Finding Wally

Parallel Object Recognition

Cian Booth

MSc. Dissertation Report

Abstract

A report on the potential of parallel computer vision for everyday use. Discussed within are algorithms and libraries that could make this a possibility. Special focus is put on the computer vision library OpenCV. Where's Wally? puzzles are used as a test case for parallel object recognition.

Contents

1 Introduction **2**
1.1 Computer Vision . . . 2
1.2 High Performance Computer Vision . . . 3
1.3 Potential Use Cases . . . 3
1.4 Where's Wally? as a Test Case . . . 5
1.5 Goals . . . 5
1.6 Overview of Report . . . 6

2 Background Information **6**
2.1 Computer Vision . . . 6
2.2 Object Recognition . . . 7
2.3 Shape and Colour Analysis . . . 7
2.3.1 Feature Analysis . . . 8
2.4 Computer Vision Libraries . . . 9
2.5 Parallel Programming . . . 9
2.6 Parallel Computer Vision . . . 10
2.7 Shared Memory Parallelism . . . 12

3 Algorithms **13**
3.1 Colour Analysis . . . 14
3.1.1 Colour Extraction . . . 14
3.1.2 Blur . . . 14
3.1.3 Sharpen . . . 16
3.2 Region Detection . . . 17
3.3 Naive Solution . . . 17
3.3.1 Flood Fill . . . 18
3.3.2 Connected Component Labelling . . . 19
3.4 Line Width Estimation . . . 21
3.4.1 Average Distance to a Zero Valued Pixel . . . 23
3.4.2 Standard Deviation in Distance to Zero Valued Pixels 25
3.4.3 Combining the Methods . . . 27
3.4.4 Testing . . . 29
3.4.5 Parallelism . . . 29
3.5 Shape Analysis . . . 30
3.6 Feature Detection . . . 30
3.6.1 Scale-Invariant Feature Transform . . . 30
3.6.2 Speeded Up Robust Features . . . 31

4 Implementation **31**
4.1 Red And White . . . 31
4.2 Colour Analysis . . . 31
4.2.1 Function: `get_colour_in_image` . . . 32
4.2.2 Function: `get_greyscale_in_image` . . . 33
4.2.3 Function: `find_regions_from_mask` . . . 34
4.3 Finding Red and White Stripes . . . 34
4.3.1 Weaknesses . . . 35
4.3.2 Testing . . . 36
4.3.3 Parallelism . . . 36
4.4 Blue Trousers . . . 37

4.5 Find Glasses 37
4.6 Find Features 37

5 Results **37**

6 Line Width **37**

7 Conclusion and Evaluation **37**
7.1 Evaluation 37

List of Figures

1.1 A map of Edinburgh 4
1.2 Characters from Where's Wally? 5
3.1 Gaussian Blurring 15
3.2 Steps in a naive region detection algorithm 17
3.3 The Flood Fill Algorithm 18
3.4 The Connected Component Labelling Algorithm 20
3.5 Graphic description of parallel region detection 21
3.6 Examples of the zero-distance of black lines 22
3.7 Relationship between line width and mean zero-distance 24
3.8 Relationship between line width and standard deviation . 26
3.9 Maximum zero-distance reduction to produce sane mask 27
3.10 Sensitivity of mean and standard to errors 28
3.11 Test images for `estimate_black_line_thickness` 29
4.1 Isolating yellow from an image using `get_colour_in_image` 33
4.2 Example of finding red and white striped regions 35
4.3 Fail case for the 'Red and White' pattern 35
4.4 Test images used to check the red and white pattern . . . 36
6.1 Time and Speedup of parallel line width analysis 38

List of Tables

3.1 Example of Colours in Hexadecimal format 14
3.2 Process of sharpening an image 16
3.3 Table of estimated line widths of test images 29

List of Listings

1 'A simple loop parallelised with OpenMP' 12
2 'A simple loop parallelised with MPI' 13

Acknowledgements

Many thanks go to my supervisor, Alistair Grant, for directing me to such an interesting project, and keeping me on the rails.

1 Introduction

The field of computer vision studies the use of computers to process and analyse images. Some aspects allow robots and other machines to process visual data. Other aspects are an attempt to augment the human ability to percieve visual information. This portion of computer vision allows users to reduce the time taken to produce information about an image. It can also help to increase the confidence that results related to that information is correct.

1.1 Computer Vision

Recognising an object is one of the most important things that human vision is able to do. This means that the brain is able to decipher the basic information received from the eyes, identifying any objects the information describes. For the average human can expect to see the local environment and immediately recognising objects within. The act of object recognition is a subconscious one; humans do not have to actively think about their environment to understand what objects exist there. As a subconcious act, the complexity associated with human object recognition is not known. The level of information contained in a single 'frame' of vision is extremely high. A normal room can contain hundreds or thousands of distinct shapes and surfaces, which must be pieced together to form a cohesive whole. Light levels vary from one position in a room to another, so objects may appear different on two sides. The human brain is able to analyse this data in real time, and make decisions about objects in the image.

The adage "a picture is worth a thousand words" is particularly meaningful here; fully creating and expressing a logical definition of an object would be an extensive and verbose task. This logical expression would have a high level of complexity, needing precise measurements of colour, shape, texture and more. It is a complex task to begin to define these values in a meaningful way.

Implementing this on a computer is not simple. The normal description of a computer is a machine that is designed to calculate mathematical operations at speed. Modern computers are binary devices, designed to store and act upon precise values. Human vision, as currently understood, does not directly map onto the types of operations that computers excel at.

Despite these difficulties, computer vision is an impressively mature field. Algorithms such as Scale-Invariant Feature Transform (SIFT)[1] have been developed, which are able to find known objects under various transformations. It is used in numerous areas, from robotics control, to automation defect recognition in manufacturing and tracking user motions with devices like the Kinect. Computer vision has the potential to be be useful in countless fields, and in myriad aspects of everyday life.

1.2 High Performance Computer Vision

One of the major obstructions for widespread use of computer vision is the limiting relationships between speed, accuracy and genericism. For this report, speed is defined as the wall time (externally measured time taken to complete) of the program. Accuracy is the confidence with which results can be said to be true, and genericism is how easily the techniques can be adapted to different problems. The human eye can detect most objects quickly, accurately and generically. Computer vision is not so advanced, . Achieving real time speeds comes at the cost of accuracy or generality. Conversely, creating a fast or generalised system comes at the cost of a greatly increased completion time.

For each way that a human can describe an object, multiple techniques emerge to locate the object. Each technique has varying requirements and reliability. Some need specific descriptions (e.g. a sample image) and produce very reliable results. Others do not require such precise descriptions, but do not create dependable results. By combining these techniques together, the reliability and robustness of computational object recognition can be improved. Using all available techniques can considerably increase the runtime of the program. This can be mitigated by computing techniques in parallel.

An suitable method of parallelism for this type of problem is the task farm. This means that general tasks (here, identification methods) will be run concurrently. Task farming is useful in this case, because it allows the use of serial libraries and algorithms that complicate or prevent parallelism. Computing results this way, a task farm allows computer vision programs to produce results quickly, reliably and generically.

1.3 Potential Use Cases

Parallel object recognition is a powerful technique that could be useful for many different areas.

The UK Missing Person Bureau released data on missing persons in 2010-2011 [2]. During this period, two thirds of missing people in the UK were under the age of 18. This group is particularly vulnerable to abduction and abuse if left unsupervised. Although the majority of missing people were found within the 5 miles of their homes, up 21% of people were further out. A 5 mile radius is a large area to look for one person, especially in an urban area. Further out, and it becomes untenable to search efficiently. Figure 1.1 shows the area that a 5 mile radius contains encompasses most of the urban area of Edinburgh city. Almost 4/5 of missing people are found within the first 16 hours. According to information released from a survey of young runaways [3], 34% of said they had been harmed or in a risky experience more than once, and 11% expressly said they had been hurt or harmed. It is critical to find at risk individuals, before they are harmed. Computer vision is already used to facilitate searches, but the volume of data available can exceed the capabilities of existing programs. Furthermore, surveillance data from CCTV equipment produces data at real time speeds, so faster than real time processing speeds are desired. Real time speeds means that the program is able to analyse the data faster than it is being produced. This may in-

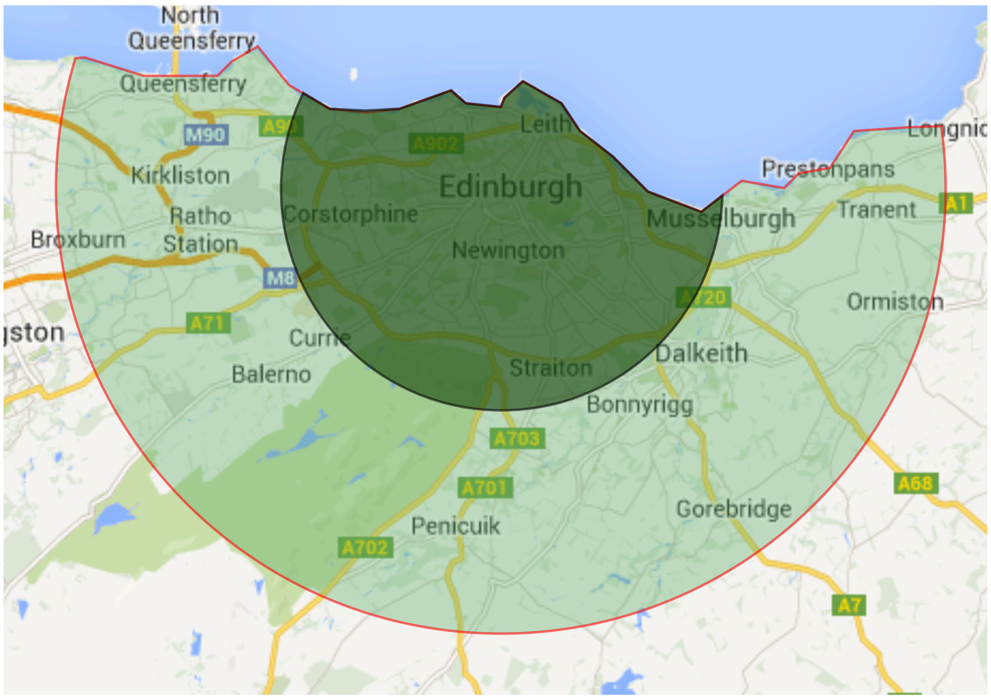


Figure 1.1: A map of Edinburgh, with 5 and 10 mile radius ares displayed. A 5 mile radius circle includes the majority of the urban areas of Edinburgh city. Base image courtesy of Google Maps.

volve reducing the amount of data being input i.e analysing 1 in every 30 images. Implementing a parallelised version of image recognition tools should greatly reduce the time an image takes to search. Governmental agencies, such as the police, could have access to very large computer systems, allowing for a high degree of parallelism.

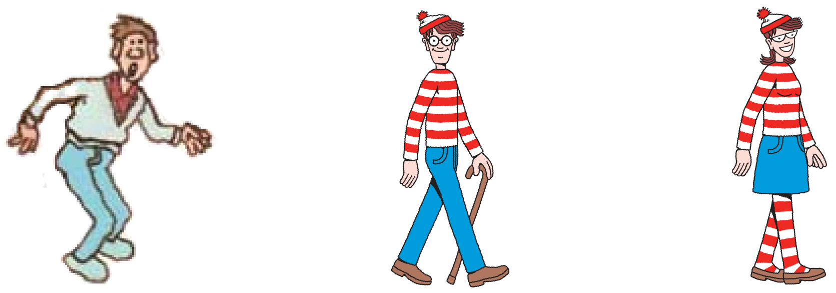
Another potential use can be found with surveying populations of wild animals. Wildlife conservation is a delicate task, which could benefit from rapid computer vision techniques. To accurately know which species are endangered, it is important to have an accurate count of the members of the species for a given region. Actively surveying the population by means of physically interacting with members can have adverse effects on the population. Nielsen [4] discusses the negative effects of electrofishing on rare fish populations. She dicusses the lack of non-invasive methods of surveying the population, without which population counts cannot be maintained. Directly surveying endangered species can be inefficient, slow or dangerous to either the researcher or the animal in question. Passive techniques, such as photography, allow the researcher to estimate populations without interacting with the environment. Ideally a researcher would be constantly vigilant and able be to immediately identify each species correctly. This is rarely the case; a single human is fallible and a team is often beyond the funding of the endeavour. Instead, with access to any modern laptop and a digital camera, parallel computer vision may be able to assist in many ways. A video feed would allow observation for as long as the battery lasts, and a database of the features of regional species would help with identification. Parallel species recognition would allow multiple species to be surveyed at a time. It would allow non-experts to monitor the survey of the populations, freeing up experts for more specialised tasks.

An everyday use of parallel object recognition is with nationwide traffic monitoring. Using existing roadside cameras, such as CCTV or speed cameras, a network could be built that monitors traffic on a large scale. This would help to improve commute times and general congestion issues, by advising drivers of congested areas and providing alternate routes. Paralleism could be applied to this situation. The sheer quantity of data for a large scale system like this would prevent real time analysis for a linear program.

A simple and approachable usage case can be found in Where's Wally? puzzles.

1.4 Where's Wally? as a Test Case

Where's Wally? puzzles are a good testbed for parallel computer vision. Each puzzle is a simple cartoon, a large image filled with various characters, who wear simply coloured clothing, see Figure 1.2(a). One of these characters is the eponymous Wally, who dresses distinctly from most others characters, see Figure 1.2(b). Similarly dressed characters exist, Figure 1.2(c), to add complexity in finding Wally correctly. The cartoon nature of the characters means that shapes are boldly coloured and often bordered by a black line. As Where's Wally? is a puzzle, Wally will be hard to find; he can be obscured, camouflaged or simply small. Creating a program to solve Where's Wally? is non-trivial, requiring a combination of computer vision techniques. Thus the puzzle provides a good base to develop parallel object recognition.



(a) A normal person **(b)** Wally **(c)** Wenda

Figure 1.2: Characters from Where's Wally?

1.5 Goals

This report presents Where's Wally? as a testbed to determine if High Performance Computer Vision can feasible used in in everyday life. This includes the production of a suite of functions that, though tuned to locating Wally, could be used to find other characters. The use of directive based parallelism will be added, to determine if a user-friendly system is viable. A simple and usable system is critical for everyday use. The more expertise required to implement parallel computer vision techniques, the fewer people are capable of creating solutions. Fewer solutions means that the problems will be restricted to the most practical problems, which in turn reduces the number of everyday applications. The generality of the suite of functions will be tested on non-Wally puzzles, to determine how generic the solution is.

1.6 Overview of Report

This report will begin by discussing the underlying information required to comprehend parallel object recognition. This includes literature reviews. The next section discusses the patterns used to recognise Wally. Each includes an analysis of the algorithm selection and a details of the level of parallelism which can be exposed. This will be used to produce a Where's Wally? solver, which examines the parallelism used to increase the speed of the Where's Wally solver.

The report will comment on the results produced by the patterns and the solver. Following this, will be the concluding statements and ideas, along with recommendations for future use. The report will close with an evaluation of the project as a whole. Differences between the preparation phase and the actual report period will be noted here.

2 Background Information

2.1 Computer Vision

Computer vision is a wide ranging field, with a large variety of algorithms and libraries available for use. It is involved with topics such as artificial intelligence[5], machine vision[6], and image processing. Computer vision also draws upon a large range of established fields, such as mathematics, physics and neurobiology. Many

Computer vision techniques can be used for numerous tasks, including motion analysis, image restoration and object recognition.

Motion analysis attempts to determine estimates of object velocities in a stream of images. This could be due to the motion of the camera, motion of visible objects, or a combination of the two. This has broad uses, but has seen a rise in public interest with devices like the Xbox Kinect[7]. The Kinect enables the user to interface with the Xbox through by tracking the motion of their body. Users do not need any control to select menu options, scroll windows or even play games. This is both a good example of how computer vision can be applied, and a good indication of how ready it is for everyday use.

Image restoration is the recovery of corrupt or otherwise marred images. One of the most interesting applications of this is in the recovery of medical image[8]. Obtaining high resolution images from MRI scans normally requires a large number of measurements. This is time consuming, which can be distressing for the patient and expensive for the hospital. However, taking a smaller number of measurements on purpose can help to improve the situation without forgoing quality. In this case, this is because image restoration allows the reconstruction of high resolution images with only a small amount of input data.

The computer vision topic that concerns Where's Wally? images the most is object recognition. This is covered in depth in the section 2.2.

2.2 Object Recognition

Object recognition is a key technique in computer vision. In 1965, Roberts wrote the first paper on computer vision[9].

Finding Wally in a these puzzles is by definition a problem of object recognition. Wally must be correctly identified from other characters, furniture and even food. The image needs to be analysed so that the correct Wally can be located. We discuss two of most directly useful techniques for defining objects in a cartoon style image.

2.3 Shape and Colour Analysis

One way of analysing an image is to break it down into shapes and colours. In linguistic terms, it is often simplest to depict an object by describing it's shape and colour, i.e. "the red box" or "the green hand". This is conceptually simple to explain and understand, and thus is more intuitive to program.

Everyday access to object recognition would generally be done through digital cameras, scanners and other similar devices. Most images saved this way are stored as raster images (e.g. PNG, BMP, GIF), which is a 2D array of colours that directly map to pixels on the screen. This is because the input devices do not have the capability of recognising objects in the images they produce. This is opposed to vector images (e.g. PDF, SVG, SWF), which store the location and colour of geometric primitives (squares, circles, triangles etc.). It is often much simpler to perform shape and colour analysis on vector images.

Regardless of format, methods of colour analysis are simple to implement programmatically. This is because colour is intrinsic to all methods of storing the properties of an image. It is nearly impossible to describe a specific object without discussing some aspect of it's colour, even it that colour is on the greyscale.

Shape analysis is more complex for raster images. Unlike vector images, shape boundaries are not clearly defined. The global boundary is normally found through edge detection. The global boundary is defined here as a single object composed of every boundary in the image. Detecting the shapes requires the specific boundaries to be found, which requires further analysis. The global boundary is analysed to find points of boundary intersection, and the curvature between those points. The group of curves must then be examined to find combinations that produce shapes.

Shape and colour analysis allows for a descriptional, heuristic method for locating objects. In Where's Wally puzzles, this means no previous image of Wally is needed. All that is required is a basic description, such as "red and white stripes" or "black glasses". This allows users to extend the solution past Wally, and towards other characters, who have not previously been seen. Care must be taken as this form of analysis can lead to false positives. To prevent this, many different types of analysis should be combined. For example, a match for "red and white jumpers" that also has a nearby match for "skin colour" would produce a result that has more confidence of being correct.

An example of the usefulness of shape and colour analysis, beyond Where's Wally, could be found in augmented reality (AR) technology. Google is developing this technology with the Google Glass device[10].

A common experience is the misplacement of keys. Users with access to AR devices could use colour and shape analysis to enhance their own searching (i.e. if they are visible, but in a cluttered area). As keys, with some exceptions, have a few well defined shapes and possible colour schemes; the device would not have to store what the specific keys look like in advance. Assuming that parallelism is available in such devices, this technique could potentially provide real time analysis of the scene, significantly helping the user.

2.3.1 Feature Analysis

Some of the most reliable computer vision algorithms (such as SIFT[1]) were developed while considering the neuroscience of human vision. Tanaka[11] and Perrett and Oram[12] studied this in detail. They found that human vision identifies objects with features that are invariant to brightness, scale and position. Explicitly, this means that humans are able to recognise the same objects under differing light levels, at different distances and in different positions in a room. These results have been used as inspiration for feature analysis.

This technique finds 'features', which are points in an image that are scale, rotation and illumination invariant. These features are most immediately useful when compared with the features of another image. For example, the features from a solo image Wally can be used to locate the same Wally in a group image.

The image that is being searched is often referred to as the Scene. Images that are being searched for in the Scene are known as Objects [1]. The keypoints of objects are known properties, and can be searched against the unknown keypoints of the scene.

This method generally requires an existing image to find Wally, which restricts the flexibility of the search. When correctly implemented, this method is very reliable. If Wally is not obscured, results found can be assigned high confidence. Normally Wally is obscured, so this method should be combined with other techniques. Combination helps to provide more flexibility.

Feature analysis is useful in the manufacturing industry. Mass manufactured products need to be checked for defects. Due to the regularity of most merchandise produced this way, this is ideal for feature analysis. If products in the Scene do not match the features of the list of Objects, then the product can be determined to be faulty. This can simply be parallelised; multiple production streams can be scanned simultaneously. Using a centralised system could be cheaper than implementing per-line machines.

[1]Scene and Object are capitalised here to avoid confusion with the more general term of object

2.4 Computer Vision Libraries

Since the advent of computer vision, many libraries have been developed to implement and group computer vision techniques. Arguably, the chief among these is OpenCV.

OpenCV[13] is an open source library used for implementing a wide range of computer vision techniques. Using in-built functions, users have immediate and simplified access to complicated algorithms. OpenCV is available in three of the most major operating systems, Linux, Windows and Mac OS. The library is aimed at developing real-time solutions to computer vision problems[14]. The online documentation for OpenCV is extensive, including tutorials for common topics such as image recognition, machine learning and image processing. These properties make OpenCV ideal for implementing computer vision on a wide scale. OpenCV has some drawbacks. It makes use of bespoke classes for dealing with arrays, called `Mat`. The `Mat` class helps to minimise the memory usage of programs using large arrays. Direct pixel access to `Mat` classes is not accessible in the normally expected C++ fashion. A template function, `Mat::at<type>(x,y)`, is instead the method of access. This could confuse new users.

Another computer vision library is libCVD[15], based in Cambridge University. This is a versatile library, designed for speed and portability. It is often used to access streams of video data. Unlike OpenCV, it stores in pixels in readily accessible STL vectors. This is useful, because it makes accessing pixels more intuitive. However, libCVD does not have the range of implemented computer vision techniques found in OpenCV. It also lacks the level of documentation that OpenCV offers. LibCVD can be used in combination with OpenCV[16]. This allows the user access to the breadth of functions available in OpenCV, as well as the speed of input that libCVD brings.

OpenCV will be the sole library used to implement the Where's Wally? solver. Using multiple libraries overly complicates the task, where a key aim is to produce a simple everyday implementation. OpenCV has many attractive properties for non-expert users, when compared with other libraries. These do not have the documentation or community to help with development issues.

2.5 Parallel Programming

When implementing parallelism, it is desirable to not only speed up a program, but also to use an efficient number of cores. A useful definition is speedup; the scale to which a parallelised version of a program is faster than an efficient serial implementation.

$$\text{Speedup}(P) := \frac{T_{\text{serial}}}{T_{\text{parallel}}(P)} \tag{2.1}$$

Equation 2.1 describes the speedup of a system. Here P is the number of computing units (threads, processors etc.), T_{serial} is the time the serial program takes and T_{parallel} is the parallel time. For most systems, the maximum speedup possible is equivalent to P, known as linear speedup.

Most implementations do not achieve linear speedup, especially for large numbers of computing units. A program that solves a problem of fixed size with near-linear speedup is said to experience 'strong scaling'. In general, scaling will be limited being strong long before this happens, due to Amdahl's law, seen in equation 2.3.

$$T_{parallel}(P) = T_{serial}\left(\alpha - \frac{1-\alpha}{P}\right) \tag{2.2}$$

Here, α is the portion of the code that is purely serial. Inserting this into the previous definition of speedup in equation 2.1, we see Amdahl's Law emerge.

$$\text{Speedup}(P) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(P)} = \frac{T_{\text{serial}}}{T_{\text{serial}}\left(\alpha - frac1 - \alpha P\right)} = \frac{1}{\alpha + \frac{1-\alpha}{P}} \tag{2.3}$$

As P tends to large numbers the speedup approaches the constant value of $1/\alpha$.

Parallel programs instead often aim for to achieve 'weak scaling', described by Gustafson's law. This is the case if the problem size scales with the number of cores for a fixed amount of work per computing unit.

$$\begin{aligned} T(P,N) &= T_{\text{serial}} + T_{\text{parallel},N} = \alpha + \frac{N(1-\alpha)}{P} \\ T(1,N) &= T_{\text{serial}} + NT_{\text{parallel},N} = \alpha + N(1-\alpha) \end{aligned}$$

Here, P is the number of computing units, and N is the size of the problem. We define $T(P,N)$ as the time for the program to complete the purely serial sections. T_{serial} is the time it takes for each of the P processors to complete one task of size $1/N$, $T_{\text{parallel},1/N}$. The time for 1 processor to do an equivalent amount of work is defined in $T(1,N)$. This is the sum of the serial time T_{serial} and every bit of work the processors would do, i.e. $NT_{\text{parallel},N}$.

$$\text{Speedup}(P,N) = \frac{T(1,N)}{T(P,N)} = \frac{\alpha + N(1-\alpha)}{\alpha + \frac{N(1-\alpha)}{P}}$$

$$\text{Speedup}(P,\beta P) = \frac{\alpha + \beta P(1-\alpha)}{\alpha + \beta(1-\alpha)} \propto P \tag{2.4}$$

Here, β is some scaling constant. It is clear that Gustafson's Law produces better scaling, as long as N scales with P.

Everyday users have most regular access to computer vision techniques through mobile devices, such as phones, laptops and ipads. These devices rarely have more than 2 cores. The most cores that can be expected is within a home computer, having no more than 8 cores Any parallelism applied should not require the typically large number of cores. This also implies that scalability is only a concern for programs with a high degree of seriality.

2.6 Parallel Computer Vision

Computer Vision poses an interesting challenge for parallelisation. In some regards, computer vision is a typical data parallel problem. Data parallelism uses the available computing units to act on subsets of the total data. Such program simply needs to act as efficiently as possible on multi-dimensional arrays (normally 2 physical dimensions each with 3 colour dimensions). This is a common form of parallelism, implemented in fields such as parallel Fourier transforms, simulation of crystalline matter and database analysis. Image processing, a subset of computer vision, normally falls under this category.

More complicated elements of computer vision fit task parallelism better. This is because they require several independent tasks to be completed.

Downton[17] discusses the use of pipeline processing farm in computer vision. This is similar to a task farm, but has a continuous flow of data to process. By parallelising independent tasks, the latency of image analysis can be reduced. This paper was written in 1994 and does not take into account modern technology when discussing potential uses. The advent of multi-core camera phones broadens potential implementations beyond the encoding algorithms and handwriting recognition discussed.

A current trend in parallel computing is to use accelerators to improve performance. Graphical Processing Units (GPUs), are the most common choice, due to their relatively high performance/price ratio. Fung[18] implements the GPU based acceleration for several computer vision techniques, notably including feature detection. The features are detected using Harris detectors, which are often not used in favour of SIFT-like keypoints. The paper does not discuss the details or the benefits of implementing parallel Harris detectors.

Parallel implementations exist for many algorithms, such as SIFT and Speeded Up Robust Features (SURF), discussed further in section 3.6.2. Yimin Zhang[19] implements two forms of parallelism for SIFT, both showing large increases in the amount of frames that can be calculated per second. At 640x480 pixels, the image size used to obtain these speedups is small in comparison to an average Where's Wally? puzzle. Current digital cameras, which might be used with everyday computer vision, typically offer images that are orders of magnitude larger. Despite speedups, the size of these images might inhibit analysis at a useful speed.

Expanding on previous work, Zhang et. al. discuss in depth the effects that limit scalability [20]. The paper shows the purely serial portion of the code takes up less than 2% of the runtime. This shows that reduced scalability is due to more complicated factors, which should be avoided by novice users.

Nan Zhang presents the multi-core implementation of parallel SURF[21]. Within, Zhang shows that the multi-core implementations can have speed comparable to GPU implementations. Low-end computers (and by extension, cheap mobile devices) are suggested to lack the quality of GPU that can implement high speed algorithms. This means that CPU based implementations are preferable for widespread usage.

The existing parallel implementations of SIFT/SURF algorithms are not directly available. This adds complexity to the project, which aims to maintain simplicity to ensure a wide user base.

2.7 Shared Memory Parallelism

This project will implement parallelism through the shared memory multiprocessing API known as OpenMP.

Shared memory multiprocessing is a form of parallelism that relies on the presence of shared memory. This is a region of memory that can be accessed equally by several processors. Normal examples of this are in multi-core systems, which have a shared L2 or L3 cache. Having shared memory allows data to be quickly shared amongst processors. This removes redundant and temporally expensive copying of data from main memory, and reduces the overall cache usage. Using less of the available cache means that a larger amount of other values can be stored, also reducing calls to main memory. In a similar way, messages can also be passed between processors at high speeds. Communication through a local cache is considerably faster than typical messaging systems. Most recently manufactured computers are multi-core, so it is likely that everyday devices will be able to benefit from shared memory parallelism.

OpenMP implements shared memory multiprocessing through the use of directives and routines. A directive is a compiler flag that informs the compiler that the user intends for a specific behaviour to occur. These are simple to include into a program, often requiring little to no changes to a serial program to parallelise it. Listing 1 shows the simplicity of implementing parallelism.

Listing 1: 'A simple loop parallelised with OpenMP'

```
// a simple for loop
int x[4];
for(int i=0; i<4; i++) {
  x[i] = i;
}

// and again parallelised with OpenMP
int y[4];
#pragma omp parallel for default(none) shared(y)
for(int i=0; i<4; i++) {
  y[i] = i;
}
```

There are some issues that come with the use of shared memory multiprocessing. The main one is that scaling is often poor. This is due to the fact that there is a fixed amount of processors attached to any block of shared memory. Once the system is scaled beyond this amount, the program starts to become dominated by the speed of message between the two memory systems. Furthermore, the speed at which the CPU can write to the memory is limited. Increasing the number of processors attempting to write to memory through the same CPU causes a bottleneck to occur.

Another popular form of parallelism is through message passing. This is passing messages between processors. In this way data can be shared between processors, allowing for processors to interact. Unlike shared memory parallelism, message passing protocols typically do not rely on shared memory. Messages are instead sent over the bus, meaning that the processors being used can be in different nodes.

One of the most commonly implemented message passing standards is the Message Passing Interface (MPI). MPI scales to very large numbers of processors. However, converting existing code for use in MPI is time consuming, often considerably increasing the maintenance for the code. Listing 2 shows one potential way to parallelise the same loop from listing 1 using MPI.

Listing 2: 'A simple loop parallelised with MPI'

```
// a simple for loop
int x[4];
for(int i=0; i<4; i++) {
  x[i] = i;
}

// and again parallelised with MPI
int y[4];
int rank;
MPI_Status status;
MPI_Comm_rank(&rank);
if(rank == 0) {
  y[0] = 0;
  MPI_Recv(&y[1], 1, MPI_INT,0,0,MPI_COMM_WORLD, &status);
  MPI_Recv(&y[2], 1, MPI_INT,0,0,MPI_COMM_WORLD, &status);
  MPI_Recv(&y[3], 1, MPI_INT,0,0,MPI_COMM_WORLD, &status);
} else {
  MPI_Send(rank,1, MPI_INT,0,0, MPI_COMM_WORLD);
}
```

MPI requires experienced users to produce efficient code. Untrained users may have difficulty implementing an appropriate model of parallelism for a given problem. Users of a parallel computer vision system may have to produce custom functions as they go. The complexity of MPI restricts the amount of users who could generate new definitions. This opposes the idea of an everyday use of parallel computer vision. Expertise would be required to tailor parallel computer vision to each new problem.

OpenMP is a good fit for implementing everyday parallel computer vision. Existing code requires only a small amount of modification, and parallelism does not require expertise in parallel techniques. The scalability issues associated with shared memory parallelism are unlikely to be prominent in everyday systems.

3 Algorithms

In this section, the algorithms used to implement object recognition are discussed. These are broken up into three main topics; colour analysis, shape analysis and feature recongition. Region detection and line width recognition, tools within colour analysis, are also given a section. They are require more indepth discussion that the other topics in colour analysis. Special focus is given to the algorithms that will be used for solving Where's Wally puzzles.

3.1 Colour Analysis

The ability to analyse the colours of an image is critical for object recognition. Colour extraction is one of the primary ways of analysing the information contained by colour. Manipulation of pixels, such as blurring or sharpening an image, is also extremely useful.

3.1.1 Colour Extraction

Extracting specific colours is one of the most important techniques in colour analysis. In raster images, extracting shades of the primary colours (red, green and blue) is simple. Colours are stored as combinations of these colours, so extracting the specific values is simple.

For non-primary colours, a technique is needed to clearly describe them. A commonly used notation is the hexadecimal format, also known as a hex triplet. Values are stored in the form '#RRGGBB', where RR, GG and BB are the hexadecimal values for the red, green and blue components of a pixel. Examples of this can be seen in table 3.1.

Colour	Common Name	Hexadecimal
⬜	White	#FFFFFF
⬛	Black	#000000
🟥	Crimson	#DC143C
🟩	Sea Green	#2E8B57
🟪	Orchid	#DA70D6

Table 3.1: An example of common colours with hexadecimal outputs

Using the hexadecimal notation, colours can be logically defined. By finding the regions of the image that satisfy the red, green and blue colour values described in a hexadecimal value, specific colours can be located within an image.

This extends to searching for ranges of colours. This is useful, because images are regularly not in blocks of a single colour. For example, gradients (one colour gradually shifting into an other) are commonplace in most images. Attempting to hilight gradient with a single colour would reveal a small subsection of the desired area.

The ability to search ranges of colours enables the user to search more generally. If the precise colours used changes between images, using a colour range extends the generality of the function it is used in.

3.1.2 Blur

Another useful tool in colour analysis is blurring. Blurring allows the user to merge nearby colours together, or to help mitigate the effect of compression artifacts.

Two common methods of blurring are Median and Gaussian blurring. A median blur[22] causes a given pixel to take the median value of it's neighbouring pixels. The 'size' of the blur describes the radius within which the median is calculated. Median blurs are often used for reducing the noise in an image, such as a photo taken in low light.

The Gaussian blur uses the Gaussian function (equation 3.1) to develop a weighted average of neighbouring pixels.

$$G(x,y) = \frac{1}{2\pi\sigma}e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.1}$$

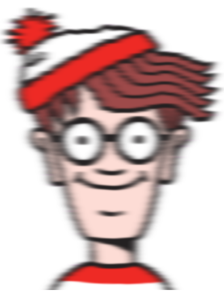
This produces an image that appears to be blurred more smoothly than with the median blur, see figure 3.1. This is useful for Laplacian edge detection schemes, which is sensitive to noise.

(a) An image with no blur

(b) 4 pixel blur

(c) 8 Pixel blur

(d) 16 pixel blur


(e) Horizontal blur

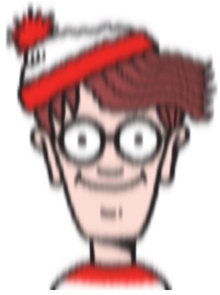

(f) Vertical blur

Figure 3.1: Gaussian blur of an image

For this project, blurring will be done with Gaussian blurs. Features with fine detail, as regularly occurs in Where's Wally? puzzles, can lose important information with the median blur. For example, a line that is a single pixel wide could be completely removed by a median blur. In Where's Wally puzzles, a fine detail could be critical to locating Wally.

The Gaussian method avoids this, as a pixel in strong contrast with it's neighbour will maintain some level of contrast. As seen in section

0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.36	0.00	0.00
0.00	0.36	1.00	0.36	0.00
0.00	0.00	0.36	0.00	0.00
0.00	0.00	0.00	0.00	0.00

a An image to be sharpened

0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	1.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00

b An ideally sharpened image

0.00	0.00	0.13	0.00	0.00
0.00	0.11	0.30	0.11	0.00
0.13	0.30	0.62	0.30	0.13
0.00	0.11	0.30	0.11	0.00
0.00	0.00	0.13	0.00	0.00

c The blurred image

0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.17	0.00	0.00
0.00	0.17	1.00	0.17	0.00
0.00	0.00	0.17	0.00	0.00
0.00	0.00	0.00	0.00	0.00

d The sharpened image

Table 3.2: The process of sharpening an image. The arrays represent images in the process of sharpening. The sharpened image is obtained by subtracting the blurred array from the original with equal weighting. The result was then rescaled

3.2, Gaussian blurs are also useful for blending nearby masks. This is because Gaussian blurs will rapidly 'spread' binary values that are grouped together.

Blurring is not a trivially parallel task. For each pixel that is to be blurred, a large group of neighbouring pixels are required. Pixels on the border can often develop visual errors known as artifacts. Decomposing the image into subimages could produce artifacts on interior borders. To avoid this, each subimage would need a halo of data that it could read but not write to. The size of the halo would be dependent on the size of the blur being performed.

3.1.3 Sharpen

The sharpen techinque is another useful tool for colour analysis. Sharpen is intended to allow a user to make an image more focused[23]. This is very useful for when details in an resized image have been blurred or compressed.

Sharpening an image is, in many ways, the opposite of blurring an image. Noting this, the most common method of sharpening, known as "unsharp mask" creates a blurred copy of the image, and remove it from the original using a weighting. Table 3.2 shows this process numerically. With an appropriate choice of weighting, the ideally sharp image can be obtained using the unsharp mask.

The unsharp mask method is sufficiently simple to implement for an everyday user.

Sharpening, like blurring, requires some message passing when parallelised. Though subtracting the weighted blur can be done independently, blurring subimages will lead to artifacts at the borders on the main image.

3.2 Region Detection

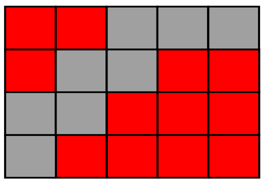
Region detection is a tool that allows the discrete labelling of connected pixels in a binary image. Although this may be obvious to the human eye, a binary mask does not immediately contain the information needed to deduce connected regions computationally.

3.3 Naive Solution

A computationally naive way to do detect regions is shown in Figure 3.2. Each non-zero pixel is assigned a unique integer value. Each pixel in the image is assigned the maximum value of itself and it's four nearest neighbours. This is repeated until the image is stable and no no pixel changes value. This method takes the maximum value of neighbouring pixels, and ignores non-zero pixels, so maximal values can not be spread outside of a region's boundary.

Within a region, it is evident that every pixel will have the value of the maximum pixel within that region. As each pixel has a unique value, it follows that each regional maximum must also be unique. The number of regions can be found by counting the unique values in the matrix. Similarly, properties of a region can be calculated by analysing pixels that have the same value as each other. For example, the mean position of the pixels within the region, the size of the region, and the bounding box around the region can be calculated this way.

This method is suitable for shared memory parallelism, but is not efficient. To determine if a pixel should have it's value changed, 4 other pixels must be read. This must be done an indeterminate number of times, as stability is dependent on the arrangement of the mask.


(a) Input A

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

(b) B, such that $B_{ij} = iY + j$

1	2	0	0	0
6	0	0	9	10
0	0	13	14	15
0	17	18	19	20

(c) $C^0_{ij} = A_{ij} \cdot B_{ij}$

6	2	0	0	0
6	0	0	14	15
0	0	18	19	20
0	18	18	19	20

(d) $C^{t+1}_{ij} = max\left(C^t_{ij}, nbr(C^t_{ij})\right)$ for $C^t_{ij} \neq 0$

6	6	0	0	0
6	0	0	20	20
0	0	20	20	20
0	20	20	20	20

(e) The final result, when $C^{t+1} = C^t$

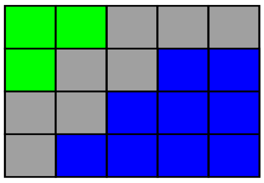

(f) A colour representation of the regions

Figure 3.2: Steps in a naive region detection algorithm

3.3.1 Flood Fill

Flood fill[24] is an algorithm commonly used in 'paint' programs for filling an indicated region with colour. The algorithm can be implemented in several ways. The simplest to understand is the recursive version.

The recursive algorithm is initialised with a starting pixel. From this starting point, the algorithm scans the neighbouring pixels in a specific order. If the neighbouring pixels have a non-zero value, then a new function is started, with that neighbour as the new starting point. When all neighbours have been scanned, the function returns. The region has been filled when the original function returns. The flood fill algorithm can also be implemented in a queue based system (last in, first out), which avoids recursion. A graphical example of the flood fill algorithm can be seen in figure 3.3.

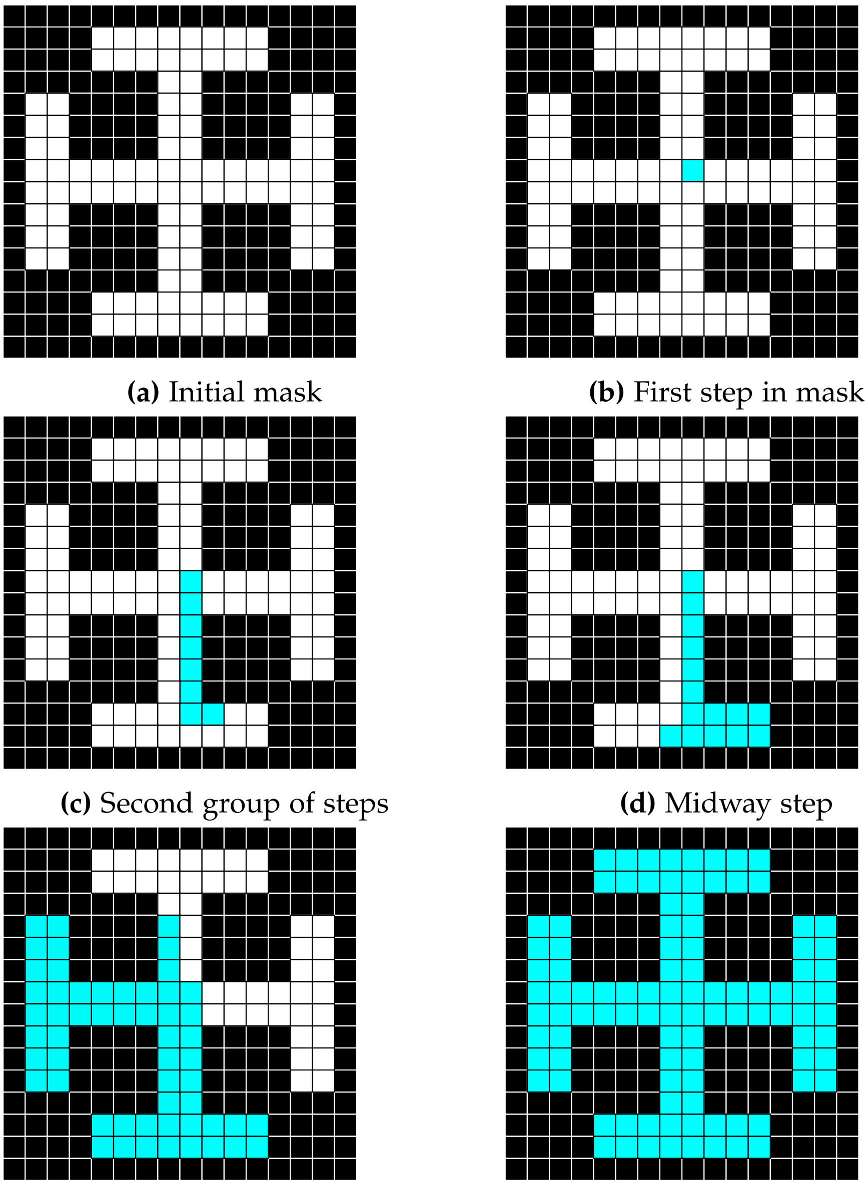

(a) Initial mask
(b) First step in mask
(c) Second group of steps
(d) Midway step
(e) Another midway step
(f) Final step

Figure 3.3: Various steps in the flood fill algorithm

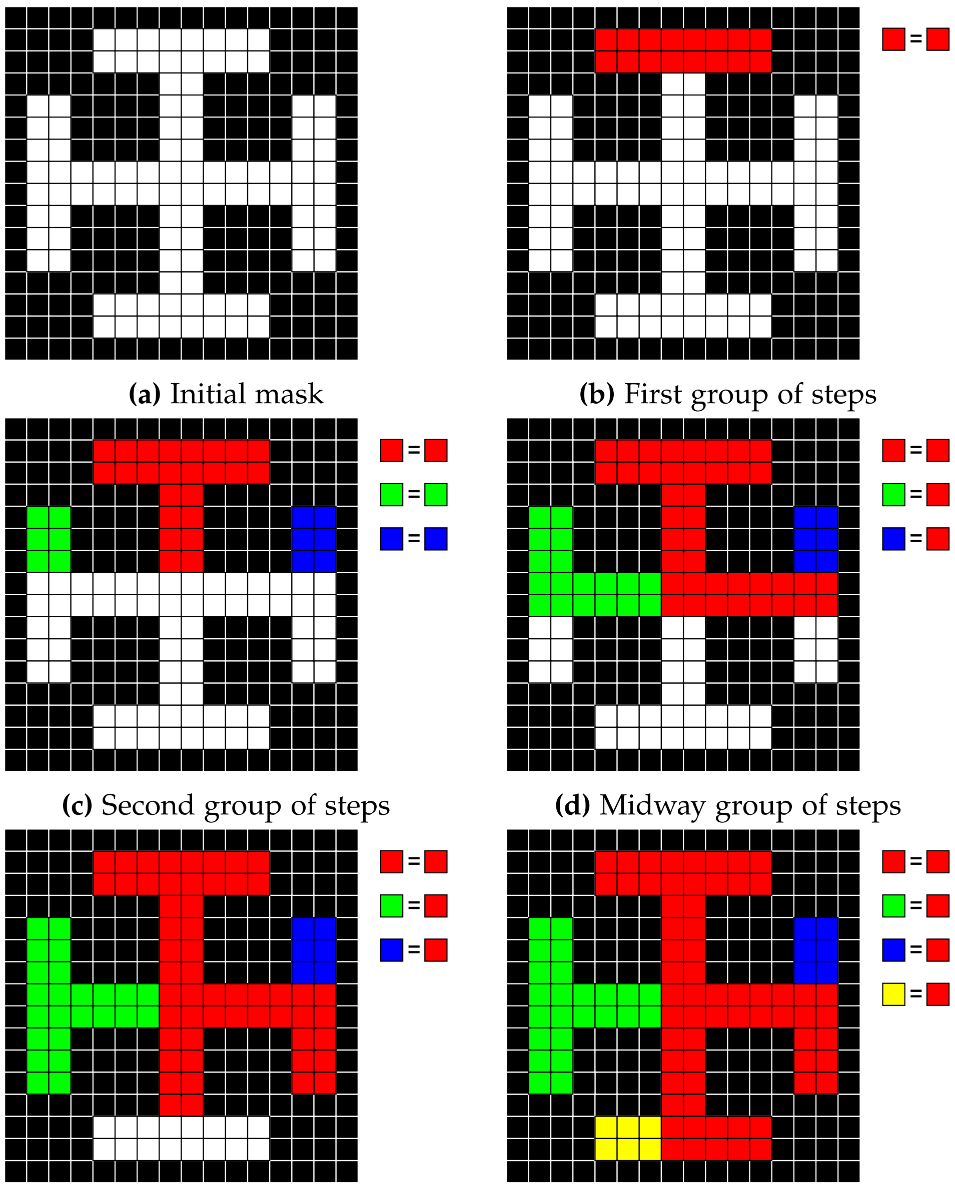
The flood fill algorithm is useful when a user is able to give a hint for a good starting point. When trying to detect all the regions in a mask, the starting point needs to be automatically calculated.

Flood fill can be parallelised, but needs halo data to be passed. When the image is decomposed, regions could be split along a subimage border. This means that halo data must be analysed to match regions that are equivalent.

3.3.2 Connected Component Labelling

Lifeng [25] describes the method of connected-component labelling.

The algorithm, seen graphically in figure 3.4, works as follows; A pixel $p_{i,j}$ that has zero-valued or non-existent neighbours $p_{i-1,j}$ and $p_{i,j-1}$ is assigned a new temporary label. Otherwise, if the upper pixel $p_{i,j-1}$ is non-zero, it has a label (thanks to the ordering of the algorithm). The pixel $p_{i,j}$ is then assigned with the label of $p_{i,j-1}$. If the label of $p_{i,j}$ has not been set, then it gets the label of the non-zero left pixel $p_{i-1,j}$. If $p_{i-1,j} = p_{i,j-1}$ but they do not have the same label, then a label equivalence is established. Once the matrix has been traversed, equivalent labels are merged, and all regions have been detected.



(a) Initial mask **(b)** First group of steps

(c) Second group of steps **(d)** Midway group of steps

(e) Penultimate group of steps **(f)** First iteration complete

Figure 3.4: The steps in the first iteration of the connected component labelling algorithm

This method is both simple and memory efficient enough to be suitable for parallelism in this project. The algorithm maintains correctness on decomposed images, as long as halo data is passed between the processors. To find label equivalence across threads, only a single row or column is needed to be read. Between threads, the process is one-directional; data is only required to travel north and west of the current thread. This can be seen graphically in figure 3.5. In the case of shared-memory parallelism, accessing the neccessary data is a minimal overhead. Critically, this process is largely the same as the serial version of the code. This provides a simple platform for everyday users to enable parallelism.

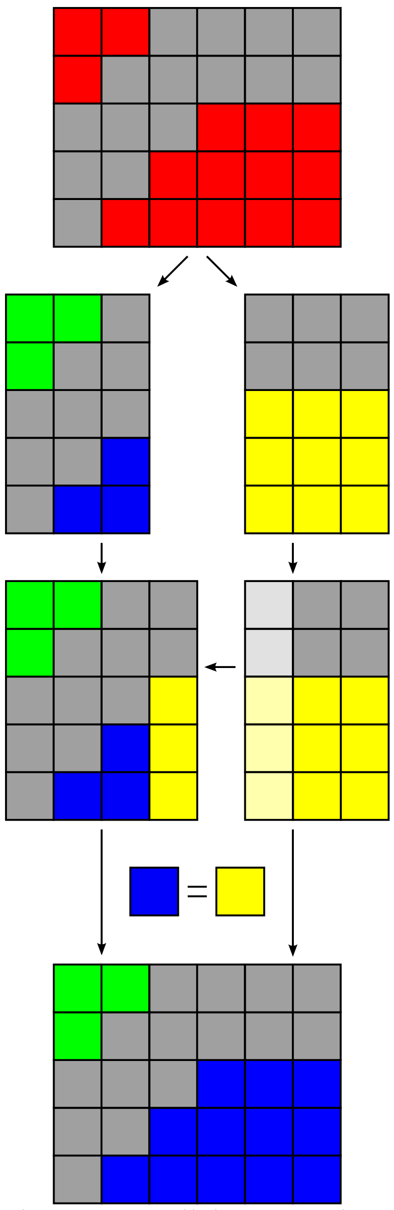


Figure 3.5: A diagram explaining parallel region detection. The initial image is split into two sections, and each region inside the subimage is given a unique id. One thread sends halo data to the other, which then calculates what threads are the equivalent. The main image is then put back together, with the parallel region equivalences applied.

Connected component labelling will be used for region detection in the Where's Wally? solver. It maintains simplicity under parallelism when compared to the flood fill algorithm. It is considerably more efficient than the naive solution.

3.4 Line Width Estimation

In cartoon images, characters and objects are regularly bounded by black lines. Within a given image style, these lines can be taken as a reference point for the scale of the image. For example, take one image with lines that are 4 pixels wide and another with lines that are 2 pixels wide. It can be inferred that the first image is twice the scale of the second. This can be used to put an upper limit on how large a match between an

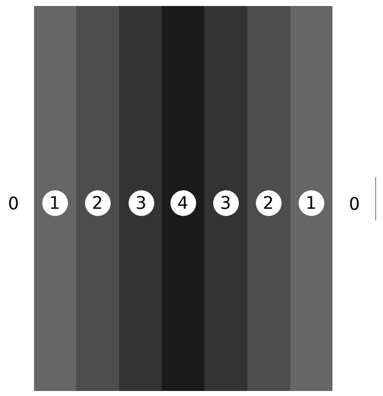
Object and a Scene can be. Estimating the line width can be done using a combination of a few techniques.

For images with no intesecting lines, only a few steps need be followed.

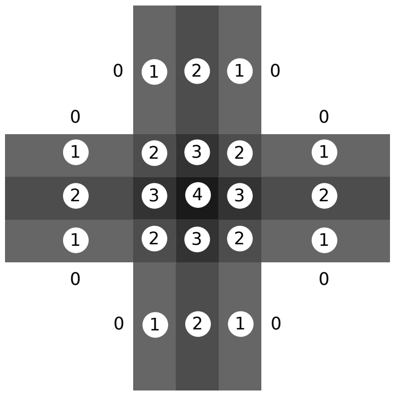
- Produce a mask of all the black in the image.
- Count the distance to the nearest zero-valued pixel (called zero-distance hereafter). This can be done with the OpenCV `distanceTransform` function.
- Find the maximum zero-distance in the image.

In the example image, figure 3.6(a), a 7 pixel wide line has a maximum zero-distance of 4. For an 8 pixel wide line, it follows that the zero-distance would be 4. The maximum zero-distance can resolve line width of simple images to within 1 pixel.

For images with intersecting lines, this method will fail to produce correct results. Figure 3.6(b) shows a situation where the method fails. Two orthogonal 3 pixel wide lines cross over each other, creating a maximum zero-distance of 4. This would indicate that the line width of this image is 7 or 8, which is wrong by nearly a factor of 3. Images with large regions of black shading would cause this method to fail with larger errors.



(a) A representation of a line that is seven pixels wide, broken down into reach region with different zero-distances



(b) Two intersecting lines, that are 3 pixels wide. The point of intersection produces a maximum zero-distance of 4

Figure 3.6: Two examples of lines with their zero-distances calculated.

A more complicated method, described in section 3.4.1, relies on two statistical properties of the image; the average zero-distance of the pixels and standard deviation of each pixel from that average.

3.4.1 Average Distance to a Zero Valued Pixel

Inspecting figure 3.6(a), it is clear that the average distance is dependent upon something approaching a sum of incremental integers. Lines with even and odd widths will differ slightly; even values have two maximum zero-distances, odd values only have one. This can be seen in equations 3.2, where PixelDistance(n) lists the zero-distances in an n pixel wide line.

$$\begin{aligned}\text{PixelDistances}(4) &= \{1,2,2,1\}\\ \text{PixelDistances}(5) &= \{1,2,3,2,1\}\\ \text{PixelDistances}(6) &= \{1,2,3,3,2,1\} \qquad (3.2)\end{aligned}$$

For a general N-pixel wide line, the sum of all values is in the range $[0, N/2]$ on one side, and $[N/2, 0]$ on the other. This has the mathematical form of a geometric series. This particular geometric series can be conveniently expressed as simple formula, described in equation 3.3.

$$\text{Sum}(N) := \sum_{i=0}^{N} i = \frac{N(N+1)}{2} \qquad (3.3)$$

Even valued widths are considered first, as this is the simplest case to analyse. This can be calculated as twice the sum of integers in the range $[0, N/2]$, as seen in equation 3.4.

$$\begin{aligned}\text{EvenSum}(N) :&= 2\ \text{Sum}(N/2)\\ &= 2\sum_{i=0}^{N/2} i\\ &= 2\frac{(N/2)(N/2+1)}{2}\\ &= N\left(\frac{N}{4}+\frac{1}{2}\right)\\ &= \frac{N^2+2N}{4} \qquad (3.4)\end{aligned}$$

For an odd-valued N, the sum is in the range $[0, (N+1)/2]$ and $[(N-1)/2, 0]$. This can be reduced to twice the sum of integers in the range $[0, (N-1)/2]$ plus $(N+1)/2$, shown in equation 3.5.

$$\begin{aligned}\text{OddSum}(N) :&= \frac{N+1}{2} + 2\ \text{Sum}((N-1)/2)\\ &= \frac{N+1}{2} + 2\sum_{i=0}^{(N-1)/2} i\\ &= \frac{N+1}{2} + 2\frac{((N-1)/2)((N-1)/2+1)}{2}\\ &= \frac{N+1}{2} + \frac{N-1}{2}\frac{N+1}{2}\\ &= \frac{N+1}{2}\frac{N+1}{2}\\ &= \frac{N^2+2N+1}{4} \qquad (3.5)\end{aligned}$$

These values are very close, differing by only $\frac{1}{4}$ of a pixel. When these formulas are used to calculate the average zero-distance, this difference further decreases. Equation 3.6 demonstrates this. The estimation formula loses reliability for $N < 1$. As pixels are positioned on a discrete grid, the only possible value for $N < 1$ is zero, which represents no line.

$$\begin{aligned}\text{AvgDist}(N) := &\begin{cases}\frac{\text{EvenSum}(N)}{N} = \frac{1}{4}\left(N+2\right) \text{ if } N \text{ even} \\ \frac{\text{OddSum}(N)}{N} = \frac{1}{4}\left(N+2+\frac{1}{4N}\right) \text{ if } N \text{ odd}\end{cases} \\ = &\frac{N+2}{4} + \mathrm{O}(N^{-1})\end{aligned} \tag{3.6}$$

For large values of N, the average distance approaches the EvenSum formula. This can be seen in figure 3.7. The background shading represents the values bounded by the values of OddSum and EvenSum. The figure also shows that the maximum error for any given value of N is at most a quarter of a pixel.

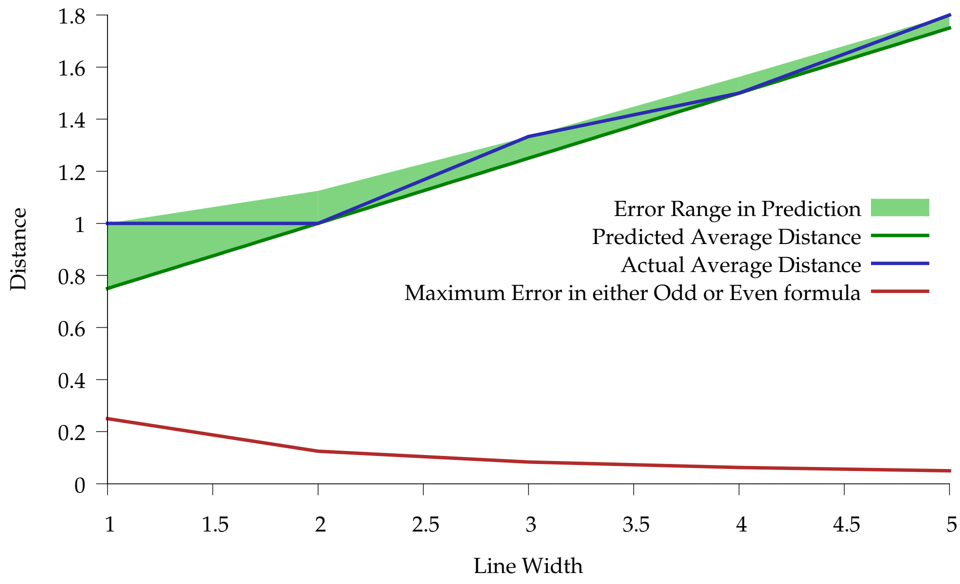


Figure 3.7: Graph showing the estimation of the average distance to a zero valued pixel using the Odd and Even formulae.

Equation 3.6 shows that there is an approximate linear relationship between the average zero-distance and the width of the line drawn. It is possible to invert this equation. This produces a formula, equation 3.7, to calculate line width from the average zero-distance.

$$\text{LineWidth}(avg) := \text{AvgDist}^{-1}(avg) = \frac{avg - 0.5}{0.25} \tag{3.7}$$

Using this equation, the line width can be approximated using an easily calculable property of the image; the average distance to a zero-valued pixel. The maximum error in equation 3.6 was a quarter of a pixel. Equation 3.7 accordingly has an maximum error of 1 pixel (at $N = 1$). The formula has a minimum error of 0 for all even valued widths. An unusual effect of this is that the method cannot determine a difference between a line of width 1 pixel and a line of width 2. This means that this method can be used only as a upper bound on the scaling between two images.

3.4.2 Standard Deviation in Distance to Zero Valued Pixels

The line width can be deduced from another calculable property of the system; the standard deviation. Standard deviation, here, is defined to be the normalised deviance of all pixels from the average position of each pixel, x_i.

$$\text{StdDev}(N) := \sqrt{\frac{\sum (x_i - \text{AvgDist}(N))^2}{N}}$$

The sum, here called Deviance within the square root can be simplified.

$$\text{Deviance}(N) := \sum_{i=0}^{N} (x_i - \text{AvgDist}(N))^2 = \begin{cases} \mathit{DevEven}(N) \text{ if } N \text{ is even} \\ \mathit{DevOdd}(N) \text{ if } N \text{ is odd} \end{cases}$$

As before, x_i will take values from $[0, .., N, ...0]$, and AvgDist(N) is defined in equation 3.6. We will show the method for finding the standard deviation for even valued widths, odd values follow from above. We write the even equation as

$$\begin{aligned}
\text{DevEven}(N) &= 2\sum_{i=0}^{N/2} \left(i - \frac{N+2}{4}\right) \\
&= 2\sum_{i=0}^{N/2} \left(i^2 - 2i\frac{N+2}{4} + \frac{(N+2)(N+2)}{16}\right) \\
&= 2\left(\sum_{i=0}^{N/2}(i^2) - 2\frac{N+2}{4}\sum_{i=0}^{N/2}(i) + \sum_{i=0}^{N/2}\left(\frac{(N+2)(N+2)}{16}\right)\right) \\
&= 2\left(\sum_{i=0}^{N/2}(i^2) - \frac{N+2}{4}\frac{N(N/2+1)}{2} + \frac{N}{2}\frac{(N+2)(N+2)}{16}\right)
\end{aligned} \tag{3.8}$$

The sum in equation 3.8 is, as with equation 3.3, easily expanded using geometric identities. In this case, the formula is described in equation 3.9.

$$\sum_{i=0}^{N} i^2 = \frac{1}{6}N(N+1)(2N+1) \tag{3.9}$$

Continuing the expansion of DevEven(N);

$$\begin{aligned}
\text{DevEven}(N) &= \frac{1}{3}\frac{N}{2}(N/2+1)(N+1) - \frac{N(N+2)(N+2)}{4} + \frac{N(N+2)(N+2)}{16} \\
&= \frac{1}{48}\left(N^3 - 4N\right)
\end{aligned} \tag{3.10}$$

Using this formula for DevEven, the even width standard deviation becomes

$$\begin{aligned}
\text{StdDevEven} &= \sqrt{\frac{\text{DevEven}(N)}{N}} \\
&= \sqrt{\frac{N^3 - 4N}{48N}} \\
&= \frac{N}{4\sqrt{3}} + O(\sqrt{N})
\end{aligned} \tag{3.11}$$

Using similar calculations for odd widths, StdDevOdd has a value of

$$\begin{aligned}\text{StdDevOdd} &= \sqrt{\frac{\text{DevOdd(N)}}{N}} \\ &= \sqrt{\frac{N^2}{16N} + \frac{N^2 - 3N + 2}{48N}} \\ &= \frac{N}{4\sqrt{3}} + O(\sqrt{N}) \qquad (3.12)\end{aligned}$$

As these functions are equivalent (up to $\mathrm{O}(\sqrt{N})$), we define the Standard Deviation approximation as

$$\text{StdDev}(N) := \frac{N}{4\sqrt{3}} \qquad (3.13)$$

Figure 3.8 shows the comparison of these formulae with actual data. The data and approximate standard deviation are well bounded inside the Odd and Even standard deviations. The maximum error that can be expected from the approximation is less than 0.3 pixels. This is at the boundary case of 2 width pixels. At all other points, the error is a small fraction of the actual value.

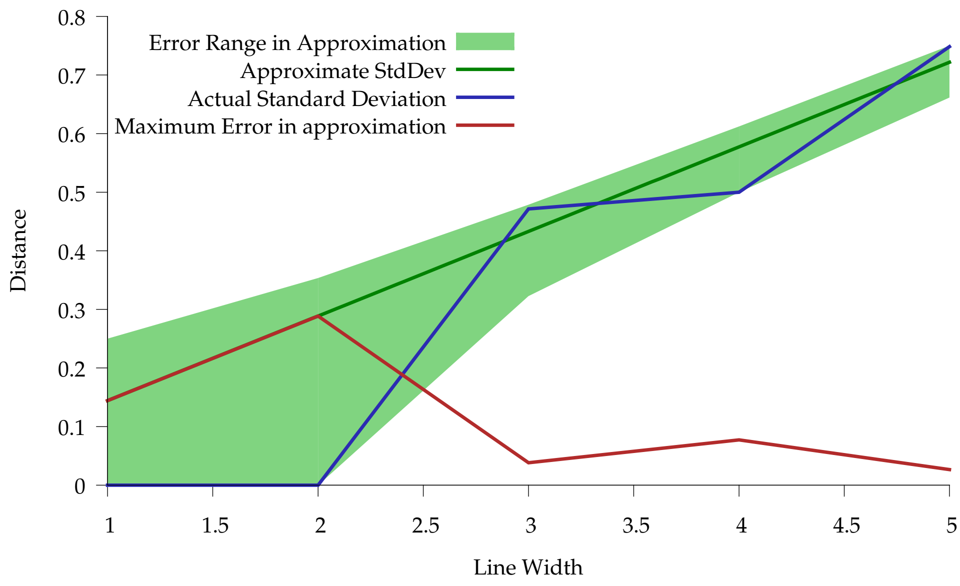


Figure 3.8: Graph showing the relationship between line width and the Standard Deviation of pixel distance to zero valued pixels.

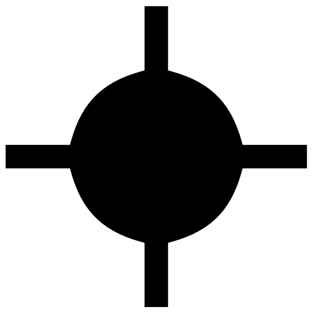
Again, as a linear formula has been produced, the relationship can be reversed, to get line width from standard deviation.

$$\text{LineWidth}(stddev) := \text{StdDev}^{-1}(stddev) = 4\sqrt{3} \cdot stddev \qquad (3.14)$$

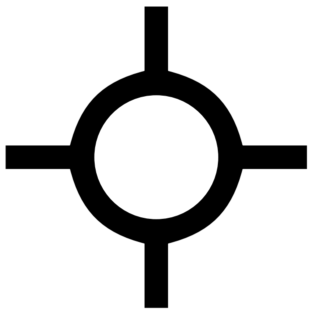
Equation 3.14 can not give an exact estimation of line widths; it will overestimate odd widths and underestimate even widths. For the correct standard deviation, it will give the correct line width to within ± 1 pixel, for all values. Thus the standard deviation method can put an upper bound on the scaling between two images.

3.4.3 Combining the Methods

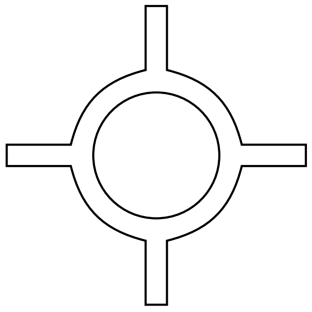
Each of these methods is flawed when introduced into an image with overlapping lines, as in figure 3.6(b). This can be compensated for by successively removing the pixels with the largest distance from the map. At some point during this removal, it is expected that a "sane" mask will be produced. A sane mask, in this case, is a mask that displays only the boundary lines of an image. For example, a filled in square would be reduced to the 4 border lines that define it's shape. A visual example can be seen in figure 3.9. The sane mask should produce the most correct line width estimations from either formula. Determining which mask is the sane one is not immediately obvious, but can be deduced using a combination of both methods.



(a) The original image, with an equation distorting solid block of black in the center



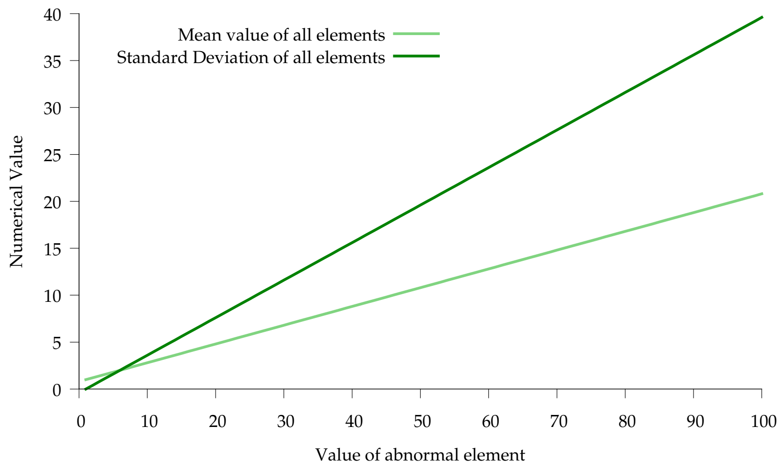
(b) The sane mask, replacing solid circle in the center with it's border



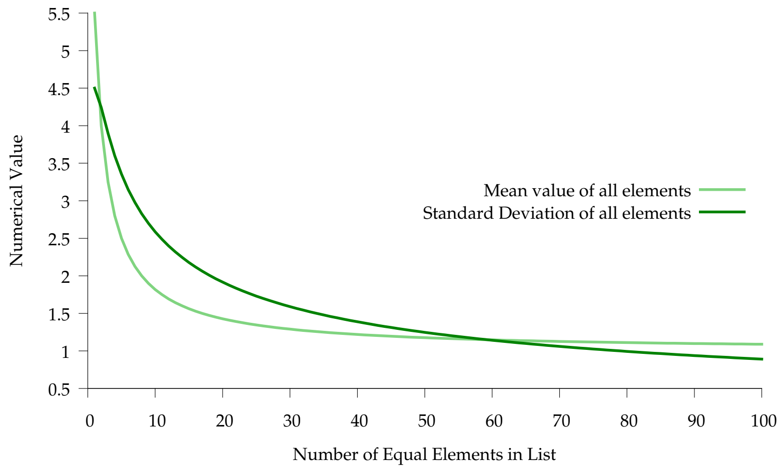
(c) The image after too many maxima have been removed, which would also distort the equation

Figure 3.9: A representation of the removal of the highest zero-distance pixels, to reveal a sane mask

Each method reacts differently to the removal of the largest zero-distance pixels. This can be seen in figure 3.10, the mean and the standard deviation have very different responses. These differences allow a combination of the two methods; only when they agree can a mask be described as sane.



(a) Changing the value of a single abnormal element in a list



(b) Changing the ratio of normal data to abnormal data

Figure 3.10: The response of mean and standard deviation to various types of erroneous data

It is unlikely a perfectly sane mask will be located. In this case, the mask with the minimal difference between the two estimations of line width will be the sane mask. The average of the two methods will be the line width estimation used. The line width is reliable to the nearest integer, rounded up, except in the case that the line width is 1 pixel. This is because a 1 pixel width line is indistinguishable from a 2 pixel width line, using the mean and standard deviation. To demonstrate this, we define Z_n as the list of zero distances of each pixel in the slice of an n wide pixel.

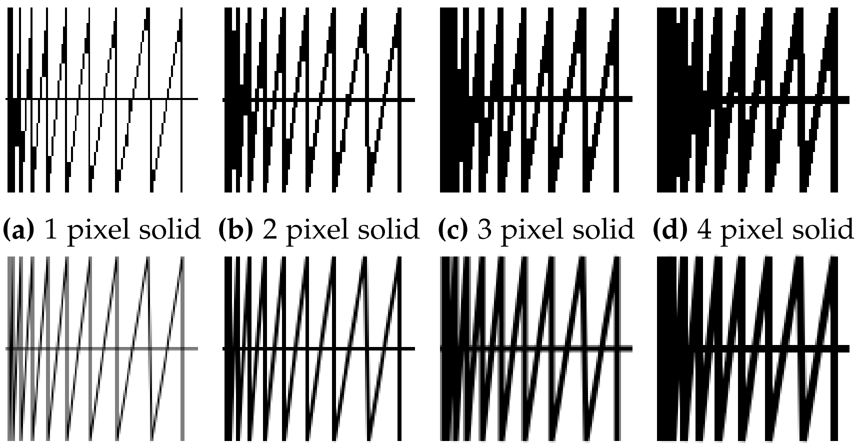
$$Z_1 = \{1\},\ Mean(Z_1) = 1,\ StdDev(Z_1) = 0$$

$$Z_2 = \{1, 1\},\ Mean(Z_2) = 1,\ StdDev(Z_2) = 0$$

This being the case, a 1 pixel line may be interpreted to be a 3 pixel line. Using this method, a maximum line width can be estimated. Comparing these line widths, the scaling between two cartoon images can be approximated.

3.4.4 Testing

This method was tested with an image with several thickness. The base image contains vertical lines that are increasingly further apart from each other, with lines connecting each point. This can be seen in figure 3.11. There are two types of test, solid lines, which have clearly defined boundaries, and aliased lines, which are more realistic. The results of using this method of each of these images is listed in table 3.3. The line widths are almost all correctly predicted to within 1 pixel, with the exception of 1 pixel on a solid line. As discussed earlier, this has been anticipated.



(a) 1 pixel solid **(b)** 2 pixel solid **(c)** 3 pixel solid **(d)** 4 pixel solid

(e) 1 pixel aliased **(f)** 2 pixel aliased **(g)** 3 pixel aliased **(h)** 4 pixel aliased

Figure 3.11: Images used to test `estimate_black_line_thickness`. They are designed to stress estimation, with lines that overlap regularly. The top row are solid pixels, with clear boundaries. The bottom row show aliased lines, which are more realistic.

Line Width	Solid Line Predicted Width	Aliased Line Predicted Width
1	2.42688	1.31487
2	2.74886	2.75531
3	3.05320	2.87646
4	4.21705	4.33824

Table 3.3: A comparison of the line width analysis technique on images from 3.11 All but one of the values are within 1 pixel of the actual value. The anomalous 1 pixel result can be mistaken as a 2 pixel width. The technique cannot guarantee distinction between 1 and 2 pixels.

3.4.5 Parallelism

Analysing the line width is a parallelisable task. Calculating the zero-distance of each pixel in the map needs to be done in serial. This is because bisecting a line could make a large change to the standard deviation, if there aren't many lines. The bisection can't easily be avoided without knowing the width of the lines, so using halo data is not viable. However, splitting the mask after the zero-distance has been calculated is parallelisable. The average can be calculated with a reduction operation, as can the summation of the standard deviation. The overhead of using

OpenMP for small images is likely to reduce usefulness of parallelism. Large images may benefit from this.

3.5 Shape Analysis

Shape analysis is the computer vision technique used to analyse shapes that are contained in a binary mask.

Suzuki[26] discusses two closely related techniques for border following. This is the method of 'following' edges that are defined in a binary image. By following these edges and recording the path taken, all shapes defined within the mask can be located. As the recorded path contains a large number of points, it may be preferable to simplify the shape before analysis.

The Ramer-Douglas-Peucker algorithm[27][**?**] can be used to approximate curves. This algorithm removes points on the curve that don't sufficiently alter the shape of the line. The algorithm, which is recursive, can be seen below

1. Begin with first and last points on a curve, A and B.
2. Locate the point, C, that is furthest from the the line that joins the two points.
3. If it is greater than ϵ away from the line, then it contributes importantly to the shape of the line, and is kept. Otherwise the point does not contribute, and can be removed.
4. The curve AB is now split into two curves, AC and BC. Repeat steps 1-4 on these curves until all curves have been approximated

This function is computationally expensive, as it is recursive and requires large numbers of perpendicular distance calculations for each point calculation. The complexity of this algorithm is, in worst case scenarios, O(n^2)[28]. For large images, which would contain large numbers of complex shapes, this could dramatically slow the runtime.

OpenCV has an inbuilt function, `findContours`, which implements Suzuki's border following. It also has an implementation of the Ramer-Douglas-Peucker algorithm, `approxPolyDP`.

3.6 Feature Detection

There exists a large number of algorithms for locating features[1][30][31][32]. Two of the most prominent algorithms, SIFT and SURF, will be analysed in detail. These were considered more thoroughly than the others, because OpenCV provides implementations that are ready to use.

3.6.1 Scale-Invariant Feature Transform

Scale Invariant Feature Transform (SIFT), as discussed in section 2.3.1, is an algorithm that detects keypoints in an image. These keypoints are chosen to be invariant under changes in scale, translation and rotation. They are also designed to be partially invariant to affine changes and varying illumination.

The keys contain feature vectors, which describe the area around them. This algorithm is very reliable at finding a given sub-image within a larger image. However, some of the techniques required to produce reliable results require unexpected amounts of memory. To create a scale-space version of the image, four versions of the image must be produced, one of which is scaled to be twice the size of the original. This is not a problem when analysing a single image, or analysing multiple images linearly. However, if the algorithm is used for analysing multiple images concurrently, memory constraints would limit it's efficiency. This can be seen with Zhang's parallel implementation of SIFT [19], where the scale-space creation is one of the areas the program spends most time on. This paper shows reasonable parallel efficiency, but only has results up to 32 cores. The paper only solves 5 images at a time, which gives a 7 times speed-up over an optimised version of SIFT. This number of images is not large enough to show the true limits of shared memory parallelism.

3.6.2 Speeded Up Robust Features

Speeded-Up Robust Features (SURF), developed by Bay et. al.[30], offers a similarly robust solution, but with greater efficiency. This algorithm replaces Laplacian of Gaussian filters used in SIFT with box filters, which calculate in constant time, once an integral image has been produced. It has greater potential for parallelism than SIFT; calculating versions of the image for the scale space are independent of the previous level and can be done in parallel.

4 Implementation

4.1 Red And White

Regions with red and white stripes can be found using a few simple techniques. Here, we find two masks (arrays with binary values) that describe the location of sufficiently red and white regions in the image. The term sufficiently is used, as defining 'red' and 'white' is not a simple task for a computer. The function `get_colour_in_image`, discussed in section 4.2.1, covers the issues with colour definition.

Once the masks have been created, they are vertically blurred using a Gaussian blur. This is done so that when the two masks are multiplied together, there will be non-zero values found in overlapping areas. The multiplication now describes regions that have red and white regions of colour that are above each other. As seen in figure 4.2(f), the region found may not encompass the entire area desired. This is solved by blurring the located stripe, making a combined region with nearby stripes. The regions are then identified using the `find_region_from_mask` function.

4.2 Colour Analysis

The colour analysis techniques were applied as several functions.

Colour analysis is arguably the easiest of the three main techniques to implement. It only requires that an image be accessible as an array of pixels, which is the common storage format for images. This project only considers Where's Wally images, with pixels that are accessed in the RGB format. This means that each pixel contains information on how much Red, Green and Blue (RGB) to show. This is in contrast to HSV, or hue, saturation and value. Hue represents the pixel's position on a colour wheel, saturation represents it's vibrancy and value is a measure of it's brightness. RGB maps easily to HSV, but in cartoon images, there isn't generally a large change in an objects lightness level. It is easier to work with RGB images, but the methods described below work for HSV.

What follows is a list of functions and search patterns that utilise colour analysis.

4.2.1 Function: `get_colour_in_image`

usage `get_colour_in_image(image, colourA, colourB, rg,gr,rb,br,bg,gb)`

image The input image, as loaded by OpenCV.

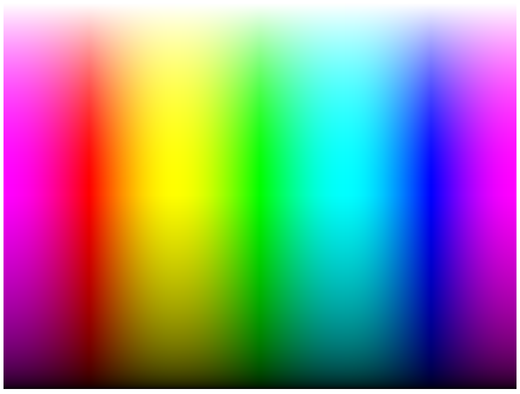
colourA,colourB Strings describing the limits of colours to allow, in the form "#RRGGBB"

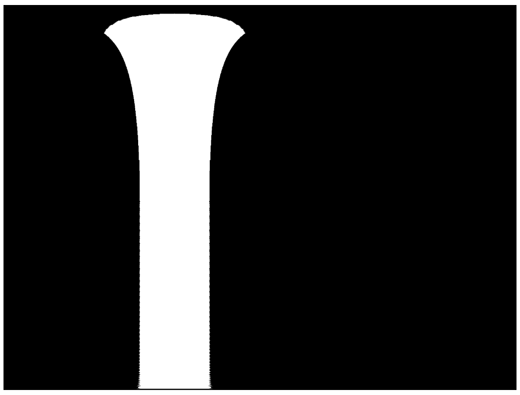
rg,gr,... The allowed ratio of each colour. For example, $R >= rg \times G$, and $G >= gr \times R$.

returns Binary mask the same size as the input image, with values of 255 if pixel was within the required values, and 0 otherwise.

This function allows a user to search the image for a specific range of colours. Moreover, the user is able to restrict the results based on the relationships between the pixels colour values. For example, a user can search for colours that are yellow by searching the entire colour space and ensuring that R and G have similar values, and that they are both greater than B;

```
yellow =
get_colour_in_image(input,"#000000","#FFFFFF",0.8,0.8,1.1,0,1.1,0)
```


(a) A typical colour spectrum


(b) The mask produced by the function. White represents a pixel that was considered to be yellow.

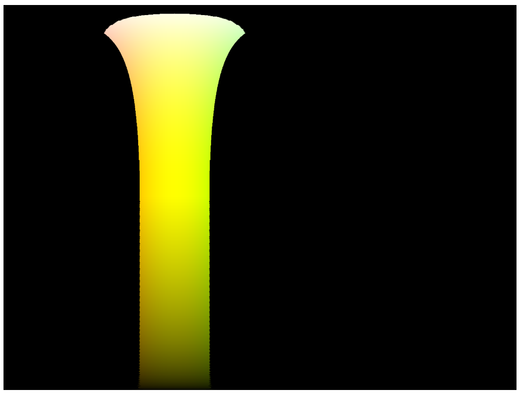

(c) The mask overlayed on the original image, revealing "yellow" colours.

Figure 4.1: Isolating yellow from an image using `get_colour_in_image`

The output of that function can be seen in Figure 4.1. The description "yellow" is not clearly defined; there is no single colour code for yellow. However, it is fair to describe yellow as some combination of red and green, with little to no blue. Without describing the specific colour code being searched for, yellow colours can be found, and results can be produced. This function allows the user to search for objects in a more robust fashion, if just for some colour.

4.2.2 Function: `get_greyscale_in_image`

usage `get_greyscale_in_image(image, low, high, tolerance)`

image The input image, as loaded by OpenCV.

low,high The range of values that should be allowed on the greyscale.

tolerance This is a finite tolerance, with non-zero values allowing nearly greyscale colours to be included.

returns Binary mask the same size as the input image, with values of 255 if pixel was within the required values, and 0 otherwise.

Although greyscale is a special case of colour, slightly different requirements were discerned for greyscale searches. Users are able to define a tolerance, which allows for colours that are not strictly grey to be included in results. For example;

```
black = get_greyscale_in_image(input, 0, 40, 20)
```

will include blacks and very dark greys, as well as very dark reds, greens and blues.

4.2.3 Function: `find_regions_from_mask`

usage `find_regions_from_mask(mask)`

image The input mask, a binary single channel matrix.

returns A list of structs, containing information about the size of the region, the average position and bounding box.

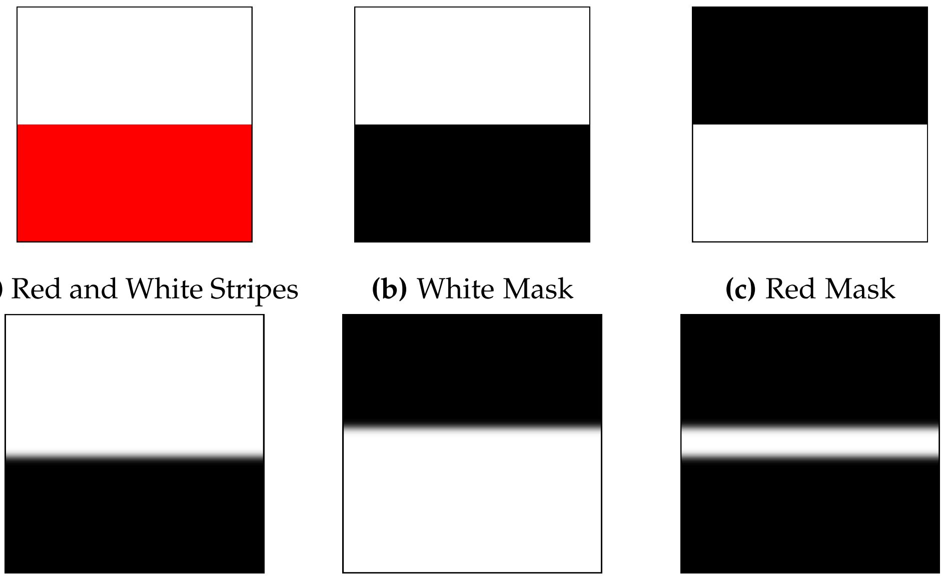
It is important to be able to discretise the results of the above functions. Something is needed that will convert the binary matrix into a list of results. This can be done by counting and collecting data about the distinct regions available in the matrix. A region here is defined as a group of pixels that are connected through non-zero nearest neighbours.

4.3 Finding Red and White Stripes

When trying to locate Wally, one of his most prevalent features is his red and white jumper. Few other elements of the puzzles use the colour scheme of red and white, and less use red and white stripes. Finding regions with red and white stripes is one of the most immediate ways to identify Wally. This can be done by following these steps, seen graphically in figure 4.2;

1. Create a mask that shows all white pixels in the image, figure 4.2(b)
2. Create a mask that shows all red pixels in the image, figure 4.2(c)
3. Blur or displace white mask in vertical direction, figure 4.2(d)
4. Blur or displace red mask in vertical direction, figure 4.2(e)
5. Multiply binary masks together, producing an area that shows where the blurs overlap,figure 4.2(f))
6. Create a binary version of multiplied mask, such that any non-zero value takes the 'on' value.
7. Blur the binary mask in the vertical and horizontal directions, to merge nearby values.

Optional. Repeat with horizontal blur and remove common elements from the vertical mask.



(a) Red and White Stripes **(b)** White Mask **(c)** Red Mask

(d) White Mask + Blur **(e)** Red Mask + Blur **(f)** Red and White blurs

Figure 4.2: Example of finding red and white striped regions

Following the optional step allows the user to prevent

4.3.1 Weaknesses

Finding Wally by his stripes is not wholly reliable. Although Wally is one of the few characters to wear red and white, he is not the only one. A good example of this is Wenda, figure 1.2(c), who wears a similar jumper. Furthermore, objects in the image regularly have a red and white motif, such as skirts, umbrellas and cakes. Without user intervention, it is hard to prioritise results based solely on this information. One method is to identify the largest regions of red and white stripes in the image as most likely to be Wally. This is often incorrect because of the presence of large red and white objects, see figure 4.3.

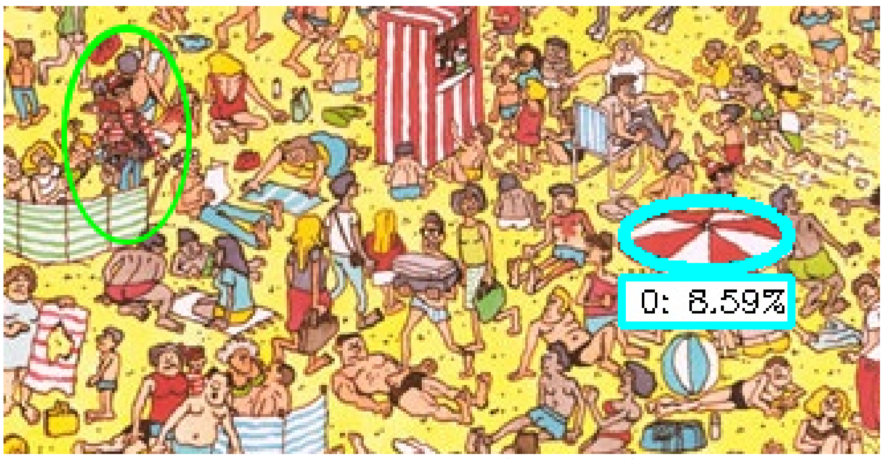
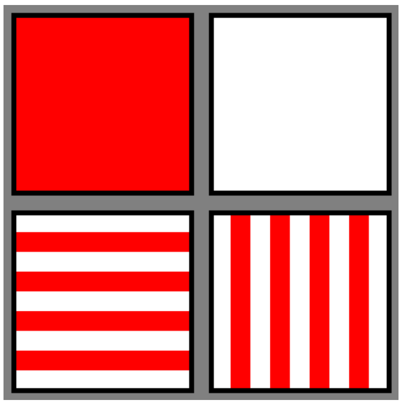


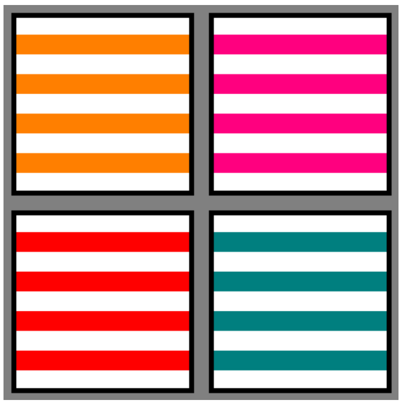
Figure 4.3: The Red and White Stripes pattern failing to find Wally. The blue ring indicates the top result (an umbrella). The green ring indicates Wally's actual location, which is not even ranked in the top 100 results.

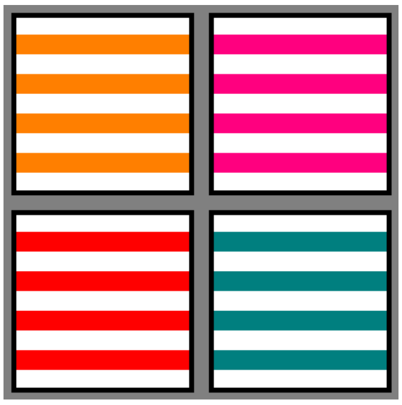
One way to mitigate this is to remove 'horizontal' matches from the vertically blurred mask. Wally is normally found standing upright, and so his jumper normally is striped vertically. Horizontally stripes are unlikely to be from potential Wallys, so they are removed from the search. This involves repeating the same techniques required to find the regions originally, but with a horizontal blur. Pixels that match the horizontal blur can be removed from the original mask, preventing their inclusion in the region detection.

Good values for the blur used for merging nearby regions are dependent on the size of Wally's stripes. If the value is too small, Wally's jumper as a whole is never located. Too large and too many incorrect regions are included in the definition of Wally's jumper. The size of Wally's stripes are not a known property, so a guess must be made as to a good value. One way of doing this is by estimating the average line width of the image, and extrapolating the size of stripes from there. For low resolution images, however, the ratio of stripe size to line width varies wildly, making it hard to estimate correct values.

4.3.2 Testing


(a) R&W test 1


(b) R&W test 2


(c) R&W test 3

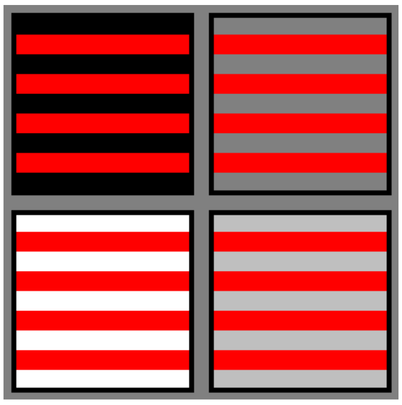

(d) R&W test 2

Figure 4.4: Test images used to check the red and white pattern

4.3.3 Parallelism

Finding Wally's stripes is very parallelisable. The image can be split into parallel subimages to find the red and white masks. The creation of the masks is a per-pixel operation, so requires no information from neighbours. This means that each parallel thread is entirely independent of it's neighbours, reducing the need for synchronisation.

The vertical blurring only needs a small amount of information from any vertical neighbour, due to the vertical blurring. Choosing to decompose the image horizontally avoids the need for any halo data to be swapped. The optional stage in the algorithm can be done by switching the decomposition, or by simply ignoring the halo data. This is because the bulk of usable information should come from the vertically blurred mask.

As with the creation of the red and white masks, the matrix multiplication is per element. This again means that the problem can be decomposed into subimages to give large speedup.

4.4 Blue Trousers

4.5 Find Glasses

4.6 Find Features

This is a search pattern that uses feature recognitions techniques to locate Wally. As noted earlier, features are areas of an image that can be used to identify it. Features that a computer can recognise are commonly not the features that a human might identify.

5 Results

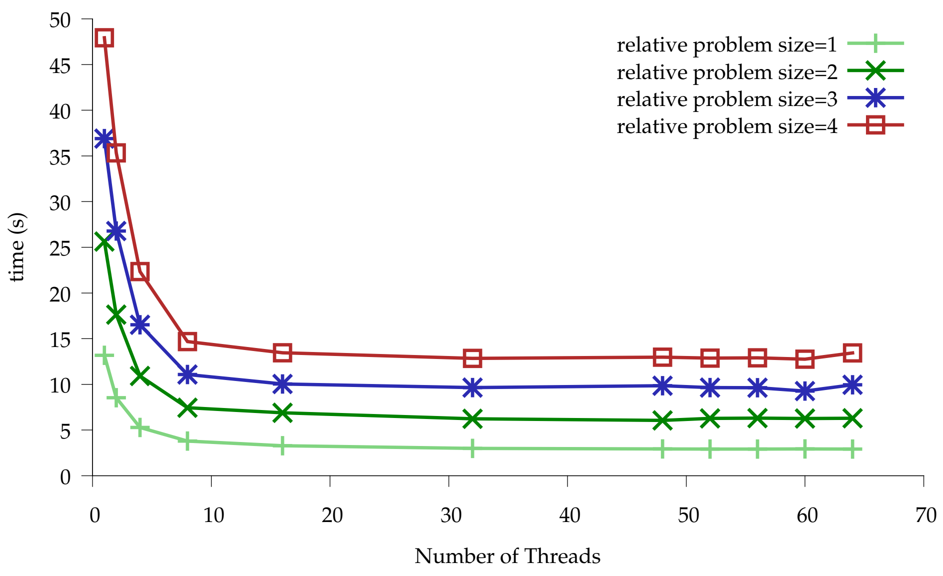
6 Line Width

7 Conclusion and Evaluation

Producing an implementation of parallel is not a simple task.

7.1 Evaluation

This project suffered from poor time management, coupled with an overestimation of how much work could be completed. This can, in part, be blamed upon lack of experience in project work like this.



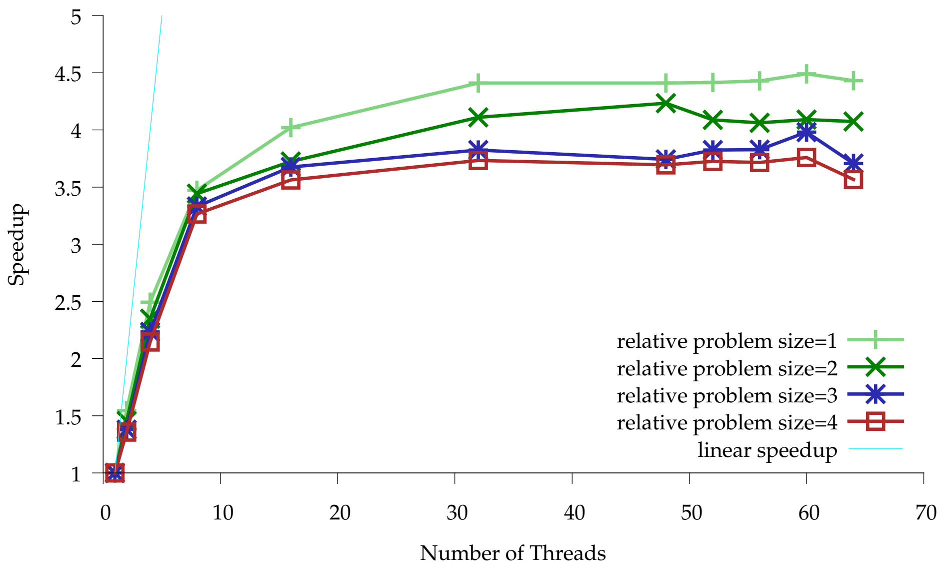


Figure 6.1: Time for 200 repetitionsSpeedup of line width analysis due to parallelism

References

[1] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2, pp. 1150–1157, Ieee, 1999.

[2] UK Missing Persons Bureau, "Missing persons data and analysis 2010-2011." `missingpersons.police.uk/en/resources/missing-persons-data-and-analysis-2010-2011`, August 2013.

[3] The Children's Society, "Still running 3 full report," 2011. `makerunawayssafe.org.uk/sites/default/files/tcs/u24/Still-Running-3_Full-Report_FINAL.pdf`.

[4] J. L. Nielsen, "Scientific sampling effects: Electrofishing california's endangered fish populations," *Fisheries*, vol. 23, no. 12, pp. 6–12, 1998.

[5] Y. Sun and R. Fisher, "Object-based visual attention for computer vision," *Artificial Intelligence*, vol. 146, no. 1, pp. 77–123, 2003.

[6] J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber, "An integrated, modular framework for computer vision and cognitive robotics research (icvision)," in *Biologically Inspired Cognitive Architectures 2012*, pp. 205–210, Springer, 2013.

[7] Microsoft, "Kinect for xbox 360," august 2013. `http://www.xbox.com/en-GB/KINECT`.

[8] V. M. Patel, R. Maleh, A. C. Gilbert, and R. Chellappa, "Gradient-based image recovery methods from incomplete fourier measurements," *Image Processing, IEEE Transactions on*, vol. 21, no. 1, pp. 94–105, 2012.

[9] L. G. Roberts, "Machine perception of three-dimensional solids," tech. rep., DTIC Document, 1963.

[10] Google, "Google glass," august 2013. `http://www.google.com/glass/start/`.

[11] K. Tanaka, "Mechanisms of visual object recognition: monkey and human studies," *Current opinion in neurobiology*, vol. 7, no. 4, pp. 523–529, 1997.

[12] D. I. Perrett and M. W. Oram, "Visual recognition based on temporal cortex cells: Viewer-centred processing of pattern configuration," *Zeitschrift fur Naturforschung C-Journal of Biosciences*, vol. 53, no. 7, pp. 518–541, 1998.

[13] OpenCV, "Opencv home website," august 2013. `opencv.org`.

[14] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'reilly, 2008.

[15] Edward Rosten, "libcvd homepage," august 2013. `http://www.edwardrosten.com/cvd/`.

[16] V. Fragoso, S. Gauglitz, S. Zamora, J. Kleban, and M. Turk, "Translatar: A mobile augmented reality translator," in *Applications of Computer Vision (WACV), 2011 IEEE Workshop on*, pp. 497–502, IEEE, 2011.

[17] A. C. Downton, R. W. Tregidgo, and A. Cuhadar, "Top down structured parallelisation of embedded image processing applications," *IEE Proceedings-Vision, Image and Signal Processing*, vol. 141, no. 6, pp. 431–437, 1994.

[18] J. Fung and S. Mann, "Openvidia: parallel gpu computer vision," in *Proceedings of the 13th annual ACM international conference on Multimedia*, pp. 849–852, ACM, 2005.

[19] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, "Sift implementation and optimization for multi-core systems," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, IEEE, 2008.

[20] H. Feng, E. Li, Y. Chen, and Y. Zhang, "Parallelization and characterization of sift on multi-core systems," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 14–23, IEEE, 2008.

[21] N. Zhang, "Computing optimised parallel speeded-up robust features (p-surf) on multi-core processors.," *International Journal of Parallel Programming*, vol. 38, no. 2, pp. 138 – 158, 2010.

[22] D. Brownrigg, "The weighted median filter," *Communications of the ACM*, vol. 27, no. 8, pp. 807–818, 1984.

[23] H. Malepati, *Digital media processing [electronic resource] : DSP algorithms using C / Hazarathaiah Malepati.* Burlington, Mass. : Newnes/Elsevier, [2010], Âl'2010., 2010.

[24] S. BURTSEV and Y. KUZMIN, "An efficient flood-filling algorithm.," *COMPUTERS AND GRAPHICS*, vol. 17, no. 5, pp. 549 – 561, n.d.

[25] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009.

[26] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, 1985.

[27] U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," *Computer Graphics and Image Processing*, vol. 1, no. 3, pp. 244–256, 1972.

[28] J. E. Hershberger and J. Snoeyink, *Speeding up the Douglas-Peucker line-simplification algorithm*. University of British Columbia, Department of Computer Science, 1992.

[29] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision–ECCV 2006*, pp. 404–417, Springer, 2006.

[30] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 10, pp. 1615–1630, 2005.

[31] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 886–893, IEEE, 2005.

[32] OpenCV, "Opencv 2.4 cheat sheet (c++)." `docs.opencv.org/trunk/opencv_cheatsheet.pdf`, august 2013.