Abstract

The COSFIRE algorithm [Azzopardi and Petkov 2013] can be used for visual recognition of many different patterns and features in images. In this text we describe the development of a re-implementation of the program in a new programming paradigm. Hereafter, we try to optimize the program and parallelise it. This leads to new multi-threaded piece of software in Python. Additionally, some background theory is provided about the underlying principles in biology and mathematics.
Contents

1 Introduction 3
   1.1 Motivation ......................................................... 3
   1.2 Aims and Evaluation ............................................. 4
   1.3 Structure .......................................................... 4

2 Background 5
   2.1 Biology ............................................................ 6
   2.2 From Neurons to Filters ........................................ 7
   2.3 The COSFIRE algorithm .......................................... 9
   2.4 Applications ..................................................... 12

3 Python Implementation 14
   3.1 Framework ....................................................... 14
   3.2 Design ............................................................ 15
   3.3 Class UML Diagram .............................................. 16
   3.4 Implementation .................................................. 17
   3.5 Analysis ........................................................... 17
   3.6 Discussion ........................................................ 19

4 Program parallelization 20
   4.1 Motivation ....................................................... 20
   4.2 Exploration ...................................................... 20
   4.3 Methods ........................................................... 21
   4.4 Results ............................................................ 22
   4.5 Analysis ........................................................... 29
   4.6 Discussion ........................................................ 29

5 Summary and Final Conclusions 31
1 Introduction

This text describes a project based on an existing pattern recognition algorithm named the Combination Of Shifted Filter REsponses, or COSFIRE, algorithm. This algorithm has many applications in computer vision for recognizing various patterns in visual data. We’ll elaborate on this in Section 2.4. However, the goal of this project is focused rather on the re-implementation of this algorithm, using an example application for verification. Specifically, we’ll reproduce the application as it has been proposed in [Azzopardi et al., 2015], namely the recognition and delineation of vessels in retinal images. Implementing the existing Matlab program in a new programming language will open up the possibility to parallelise and optimize the program such that it runs efficiently, using all available system resources and leveraging the hardware of modern computers, specifically their multi-core capabilities. We will do an assessment of the computation time of the program and the possible achieved speedup, as well as discuss the overall software design choices we made.

1.1 Motivation

Currently, the algorithm has been implemented in the programming language Matlab. However, this code has grown over time to be of significant size. Matlab is inherently procedural in style, and the current implementation consists of many functions with many lines of code in every function accounting for all the different possible methods and variations. The program could improve a lot by being implemented in a newly structured way, in a more object-oriented style. As will be explained in section 3.1 Python is a very popular programming language especially in the field of machine learning. The implementation of the COSFIRE algorithm in Python will also open it up to more interested parties. At the same time, we would like to make the algorithm faster and parallelise it which will be easier to do in a complete programming environment.

---

The proposed application can be used to recognize elongated linear structures specifically. Vessels are a good example of such structures.
1.2 Aims and Evaluation

The goal of the project should be clearly defined within a certain scope. The central purpose of this project consists of the development, optimization and parallelization of the COSFIRE Python Implementation. While there are several variations of implementations of the COSFIRE algorithm, we will focus in this project on a single one, specifically designed to detect vessels or other elongated linear structures. However, this application is emphatically not a part of our goals but simply a means of evaluation as it will be used to assess the quality of the new implementation. Before we do this, we must do some background study to have a good understanding of the underlying principles. Once the code has been written, the attempt to parallelise the program should be aimed toward maximizing speedup and minimizing computation time. Following each of these objectives, we will have a Analysis and a Discussion section to evaluate and discuss the results.

1.3 Structure

This document has been divided into several chapters, each with very distinct purposes. These parts are also ordered, as each one logically follows the previous one. The first part is the theoretical study in section 2. Here, we will look into the background theory of the COSFIRE algorithm. This information, based on the literature, serves to provide a foundation of the principles used by the algorithm. In the next chapter, the design and implementation of the algorithm in Python will be discussed. In section 4 we build further from the created implementation by trying to speed up the program by means of parallelization.
2 Background

There are some theoretical background concepts that are useful and relevant to include here which serve as a foundation for the inner workings of the algorithm we will discuss.

When it comes to computer vision, concepts are often studied and taken from the biological study of the visual system in humans or other mammals. As such, image processing has been described in [Huang, 1996] to have a dual goal, which entails:

1. to come up with computational models of the human visual system
2. to build autonomous systems which could perform some of the tasks which the human visual system can perform (and even surpass it in many cases)

Oftentimes, these two goals go hand in hand because in order to engineer a system that can perform the same tasks as a human visual system one could argue the best way to do this is to try and model the actual human visual system. However, this dual goal also poses a peculiar challenge in computer vision where we might encounter situations where there is no unambiguous measure of the quality of a system because this measure is based on the response of a human assessor. This bias with respect to the real visual system can be seen in applications such as edge detection (as seen in figure 1), but also in our example application of line delineation.

Figure 1: Example of applying an edge detector to a picture of the Eiffel tower. The detector used in this case is a commonly used one created by John F. Canny [Canny, 1986]. There is no absolute measure of the quality of the output image, however a human might denote it as high or low quality. Moreover, differences in preferences like high or low detail in the edges detected only further elaborates this problem.
Tony Lindeberg, a Swedish professor in Computing Science and Computer Vision, has developed several mathematical models for the engineering of an artificial visual system and explains in Lindeberg [1998] why the problem of edge detection is a hard task. The nature and the scale of objects in a picture can vary in many ways and therefore a theoretical edge detector must be able to handle all these different types of edges. It can be argued that the exact definition for "an edge" does not exist, for if there was such a definition the problem would be trivial and could be reduced to searching for locations where the mathematical definition holds. However, this is indeed not the case and our definition of an edge generally comes from human comparison. Alternatively, artificial comparison methods exist Fram and Deutsch [1975], but these do not apply to real-world imagery.

2.1 Biology

To better understand modeling the human visual system, it is important to outline some concepts from neurology and anatomy with regards to this system. We are mainly interested in the neural counterpart of the visual system, i.e. the processing of imagery after it has been caught by the retina and converted to a neural pulse. Upon reaching the lateral geniculate nucleus, the information is relayed to the first visual cortex. In total, a human has six visual cortices, numbered as V1 through V6. For our topic specifically V1, V2 and V4 are interesting because these make up the ventral system as hypothesized in Goodale and Milner [1992]. The ventral system is mainly occupied with the recognition of objects, as opposed to the dorsal system which handles the location of objects and more time-dependent functions like motion recognition and prediction McFarland [2002].

Within the ventral system, the primary visual cortex (V1) is tasked with the first real processing of incoming information, specifically applying edge detection. V2 has a similar purpose, but also takes depth into account and recognizes additional more complex properties like illusory contours von der Heydt et al. [1984] and stereoscopic edges von der Heydt et al. [2000]. Finally, the information travels through V4 to the temporal lobe.

Figure 2: Strongly simplified overview of the dorsal stream (top) and the ventral stream (bottom) and their functions. Perry et al. [2014]
2.2 From Neurons to Filters

It has been observed in the visual system of mammals that different neurons in the visual cortex are specifically sensitive to specific base features like edges and bars (or alternatively gratings [Albrecht et al., 1980] and in specific orientations and scales. Lindeberg explains in [Lindeberg, 2013] how mammalian vision has developed receptive fields that are tuned to different sizes and orientations in the image domain. Higher level cells would be capable of combining the responses from these lower level cells to respond to more complex visual structures. These lower level cells, or simple cells as they were called when they were discovered by [Hubel and Wiesel, 1959], have been compared mathematically to linear Gabor filters, specifically with regard to the orientation- and scale-specific property of this filter and its capability to respond to edges [Jones and Palmer, 1987]. A 2D representation of such a Gabor-filter can be seen in Figure 3. Gabor filters are a combination of a (Gaussian) normal distribution and a sinusoid, with several parameters denoting many unique configurations. For example, \( \theta \) denotes the orientation of the wave, which dictates the orientation of edges the filter will detect and the phase \( \psi \) can shift the inhibitory and facilitatory areas in the filter. Using the results from these simple cells, more complicated structures can be recognized by combining many neuron’s responses and comparing them with earlier responses. In other words, to recognize a specific structure or object, the brain must already have a record of recognizing the same object. If the brain has learned to do so earlier, it can base its classification on the firing of specific neurons that correspond to specific visual elements [DiCarlo et al., 2012]. Similarly, we can combine multiple linear filter responses of an input image to recognize a specific combination of edges, i.e. a target pattern.

Figure 3: Gabor filter-type receptive field typical for a simple cell. Blue regions indicate inhibition, red facilitation [From Wikimedia Commons]

Figure 4: Different configurations of gabor filters recognize different orientations of lines or edges.
However, there have also been different models for simple cells that are not based on Gabor filters. Gabor filters, while a useful and mathematically elegant model for simple cells, are not a complete and entirely correct model for these cells, according to Stork and Wilson [1990]. Hubel and Wiesel [1962] theorized a model based on cells in the lateral geniculate nucleus in the thalamus, the body that precedes the first visual cortex in the visual pathway (as has been mentioned in section 2.1). Here, orientation selectivity is attained by a combination of center-on and center-off cells in a straight line.

Figure 5: Four linearly positioned center-on cells combine their response to detect a specific orientation.

When taking these cells from the LGN (lateral geniculate nucleus) into account, we can create a filter that more closely approximates the inner workings of the actual visual system. As explained in Azzopardi and Petkov [2012], this model is named a CORF model for Combination of Receptive Fields. Instead of using Gabor filters to simulate simple cells, Difference of Gaussian filters are used to simulate these LGN receptive fields. At a later stage, the responses from these DoG (Difference of Gaussian) filters are combined to recognize structures like lines. This model lays the foundation for the line delineation detection algorithm central in this document. In the next section we will further elaborate on the complete algorithm which is central to this text.

Figure 6: A model of LGN receptive fields as sketched in Azzopardi and Petkov [2012].
2.3 The COSFIRE algorithm

The subject of this project is the COSFIRE pattern recognition algorithm as published in [Azzopardi and Petkov, 2013]. As explained in this paper, the COSFIRE filter, being short for Combination Of Shifted Filter REsponses is computed as a function of the shifted responses of simpler filters. For example, this can be a linear filter like Gabor (as proposed in the beforementioned paper) or DoG, Difference of Gaussian, as proposed in a later paper [Azzopardi et al., 2015].

![Figure 7: Illustration of a DoG and a Gabor response in 1 dimension.](image)

The algorithm learns to recognize a specific pattern by feeding it a prototype image. This picture should give the "perfect" example of the pattern that should be recognized. In the example of detecting lines, this prototype should consist of a single white line on a black background. After applying a selected filter (Gabor, DoG), the program can find the prototypical locations of the input prototype by searching for the peaks of response strength on several circles around the target point, i.e. the point of interest. The following illustration shows this process, where the blue dot denotes the point of interest (in this case, we want to recognize lines from their center), the green circles represent the circles that were evaluated and the red crosses represent the found prototypical locations.

![Figure 8: Illustration of the process of fitting a prototype in the COSFIRE algorithm.](image)

Once this process has been completed the filter has fitted a prototype. When properly configured, it is now able to recognize the pattern. The result is several tuples of values describing the found points. Each tuple contains the polar coordinates of the found locations and the filter parameters (depending on the used filter). For a similar prototype as used in Figure 8, this set $S$ of tuples could look like the following.
With the use of these tuples we can transform an input image, i.e. find the required pattern by following the same steps for every tuple:

- Apply the specified filter on the input image with the parameters given in the tuple ($\sigma$ in the example case).
- Set all values in the filtered response below 0 to 0, in other words half wave rectification.
- Blur the image with a blurring factor $\sigma_{\text{blur}} = \sigma_0 + \alpha \rho$ where $\rho$ is a member of the tuple and $\sigma_0$ and $\alpha$ are parameters given to the algorithm.
- Shift the blurred filter response using the polar coordinates $(\rho, \phi)$ in the reversed direction.

Figure 9: A subject image and its output with the corresponding intermediate shifted responses. The grid has been superimposed in order to be able to see the performed shifting operation. Sub-figure 1 shows the subject image as given to the program, while 6 shows the corresponding output. The output image shows that the program has successfully detected all vertical lines. Figures 2-5 represent the intermediate results, each one corresponding to one of the four tuples that had been learned earlier.
Because the filter responses are shifted in reversed direction, the tuple locations are, in every response, shifted to the target location of interest. Figure 9 shows this, with the filtered, blurred and shifted responses of the 4 tuples in a variation of the line detection prototype. Notice this program only recognizes vertical lines.

The resulting combined response is calculated by taking the geometric mean of the shifted responses. The geometric mean is a multiplication operation, and can therefore be compared to a logical AND in the case of boolean operations. This is important as we are looking for locations in the input picture where all of the responses are strong. The usage of multiplication over addition has been researched before in [Gheorghiu and Kingdom, 2009] where psychophysical evidence was provided for neural multiplication in human visual processing of shape.

This resulting mean response is the output of the COSFIRE filter. However, as can be seen in figure 9, this approach only takes a single orientation into account, as well as a single scale. By repeating all the steps after slightly altering the set $S$ of tuples, we can gain results for different orientations and scales.

To achieve rotation invariance we must change the set of tuples by adding to the angle $\phi$. From Azzopardi et al. [2015]: One can manipulate the parameter in the set $S$, which corresponds to the orientation preference $0^\circ$, to obtain a new set $R_\psi(S)$ with orientation preference $\psi$:

$$R_\psi(S) = \{(\sigma_i, \rho_i, \phi_i + \psi) \mid \forall (\sigma_i, \rho_i, \phi_i) \in S\}$$

Depending on the goal, different values of $\psi$ would have to be chosen. To achieve optimal full rotation invariance we must use a set of equidistant orientations:

$$\Psi = \{2\pi \frac{i}{n_r} \mid 0 \leq i < n_r\}$$

Alternatively, $\pi$ can be used instead of $2\pi$ if the filter is symmetrical.

To achieve scale invariance a similar approach can be used, scaling up the values of $\rho$ (by multiplication instead of addition). However, depending on the filter type, the parameters of the used filter must be scaled along with $\rho$. For example for a Gaussian-based filter, the value for $\sigma$ must be scaled up with the same factor.

When using rotation and/or scale invariance, we produce many combined shifted filter responses depending on the degree of invariance. These responses are combined in a maximum manner. For example in the case of rotation invariance:

$$\hat{r}_s(x, y) = \max_{\psi \in \Psi} \{r_{R_\psi(S)}(x, y)\}$$

Here, $\hat{r}_s(x, y)$ refers to the final output in the filter.

When applying the filter, it is normally necessary and/or preferable to apply some filters before and after applying the COSFIRE algorithm. This may include increasing the contrast in the picture or taking out elements not part of the area of interest. These steps are called preprocessing and postprocessing and are not considered to be part of the COSFIRE algorithm.

---

2The degree of invariance refers to the amount of different tuple sets that are applied. For example in the case of rotation invariance: $|\Psi|$. 

11
2.4 Applications

As mentioned before, the COSFIRE algorithm can be broadly used to recognize different types of patterns. Some applications have been tested with successful results (not an exhaustive list):

- In [Azzopardi and Petkov, 2013], the detection and recognition of retinal vascular bifurcations, handwritten digits and traffic signs in complex scenes.
- Letter and keyword spotting in handwritten manuscripts and object spotting in complex domestic scenes in [Azzopardi and Petkov, 2014].
- A color-sensitive approach in [Gecer et al., 2017] of the recognition of traffic signs and butterflies.

Some more example use-cases are shown in the following two sections.

2.4.1 Edge detection

With an early version of the algorithm, it was shown in [Azzopardi and Petkov, 2012] that the algorithm can be used to perform contour or edge detection. Of course this makes sense looking back at the background theory, specifically section 2.2. Again, the model is based on the concept of simple cells in the primary visual cortex of mammals. The modeled receptive fields have a property comparing center and surround of a small area, thus registering local differences like edges. The proposed algorithm in this paper is therefore named Combination Of Receptive Fields (CORF). It is later improved upon in [Azzopardi et al., 2014] by the addition of push-pull inhibition.

Figure 10: Edge detection as performed by a COSFIRE filter.
2.4.2 Line detection

As shown in [Azzopardi et al., 2015], a line-shaped prototype combined with a prototype of a line-ending can be used to detect lines such as the vessels in retinal images. The result of this is shown in Figure 11. Alternatively, the algorithm can also be used to detect interesting features in such images like vascular bifurcations and crossovers.

![Figure 11: Vessel delineation as performed by a COSFIRE filter.](image)
3 Python Implementation

As one the main goals of this project, the next step is the implementation of the COSFIRE program in the programming language Python\(^3\). We will first assess the different parts of what we are going to build. Then, we will discuss how we are going to approach doing so in section \[3.1\] Once we have a clear picture of the what and how, we will present a complete design of the software components (section \[3.2\]). Once we have written our implementation (section \[3.4\]), we will analyse and discuss the program and its results (sections \[3.5\] and \[3.6\]). For example, we will subject the program to functional tests and timing benchmarks. Moreover, we will discuss the things that can be improved upon in the code as well as the encountered obstacles.

The code that has been written for this project is hosted on GitHub\(^4\). It is not a completed software product but rather a continuous effort.

Most computer programs, as a simple definition, take an input and produce an output. This property holds in general terms for the COSFIRE program. As such, the objective of re-implementing the existing Matlab code will be to produce the same output for equal inputs. As a reference to the Matlab implementation we used \[https://gitlab.com/nicstrisc/B-COSFIRE-MATLAB\]. The main functionality of this library consists of a) configuring a new COSFIRE filter with the appropriate variables b) learning a desired pattern from a prototype, a process called fitting, and c) transforming a subject with the filter, i.e. applying it to a new picture.

To be able to reproduce this functionality, the library is required have at least a certain set of operations at its disposal. A prominent example of such an operation is the computation of a linear filter i.e. convolution for such filters as Gaussian blur, DoG and Gabor. Some more methods that are definitely required are searching for local maxima (for the fitting step), shifting an image (for shifting filter responses in the transform step) and half-wave rectification. The appropriate programming frameworks, libraries and patterns will have to be chosen in order to make all these things possible.

3.1 Framework

As has been mentioned before, the new program will be written in Python. This has been a requirement given to us and makes sense as a choice for several reasons. Python has proved to be a popular programming language, suitable for scientific applications \[Perez et al., 2011\]. Additionally, it is currently relevant as a tool in the field of machine learning. Evidence for this is the abundance of active libraries in this field as well as contemporary research being done with Python in this field (e.g. Chen et al., 2018).

Perez et al. (2011) explain in their paper that in the Python world, scientists have collaboratively developed an open ecosystem of foundational tools that provide the key functionality needed for everyday computing work. Some of such tools that are mentioned include NumPy and SciPy\(^5\) for handling numerical arrays like matrices and manipulating them in various ways, and matplotlib\(^6\) for creating high quality plots. The latter has also been used in this document to generate most if not all plots.

\(^3\)Python Software Foundation. Python Language, version 3.6.4 as used by us. Available at \[python.org\]
\(^4\)At the time of writing, the repository can be found at \[https://github.com/Theys96/COSFIRE\]
\(^5\)\[https://scipy.org/\]
\(^6\)\[https://matplotlib.org/\]
Python on its own provides a framework that is strongly driven by the usage of packages. Most built-in functionality of python (i.e. "standard functions") is divided into built-in packages and users are encouraged to create new packages to provide to the community. We must make an assessment as to which packages will be required by our project. Naturally, we do not want to reinvent the wheel and will therefore attempt to leave as much as possible to external dependencies. Additionally, we expect these dedicated libraries to always outperform our own implementation.

It has also been given to us as a requirement that the resulting module will be compatible with scikit-learn\(^7\) as a plug-in. This is to allow for the COSFIRE algorithm to have a standardized interface, as well as to open the possibility to use it in various machine learning routines offered by scikit-learn like grid-searching capabilities. In the future, this feature may be used to develop supervised learning programs with the COSFIRE algorithm.

Regarding the libraries which will be used in the project, it’s important to assess precisely which of these are a certain requirement to successfully run the COSFIRE program. Firstly, SciPy and with it NumPy are such absolute requirements to work with matrices. The pictures we want to manipulate are to be loaded as matrices and another library is required to load these pictures from file. The choice can be made between OpenCV\(^8\) and PIL\(^9\). Both are possible to be used by users and we must support both. For our purposes however, we will choose cv2 (OpenCV) where it is needed.

3.2 Design

Now that we have established a framework of Python libraries we can start to design the structure of the program. Using an object-oriented approach we can best compartmentalize the different components [Schulz, 1998]. The main component, of course, will be the COSFIRE operator. It has been requested that the main class can be configured with all parameters, including the so-called strategy. This strategy describes the way the COSFIRE algorithm is applied. The logic of the program must therefore be located in a separate class for the specific strategy. One such strategy is the circle strategy, which we have discussed in section 2.3. In scikit-learn terms, this main operator will be a class of the transformer type. This means it has a fit() method which learns from a prototype and a transform() method which transform an input image based on the fitted prototype.

There are some more required classes that we’ll have to add. The different linear filters (Gaussian, DoG, Gabor) will have their own class with a common interface and will also be a transformer. Another class will be created to serve as an abstract stack of images that can be accessed and manipulated simultaneously.

We will still also need some separate functions for various routines which we’ll include in their own file. However, this is a point of improvement as it contradicts pure OOP guidelines.

We can construct a UML class diagram of the software package, which schematically describes the layout of the software and clarifies its inter-connectivity [Larman, 2002]. This diagram can be found on the next page.

---

\(^7\) scikit-learn.org
\(^8\) opencv.org
\(^9\) pythonware.com/products/pil/
3.3 Class UML Diagram
3.4 Implementation

A history of the implementation process can be found on the GitHub repository, where different branches were used to keep track of the different versions. Additionally, Travis-CI\textsuperscript{10} provided continuous testing of the written code. As a first step, we made the classes for the different filters. To leverage the power of object-oriented programming we saved the kernel matrices of these filters in their own object. In other words, a filter only has to be constructed from parameters once for it to be used to transform an input picture many times. As mentioned before, the included filters are Gaussian, DoG and Gabor. We compared the filters’ kernels and output to their Matlab counterparts and confirmed that they were the same. After this was done, we started with the development of the fitting step in the COSFIRE operator. An early obstacle here was to create a good peak-searching algorithm. Such an algorithm, although mathematically well-defined seemed to exhibit some ambiguity in terms of existing libraries regardless. The algorithm should return the same locations as the Matlab implementation and for these reasons, we wrote a custom function for this purpose. At this point, we required the abstract image stack. This class was later rewritten to allow for easy application of filters. The ImageStack object serves to be able to control multiple images at once and search through them. Once we had completed the COSFIRE operator with its \texttt{transform} method, we started matching the results from our implementation and the Matlab code. There turned out to be some discrepancies which required for us to precisely compare all intermediate results. This way, we could find some places where our code executed slightly differently as in the Matlab version. Sometimes, the problem was to be found a bit deeper, when the behavior of built-in functions differed across the two languages. One such case applied to the computation of Gaussian blurs in a situation where the kernel size could be of even size. In that case, the output of the filter is ambiguous if the result is cropped back to the size of the input picture. This happens because there is no unambiguous center pixel in the kernel, so one has to be chosen. The problem was fixed by not allowing even-sized kernels to be used all together. Additionally, we added timing to our code which saves the time (in wallclock\textsuperscript{11} seconds) for every step performed.

3.5 Analysis

To assess the quality of the created software we can analyse the produced outputs of the program and test its speed in terms of computation time. We have compared the output of an example case from a previous experiment with the same setup in our new Python implementation. These pictures can be seen in figure\textsuperscript{12}

Numeric comparison does show some difference between these images, however this can be mostly attributed to floating-point value errors and different value stretching methods. When comparing the binary output pictures, which are generated by converting the values to 0 or 1 based on them being above or below a certain threshold, the pictures from the two implementations match completely.

\textsuperscript{10}travis-ci.org
\textsuperscript{11}i.e. based on the internal clock of the machine, not directly on CPU cycles.
\textsuperscript{12}In some cases, the gray-scale values in a picture are “stretched” in such a way that the complete channel of values 0-255 is used.
Figure 12: Some output images next to each other as produced by the Python and Matlab implementation.

To investigate the program’s speed, we can look at the timings that are done by the program itself. For a typical usage of the program, this looks like the following.

Clearly, fitting the prototype takes nearly no time. More significantly, the most time-intensive in this case is the shifting and combining of the different orientation. Because
there are many (24) of these, this takes quite some time. The recorded computation times are similar to those of the Matlab implementation. When we use parameters that we expect will decrease the computation time (fewer values for $\rho$ and $\psi$), the results are as follows.

--- TIME MEASUREMENTS ---

1. Finding tuples for rho=0: 0.01 ms
2. Finding tuples for rho=2: 5.87 ms
3. Finding tuples for rho=4: 6.15 ms
4. Finding tuples for rho=6: 5.46 ms
5. Finding tuples for rho=8: 5.99 ms
6. Finding all 9 tuples for 5 different values of rho: 23.49 ms
7. Applying 1 filter(s): 60.21 ms
8. Precomputing 5 filtered+blurred responses: 75.62 ms
9. Shifting and combining the responses for psi=0.00: 51.21 ms
10. Shifting and combining the responses for psi=0.26: 45.74 ms
11. Shifting and combining the responses for psi=0.52: 40.39 ms
12. Shifting and combining the responses for psi=0.79: 40.93 ms
13. Shifting and combining the responses for psi=1.05: 40.93 ms
14. Shifting and combining the responses for psi=1.31: 40.39 ms
15. Shifting and combining the responses for psi=1.57: 40.39 ms
16. Shifting and combining the responses for psi=1.83: 40.39 ms
17. Shifting and combining the responses for psi=2.09: 40.39 ms
18. Shifting and combining the responses for psi=2.36: 40.39 ms
19. Shifting and combining the responses for psi=2.62: 40.39 ms
20. Shifting and combining the responses for psi=2.88: 40.39 ms
21. Shifting and combining all responses: 584.74 ms

These results confirm that indeed the computation time relies heavily on the amount of values from $\psi$, because of the amount of computed orientations. Moreover, the amount of values for $\rho$ influence the amount of found tuples. The amount of tuples also seems to have a direct influence on the computation time of every orientation, as we can see by comparing the two lists of measurements. In the previous two examples, 23 tuples and 9 tuples were used respectively.

### 3.6 Discussion

In this section we will reflect on the construction of the COSFIRE Python Library and make some remarks about possible improvement in the future. The most challenging part in creating this software as experienced by us was getting the program to produce precisely the same output as the Matlab implementation, whereas the easy part was implementing the algorithm itself. The program can definitely be improved upon by some refactoring and adding even more tests, including unit tests. Furthermore, it’s probably interesting to test out several different applications of the library. This way, we will find things to improve upon or add. The goal is to release the package into the Python/Scikit-learn community.

In the next chapter, we will continue with the parallelization of the program. In line with this, we think it’s interesting to explore different ways to make the Python implementation faster. This may be possible through refactoring as well as hardware optimization.
4 Program parallelization

With the functional Python library created, the next step is to optimize the program to run as fast as possible, leveraging the power of the multi-core hardware of today’s computers. The focus in this text lies with the parallelization of the algorithm itself. There are also possibilities regarding the parallelization and hardware-optimization for speeding up the computation time of various used techniques like convolution, shifting and the geometric mean, but these are considered out-of-scope in this document.

The research questions we hope to answer at the least are:

- What is the best parallelization implementation we can create?
- Is the program suitable at all for parallelization and if so, to what degree and if not, why not?
- What are ways to work towards a better implementation in the future?

4.1 Motivation

In the construction of the Python implementation of this algorithm, we noted there is a high degree of step-wise processing of the input images. Some of these steps are repeated many times depending on the parameters used, and this creates opportunity to parallelise the algorithm. We hope to achieve considerable speedup by doing so, as many parts that take a lot of computations are indeed repeated steps.

4.2 Exploration

We will take a look at the program as it has been structured by us to find the most prominent loops in the code. These we will mark as being interesting to parallelise.

We will only go into the parallelization of the transformation step, as in our previous results we found this to be the most computationally expensive.

The transformation step can be described in pseudo-code as follows.

1. Compute all the filter (DoG) convolutions.
2. Apply all possible blurs to the filtered images.
3. For every rotation-invariant:
   4.   For every scale-invariant:
      5.       Compute the combined shifted responses of the rotated/scaled tuples.
      6.       Add the result to the main result by means of a maximum.

For computing the combined shifted responses of the tuples, we can write another pseudo-code routine as:

1. Adjust the base tuples by the given rotation-invariant and scale-invariant.
2. For every tuple:
   3.       Retrieve the associated blurred filter response.
   4.       Shift the response based on rho and phi.
   5.       Record the resulting image.
   6. Return the geometric mean of the images for every tuple.
For our purpose, the interesting notion is the fact that we have three loops within each other, with operations on all levels. First we have the two loops going over all rotation/scale invariants ($\nu \in \Upsilon$, $\psi \in \Psi$), then in the shift/combining routine we loop over every tuple ($s \in S$). The two outer loops we will regard as one. These loops can be schematically described as:

\[
\text{For } (\psi, \nu) \text{ in } \Psi \times \Upsilon:
\]

\[
\text{For } s \text{ in } S:
\]

\[
\text{operations}
\]

We will try several variations parallelizing either or both of these loops. As an alternative to adapting the existing code exactly as it is, we can restructure the program to be two big loops for first shifting all responses and then combining them respectively. Making this division may improve the parallizability of the program and therefore result in more speedup.

### 4.3 Methods

When parallelizing a program there are several things to take into consideration. To achieve the goal of utilizing all of a machine’s available cores, routines have to be divided into separated pieces of work that can be executed by multiple processes or threads. We must thoroughly determine which variations of parallelization we must use.

The distinction between processes and threads is indeed relevant here. Threads, unlike processes, share their heap of memory and can therefore work on the same global set. This is very useful for this particular application since the input image and its filtered responses are shared resources. Transferring this data back and forth between processes would take a comparatively large amount of time since the performed operations take relatively short.

#### 4.3.1 Software framework

Python by default offers multiprocessing pools as an abstraction on a set of processes or threads which can be applied to a data set with a given processing method to process the data set concurrently using the predefined amount of processes or threads in the pool. We will use the pool of threads, or threadpool.

There are some things to take into consideration when using threadpools. It is common for many threads to be alive on a system at the same time in order to do many things seemingly at the same time. However, when we want to achieve a successful parallelization, using more threads than a computer has processing units will likely not be very effective.

There is a problem of deadlocks that can arise from using the same threadpool for different loops, specifically loops within loops. For example, when a certain outer loop is being executed by a threadpool, it will immediately give all its threads one of the iterations of this outer loop. So, if there exists an inner loop in the iteration of the outer loop, the same threadpool can never be used to also process this inner loop. Instead, it is possible to use another single threadpool for all of the times the inner loop is executed. This might in fact be desirable as it is often counter-effective to start too many threads. Using the a global threadpool for an inner loop will keep the amount of threads at a given level and threads can be reused instead of being started and terminated for a single task. As an alternative to this, restructuring the program to get rid of nested loops altogether is another possibility, like mentioned in section 4.2.
4.4 Results

Note: In all tests of which the results are presented in this section, the COSFIRE program is executed in the same typical situation several times per set of test parameters.

We decide to begin with the load-balancing of either the outer or inner loop, as we explained in section 4.2. We hypothesize that parallelizing the outer loop will be much more effective as the creating and termination of threads is also costly in time, but relatively less so when the individual tasks are larger. The test we run is on a machine with 4 processing units, and so we don’t expect to see any speedup beyond using that many threads. The following graphs show the computation time of the step of interest, the shifting and combining of all responses for all rotation/scale invariants.

- **variation 1**: Parallelizing the computation of every orientation (the outer loop) with a single global threadpool of a varying amount of threads.
- **variation 2**: Parallelizing the computation of every tuple (the inner loop) with a single global threadpool of a varying amount of threads.
As expected, variation 1 was most effective in achieving speedup. However, variation 2 did also have an effect. In this case we see strongly the result of using too many threads: the computation slows down again beyond 4 threads. Variation 1 seems to stabilize and even improve slightly beyond 4 threads. This can be attributed to, for example, time where one thread must wait for data to be loaded not being wasted as another thread can do computations at the same time. However, this effect is very small. We are interested in combining both variations to see if we can improve by applying parallelization on both loops. As mentioned before, another threadpool must be used for the inner loop in this case to avoid a deadlock. As a first test, we decide to create a threadpool in every iteration of the outer loop to be used for the inner loop. The size of this temporary threadpool will be the same as the outer loop’s. This will be referred to as variation 3 and the test setup is the same as in the previous cases.

Not surprisingly, the effect observed in variation 2 where the computation time increases after 4 threads can also be observed in this variation. We would like to get rid of this effect and decide to make the temporary threadpool for the inner loop of fixed size. It will always be the size of the amount of the machines’ processing cores. This will be referred to as variation 4 and the test setup is the same as in the previous cases.
We can see that the slowdown has been decreased dramatically. It is however still not completely gone and our hypothesis is that the remainder of the effect can be attributed to the fact that we create and terminate a threadpool for every iteration of the outer loop to do the parallelization of the inner loop. We therefore decide to make this threadpool for the inner loop a global one used by every iteration. It will still be the same size as the amount of the machines’ processing cores. This will be referred to as variation 5 and the test setup is the same as in the previous cases.
The results we can observe now are approximately as good as in variation 1, or better. Some more testing should reveal which variation is in fact superior. We prefer the variation with the highest amount of parallelization, variation 5, as we believe it to have the largest potential.

To give a more elaborate picture we have also executed the same tests on a machine with 24 processing cores. Since this machine has many more cores however, we decided to measure up to 64 threads instead of 16. The results are shown below.

The results from the 24-core machine seem to imply that using more than 8 or 16 threads is never effective. This would be undesirable as we want to leverage the added value of 24 cores over 4.
We would first like to get a good picture of what happens when the amount of work is higher. We will test this by running the transformation step with varying amounts of tuples to work with, as well as varying amounts of threads. The results on both machines (4 and 24 core) are shown below.

**Number of threads vs. Computation time of all shift/combine steps**

*variation 5   4 cores*

*Variation 5   24 cores*
As an alternative to the given variation, we have also tried to restructure the program to be two big loops for shifting all responses and then combining them respectively. When comparing the computation time, we get the following results on our 4 and 24-core machines.

It is apparent that this program has a better runtime, regardless of the parallelization of it. As a trade-off however, we must note that this new variation uses a very high amount of memory, storing many images at any point during computation. We can see this when comparing the relative speedup for different runs of the program for increasing amounts of tuples. Notice that the drop in speedup for 63 and 127 tuples is caused by running out of memory.

The picture on the right shows the results after doing the experiment in steps, freeing memory again in the middle. In the left picture, no such measure was taken.

In this experiment, the COSFIRE program was run several times, but it also has a considerable amount of RAM at its disposal. Using this amount of this much memory is not acceptable.
In the end, we produced three options for parallelization implementations.

1. The final variation of parallelizing the two loops (variation 5).

2. The alternative strategy restructuring the program to be two big loops for shifting all responses and then combining them (variation 6 / strategy B).

3. A composite strategy based on variation 5 but enhanced with techniques from strategy B (strategy A).

A comparison in computation times shows that strategy A can be considered optimal at this point, as it also uses a normal amount of RAM.

4.4.1 Remarks

There are some results that have been omitted up until now, as they were not relevant as serious improvements. For example, there is a distinction to be made when using threadpools of pools of processes. Processes, as opposed to threads, have their own memory and must therefore communicate data in order to parallelise a task. For the different implementation variations we have also tested a version with processes instead of threads. However, these attempts resulted in an increase in runtime of at least a factor of 5 and have been discarded for this reason.
4.5 Analysis

In the previous section, we presented the results from several implementations of parallelization of the designed program. Several conclusions can be drawn from these results. We are particularly interested in answering our research questions from section 4.

4.5.1 What is the best parallelization implementation we can create?

To answer this question, we look back at the results presented in section 4.4. We can up with different variations and strategies with different implementations of the COSFIRE algorithm. Some used different ways to handle the data in memory and used too much RAM. Implementations that take such high amounts of memory have been taken out of consideration when determining the optimal algorithm. The best algorithm that still qualifies is referred to in section 4.4 as Strategy A and only parallelises the outer loop concerning different orientations. As such, it uses a threadpool to divide the different variations of invariance and for each of these a single thread goes through the complete COSFIRE pipeline. This resulted in relatively large chunks of work for each thread which is better for optimization.

In a sequential situation, this implementation also outperforms all others. On the other hand, it has a lower relative speedup. In other words, this implementation can rather be described as an optimized implementation of the algorithm itself instead of the optimal implementation of parallelizing the algorithm. However, we do not regard this as a problem since it definitely has the best computation time comparatively.

As mentioned in section 4.4.1, the usage of threads over processes for parallelization is essential for the usage of fast shared memory and with this the parallelization implementation. However, this has its disadvantages. The extensive use of the shared memory has made it so that the actual speedup that could be achieved was at most just over 3 for a machine with 24 cores. This is not as high as one would expect for this amount of cores. Instead, speedup seems to stagnate between 8 and 16 threads.

4.5.2 Is the program suitable at all for parallelization?

As mentioned before, the shared memory model used in the parallelization has its disadvantages when it comes to the potential speedup. The algorithm itself has good speedup potential because there are many independent loops. However, the handling of large chunks of data (i.e. images) makes it harder to take full advantage of this.

4.6 Discussion

We have created a parallelised version of the COSFIRE Python Implementation. It is able to speed up the program by some up to about 3x, depending on the system hardware. This is not an optimal result, but a positive one nonetheless. In the future, possibly better clever parallelization implementations may be found to further increase the speedup.

One such alternative may be found in constructing a data-based parallelization. The way we approached the problem in this project should be classified as task parallelism, the parallelization of the executed steps. Alternatively, the COSFIRE algorithm is modestly suitable for data parallelism because applying the full program to parts of the input image separately and sticking the results together again should yield the correct complete result. We will discuss this idea in a bit more detail in the following appendix, as we believe it to be interesting for future work.
Appendix: Discussion on Data Parallelism

As mentioned in section 4.6, an interesting alternative to the parallelization of the algorithm is by leveraging data parallelism over task parallelism. It is theoretically possible to apply the COSFIRE algorithm to multiple chunks or sub-images of the input image concurrently, and afterwards assembling the results again to create a final result. This idea has some challenges however.

Firstly, because of the shift operation every chunk needs some knowledge of a certain amount of pixels around itself in order to produce correct results. In any direction, this amount of pixels can be determined by finding the maximum value of $\rho_i$ in the set of tuples. This dictates how far off the edges pixels can influence the results in the frame of interest. The solution to this problem is giving the chunks the required amount of overlap, afterwards extracting the relevant inner frames again.

Secondly, sometimes for operations global properties of the input image or an intermediary result may be needed for some operation. In some cases this problem can be averted, but otherwise there will still be the need for some communication between the computation of chunks or an adjusted approach.

The major advantage of data parallelism in this case could be that the use of processes over threads will eliminate the bottleneck that shared memory created. Instead, the entire operation is divided into smaller jobs for several processes with their own memory. As was mentioned in section 4.4.1 processes were counter-effective in the proposed task-parallelization method and created a slowdown of around 5 (i.e. the program ran 5 times slower). However, with a simplified version of an implementation in this paradigm we achieved a speedup with 2 processes on a 4 core machine from $1640\text{ms}\pm12\text{ms}$ to $1280\text{ms}\pm34\text{ms}$. In other words, approximately a speedup of 28%. This made us wonder whether this approach is also effective for a higher number of cores in a way that task parallelism wasn’t (recall it stopped improving around 8 cores). We extend the simplified approach to allow for an arbitrary amount of split parts. The results, compared to our final implementation are as follows.

These results seem to suggest that implementing data parallelism does indeed have much more potential than our task parallelism. We expect the results to be even better with larger images, as this relatively shrinks the overhead caused by inter-process communication and the overlap in the data chunks. It has to be noted that this would still require some new research on dealing with the before-mentioned challenges before a good implementation can be created.

*i.e. $1640\text{ms}$ on average with a sample standard deviation of $12\text{ms}$ over several consecutive runs.*
5 Summary and Final Conclusions

In this final section I will briefly go over the results achieved in this project as presented in this document.

In the first part, we recited the background theory that serves as a foundation for the COSFIRE algorithm. These biological and neurological principles can be used to understand the inner workings of a visual system and should also help in the future to create better models.

As the main goal of this project, we had the dual task of re-implementing and parallelizing the algorithm at hand. In section 3, we discussed the design process of creating a new Python library package for the COSFIRE algorithm. The results achieved by this package matched those of the original Matlab code in quality and speed.

Ultimately, the parallelization of the program using a task parallelism adaptation was attempted. This had modestly positive results, and a speedup could indeed be achieved by implementing this shared-memory model parallelizing the pipeline of the COSFIRE program. We tried different variations in this, parallelizing different loops, with varying results. In the end, we chose a simple solution, only parallelizing the largest (outer) loop. We could conclude that the program is indeed suitable for this form of parallelization, but hypothesized that the main bottleneck is the shared-memory model that is used which, could possibly be omitted by attempting a data parallelism adaptation. We did an experiment to confirm this, which showed that indeed it is probable that data parallelism is the better pick for this problem. However, creating a good data parallelised implementation of the COSFIRE algorithm is a more complicated task which can be interesting for the future.
References


