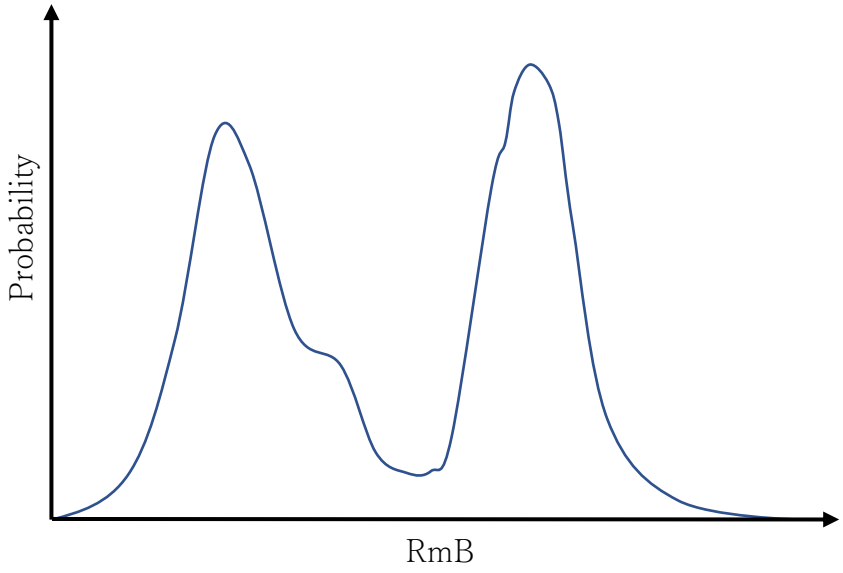


Figure 10: Typical probability distribution

It will be necessary to compare the distributions returned by Python and MATLAB functions especially the shapes and peak values for the following steps.

d. Determine the threshold RmB value corresponding to pixels on the shoreline.

The probability distribution will contain two peaks. The peak on the left corresponds to a lower RmB value which is therefore the most probable RmB of pixels in the water region (RmBwet). Similarly the right peak corresponds to a higher RmB which is therefore the most probable RmB of pixels in the sand region (RmBdry).

CoastSnap uses the RmB values at these peaks to determine the threshold RmB value which corresponds to the shoreline. The default version of CoastSnap determines the threshold by applying weightings to each peak value using equation:

$$Threshold\ RmB\ = \frac{1}{3}RmBwet + \frac{2}{3}RmBdry$$

It must be noted that the weightings were determined during CoastSnap's development in South-East Australia (Harley *et al.*, 2019). The colours of sand and water on beaches around the world can vary greatly to those found in this region and the threshold weightings should be adapted to account for this. Figure 11 illustrates the location of the threshold and the peaks used in its determination.

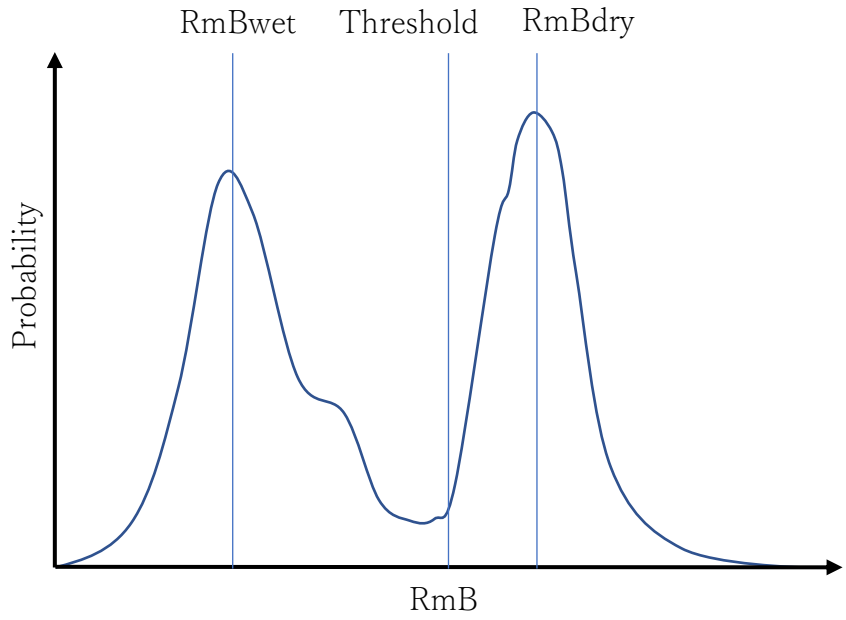


Figure 11: Threshold determination

The same values of the peaks RmBwet and RmBdry produced in MATLAB and Python will be required to produce the same threshold value. The value returned in Python must be closely analysed to check that they match those produced in the MATLAB code.

e. Extract contours along the threshold value.

Now that the threshold RmB value of the shoreline is established, CoastSnap searches the region bounded by the transects for strings neighbouring pixels which have an RmB value equal to the threshold. The of the contours generated, one represents the shoreline. Figure 12 presents an example of a series of threshold contours found within the transect region at Manly beach.

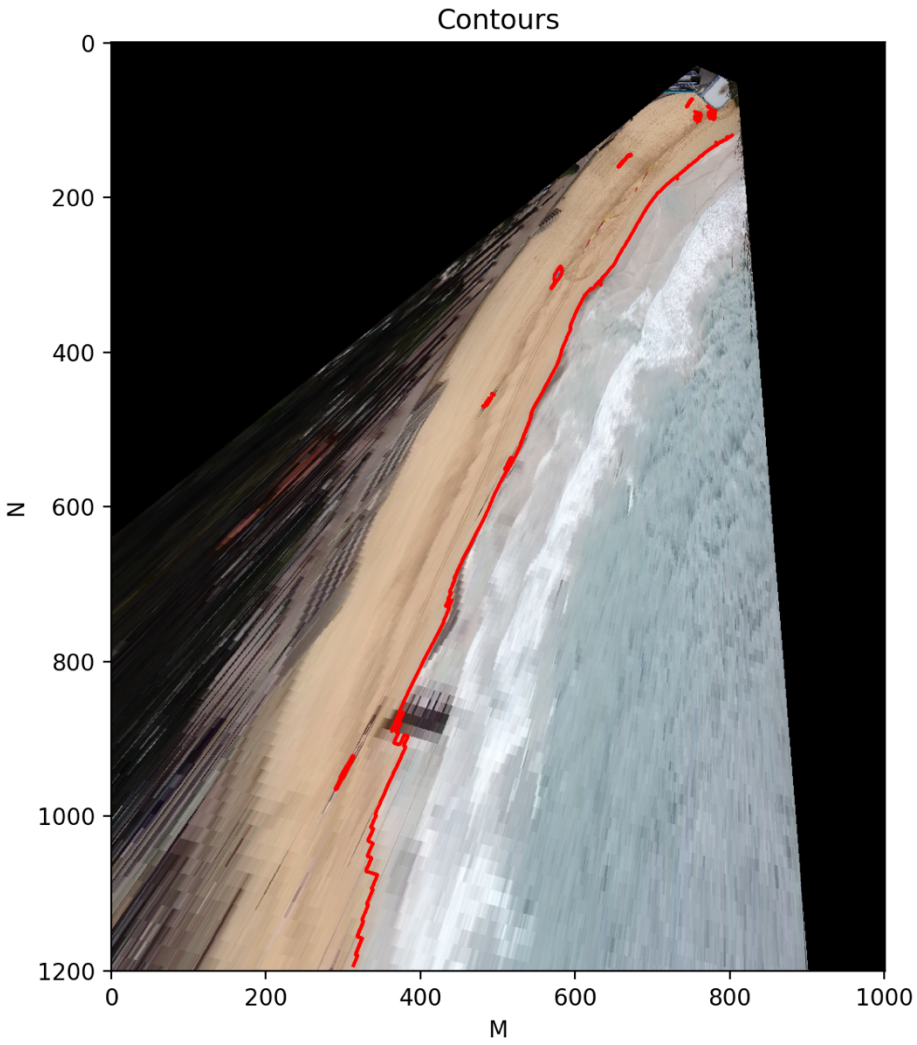


Figure 12: Contours of threshold

f. Determine the contour of the shoreline.

Finally, the shoreline is ascertained in CoastSnap by identifying the longest contour generated.

The shoreline locations generated by the Python and MATLAB functions should be analysed. Ideally these should be identical where the same threshold value is used.

Results and discussion

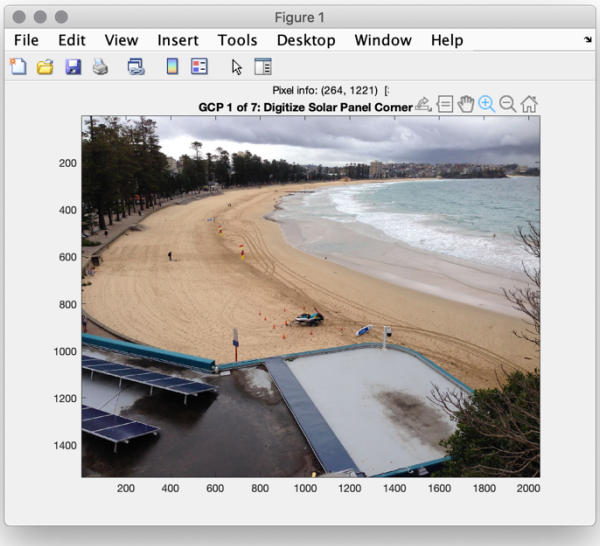
The following results and discussion section follows the operation of CoastSnap as presented in the methodology. The MATLAB code which enables each process has been presented together with the Python code recommended to achieve the equivalent functionality.

The MATLAB and Python code has been presented side by side. The code and its descriptions have been arranged following a consistent format demonstrated here:

MATLAB CODE	PYTHON CODE
`MATLAB function`	`Equivalent function in Python`

Listing 0 - 1

MATLAB Figure



Python Figure

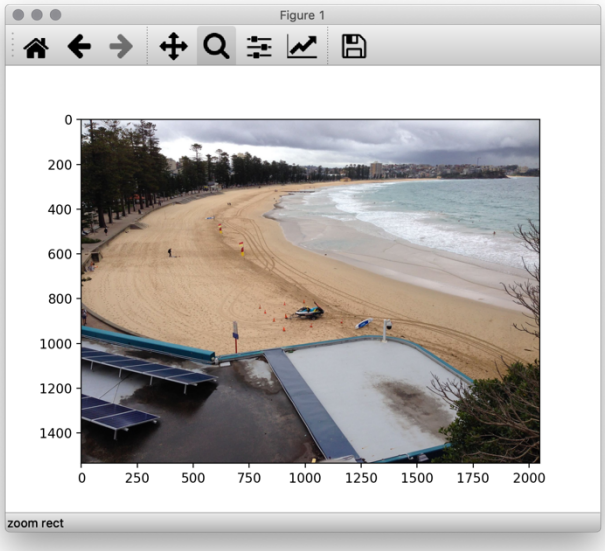


Figure 13

- ***Functions*** are in bold and italic font.
- **Variables** are in bold font.

Jupyter notebooks presenting both the MATLAB and Python code displayed in the following sections can be found on GitHub at the following address:

github.com/nicheaney/CoastSnap_Python_Jupyter_Notebooks

These are intended to clarify the material presented by enabling the code to be run simultaneously.

Image Georectification

The following section demonstrates the fundamental processes used in CoastSnap to georectify each image. The most important sections of the MATLAB code are followed and their roles explained. Beside this, the proposed Python code is presented.
Several initial steps concern the retrieval and organisation of variables to be used for the georectification process. These steps are important but are not foreseen to present major coding challenges. These are steps 1 – 4 below.

1. Get data about the selected image and the site from previous stages.
2. Check if the image has already been georectified.
3. Retrieve the GCP (ground control point) real world location information from the database.
4. Define the initial external camera parameters.
5. Prompt the user to interactively select the location of each GCP within the image:

Listing 1 - 1 presents the code used to provide users with an interface to identify the location of each ground control point (GCP) within the oblique image.

```
image(I)
for i = 1: nGcps
    title(['GCP ' num2str(i) ' of '
    num2str(nGcps) ': Digitize '
    gcp(i).name]));
    % Let the mouse around and see the
    values.
    hPixelInfo = impixelinfo();
    set(hPixelInfo, 'Unit',...
        'Normalized',...
        'Position', [.45 .96 .2 .1]);
    % you can zoom with your mouse and
    when your image is okay, you press
    any key
    zoom on;
    pause()
    zoom off;
    UV(i,:) = ginput(1);
    hold on
    plot(UV(i,1),UV(i,2),'go',...
        'markerfacecolor', 'g',...
        'markersize', 3);
    zoom out
end
```

```
%matplotlib qt
fig, axes = plt.subplots()
axes.imshow(I)
UV = np.zeros((nGcps,2))

for i in np.linspace(1, nGcps, nGcps,\
        dtype=int, endpoint=True):


    # zoom is enabled until any key is
    pressed
    fig.canvas.toolbar.zoom()
    while not \
        plt.waitforbuttonpress(): \
        pass
    fig.canvas.toolbar.zoom()

    A = plt.ginput(n=1, timeout=0)

    axes.plot(A[0][0], A[0][1],'go',\
         markersize = 3)
    UV[(i-1),:] = A[0]
```

Listing 1 - 1

For each GCP specified in the location's CoastSnap database, the user is required to zoom into the relevant location within the image (as shown in Figure 14) before

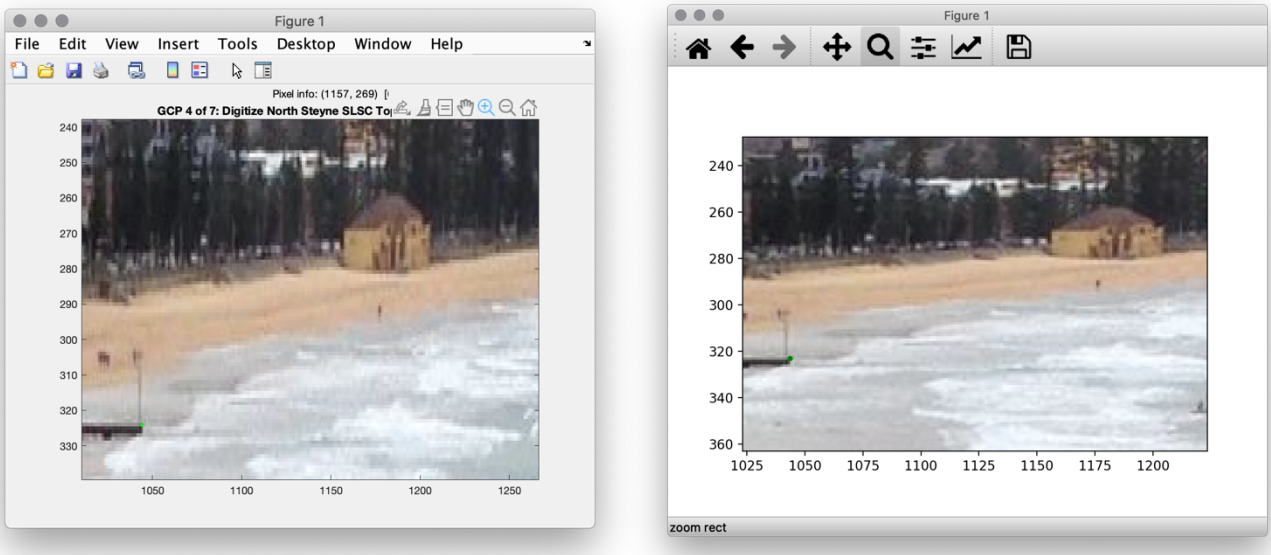


Figure 14: GCP location selection interface using zoom

pressing any key to initiate the following steps. This is achieved using MATLAB's *pause* and matplotlib's *plt.waitforbuttonpress* within Python.

Thereupon, a *ginput* (*plt.ginput* in Python) function is initiated. These functions enable the extraction of the U,V coordinates of a mouse click within an image. This procedure is carried out for each GCP via a for loop and the newly identified GCP location is plotted onto the image after each iteration as a green dot (as shown in Figure 15 e.g. at the solar panel corners).

Each iteration also assigns the coordinates of the identified GCP location to a new row in the variable **UV**. In the example used in figures 14 and 15 above there are 7 GCPs defined in the location's database and therefore the output variable **UV** is a 7 x 2 array, where each row contains the U and V (pixel) coordinates of the corresponding GCP.

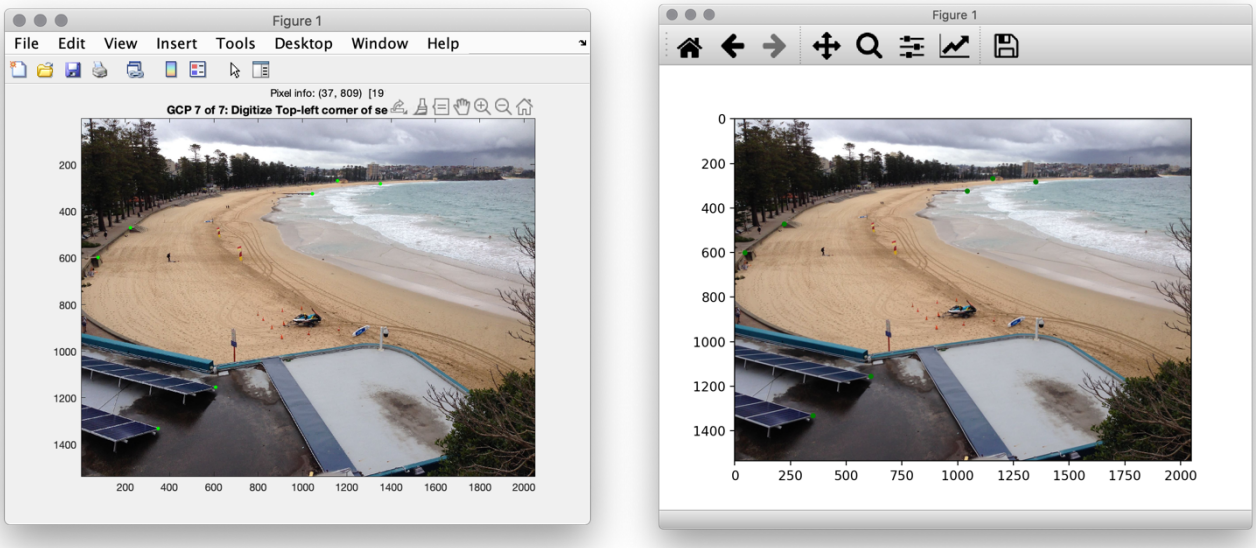


Figure 15: Selected GCP locations plotted onto interface

6. Establish the unknown camera parameters:

a. Create a range of focal length trial values (achieved using the code in Listing 1 – 2).

```
fx_max =0.5*Nu/tan(HFOV_min*pi/360)...
   %From Eq. 4 in Harley et al. (2019)
fx_min =0.5*Nu/tan(HFOV_max*pi/360)...
   %From Eq. 4 in Harley et al. (2019)
fx_min = interp1([5:5:500000],...
    [5:5:500000],fx min,'nearest');
fx_max = interp1([5:5:500000],...
    [5:5:500000],fx_max,'nearest');
fx = fx_min:5:fx_max;
```

```
fx_max = 0.5*Nu/np.tan\
    (HFOV_min*np.pi/360)
fx_min = 0.5*Nu/np.tan\
    (HFOV_max*np.pi/360)
A = np.arange(5, 500005, 5)
B = np.arange(5, 500005, 5)
fx_min = interpolate.interp1d\
    (A, B, kind='nearest')(fx_min)
fx_max = interpolate.interp1d\
    (A, B, kind='nearest')(fx_max)
fx = np.arange(fx_min, fx_max+5, 5)
```

Listing 1 - 2

The minimum and maximum horizontal field of view values (**HFOV_min** and **HFOV_max**) are extracted from the location's database. These are used with the equation relating the horizonal field of view, **HFOV** to the focal length, **f** (restated below, where **Nu** is the image width in pixels) to derive the corresponding minimum and maximum focal length values (**fx_min** and **fx_max**).

$$HFOV = 2tan^{-1}\frac{N_u}{2f}$$

Once determined, a range of values are defined between the minimum and maximum focal lengths.

To do so, **fx_max** and **fx_min** are first rounded to the nearest multiple of 5. This is achieved using MATLAB's *interp1* function and *interp1d* from SciPy's Interpolate library in Python. These functions are used to interpolate between points located at multiples of 5 using the nearest neighbour method. Essentially each interpolated point

is assigned the value of the closest multiple of 5. The interpolated value corresponding to **fx_max** and **fx_min** are deduced and set as the new **fx_max** and **fx_min** values.

The range of focal length values, **fx** is now defined beginning with the adjusted value for **fx_min**, increasing in increments of 5 until the adjusted value for **fx_max**. The range **fx** represents the possible values of the camera's focal length.

b. **Fit a non-linear regression between the user-identified U,V (pixel) coordinates and the measured x,y,z (real world) coordinates of the ground control points (GCPs).**

For each focal length trial value, a non-linear regression of the function *findUVnDOF* is undertaken using a least squares estimation to relate the known world coordinates (x,y,z) to the pixel coordinates (U,V) of the GCPs. The function *lcpBeta2P* applies the pinhole camera model and is called from within *findUVnDOF.* Three unknown camera parameters azimuth (α), tilt (**t**), and roll (**r**) are passed into *findUVnDOF*. Therefore, fitting a non-linear regression of *findUVnDOF* generates an estimate of these unknowns for the given focal length value.

For the purpose of this research, it was not suitable to convert *findUVnDOF* and each of the functions called from within it to Python. The main requirement in this section is to identify a suitable alternative function in Python to carry out the non-linear regression fit. However, to do so, a function similar to *findUVnDOF* is required to be passed as an argument into the function performing the non-linear regression fit. Therefore, a simplification of *findUVnDOF* has been created by extracting the required operations to apply the pinhole model. These are mainly from *lcpBeta2P*

(called from within *findUVnDOF*). This simplified function is named *TESTFITFUNC* and was able to be straightforwardly converted to Python.

```
function [UV] = ...
    TESTFITFUNC(beta, xyz)

load('finalnonlinearfittest')
%note: 'finalnonlinearfittest.mat
%contains the variables required to
%define the matrix K below



K = [fx 0 c0U;
    0 -fy c0V;
    0  0 1];

R = angles2R(beta(4),...
    beta(5), beta(6));




IC = [eye(3) -beta(1:3)'];

P = K*R*IC;
P = P/P(3,4);

UV = P*[xyz'; ones(1,size(xyz,1))];
UV = UV./repmat(UV(3,:),3,1);




UV = [UV(1,:) UV(2,:)]';
```

```
def TESTFITFUNC(xyz, beta0, beta1,\
     beta2, beta3, beta4, beta5):
    mat = scipy.io.loadmat\
        ('RectifyImagePython.mat')
    fx = mat['fx'][0,0].astype(float)
    fy = mat['fy'][0,0].astype(float)
    c0U = mat['c0U'][0,0]
    c0V = mat['c0V'][0,0]


    K = np.array([[fx, 0, c0U],\
        [0, -fy, c0V],\
        [0, 0, 1]]).astype(float)

    R = np.zeros((3, 3))
    angles2R(beta3, beta4, beta5)
    #angles2R definition in appendix
    I = np.eye(3)
    C = np.array([beta0, beta1,\
        beta2]).astype(float)
    C.shape = (3,1)
    IC = np.hstack((I,-C))

    P = np.matmul(np.matmul(K,R),IC)
    P = P/P[2,3]

    UV = np.matmul(P,np.vstack(\
        (np.transpose(xyz), np.ones(\
        (1, len(xyz)), \
        dtype = float))))
    UV = UV/np.matlib.repmat(\
        UV[2,:],3,1)

    UV = np.transpose(np.concatenate(\
        (UV[0,:], UV[1,:])))
    return UV
```

Listing 1 - 3

Essentially, *TESTFITFUNC* first defines the matrices K, R, and [I | - C] (defined as IC in Listing 1 – 3) for the pinhole camera model calculation which is thereafter undertaken.

The matrices are defined using:

- Global variables (predefined and loaded at the beginning of the function)
- principle points (defined as **c0U** and **c0V** in listing 1-3)
- focal lengths (**fx** and **fy**)

Function arguments

- represented by **beta** in MATLAB and **beta0** : **beta5** in Python which are:
- **beta** = [camera x, camera y, camera z, azimuth, tilt, roll]
- **beta0** = camera x, **beta1** = camera y, **beta2** = camera z,
- **beta3** = azimuth, **beta4** = tilt, **beta5** = roll.

Note: The reason that the unknowns in the Python code are represented by individual variables rather than a single array (as is the case in the MATLAB code) is due to the requirements of the regression fitting function (scipy.optimize.curve_fit) in Python. This function requires the unknown parameters that it will fit to be separate arguments into *TESTFITFUNC*.

Now that *TESTFITFUNC* has been defined in both MATLAB and Python, they can be passed into the non-linear regression fitting functions. CoastSnap uses MALAB's *nlinfit* and the most suitable alternative in Python has been found to be *curve_fit* from SciPy's optimise library.

The arguments for both *nlinfit* and *curve_fit* are:

The ground control point (GCP) coordinates

- **xyz** = the (x,y,z) real world coordinates as a nGCPs x 3 array
- **UV** = the (U,V) image pixel coordinates as selected by the user. The elements of **UV** are rearranged to provide an argument in the form of a 2nGCPs x 1

array (i.e. in a single column with all of the U coordinates first before all of the V coordinates).

The model function and an initial estimate of this function's parameters

- *TESTFITFUNC* is the model function which will be used to fit a regression relating **xyz** and **UV**.
- **beta0** is an array containing preliminary estimations of the 6 arguments of *TESTFITFUNC* (camera's x, y and z coordinates, azimuth, tilt and roll).

```
beta = nlinfit(xyz, [UV(:,1);...
    UV(:,2)], 'TESTFITFUNC', beta0)
```

```
beta, Cov = curve_fit(TESTFITFUNC\
    ,xyz, np.concatenate((UV[:,0],\
    UV[:,1])),beta0)
```

Listing 1 - 4

nlinfit and *curve_fit* return the array beta which has the same arrangement as the argument **beta0**, however these parameters have now been optimised and represent the arguments of *TESTFITFUNC* which generate the non-linear regression relating the real world coordinates (**xyz**) to the pixel coordinates (**UV**).

When testing the performance of curve_fit in comparison to nlinfit, it was found that the two functions return the same parameter values:

```
beta =

   -0.1718   0.1558   17.1404...
   -0.4058    1.3028   -0.0056
```

```
beta = array([-1.71807884e-01,\
               1.55786045e-01,\
               1.71404018e+01,\
              -4.05849281e-01,\
               1.30282771e+00,\
              -5.59067456e-03])
```

Listing 1 - 5

Note: The camera's x, y and z coordinates are known, however they are re-estimated by the non-linear regression fitting functions since they are arguments of the model function being fitted (*TESTFITFUNC*). In CoastSnap, these re-estimated values are ignored. The non-linear regression fit is only used to resolve the three unknowns: azimuth, tilt and roll.

c. **Determine the mean squared error of the regression fit for each possible value of the focal length.**

The fourth unknown, focal length, is resolved by generating a non-linear regression for each member in the range of possible focal lengths (**fx**). The member resulting in the regression with the minimum mean squared error is taken as the camera's focal length. Azimuth, tilt and roll are taken as their optimised values in this same regression.

Using MATLAB, CoastSnap is able to receive a mean squared error directly from *nlinfit*. Using Python, *curve_fit* does not have the option to return a mean squared error value. This is done instead using *mean_squared_error* from Scikit-Learn's Metrics module.

```
[beta, ~, ~, ~, mse, ~] = nlinfit...
    (xyz, [UV(:,1); UV(:,2)],...
    'TESTFITFUNC', beta0)
```

```
UV_pred = TESTFITFUNC(xyz, beta[0],\
    beta[1], beta[2], beta[3],\
    beta[4], beta[5])
mse = mean_squared_error(UV_true,\
    UV pred)
```

Listing 1 - 6

Listing 1 – 6 demonstrates how the mean squared error value is attained. In Python, the optimised **beta** values are passed back into *TESTFITFUNC* in order to return the

UV array predicted by the non-linear regression, UV_pred. This is required to be passed into *mean_squared_error* along with the original, user identified UV array, now UV_true. The mean squared error between UV_pred and UV_true is then returned. However the mean squared error returned by *nlinfit* is different to that returned by *mean_squared_error:*

```
mse =
    10.0105
```

```
mse = 5.720282577441515
```

Listing 1 - 7

Listing 1 – 7 presents the mean squared errors returned for the test variables used to produce the beta values in Listing 1 – 5. The reason for the variation between the outputs is the method used to calculate the mean squared error:

- *nlinfit*'s method takes into account the number of unknown parameters in the non-linear regression fit (represented by nBetas). The squared error is divided by the number of observations (the combined number of individual U and V coordinates, i.e. 2 x the number of ground control points) minus the number of unknown parameters (nBetas):

$$mse = \frac{(error)^2}{(2 \times nGCPs) - nBetas}$$

- *mean_squared_error* operates independent from the non-linear regression in Python and therefore is unable to take the influence of unknown parameters into account. Instead *mean_squared_error* uses the universal method of dividing the error by the number of observations:

$$mse = \frac{(error)^2}{(2 \times nGCPs)}$$

The mean squared error returned from *mean_squared_error* can thus be converted to the mean squared error returned from *nlinfit* through the following operation:

$$mse_{nlinfit} = mse_{mean\_squared\_error} \times \frac{(2 \times nGCPs)}{(2 \times nGCPs) - nBetas}$$

This can be demonstrated using the test from Listing 1 – 7 in which the number of GCPs was 7 and the number of unknown parameters was 6:

$$mse_{nlinfit} = 5.7202 \times \frac{(2 \times 7)}{(2 \times 7) - 6} = 10.0105$$

The operation to calculate the mean squared error in Python can therefore be adapted to that shown in Listing 1 – 8. This matches the output from MATLAB.

```
mse = mean_squared_error(UV_true,\
    UV_pred)*((2*nGcps)/ \
    ((2*nGcps)- beta0.shape[1]))
```

Listing 1 - 8

Here it is necessary to note the consequences of adjusting the Python mean squared error (mse) values in this way for the following operations in CoastSnap:

- **Identifying the regression fit with the minimum mse and thus the optimal focal length.** In this step, the mse values from each iteration of regression fitting only need to be compared to identify the minimum value. Since the number of GCPs and unknown parameters remains constant for each iteration, *nlinfit* and

mean_squared_error return values in proportion to each other and thus, the minimum mse returned from both will correspond to the same focal length. Adjusting the mse in Python is therefore inconsequential for identifying the optimal focal length.

- Checking the root mean squared error (rmse) is suitable to continue. Before saving the results from the georectification process, CoastSnap first checks the rmse is less than 10. If not, the user is prompted to begin the process again. The rmse is generated from the mse value and therefore by not adjusting the mse, Python will generate an rmse lower than those generated using the CoastSnap MATLAB code. The rmse threshold of 10 has been established for the MATLAB code and therefore if the threshold of 10 is adopted by the Python code while the mse remains unadjusted, georectifications with greater errors will be able to proceed. Therefore adjusting the mse in Python is important to identify where the rmse is too high to continue. In this case, an alternative to adjusting every mse is to adjust the rmse threshold of 10 to a greater value. However this threshold would be dependent on the number of GCPs being used, requiring recalibration at different sites.

7. Plot the GCP locations generated by the resolved camera parameters onto the image.

Now that all camera parameters required for the pinhole camera model have been resolved, the U,V coordinates generated by the model (UV_computed: red circles in Fig. 16) can be determined using the known x,y,z coordinates. These can then be compared with the U,V coordinates determined by the user (UV: green points in Fig. 16) in a plot of the image.

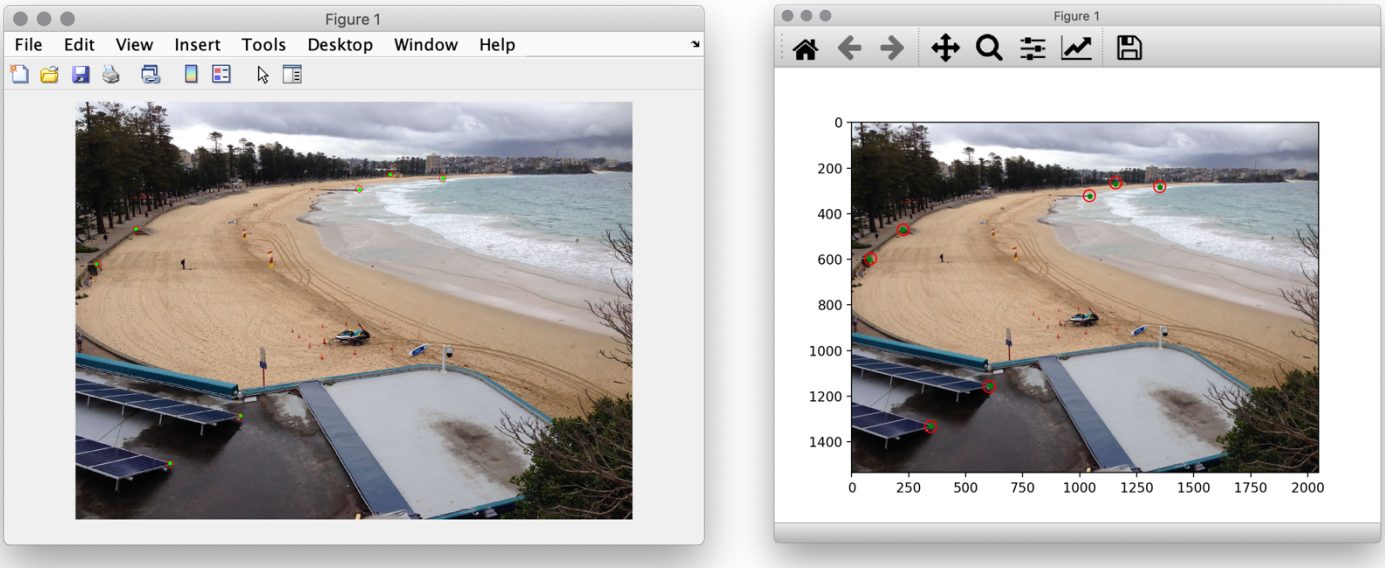


Figure 16: GCP locations using the pinhole camera model compared to those identified by the user

The code required to return **UV_computed** is presented in the MATLAB section of Listing 1 – 9. This requires passing the resolved camera parameters back into ***findUVnDOF*** with the known world (x,y,z) coordinates. As ***findUVnDOF*** has not been converted to Python, this operation has been left out of the Python section in Listing 1 – 9. Instead the **UV_computed** values returned in MATLAB have been loaded in Python in order to demonstrate the plot shown in Fig. 16.

```
UV_computed = findUVnDOF...
    (betas(1,:), xyz, globs);
UV_computed = reshape(UV_computed,...
    [],2);
imshow(I)
hold('on')
plot(UV(:,1), UV(:,2), 'go',...
    'markerfacecolor', 'g',...
    'markersize', 3)
plot(UV_computed(:,1),...
    UV_computed(:,2),'ro');
```

```
UV_computed = mat['UV_computed']\
    .reshape((-1,2))
%matplotlib qt
fig, axes = plt.subplots()
axes.imshow(I)
axes.plot(UV[:,0], UV[:,1],'go',\
     markersize = 3)
axes.scatter(UV_computed[:,0],\
    UV_computed[:,1], s=80,\
    facecolors='none',edgecolors='r')
```

Listing 1 - 9

Note: In Python, *scatter* from Matplotlib's Pyplot library has been used to plot the points represented by UV_computed. This is to allow the points to be plotted as hollow circles.

8. Create the plan image.

The final step of the georectification process is to create the plan image. Code to carry out this process exists in Python in another coastal imaging software, SurfRCaT (Conlin et al., 2020). Few additional steps were required to correctly format the data generated in CoastSnap for it to be passed into the SurfRCaT code. However, despite this minimal additional programming effort, the runtime for the program to generate a plan image was significantly longer than that in CoastSnap's MATLAB code. The runtime for the computer used in testing was always greater than 400 seconds. To give a comparison, for the same image, CoastSnap's MATLAB code had an average runtime of 0.494 seconds while the Python program had a runtime of 407.120 seconds. The code adapted from SurfRCaT can be found in Appendix II.

This runtime has been considered unfeasible for CoastSnap here due to the requirement to process large numbers of images and for a person to be present for this operation.

Further analysis and modifications of the adapted SurfRCaT code could be undertaken to improve its runtime. An initial investigation, identified the function *griddata* from SciPy's Interpolate library (used to interpolate data between grid points) as the cause of the significant runtime. A survey of forums discussing *griddata* identified a general experience of long runtimes (e.g. (Shahar, 2020)). However a suitable solution to adapt the code could not be found.

CoastSnap's method to generate the plan image does not interpolate data between grid points and therefore *griddata* is not required in a Python conversion of the MATLAB code. It has been decided, therefore, to continue to convert the CoastSnap code directly from MATLAB. This is important so that CoastSnap can run in a feasible time in Python. The process is demonstrated below.

The code in Listing 1 – 10 generates a grid representing the real-world space in the x-y plane that the plan image will cover. This is done using x and y coordinate limits and a specified grid resolution, all of which are predetermined by the user in the CoastSnap database. This information is extracted at this stage and held in the variable **xy** which takes the following form:

$$\boldsymbol{xy} = \begin{bmatrix} Xmin & Xresolution & Xmax \\ Ymin & Yresolution & Ymax \end{bmatrix}$$

With this information the 1D arrays, **x** and **y** are created, containing the x and y axis coordinates. To generate a 2D grid from **x** and **y**, *meshgrid* (MATLAB) and Numpy's *np.meshgrid* (Python) have been used. The same outputs are returned by both: **X** and **Y** which contain the x and y coordinates for every point within the grid.

```
x = [xy(1):xy(2): xy(3)];

y = [xy(4):xy(5): xy(6)];


[X,Y] = meshgrid(x,y);
```

```
#Arrays to be passed to the meshgrid
#function
x = np.arange(xy[0,0], \
    (xy[0,2] + xy[0,1]), xy[0,1])
y = np.arange(xy[0,3], \
    (xy[0,5] + xy[0,4]), xy[0,4])
#Meshgrid generated
X, Y = np.meshgrid(x, y)
```

Listing 1 - 10

X and Y are next used to create an array, **xyz,** representing the (x,y,z) coordinates of every point in the grid. **xyz** takes the form (*n* is the total number of grid points):

$$xyz = \begin{bmatrix} x_{point1} & y_{point1} & z_{point1} \\ x_{point2} & y_{point2} & z_{point2} \\ \vdots & \vdots & \vdots \\ x_{pointn} & y_{pointn} & z_{pointn} \end{bmatrix}$$

To create this (as is done in Listing 1 - 11), X and Y must be reshaped to 1 column in column major (column by column) order. i.e:

$$X = \begin{bmatrix} x_{point1} & \cdots & \vdots \\ x_{point2} & \ddots & x_{pointn-1} \\ \vdots & \cdots & x_{pointn} \end{bmatrix} \rightarrow xyz_{column1} = \begin{bmatrix} x_{point1} \\ x_{point2} \\ x_{point3} \\ \vdots \\ x_{pointn} \end{bmatrix}$$

Reshaping in this order is done by default in MATLAB using X(:). To achive the same using numpy in Python requires X.flatten('F'), where F is critical to indicate column major order.

```
xyz = [X(:) Y(:) repmat...
    (z, size(X(:)))];
```

```
#Create an array representing the
#(x,y,z) coordinates of every point
#in the grid. (nPoints, 3)
xyz = np.column_stack\
    ((X.flatten('F'), Y.flatten('F')\
    , np.matlib.repmat\
    (z, len(X.flatten('F')), 1)))
```

Listing 1 - 11

Note: the z coordinate for all grid points represented in **xyz** is constant. It is the water level at the shore line determined by the *tide level + tide offset (dependent on wave*

height, period and the beach face slope). The values used to calculate z are extracted from the database. To generate the column of constant z values for **xyz**, ***repmat*** (MATLAB) and ***np.matlib.repmat*** from Numpy's Matlib library (Python) are used, specifying a length equal to the other columns.

In Listing 1 – 12 the (U,V) coordinates corresponding to each point in the grid are generated. This is done by passing **xyz** to ***findUVnDOF***. The returned coordinates in **UV** are rounded to the nearest integer using ***round*** (MATLAB) and Numpy's ***np.around*** (Python). This is necessary as the pixel information of the oblique image requires integer indices to be accessed.

Note: in Python ***TESTFITFUNC*** is still used as an alternative to ***findUVnDOF*** here. It does not affect the demonstration and testing of the processes.

```
UV = round(findUVnDOF...
    (beta,xyz,glob));
```

```
#Create an array of (U,V) coords
#(rounded to nearest integer)
#corresponding to each grid point
UV = np.around(TESTFITFUNC(xyz,\
    beta[0,0], beta[0,1], beta[0,2],\
    beta[0,3], beta[0,4], beta[0,5]))
```

Listing 1 - 12

UV contains coordinates corresponding to the entire real-word grid defined, however, the oblique image only covers a region of the coordinates in **UV**. To determine the rows in **UV** which can be supplied data from the oblique image, a set of conditional statements (specifying the oblique image limits) are passed to ***find*** (MATLAB) and Numpy's ***np.where*** in Listing 1 – 13.

```
yesNo = zeros(size(U,1),1);
on = find((U>=Umin) & (U<=Umax)...
    & (V>=Vmin) & (V<=Vmax));
yesNo(on) = ones(size(on));
```

```
#yesNo has a 1 designated to UV
#coords which exist in the oblique
#image (ie they have image data)
yesNo = np.zeros((len(U),1))
on = np.where((U>=Umin) & (U<=Umax) \
    & (V>=Vmin) & (V<=Vmax)) [0]
yesNo[on] = 1
```

Listing 1 - 13

After identifying the rows in UV which can extract data from the oblique image, the coordinates represented in these rows are converted to linear indices corresponding to pixels in the oblique image. The process described is illustrated in Fig 17 (adapted from (The MathWorks Inc., 2021a)).

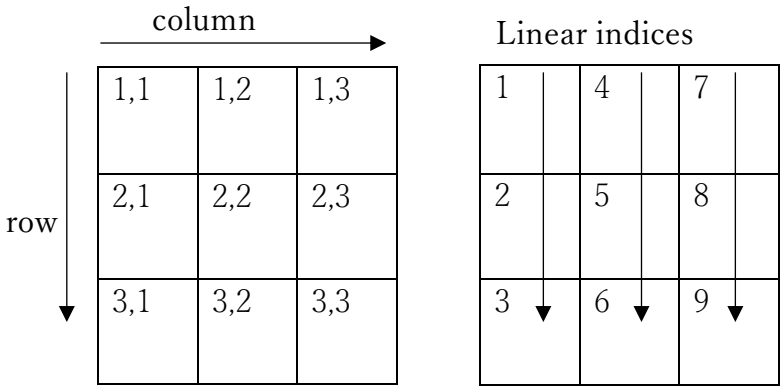


Figure 17

In Listing 1 – 14, the functions which enable the index conversion are ***sub2ind*** (MATLAB) and Numpy's ***np.ravel_multi_index*** (Python). Specifying *order = 'F'* in ***np.ravel_multi_index*** is again critical to ensure the linear indices returned are in column-major (column by column) order.

```
good = find(yesNo);
ind = sub2ind([NV NU],UV(good,2),...
    UV(good,1));
```

```
#returns the linear indices of the
#usful coordinates
ind = np.ravel_multi_index(arr, \
    (NV, NU), mode='clip', order='F')
```

Listing 1 - 14

Using the linear index format, the grid points which correspond to pixels in the oblique image are assigned this pixel data through a for loop. The grid pixel data is contained in a 3D array, **finalImages_timex,** of shape (grid y-axis length, grid x-axis length, 3) to hold each of the red greed and blue pixel values. This can then be plotted using the code in Listing 1 – 15.

```
imagesc(x, y, finalImages.timex);
```

```
#The bounds of the image plot are set
#To the predefined x and y values
#using the extent parameter
plt.imshow(finalImages_timex, \
    extent = (xy[0,0], xy[0,2],\
     xy[0,5], xy[0,3]))
```

Listing 1 - 15

The grid axes **x** and **y** are passed to ***imagesc*** in MATLAB, to directly define the image axes. In order to do the same in Python, using Matplotlib's ***plt.imshow*** the image axes' extents must be specified as an optional parameter using ***extent =***. Figure 18 displays the plan images produced.

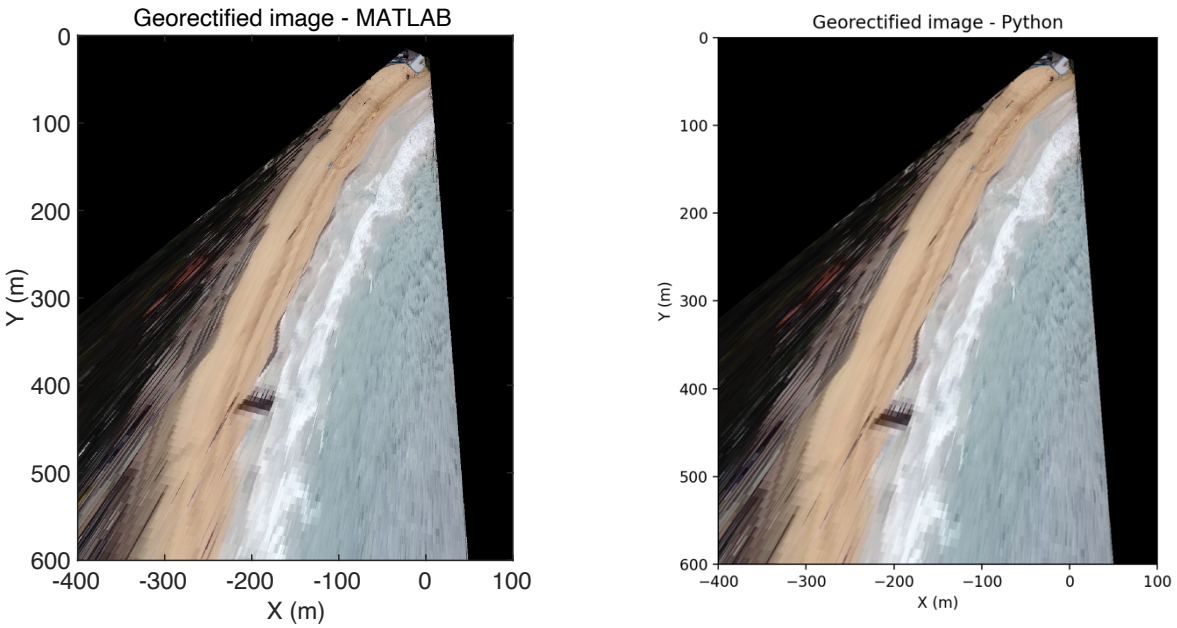


Figure 18: Plan images

After completing the conversion of the CoastSnap code from MATLAB to Python for the plan image creation, the runtime was again tested. The Python code required an average runtime of 0.789 seconds. Proportionally, this is still significantly higher than the 0.494 seconds of the original MATLAB code, however, in absolute terms, an increase of 0.295 seconds will cause a minimal impact on the user experience. When comparing the 0.789 seconds to the 407.120 seconds required to run the code adapted from SurfRCaT, the converted code is significantly advantageous.

Map Shoreline

The following section demonstrates the fundamental processes used in CoastSnap to map the shoreline in the georectified image. The most important sections of the MATLAB code are followed and their roles explained. Beside this, the proposed Python code is presented. Initial steps concern the retrieval and organisation of variables from previous steps. These steps are important but are not foreseen to present major coding challenges. These are steps 1 – 3 below.

1. Get the plan image data.
2. Check if the shoreline has already been mapped.
3. Load the transect data from the Shorelines file.
4. Call the function *mapShorelineCCD* to determine the shoreline location
    a. Extract the red, green and blue (RGB) pixel values for the points along each transect.

CoastSnap samples the RGB pixel data existing throughout the region of the beach in the plan image. The samples are taken along transects which have been predetermined for the CoastSnap site. The MATLAB function used to achieve this is *improfile* which enables the world coordinate system to be used by passing the x and y grid axis coordinates contained in **xgrid** and **ygrid**. By defining the coordinate system for *improfile* in this way, the transect start and end coordinates can be passed to the function as they are (world coordinates). The chosen alternative to *improfile* in Python is *profile_line* from the Measure module of Scikit-Image. *profile_line* does not enable the world coordinate system to be defined for the image data. Therefore, the transect start and endpoints must be passed to *profile_line* as indices corresponding to their location in the array representing the image. Figure 19, below, illustrates the different method of defining transect start and end points for *improfile* and *profile_line*. One example of a transect line has been displayed in red.

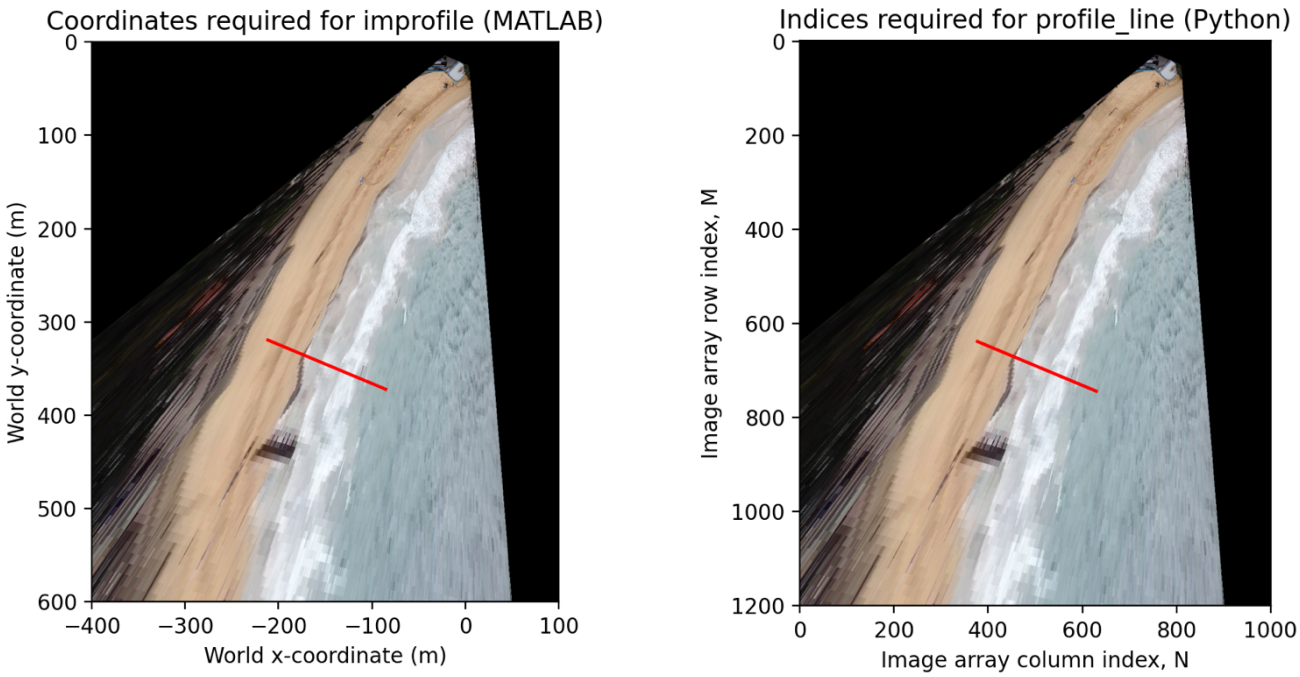


Figure 19: Different transect locating systems

Thus, the transect start and end locations are passed to ***improfile*** in the form of (x,y) world coordinates, while the same locations are passed to ***profile_line*** in the form of (M,N) image array indices. The transect files provide the start and end locations in world coordinates (x,y) and can therefore be passed directly to ***improfile***, however, before they are passed to ***profile_line***, they must be converted to image array indices (M,N) through the following transformation:

$$M = y * 2$$

$$N = (x + 400) * 2$$

To check that this transformation method is valid, it has been tested with ***improfile*** in MATLAB. In this test, the world grid axis coordinates were not passed to ***improfile*** so that the transect coordinates required to be passed matched the indices required for ***profile_line***. ***Improfile*** returned the same RGB values in both tests (one when the world coordinates were used and the other when the array indices were used) confirming that the conversion used to generate image array indices from world coordinates is valid.

The final step to generate the M and N indices corresponding to the correct location is to subtract 1 from M and N. This is due to Python indexing beginning from 0, while

indices begin from 1 in MATLAB. Therefore to locate the same point, M and N in Python must be 1 less than those in MATLAB. The transformation from (x,y) to (M,N) thus becomes:

$$M = (y * 2) - 1$$

$$N = \big((x + 400) * 2\big) - 1$$

Listing 2 – 1 presents the code used to extract the RGB image values. The conversion of the transect world coordinates to M and N can be observed in the Python code.

```
P = improfile(xgrid,ygrid,Iplan,...
    transects.x,transects.y);
```

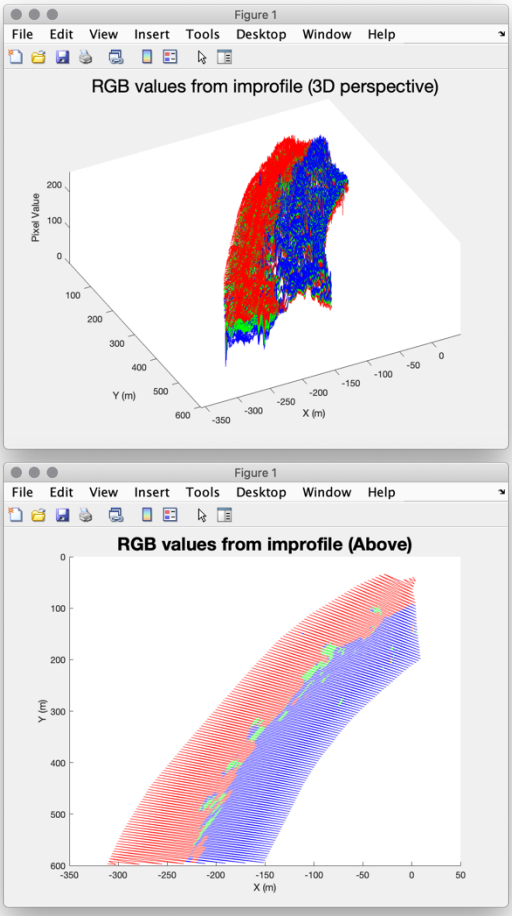


Figure 20: Graph returned from improfile from different perspectives

```
P = np.empty((0,3))
for i in np.arange(1,transectsX.shape\
    [1]+1,1):
    M1 = transectsY[0,i-1]
    M1 = (M1*2)-1
    M2 = transectsY[1,i-1]
    M2 = (M2*2)-1
    N1 = transectsX[0,i-1]
    N1 = ((N1 + 400) * 2)-1
    N2 = transectsX[1,i-1]
    N2 = ((N2 + 400) * 2)-1
    prof = profile_line(Iplan, \
        (M1, N1), (M2, N2))
    P = np.append(P, prof, axis = 0)
    if i == transectsX.shape[1]:
        break
    else:
        M1 = transectsY[1,i-1]
        M1 = (M1*2)-1
        M2 = transectsY[0,i]
        M2 = (M2*2)-1
        N1 = transectsX[1,i-1]
        N1 = ((N1 + 400) * 2)-1
        N2 = transectsX[0,i]
        N2 = ((N2 + 400) * 2)-1
        prof = profile_line(Iplan, \
            (M1, N1), (M2, N2))
        P = np.append(P, prof, \
            axis = 0)
```

Listing 2 - 1

improfile enables the entire transect system of the beach to be sampled in a single call. However, ***profile_line*** only allows one transect to be sampled per call. A for loop is therefore applied for the number of transects present.

Note: during each iteration of the for loop in Python, ***profile_line*** is applied twice (the second application is activated with an if loop). The first application samples along the transect defined. The second application samples along the line between the end of the transect and the start of the next transect. This essentially creates a zig-zag pattern of samples as shown in fig. 21. The reason for doing this is that ***improfile*** operates in this way in CoastSnap in MATLAB and has therefore been mimicked in the Python code applying ***profile_line*** in order to compare the results generated in each case.

The if loop is required to initiate the second ***profile_line*** application as the second application is not possible during the last iteration since there is no further transect to link to. To avoid an error, the if statement breaks the loop during the final iteration.

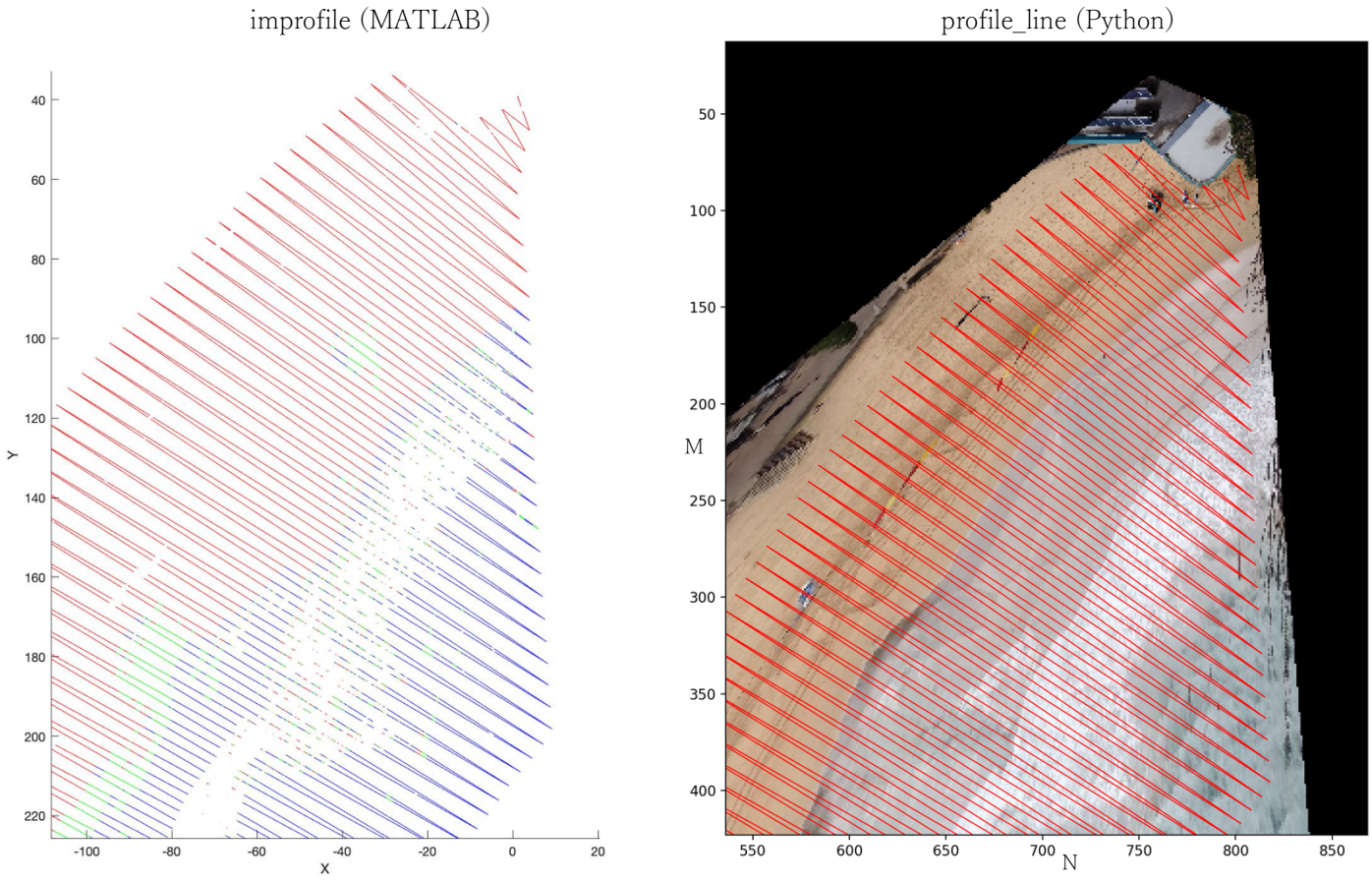


Figure 21: zig-zag pattern of lines sampling RGB values

improfile and the Python code using *profile_line* return very similar, however, not identical outputs. When testing using identical plan images and transect data, *improfile* returned 60736 sample points (each containing a red, green and blue pixel value), extracted from the transect system. Meanwhile the Python code using *profile_line* returned 67596 sample points. The Python code therefore extracts 11.3% more sample points that the MATLAB code which should not disrupt the operation of the Python code in comparison to the original provided the extra samples are evenly distributed throughout the beach area. The pixel data of the sample points is allocated to the variable **P**. Analysing the RGB pixel values in **P** directly (the first 10 rows are listed in Listing 2 – 2), it can be observed that similar values are returned in both cases.

```
P(1:10,:) =

   204   174   146
   199   173   140
   198   172   139
   198   172   139
   198   171   141
   198   171   142
   195   168   141
   194   166   142
   193   165   143
   202   174   153
```

```
P[0:10,:]=

       array([[202., 172., 146.],
              [204., 174., 146.],
              [201., 173., 143.],
              [199., 173., 140.],
              [198., 172., 139.],
              [198., 171., 140.],
              [198., 171., 141.],
              [198., 171., 142.],
              [196., 169., 142.],
              [194., 166., 142.]])
```

Listing 2 - 2

However, since the Python and MATLAB code return a different number of samples, it is not appropriate to compare the results row for row. A better method for comparing the results in this case, is to compare the graphs generated by each set of data to analyse how well their shapes match. This enables the ability to check that the same lines are being sampled, regardless of the number of samples extracted (to an extent). To demonstrate a clear comparison of the shapes, one transect has been isolated for analysis. In figure 22 the red pixel data from this transect are plotted. It can be seen that the graphs from both MATLAB and Python have similar shapes, however, there is a noticeable constant shift of the Python returned data towards the

positive x-axis. At first observation, the probable cause of this is inconsistently defined transect coordinates between MATLAB and Python. This was checked by revising the transect location conversion from real world coordinates to image array indices.

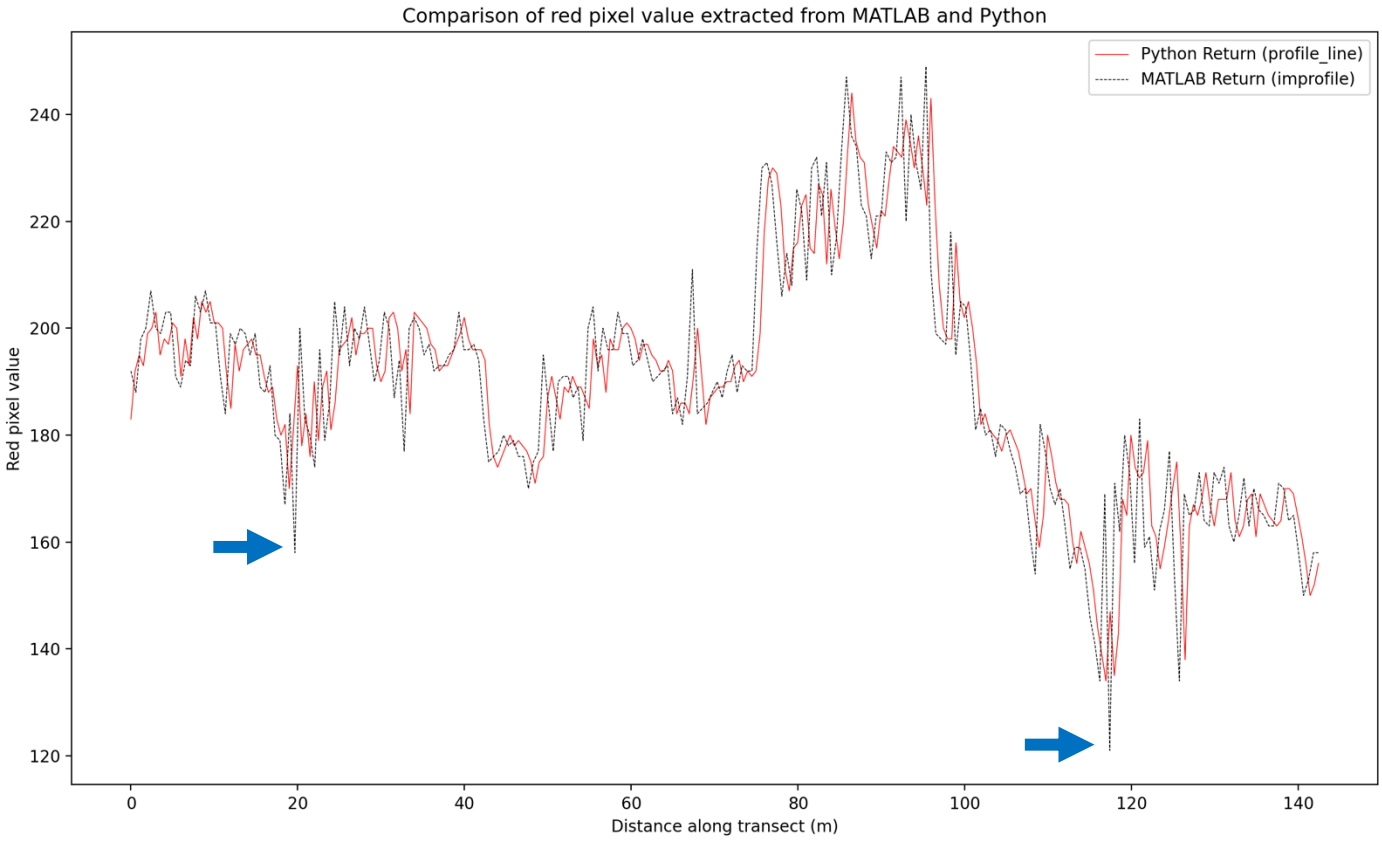


Figure 22: Comparison of red pixel values

Where the translation from (x,y) to (M,N) had previously been taken as:

$$M = (y * 2) - 1$$
$$N = \big((x + 400) * 2\big) - 1$$

This was revised to:

$$M = (y * 2)$$
$$N = \big((x + 400) * 2\big)$$

Using the revised transect location translation, the constant shift observed in Fig. 22 has been eliminated in Fig. 23. Figure 23 also demonstrates an improved similarity of the shape due to the revised transect location translation. This is most noticeable in the locations indicated by blue arrows in Fig. 22, where the Python code previously did not return these extreme values seen in MATLAB. The extreme values are now

generated by the Python code too. Due to the significant improvement in result similarity generated by this revision, various other possible revisions were tested. However, these all caused a deterioration to the result similarity.

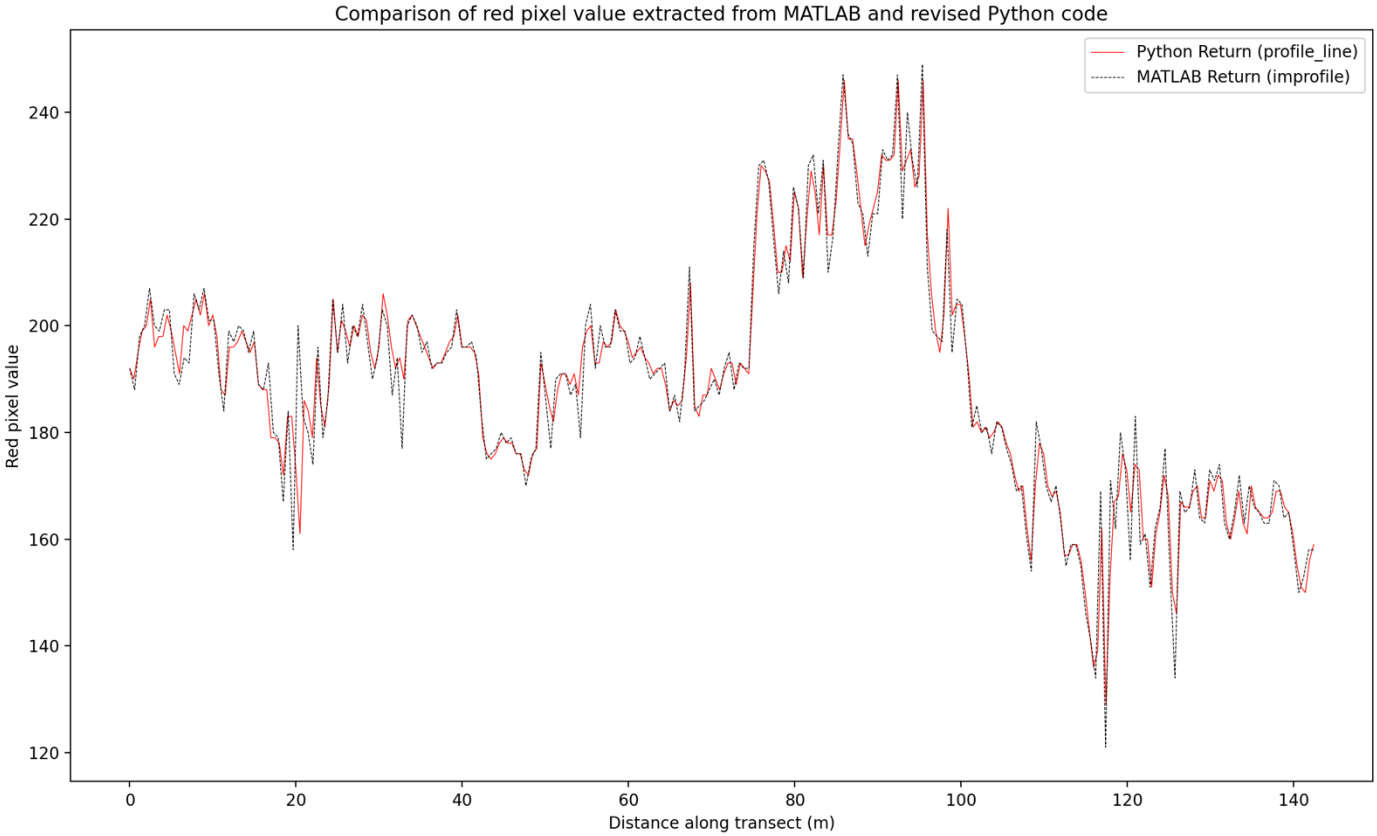


Figure 23: Comparison of red pixel values using revised transect location translation

The MATLAB and Python sample data returned to P is not identical, however this cannot be straightforwardly overcome, as neither *improfile* nor *profile_line* enable the number of sample points to be specified. Both data sets produced cover the same range of pixel values and at a similar resolution, despite the greater number of points collected using Python (for this isolated transect, 240 points were sampled by MATLAB and 286 points by Python). Furthermore, by understanding the way that the graphs in Fig. 23 have been plotted, it has been deduced that the increased number of points sampled in Python are evenly distributed along the transect. This is because the MATLAB plot is automatically generated with *improfile* associating the pixel values to a distance along the transect (ensuring an accurate graph shape). In Python, however, the graph must be generated manually and the pixel data is not assigned a

corresponding distance to generate a plot with. Therefore Numpy's *np.linspace* was used to generate an equally spaced range of distance values from 0 to the transect length (142.40m) with the same number of elements as pixel samples. The Python samples were then plotted against these distances. By using this method to plot the Python data, if the extra data points were not evenly distributed across the transect and concentrated in certain regions, the Python graph's shape would not match that of the MATLAB graph. The extra points must therefore be evenly distributed along the transect as the graph shapes match well. This means that the Python code will not be impaired by the extra sample points it generates, as they are not skewed towards any direction.

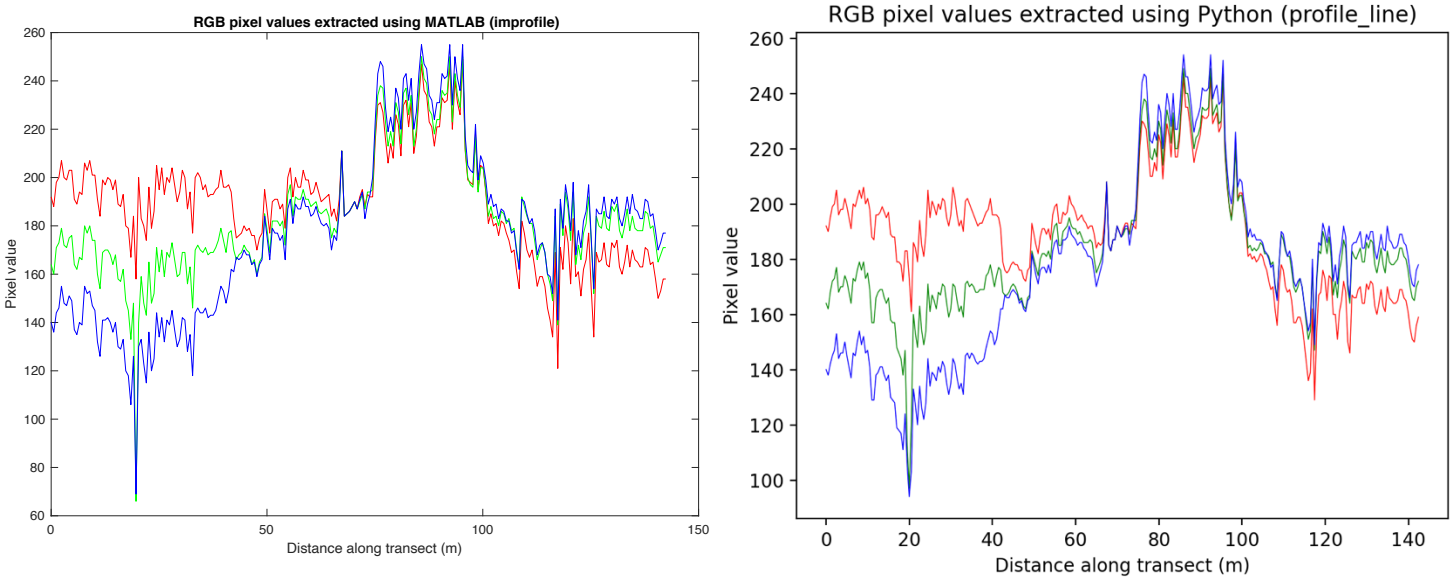


Figure 24: Comparison of RGB data extracted from improfile and profile_line

The similarity of the data is further illustrated by Fig. 24, displaying each of the red, green and blue value graphs for the isolated transect using both MATLAB and Python. Whilst it cannot be certified that exactly the same transect start and end locations are registered by *improfile* and *profile_line*, the shapes generated by the graphs of the pixel values indicate that the samples are being extracted from the same (or very close) paths. The transects are defined at intervals along the beach, in order to extract a

general sample of beach image pixel values and it is therefore not critical that exactly the same pixels are extracted. Furthermore, if the image is georectified differently (e.g. by inconsistently locating ground control points) different pixels will exist along the transect locations. Equally, initially defining transects in different positions will result in different RGB pixel data being extracted.

Another issue which must be considered for the Python code using *profile_line* is its runtime. Due to the multiple operations of *profile_line* during the for loop, a considerable runtime is possible and must be investigated. For the test image used in this section, an average runtime of 3.738 seconds was observed. By comparison, the runtime required for *improfile* in MATLAB was 3.280 seconds. An increase of 0.458 seconds is small relative to the overall time of *improfile*, however at over 3 seconds this process is already timely so this extension in time is undesirable.

b. Create a probability distribution estimate for the range of red minus blue values of the sampled pixels.

After extracting each of the red, green and blue pixel values from the sampled points, CoastSnap uses the red and blue values for its operation hereon in. The blue value is subtracted from the red value to from the red minus blue value (RmB). An RmB value is generated from each sample point in P and is represented by the code in Listing 2 – 3 where the third column in P holding the blue pixel values is subtracted from the first column in P holding the red pixel values.

```
RmB = P(:,:,1)-P(:,:,3)
```

```
RmB = P[:,0]-P[:,2]
```

Listing 2 - 3

Note: P is a nx1x3 matrix in MATLAB (3 dimensions) but an nx3 array in Python (2 dimensions) where n is the number of points sampled.

The next step is to generate a probability distribution describing the probability of the occurrence of RmB values. This is done using a probability density function (PDF) to determine the relationship. To generate the PDF, the functions ***ksdensity*** in MATLAB and ***gaussian_kde*** from SciPy's Stats module are used following the code presented in listing 2 – 4.

```
[pdf_values,pdf_locs] = ksdensity...
    (P(:,:,1)-P(:,:,3));
```

```
RmBsample = P[: , 0] - P[: , 2]
kde = stats.gaussian_kde(RmBsample)
pdf_locs = np.linspace(RmBsample.min\
    (), RmBsample.max(), 100,\
     endpoint=True)
pdf_values = kde(pdf_locs)
```

Listing 2 - 4

In MATLAB, ***ksdensity*** directly returns estimated values of the PDF (in the variable **pdf_values**) and the corresponding 100 equally spaced RmB values at which these estimates were taken (in the variable **pdf_locs**). **pdf_locs** covers the range of RmB values passed to ***ksdensity***. In contrast to ***ksdensity, gaussian_kde*** in Python does not return estimated values of the PDF at particular locations. Instead an estimation of the PDF itself is returned (represented by ***kde*** in Listing 2 – 4). In Python, **pdf_locs** and **pdf_values** must be generated manually thereafter. To be consistent with MATLAB **pdf_locs** is defined in the Python code as an array containing 100 equally spaced points between the minimum and maximum RmB values using ***np.linspace***. These can now be passed to the newly defined ***kde*** to generate the estimated values of the PDF (**pdf_values**) corresponding to the RmB values in **pdf_locs**. To understand **pdf_locs** and **pdf_values** it is more effective to visualise them using the plot in Fig. 25.

Note: To generate the probability distributions in Fig. 25 the RmB data generated by MATLAB and Python independently was used for their respective plots.

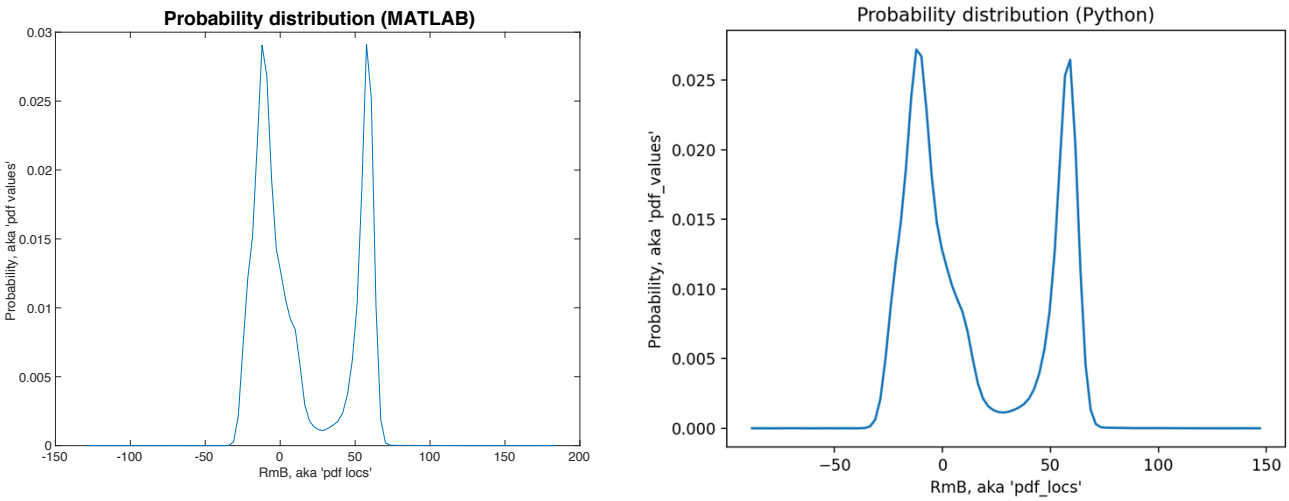


Figure 25: Probability distributions

The plots generated are similar, however, there are some noticeable differences, for example, the shapes of each peak.

c. Determine the threshold RmB value (the value corresponding to the pixels along the shoreline).

The RmB probability distribution can now be used to determine the threshold value representing the shoreline. First the threshold weightings (which can be adapted to each individual beach) are defined. These weightings are used to determine the location of the optimum RmB value (thresh) between the two peaks of the probability distribution. The RmB values at the peaks are defined as **RmBwet** for the left peak and **RmBdry** for the right peak. This is because the left peak corresponds to pixels with greater blue values and lower red values (occurring in the region of the image covering water) while the right peak corresponds to pixels with greater red values and lower blue values (occurring in the region of the image covering sand). On the downloading of CoastSnap the weightings are set at 1/3 for **RmBwet** and 2/3 for **RmBdry**. That is, the threshold, **thresh** = 1/3 **RmBwet** + 2/3 **RmBdry**. These are the default threshold weightings because they are the most suitable for South-East Australia where CoastSnap was created (Harley *et al.*, 2019). They can be altered in regions where sand and ocean colours are different to those in Australia.

```
thresh_weightings = [1/3 2/3];

thresh_otsu = multithresh...
    (P(:,:,1)-P(:,:,3))

I1 = find(pdf_locs<thresh_otsu);
[~,J1] = max(pdf_values(I1));

I2 = find(pdf_locs>thresh_otsu);
[~,J2] = max(pdf_values(I2));

RmBwet = pdf_locs(I1(J1))
RmBdry = pdf_locs(I2(J2))

thresh =thresh_weightings(1)*RmBwet...
     + thresh_weightings(2)*RmBdry
```

```
thresh_weightings = [1/3, 2/3]

thresh_otsu =threshold_otsu(RmBsample)


I1 = np.argwhere(pdf_locs<thresh_otsu)
J1 = np.argmax(pdf_values[I1])

I2 = np.argwhere(pdf_locs>thresh_otsu)
J2 = np.argmax(pdf_values[I2])

RmBwet = pdf_locs[I1[J1,0]]
RmBdry = pdf_locs[I2[J2,0]]

thresh = thresh_weightings[0]*RmBwet \
    + thresh_weightings[1]*RmBdry
```

Listing 2 - 5

Listing 2 – 5 presents the code used to calculate the optimum RmB, **thresh**. To find the peak RmB values, **RmBwet** and **RmBdry**, all RmB values are classified into two groups, 'wet' or 'dry', using Otsu's thresholding method. Otsu's method identifies a threshold below which the RmB values are classified as 'wet' in **I1** and above which the RmB values are classified as 'dry'. This is achieved using ***multithresh*** in MATLAB and ***threshold_otsu*** from Scikit-Image's Filters module in Python. The peak in the 'wet' RmB region represents **RmBwet** and the peak in the 'dry' RmB region represents **RmBdry** and both can be found using the ***max*** (MATLAB) or Numpy's ***np.argmax*** in Python. With the peak values identified, the threshold weightings are applied in order to determine **thresh** (the optimum RmB value corresponding to the pixels on the shoreline).

Figure 26 presents each of the variables, **RmBwet**, **RmBdry**, **thresh_otsu** required for the determination of **thresh**. The values generated for each variable displayed are recorded in Table 1.

Note: The RmB data generated independently by MATLAB and Python was again used to produce the respective plots in Fig.26. This has been done to compare the **thresh** generated by the entire code in Python.

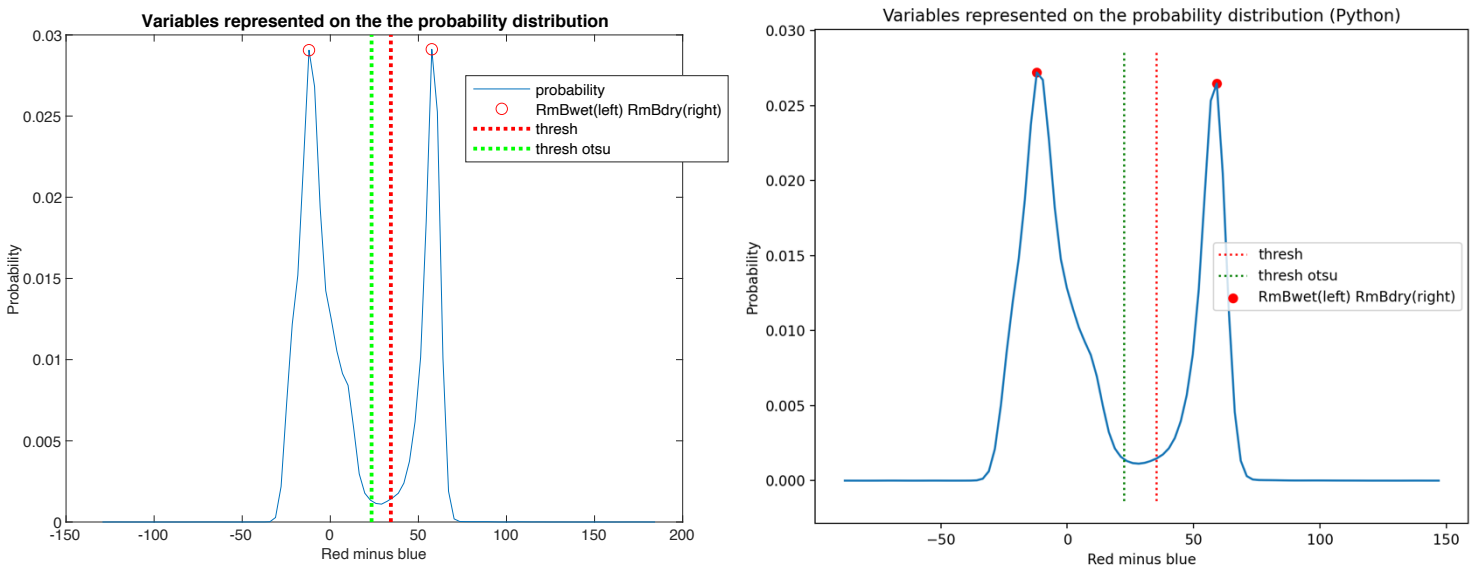


Figure 26: Variables represented within the probability distribution

Table 1: Variable values generated while determining threshold

	Value from MATLAB code	Value from Python code
RmBwet	-12.1031	-12.0404
RmBdry	57.5984	59.1717
thresh_otsu	23.4235	22.6152
thresh	34.3645	35.4343

From Table 1 it is visible that the thresh value returned from the Python code varies from that returned from the MATLAB code and hence, the functions are not operating in the same way. However, this was already expected due to the variation in the sampling points to extract RGB data. To only compare the performance between the functions used to determine **thresh** in MATLAB and Python, the **RmB** data generated in MATLAB has been transferred and run using the Python code to see if the same values are returned. Table 2 presents the **thresh** values returned from doing so.

Table 2: Thresh value generated using the same RmB data sample

	Value from MATLAB code	Value from Python code
thresh	34.3645	36.0000

It is observed that using the MATLAB generated **RmB** data returns a threshold value further from **thresh** than the original test.

Further investigation has been made into the operation of *ksdensity* and *gausian_kde* which generate the probability distribution data. It was found that the 100 equally spaced RmB values at which the probability is estimated (**pdf_locs**) were different in the MATLAB and Python operations. The Python code had been set to generate the 100 equally spaced values starting at the minimum RmB value and ending at the maximum RmB value. Meanwhile, by analysing the MATLAB output for **pdf_locs**, it could be seen that these 100 equally spaced points begin and end at values outside of the maximum and minimum RmB values of the sample data (how the start and end points have been determined in MATLAB is not clear and will be discussed further). To account for this discrepancy between the MATLAB and Python code, the Python code was altered so that the start and end values of **pdf_locs** were equal to that in MATLAB. Doing so resulted in the **thresh** values presented in Table 3.

Table 3: Thresh values returned from the updated Python code

	Value from MATLAB code	Value from Python code
thresh	34.36453958631399	34.36453958631399

Exactly the same **thresh** is now being returned from the MATLAB and Python functions. However, while it was possible to align the Python and MATLAB code for this specific test, a general alignment of the code for all images being tested has not been possible. This is because the determination of the **pdf_locs** start and end values in MATLAB cannot be ascertained. Table 4 presents the minimum and maximum RmB values obtained from the sampled pixels in 6 test images. The start and end values of **pdf_locs** from MATLAB are presented for these same tests to analyse if there is an inferable connection to the minimum and maximum RmB values.

*Table 4: **pdf_locs** start and end values generated by MATLAB in different tests*

test	pdf_locs start	RmB min	RmB max	Pdf_locs end
1	-129.3283	-121	176	184.3283
2	-84.0951	-69	103	118.0951
3	-122.4925	-110	10	22.4925
4	-125.3283	-117	119	127.3283
5	-83.6156	-68	101	116.6156
6	-91.8619	-70	153	174.8619

A general rule which determines the **pdf_locs** start and end values in MATLAB was not inferred and no information is provided in MATLAB's documentation (The MathWorks Inc., 2021b). The Python code is not able to be aligned to the MATLAB code as a result.

It should be noted, however, that the reason for varying **thresh** values when **pdf_locs** is defined differently in MATLAB and Python is due to the fact that **RmBwet** and **RmBdry** must take a value from the **pdf_locs** range. **pdf_locs** is a range of discrete values and therefore if they are not identical in MATLAB and Python, it is impossible to for the same **RmBdry, RmBwet,** and thus **thresh** to be returned by both codes. Furthermore, regardless of the programming language used, if the **pdf_locs** values used do not correspond well to the true peaks of the probability distribution, there will be an error in the returned **RmBwet** and **RmBdry** values.

To reduce this error, it is beneficial for the points at which the probability density function is evaluated to occur more frequently (i.e. **pdf_locs** should contain a larger number of equally spaced RmB values). With this higher resolution, the identified **RmBwet** and **RmBdry** will be closer to the true location of the peaks. To test the effect

of this, the shoreline threshold value, **thresh** has been recalculated several times in Python and MATLAB with **pdf_locs** comprising a different number of equally spaced points each time. The results are presented in Table 5.

Table 5: Thresholds returned when using different numbers of evaluation points

Length of **pdf_locs**	**thresh** (MATLAB)	**thresh** (Python)	Python runtime (s)
100	34.3645	36.0000	0.2025
200	35.1181	35.2111	0.3243
300	35.3676	35.6120	0.4370
400	35.2300	35.3158	0.5621
500	35.3571	35.1382	0.6776
600	35.4418	35.3506	0.7935
700	35.0535	35.2189	0.9239
800	35.1549	35.3679	1.0400

The results in Table 5 show that the values of **thresh** returned converge to values between 35.0000 and 35.5000 as the number of evaluation points (in **pdf_locs**) is increased. This reveals that the original MATLAB and Python returned **thresh** values of 34.3645 and 36.0000 contain a significant error compared to tests using more points in **pdf_locs**. It is therefore suggested that the Python code should define **pdf_locs** variable comprising of more points in order to improve the accuracy of the shoreline threshold value, **thesh**.

The length of **pdf_locs** suggested here is 400. This is has been decided by compromising the requirements of sufficient convergence and minimal runtime. Based on the tests in Table 5, the **thresh** values have converged significantly for a **pdf_locs** length of 400 and the runtime is tolerable at 0.5621 seconds. For comparison, the original MATLAB and Python runtimes were 0.1955 and 0.2025 seconds respectively.

d. Extract the contours in the image with the threshold value corresponding to the shoreline.

The shoreline threshold value is an RmB value and therefore the RmB value of every pixel within the image must be calculated to determine which of them represent the shoreline. Listing 2 – 6 demonstrates the definition of **RminusBdouble**, the variable which contains each of these RmB values.

```
RminusBdouble = double(Iplan(:,:,1))-
double(Iplan(:,:,3));
```

```
RminusBdouble = Iplan[:,:,0] \
    - Iplan[:,:,2]
```

Listing 2 - 6

The plots of **RminusBdouble** in Fig. 27 display the variation of RmB values throughout the plan images (reproduced in Fig. 28).

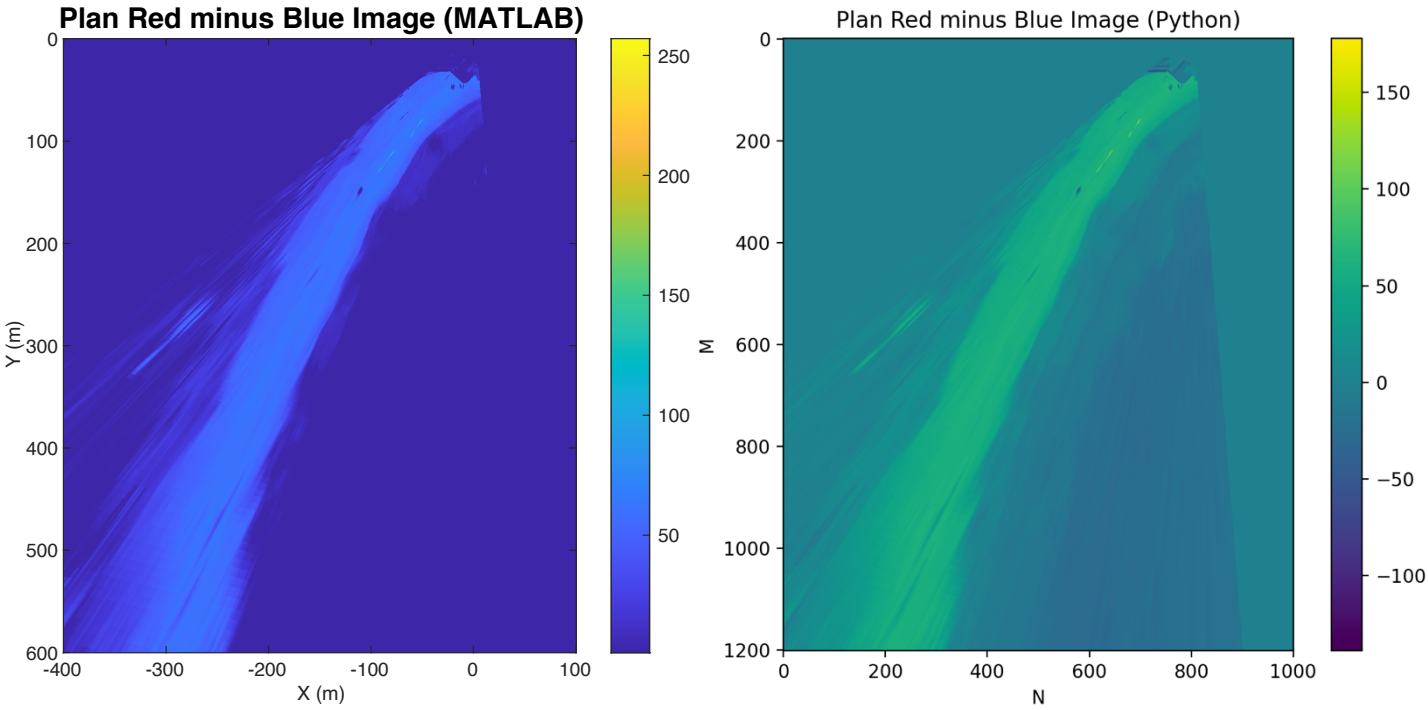


Figure 27: Red minus Blue plotted for each pixel in the plan image

Note: the MATLAB plot does not show the negative RmB values as the image plotting function used accepts only positive pixel values. The negative values are still defined within **RminusBdouble**.

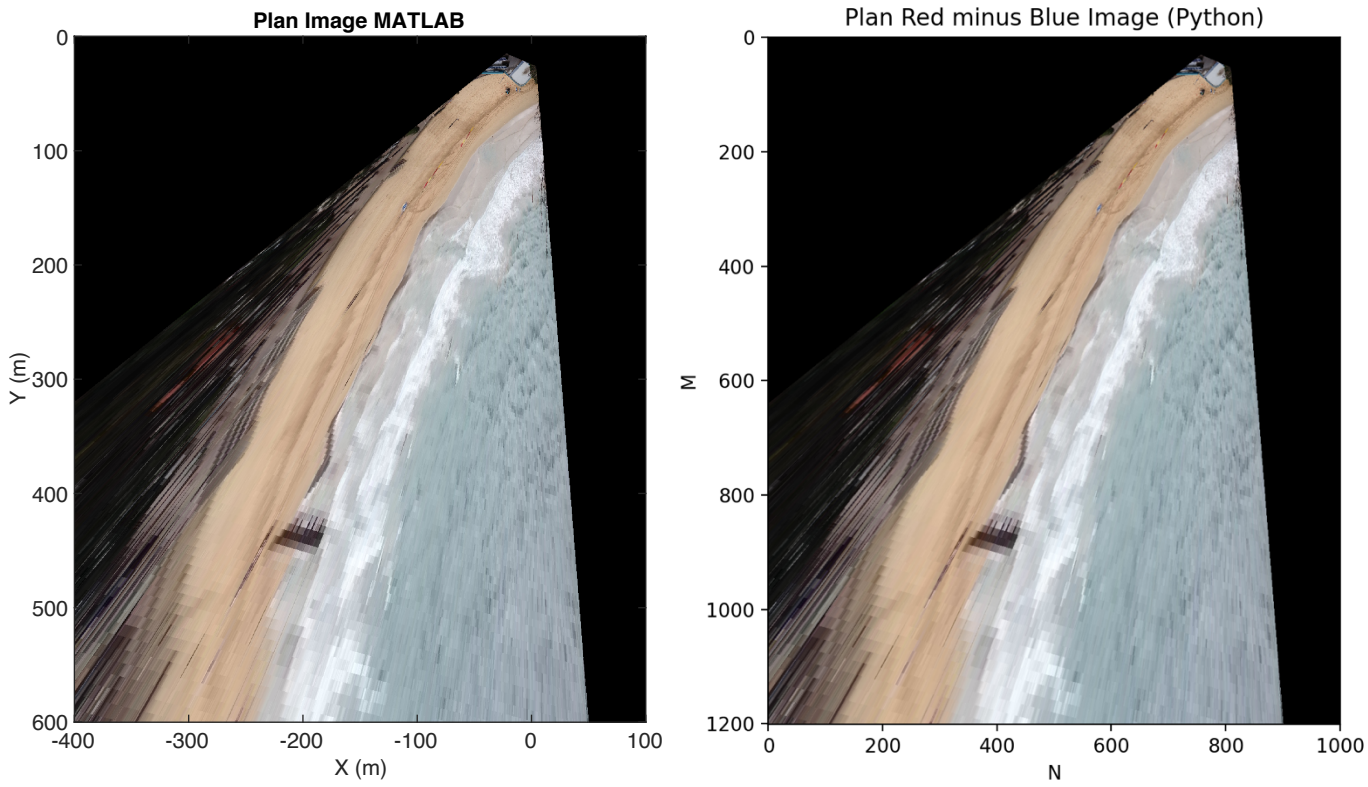


Figure 28: Plan images from which RminusBdouble has been generated

The area to be searched for the threshold contour is only that covered by the transects. This region of interest is bounded by the start and endpoints of the transects and the x and y coordinates of these points are appointed to the variables **ROIx** and **ROIy** respectively. As is shown in Listing 2 – 7.

```
ROIx = [transects.x(1,:) ...
    fliplr(transects.x(2,:))];
ROIy = [transects.y(1,:) ...
    fliplr(transects.y(2,:))];
```

```
ROIx =np.concatenate((transectsX[0,:]\
    , np.flipud(transectsX[1,:])))
ROIy =np.concatenate((transectsY[0,:]\
    , np.flipud(transectsY[1,:])))
```

Listing 2 - 7

Note: the functions ***fliplr*** (MATLAB) and Numpy's ***np.flipud*** (Python) are used to arrange the x and y coordinates in **ROIx** and **ROIy** in a clockwise order. This is a requirement for them to be passed into the region defining functions ***inpoly*** (MATLAB) and ***points_in_poly*** from Scikit-Image's Measure module (Python). The bounding points represented in **ROIx** and **ROIy** are plotted in Fig.29.

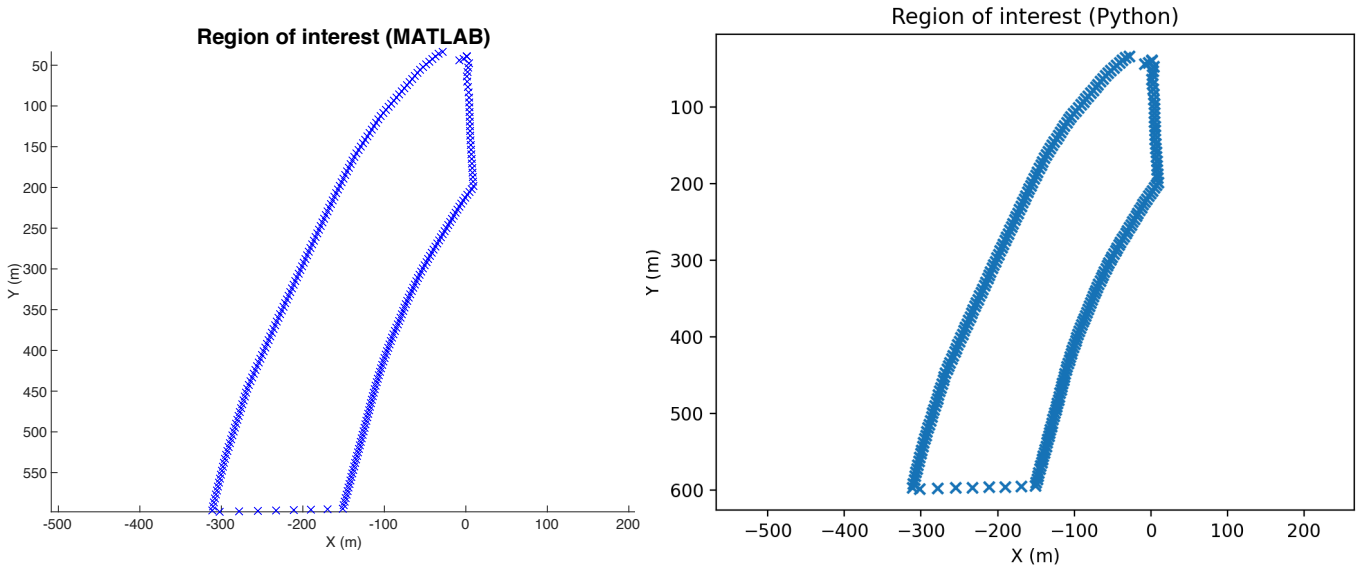


Figure 29: Transect end points bounding the region of interest (ROI)

To isolate the region of interest ***inpoly*** and ***points_in_poly*** are used to determine which grid points of the RmB image are inside the area bounded by the points shown in Fig. 29. The grid coordinates in (x,y) format are passed to ***inpoly*** and ***points_in_poly*** along with the points bounding the region (from **ROIx** and **ROIy**). The grid points determined to be outside the ROI are then assigned NaN values. Listing 2 – 8 presents the code used for this procedure.

```
#Determine the points outside the ROI
Imask = ~inpoly([X(:) Y(:)],...
    [ROIx',ROIy']);

#Mask points outside ROI in
#RminusBdouble
RminusBdouble(Imask) = NaN;
```

```
#format the plan image grid points
XFlat = X.flatten()
YFlat = Y.flatten()
points = np.column_stack((XFlat,\
    YFlat))
#format the ROI vertice coordinates
verts = np.column_stack((ROIx, ROIy))
#Determine the points outside the ROI
Imask = ~points_in_poly(points, verts)
Imask = np.reshape(Imask,[X.shape[0],\
    X.shape[1]])
#Mask points outside ROI in
#RminusBdouble
RminusBdouble[Imask] = np.nan
```

Listing 2 - 8

The adapted **RminusBdouble** variables are exactly the same in MATLAB and Python. This is presented visually in Fig. 30.

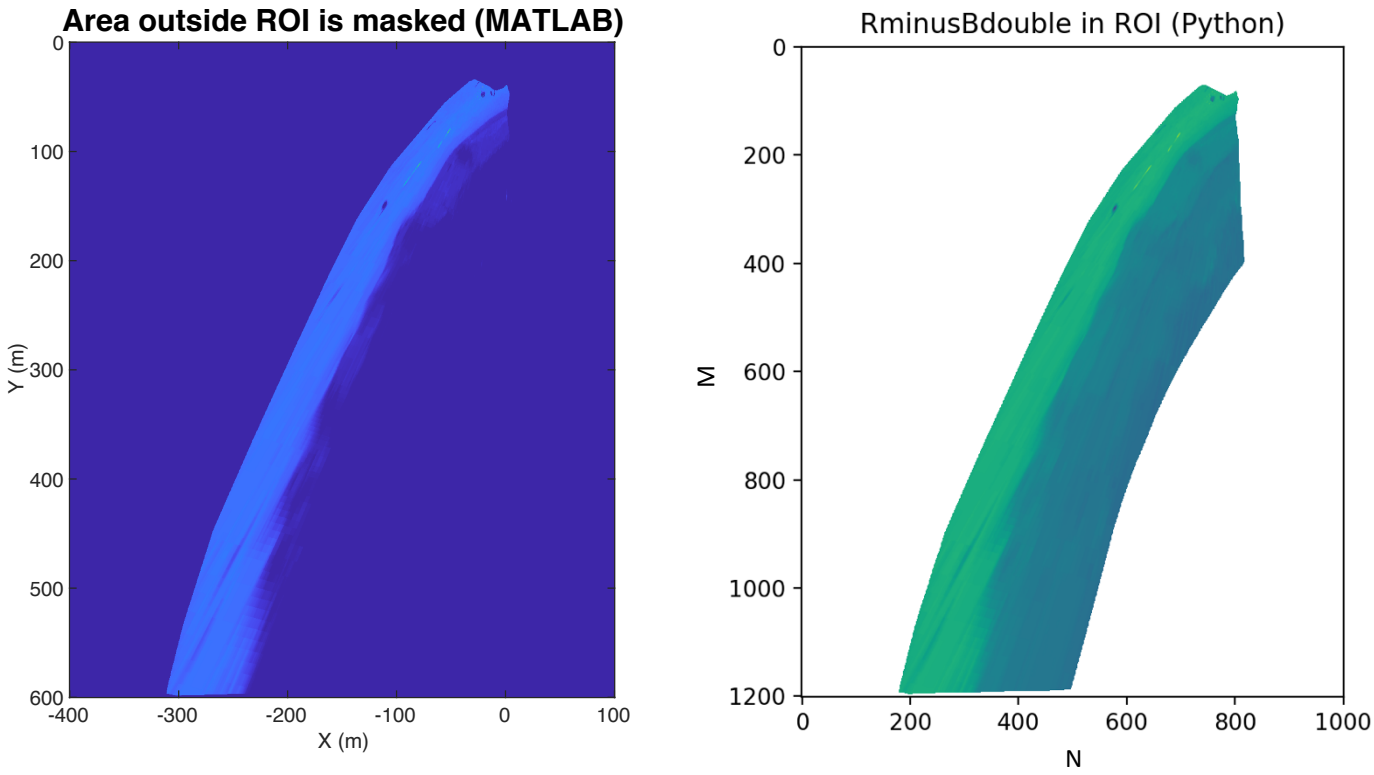


Figure 30: RminusBdouble with points outside ROI masked

To locate the contours of neighbouring points with the threshold value (**thresh**), the functions *contours* (MATLAB) and *find_contours* from Scikit-Image's Measure module (Python) are called using code in Listing 2 - 9. The contours generated are plotted in Fig. 31.

```
c = contours(X,Y,RminusBdouble,...
    [thresh thresh]);
```

```
c =find_contours(RminusBdouble,thresh)
```

Listing 2 - 9

MATLAB's *contours* function enables the x and y coordinate system of the values inside RminusBdouble to be passed as an argument. As a result, the contours are returned in world (x,y) coordinates. Contrastingly, the contours returned by *find_contours* in MATLAB are in array indices.

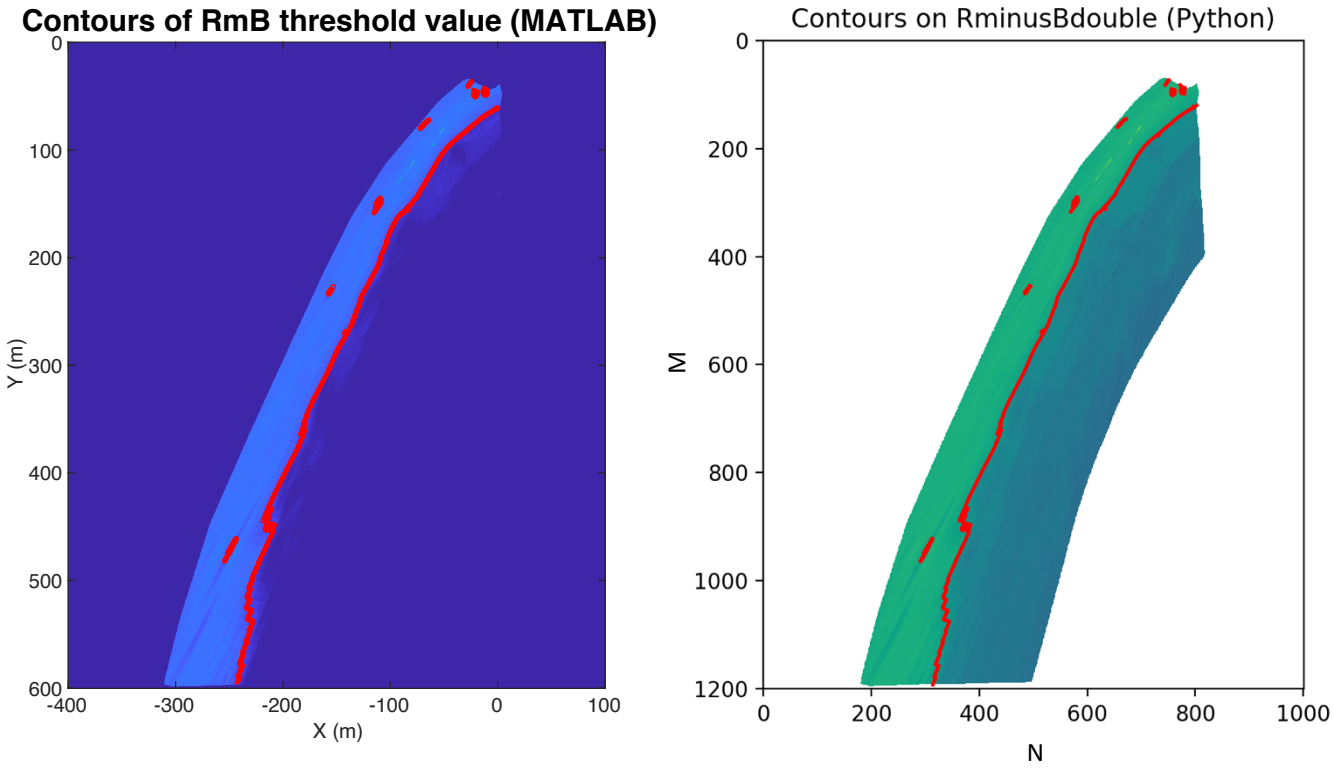


Figure 31: Contours plotted on the RminusBdouble image

e. Determine the contour representing the shoreline.

Finally, the shoreline position can be located by identifying which of the contours generated corresponds to the shoreline. CoastSnap does this by identifying the longest contour. Figure 32 displays the longest contours generated in both MATLAB and Python (i.e. the shoreline). The locations of the shorelines are found to be identical when the same threshold (**thresh**) value is passed into *contour* in MATLAB and *find_contours* in Python. To establish that the same locations were returned, the (M,N) indices returned by *find_contours* were converted to their corresponding world coordinates. Listing 2 – 10 presents a selection (the 3 first and last) points of the shoreline contour points. The world coordinates obtained after converting the Python indices are presented too for comparison with the MATLAB output. **Note:** the original Python output contour NM, is in the form [column index, row index] (opposite to convention).

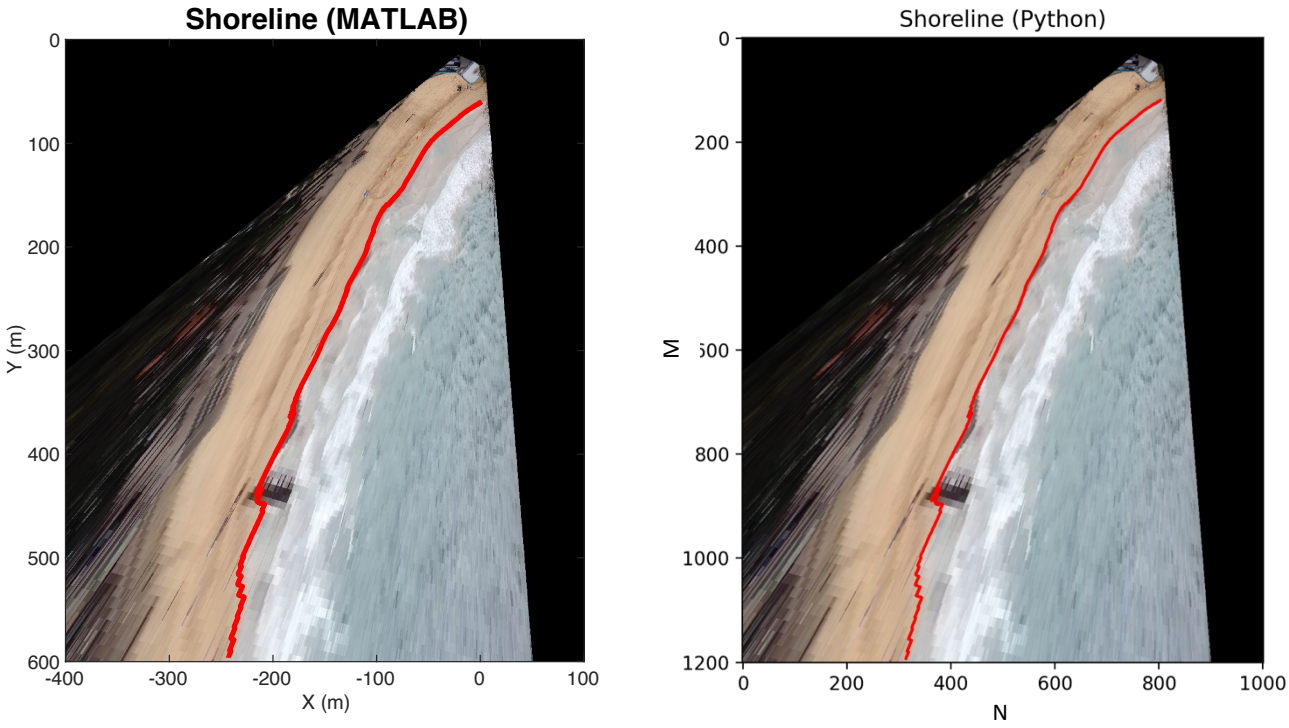


Figure 32: Contour representing the shoreline.

```
contourxy =

    1.5000   59.7726
    1.1589   60.0000
    1.0000   60.1589
    ...
   -243.0000  595.9089
   -243.0456  596.0000
   -243.0911  596.5000
```

```
contourNM =

array([[ 803.        ,  119.54515347],
       [ 802.31773021,  120.        ],
       [ 802.        ,  120.31773021],
       ...,
       [ 314.        , 1191.81773021],
       [ 313.9088651 , 1192.        ],
       [ 313.81773021, 1193.        ]])
```

```
Conversion from (M,N) indices
to (x,y) world coordinates:

x = N/2 - 400

y = M/2
```

```
Python_contourxy =

array([[   1.5       ,  59.77257674],
       [   1.1588651 ,  60.        ],
       [   1.        ,  60.1588651 ],
       ...,
       [-243.        ,  595.9088651],
       [-243.04556745,  596.        ],
       [-243.0911349 ,  596.5       ]])
```

Listing 2 - 10

Conclusions

The coastline is experiencing unprecedented wave and sea level conditions due to climate change. It is crucial to monitor the resulting morphological changes of the cost in order to implement appropriate coastal management practices. Established coastal imaging techniques are costly and unfeasible to implement at a large spatial coverage. CoastSnap is an alternative coastal imaging approach which uses the principles of citizen science to address this problem by crowdsourcing beach images from the public and extracting their shoreline data. A financial barrier due to MATLAB licensing continues to constrain the practicability of CoastSnap, however, this can be avoided by converting the software to the free and open source alternative Python.

This thesis investigates and initiates the conversion of CoastSnap from MATLAB to Python. Focusing on the 'image georectification' and 'map shoreline' sections of the code the underlying theory enabling the processes has been explained. The associated critical functions performing these processes have been identified in MATLAB and an equivalent in Python has been recommended.

Generally the recommended Python functions have performed very well with regard to the data output similarity with the MATLAB functions. Often the outputs are identical. In some cases the Python function output has been unexpected. For example, the mean squared error function initially returned a value close to half the value returned from MATLAB. In this case the cause has been identified as a discrepancy between the two functions and a general adaptation was applied to the Python code. In other situations where the operation of the Python and MATLAB functions have deviated, it has not been possible to code a general adaptation to enable identical

results. For example the MATLAB function *ksdensity* calculated the probability density functions for a value range which could not be replicated using a general rule in Python. In this case, a measure to mitigate the deviation between the results was implemented and in doing so identified inaccuracies in the original MATLAB code.

This work did not test the variations in the shoreline location which result from any inconsistencies between Python and MATLAB functions. This variation should be investigated in further work to identify the significance of the inconsistencies. When very small inconsistencies were found in the outputs from the RGB extracting functions, the inconsistencies were assumed to be negligible in comparison to the overall data, however it is possible that small errors may be exacerbated as they are passed from on function to the next. Testing for exacerbated errors was not undertaken as the focus was on testing individual functions which usually took place in isolation from other functions.

It should also be noted that the data used to test and compare the Python functions is the data from Manly beach in South-East Australia. It would be advantageous to test the functions using data from locations outside this region to ensure the CoastSnap Python code will maintain high performance in locations around the world. It will be particularly important to test the functions which do not perform identically to MATLAB in these regions to test if the output deviations change depending on the region. Ultimately, if the Python code can be made to operate identically to the MATLAB code it is best to initially calibrate the software to South-East Australia as this is the default of the MATLAB code from which all CoastSnap locations around the world have adapted.

It is hoped that the material provided in this thesis will provide the best opportunity for the CoastSnap MATLAB code conversion to continue. By explaining the fundamental theory behind the processes along with the MATLAB code, the initial analysis of the MATLAB code should be straightforward for future colleagues continuing the code conversion. The Python functions recommended and the associated discussion are hoped to provide a framework from which the Python code can be completed.

References

Anthony, E. J. (2019) 'Beach Erosion', in Finkl, C. W. and Makowski, C. (eds) *Encyclopedia of Coastal Science*. Cham: Springer International Publishing, pp. 234–246. doi: 10.1007/978-3-319-93806-6_33.

Buckheit, J. B. and Donoho, D. L. (1995) 'WaveLab and Reproducible Research', in Antoniadis, A. and Oppenheim, G. (eds) *Wavelets and Statistics. Lecture Notes in Statistics*. Springer, New York, NY, pp. 55–81. doi: 10.1007/978-1-4612-2544-7_5.

Carter, Bill, and Woodroffe, C. D. . (1994) *Coastal evolution : late Quaternary shoreline morphodynamics*. Edited by Bill Carter, C. D. Woodroffe, and International Geological Correlation Programme. Cambridge: Cambridge University Press.

Castelle, B. and Harley, M. (2020) *Extreme events: impact and recovery, Sandy Beach Morphodynamics*. Elsevier Ltd. doi: 10.1016/b978-0-08-102927-5.00022-9.

Chandler, M. *et al.* (2017) 'Contribution of citizen science towards international biodiversity monitoring', *Biological Conservation*, 213, pp. 280–294. doi: 10.1016/j.biocon.2016.09.004.

Chen, D. *et al.* (2010) 'Assessment of open source GIS software for water resources management in developing countries', *Journal of Hydro-Environment Research*, 4(3), pp. 253–264. doi: 10.1016/j.jher.2010.04.017.

Citizen Science Association (2020) *Citizen Science Association : Home* . Available at: https://www.citizenscience.org/ (Accessed: 12 December 2020).

Citizens of the Great Barrier Reef (2020) *The Great Reef Census*. Available at: https://greatreefcensus.org/ (Accessed: 12 December 2020).

Committee on Climate Change (2018) *Managing the environment in a changing climate, Committe on Climate Change*.

Conlin, M. (2020) *GitHub - conlin-matt/SurfRCaT: SurfRCaT: Surf-Camera Remote Calibration Tool*. Available at: https://github.com/conlin-matt/SurfRCaT (Accessed: 23 April 2021).

Conlin, M. P. *et al.* (2020) 'SurfRCaT: A tool for remote calibration of pre-existing coastal cameras to enable their use as quantitative coastal monitoring tools', *SoftwareX*, 12, p. 100584. doi: 10.1016/j.softx.2020.100584.

Conrad, C. C. and Hilchey, K. G. (2011) 'A review of citizen science and community-based environmental monitoring: Issues and opportunities', *Environmental Monitoring and Assessment*, 176(1–4), pp. 273–291. doi: 10.1007/s10661-010-1582-5.

Cooper, J. A. G. *et al.* (2020) 'Sandy beaches can survive sea-level rise', *Nature Climate Change*. Nature Research, pp. 993–995. doi: 10.1038/s41558-020-00934-2.

Cooper, J. A. G. and McKenna, J. (2008) 'Social justice in coastal erosion management: The temporal and spatial dimensions', *Geoforum*, 39(1), pp. 294–306. doi: 10.1016/j.geoforum.2007.06.007.

Dasgupta, S. *et al.* (2009) 'The impact of sea level rise on developing countries: A comparative analysis', *Climatic Change*, 93(3–4), pp. 379–388. doi: 10.1007/s10584-008-9499-5.

Eichentopf, S., Karunarathna, H. and Alsina, J. M. (2019) 'Morphodynamics of sandy beaches under the influence of storm sequences: Current research status and future needs', *Water Science and Engineering*, 12(3), pp. 221–234. doi: 10.1016/j.wse.2019.09.007.

Fitton, J. M., Hansom, J. D. and Rennie, A. F. (2016) 'A national coastal erosion susceptibility model for scotland', *Ocean and Coastal Management*, 132, pp. 80–89. doi: 10.1016/j.ocecoaman.2016.08.018.

Forsgren, N. *et al.* (2020) *The 2020 State of the Octoverse | No 2 | Empowering Healthy Communities*.

HabitatMap (2020) *Open Source Software*. Available at: https://www.habitatmap.org/blog/categories/open-source-software#category-list (Accessed: 10 December 2020).

Harley, M. D. *et al.* (2019) 'Shoreline change mapping using crowd-sourced

smartphone images', *Coastal Engineering*, 150(April), pp. 175–189. doi: 10.1016/j.coastaleng.2019.04.003.

Hartley, R. and Zisserman, A. (2011) 'Camera Models', in *Multiple View Geometry in Computer Vision*. Cambridge University Press, pp. 153–177. doi: 10.1017/cbo9780511811685.010.

Hemer, M. A. *et al.* (2013) 'Projected changes in wave climate from a multi-model ensemble', *Nature Climate Change*, 3(5), pp. 471–476. doi: 10.1038/nclimate1791.

Holman, R. A. and Stanley, J. (2007) 'The history and technical capabilities of Argus', *Coastal Engineering*, 54(6–7), pp. 477–491. doi: 10.1016/j.coastaleng.2007.01.003.

Li, V. O. *et al.* (2018) 'Air pollution and environmental injustice: Are the socially deprived exposed to more PM2.5 pollution in Hong Kong?', *Environmental Science and Policy*, 80(June 2017), pp. 53–61. doi: 10.1016/j.envsci.2017.10.014.

Lim, C. C. *et al.* (2019) 'Mapping urban air quality using mobile sampling with low-cost sensors and machine learning in Seoul, South Korea', *Environment International*. doi: 10.1016/j.envint.2019.105022.

Luijendijk, A. *et al.* (2018) 'The State of the World's Beaches', *Scientific Reports*, 8(1), pp. 1–11. doi: 10.1038/s41598-018-24630-6.

Luijendijk, A. and de Vries, S. (2020) *Global beach database*, *Sandy Beach Morphodynamics*. Elsevier Ltd. doi: 10.1016/b978-0-08-102927-5.00026-6.

MacPhail, V. J. and Colla, S. R. (2020) 'Power of the people: A review of citizen science programs for conservation', *Biological Conservation*. Elsevier Ltd, p. 108739. doi: 10.1016/j.biocon.2020.108739.

Magaña, P. *et al.* (2020) 'Approaching software engineering for marine sciences: A single development process for multiple end-user applications', *Journal of Marine Science and Engineering*, 8(5). doi: 10.3390/JMSE8050350.

Masselink, G., Castelle, B., *et al.* (2016) 'Extreme wave activity during 2013/2014 winter and morphological impacts along the Atlantic coast of Europe', *Geophysical*

Research Letters, 43(5), pp. 2135–2143. doi: 10.1002/2015GL067492.

Masselink, G., Scott, T., *et al.* (2016) 'The extreme 2013/2014 winter storms: hydrodynamic forcing and coastal response along the southwest coast of England', *Earth Surface Processes and Landforms*, 41(3), pp. 378–391. doi: 10.1002/esp.3836.

Le Mauff, B. *et al.* (2018) 'Coastal monitoring solutions of the geomorphological response of beach-dune systems using multi-temporal LiDAR datasets (Vendée coast, France)', *Geomorphology*, 304, pp. 121–140. doi: 10.1016/j.geomorph.2017.12.037.

Neumann, B. *et al.* (2015) 'Future Coastal Population Growth and Exposure to Sea-Level Rise and Coastal Flooding - A Global Assessment', *PLOS ONE*. Edited by L. Kumar, 10(3), p. e0118571. doi: 10.1371/journal.pone.0118571.

NSW Department of Planning Industry and Environment (2020) *CoastSnap beach monitoring*. Available at: https://www.environment.nsw.gov.au/research-and-publications/your-research/citizen-science/get-involved/coastsnap (Accessed: 1 November 2020).

Oppenheimer, M. *et al.* (2019) 'Sea Level Rise and Implications for Low Lying Islands, Coasts and Communities.', *IPCC Special Report on the Ocean and Cryosphere in a Changing Climate*.

Pocock, M. J. O. *et al.* (2019) 'Developing the global potential of citizen science: Assessing opportunities that benefit people, society and the environment in East Africa', *Journal of Applied Ecology*. Edited by A. McKenzie, 56(2), pp. 274–281. doi: 10.1111/1365-2664.13279.

Pucino, N. *et al.* (2021) 'Citizen science for monitoring seasonal-scale beach erosion and behaviour with aerial drones', *Scientific Reports*, 11(1), p. 3935. doi: 10.1038/s41598-021-83477-6.

Roche, J. *et al.* (2020) 'Citizen Science, Education, and Learning: Challenges and Opportunities', *Frontiers in Sociology*, 5, p. 14. doi: 10.3389/fsoc.2020.613814.

Roger, E. *et al.* (2020) 'Maximising the potential for citizen science in New South

Wales', *Australian Zoologist*, 40(3), pp. 449–461. doi: 10.7882/AZ.2019.023.

Shahar, D. (2020) *Very slow interpolation using `scipy.interpolate.griddata' - Javaer101*. Available at: https://www.javaer101.com/en/article/12756392.html (Accessed: 23 April 2021).

Short, A. D. and Jackson, D. W. T. (2013) 'Beach Morphodynamics', in *Treatise on Geomorphology*. doi: 10.1016/B978-0-12-374739-6.00275-X.

Splinter, K. D., Harley, M. D. and Turner, I. L. (2018) 'Remote sensing is changing our view of the coast: Insights from 40 years of monitoring at Narrabeen-Collaroy, Australias', *Remote Sensing*. MDPI AG, p. 1744. doi: 10.3390/rs10111744.

Splinter, K. D., Turner, I. L. and Davidson, M. A. (2013) 'How much data is enough? The importance of morphological sampling interval and duration for calibration of empirical shoreline models', *Coastal Engineering*, 77, pp. 14–27. doi: 10.1016/j.coastaleng.2013.02.009.

The MathWorks Inc. (2021a) *Convert subscripts to linear indices - MATLAB sub2ind - MathWorks United Kingdom*. Available at: https://uk.mathworks.com/help/matlab/ref/sub2ind.html (Accessed: 24 April 2021).

The MathWorks Inc. (2021b) *Kernel smoothing function estimate for univariate and bivariate data - MATLAB ksdensity - MathWorks United Kingdom*. Available at: https://uk.mathworks.com/help/stats/ksdensity.html (Accessed: 27 April 2021).

Thom, B. (2020) *Management As Contested Spaces, Sandy Beach Morphodynamics*. Elsevier Ltd. doi: 10.1016/B978-0-08-102927-5/00029-1.

Ton, A. *et al.* (2020) *Beach and nearshore monitoring techniques, Sandy Beach Morphodynamics*. Elsevier Ltd. doi: 10.1016/b978-0-08-102927-5.00027-8.

Turner, I. L. *et al.* (2016) 'A multi-decade dataset of monthly beach profile surveys and inshore wave forcing at Narrabeen, Australia', *Scientific Data*, 3(1), pp. 1–13. doi: 10.1038/sdata.2016.24.

Vermeir, K. *et al.* (2018) *Global Access to Research Software. The Forgotten Pillar of*

Open Science Implementation. Halle.

de Vries, S., Wengrove, M. and Bosboom, J. (2020) *Marine sediment transport, Sandy Beach Morphodynamics*. Elsevier Ltd. doi: 10.1016/b978-0-08-102927-5.00009-6.

Appendices

Appendix I

Code to generate the rotation matrix

```
def angles2R(a, t, s):
        R[0,0] = np.cos(a) * np.cos(s) + np.sin(a) * np.cos(t) * np.sin(s)
        R[0,1] = -np.cos(s) * np.sin(a) + np.sin(s) * np.cos(t) * np.cos(a)
        R[0,2] = np.sin(s) * np.sin(t)
        R[1,0] = -np.sin(s) * np.cos(a) + np.cos(s) * np.cos(t) * np.sin(a)
        R[1,1] = np.sin(s) * np.sin(a) + np.cos(s) * np.cos(t) * np.cos(a)
        R[1,2] = np.cos(s) * np.sin(t)
        R[2,0] = np.sin(t) * np.sin(a)
        R[2,1] = np.sin(t) * np.cos(a)
        R[2,2] = -np.cos(t)
```

```
function R = angles2R(a,t,s)

% R = angles2R(a,t,s)
% Makes rotation matrix from input azimuth, tilt, and swing (roll)
% From p 612 of Wolf, 1983

R(1,1) = cos(a) * cos(s) + sin(a) * cos(t) * sin(s);
R(1,2) = -cos(s) * sin(a) + sin(s) * cos(t) * cos(a);
R(1,3) = sin(s) * sin(t);
R(2,1) = -sin(s) * cos(a) + cos(s) * cos(t) * sin(a);
R(2,2) = sin(s) * sin(a) + cos(s) * cos(t) * cos(a);
R(2,3) = cos(s) * sin(t);
R(3,1) = sin(t) * sin(a);
R(3,2) = sin(t) * cos(a);
R(3,3) = -cos(t);
```

Appendix II

Code adapted from SurfRCaT (Conlin, 2020) used for the initial attempts to generate the plan image in Python.

```
import numpy as np
import numpy.matlib
import scipy.io
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt
import time
GlobsOutput = scipy.io.loadmat('GlobsOutput.mat')

xyz = GlobsOutput['xyz'].astype(float)
UV = GlobsOutput['UV'].astype(float)

beta = [0, 0, 17.301000000000000, -0.349065850398866, 1.396263401595464, 0]
```

Continued...

```
#The above curve curve fit returns the azimuth, tilt and roll (and the
# xCam, yCam and zCam) but to use the function for final image
# rectification from SurfRCaT, the azimuth, tilt and roll need to be
# converted to omega, phi and kappa, using the function below

beta_opt, Cov = curve_fit(TESTFITFUNC, xyz, np.concatenate((UV[:,0],
UV[:,1])),beta)
def calibrate_GetInitialApprox_ats2opk(a,t,s):
'''
Inputs:
    a: (float) Estimated azimuth of the camera (in degrees)
    t: (float) Estimated tilt of the camera (in degrees).
    s: (float) Estimated swing of the camera (in degrees).

Outputs:
    omega: (float) Initial approximation for omega for the camera (in
radians)
    phi: (float) Initial approximation for phi for the camera (in radians)
    kappa: (float) Initial approximation for kappa for the camera (in
radians)
'''
    import math

    # Create the rotation matrix with the ats convention #
    m11 = -(math.cos(a)*math.cos(s)) - \
        (math.sin(a)*math.cos(t)*math.sin(s))
    m12 = (math.sin(a)*math.cos(s)) - (math.cos(a)*math.cos(t)*math.sin(s))
    m13 = -math.sin(t)*math.sin(s)
    m21 = (math.cos(a)*math.sin(s)) - (math.sin(a)*math.cos(t)*math.cos(s))
    m22 = (-math.sin(a)*math.sin(s)) - \
        (math.cos(a)*math.cos(t)*math.cos(s))
    m23 = -math.sin(t)*math.cos(s)
    m31 = -math.sin(a)*math.sin(t)
    m32 = -math.cos(a)*math.sin(t)
    m33 = math.cos(t)

    # Get omega,phi,kappa from the matrix #
    phi = math.asin(m31)
    omega = math.atan2(-m32,m33)
    kappa = math.atan2(-m21,m11)

    return omega,phi,kappa

omega, phi, kappa = calibrate_GetInitialApprox_ats2opk(beta_opt[3],\
    beta_opt[4], beta_opt[5])

#creating the calibVals variable to input into rectify_RectifyImage
# below the order of constituents is: (omega, phi, kappa, xCam,
# yCam, zCam, fx, c0U, c0V)
```

Continued...

```
#f: (float) initial approximation for focal length (in pixels)
#x0: (float) initial approximation for x principal point coordinate
#y0: (float) initial approximation for y principal point coordinate

def create_calibVals(omega, phi, kappa, xCam, yCam, zCam):

    mat = scipy.io.loadmat('RectifyImagePython.mat')

    fx = mat['fx'][0,0].astype(float)
    c0U = mat['c0U'][0,0].astype(float)
    c0V = mat['c0V'][0,0].astype(float)

    calibVals = np.array([omega, phi, kappa, xCam, yCam, zCam, fx, c0U,
c0V])

    return calibVals

calibVals = create_calibVals(omega, phi, kappa, beta_opt[0], beta_opt[1],
beta_opt[2])

#define the other inputs for the rectify_RectifyImage function below

full_image = scipy.io.loadmat('Full_Image_Rectification.mat')

Oblique_Image = full_image['I'].astype(float)
xmin = -400
xmax = 100
dx = 0.5
ymin = 0
ymax = 600
dy = 0.5
z = -0.0280

#=====================================================================#
# Perform Rectification #
#=====================================================================#
def rectify_RectifyImage(calibVals,img,xmin,xmax,dx,ymin,ymax,dy,z):
    '''
    Function to rectify an image using the resolved calibration parameters.
    User inputs a grid in real world space onto which the image is
    rectified.
    Inputs:
        calibVals: (array) The calibration vector returned by
        calibrate_PerformCalibration function
        img: (array) The image to be rectified
        xmin: (float) minimum x-coordinate of real-world grid
        xmax: (float) maximum x-coordinate of real-world grid
        dx: (float) spacing in x-direction of the grid
        ymin: (float) minimum y-coordinate of real-world grid
        ymax: (float) maximum y-coordinate of real-world grid
        dy: (float) spacing in y-direction of the grid
        z: (float) elevation onto which the image is projected #
```

Continued...

```
    Outputs:
        im_rectif: (array) The rectified image
        extents: (array) The geographic extents of the rectified image,
        for plotting purposes
    '''
    import math
    import numpy as np
    from scipy.interpolate import interp2d,griddata
    # Define the calib params #
    omega = calibVals[0]
    phi = calibVals[1]
    kappa = calibVals[2]
    XL = calibVals[3]
    YL = calibVals[4]
    ZL = calibVals[5]
    f = calibVals[6]
    x0 = calibVals[7]
    y0 = calibVals[8]

    # RESET IMAGE (This is an aleration Nick Heaney has made during testing
    # of CoastSnapPython)
    img = np.fliplr(img)

    # Set up the rotation matrix #
    m11 = math.cos(phi)*math.cos(kappa)
    m12 = (math.sin(omega)*math.sin(phi)*math.cos(kappa))\
        + (math.cos(omega)*math.sin(kappa))
    m13 = (-math.cos(omega)*math.sin(phi)*math.cos(kappa))\
        + (math.sin(omega)*math.sin(kappa))
    m21 = -math.cos(phi)*math.sin(kappa)
    m22 = (-math.sin(omega)*math.sin(phi)*math.sin(kappa))\
        + (math.cos(omega)*math.cos(kappa))
    m23 = (math.cos(omega)*math.sin(phi)*math.sin(kappa))\
        + (math.sin(omega)*math.cos(kappa))
    m31 = math.sin(phi)
    m32 = -math.sin(omega)*math.cos(phi)
    m33 = math.cos(omega)*math.cos(phi)

    # Set up object-space grid #
    xg = np.arange(xmin,xmax,dx)
    yg = np.arange(ymin,ymax,dy)
    xgrd,ygrd = np.meshgrid(xg,yg)
    zgrd = np.zeros([len(xgrd[:,1]),len(xgrd[1,:])])+z
    extents = np.array([(-.5*dx)+min(xg),max(xg)+(.5*dx),\
        min(yg)-(.5*dy),max(yg)+(.5*dy)])

    # Get image coordinates of each desired world coordinate
    # based on calib vals
    x = x0 - (f*(((m11*(xgrd-XL)) + (m12*(ygrd-YL)) + (m13*(zgrd-ZL))) / \
        ((m31*(xgrd-XL)) + (m32*(ygrd-YL)) + (m33*(zgrd-ZL)))))
    y = y0 - (f*(((m21*(xgrd-XL)) + (m22*(ygrd-YL)) + (m23*(zgrd-ZL))) / \
        ((m31*(xgrd-XL)) + (m32*(ygrd-YL)) + (m33*(zgrd-ZL)))))
```

Continued...

```
    # Create grid for the photo coordinates #
    u = np.arange(len(img[0,:,1]))
    v = np.arange(len(img[:,0,1]))
    ug,vg = np.meshgrid(u,v)

    # Interpolate xy (image coordinates) of world points to photo
    # coordinates to get color
    uInterp = np.reshape(ug,[np.size(ug)])
    vInterp = np.reshape(vg,[np.size(vg)])
    rInterp = np.reshape(img[:,:,0],[np.size(img[:,:,0])])
    gInterp = np.reshape(img[:,:,1],[np.size(img[:,:,1])])
    bInterp = np.reshape(img[:,:,2],[np.size(img[:,:,2])])
    xInterp = np.reshape(x,[np.size(x)])
    yInterp = np.reshape(y,[np.size(y)])

    col_r = griddata((uInterp,vInterp),rInterp,(xInterp,yInterp))
    col_g = griddata((uInterp,vInterp),gInterp,(xInterp,yInterp))
    col_b = griddata((uInterp,vInterp),bInterp,(xInterp,yInterp))

    col_r = np.reshape(col_r,[len(x[:,0]),len(x[0,:])])
    col_g = np.reshape(col_g,[len(x[:,0]),len(x[0,:])])
    col_b = np.reshape(col_b,[len(x[:,0]),len(x[0,:])])

    # Create the rectified image #
    im_rectif = np.stack([col_r,col_g,col_b],axis=2)
    im_rectif = np.flipud(im_rectif)

    return im_rectif,extents
'''
#    This program is free software: you can redistribute it and/or
#    modify it under the terms of the GNU General Public License as
#    published by the Free Software Foundation, version 3 of the
#    License.

#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.

#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see
#                                <http://www.gnu.org/licenses/>.
'''


im_rectif,extents = rectify_RectifyImage(calibVals, Oblique_Image, xmin, \
    xmax, dx, ymin, ymax, dy, z)
im_rectif = im_rectif.astype(np.int32)
plt.imshow(im_rectif)
```