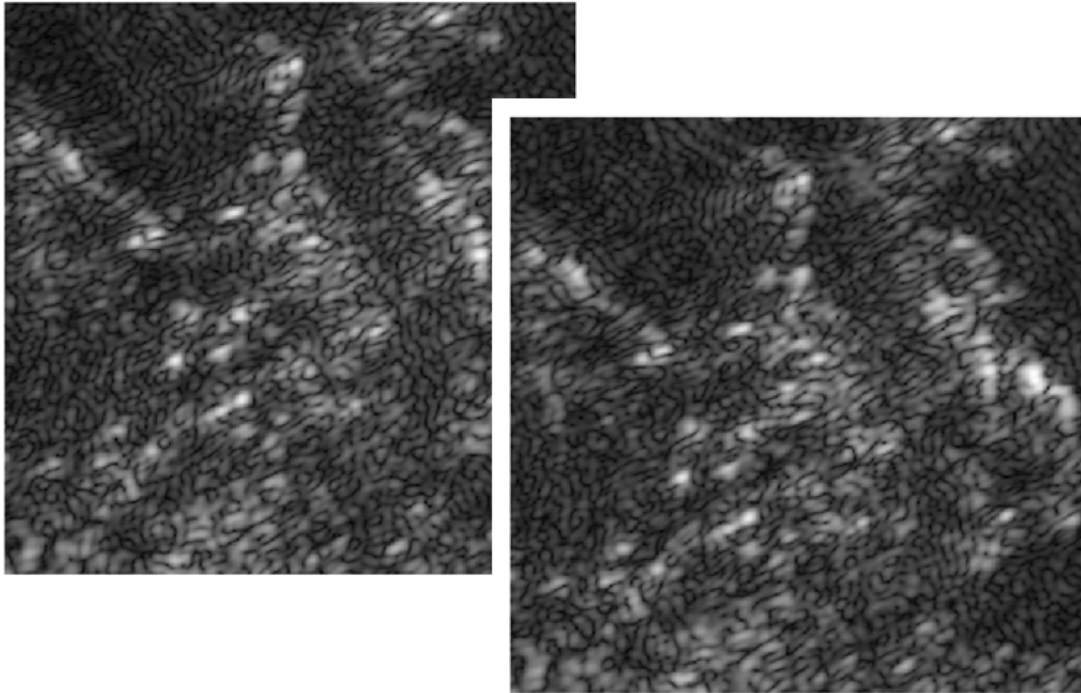




CHALMERS
UNIVERSITY OF TECHNOLOGY



Real-Time Change Detection in SAR Images

Master's thesis in Systems, Control and Mechatronics

MAX NORDIN

MASTER'S THESIS EX048/2017

Real-Time Change Detection in SAR Images

MAX NORDIN



Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Real-Time Change Detection in SAR Images
MAX NORDIN

© MAX NORDIN, 2017.

Supervisor: ANDERS ÅHLANDER, Saab AB
Examiner: TOMAS MCKELVEY, Department of Electrical Engineering

Master's Thesis EX048/2017
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Two SAR images of the same area taken at two different occasions. Two targets have shown up in the right image and can be seen as two white spots in the middle of it.

Typeset in L^AT_EX
Gothenburg, Sweden 2017

Real-Time Change Detection in SAR Images
MAX NORDIN
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Carabas is a surveillance system that uses a synthetic aperture radar (SAR) to take images of the ground from the air. Images taken at separate occasions can be compared with each other to find so called targets, such as vehicles hidden in the forest. This process is known as change detection. A new change detection method based on Bayesian statistics has been implemented in form of a C++ program. A few additions has been made to the change detection method to automate the process without human interaction. A new median filtering method has also been developed that was proved to be faster than using selection algorithms. The program utilizes a multi-core processor architecture that is already used for existing Carabas signal processing. To fully utilize the hardware, a pipeline structure was created that allows for different images to be processed in parallel and to divide the change detection of one image into parallel parts. From the C++ program it was found that change detection can be run in real-time with Carabas. The time it takes to do change detection is roughly one tenth of the time it takes to produce a SAR image. Using the pipeline structure it is possible to integrate change detection with the rest of the Carabas signal processing and reduce the computational time. Finally, a test environment were change detection can be tested has also been implemented.

Keywords: change detection, Bayesian statistics, synthetic aperture radar, parallel processing, POSIX threads, OpenMP, median filtering.

Acknowledgements

I would like to thank the people at Saab AB for giving me the opportunity to carry out my master's thesis work with them. First and foremost, I would like to thank Anders Åhlander for supervising me and providing me with valuable feedback. I would also like to thank Jonas Lindgren for his supervision and insights. Finally, I would like to thank Hans Hellsten for his help with tackling the change detection algorithm.

Max Nordin, Gothenburg, June 2017

Contents

List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Purpose	1
1.2 Scope	2
1.3 Ethical aspects	2
1.4 Sustainability aspects	2
1.5 Thesis outline	2
2 SAR and change detection	5
2.1 Synthetic aperture radar	5
2.1.1 Carabas	7
2.1.2 Change detection	7
2.2 The change detection algorithm	7
2.2.1 Basic principle	7
2.2.2 Target statistics	9
2.2.3 Clutter statistics	10
2.2.4 Median filtering	11
2.3 Helpful algorithms	11
2.3.1 Welford's method	11
2.3.2 Sorting algorithms	12
2.3.2.1 Bucket sort	13
2.3.2.2 Quicksort	13
2.3.2.3 Merge sort	13
3 Parallel processing	15
3.1 Parallel hardware	15
3.2 Parallel software	16
3.3 Processes and threads	16
3.4 Data races	17
3.5 Synchronization primitives	17
3.5.1 Mutex	17
3.5.2 Condition variable	18
3.5.3 Other primitives	18
3.6 Performance	18

3.6.1	Amdahl's law	19
4	Methods	21
4.1	Hardware	21
4.2	Implementation of the change detection algorithm	21
4.2.1	Target statistics	22
4.2.2	Pre-processing	24
4.2.3	Histogram creation and maintaining	24
4.2.4	Clutter statistics	25
4.2.5	Target locating	26
4.2.6	Median filtering	26
4.2.6.1	Filtering the first row	27
4.2.6.2	Filtering the second and subsequent rows	28
4.2.7	Target evaluation	30
4.3	Parallelization	31
4.3.1	The master-worker model	32
4.3.2	Parallel filters	34
4.3.2.1	Parallel change detections	34
4.4	Verification and testing of the implemented C++ program	36
4.4.1	Time measurements	36
4.4.2	Artificial targets	37
5	Results	39
5.1	Evaluation and comparisons	39
5.1.1	Comparison with MATLAB scripts	39
5.1.2	Detection of false targets	42
5.2	Execution time and parameters	42
5.2.1	Automatic stop	44
5.2.2	Type casting	44
5.2.3	Median filter	45
5.2.4	Parameters	46
5.3	Parallelization with OpenMP and pthreads	47
5.3.1	Streaming data	48
5.4	Hardware and Limitations	49
6	Conclusion	51
	Bibliography	53

List of Figures

2.1	A SAR image obtained using Carabas. It is also possible to obtain images looking more like a photograph by using higher frequencies than the range 140-360 MHz used for this image.	6
2.2	Data points from a real SAR image inserted into two different histograms. Red color indicates a high density of data points and blue color a low density.	10
2.3	Median filtering with the unfiltered image on the left and the filtered image to the right. The red square on the left illustrates the filtering window. The median of the values within the window is the result in the filtered image in the red square to the right.	11
4.1	Overall structure of the C++ function for change detection where each block is a sub function.	22
4.2	The information from the window in Figure 2.3 sorted.	27
4.3	The second pixel the filtered value is computed for is marked red on the right and the corresponding filtering window in red on the left. The previous window in blue has been sorted in an array A_1 and the elements in the yellow area in an array B_1	28
4.4	Illustration of the merging. A_1 (blue) is merged with B_1 (yellow) into C_1 (red).	29
4.5	The filtering process for all pixels not in the first row or column (in the filtered image). Here the previous filtering window (blue) has a sorted array A_2 . There is also a sorted array B_1 saved from before (green). There is also one element (yellow) not in A_2 or B_1 that is included in the current window.	30
4.6	The master-worker model. A master thread puts items in a buffer and the worker threads removes them and do some work associated with them. For change detection, the master divides an image into several sub-images and the workers do the change detection on the sub-images.	32
4.7	The structure of a filter which was implemented to be used for change detection and other types of signal-/image processing.	35
5.1	Illustration of two targets appearing in the reference image. They can be seen as two small dots in the center of the white circle in the update image. The circles are not a part of the original SAR images.	40
5.2	The full 36 Mpix reference image containing two real targets.	41

5.3	Data that is corrupted which can be seen as diagonal lines within the images. Due to these lines differing, targets were detected despite there being no real deployed targets.	41
5.4	The full 16 Mpix reference image where a false targets were detected.	43
5.5	False targets were detected by the bright spot in the middle of the circle. No big difference is seen between the two images. The circles are not a part of the original SAR images.	43
5.6	Computation time in seconds of median filtering using different window sizes m	45

List of Tables

4.1	Hardware and operating system specification.	21
4.2	All parameters that have to be set for the implemented change detection function.	23
5.1	Computation time in seconds of median filtering using different window sizes m on a 5000×5000 matrix containing double precision floating point values.	46
5.2	Execution time and speedup for change detection on one SAR-image divided into 36 sub-images.	48
5.3	Average execution time and speedup from streaming SAR-images. . .	48

1

Introduction

Using a Synthetic Aperture Radar (SAR), the Carabas system developed at Saab, is a surveillance system used to take high resolution images of the ground from an aircraft. By making measurements while moving and processing the data in a certain way, SAR allows for higher resolutions that otherwise would require a much longer antenna. The frequencies at around 20 – 360 MHz used with Carabas enables the electromagnetic waves to penetrate foliage, buildings and even the ground. Unlike other radars operating at higher frequencies, Carabas can be used to detect concealed objects such as a vehicle hidden under a tree. To find the vehicle in the image, Carabas uses Change Detection (CD) which finds objects referred to as targets that have appeared/disappeared when comparing to older images stored in a database.

Change detection with Carabas has been an ongoing research area and other CD methods have been investigated for real-time use [1]. The new CD-algorithm developed at Saab is based on Bayesian statistics and uses Bayesian methods to compute target probabilities. This method finds targets fast with similar low false alarm probabilities as more time consuming methods. It has been tested using Matlab with positive results and is now waiting to be implemented and integrated with Carabas. The CD-algorithm is presented by Hellsten [2] who describes how it works and is derived. Hellsten also provides some information about the implementation in Matlab [3].

1.1 Purpose

The purpose of this thesis work is to assist Saab with the development of Carabas by contributing to the integration of the change detection algorithm. The result should be a realization in C++ that takes two SAR-images and outputs possible targets alongside their probabilities. It should be well optimized and utilize parallel processor architecture. The execution time should not be slower than the time it takes to create SAR-images (around 40 seconds).

1.2 Scope

The realization is based on the novel CD method developed by Saab with the purpose to be integrated with Carabas. Therefore, any other potential methods which to do change detection with is not looked into.

At firsthand the realization has been made to work with the multi-core processor hardware currently used with Carabas. Thus, the thesis work does not go in depth with different kinds of hardware, for instance GPUs or manycore processors.

1.3 Ethical aspects

Carabas, which the thesis work will be a part of, is a surveillance system partially used for military operations. One can argue that the use or development of these types of systems are wrong due to the damage caused by warfare throughout history. On the other hand they can also be used as a defence to prevent damage. By researching these areas one can contribute to that these systems are made to prevent damage rather than cause damage. One can also make people feel safe and secure knowing that these defence systems will protect them.

1.4 Sustainability aspects

While the main purpose of Carabas is for surveillance in military operations it has other uses as well. Examples are terrain topography mapping, biomass estimation and disaster monitoring. For instance, biomass estimation can help measuring deforestation which causes global warming [4] and a decline in biodiversity [5]. Thus, this thesis has applications that plays a part in a sustainable future.

1.5 Thesis outline

The thesis is divided into five other chapters. The content for each chapter is as follows:

- Chapter 2 gives a brief explanation of the SAR technology. After that the new change detection algorithm is explained in detail and some additional useful algorithms are introduced.
- Chapter 3 introduces parallel processing from both a hardware and software perspective. It continues by going more in depth on the software side.

- Chapter 4 describes how change detection was realized as a C++ program. First, how the algorithm was made to run efficient as code is explained which is followed by how it was parallelized.
- Chapter 5 presents the end results. The correctness of the implementation as well as the execution time is considered and discussed.
- Chapter 6 is an conclusion of the thesis and discusses future work.

2

SAR and change detection

To be able to follow the whole thesis, some basic concepts are explained in this chapter. This includes an explanation of SAR imaging and the new change detection algorithm. Some theory which is useful when working with the algorithm is also introduced.

2.1 Synthetic aperture radar

A synthetic aperture radar, abbreviated SAR, is a type of radar that is used to create images, often of terrain taken from air crafts or satellites. The basics of SAR are the same as with any radar. It uses an antenna to transmit electromagnetic waves out in the environment. Some of these waves are reflected back at the radar and received by an antenna, which could be and often is the same antenna that transmitted the signal. The distance to different objects are obtained by measuring the time it takes for the signal to come back.

What is special about SAR is that it uses a moving antenna. The moving antenna makes it so that the signals are transmitted and received at different positions, creating a so called synthetic aperture. This gives the radar a higher resolution which allows the creation of high resolution images of the terrain. SAR has several advantages over cameras, one being that the radio waves used are mostly unaffected by weather unlike visible light. It is also not dependent on an external source such as the sun which makes it work independent of the time of the day.

Unlike with cameras, the data received with SAR does not resemble an image until after a few signal processing steps. By filtering the data both 2D and 3D images can be obtained where the 2D images are the focus for change detection. Each pixel in the 2D images has a complex value representing the amplitude and phase of the received signal. An example of an image taken with Carabas where the amplitude is plotted can be seen in Figure 2.1. A detailed explanation of the SAR technology along with the signal processing has been made by Moreira et. al [6].



Figure 2.1: A SAR image obtained using Carabas. It is also possible to obtain images looking more like a photograph by using higher frequencies than the range 140-360 MHz used for this image.

2.1.1 Carabas

The Carabas system [7] developed at Saab stands out by using unusually low frequencies for SAR imaging. Two frequency bands are used at 20-90 MHz and 140-360 MHz respectively (in the VHF/UHF range). The electromagnetic waves at these frequencies, especially the lower band, have the ability to penetrate foliage, certain types of camouflage and can even go through buildings and a distance underground. While the images created might not be as nice to look at, it serves a purpose which is to be able to detect hidden objects known as targets. These are practically invisible to other types of radar or sensors. A typical target is a military vehicle hidden under a tree. Everything detected that is not a target will henceforth be referred to as clutter. Carabas is used to surveil large areas and multiple images are created one at a time, resulting in a stream of images to perform change detection on.

2.1.2 Change detection

To find a target in an image a technique called change detection is utilized. As can be seen in Figure 2.1 trees appear as dots in the image, which is also true for vehicles or other similar sized objects that reflect radio waves. Change detection is based on comparing the image with an older reference image of the same area, to find targets that looks similar to and could occupy the same space as other objects. Due to different flight paths, calibrations, etc. the clutter will change between the two images. The problem is thus to find changes in the images that are due to targets appearing and not something else. A change detection algorithm should be able to detect targets reliably while keeping the false alarm rate low.

2.2 The change detection algorithm

The change detection algorithm used in this project was recently invented by Hellsten [2, 3] and has not been tested thoroughly. It has however been implemented in MATLAB and has given successful results on several occasions. Implanted targets have been found while other uninteresting changes from buildings, fences and power lines have been neglected. This result is better than those for other alternatives of change detection algorithms.

2.2.1 Basic principle

Given a reference image R and an updated image U , the algorithm tries to find targets that are in the updated image but not in the reference image. This is done by computing the conditional probability $p(x = x_T | a_U(x), a_R(x))$ that there is a

target at pixel x given the measured real-valued amplitudes of the images at x , $a_R(x)$ and $a_U(x)$. Here, x should be the same geographical location for the two images. This conditional probability can with the help of Bayes' theorem and the law of total probability be expressed as

$$p(x_T|a_U, a_R) = \frac{1}{1 + \frac{(1-p(x_T|a_R))p(a_U|x_C, a_R)}{p(x_T|a_R)p(a_U|x_T, a_R)}} \quad (2.1)$$

where x_T denotes there is a target at location x and x_C denotes there is no target i.e. only clutter. The likelihood function $p(a_U|x_T, a_R)$ is obtained by proposing a probability density function $p_T(a)$ for the amplitude of targets which is described in Section 2.2.2. The clutter counterpart $p(a_U|x_C, a_R)$ is estimated using data from the two images and the method used is described in Section 2.2.3. Finally assuming each image has N pixels, there are k targets each of size M pixels and each pixel is equally likely to contain a target, then

$$p(x_T|a_R) = p(x_T) = \frac{Mk}{N}. \quad (2.2)$$

Inserting this into equation (2.1) with $1 - p(x_T|a_R) \approx 1$ one obtains

$$p_k(x_T|a_U, a_R) = \frac{1}{1 + \frac{N}{Mk} \frac{1}{\eta(x)}} \quad (2.3)$$

where $\eta(x) = p(a_U|x_T, a_R)/p(a_U|x_C, a_R)$. The pixels which are most likely to contain targets are the ones with the highest values for $p_k(x_T|a_U, a_R)$ and also $\eta(x)$ since $p_k(x_T|a_U, a_R)$ is monotonically increasing with respect to $\eta(x)$. Because of clutter fluctuations, there is a chance for a high $\eta(x)$ even if there is no target. Thus, the target nominees are the ones that instead maximizes $p_k(x_T|\Lambda_T(x))$ which is the median of $p_k(x_T|a_U, a_R)$ evaluated in a square $\Lambda_T(x)$ with an area of size M and with x in its center. This filters out large pixel-to-pixel changes but not targets.

Multiple targets can be found in an image through an iterative process. This process starts by assuming there are $k = 1$ targets in the updated image. Because the number of targets and their locations are unknown, the data used to estimate $p(a_U|x_C, a_R)$ (i.e. the likelihood of a_U given a_R when there are no targets) includes every pixel in the image pair. This data set, containing every pixel, is denoted S_0 and the corresponding estimated likelihood is denoted $p_0(a_U|x_C, a_R)$. If there is a target in S_0 there will be an error in $p_0(a_U|x_C, a_R)$ which can later be removed and is why this iterative process takes place. The first target nominee, denoted $x_{1,0}$, is found by maximizing $\hat{p}_1(x_T|\Lambda_T(x))$ which in turn is obtained from $p_0(a_U|x_C, a_R)$ using (2.3). Thereafter, $p(a_U|x_C, a_R)$ is estimated once again, now excluding an area $\Lambda(x_{1,0})$ of size greater than M around $x_{1,0}$. This new data set $S_1 = S_0 \setminus \Lambda(x_{1,0})$ yields $p_1(a_U|x_C, a_R)$ as well as $p_1(x_T|\Lambda_T(x))$. If $x_{1,0}$ is a target, the error from $p_0(a_U|x_C, a_R)$ is not in $p_1(a_U|x_C, a_R)$. At last, $p_1(x_T|\Lambda_T(x))$ is maximized with the final target nominee $x_{1,1}$. This can be repeated multiple times with the next iteration assuming $k = 2$ targets. The whole process is summarized below:

1. Set S_0 to be all pixels in the image pair and compute $p_0(a_U|x_C, a_R)$ from S_0 .

2. Starting with $k = 1$, use the already computed $p_{k-1}(a_U|x_C, a_R)$ to estimate the probability of a target $\hat{p}_k(x_T|\Lambda_T(x))$ at all pixels in S_{k-1} .
3. Find the pixels $\{x_{1,k-1}, x_{2,k-1}, \dots, x_{k,k-1}\}$ that maximizes $\hat{p}_k(x_T|\Lambda_T(x))$ and form $S_k = S_0 \setminus \Lambda(x_{1,k-1}) \setminus \Lambda(x_{2,k-1}) \setminus \dots \setminus \Lambda(x_{k,k-1})$.
4. Compute $p_k(a_U|x_C, a_R)$ from S_k .
5. Increment k by 1 and repeat from step 2 or end. The algorithm can run a set number of iterations or it can run until no more target nominees with high probability are found.

2.2.2 Target statistics

To obtain $p(a_U|x_T, a_R)$, it is first assumed that there is no coupling between target and background. Thus the complex image amplitude for the updated image is $a_U e^{i\varphi_U} = a_T e^{i\varphi_T} + a_C e^{i\varphi_C}$ where T and C denotes target and clutter respectively. The variables a_U , a_T and a_C are real and non-negative. To account for wave interference $p(a_U|x_T, a_R)$ is calculated using complex distributions according to

$$\wp(a_U|x_T, a_R) = \oint \int_0^\infty \wp_T(a_U - a_C e^{i\varphi}) \wp(a_C|x_C, a_R) a_C da_C d\varphi \quad (2.4)$$

where \wp represents a complex density function. These are independent on phase and therefore $p(a) = 2\pi a \wp(a)$. The density $\wp_T(a e^{i\varphi})$ is the probability density of a target having the complex amplitude $a e^{i\varphi}$. This should be set to characterize targets and has in this context been set to a constant complex probability

$$\wp_T(a e^{i\varphi}) = \begin{cases} \frac{1}{\pi(a_{\max}^2 - a_{\min}^2)} & a_{\min} \leq a \leq a_{\max} \\ 0 & \text{otherwise} \end{cases} \quad (2.5)$$

where a_{\min} and a_{\max} are the minimum and maximum amplitude that can be expected from a target. The likelihood $\wp(a_C|x_C, a_R)$ is recognized as a complex version of $p(a_U|x_C, a_R)$. Due to a second order influence it can in this case be approximated as a Dirac delta function

$$\wp(a_C|x_C, a_R) = \frac{\delta(a_C - a_R)}{2\pi a_R}. \quad (2.6)$$

Evaluating (2.4) using (2.5) and (2.6) results in

$$p(a_U|x_T, a_R) = 2\pi a_U \wp(a_U|x_T, a_R) = \frac{2a_U(\Delta\varphi_{\max}(a_U, a_R) - \Delta\varphi_{\min}(a_U, a_R))}{\pi(a_{\max}^2 - a_{\min}^2)} \quad (2.7)$$

where

$$\Delta\varphi_{\max}(a_U, a_R) = \arctan \frac{a_{\max}^2 - a_U^2 - a_R^2}{\operatorname{Re} \sqrt{a_{\max}^2 - (a_U - a_R)^2} \operatorname{Re} \sqrt{(a_U + a_R)^2 - a_{\max}^2}} \quad (2.8)$$

and

$$\Delta\varphi_{\min}(a_U, a_R) = \arctan \frac{a_{\min}^2 - a_U^2 - a_R^2}{\operatorname{Re} \sqrt{a_{\min}^2 - (a_U - a_R)^2} \operatorname{Re} \sqrt{(a_U + a_R)^2 - a_{\min}^2}}. \quad (2.9)$$

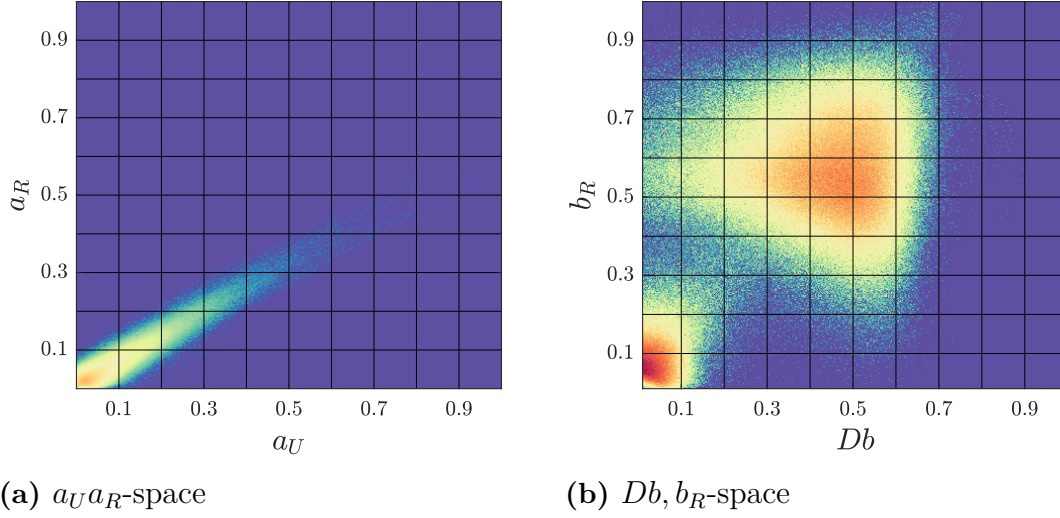


Figure 2.2: Data points from a real SAR image inserted into two different histograms. Red color indicates a high density of data points and blue color a low density.

2.2.3 Clutter statistics

The last part is to determine $p(a_U|x_C, a_R)$. Due to this likelihood function changing between image pairs, it is estimated every time there is a new pair of images. The suggested method is using histograms to obtain the cumulative distribution at a set number of points by summing up the histogram bin counts. Values for additional points are obtained with an interpolation method. The likelihood $p(a_U|x_C, a_R)$ is then found as the derivative to the cumulative distribution. As can be seen in Figure 2.2a, simply dividing the $a_U a_R$ -space into histogram bins would not yield a good result since most of the bins would be empty and a few number of bins would hold the majority of the pixel pairs. Instead it is noticed that $p(a_U|x_C, a_R)$ is spread along a line $a_R = ka_U$ where k is the gradient of the line. Denote $v = [v_1 \ v_2]^T$ the eigenvector corresponding to the largest eigenvalue to the covariance matrix

$$Q = \begin{bmatrix} \text{Var}(a_U) & \text{Cov}(a_U, a_R) \\ \text{Cov}(a_U, a_R) & \text{Var}(a_R) \end{bmatrix} = \begin{bmatrix} \sigma_U & \sigma_{UR} \\ \sigma_{UR} & \sigma_R \end{bmatrix} \quad (2.10)$$

with which the gradient is given by $k = v_2/v_1$. Instead of the $a_U a_R$ -plane, look at the Da, a_R -plane where

$$Da \triangleq ka_U - a_R \quad (2.11)$$

and $p(Da|x_C, a_R)$ will be spread along the Da -axis instead. The letter D represents the difference. Since targets appearing in an image practically always increases the amplitude it is safe to assume that $p(Da|x_C, a_R) = 0$ for $Da < 0$. Thus only $Da > 0$ has to be included in the histogram. Finally, let Da and a_R relate to Db and b_R as a relates to b in $a = Ce^{\rho b} + D$ where ρ , C and D are constants that can be determined by testing different combinations of values. By using the Db, b_R -plane, samples from $p(a_U|x_C, a_R)$ are much more evenly spread across the bins, see Figure 2.2b.

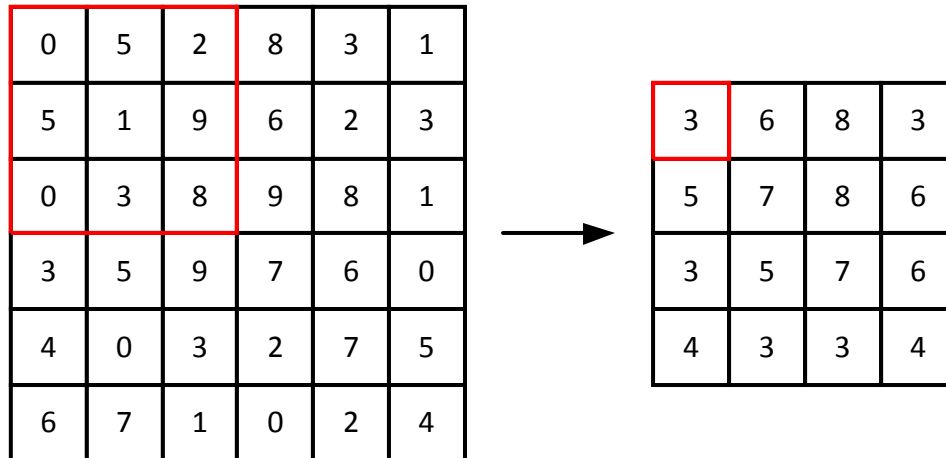


Figure 2.3: Median filtering with the unfiltered image on the left and the filtered image to the right. The red square on the left illustrates the filtering window. The median of the values within the window is the result in the filtered image in the red square to the right.

2.2.4 Median filtering

Median filtering on 2D images functions like any other image filter. Each pixel in the filtered image will have a value that depends on the same pixel in the unfiltered image and an area around that pixel. The area is referred to as a window or kernel. For a linear filter the filtered value is a linear combination of all values within the window and for a median filter it is the median of the values in the window. An example of a median filtered image is seen in Figure 2.3. Here the window is a 3-by-3 square. The edges are not filtered in this case resulting in an image that is smaller. Further information about image processing can be found in *Computer Vision* [8].

2.3 Helpful algorithms

For an implementation of the change detection algorithm there are certain algorithms that can be used or are useful to know about. These are described below.

2.3.1 Welford's method

When using computers to solve mathematical problems there are numerical errors that occur due to a computers inability to represent all real numbers. When computing variance, several digits of precision can be lost or an entirely wrong answer

can be obtained if the problem is approached in the wrong way. The problem with covariance is that involves summing up squared numbers. Computing this using the wrong approach in a computer program can not only cause loss of precision, but also cause overflow problems when working with huge data sets or big numbers. Welford's method [9] was developed to prevent these problems when computing for instance variance. The method is an iterative process working on a data set $\{x_1, x_2, \dots, x_n\}$. Let m_k be the mean of x_1, x_2, \dots, x_k where $1 \leq k \leq n$ and let

$$s_k = \sum_{i=1}^k (x_i - m_k)^2. \quad (2.12)$$

Using $m_0 = s_0 = 0$, s_n is then the result from iterating through the following equations n times:

$$m_k = m_{k-1} + \frac{1}{k}(x_k - m_{k-1}) \quad (2.13)$$

$$s_k = (x_k - m_k)(x_k - m_{k-1}) \quad (2.14)$$

2.3.2 Sorting algorithms

Hellsten's change detection algorithm makes use of median filtering. The median for a set of numbers is the middle element when all numbers have been ordered numerically. To obtain the median some type of selection or sorting is thus required. Sorting algorithms involves sorting an entire array of numbers while selection algorithms involves finding a number at a certain position. Because the whole array does not have to be sorted, selection algorithms are generally the fastest way to obtain the median [10]. This is however not the case when working with two-dimensional median filters where there are fast methods based on sorting [11, 12].

There is no best sorting algorithm for all purposes as different algorithms have their advantages and disadvantages [13]. They are often compared in terms of computational complexity, meaning how much the computation time scales with the number of elements in the data set. Different data also affects the computation time creating a best-case, worst-case and average scenario. For instance, the bubble sort algorithm has a best case performance of $O(n)$ meaning it scales linearly with the number of elements n . This occurs whenever the set is already sorted. The average performance is however $O(n^2)$, which is worse than the more efficient algorithms with an average performance of $O(n \log n)$. Another important aspect is the memory usage of the algorithms. Some algorithms requires close to no extra memory at all while some requires extra memory for larger arrays.

A few different algorithms are explained below. These are useful to know about when considering two-dimensional median filtering. For further reading about sorting and sorting algorithms one can look into *The Art of Computer Programming: Sorting and Searching* [14].

2.3.2.1 Bucket sort

Bucket sort can be a very efficient sorting algorithm making use of so called buckets or bins. An example could be that integers between 0 and 99 are to be sorted. Ten buckets are created where values 0 to 9 will be sorted into the first bucket, values 10 to 19 into the second bucket and so on. Deciding which bin to put a value in this case is done by looking at the most significant digit. For instance number 52 gets sorted into the bin indexed 5 i.e. the sixth bin. The elements in each bin can then be sorted with each other using bucket sort or any other sorting algorithm. The strength of bucket sort comes from that it does not have to compare different elements with each other, making it sometimes superior to other comparison based methods such as quicksort and merge sort.

2.3.2.2 Quicksort

A very popular sorting algorithm is quicksort because of its all-round good performance. The algorithm starts by finding a so called pivot element which can be chosen randomly or with the help of another algorithm. The array is then partitioned meaning all elements smaller than the pivot elements gets placed on one side of it, while those larger than the pivot element gets placed on the other side. The process of choosing a new pivot element and partitioning is repeated with the two parts of the array divided by the pivot element. The algorithm continues recursively until the whole array is sorted.

A variation of quicksort known as quickselect is a selection algorithm that can be used to find the median. With quickselect, only the side of the pivot element that is containing the median value is used for the next recursion. This is thus a fast way of finding the median without having to sort the entire array.

2.3.2.3 Merge sort

Merge sort is another popular sorting algorithm. First the array is split into n sub arrays, each of length one. Shorter arrays are then merged together into longer arrays until only one array is remaining. The merging process also keeps the arrays sorted. Two arrays are merged together by comparing the first element in each of the arrays. The smallest element (or greatest depending on how you sort) of the two are removed from the array and gets placed in a new array. This continues until all elements have been added in numerical order to the new array.

3

Parallel processing

During the last decade, parallel hardware has become widely popular and are used frequently in computers and embedded systems [15]. Modern CPUs are typically made up of several cores acting like separate processing units, allowing multiple instructions to be carried out at the same time. For a program to utilize the hardware to its full potential it has to be tailored to do so. With this comes different challenges concerning both hardware and software. Parallel programming is similar to asynchronous programming in a sense that programs are made with non-dependant tasks that can run in any order. The difference being that in the parallel case different tasks are processed at the same time, whereas in the asynchronous case only one task is processed at any given time. The basics to parallel programming is described in *An Introduction to Parallel Programming* [16].

3.1 Parallel hardware

There are different kinds of hardware specialized in different things that can be classified using Flynn's taxonomy. The classification is based on whether the system architecture is made to process one or multiple instruction streams, as well as one or multiple data streams. The instruction stream tells the processor what to do, such as multiply two numbers, and the data stream provides the data i.e. the numbers which to multiply.

The two important classifications for this thesis' purpose are MIMD (Multiple Instructions Multiple Data) and SIMD (Single Instruction Multiple Data). MIMD includes CPUs with more than one core, also known as multi-core processors, where each processor core works with its own set of instructions and its own data. A MIMD system can be either a shared-memory system or a distributed memory system. The difference being that in a shared-memory system the different processors or cores share memory, while in a distributed-memory system each processor has its own private memory. A single PC is in most cases a shared-memory system but multiple computers connected together can create distributed memory systems such as supercomputers. Examples of the SIMD classification are GPUs, made to repeat the same instructions a multitude of times on different data which is useful

in computer graphics. Some GPUs could be classified as MIMD and are able to run multiple instruction streams at the same time and general purpose computing on GPUs (GPGPU) has made it possible for GPUs to perform computations normally handled by a CPU.

3.2 Parallel software

The way software is created differs depending on the hardware. For shared-memory MIMD/SIMD and C/C++ there are different APIs (application programming interfaces) for parallel programming. Two common ones are OpenMP and POSIX threads (pthreads). The goal of OpenMP and pthreads is essentially the same thing, split parts of a program that does not depend on each other and have these tasks be processed in parallel. The implementation differs slightly giving the two APIs some advantages and disadvantages over each other. OpenMP is a more straightforward approach allowing parallelization with as little effort as possible. Pthreads on the other hand is a little bit more in depth allowing better control over the program structure. Pthreads generally takes more time and code to implement but has a greater potential in terms of performance.

3.3 Processes and threads

An instance of a computer program is called a process. In a normal computer there are always several processes running at once. Since a processor core only can work with one process at a time someone or something has to decide which process is allowed to run. This is known as scheduling and is handled by the operating system. The scheduling can also be influenced by the programmer. For instance, threads can be set to be prioritized over other threads.

Processes work mostly unknowingly of each other with their own resources needed to maintain the process. A process contains one or more threads which in turn can share resources with other threads within the process. Because of this, creating, communicating between, and sharing data between threads are faster than for processes. A thread is processed in a sequential order making it unable to be used by more than one processor or processor core at once. To use more than one core, multiple threads have to be used, either in the form of multiple processes with one thread, one process with multiple threads or a combination of the two. Because of the speed advantage, OpenMP and pthreads are used to creating and maintaining threads within the same process.

3.4 Data races

When working with multiple threads there are problems that can occur when more than one thread is trying to access a shared resource. These problems are called data races or race conditions and occur when multiple threads are trying to access the same variable and at least one of the threads is trying to write to the variable. The problem lies in that the outcomes may differ depending on the order the instructions of the different threads are executed in.

In *Pthreads programming* [17] there is a good example of a data race where a man and a woman are withdrawing money from the same bank account using two different ATMs. The ATMs operate in parallel i.e. on different threads while sharing the bank account balance. The man makes a withdrawal of \$50. The ATM reads the balance which is \$125, subtracts \$50 from it and writes the result \$75 to the balance before giving the man his money. At the same time, the woman also makes a withdrawal of \$50. The man has not yet finished his withdrawal so this ATM also reads the balance as \$125. However, while in the process of subtracting, the man finishes his withdrawal which changes the bank account balance. The second ATM which also gets a result of \$75 overwrites the balance (which already is \$75) and gives \$50 to the woman. Thus the man and woman did end up withdrawing a total of \$100 while only subtracting \$50 from the bank account balance.

3.5 Synchronization primitives

To avoid data races and to aid interaction between threads there are so called synchronization primitives. These primitives come with the API's but are implemented differently whether OpenMP or pthreads is used. Since pthreads has been the main focus during the thesis, this section describes how synchronization primitives are used with pthreads.

3.5.1 Mutex

A common primitive is a lock often called mutex. When a thread locks a mutex it will be the owner of that mutex and is the only thread that can unlock it. If another thread requests to lock the mutex it immediately has to stop and wait for the mutex to be unlocked by its owner before proceeding. A mutex thus makes it impossible for more than one thread to enter a critical section with shared resources. A problem that comes with mutexes are deadlocks where one or more threads are permanently blocked and will never proceed. This can happen if for example a thread forgets to unlock a mutex or if two threads each are waiting for a mutex owned by the other thread.

3.5.2 Condition variable

Condition variables are another type of primitives used by threads to signal each other that a certain event has occurred. An event is something required for certain threads to proceed and could for instance be a set of data that is finished with being processed. If a thread locks a mutex, enters a critical section and sees that certain conditions are not satisfied for it to continue, it can use a condition variable to wait for those conditions to be met. While waiting it will unlock the mutex allowing other threads to enter the critical section. One of these threads will (hopefully) modify the shared resources so that the conditions are met i.e. the desired event occurs, and will use the condition variable to signal the waiting thread. Before resuming, the waiting thread must once again become the owner of the mutex. There are two types of signaling that can be done with a condition variable. A normal signal will wake one thread that is currently waiting. There is also broadcasting which acts like a signal on all threads that are waiting.

3.5.3 Other primitives

Apart from mutexes and condition variables there are various other synchronization primitives. Semaphores are one of those. They act similar to mutexes in a sense that they protect shared resources. Semaphores are useful when the order in which the threads have to access shared resources is important or when more than one thread can be allowed into a critical section. A barrier is another type of synchronization primitive. When a thread reaches a barrier it has to wait for all other threads participating in the barrier to reach it before said thread can continue.

3.6 Performance

For a parallelization to be justified it has to yield a performance gain in some way. Speedup S is a way of measuring how much faster the execution time of a parallel program T_p is compared to the execution time of the sequential program T_s doing the same computations. The speedup is defined as

$$S = \frac{T_s}{T_p}. \quad (3.1)$$

A linear speedup means that the speedup equals the number of cores. In this case, all computations have been equally split between the cores and nothing extra that takes time has been added. This is seen as a best case scenario which is unlikely to be obtained in practice. Another way of measuring performance is using efficiency E defined as

$$E = \frac{S}{p} = \frac{T_s}{T_p p} \quad (3.2)$$

where p is the number of processor cores. An efficiency of 100% is the same as a linear speedup.

3.6.1 Amdahl's law

Even though parallelization is useful to gain performance and reduce execution time, there are still limitations. For instance if 90% of computations in a sequential program is parallelizable, the total execution time can never be faster than that last 10%. Assume a fraction f of a program, $0 \leq f \leq 1$, is not parallelizable but that the rest is parallelizable with a linear speedup. Then the execution time is

$$T_p = (f + \frac{1-f}{p})T_s \quad (3.3)$$

for the parallel program. From this an upper bound to the speedup is obtained according to

$$S = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}. \quad (3.4)$$

which is known as Amdahl's law.

4

Methods

A C++ function was created that performs change detection according to the algorithm in Section 2.2. Since there was a desire to speed up other computations, a C++ class was created that not only allows for parallelization of change detection, but also for other computations. A program that allows for testing of how well the change detection function works and how fast the execution time is was created alongside the change detection function.

4.1 Hardware

The C++ implementation and testing of the change detection was carried out on the same hardware that have been used for the rest of the signal processing in Carabas. This was a Linux server with specifications that can be seen in Table 4.1. With a total of 12 processor cores split between two processors, parallelization was an important part of reducing the execution time.

4.2 Implementation of the change detection algorithm

The change detection algorithm introduced in Section 2.2 was implemented as a sequential C++ function. It was divided into several sub functions with a structure illustrated in Figure 4.1 where the different sub functions are explained in more detail below. The inputs to the created function were two 2D complex valued SAR images,

OS distribution	SUSE Linux Enterprise Server 11
Processor(s)	2× Intel Xeon CPU X5675 @ 3.07 Ghz
Number of CPU cores	2 × 6
RAM	100 GB

Table 4.1: Hardware and operating system specification.

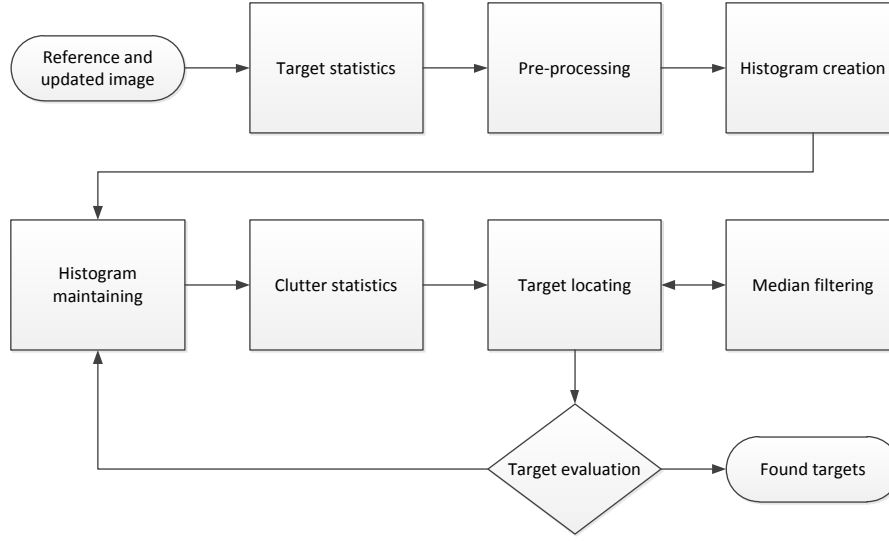


Figure 4.1: Overall structure of the C++ function for change detection where each block is a sub function.

as well as some different parameters, see Table 4.2. Its output is a list of targets containing the location of the targets and the computed value for $p(x_T|a_U, a_R)$ from equation 2.1.

The basic approach was to first compute everything not dependant on results from the iterative process, save it to memory so that it does not have to be computed more than once and then start iterating in a loop. Because at least two iterations are needed to find only one target, it saved time in even the simplest of cases. An extra part was also added to the algorithm to save additional computation time. This part, labeled *Target Evaluation* in Figure 4.1, was given the task to decide the number of iterations. For example, if the algorithm should be able to find ten targets, it has to go through at least eleven iterations. In most cases in practice, there are no targets to find within an image pair (targets are rare) and doing eleven iterations each time takes up resources that can be used for other things such as parallel change detections, see Section 4.3. In these cases the target evaluation would see that there were no high probability target nominees within the first few iterations and end the change detection. Below follows descriptions of this and all the other the different sub functions and how they were implemented.

4.2.1 Target statistics

Everything necessary to compute $p(a_U|x_T, a_R)$ in equation (2.7) was placed within its own function. The target hypothesis did not change between iterations and thus the computations could be done before the loop. In fact, since the targets upper and lower amplitudes (a_{\max} and a_{\min}) were implemented as inputs to be set manually, it also did not depend on the images and could have been done before

Parameter	Description
a_{\max}	Maximum amplitude a target can have
a_{\min}	Minimum amplitude a target can have
Δa_R	Step size for the grid the different probabilities are evaluated at
ΔDa	Step size for the grid the different probabilities are evaluated at
$b_{n,a}$	Number of histogram bins in the a_R -direction
$b_{n,Da}$	Number of histogram bins in the Da -direction
ρ_a	Each consecutive bin is e^{ρ_a} times larger than the previous one
ρ_{Da}	Each consecutive bin is $e^{\rho_{Da}}$ times larger than the previous one
m	Target size in pixels, $M = m^2$
d_{\min}	Minimum distance between two target nominees, $d_{\min} \geq m$
k_{\max}	Maximum number of iterations
p_t	Minimum probability required for a target
Auto stop	If the algorithm can stop itself before k_{\max} has been reached
Δp	Probability change required from nominees to not auto stop
$k_{\Delta p}$	Number of iterations without a change Δp before auto stop

Table 4.2: All parameters that have to be set for the implemented change detection function.

starting the change detection. However, since the computations were relatively fast it gave no advantage to do so. The proposed constant complex probability was implemented with some slight modifications to equations (2.8) and (2.9) to avoid working with complex numbers and dividing by zero. When $(a_U + a_R)^2 > a_{\max}^2$ and $(a_U - a_R)^2 < a_{\max}^2$ everything under the square roots are positive which makes the square roots real. When $(a_U + a_R)^2 \leq a_{\max}^2$ or $(a_U - a_R)^2 \geq a_{\max}^2$ the denominator is zero and the numerator is positive or negative respectively. Equation (2.8) could thus be modified to the following:

$$\Delta\varphi_{\max}(a_U, a_R) = \begin{cases} -\frac{\pi}{2} & |a_U - a_R| \geq a_{\max} \\ +\frac{\pi}{2} & a_U + a_R \leq a_{\max} \\ \Delta\tilde{\varphi}_{\max}(a_U, a_R) & \text{otherwise} \end{cases} \quad (4.1)$$

$$\Delta\tilde{\varphi}_{\max}(a_U, a_R) = \arctan \frac{a_{\max}^2 - a_U^2 - a_R^2}{\sqrt{a_{\max}^2 - (a_U - a_R)^2} \sqrt{(a_U + a_R)^2 - a_{\max}^2}} \quad (4.2)$$

Analogously, $\Delta\tilde{\varphi}_{\min}(a_U, a_R)$ was obtained from $\Delta\varphi_{\min}(a_U, a_R)$ and a_{\min} . With these new expressions $p(a_U|x_T, a_R)$ was evaluated at a grid in the Da, a_R -space. The grid-points were separated by the distances ΔDa and Δa_R in the range 0 to 1 at points $Da = \frac{1}{2}\Delta Da, \frac{3}{2}\Delta Da, \dots, 1 - \frac{1}{2}\Delta Da$ and $a_R = \frac{1}{2}\Delta a_R, \frac{3}{2}\Delta a_R, \dots, 1 - \frac{1}{2}\Delta a_R$. This grid was used later to obtain $\eta(x)$. The cmath library (math.h in C) was used to compute the arctangent, the square root and later also the logarithm.

4.2.2 Pre-processing

The pre-processing of the implementation took care of the computations of the difference image, that is the image with values Da , using the two complex input images. Since these computations were the same between iterations they were placed before the loop, hence the name pre-processing. The first and most straightforward part of the pre-processing was to compute the amplitudes a_R and a_U by taking the absolute value of the complex numbers in the input images. Next up was the difference image which required the covariance matrix Q to be computed. The three different elements in the 2-by-2 matrix were computed in conjunction with each other using the Welford's method, which was chosen because of its robustness. For a 2-by-2 matrix there is an expression that was used for the largest eigenvalue

$$\lambda = \frac{\text{Tr}(Q)}{2} + \sqrt{\frac{\text{Tr}(Q)^2}{4} - |Q|} \quad (4.3)$$

where $\text{Tr}(Q)$ denotes the trace of Q and $|Q|$ the determinant. The corresponding eigenvector is

$$v = \begin{bmatrix} \lambda - \sigma_R \\ \sigma_{UR} \end{bmatrix}. \quad (4.4)$$

This assumes σ_{UR} is nonzero which is true if the values a_R and a_U indeed are scattered along a line. These expressions were used to obtain the difference image according to equation (2.11). Lastly the resulting images were normalized to contain values between 0 and 1. This was done to be consistent with the MATLAB implementation and to be able to use similar parameter values (mainly a_{\max} and a_{\min}).

4.2.3 Histogram creation and maintaining

Two similar functions were implemented to move some of the computations used for $p(a_U|x_C, a_R)$ outside of the loop. First the histogram that was used to estimate $p(a_U|x_C, a_R)$ was created outside the loop and later it was also updated each iteration to exclude target nominees. To decide which bin in the histogram an amplitude a belonged to Hellsten used the equation

$$a = Ce^{\rho b} \Rightarrow b = \frac{1}{\rho} \log\left(\frac{a}{C}\right) \quad (4.5)$$

where $\rho = 0.5$, $C = 0.001$ and a can have values ranging from 0 to 1.0967. This created 15 bins where pixels with $b \in [-\infty, 0)$ were put in the first bin, $b \in [0, 1)$ in the second bin, $b \in [1, 2)$ in the third bin and so on. The equation was modified to

$$a = \frac{e^{\rho b} - e^{-\rho}}{e^{\rho b_n} - e^{-\rho}} \Rightarrow b = \frac{1}{\rho} \log\left(a(e^{\rho b_n} - 1) + 1\right) \quad (4.6)$$

where once again $\rho = 0.5$, b_n are the total number of bins and a are normalized to contain values from 0 to 1. This equation allowed for an easier change in the

number of bins since it could just be inserted in the equation. It also gained a small performance boost in terms of execution time. The pixels with $b \in [0, 1)$ were put in the first bin, $b \in [1, 2)$ in the second and so on. By indexing the bins $0, 1, \dots, b_n$ the index to use was obtained by taking the floor of b . In practice this was done by type casting the floating point number to an integer and use the integer as the index to an array. This was found to be faster than looping through the different bins and use comparisons to decide which bin an amplitude belongs to which had been used in the MATLAB implementation. The first equation could have been used with type casting in the same way. However, there first would have had to be a check whether b is non-positive to see if it should have been put in the first bin. The faster execution time can relate to how bucket sort is sometimes faster than comparison based sorting.

Before the loop a two dimensional histogram was created (b_R values in one dimension and Db values in the second dimension) using all pixels. In each iteration this histogram was then copied and the pixels in the vicinity of a target nominee removed from the histogram. This was faster than creating a new histogram every iteration because the number of pixels removed were much smaller than the total number of pixels. The area removed from the histogram due to targets had its center at the nominees found the previous iteration and was square shaped with a side of length $6m + 1$ (same as for the MATLAB script). An extra array with the same size as the two images was used to keep track of which pixels had been removed from the histogram. This was necessary to ensure that the same pixel was not removed twice if two target nominees were close to each other. The resulting histogram was used for the clutter statistics.

4.2.4 Clutter statistics

Here the final computations were made that resulted in $p(a_U|x_C, a_R)$. They all relied on the histogram so everything had to be placed inside the loop. The cumulative distribution $P(Db|x_C, b_R)$ was estimated at every point $Db = 1, 2, \dots, b_n$ for each row corresponding to different b_R values. This was achieved by summing up the total number of pixels with a lower amplitude, i.e. summing up the count with each bin prior, and dividing with the total count for all bins with the current b_R value. The result was a grid with $P(Db|x_C, b_R)$ estimated at values $Db = 0, 1, \dots, b_n$ and $b_R = \frac{1}{2}, \frac{3}{2}, \dots, b_n - \frac{1}{2}$. For example, the grid point $b_R = \frac{1}{2}$ corresponds to the bins containing b_R values 0 to 1. The values for additional points were obtained by interpolation. Bilinear interpolation was used because it had been proven to work in the MATLAB implementation and it prevented the cumulative distribution ending up with a negative derivative. The interpolation points were chosen in the Da, a_R -plane as $a_R = \frac{1}{2}\Delta a_R, \frac{3}{2}\Delta a_R, \dots, 1 - \frac{1}{2}\Delta a_R$ and $Da = 0, \Delta Da, 2\Delta Da, \dots, 1$. The probability density $p(a_U|x_C, a_R)$ was then obtained at points $Da = \frac{1}{2}\Delta Da, \frac{3}{2}\Delta Da, \dots, 1 - \frac{1}{2}\Delta Da$ as the derivative using

$$p(\frac{n}{2}\Delta Da|x_C, a_R) = \frac{P(\frac{n+1}{2}\Delta Da|x_C, a_R) - P(\frac{n-1}{2}\Delta Da|x_C, a_R)}{\Delta Da} \quad (4.7)$$

where n is an odd integer. Finally the ratio $\eta(x)$ at these grid points was computed with the result from Section 4.2.1.

4.2.5 Target locating

Left to do was to find the pixels that maximizes $\eta(x)$. The values of $\eta(x)$ was however not yet known for each pixel, only on a grid in the Da, a_R -plane. Each pixel was given the same value as the closest grid point functioning like a piecewise constant interpolation. Once again this was the same approach as for the MATLAB scripts. By indexing the grid points $0, 1, \dots$ in the a_R -direction, the index to choose for a pixel x_0 was $\lfloor \frac{a_R(x_0)}{\Delta a_R} \rfloor$. The same applied in the Da -direction. The final probability was not necessary to compute for every pixel since finding the maximum of $p(x_T|Da, a_R)$ equals finding the maximum of $\eta(x)$. Before finding the maximum however, the $\eta(x)$ -image had to be median filtered, see Section 4.2.6. As with the maximum, the median could also be calculated for $\eta(x)$. After filtering, the pixel with the highest value of $\eta(x)$ was found by searching through all pixels in the image and compare each one with the so far highest value. The k potential targets were found one by one by finding the maximum and after each target had been found the pixels within a distance d_{\min} had their values for $\eta(x)$ set to zero to not find the same target twice.

4.2.6 Median filtering

One way median filtering can be done, is to for every pixel in the image, independent of each other, sort all pixels or elements within the filter window and take the middle value. Yet, not all elements in the window has to be sorted to find the median and there are fast selection algorithms for doing this [10]. This is however in many cases not the most effective way of filtering a whole image, especially when the filtering window is large and it takes more time to obtain the median. Since two filtering windows next to each other contains a lot of the same pixels some of the information used to obtain the median in one window can be carried over to the next.

There have been several studies on optimization of median filtering [11, 12]. Most applications have however been on images containing pixel values ranging from 0 to 255. These make up the normal digital representation of images or pictures made to look at. Often the median filtering of these images have been making use of bin sorting by putting the pixel values in a histogram with 256 bins from which the median is obtained. The window is moved to the adjacent pixel and the histogram is updated by removing/adding the pixels that left/entered the window. The result is a combination of fast sorting and only have to partially update the histogram at every iteration. The sorting method using a histogram works very similar to the bucket sort algorithm.

0	0	1	2	3	5	5	8	9
(0,0)	(2,0)	(1,1)	(0,2)	(2,1)	(0,1)	(1,0)	(2,2)	(1,2)

Figure 4.2: The information from the window in Figure 2.3 sorted.

Back to this median filtering problem, the $p(x_T|Da, a_R)$ - or η -“images” could in theory have an infinite amount of different values and were only limited by the precision of the data representation. So a few pixels with a big range of values had to be sorted. This made it hard to make use the same type of sorting in an effective way and thus, a different way of doing 2-dimensional median filtering was developed making use of comparison based sorting methods. For the median filter the window size was $m \times m$ where m was assumed to be an odd number. The filtering was done row by row where the window was fully sorted for every pixel. The sorting utilizes information from the previous pixel on the same row and from the pixel in the previous row but on the same column. The filtering for the first row and column was therefore slightly different. The edges were not filtered resulting in an image where $\frac{m-1}{2}$ pixels had been cut of from each side.

4.2.6.1 Filtering the first row

For the first pixel in the first row all pixels in the window were sorted using quicksort. The result was an array containing elements in this case sorted by their η -values from smallest to largest. The elements were represented by a structure in C++ which apart from η also held the information about which row and column the pixel had in the non-filtered image. Only a pointers are sorted to avoid copying extra amounts of data. As an example consider Figure 2.3 where $m = 3$. For this image the sorted array which is denoted A_1 is seen in Figure 4.2. The median is the middle value (3 for the example). The choice of sorting method was not very important since this sorting of m^2 elements was only for one pixel in an image that could contain millions of pixels.

For the next pixel one step to the right on the first row, the median was obtained by first considering the previous window. Looking at the example again, there are m elements not in A_1 that has to be added and m elements in A_1 that has to be removed, in order to obtain the new window. Figure 4.3 illustrates how the two different windows are related. A merging method based on merge sort was created that merges the new elements with the old window ,while also removing the elements no longer in the window. To do this, the new elements first had to be sorted and quicksort was used one again. For a large image and small window this sorting of m elements was done approximately once for every pixel in the first row and later

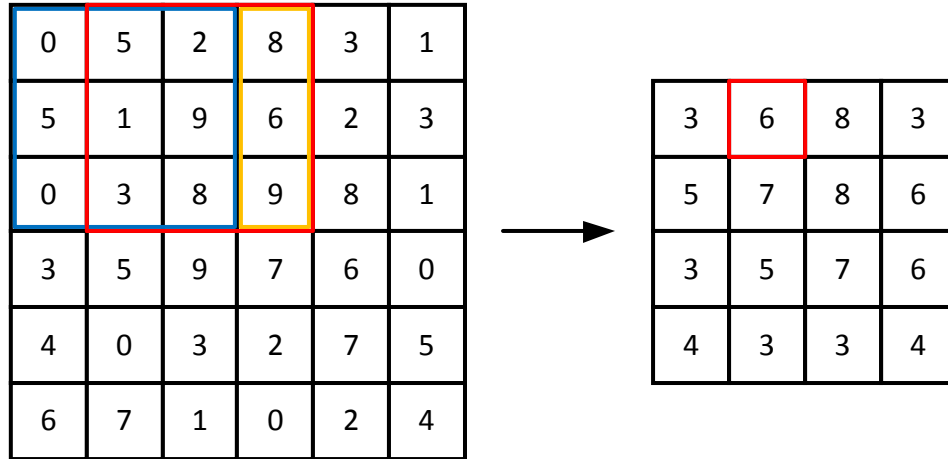


Figure 4.3: The second pixel the filtered value is computed for is marked red on the right and the corresponding filtering window in red on the left. The previous window in blue has been sorted in an array A_1 and the elements in the yellow area in an array B_1 .

also once for every pixel in the first column. This was still a small fraction of the total number of pixels in an image and any established sorting method would have been sufficient.

Going back to the example, A_1 and the sorted array of new elements B_1 are merged together into an array C_1 of the same length as A_1 using a merging technique similar to what is used in merge sort. Before the first elements in A_1 and B_1 are compared with each other however, it is checked whether the element from A_1 is still in the window. Hence, the need of a structure containing both value and position. Figure 4.4 illustrates how a merging works. The first element in array A_1 is looked at first. If it originated from in the first column it is removed from array A_1 . Otherwise it is compared with the first element in B_1 . The smallest of them is in that case removed from its array and added to C_1 . This is repeated until C_1 is full after which the median is obtained as the middle value. The merging was used repeatedly until the whole first row had been filtered.

4.2.6.2 Filtering the second and subsequent rows

The second and following rows made use information obtained from the prior row. For the first row, the results from quicksort were therefore saved to memory. This included the initially sorted window array of size r^2 as well as the smaller sorted arrays of size r . At the same time as the filtered values for the second row were computed, the same information was stored to be used for row three and so on.

The filtering of the second and subsequent rows used the same general principle

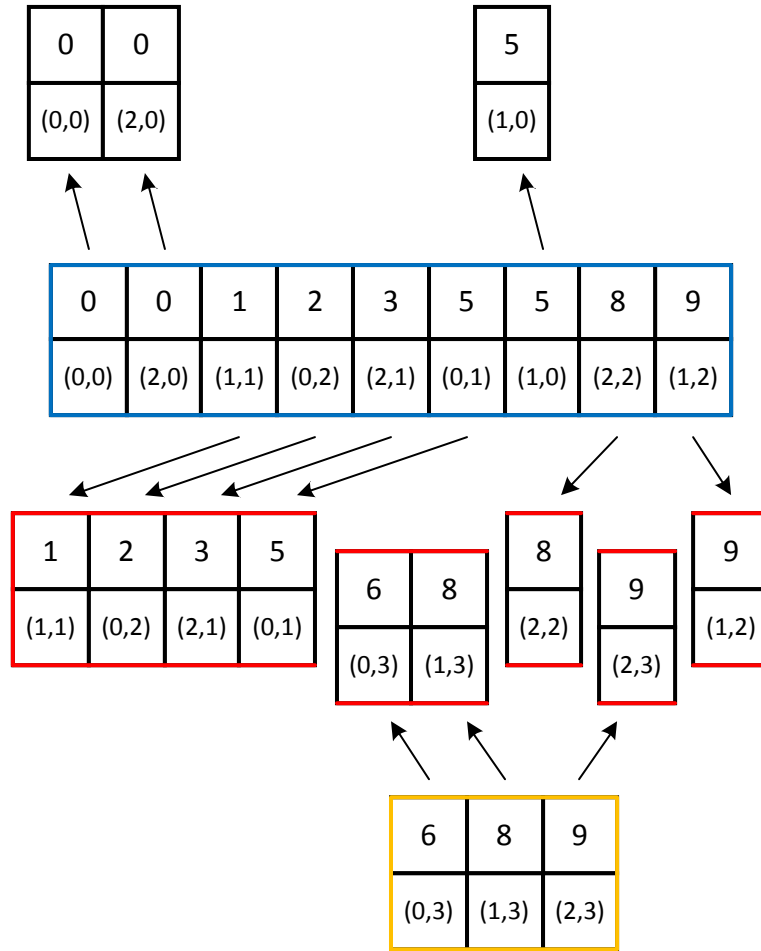


Figure 4.4: Illustration of the merging. A_1 (blue) is merged with B_1 (yellow) into C_1 (red).

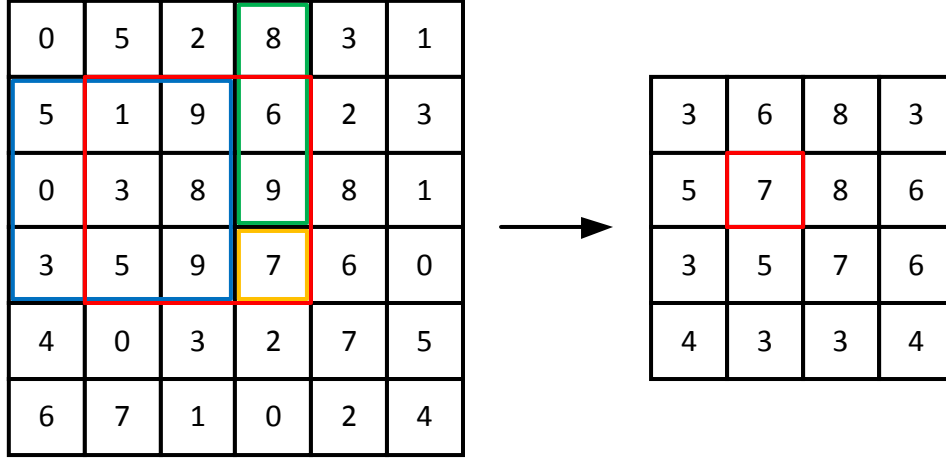


Figure 4.5: The filtering process for all pixels not in the first row or column (in the filtered image). Here the previous filtering window (blue) has a sorted array A_2 . There is also a sorted array B_1 saved from before (green). There is also one element (yellow) not in A_2 or B_1 that is included in the current window.

as for row one with a few key differences. The first difference was that there was no need to initially sort $m \times m$ values for each row. Instead, the first column was processed in exactly the same way as the first row, so only m elements are sorted and then merged with the sorted window from the previous row. For the rest of the columns the quicksort was replaced with a merge. An illustration can be seen in Figure 4.5. This time, there is a new sorted array A_2 and a sorted array saved from the previous row B_1 . The desired sorted array for the window C_2 requires a merge between sorted arrays A_2 and B_2 , just as for row one. Instead of sorting B_2 starting from scratch it is noticed that B_1 and B_2 only differs by one element. Therefore B_1 can with the same merging technique as before merged with the new element to create B_2 .

4.2.7 Target evaluation

The final part of the change detection was not implemented by Hellsten but was added to fully automate the process and to be precise, make two decisions. First there had to be a decision on when to stop the change detection i.e. do no more iterations. Then there also had to be a decision on which of the target nominees that should be considered targets and given as the output. For the first decision a user was given two options. Change detection could either be set to run a fixed number of iterations k_{\max} or it could be set to try and stop automatically. The automatic stop made use of the observation that targets within an image were found to end up as the first target nominees. Looking at the properties of the estimation to $p(a_U|x_C, a_R)$, when targets are removed the estimation changes since a target is different from the

clutter. Removing only clutter however, does not change the estimation. Thus, if probabilities for all nominees (which are affected by the estimation of $p(a_U|x_C, a_R)$) had seem to converged and the next found nominee had a low probability, it could be assumed that all targets had been found.

The automatic stop was implemented by taking two parameters Δp and $k_{\Delta p}$ as input. A target nominee can be said to have converged if for $k_{\Delta p}$ iterations the probability of it being a target have not increased by more than Δp , where $k_{\Delta p}$ started counting from the last time time the probability had an increase by more than Δp . If all target nominees that had been around for $k_{\Delta p}$ iterations had converged and the rest had a probability lower than Δp , the algorithm would stop. Since the number of assumed targets affected the probability it was for this decision always assumed $k = 1$ targets for the first target nominee (with the highest probability), $k = 2$ for the second and so on.

Before the change detection was finished the target evaluation would also decide which of the final nominees that were to be considered targets. This was done by comparing the probabilities with a threshold p_t where targets have a probability greater than p_t . From equation (2.3) it was noticed that the number of assumed targets k affects the probability. Therefore, if the number of nominees decreased after the threshold check, the probability was computed again with the new k . Since the probabilities changed, there had to be another threshold check and this was repeated until the number of targets did not decrease anymore. Thus, the value for k that had been used for the probabilities was equal to the number of outputted targets, while keeping all probabilities greater than p_t .

4.3 Parallelization

When parallelizing the change detection algorithm, there were two approaches that were considered. The first was to try and look into how to parallelize the different components of the change detection. The second was to keep the algorithm sequential but have several change detections running at the same time. The latter was chosen due to the nature of the algorithm. SAR images are often taken over a large area around 6 km². The scenery may differ a lot in different parts of the image causing the clutter distribution to also be different. Therefore it was natural to split a SAR image into smaller parts and doing change detection on each part separately. These smaller part will for future reference be referred to as sub images.

Having different sub images processed in parallel is from the above statement appropriate. It is also much easier than having to go through and parallelize all components of the change detection. Also, if there were parts of the change detection that could not have been made parallel with the former approach, causing limitations given by Amdahl's law, it did not matter with the latter. A big disadvantage with the chosen approach was however that the processor cores stops working at different

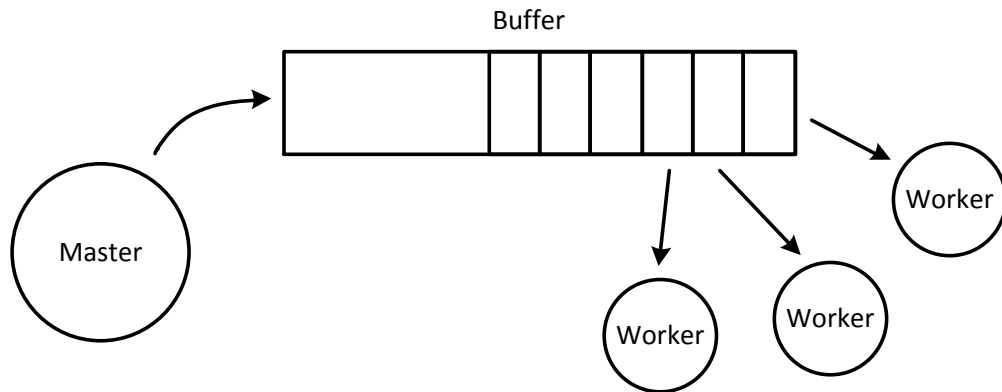


Figure 4.6: The master-worker model. A master thread puts items in a buffer and the worker threads removes them and do some work associated with them. For change detection, the master divides an image into several sub-images and the workers do the change detection on the sub-images.

times. For example if there are 16 sub-images to do change detection on but only twelve cores, each core would work on one sub-image leaving behind four. Then, if the first twelve are finished in an equal amount of time there are only four sub-images left leaving eight of the cores to do nothing. These cores could however do something else which is why a pipeline structure was implemented, allowing the other cores to work on something else such as the creation of SAR images.

4.3.1 The master-worker model

The change detection computations were split between the different processor cores using something similar to a so called master-worker, master-slave or boss-worker model [17]. The model consists of one master thread and one or usually more worker threads. As different threads they could do computations concurrently and by creating a number of threads at least equal to the number of cores, they were able to run in parallel to take advantage of all cores.

The job of the master thread was to simply to distribute the sub images to the workers. Since the change detection had to handle a constant stream of data, a buffer system was implemented where the master thread put work in a buffer. The workers would pick an item from the buffer which contained data (a void pointer) and instructions on how to process the data (reference to a function with a void pointer as input parameter). After that, the workers would do the work by processing the data (executing the function passed) and then proceed to pick another item from the buffer. The master-worker model is illustrated in Figure 4.6 and was implemented as a class using pthreads and was used to parallelize change detection, see Section 4.3.2.

Since both the master thread and the worker threads wanted to access the buffer at some point it was possible for race conditions to occur which had to be prevented. Therefore, mutexes were used to allow only one thread to modify the buffer at a time. Condition variables were also used to put worker threads to sleep while the buffer was empty or to put the master thread to sleep if the buffer was full. In pseudocode the master thread would access the buffer according to Algorithm 1. The workers would access the buffer according to Algorithm 2.

Algorithm 1 Access the buffer with the master thread.

```

lock( $m$ )                                ▷  $m$  is a mutex
while buffer is full do
    wait( $c, m$ )                            ▷  $c$  is a condition variable
end while
Write to buffer
if buffer has exactly one item in it then
    broadcast( $d$ )                            ▷  $d$  is a condition variable
end if
unlock( $m$ )

```

Algorithm 2 Access the buffer with a worker thread.

```

lock( $m$ )                                ▷  $m$  is a mutex
while buffer is empty do
    wait( $d, m$ )                            ▷  $d$  is a condition variable
end while
Read from buffer
if buffer has exactly one empty space then
    signal( $c$ )                            ▷  $c$  is a condition variable
end if
unlock( $m$ )

```

Algorithm 1 is explained as follows. When the master thread wants to put something in the buffer it starts by locking the mutex. If the mutex is already locked the thread has to wait and will be put to sleep until the mutex is unlocked. If the buffer is full, the thread has to wait for one or more items to be removed. It will therefore unlock the mutex and wait for a signal from one of the worker threads. When the buffer has space the master thread will lock the mutex again and can finally put an item in the buffer.

If an added item is the only one in the buffer there is a possibility of there being worker threads waiting for items similarly how the master thread will wait when the buffer is full. The master will therefore broadcast to signal all worker threads sleeping. After finishing the broadcast, or if there are more than one item in the buffer, the mutex will be unlocked.

If broadcasting is not supported, a similar behaviour can be achieved by signaling one sleeping worker thread every time an item is added to the buffer or by introducing an additional mutex. The worker threads go through a loop in which they first access the buffer and afterwards proceed to process the data. The way a worker would access the buffer is very similar to that of the master thread.

4.3.2 Parallel filters

The master-worker class was utilized in another implemented class. This filter class was made with the purpose of receiving SAR-images and performing change detection on them. It was however generalized so that any signal processing step could use its thread and buffer structure and was implemented as an abstract class where different filters such as change detection are classes derived from it. Multiple of these filters can create a pipeline structure [17] where every step in the pipeline is a filter with its own set of workers.

The implemented filter structure made it possible to run separate filter steps at the same time so that a new image could start being processed at the same time as change detection was performed on the previous image. It also made it possible for several change detections on different images to run at the same time by keeping track of multiple images.

Two new threads were introduced, the producer thread and the consumer thread, where the producer can be seen as the master thread. The producer was given the job to remove data from an input buffer and distribute work to the worker threads. Once all work associated to that data has been finished the consumer generates the output and place it in the input buffer to another filter. This creates a chain of filters which order can be modified for the user's need and is suitable for the Carabas signal processing chain. An illustration of the filter structure is seen in Figure 4.7.

Synchronization primitives were also used here in a similar manner to prevent two threads from accessing shared resources at the same time and to put threads to sleep when buffers are full or empty. The reason the master thread was replaced by two different threads was that it was easy to give flexibility to the program and prevent deadlocks. For instance, the consumer in one filter would sometimes have to wait and be put to sleep when outputting the result. With a producer thread the rest of the filter could however continue to receive and process data.

4.3.2.1 Parallel change detections

The parallelization of change detection was achieved with a class derived from the filter class. This CD-class has instructions specific for change detection but could be modified for other types of computations. A full filtering process starts with an item being added in the input buffer. For change detection, this is a structure that

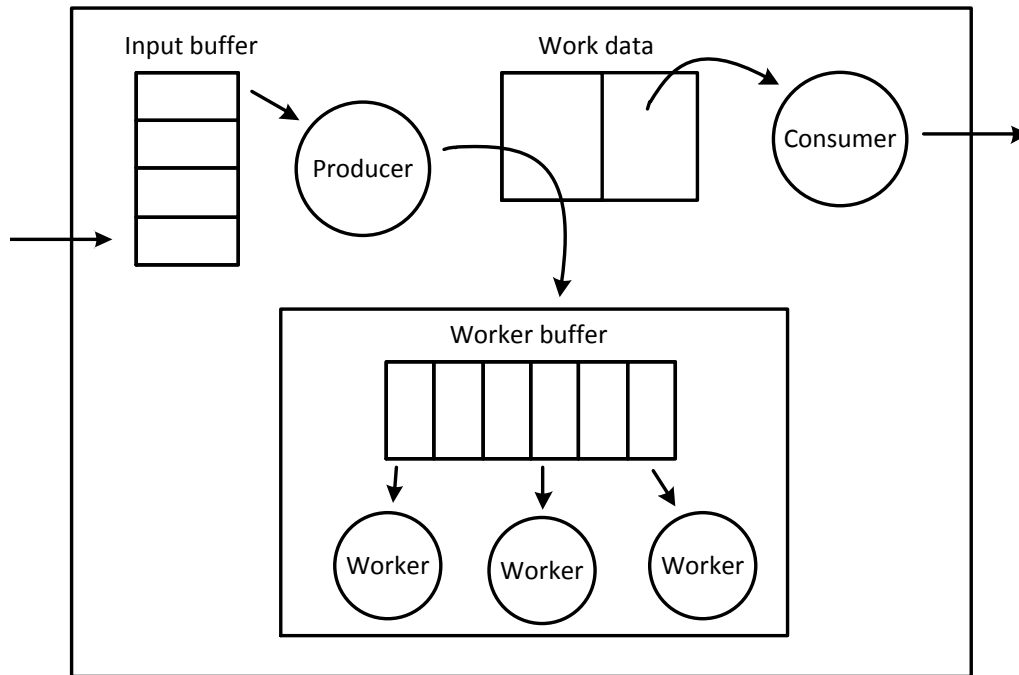


Figure 4.7: The structure of a filter which was implemented to be used for change detection and other types of signal-/image processing.

contains

- A newer SAR-image (update)
- An older SAR-image (reference)
- Change detection parameters from table 4.2
- Parameters defining a grid of sub images
- Parameters on artificial generation of targets
- An array of found targets (empty before change detection)
- An array of artificially generated targets (empty before change detection)

where generation of targets are discussed later in Section 4.4.2. The item is picked up by the producer, which in turn puts it in a work array for data that is being processed. It then distributes work to the worker threads in a way specified by the derived class. For change detection the producer acts like a master thread from Section 4.3.1 and adds items to the worker buffer equal to the amount of sub images defined. It also creates a list where found targets are stored which with the two images, definition of sub area and CD-parameters are passed as data to the worker buffer. The producer then proceeds by going in a loop and starts by looking at the input buffer again.

The worker function used by the workers is also defined by the derived class. In the case of the CD-class, it first performs change detection and then write the result to the list. When all sub images are finished, the consumer thread takes the item

in the work area and puts it in an output buffer, which can be the input buffer to another filter. The SAR images are not changed in the change detection case, so the consumer has an additional task to add all found targets to an array in the item.

For the consumer to know when a set of data has been fully processed a counter associated to that data set is used. The counter starts at zero and after the producer is finished with adding items to the worker buffer it increments the counter by an amount equal to the number of items it added to the buffer. When a worker is finished with processing its subset, the worker decrements the counter by one. When a thread changes the counter it also checks whether the counter is zero afterwards. If it is, all workers and the producer must be finished and the consumer that is most likely sleeping is signaled.

More than one set of data can be processed at a time (such as change detection on two different image pairs). Hence, two list, "ready" and "done", containing indices are used to keep track of the different data sets. The producer checks the first item in the ready list, removes it and adds the data to the corresponding place in the work data array. When the data is processed and the consumer is signaled, the index to that data in the work data array also gets added to the end of the done list. The consumer keeps removing indices that are in the first place in the done list and adds them to the end of the ready list. Thus, the ready list contains the empty places in the work data array and the done list contains the data that has finished processing.

4.4 Verification and testing of the implemented C++ program

To verify that the change detection method had been implemented correctly, it was compared with the existing MATLAB implementation. Using the same image pair as input, the outputs were compared with each other. This was done with three separate image pairs. Certain individual parts of the change detection method were also tested with extra data. For instance, the median filter was tested with some test matrices to ensure that the output was correct. The performance and performance gain due to parallelization was also observed by measuring the execution time.

4.4.1 Time measurements

The execution time was measured by looking at the wall time at start and end and taking the difference. A downside is that if there are other processes they may delay the one being measured. Since other processes did not consume a lot of resources during the measurements wall time was a valid choice. Measurements in C++ were made using the `sys/time.h` library (UNIX) and the function `gettimeofday()`. When

measuring time in MATLAB, the tic/toc stopwatch was used.

4.4.2 Artificial targets

To aid future testing of the algorithm, a way to create artificial targets was added. Creating new SAR-images that also have targets in them are expensive and time consuming. For instance, to see if the algorithm has a low enough false alarm rate, these kind of artificially created images might be required. Making use of the filter class, a new derived class was made that adds a template target to the updated image. The following parameters are considered and can be set or randomized with a homogeneous distribution:

- Number of targets
- Amplitude of targets
- Position of targets

An array of added targets is also saved to the output.

5

Results

The C++ realization of the change detection algorithm was in the end very successful. It produced exactly the same targets as with MATLAB and the execution time was much faster than what was expected from the beginning. This chapter presents a more detailed description of the different results.

5.1 Evaluation and comparisons

The output from the C++ program that is of interest for comparison is a list of targets found by the change detection algorithm. Each target have a position, which is the coordinate in the image, and a probability $p(x_T|a_U, a_R)$. Because of several differences with the MATLAB scripts the outputs are not expected to have exactly the same numerical values. Firstly, the MATLAB scripts included some extra processing parts that slightly altered the input data. Secondly, as previously discussed, the change detection function was implemented slightly different. Nevertheless, the two different programs are expected to have the same behaviour and the C++ program was, for the comparisons set, to mimic the behaviour of the scripts.

5.1.1 Comparison with MATLAB scripts

A part of an image pair that was used is seen in Figure 5.1. Here there are two real targets about 10m from each other. The C++ program was able find and locate these as two separate targets with probabilities of being targets greater than 99%. Also, no other target nominees with high probabilities were found within that area. This matches very well with what also was observed using the MATLAB scripts. The images were of size 36 Mpix but the change detection was done on smaller areas at around 1-2 Mpix.

With the time consuming MATLAB scripts, only the area around the known targets was studied. The much faster C++ program could however cover the whole image pair. What should be noticed is that large parts of the two images were corrupted as can be seen from a full 36 Mpix image in Figure 5.2. This is due to the so called

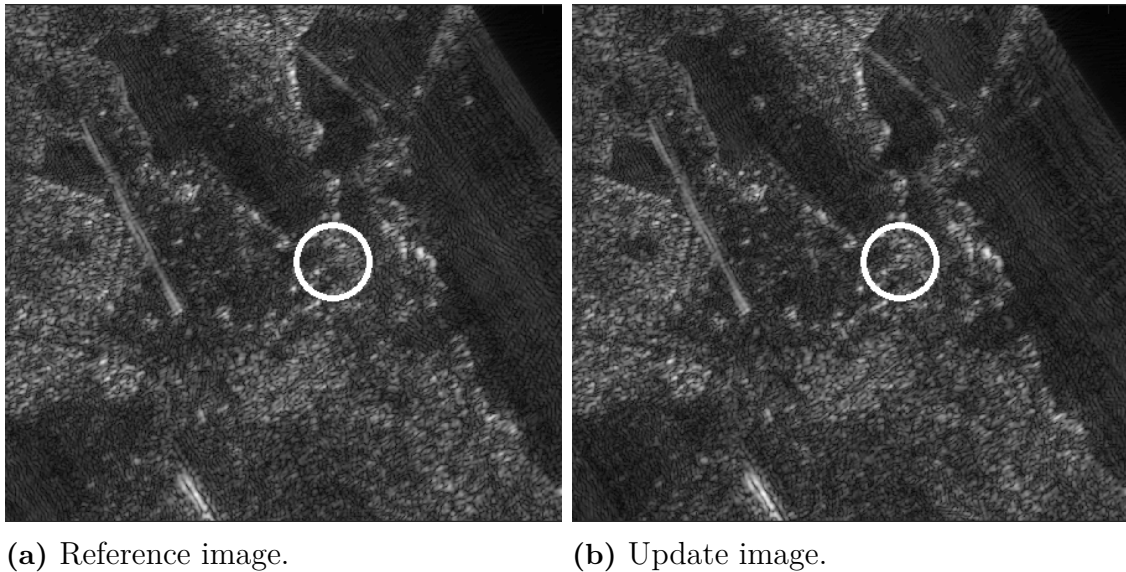


Figure 5.1: Illustration of two targets appearing in the reference image. They can be seen as two small dots in the center of the white circle in the update image. The circles are not a part of the original SAR images.

nadir echo reflected from the ground below the flight path. Normally this is avoided by not taking images directly below the aircraft. In the image, everything right of the nadir track is just a mirror image of what is left of it.

For a run with change detection on a grid with 36 squares of size 1 Mpix, 13 presumably false targets were detected. A first group of targets were found to the right of the flight path where there is no useful data. There were also targets found in the bottom left corner. Due to the nadir echo the two images differs too much creating unreliable clutter statistics. From Figure 5.3 this is apparent from what can be seen as diagonal lines and brighter spots in the images. Lastly, targets were also found around the dark areas in the bottom of the image pair. Here there is not enough data collected to accurately depict the area. For an area to have high resolution it has to be seen from a large range of angles looking from the aircraft. The flight path for both images in this pair was however rather short, resulting in worse resolution and image quality.

The example above shows that with unreliable data the method produced false targets. The intended use for change detection is nevertheless for streaming data with full resolution and without artifacts caused by the nadir echo. The method however worked as intended in the non-corrupted areas where the two real targets were found and no additional ones. The targets placements in the sub images did also not seem to matter. In fact, if a target was located on the border between two sub images it was detected twice, once in each sub image. An additional method might thus be required to prevent double detection of the same target. For a second pair of images containing one target, the C++ program was able to find the target just like the MATLAB scripts.

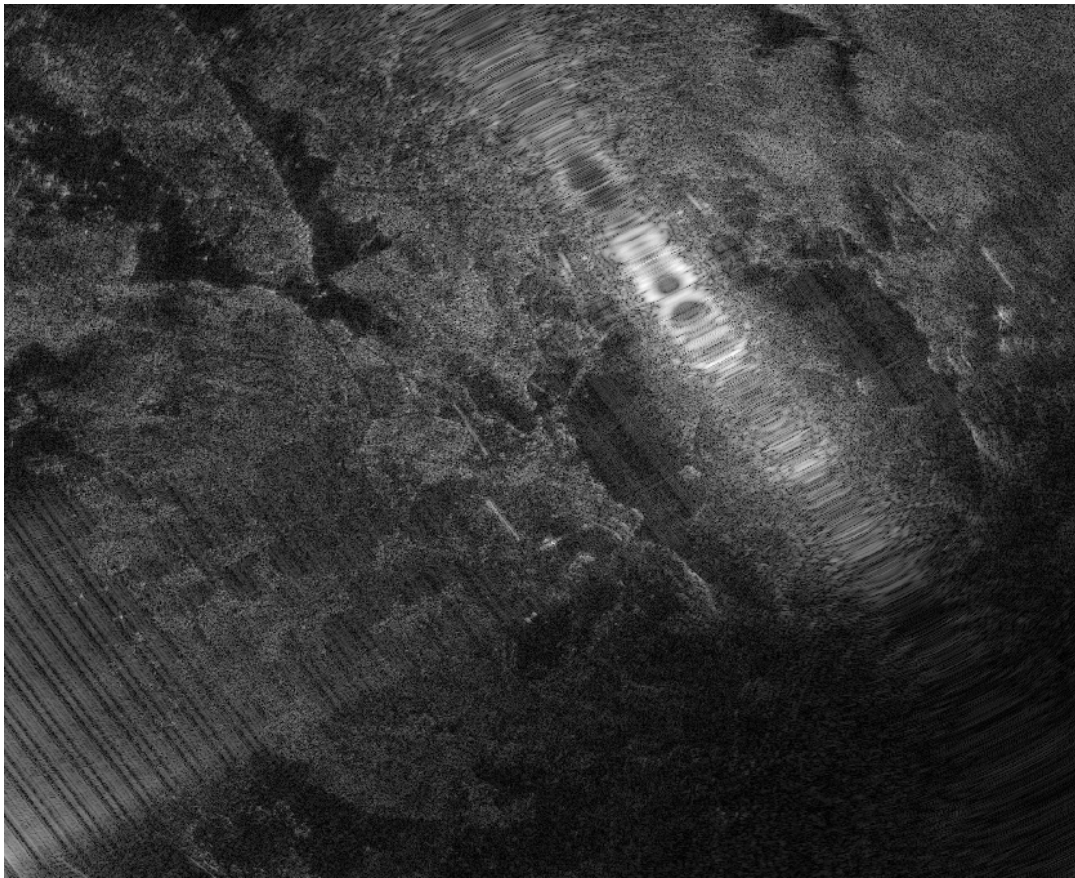
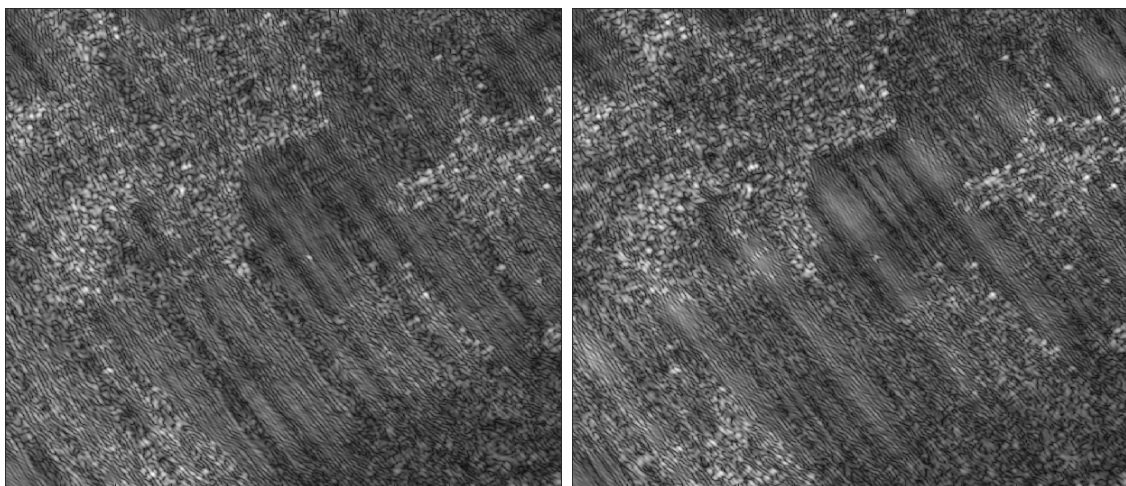


Figure 5.2: The full 36 Mpix reference image containing two real targets.



(a) Reference image.

(b) Update image.

Figure 5.3: Data that is corrupted which can be seen as diagonal lines within the images. Due to these lines differing, targets were detected despite there being no real deployed targets.

5.1.2 Detection of false targets

For testing purposes, one additional image pair from which the reference image is seen in Figure 5.4, was put through the change detection algorithm. Here there is as before, a nadir echo on the right hand side of both images. Therefore, the right side was neglected and only the left hand side was studied. No targets had been deployed between the images so in the best case scenario no targets should be detected. One object however caused the algorithm to detect targets. This object is seen in Figure 5.5 and is strongly reflecting. Comparing the SAR-image with an aerial photograph the reflection is likely due to a building with a metal roof. Some variations between the two images are (even if they should not be) detected as targets. As these variations are not typical for the sub image, $p(a_U|x_C, a_R)$ becomes low and thus $\eta(x)$ high unless $p(a_U|x_T, a_R)$ is also low.

The targets mentioned above may also be due to how $p(a_U|x_T, a_R)$ depends on the way targets are modeled using $p_T(a)$. In the current setup, the upper and lower bounds for amplitudes, a_{\max} and a_{\min} , are set to values between 0 and 1. The images are also scaled between 0 and 1 where 1 is the value for the pixel with the highest amplitude within the two images. With something highly reflective and with a large amplitude, a_{\max} and a_{\min} , which are chosen relative to this point instead of the forest around it, may become incorrect. Thus variations that are of greater amplitude than intended are seen as targets.

5.2 Execution time and parameters

The main goal of the thesis was to make the change detection run automatically in real time. The real time limit was measured by comparing the execution time of the C++ change detection implementation to the creation of SAR images with Carabas. The creation of a 25 Mpix SAR image with the current system takes about 40 seconds, which sets a requirement on the maximum execution time change detection can have to still keep up with a constant stream of images.

The execution time was found to vary between different data, settings and runs. For a 25 Mpix image with parallelization the time can be expected to be around 5 seconds. For a single sub image of 1 Mpix using typical parameters (see Section 5.2.4), one iteration takes around 0.43 s and the computations before the loop around 0.16 s. Since the goal of 40 seconds was achieved rather easily there was not as much time spent on trying to optimize it further. Instead more focus was put on parallelization for a potential integration of other SAR signal processing steps. The time aspect was nevertheless a big part of the project and below follows some observations regarding execution time.

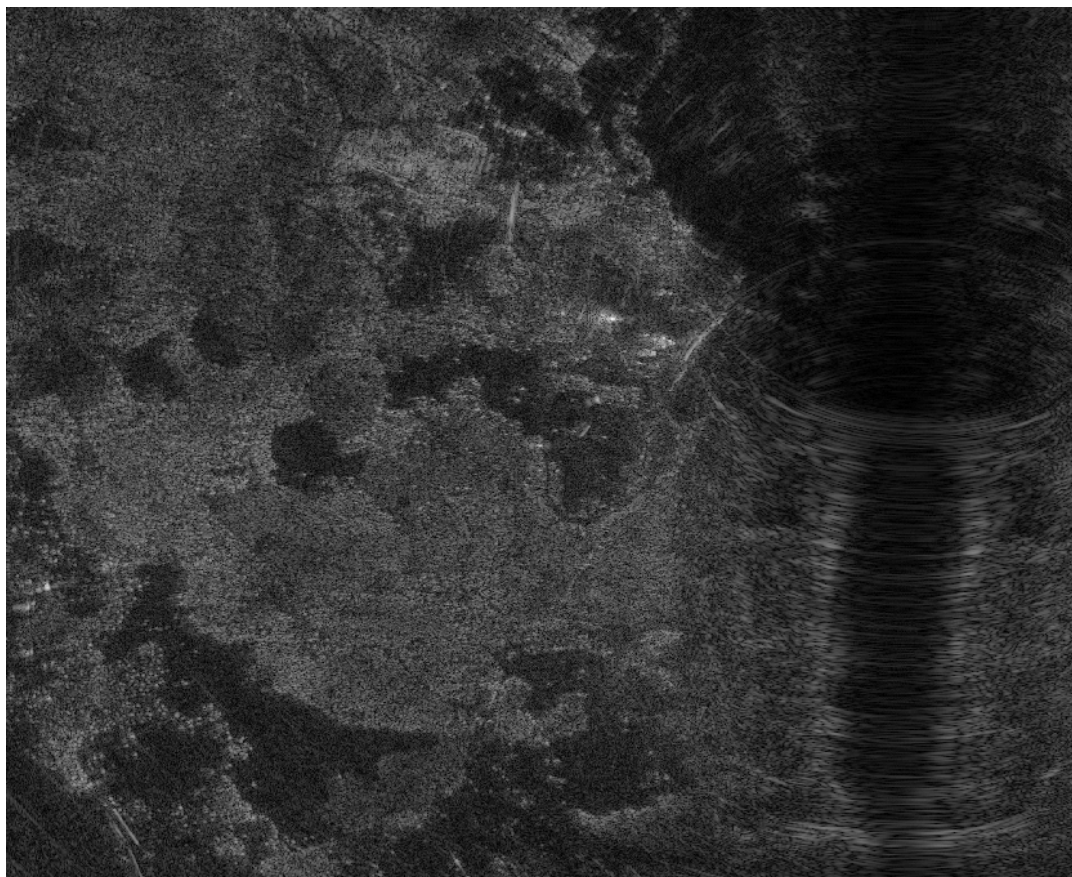
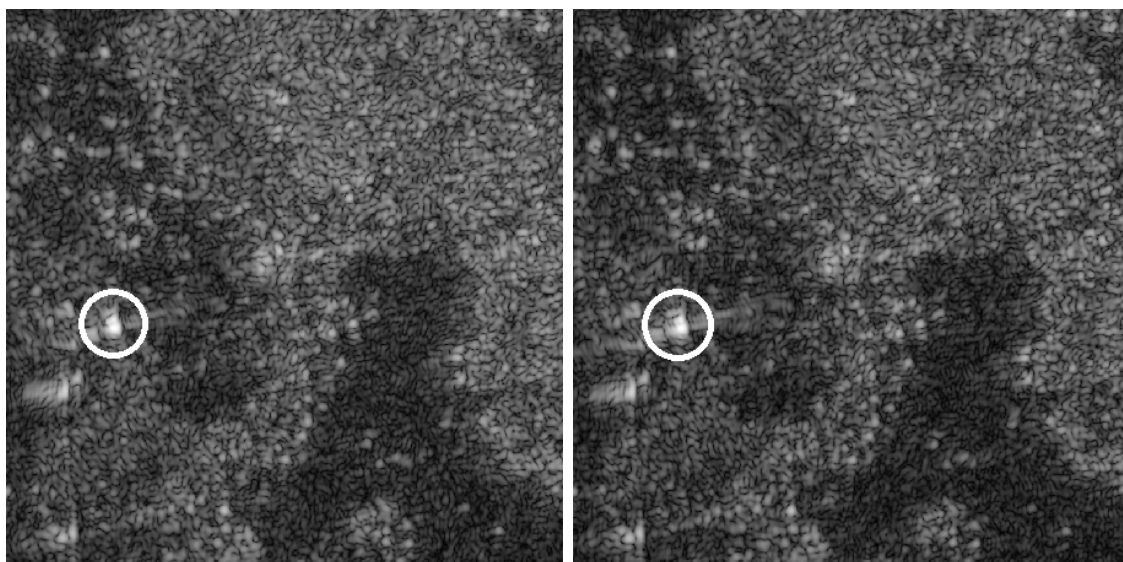


Figure 5.4: The full 16 Mpix reference image where a false targets were detected.



(a) Reference image.

(b) Update image.

Figure 5.5: False targets were detected by the bright spot in the middle of the circle. No big difference is seen between the two images. The circles are not a part of the original SAR images.

5.2.1 Automatic stop

One of the most potential aspects in terms of saving execution time is the automatic stop in the target evaluation. This functionality makes it possible, without changing any parameters, to find several targets in one sub image while still not using that many of iterations for all sub images. The time saved will be highly dependent on the data and parameters. A test was carried out with the image pair in Figure 5.1, where $k_{\max} = 3$ iterations are required to find both targets. Using the parameters $\Delta p = 0.2$ and $k_{\Delta p} = 2$ both targets were found and if the sub image was moved so that the targets were outside of it, the algorithm stopped after two iterations. Since targets are not expected most of the time, this potentially saves 33% of the computation time by not doing unnecessary computations. The percentage could potentially be even bigger if there is a desire to find more targets. Suitable values for Δp and $k_{\Delta p}$ would therefore have to be found that saves as much time as possible while not missing any targets.

5.2.2 Type casting

The most time consuming parts of the MATLAB script is when $\eta(x)$ is computed for every pixel in the image by interpolating from a grid. The grid point used for a pixel is obtained by comparing the image amplitudes for that pixel with the the grid points using the $<$ and $>$ operators. During the interpolation, each pixel's reference and update amplitude is effectively compared twice with each amplitude in the grid. So for 1 Mpix images and a 100×100 grid there is a total of 400 million comparisons (or 400 per pixel) which are also followed by some extra computations.

In the C++ implementation the same was achieved with type casts instead. The image amplitudes which are floating point values, are type casted to integer values. These integer values are used as indices to obtain the grid points. A test of the computation time for type casting and a more optimized comparison based interpolation was made. Instead of always comparing with all grid points, the comparison based method compared with them one by one until the correct interval was found. So the image amplitudes are checked whether they are smaller than 0.01, then if they are smaller than 0.02 and so on until the correct intervals are found. Random amplitudes were generated assuming a logarithmic distribution from equation 4.6. This averaged out to about 26 comparisons per pixel for the comparison based method and took for 1 Mpix 0.092 seconds. Type casting did the same in 0.014 seconds with two type casts per pixel. If the computation time scales linearly with the number of comparisons a direct translation from the MATLAB scripts would have taken 1.42 seconds. In that case 99% of the computation time was cut of by using type casting.

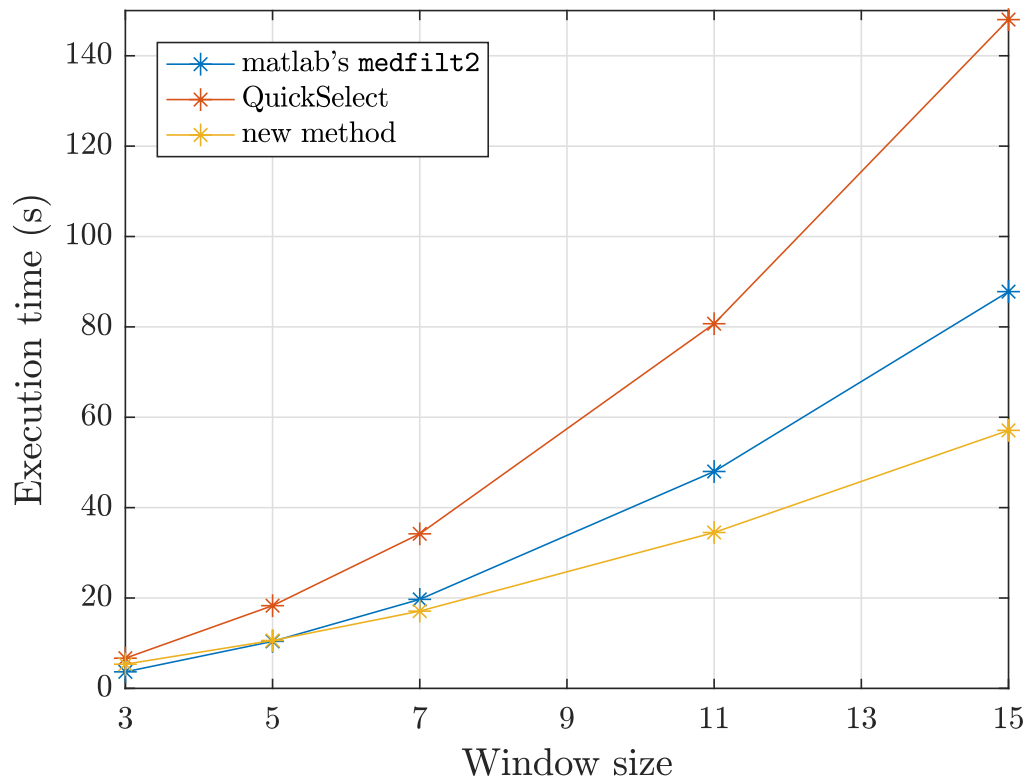


Figure 5.6: Computation time in seconds of median filtering using different window sizes m .

5.2.3 Median filter

In the final C++ implementation, the median filter uses around 90% of the computation time for an iteration in the change detection loop. The computation time of the median filter using different target or window sizes m on a 25 Mpix image is seen in Table 5.1 and in Figure 5.6. The new filter is compared with the MATLAB's `medfilt2` as well as an implementation of the quickselect algorithm [18], which is a fast method to determine the median for a variable amount of elements [10]. Looking at the results, the way of retaining information from one window to the next is faster than computing the median from scratch with quickselect. Comparing with `medfilt2` which also uses some kind of technique to retain information from one window to the next, the new method is slower for the smaller m but catches up when m grows larger. The size of a target m was typically 5 (which equals $5 \times 5 = 25$ pixels). It was however desirable to be able to change to a larger target size.

Since of the median filter consumes most of the computations, for faster change detections, faster median filtering is also required. For smaller m there are median filters faster than quickselect [10]. The downside being that they only work for one target size. A similar approach for sorting in conjunction with the new median filter could potentially speed up the process for certain values of m . It could also be tested

	$m = 3$	$m = 5$	$m = 7$	$m = 11$	$m = 15$
MATLAB's <code>medfilt2</code>	3.66	10.4	19.7	48.0	87.8
Quickselect	6.68	18.3	34.2	80.7	148
New method	5.35	10.6	17.1	34.5	57.1

Table 5.1: Computation time in seconds of median filtering using different window sizes m on a 5000×5000 matrix containing double precision floating point values.

how necessary the median filtering is and if the window size has to be the same as the target size. If the median filtering is reduced but it does not affect how well targets are found, then the execution time will be faster. Finally, a faster execution time might not be the first priority. The current SAR signal processing, which is now slower than the change detection, could instead be investigated for potential improvements.

5.2.4 Parameters

What causes different execution time for different data sets is primarily the target evaluation. For instance, the change detection of the image pair where one image is seen in Figure 5.2, resulted in several target detections. Since there are multiple high probability targets (true and false ones) it takes a longer time for all target nominees to converge which makes the algorithm run for more iterations. In the end, the execution time is longer than if there are no targets.

Changing some of the other change detection parameters also did also impact the execution time. The parameters that were found to affect computation time are the following:

- Size of a change detection sub image. This is the part of a SAR image used for change detection. A larger sub image includes more pixels and will thus require more computations. An optimal size should not have anything to do with the execution times but rather the clutter statistics. The sub image should have enough data represent the clutter while still accurately represent every part of the image and not be an over-generalization. Here sizes of around 1-2 Mpix had been used for the successful change detections.
- Number of sub images from one (or multiple) SAR images. Since different sub images uses different processor cores, which there are twelve of, there can be up to twelve sub images without impacting the computation time that much. The sizes and number of sub images goes hand in hand. If change detection is to be carried out on a full SAR image, increasing one results in a decrease of the other, thus in the end not impacting the execution time.
- Parameters deciding the number of iterations: k_{\max} , Δp and $k_{\Delta p}$. Se Section 5.2.1.

- Size of the interpolation grid. Controlled with the parameters Δa_R and ΔD_a . Making this grid contain more points could slow down the program. The number of bilinear interpolations and differentiations are in that case increased. A higher number of grid points than the used $100 \times 100 = 10000$ have not yet shown to yield any advantages in change detection capabilities. An increased execution time will likely also be the case if the number of histogram bins are increased to be much larger. This is however not practical because there would not be enough data points to fill all bins.
- Size of a target for median filtering. So far, the filtering window has been the size of a target. A bigger target would mean more numbers to sort for each median and thus more computations. An instance where a larger target would be used is if the images had a higher resolution. This is the case if the higher frequency band in Carabas is used. Because of the median filter represents around 90% of the computations, changing the target size will severely impact the computation time.
- Choice of interpolation methods. There is currently no way of changing interpolation method within the C++ program without editing the code. If however the interpolation method is changed it should be noted that a computationally heavy interpolation method will impact the performance.

5.3 Parallelization with OpenMP and pthreads

With parallelization, it was possible to utilize the hardware fully. The API pthreads was used for the final implementation but a master-worker model was utilized with OpenMP as well. A comparison between the two is seen in Table 5.2. The speedup and efficiency is however dependant on the input images, more precisely the number of sub images and the difference in execution time between the sub images. For example if there is only one sub image, only one core will be used and the execution time will not be any faster.

This test was done with 36 sub images from Figure 5.2 where there is one sub image for each processor core. This could be seen as a best case scenario where all cores do the same amount of work. However, the two targets, the additional false targets and other differences caused the sub images to take different amount of time still causing some of the cores to finish before others. While this is only one example and do not tell how well the parallelization works in general or on average, it still works as a comparison between pthreads and OpenMP. As can be seen from the table, pthreads and OpenMP performs almost identically. The OpenMP solution was easier to implement as it only required to add an extra line in front of the for-loop that was already used for the sequential test. Pthreads on the other hand required more commitment but allowed for more flexibility which ended in the filter class.

Method	Execution time (s)	Speedup	Efficiency
Sequential	45.6	1	0.0833
OpenMP	4.73	9.64	0.803
Pthreads	4.72	9.66	0.805

Table 5.2: Execution time and speedup for change detection on one SAR-image divided into 36 sub-images.

Method	Execution time (s)	Speedup	Efficiency
Sequential	57.3	1	0.0833
Pthreads	5.01	11.4	0.953

Table 5.3: Average execution time and speedup from streaming SAR-images.

5.3.1 Streaming data

Results for how the final filter class performs with streaming data is seen in Table 5.3. The change detection was done 100 times in a row so that when there are no sub images left in the first image, the leftover cores immediately start working on the next image. This mimics how SAR images would be created in a real-time environment. The efficiency achieved at 95% is very close to the theoretical limit at 100% and not much more could be asked for. Images were added as soon as the input buffer had empty spaces but it could also work in a setup where new images are generated and added over time. In practice, if a new SAR image is created every 40 seconds and change detection takes around 5 seconds, it is unlikely to be more than one image processed in change detection at once. However it is possible to do change detection more than once on using same image but with different reference images or parameters.

With the high efficiency there is also a potential in including existing signal processing that is not as efficient as this. If multiple C++ filters like the one implemented for change detection are added together, the same efficiency is not guaranteed. There were a total of 15 threads used here. One main thread creating the others and adding images to the input buffer and 14 tied to the filter where 12 are worker threads. As the workers are responsible for the majority of the computations, most of the time there are 12 threads divided between 12 cores. By chaining multiple filters together there will be more threads competing for the same resources. It was noted that increasing the number of workers past 12 seemed to increase the execution time, which may be due to resources spent on changing between threads. Limiting the number of threads and looking into scheduling would be reasonable actions to take if there would be a significant loss in performance. Scheduling can be used to help the operation system to decide which threads should be prioritized.

5.4 Hardware and Limitations

The bottleneck for the change detection computations were the processors where all available resources were used when running the program. With no other resource heavy processes there was (almost) 100% usage per core. Memory usage was only a small fraction. For the streaming data case with five images loaded at the same time, where two of them were being processed, the memory usage was around 6%.

Since the execution time is already sufficient for Carabas, looking deeper into hardware is not necessary for real-time change detection. If however, the change detection is executed on other hardware, there will eventually be a decrease in efficiency as the number of cores are increased. The number of cores that can be utilized for a SAR-image equals the number of sub images that it can be split into. The sub images cannot be too small as they require a certain amount of data for accurate clutter statistics. Different SAR-images can still be processed in parallel but an image is likely to be finished with change detection before a second image is ready. It is nevertheless possible to perform multiple change detections on one image with different reference images or parameters.

6

Conclusion

SAR-imaging and change detection can be helpful in both civilian and military applications. A new change detection method made to detect targets hidden in the forest has been realized in the form of a C++ program which has been used to show that real-time change detection for use with Carabas is possible. The algorithm was parallelized with POSIX threads to utilize a multi-core processor architecture, which resulted in 95% efficiency. The program has also been structured to allow for further testing of the change detection algorithm.

False targets has been detected which shows that the algorithm is not in a perfect state and has room for improvements. The false targets were detected both due to bad data and due to a strong reflecting object in the images. The latter is a behaviour that is undesired and is something to look further into. With the algorithm running much faster than previously and being fully automated it will be much easier to both find and analyze shortcomings like these.

The median filtering is responsible for around 90% of the computations. The new median filter method did save time over using a fast implementation of the selection algorithm quickselect. It is however still possible that even more time can be saved with another approach. For faster change detection it can also be worth looking more into the hardware and the CPU. Further analysis could reveal if there are ways to utilize the hardware better by for instance studying the cache performance. Alternatively there is also the option to use more powerful CPUs.

The next step for change detection with Carabas is to integrate it with the rest of the system. The C++ classes created allows for CD-processing at the same time as other computations and could be utilized to increase the efficiency of pre-existing computations. The artificial target creation can be used to determine accurate false alarm ratios to see which specifications the algorithm meets without having to spend resources on real flights. Finally, with a fast running algorithm, it will be possible to optimize the algorithm by varying different parameters. Both in terms of false alarm rates and execution times.

Bibliography

- [1] Eriksson, J. (2015). *Realization of efficient change detection in SAR images*. MSc Thesis, Chalmers University of Technology, Gothenburg
- [2] Hellsten, H. (2017). *Meter Wave Ground Imaging Radar*. 1st ed. Norwood, United States: Artech House.
- [3] Hellsten, H. (2016) *CARABAS subsurface polarimetric change detection based on Bayes theorem and histogram clutter statistics*, provided from Saab.
- [4] Fearnside, P. and Laurance, W. (2004). Tropical Deforestation and Greenhouse-Gas Emissions. *Ecological Applications*, vol 14, no 4, pp.982-986.
- [5] Nilsson, S. (2001). *Do We Have Enough Forests?*, American Institute of Biological Sciences.
- [6] Moreira, A., Prats-Iraola, P. Younis, M., Krieger, G., Hajnsek, I. and Papathanassiou, K.P. (2013). A Tutorial on Synthetic Aperture Radar. *IEEE Geoscience and Remote Sensing Magazine*, vol 1, no. 1, pp. 6-43.
- [7] Saab Solutions. (2017). *Carabas airborne reconnaissance system*. [online] Available at: <http://saab.com/air/sensor-systems/ground-imaging-sensors/carabas/> [Accessed 27 May 2017].
- [8] Shapiro, L. and Stockman, G. (2001). *Computer vision*. 1st ed. Upper Saddle River, N.J.: Prentice-Hall.
- [9] Welford, B.P. (1962). Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, Vol 4, No 3, pp.419-420
- [10] Devillard, N., (1998) *Fast median search: and ANSI C implementation*. [online] Available at: <http://ndevilla.free.fr/median/median.pdf> [Accessed 27 May 2017].
- [11] Huang, T.S., Yang, G.J. and Tang, G.Y. (1979). A Fast Two-Dimensional Median Filtering Algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol 27, no. 1, pp. 13-18.

[12]

- [13] Karunanithi, A.K. (2014) *A Survey, Discussion and Comparison of Sorting Algorithms*, Umeå university Weiss, B. (2006). Fast median and bilateral filtering. *ACM Transactions on Graphics*, vol 25, no 3, pp 519-526.
- [14] Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching*, vol 3 (2nd ed.), Boston, United States: Addison-Wesley
- [15] Peng, L., Peir, J., Prakash, T., Chen, Y. and Koppelman, D. (2007). Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study. *IEEE International Performance, Computing, and Communications Conference*.
- [16] Pacheco, P. (2013). *An introduction to parallel programming*. 1st ed. Amsterdam: Elsevier Science & Technology.
- [17] Nichols, B. ,Buttlar, D. and Farrell, J. (1996). *Pthreads Programming*. 1st ed. Cambridge: O'Reilly Media, Incorporated.
- [18] Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. (1992). *Numerical Recipes in C*. 2nd ed. Cambridge, United States: Cambridge University Press.