

**Technische Universität Berlin**

Fakultät IV - Elektrotechnik und Informatik

Dept. Computational Psychology



Bachelorarbeit

**Reproducibility of Computational Studies:  
Replicating and Evaluating a Camouflage  
Detection Algorithm**

Marc Tukendorf

Matrikelnummer: 390845

14.06.2023

1. Gutachter/in: Prof. Dr. Marianne Maertens
2. Gutachter/in: Prof. Dr. Guillermo Gallego Bonet



## Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

*Berlin, den 14.06.2023*

A handwritten signature in black ink, appearing to read 'Tukendorf', written over a horizontal dotted line.

Marc Tukendorf



## **Abstract**

This thesis addresses the reproducibility of scientific work and shows the causes that lead to low reproducibility in academic publications. It is also important to differentiate between reproduction and replication of algorithms, which are also defined in this work. In addition, criteria are presented that must be fulfilled to make a research reproducible. To visualize the reproducibility of scientific work, a dissertation is used as an example. This dissertation implements a camouflage detection algorithm. This algorithm returns for a given image a numerical value, also called edge power, that indicates the detectability of a circular object. For this dissertation, the criteria of a reproducible publication will be applied and also a possible approach for replicating the camouflage detection algorithm will be presented. Furthermore, the goal is to create a successful replication of the algorithm. To ensure the validity and reliability of the replication, a diagnostic tool combined with tests for each segment of the algorithm will be implemented in this approach. After successfully replicating the camouflage detection algorithm, a final test run with a large variability of input data is carried out and used to evaluate the replication. This evaluation will show that the edge power values will be the same as expected from the original implementation.

## **Zusammenfassung**

Diese Arbeit thematisiert die Reproduzierbarkeit von wissenschaftlichen Arbeiten und zeigt die Ursachen auf, die zu einer geringen Reproduzierbarkeit von akademischen Publikationen führen. Wichtig bei diesem Thema ist auch die Differenzierung zwischen Reproduktion und Replikation von Algorithmen, die in dieser Arbeit ebenfalls definiert werden. Zudem werden auch Kriterien vorgestellt, die erfüllt werden müssen, sodass eine Publikation reproduzierbar ist. Um die Reproduzierbarkeit von wissenschaftlichen Arbeiten zu visualisieren, wird hierfür eine Dissertation als Beispiel verwendet. Die Dissertation beschreibt den Camouflage Detection Algorithmus. Der Algorithmus erzeugt für ein gegebenes Bild, das ein rundes Objekt beinhaltet, einen numerischen Wert, auch Edge Power genannt. Dieser Wert gibt an, wie gut das Objekt in einem Bild sichtbar ist. Hierbei werden die Kriterien einer reproduzierbaren Publikation auf diese Arbeit angewendet und ein möglicher Ansatz für die Replikation des Camouflage Detection Algorithmus dargestellt. Zusätzlich wird eine erfolgreiche Replikation des Algorithmus angestrebt. Um die Validität und Reliabilität der Replikation sicherzustellen, wird in diesem Ansatz ein Diagnosetool kombiniert mit Tests für jedes Segment des Algorithmus implementiert. Zudem wird nach Fertigstellung der Replikation ein finaler Testlauf mit einer großen Variabilität an Input-Daten durchgeführt und anhand dessen die Replikation evaluiert. Die Evaluierung wird zeigen, dass die Edge-Power-Werte wie erwartet den Werten aus der originalen Implementierung entsprechen.

## **Acknowledgements**

I wish to thank my supervisor, Marianne Maertens, for their great support when writing my thesis.

I also want to thank my family and friends for supporting and accompanying me along the way to my first academic milestone. Specially I want to thank Dana Tukendorf, Martina Pausch, Cedric Pausch and also to Viktoria Bill and Johannes Tabelaing.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement . . . . .	1
1.2	Goal of the Thesis . . . . .	2
1.3	Outline . . . . .	2
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Reproducibility Computational Studies . . . . .	5
2.1.1	Reproducibility vs. Replicability . . . . .	5
2.1.2	Criteria for Reproducible Studies . . . . .	5
2.2	Camouflage Detection Algorithm . . . . .	8
<b>3</b>	<b>Replication of the Camouflage Detection Algorithm</b>	<b>13</b>
3.1	Approach for Replicating the Algorithm . . . . .	13
3.2	Replication of the Source Code . . . . .	14
3.2.1	Diagnostic Tool for Evaluation . . . . .	15
3.2.2	Unit Test Setup . . . . .	18
3.3	Large-Scale Test . . . . .	19
<b>4</b>	<b>Evaluating the Replicated Camouflage Detection Algorithm</b>	<b>23</b>
4.1	Challenges during the Translation Process . . . . .	23
4.2	Results of Diagnostics Tool After Translation . . . . .	25
4.3	Evaluation of Unit Tests and Diagnostics Tool Results . . . . .	28
4.4	Large-Scale Test Evaluation of the Replication . . . . .	36

<b>5 Conclusion and Outlook</b>	<b>39</b>
5.1 Conclusion . . . . .	39
5.2 Outlook and Future Research Opportunities . . . . .	39
5.2.1 Extension of the current Camouflage Detection Algorithm . . . . .	39
5.2.2 Algorithm for Edge Detection . . . . .	40
<b>References</b>	<b>41</b>
<b>Appendix</b>	<b>43</b>

# 1 Introduction

## 1.1 Problem Statement

Scientific publications in computer science include computations of various kinds. However, also many other disciplines nowadays critically depend on large-scale computations e.g., computational neuroscience, computational biology and computational psychology to name a few. Especially when we look into the field of image processing in computational psychology, the results of such computations have the goal of quantifying the human visual system.

But many scientific publications do not provide all the information which is needed to reproduce the experiment or computation. Especially for computational studies, the research papers additionally need to contain significant content related to the presented algorithms. This makes it a difficult task to create a replication that is capable of providing the same or similar result as the original computation.

There are many reasons for the lack of information. Kovac (2007) names cultural, educational and collaborative issues but also data issues and intellectual property issues as the problems that need to be considered and minimized to achieve a reproducible research. This will be described more in detail in Section 2.1.2.

These issues contribute to the replication crisis in science, which is described in Rougier et al. (2017). The case study from Vandewalle, Kovačević, and Vetterli (2019) visualizes the replication crisis. It analyzed the reproducibility of 134 different papers published in IEEE Transactions on Image Processing in 2004. The results of the case study can be seen in Table 1 of Figure 1.1. Only nine percent of the papers published their code, 33 percent contained pseudocode for the described algorithms and only 33 percent had their data set available online to name a few criteria they used to measure the reproducibility of the papers. This shows how hard it is to reproduce a computational study because there is a high chance that a given publication does not have all the information needed.

## 1 Introduction

**[TABLE 1] RESULTS OF REPRODUCIBILITY STUDY ON IEEE TRANSACTIONS ON IMAGE PROCESSING PAPERS PUBLISHED IN 2004. AVERAGE SCORES OVER THE 134 PAPERS ARE PRESENTED.**

DETAILS	ALGORITHM		PSEUDO-CODE	PROOFS	COMPARISON	CODE		DATA		
	PARAMETER VALUES	BLOCK DIAGRAM				IMPLEM. DETAILS	CODE AVAIL.	EXPLANATION OF DATA	SIZE DATA SET	DATA AVAIL.
0.84	0.71	0.37	0.33	0.27	0.64	0.12	0.09	0.83	0.47	0.33

Figure 1.1: Results of a case study which evaluated 134 different papers, which were published in IEEE Transactions on Image Processing in 2004. This table shows different criteria that needs to be fulfilled for a reproducible research and shows the average scores that the papers received for each criterion. Source: (Vandewalle et al., 2019).

## 1.2 Goal of the Thesis

In this thesis, I will address the reason for the replication crisis and describe the criteria that needs to be fulfilled to achieve a high reproducibility on scientific publications. I will also provide a method on how to replicate and evaluate a scientific publication. The dissertation "Camouflage Detection & Signal Discrimination: Theory, Methods & Experiments" by Abhranil Das (2022) will serve as an example for a scientific publication. It describes an algorithm which quantifies the human's perception of objects in an image. A detailed description is provided in Section 2.2. The goal is to successfully reproduce this algorithm in Python. In my thesis, I will relate to this algorithm as the camouflage detection algorithm.

## 1.3 Outline

This work will begin with an introduction into the background in chapter 2, which is needed for the following chapters. Here, I will describe how reproducibility and replicability in computational studies are defined and which criteria can be applied to a scientific publication to identify the reproducibility of it. Afterwards, I will present the camouflage detection algorithm.

In chapter 3, I will show the approach on how I will replicate the camouflage detection algorithm. To validate the replication, I created unit tests for each function in the algorithm and implemented a diagnostic tool. These two methods are used to improve the replication and to finally achieve a valid and reliable replication. To test the replication,

a large-scale test will be made after finalizing the replication to verify the validity on a large data set. Next, the implementation of the approach discussed in the previous chapter will be evaluated in chapter 4. First, the challenges that occurred during the process of replicating the algorithm will be covered. Furthermore, the result of the unit tests and diagnostic tool runs will be presented and the large-scale test will be evaluated. Finally, the chapter 5 will summarize this thesis and describe what can be concluded from it. Additionally, an outlook for future research opportunities will be provided.

## *1 Introduction*

## 2 Background

### 2.1 Reproducibility Computational Studies

#### 2.1.1 Reproducibility vs. Replicability

The terms reproduction and replication do not have a common definition. Each research area has a different understanding of what reproduction and replication mean. Furthermore, the environmental context, in which they conduct their experiments or computations, and the methods employed in scientific publications have a significant impact on these definitions (Fidler & Wilcox, 2021). In my thesis, I will use the definition from Rougier et al. (2017).

Reproducibility is defined as using the same software that was used in the original publication and running it under the same environmental setup. The goal is to receive the same results as described in the publication. For that, the input data for the computation must be provided. According to this definition, reproduction of a computational study does not require any own implementation.

Replicating a computational study, on the other hand, is about implementing a software or algorithm based on for example pseudocode, block diagrams or a detailed description that explains the algorithm (Rougier et al., 2017). The same definition applies when the publication already contains code, and it requires translation into a different programming language for personal purposes. Therefore, during the replication process we need to make sure that the results of the original code matches the ones from the replication or, depending on the use case, if the results are close enough to the original ones.

#### 2.1.2 Criteria for Reproducible Studies

A scientific publication is considered as a reproducible research if three main criteria are fulfilled. A reproducible research needs to contain a good explanation of the algorithm, provide access to the code and need to make the data set available and described so

## 2 Background

that it can be used to compute the results (Vandewalle et al., 2019). Many computational research papers do not provide enough information on these three main criteria to replicate the publication. There are multiple factors reducing the reproducibility of a scientific research.

Kovac (2007) names cultural, educational and collaborative issues but also data issues and intellectual property issues in the field of image processing. The cultural issues describe that normally, reproducibility is not the main focus when publishing a scientific work, while other factors such as the novelty of a paper have a higher value. This can lead to the issue that known algorithms will be modified for specific use-cases to achieve novelty. These modifications are seen as less prestigious in science, since novelty will be prioritized over practical applications.

Furthermore, educational issues can also cause a lower prioritization of reproducibility, since clear standards for code writing and sharing are missing. Specially when researchers are working with data sets from external sources, it can lead to difficulties while obtaining the permission to share the data set. This is also referred to as data issues. The same problem arises when researchers are working together with companies or similar institutions, which do not want to make specific parts of their algorithm public. In general, the ownership of algorithms and data sets makes the reproducibility of scientific work more challenging. This is the focus of the intellectual property issues and collaborative issues in Kovac (2007).

To visualize the reproducibility of a research, Vandewalle et al. (2019) created a list of questions. These questions can also be seen as criteria that need to be fulfilled to make a research reproducible. They were used to perform the case study described in Section 1.1. The criteria are grouped into three main categories, the reproducibility of algorithm, code and data set. The algorithm in a reproducible research requires exact parameter values, block diagrams, pseudocode, and proof for all theorems. A sufficient description of the algorithm is also required, as well as a comparison with other similar algorithms. For the code, the implementation details, such as the programming language, frameworks, libraries, etc., must be specified. Additionally, the code needs to be published and be available online.



Criteria Category	Criteria for a Reproducible Research
Algorithm	Detailed description
	Parameter values
	Block diagram
	Pseudocode
	Proofs
	Comparison to other algorithms
Code	Implementation details
	Code availability
Data	Data set explanation
	Acceptable size of data set
	Data availability

Table 2.1: This table represents the criteria for a reproducible research. The criteria are divided into three main categories. The algorithm criteria, code criteria and data criteria.

Similarly, the data must be available online and a sufficient description of it as well as an acceptable data size must be provided. An acceptable data size is achieved, when the data set is large enough so that it can be representative, but still small enough that the computation can be done in a specific time frame. An overview of all criteria can be seen in Table 2.1.

To apply the criteria to a research, points will be assigned for each criterion that indicates whether the criterion is fulfilled or not. The points can be set to 0, 0.5, 1 or N/A, while 0 means that the criterion is not satisfied, 0.5 stands for a partially satisfied criterion and 1 for a completely fulfilled one. N/A means that the criterion is not applicable, which will be assigned if a specific criterion will not be taken into consideration for a given research. According to this definition, a scientific publication can get a maximum of 11 points. Reaching this amount of points means that a given scientific publication is completely reproducible. These criteria will be applied to the Camouflage Detection Algorithm described in Das (2022).

## 2.2 Camouflage Detection Algorithm

The Camouflage Detection Algorithm (CDA) from Abhranil Das' dissertation is an algorithm which quantifies the visibility of a circular object that is placed in an image by a numerical value. This value is also called edge power. To determine the edge power for an image, the CDA takes the gradients across the edge of the object and computes the edge power based on the lengths of these gradients. The edge power represents the visibility of the object. An object with a high degree of visibility is associated with a large edge power value.

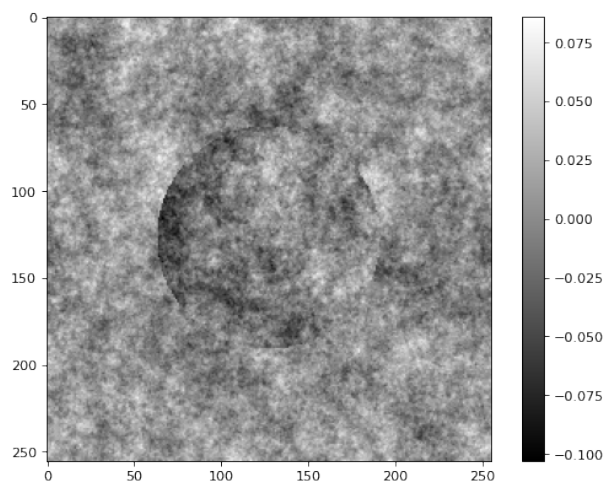


Figure 2.1: This is an example stimulus for the Camouflage Detection Algorithm (CDA). The Stimulus includes a pink noise background texture and has a circular target in the center of the stimulus, which has the same pink noise texture as the background but rotated by 90 degrees.

The image can have any arbitrary, grayscale texture as background, as well as for the object. The CDA only computes the edge power for circular objects. The background texture combined with the object, which will be referred to as the target, form together the stimulus. An example of a stimulus with a pink noise background texture and a target with a different instance of a pink noise texture can be seen in the Figure 2.1. A pink noise is a white noise which is filtered in its amplitude. Since natural images share similarities in the amplitude spectrum with the pink noise, the pink noise is used as an example texture for a camouflaged object (Das, 2022).

## 2.2 Camouflage Detection Algorithm

The Algorithm 1 represents the general structure of the CDA in pseudocode. The function that computes the edge power is also called `EDGE_POWER()` and it takes three additional arguments beside the stimulus. `target_radius` defines the radius of the target, which by default is set to 64. `n_edge` is the amount of vectors returned by the interpolation of the target's edge vectors. In the interpolation, the vectors are distributed equally across the target's edge. The `kernel_size` specifies the size of the kernel used in the steerable gradient filter.

---

**Algorithm 1** Camouflage Detection Algorithm

---

```
function EDGEPOWER(stim, target_radius=64, n_edge=1000, kernel_size=[1,
3])
    bg_size = stim.shape[0]
    target_center = floor(bg_size/2 - 1) * [1,1]
    theta_array = THETA_FIELD(bg_size, target_center)
    (mask_edge, mask_normal) = CIRCULAR_TARGET_MASK(bg_size, target_radius)
    stim_grad = STEERABLE_GRADIENT(stim, kernel_size)
    target_edge_field = NORMALIZE_GRADIENTS(stim, stim_grad, mask_normal)
    th_edge = FILTER_AND_SORT_THETA_EDGE(theta_array, target_edge_field, mask_edge)
    th_edge_warp = WARP_AND_CREATE_UNIQUE_EDGE_MATRIX(th_edge)
    target_edge_vector = INTERPOLATE(th_edge_warp, n_edge)
    target_edge_power = mean(target_edge_vector)
    return (target_edge_power, target_edge_vector)
end function
```

---

The Algorithm 1 is already the function structure, which I implemented in the Python replication. It is presented in this section, to provide a better understanding of the algorithm. This structure is not present in the original code. As shown in the example code snippet in Figure 2.2 most of the code is not separated into multiple functions. The implementation is done in one single Matlab file, except for some functions like the `STEERABLE_GRAD()` function. The following description of the algorithm is based on the implementation of Das (2023). All the inputs and outputs of the functions can be seen in a flow diagram in Figure 2.3.

First, the algorithm calls the function `THETA_FIELD(bg_size, target_center)` which computes an array that contains the polar angle of each point of a stimulus, called theta,

## 2 Background

```
83 % stimulus gradient using steerable filter:
84 stim_grad=lib.steerable_grad(stim, kernel_size);
85
86 % local luminance*contrast (std):
87 % define local patch neighbourhood
88 nhoud_radius=kernel_size(1)*kernel_size(2);
89 nhoud_size=2*ceil(nhoud_radius)+1;
90 nhoud=false(nhoud_size);
91 nhoud_center=(floor(nhoud_size/2)+1)*[1 1];
92 for i=1:nhoud_size
93     for j=1:nhoud_size
94         if norm([i,j]-nhoud_center)<=nhoud_radius
95             nhoud(i,j)=true;
96         end
97     end
98 end
99 stim_std=stdfilt(stim, nhoud);
100
101 % normal gradient
102 normal_gradient_field=mask_normal(:, :, 1).*stim_grad(:, :, 1)+mask_normal(:, :, 2).*stim_grad(:, :, 2);
103
104 % normalize by std
105 target_edge_field=normal_gradient_field./stim_std;
106
107 % table of theta and edge
108 th_edge=sortrows([theta_field(mask_edge), target_edge_field(mask_edge)]);
109
```

Figure 2.2: This figure shows a code snippet from the original code. Here we can see the part of the implementation which is covered by the `steerable_gradient()`, `normalize_gradients()` and `filter_and_sort_theta_edge()` functions in the replication. Source: (Das, 2023)

with respect to the target's center. These theta values from the *theta\_field* are later on required in the interpolation step to equally distribute the edge vectors along the target's edge. To identify the target border, a binary mask is needed that represents it. Furthermore, a mask for the normals lying on the target's edge is required so that the stimulus gradients in the further part of this algorithm can be normalized. This is done by the `CIRCULAR_TARGET_MASK(bg_size, target_radius)` function.

The crucial step happens in the `STEERABLE_GRADIENT(stim, kernel_size)` function. Here, the steerable gradient filter is computed and applied to the stimulus. This filter gives us a gradient for each pixel in the stimulus. The length of a gradient is the quantification of the edge information. So with increasing gradient length, the visibility of the edge also increases. With that, we already have all the information which is needed to compute the edge power.

To get meaningful edge information out of the stimulus' gradients, first the gradients need to be normalized in the function `NORMALIZE_GRADIENTS(stim, stim_grad, mask_normal)`.

## 2.2 Camouflage Detection Algorithm

Next, the normalized gradients will be put together with their corresponding theta values in the `FILTER_AND_SORT_THETA_EDGE(theta_array, target_edge_field, mask_edge)` function. They will be filtered by the `mask_edge` to retrieve the theta values and gradients at the edge of the target.

At this moment, the edge gradients, also called edge vectors, are not evenly spread across the target's border since our image is defined in a pixel grid where we can't describe a perfect circle. To get equally distributed edge vectors, they need to be interpolated.

Therefore, the current array containing the theta values and edge vectors, called `th_edge`, will be wrapped on each side of theta in the `WARP_AND_CREATE_UNIQUE_EDGE_MATRIX(th_edge)` function. This means, it copies the gradients two times and the theta values are being shifted by  $-2 * \pi$  and  $+2 * \pi$  and saved into the `th_edge` array. This function also removes all duplicated theta values from the array by averaging the edge vectors for the multiple occurrences of theta. Thereafter, the interpolation can be applied to the edge vectors with the function `INTERPOLATE(th_edge_warp, n_edge)`. From the resulting `target_edge_vector` array, the edge power for the stimulus is computed by the mean of the squared edge vectors.

## 2 Background

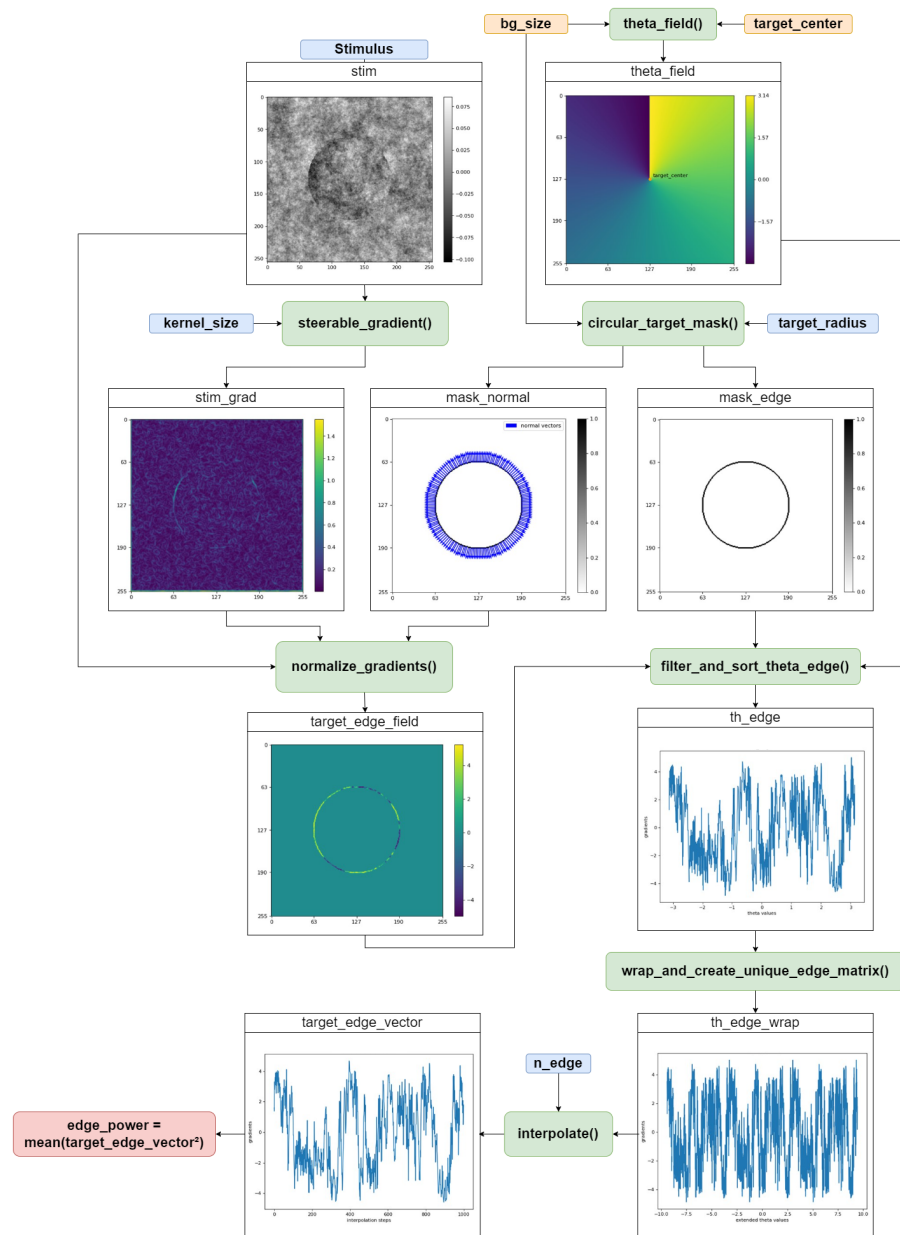


Figure 2.3: The figure visualizes the inputs and output of each function in the CDA. The blue rectangles describe the input variable of the `edge_power()` function. The orange rectangles represent variables, which are computed outside the functions. The green rectangles indicate the functions. All other variables are visualized in an image or diagram, having their name at the top of each of them. When a function has an ingoing arrow, the connected variable is considered as one of the input variables. While outgoing arrows are leading to the output variable. The returned edge power value of the algorithm is represented by a red rectangle.

# 3 Replication of the Camouflage Detection Algorithm

## 3.1 Approach for Replicating the Algorithm

To reproduce the CDA I use as the base the dissertation "Camouflage Detection & Signal Discrimination: Theory, Methods & Experiments" by Abhranil Das (Das, 2022) and also the poster from the Vision Sciences Society annual conference from May 2022 (Das & Geisler, 2022), where the dissertation was presented. But both sources together are not sufficient to reproduce the CDA. The poster provides a general overview of the topic of camouflage detection and the core statements of the dissertation, which is helpful for understanding how the CDA in works, but it does not provide any information for fulfilling the criteria of a reproducible study. Since the goal of a poster is to provide an audience a brief overview of a topic, it can be expected that for reproducing an algorithm it won't provide the necessary information.

The dissertation, on the other hand, contains more information about the CDA. Here, I will apply the criteria for a reproducible research. The scores for each criterion can be seen in Table 3.1. To achieve the goal of replicating the CDA, a proof of the theorems and a comparison with other algorithms are not providing any information that could be used for the replication. Furthermore, the criterion of having an acceptable size of the data set and its availability is not relevant in this case, since the given explanation of the data set is enough to reimplement the stimulus. With the given description, an arbitrary amount of stimuli can be generated and used in the replication. That is why all these criteria have their score set to N/A.

The dissertation provides a general description of how the algorithm works, which is sufficient to understand what the idea of the algorithm. As previously mentioned, a detailed description of the data set is also provided. These criteria received a score of 1 because of that. But to replicate the algorithm, more information like the parameter

### 3 Replication of the Camouflage Detection Algorithm

values, block diagrams and pseudocode needs to be provided. Unfortunately, they are not in the dissertation. But the crucial point was the description of the steerable filter. It was only described in words, and no functions or implementation details were provided for it. That is why, with the given information, it was not possible to replicate the algorithm.

Due to this fact, I requested the original code from the author of the dissertation. After requesting the source code from the author, it was provided in Matlab. That is why both code criteria received a score of 0.5 each since the code wasn't available online before, but it was provided after requesting it. Since the original task was to replicate the algorithm in Python, the main goal became to extract the relevant parts of the source code and reimplement it in Python.

Criteria Category	Criteria for RR	Score
Algorithm	Detailed description	1
	Parameter values	0
	Block diagram	0
	Pseudocode	0
	Proofs	N/A
	Comparison to other algorithms	N/A
Code	Implementation details	0.5
	Code availability	0.5
Data	Data set explanation	1
	Acceptable size of data set	N/A
	Data availability	N/A

Table 3.1: In this table, the scores are applied to each criterion for the algorithm described in the dissertation.

## 3.2 Replication of the Source Code

To assure that the replication is valid and reliable, I divide the replicated code into multiple segments according to their logical purpose. For each segment, I created a Python method to test them individually. Since the tests will be used to verify the validity by coverage testing, I also created a diagnostic tool, that will approve the reliability of the



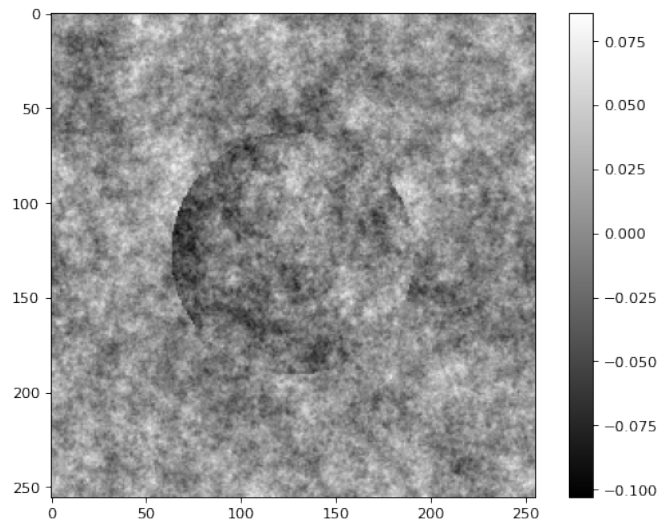


Figure 3.1: Example stimulus for the first evaluation step of the diagnostics tool. The stimulus is a pink noise background texture having a circular target.

replication. While the test will be only applied for a few stimuli, the diagnostic tool will contain a lot larger number of stimuli.

The reason for it is that after the first step of translating the original Matlab code into Python, the replication was not returning the correct edge power values as expected. The tests are a way to go through each method and improve the code step by step. It is better to test on a few stimuli, since the execution of some methods takes a relatively long time to compute the results.

#### 3.2.1 Diagnostic Tool for Evaluation

To assure the reliability of the replication, I created a diagnostics tool which gives me a measurable value to define how reliable the replication is. Due to numerical approximations in some parts of the code and also because of some differences in the predefined function in each programming language, the replication won't return the same edge power as the original code. The goal of the diagnostic tool is to generate a large set of stimuli in different scenarios, compute the edge power with the replication and original code and then compare them to see if the results are correlating. Furthermore, the

### 3 Replication of the Camouflage Detection Algorithm

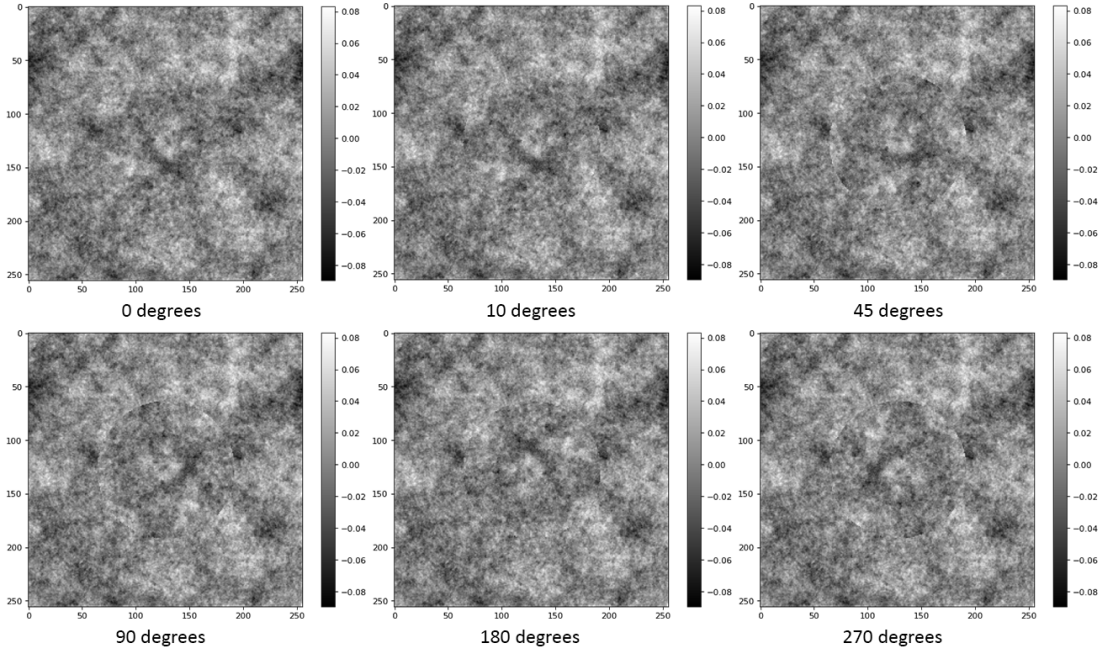


Figure 3.2: Example stimuli for the second evaluation step of the diagnostics tool. Each stimulus has a pink noise background texture with a circular target which is rotated by a certain amount of degree.

diagnostic tool computes the mean square error (MSE). The MSE is described by

$$MSE = \sum_{x=0}^X (y - x)^2, \quad (3.1)$$

where X is the dataset containing the edge power values returned by the replicated code in Python and y equals the expected value for edge power x, meaning that y is the edge power returned by the original MATLAB code. The diagnostic tool contains three evaluation steps. In the first step, I create 100 different stimuli. Each stimulus has fixed parameters, except for the seed. This will create stimuli with a pink noise texture as background with a size of 256 pixels and a circular target size of 64 pixels. The seed will be set to the value of a counter which is initialized with one and will be increased by one each iteration step up to 100. The seed ensures that the pink noise, which normally is generated randomly, will be a fixed variation of the pink noise. An example stimulus can be seen in Figure 3.1.

Then I iterate over all stimuli, compute the edge power with the original code and also

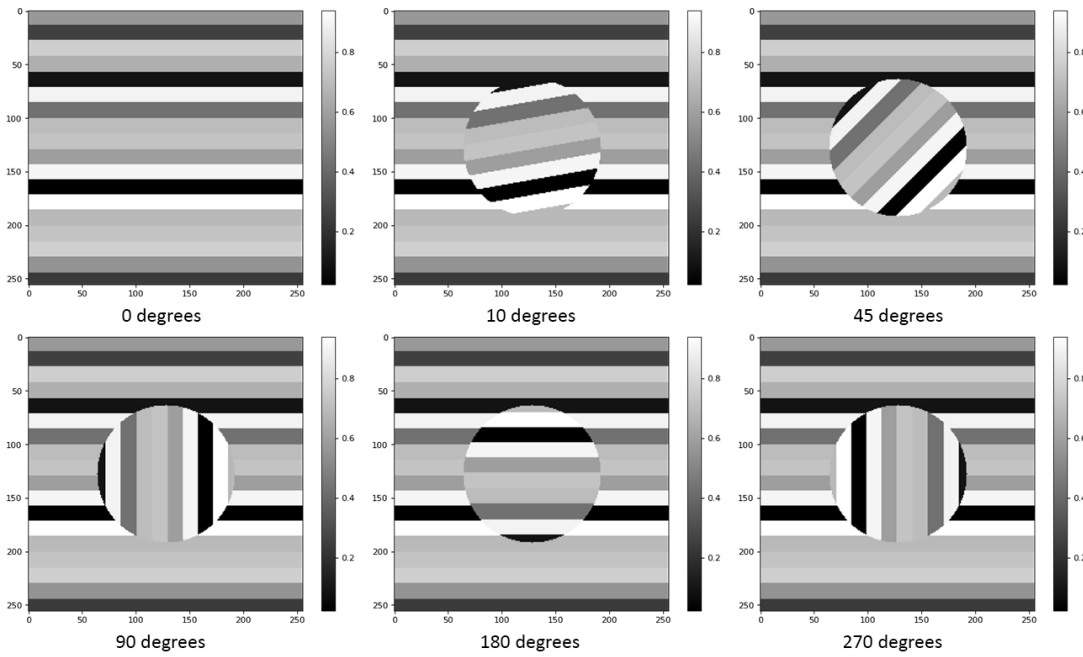


Figure 3.3: Example stimuli for the third evaluation step of the diagnostics tool. Each stimulus contains a texture with 15 pixels wide horizontal lines and a circular target which is rotated by a certain amount of degree.

with the replicated code, and then determine the MSE of the whole data set. Based on the results, I created a diagram which has the edge power value from the original code on the x-axis and the edge power values from the replication on the y-axis. If the results from the replicated code will match the ones from the original code, they will lie on the function  $y = x$  in the diagram.

In the second evaluation step of the diagnostic tool, I use a fixed stimulus where I will alter the magnitude of rotation of the target. Therefore, the stimulus will contain a pink noise texture as background with a size of 256 pixels, a circular target size of 64 pixels and a constant seed ( $seed = 1$ ). With this setup, I will create 360 different stimuli where I will rotate the target around its center. Per iteration, I will increase the magnitude of rotation by one degree starting at zero degree up to 360 degrees. Then, similar to the first evaluation step, I will compute both edge power values and create a diagram where I compare the reference values with the actual values, and also determine the MSE based on the results. Some example stimuli for this diagnostic tool step can be seen in Figure 3.2.

### 3 Replication of the Camouflage Detection Algorithm

For the last evaluation step, I use the same method as in the second step. So, I will also alter the magnitude of rotation of the target from zero to 360 degrees in one degree intervals, but with a different background texture. The texture consists of 15 pixels wide horizontal lines, where for each of them a random value between zero and one will be assigned. In this diagnostic tool stage, I employ a different texture than before to examine the possibility that some points at the edge may not have a visible border. For this points, the gradients have a length of zero, which represents a borderline case.

Besides the texture, the remaining parameter will have the same values as in step two. So, the stimulus will have a background with a size of 256 pixels, a circular target size of 64 pixels and a constant seed ( $seed = 1$ ). Some example stimuli for this diagnostic tool step can be seen in Figure 3.3. For evaluating the results, I will also determine the MSE and visualize the results in a diagram, comparing the reference values with the actual ones.

#### 3.2.2 Unit Test Setup

After I finish implementing the CDA, I start to write the tests for validating the replication. The original code isn't segmented into smaller functions, like it is in my replication. To generate the test data, I need to add a function at the specific point in the Matlab code, which writes the current state of the variables into a file. So, I added this function at every line where the corresponding segment in my replication expects an input and output. To give an example, one of my methods calculates the theta values for each pixel in the stimuli. This means that I need to save the input stimulus as the input data for the `theta_field()` function, and also save the resulting theta field array at the line where it will be computed. After this segment, the theta field array can be used as the input data for the following functions.

To have a larger variety in the test data, I generate 10 random stimuli and create the test data for them using the original Matlab code. This means that for each stimulus I save the state of the variables to use them as input or output for my methods on the tests, as it can be seen in Table 3.2.

Besides these methods, I am also testing the functions which I needed to implement by myself since Matlab has some predefined functions that can't be imported by any Python modules. These functions are the `filter2()`, `bwperim()` and `stdfilt()`. For every function, a definition and discussion on why they can't be used in Python directly will be provided

in Section 4.4. To test them, each function has their corresponding method in which I implemented them. These tests have the aim to verify that my reimplementations are returning valid results.

Filename of variable	methods input	methods output
stim.mat	STEERABLE_GRADIENT() NORMALIZE_GRADIENTS() [indirectly THETA_FIELD() CIRCULAR_TARGET_MASK()]	-
theta_field.mat	FILTER_AND_SORT_THETA_EDGE()	THETA_FIELD()
mask_edge.mat	FILTER_AND_SORT_THETA_EDGE()	CIRCULAR_TARGET_MASK()
mask_normal.mat	NORMALIZE_GRADIENTS()	CIRCULAR_TARGET_MASK()
stim_grad.mat	NORMALIZE_GRADIENTS()	STEERABLE_GRADIENT()
target_edge_field.mat	FILTER_AND_SORT_THETA_EDGE()	NORMALIZE_GRADIENTS()
th_edge.mat	WRAP_AND_CREATE_UNIQUE_EDGE_MATRIX()	FILTER_AND_SORT_THETA_EDGE()
th_edge_wrap.mat	INTERPOLATE()	WRAP_AND_CREATE_UNIQUE_EDGE_MATRIX()
target_edge_vector.mat	-	EDGE_POWER()
target_edge_power.mat	-	EDGE_POWER()

Table 3.2: This table shows all the files, which I used to save the variable states in the original code so that I can use them for testing of the different functions.

### 3.3 Large-Scale Test

After successfully assuring the validity and reliability of my replication, I will test my replication on a large set of stimuli. This time the stimuli will be generated using the stimupy library from Marianne Maertens, Lynn Schmittwilken and Joris Vincent (Schmittwilken, Maertens, & Vincent, 2023). I will utilize the binary and narrowband noise from the stimupy.noises module and also from the stimupy.noises.naturals I will

### 3 Replication of the Camouflage Detection Algorithm

use 1/f noise, pink noise and brown noise. For each of the five background textures, I will generate 1000 random stimuli. Depending on the configuration options in the stimupy library of each texture, the test setup will differ. For the binary, pink, and brown noise texture, the test setup will be the same since they got the same configuration options. For the 1000 stimuli, their size will be set to  $size \in [200, 400, \dots, 1000]$  for every 100 stimuli. For each 100 stimuli having the same size, I will also alter the intensity range (ir) according to  $ir \in [(0, 1), (0, 2), \dots, (0, 10)]$  for every 10 stimuli.

Figure 3.4 shows three different example stimuli for each texture type having a different configuration of the variables. The images A to C differ in their background size over each texture type. Image A has a size of 200 pixels, while B has a size of 600 and C a size of 1000 pixels. Additionally, the texture types of binary, pink and brown noise vary in their intensity range. The images A have a range of (0,1), while the images B have a range of (0, 5) and images C a range of (1, 10). The stimuli having the narrowband texture also have a variation in their center frequency being 0.5, 0.75 and 1 for each stimulus A, B and C. The 1/f noise stimuli, on the other hand, have also different exponent values, being 1 for image A, 1.5 for image B and 2 for image C. Since the narrowband and one\_over\_f noise texture have more options that can be altered, for these two texture types I will do the same size variation as for the other stimuli, but the intensity ranges will be fixed at (0,1). For the narrowband noise texture, I will alter the center frequency to  $cf \in [0.5, 1, 1.5, \dots, 5]$  for every 10 stimuli, while for the one\_over\_f noise texture, I will modify the exponent  $exp \in [1.0, 1.1, \dots, 2]$  for every 10 stimuli. Some example stimuli for each texture type can be seen in Figure 3.4.

In total, I will generate 5000 stimuli. For each of them I will compute the edge power by using the original code and also by using my replication. Thereafter, I will compare the results and compute the MSE to see if there are any differences. Both edge power values from each implementation should ideally return the same values, which would lead to an MSE of  $MSE = 0$ . The goal of the large-scale test is to assure the correctness of the CDA and that it is returning the expected edge power values for each stimulus of a large and diverse data set.

### 3.3 Large-Scale Test

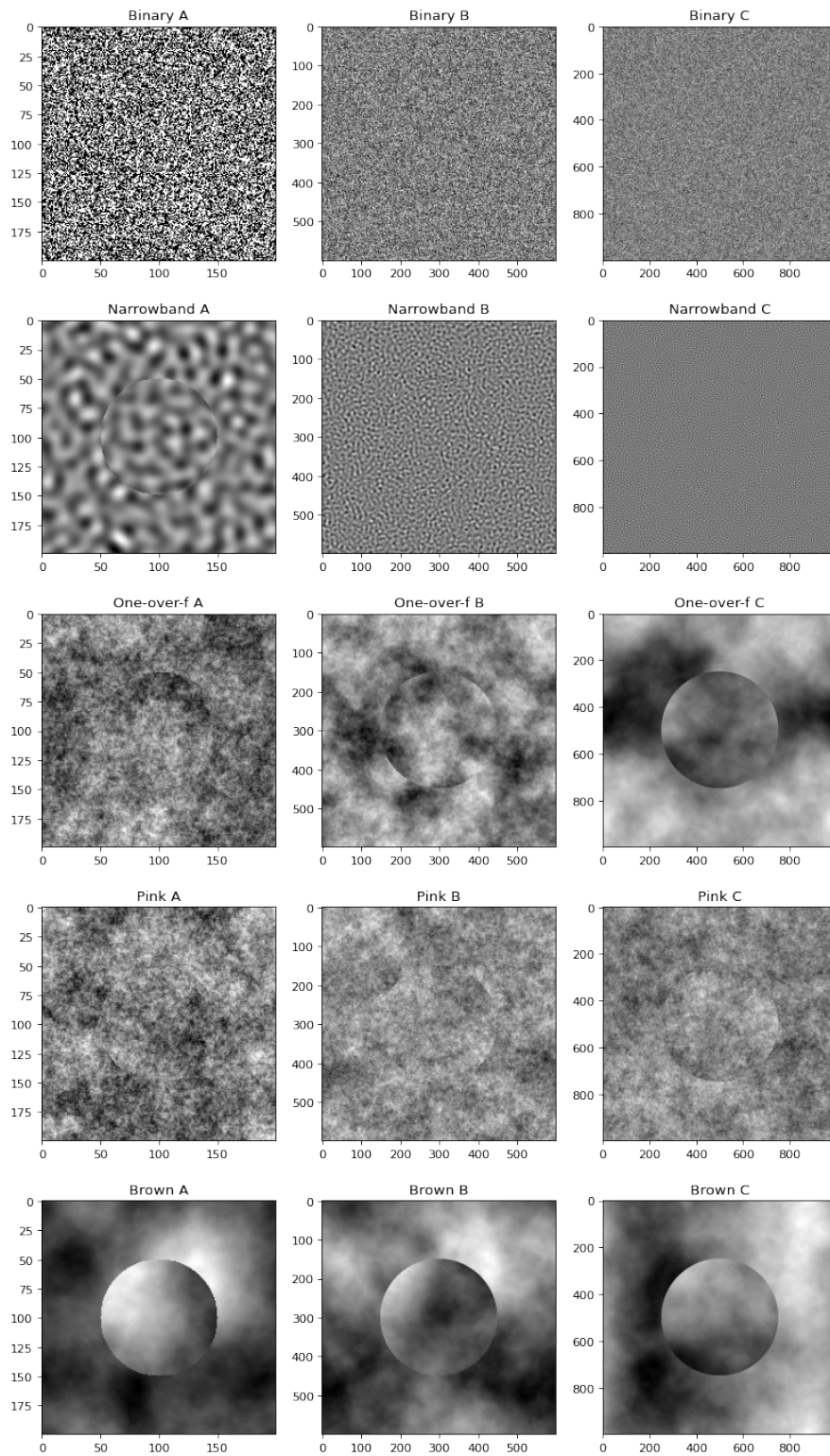


Figure 3.4: Example stimuli from the data set used for the final larger-scale test. Each row represents a different texture type. For each texture type, three different variations are shown.

### *3 Replication of the Camouflage Detection Algorithm*



## 4 Evaluating the Replicated Camouflage Detection Algorithm

### 4.1 Challenges during the Translation Process

During the process of translating code from one programming language to another, some challenges may occur. Some functions exist in one programming language, but they are not implemented in the other one. An example of it would be the `FILTER2()` or `BWPERIM()` function in Matlab. Each of these two functions provides examples of how such a translation into python might work.

The first case is shown by the `FILTER2()` function, which is defined as

$$\mathbf{function\ FILTER2}(\mathbf{C}, \mathbf{I}, \mathbf{shape}) \tag{4.1}$$

where  $\mathbf{I}$  is an array representing the image as a matrix,  $\mathbf{C}$  is the correlation template and  $\mathbf{shape}$  is a string indicating which information of the filtered data is used for the resulting array. This function applies a filter on array  $\mathbf{I}$  using the coefficients of matrix  $\mathbf{C}$  and returns the filtered array (Zhi & Jiang, 2017). In the camouflage detection algorithm, it is used to apply the steerable filter on the stimulus to get the steerable gradients. So in this case,  $\mathbf{C}$  is the steerable filter and  $\mathbf{I}$  is the input stimulus. The  $\mathbf{shape}$  is set to 'same'. This means that the central part of the filter is used for the resulting array. This way, the result will have the same size as the array  $\mathbf{I}$  (Gonzalez, Woods, & Eddins, 2009).

`FILTER2()` does not exist neither in Python nor in any common Python module like NumPy, SciPy and similar packages and can't be replicated directly. But this function also has an alternative definition, which is described as:

$$\mathbf{function\ CONVOLVE2D}(\mathbf{I}, \mathbf{rotate180}(\mathbf{C}), \mathbf{shape}) \tag{4.2}$$

So `FILTER2()` can also be interpreted as the convolution of the image  $\mathbf{I}$ , where the correlation template array  $\mathbf{C}$  is rotated by 180 degrees. Providing the same  $\mathbf{shape}$  type as

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

in the `FILTER2()` function, both functions will return similar results.

Using equivalent functions is a simple way to reproduce code in another programming language. But in some cases, it can happen that a function does not have any equivalent functions. In these cases, the functions need to be implemented in the target programming language, as it is the case for the `BWPERIM()` function used in the CDA. The `BWPERIM()` is defined as:

$$\mathbf{function\ BWPERIM(I)} \tag{4.3}$$

This function computes an array containing the perimeter of a binary image  $I$ . Since there is no implementation of this function in Python, I used the description of the documentation as a base to replicate this function. According to the documentation in The MathWorks, Inc. (2023), the `BWPERIM()` function is a filter which is applied to each pixel. A pixel  $p$  at location  $x$  and  $y$  is defined as:

$$p_{x,y} = \begin{cases} 1 & \text{if } p_{x,y} = 1 \text{ and } (p_{x+1,y} = 0 \text{ or } p_{x-1,y} = 0 \text{ or } p_{x,y+1} = 0 \text{ or } p_{x,y-1} = 0) \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

According to this definition, I replicated this function in Python.

There is also a third case which can occur during a translation process between two programming languages. In this case, a function has a corresponding function in the target programming language, but still needs some adjustments to reproduce the same results. This applies for example to the `STDFILT()` function from Matlab, which is defined as:

$$\mathbf{function\ STDFILT(I, nhood)} \tag{4.5}$$

This function applies a local standard derivation using the kernel  $nhood$ , which is an array with odd size in each dimension, and applies it on each pixel (Gonzalez et al., 2009). I used the `GENERIC_FILTER()` function from the `scipy.ndimage.filters` module to reproduce this function in Python. The `GENERIC_FILTER()` is defined as:

$$\mathbf{function\ GENERIC_FILTER(I, f, footprint)} \tag{4.6}$$

where a function  $f$  is applied on each pixel's neighborhood in the array  $I$ , while the neighborhood is specified in the  $footprint$  variable (The SciPy community, 2023). To replicate the `STDFILT()` function, a standard derivation function needs to be set for the function  $f$ , but the standard derivation can be computed in different ways. For example,

## 4.2 Results of Diagnostics Tool After Translation

in the official documentation of Matlab the standard derivation  $S$  of an array  $X$  where  $x_i \in X$ ,  $\mu$  equals the mean of  $X$  and  $N$  is the number of elements in  $X$ , is defined as:

$$S = \sqrt{\frac{1}{N-1} * \sum_{i=1}^N |x_i - \mu|^2} \quad (4.7)$$

In this equation the standard derivation is normalized by the factor  $N - 1$ . But there are other definitions, like the one in Python's NumPy Module, that uses only the factor  $N$  instead of  $N - 1$ . To get the same results as in Matlab, I needed to implement a standard derivation function that computes  $S$  like it is defined in Equation 4.7.

## 4.2 Results of Diagnostics Tool After Translation

After I finished the first version of the translation attempt, I used my diagnostic tool to evaluate the reliability of my replication. To achieve a high reliability, the results from both, the Matlab and Python implementation, need to match. Therefore, the MSE should be zero. The diagnostic tool creates numerous amount of stimuli and computes the edge power from my replication. It will also call the original code. Finally, the results are compared by calculating the MSE like it is explained in Section 3.2.1 and the resulting edge power values are visualized in a diagram.

In Figure 4.1 the results of the first diagnostic tool run are visualized. The first stage of the diagnostic tool, like it is described in Section 3.2.1, generates 100 different stimuli with a pink noise background and computes the edge power values from the original and replicated code. In the first diagram, the 100 stimuli are displayed by taking their resulting edge power values from both CDA implementations. On the x-axis, the edge power values from the original code are shown, while the edge power values from the replicated code are displayed on the y-axis. In this diagram, two functions are shown. The orange function shows the reference values. If the edge power value from my implementation matches the one from the original code, it means that they would lie on the function  $f(x) = x$ , which is represented by the reference values. The second function shows the actual values computed by the diagnostic tool. We can see that they do not match expected values. The actual values are distributed near an edge power value of  $EP \approx 4.3$ . Furthermore, as the expected edge power values increases, my implementation does not yield a correspondingly higher edge power value. This means that the results are not correlating with the expected values. This leads to an MSE of around 1333.

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

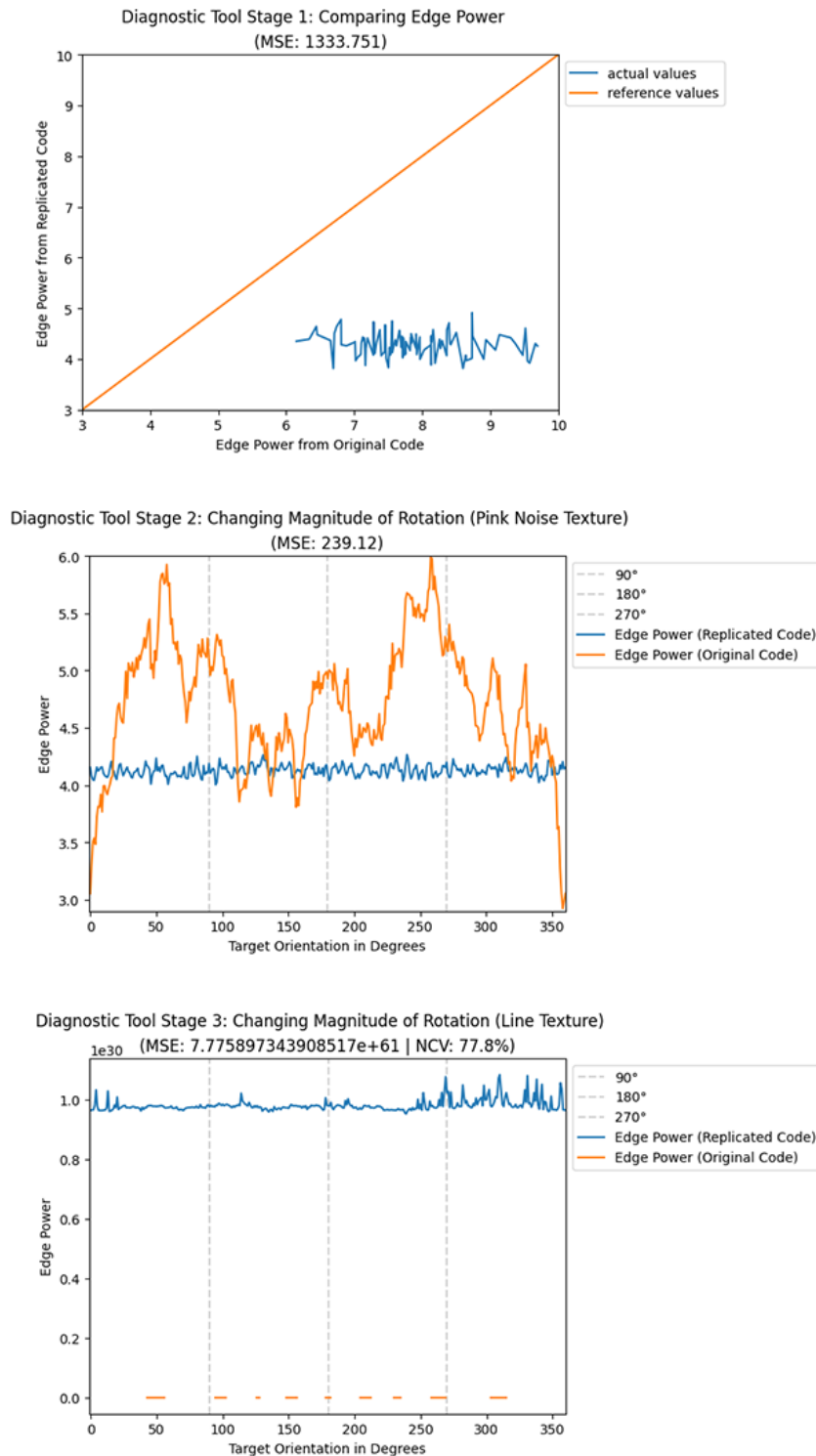


Figure 4.1: These diagrams show the results of all three diagnostic tool stages for the first execution. It shows that after the translation, the replication does not return the expected edge power values and is not correlating with the correct results.

## 4.2 Results of Diagnostics Tool After Translation

The second stage of the diagnostic tool takes one stimulus with a pink noise background and rotates the target around the target's center. The rotation takes place in one degree steps, and the edge power values from both implementations are computed for each step. Furthermore, both results will be displayed in a diagram, similar to the middle diagram in Figure 4.1. In the diagram, I plot the edge power values against the target orientation by the degree angle to compare the results of the second diagnostic tool stage. I also plotted three dotted lines which serve as a visual help to mark the rotation angles at 90, 180 and 270 degrees.

In the middle diagram in Figure 4.1, the orange function represents the results from the Matlab implementation. Depending on the rotation magnitude from the original implementation we get edge power values which start at around  $EP \approx 3$  and with increasing rotation magnitude they go up to nearly  $EP \approx 6$ . This happens since the differences between the target and the background can increase or decrease at the edge with each rotation step. This depends on how the bright and dark areas of the background are distributed by the pink noise texture. So if we rotate the target in a way that many bright areas are located next to dark ones and vice versa, then we get a high edge power like it is in the second diagnostic tool stage for the orientation  $x \approx 58$  and  $x \approx 259$ . And if these differences at the edge will get smaller again, the edge power will decrease accordingly. This can be specially seen at the orientation points  $x = 0$  and  $x = 360$ . With no rotation of the target, it won't be visible. That's why the edge power is very low at these points. We experience a strong edge power increase for the first rotation angles and a strong decrease when coming closer again to the original stimulus after rotating the target by a full 360 rotation. As shown in the middle diagram in Figure 4.1, this behaviour cannot be observed in the results of my replication. The results of my replication are displayed by the blue function. The edge power is staying constant at around 4.2 and is not correlating with the described curve progression of the orange function. This deviation creates an MSE of 239.12.

In the last stage, the diagnostic tool is doing the same as in stage two, but this time the line texture is used as the stimulus' background. The diagram has the same axes as in the previous diagram. The target's orientation will be on the x-axis and the edge power values on the y-axis. I expected here the same anomalous behaviour as in the previous stage, but this also was not the case in this stage. The results were exorbitantly high compared to the expected edge power values. While the edge power of the original

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

CDA implementation lies in a range between  $9.5 < EP < 11$ , the edge power from my implementation is around  $EP \approx 1 * 10^{30}$ . This causes the MSE to be  $\approx 7.77 * 10^{61}$ . The third stage also uses another variable to evaluate the results of this stage. It computes the percentage of not computed values (NCV). For this stage, no edge power value could be computed for 77.8% of the stimuli by the original code.

But the result of the replication show an improvement to the original code, as my implementation computes an edge power value for all target orientations. The original one only computes a finite result for 22.2% of generated stimuli. This is caused by an enhancement in the `WARP_AND_CREATE_UNIQUE_EDGE_MATRIX(th_edge)` function, where I filter out all edge vectors that contain infinite or NaN values because they would lead to an infinite edge power value. This means, that after resolving the current issues in the replication, the edge power value can be computed for a larger set of stimuli than it was possible with the Matlab implementation.

### 4.3 Evaluation of Unit Tests and Diagnostics Tool Results

In this section, I will show which functions I improved and how this influenced the results of the diagnostic tool. I did the improvements by adding the tests step by step for each function and adjusting my replication based on the error I am getting from it. I started

---

**Algorithm 2** Theta Field Function

---

```
function THETA_FIELD(stim_size, target_center)
    t_field = np.array(stim_size, stim_size)
    for i ← 1 to n do
        for j ← 1 to n do
            vec = [i, j] - target_center
            t_field[i][j] = CART_TO_POL(vec)[0]
        end for
    end for
    return t_field
end function
```

---

with the `THETA_FIELD()` function. This function computes an array which contains the polar angle (theta) of each point of a stimulus with respect to the target's center, like it is shown in Algorithm 2. This function is a good example to show how implementations

### 4.3 Evaluation of Unit Tests and Diagnostics Tool Results

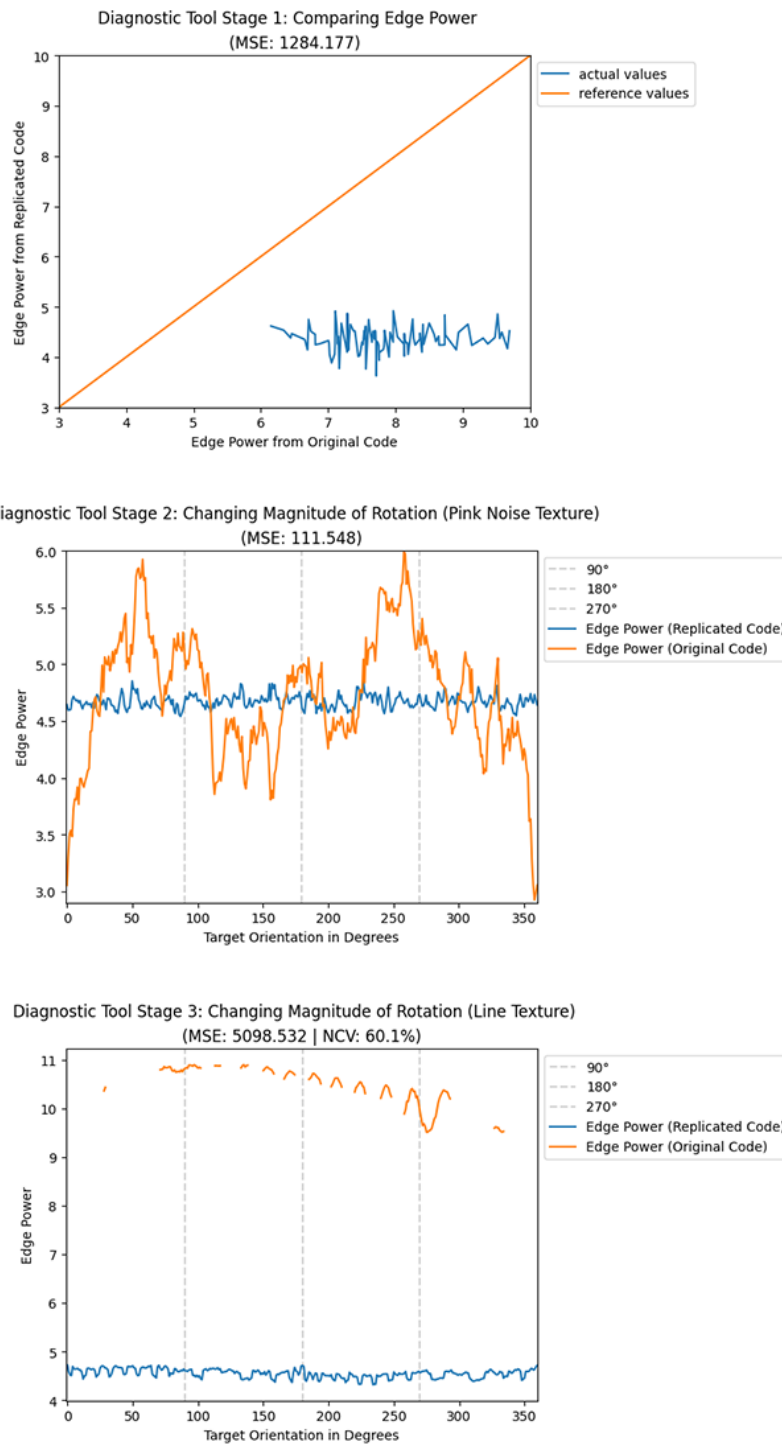


Figure 4.2: These diagrams show the results of all three diagnostic tool stages for the second execution. Here, a shift in the mean edge power of each stage in comparison to the previous run can be observed. The results from the replication are still not correlating with the expected results.

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

needs to be adjusted during a translation process to the new programming language. The replication was returning a wrong  $t\_field$  array because the array indices in Matlab start with one like in the pseudocode of Algorithm 2 in line three and four, and in Python, they do at zero. This difference causes that the vector between a pixel and the target center, which is calculated in line 5, will have a slightly different angle. So, when we extract the theta value from the polar coordinates in line 6, we get a wrong value that influences the edge power calculation in the further steps.

After correcting the shifted indices and tests for the `THETA_FIELD()` function passed, I ran the diagnostic tool again to evaluate the reliability of my replication. The result can be seen in the Figure 4.2. In the first diagnostic tool stage, we can see that the edge power values are still not correlating with the values which we are expecting. There is only a difference in the shift of the edge power values, since they are now lying at the value around  $EP \approx 4.6$ . The same behaviour can be seen in the second stage of the diagnostic tool. Before the values were lying around the value of  $EP \approx 4.3$ , but now they are lying at a value which is  $EP \approx 4.6$ . There is still no correlation visible when we compare the edge power values from the original code with the ones from the replication. The increasing mean edge power causes the mean squared error to decrease for both stages, since they dropped from  $MSE_{1,1} \approx 1333$  to  $MSE_{2,1} \approx 1284$  and from  $MSE_{1,2} \approx 239$  down to  $MSE_{2,2} \approx 111$ .  $MSE_{x,y}$  stands for the mean squared error for the diagnostic tool run  $x$  in stage  $y$ . But as we look into the third stage of the diagnostic tool, we can see a considerable improvement of the edge power values which are computed from the replicated code. The values are not at a mean edge power of  $EP \approx 7.77 * 10^{61}$  anymore and are now lying at the value of around  $EP = 4.6$ . This leads to a mean squared error of  $MSE_{2,3} \approx 5098$ , which is a significant change compared to the mean squared error from the first run. The NCV decreases from 77.8% to 60.1%.

For the next step, I wrote the tests for the `CIRCULAR_TARGET_MASK()` function. This function computes a binary mask which represents the border of the circular target. Here, only a small adaptation of the replication code was needed because the resulting target mask matched the inverted target mask from the original code. So, by inverting the target mask in the replication, the test for this function passed. This change had an influence on the results of the next diagnostic tool run, which can be observed in the diagrams of Figure 4.3.

Here we can see that the replication data is still having a very noisy-like behaviour,



### 4.3 Evaluation of Unit Tests and Diagnostics Tool Results

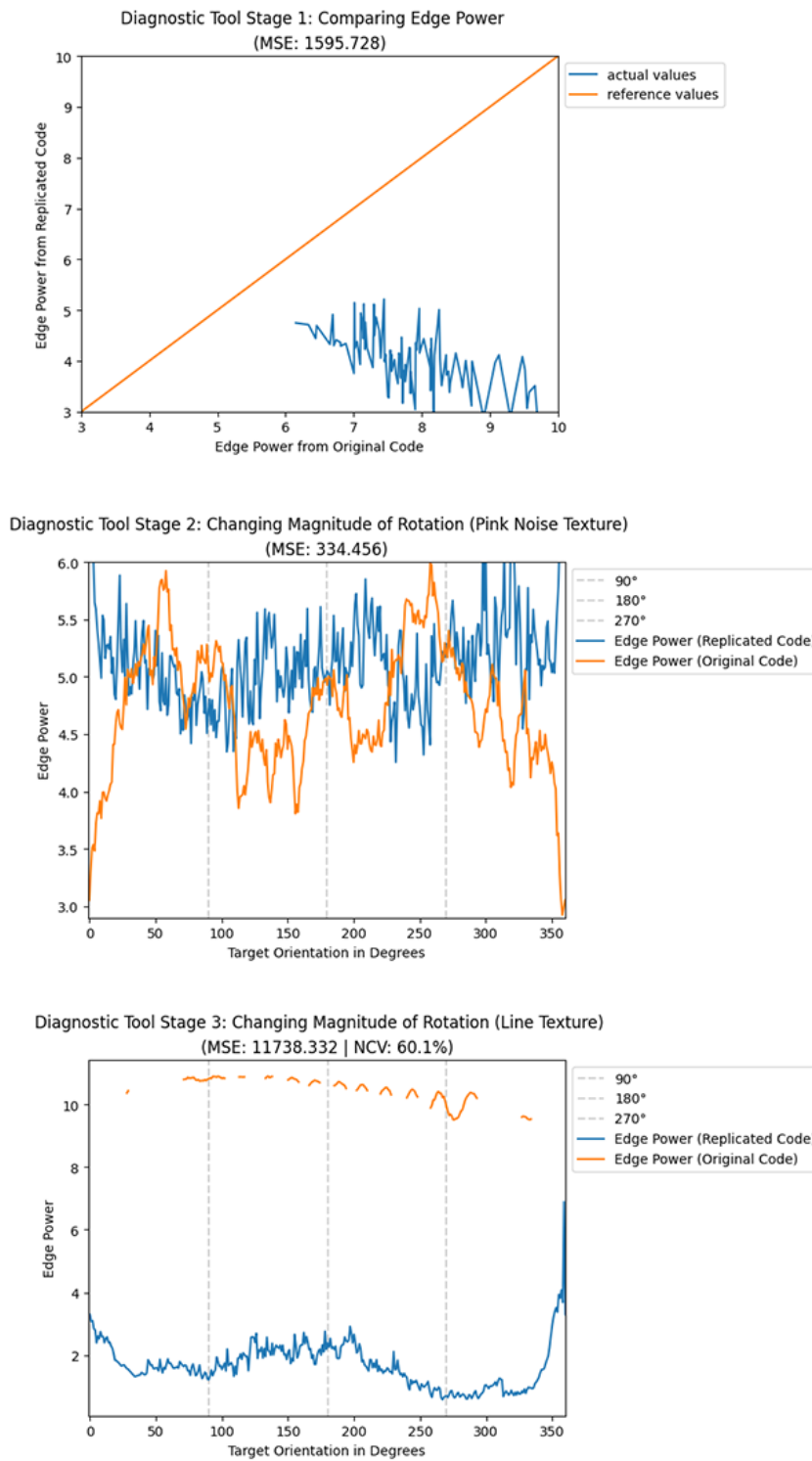


Figure 4.3: These diagrams show the results of all three diagnostic tool stages for the third execution. The results of the replication are now having a negative correlation with the expected values, but still have a very noisy behaviour.

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

but it's now correlating with the edge power values from the original code. There is a negative correlation that we can observe. A negative correlation occurs when one of the variables is increasing while the other is decreasing (Stocker & Steinke, 2017). This implies that the higher the edge power value of the original code, the lower the edge power value from the replication will be. This is what we can conclude from the first diagram of the Figure 4.3, which shows the results from the first stage of the diagnostic tool. But as we look into the last two stages, we can also see this behaviour here. Both diagrams show that the replication returns a large edge power value for target rotations that should have had a low edge power and vice versa. So to conclude, the improvement of the `CIRCULAR_TARGET_MASK()` function causes the edge power values to correlate with the expected values, but since it's a negative correlation, the MSE increases for all diagnostic tool stages. More precisely, the MSE for run one increased to  $MSE_{3,1} \approx 1595$ , while the MSE of the run two was lying at  $MSE_{2,1} \approx 1284$ . A growth in MSE is also visible for stage two, where the MSE raised from  $MSE_{2,2} \approx 111$  to  $MSE_{3,2} \approx 334$ . And last also for stage three, the MSE has a value of  $MSE_{3,3} \approx 11783$ , while we had an MSE value of  $MSE_{2,3} \approx 5098$  in the previous run.

After finishing the tests for the `CIRCULAR_TARGET_MASK()` function, I created the tests for the `STEERABLE_GRADIENT()` function. This function computes the steerable gradients for a given stimulus by applying the steerable filter on it with a provided kernel size. When writing the tests, I noticed two small issues in the code. The first one was an index shift in the code segment where the center of the steerable filter is computed. Since Matlab starts its indices at zero and Python at one, the filter center needs to be adjusted by subtracting a one from it. Second, the `STEERABLE_GRADIENT()` function uses Matlab's `FILTER2()` function to apply the steerable filter on the stimulus. Like described in Section 4.4, this function is not having a corresponding function in Python. Instead, there is the way to implement it with the `CONVOLVE2D()` function from the SciPy module. But this function contained an issue with the rotation of the steerable filter. To match the output of `FILTER2()`, the steerable filter needs to be rotated by 180. But in the code, the rotation was only happening by 90 degrees. This is probably the reason the edge power values in the third diagnostic tool run had a negative correlation with the expected values.

As we can see in the fourth diagnostic tool run results in Figure 4.4, there is now a positive correlation. In the first stage, the actual values from the replication are still

### 4.3 Evaluation of Unit Tests and Diagnostics Tool Results

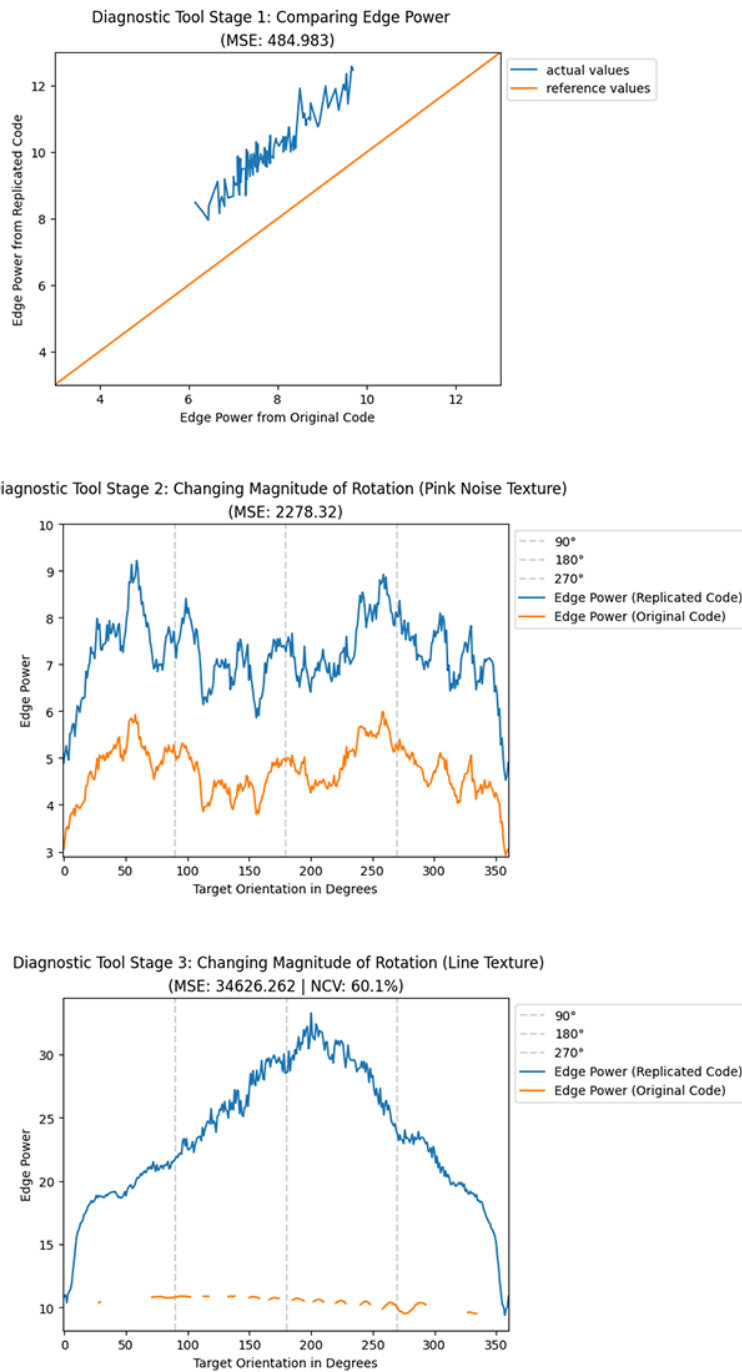


Figure 4.4: These diagrams show the results of all three diagnostic tool stages for the fourth execution. The results of the replication have a positive correlation with the expected values. But the results are still shifted by a certain amount and are not matching completely the expected values regardless of the shift. For that last stage, the edge power values even moved away from the solution.

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

having a noise in the graphs course. But the edge power values of the replication for the stimuli are now higher than their corresponding edge power value from the original code. That is the reason why the blue curve is above the orange curve. This leads to a significant decrease in the MSE value. The MSE shrinks to a value of  $MSE_{4,1} \approx 484$ . In the second stage, the correlation is even more visible than in the previous diagram. The edge power from the replication follows a similar course as the expected edge power values but shifted by an edge power of  $\Delta EP \approx 2$ . The course of the blue graph is still having a slight amount of noise in it, so it is not completely matching the course of the orange graph. When we compare the MSE from the previous one, we can see that the MSE increases with the improvement of the `STEERABLE_GRADIENT()` function. We now get a  $MSE_{4,2} \approx 2278$ , while before we had one of  $MSE_{3,2} \approx 334$ . Even if the edge power values were correlating less, the edge power value from the previous run lie closer to the original value than they do in the fourth run. But even with an increase of the MSE, we still get closer to the expected results, since the course of the blue graph improved. This means that the edge power value of the stimulus with the pink noise texture now matches better with the perceived visibility of the target than before.

But this only applies for stimuli, which have a pink noise texture. When we look into the results of the third stage, we can see that for the stimuli with the line texture, the course of the blue graph representing the edge power of the replication has moved further away from the expected graph course. This leads to an MSE increase of  $MSE_{4,3} \approx 34626$ . The NCV remains unchanged.

Next, I created the tests for the `NORMALIZE_GRADIENTS()` function. This function normalizes the steerable gradients of the stimulus and projects them onto the mask normal. For normalizing the gradients, I use the `GENERIC_FILTER()` function described in Equation 4.6 to calculate the local standard deviation for each pixel of the stimulus. This computation is an equivalent of `STDFILT()` from Matlab, like described in Section 4.4. But to get the same results as in Matlab, I can't pass NumPy's `STD()` function to the `GENERIC_FILTER()` function since the computation of the standard deviation differs. Based on the documentation of Matlab's `STDFILT()` function, I implemented the standard deviation as it is defined in Equation 4.7. This improvement causes that the replication is returning the same edge power values as the original code.

When we look at the results of the last diagnostic tool run in Figure 4.5, we can see in the first and second diagram that only the orange graph is visible, since the orange graph

### 4.3 Evaluation of Unit Tests and Diagnostics Tool Results

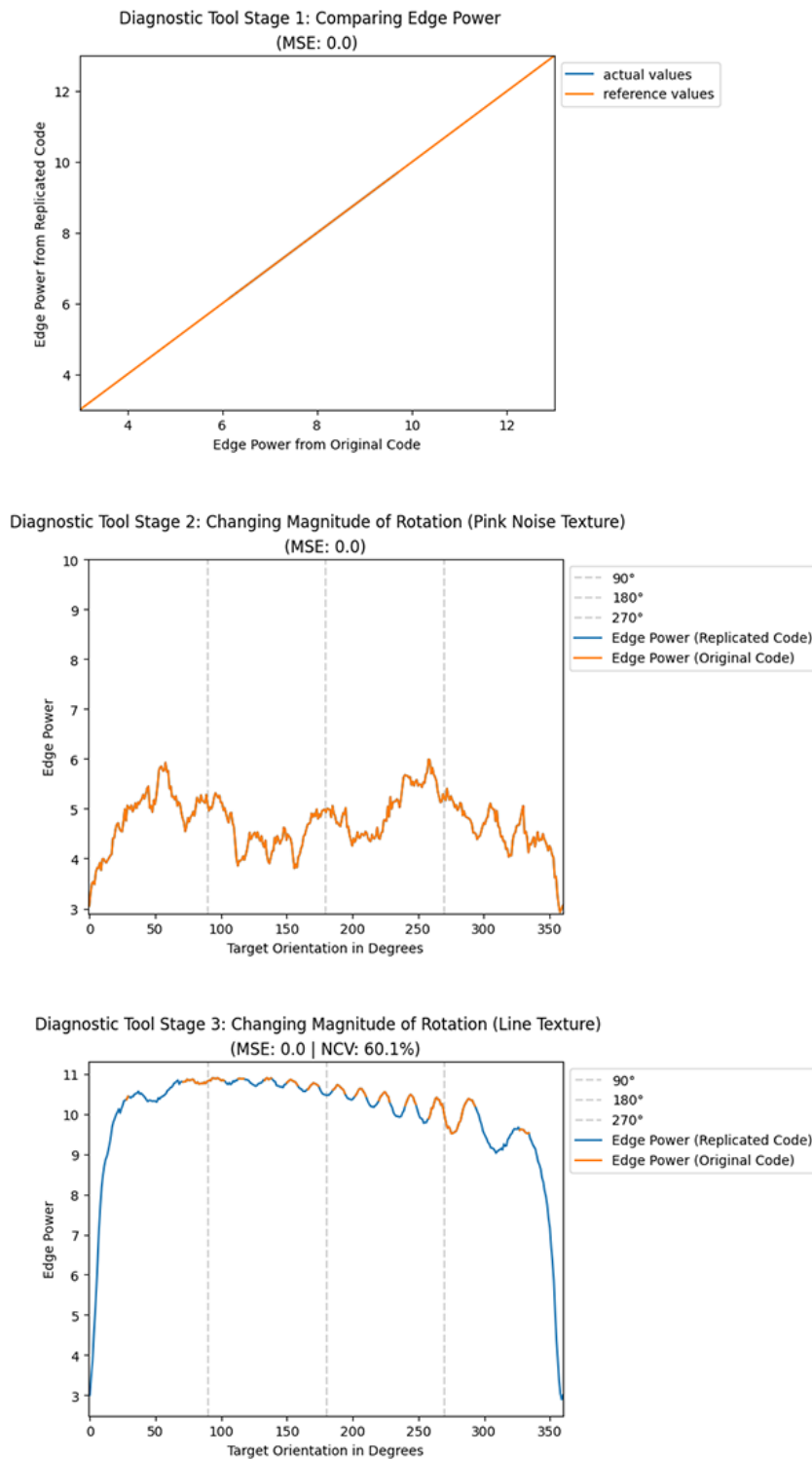


Figure 4.5: These diagrams show the results of all three diagnostic tool stages for the last execution. The results of the replication are matching the expected values.

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

lies exactly on the blue one. The MSE has a value of  $MSE_{5,1} = 0$  and  $MSE_{5,2} = 0$ . This occurs because the actual values from the replication match exactly the edge power from the original code. In the diagram of the last stage, where I rotate the target of a stimulus with a line texture, the edge power values are not only matching the original code but also filling the gaps where the original code computed an infinite or NaN value. Here we also have an MSE of  $MSE_{5,3} = 0$ . The NCV is still  $NCV = 60.1\%$ . But since the replication is returning an edge power for every stimulus, this NCV value is caused by the original code. According to the results shown in the diagram, the blue graph is still differentiable and continues the course of the orange function at each point where the gaps start and end.

### 4.4 Large-Scale Test Evaluation of the Replication

After the last diagnostic tool run, I evaluated my replication on a large data set, like described in Section 3.3. The results can be seen in the diagrams in Figure 4.6 and 4.7. Each row in Figure 4.6 and 4.7 shows the results of a particular background texture. Figure 4.6 shows the results of the binary stimuli in the first row, the narrowband stimuli in the second one and the one over f stimuli in the last row. Furthermore, the results of the pink noise and brown noise stimuli are shown in Figure 4.7. Each diagram shows the computed edge power on the y-axis for a stimulus  $x$ . The diagrams in the first column show the edge power values computed by the original algorithm, while the second column visualizes the results of the replication. The stimulus indices for each texture type represent the same stimulus in each diagram, meaning that when we have a stimulus  $i$ , the edge power value from the original algorithm can be found at  $f(x = i)$  in the left diagram. Analogously in the right diagrams, the edge power value from the replication can also be found at  $f(x = i)$ . As shown in the diagrams, the orange and blue curves match for all types of stimuli. This means if  $f_{matlab}(x)$  is a function that computes the edge power by the original CDA and  $f_{python}(x)$  is a function that computes the edge power by the replication of it, the edge power EP for a stimulus  $i$  is  $EP(i) = f_{matlab}(i) = f_{python}(i)$  for all stimuli which were generated in the large-scale test. To confirm it, I also computed the MSE between each edge power value from both implementations, and I got a mean squared error of  $MSE = 0$ . This means that based on the results of the large-scale test, the latest diagnostic tool run and the coverage tests, the replication is providing valid and reliable results.

#### 4.4 Large-Scale Test Evaluation of the Replication

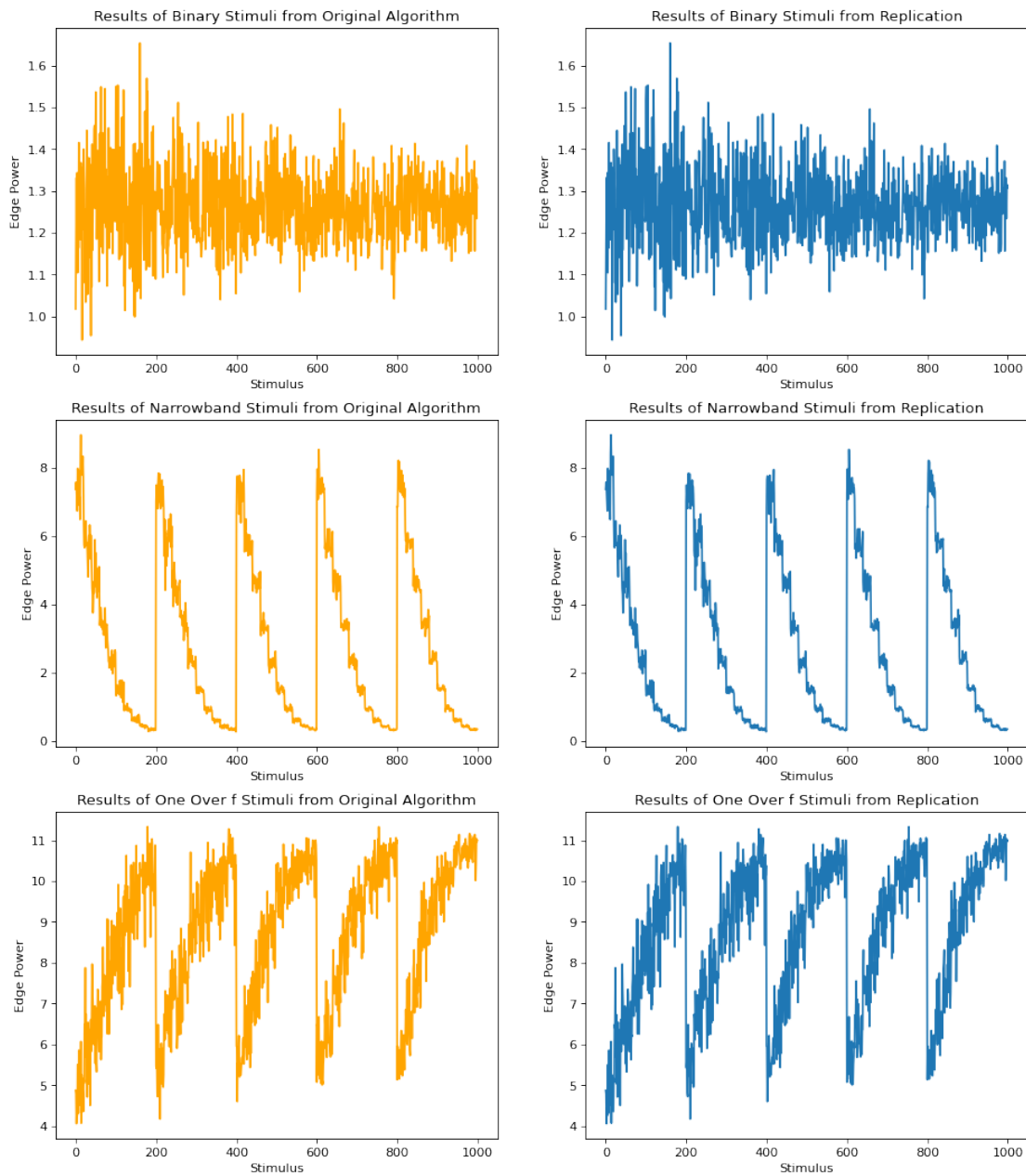


Figure 4.6: The diagrams visualize the first part of the results of the large-scale test. We can see the edge power values of the original algorithm on the left diagrams and the edge power values of the replication on the right ones. The first row shows the edge power values of the binary stimuli, while in the second row we can see the edge power values of the narrowband stimuli and in the last one the results of the one over  $f$  stimuli.

#### 4 Evaluating the Replicated Camouflage Detection Algorithm

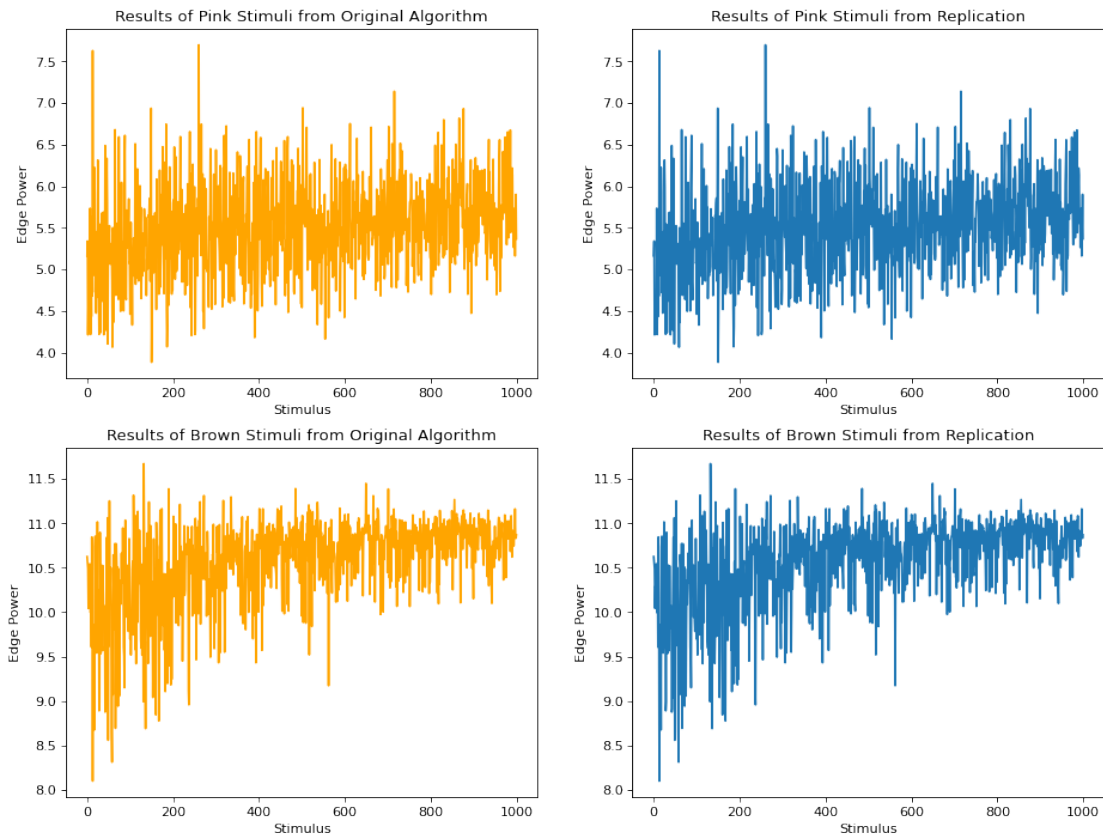


Figure 4.7: The diagrams visualize the second part of the results of the large-scale test. We can see the edge power values of the original algorithm on the left diagrams and the edge power values of the replication on the right ones. The first row shows the edge power values of the pink noise stimuli, while in the second row we can see the edge power values of the brown noise stimuli.



# 5 Conclusion and Outlook

## 5.1 Conclusion

Considering the goal of the thesis formulated in Section 1.3 and the evaluation of the replication in Section 4.3 and 3.3, we can see that replicating an algorithm from a scientific publication isn't straightforward and can be very challenging depending on the amount of information provided by the original scientific publication. But in the end, the goal of replicating the CDA was successful. The replicated CDA is returning the correct edge power values for various kinds of stimuli and even returns a finite solution when the original algorithm was returning infinite ones.

But the process of replicating scientific publications still does not have a generalized approach, which can be used as an orientation by other scientists. This is why the process of replicating the CDA was challenging in the beginning, specially when there were no tests or validation methods to validate the replicated code. But with the diagnostic tool together with the tests, the process of replicating the CDA was improved so that in the end I could achieve the goals of this thesis.

## 5.2 Outlook and Future Research Opportunities

### 5.2.1 Extension of the current Camouflage Detection Algorithm

The CDA, which was replicated in Python, is only capable of computing the edge power for stimuli with a grayscale background texture. To support colored background textures or images, the input can either be converted into a grayscale image or the algorithm has to be adjusted so that it can compute the gradients for stimuli with colored images.

Another possible option to extend the CDA is to adjust the algorithm so that for any kind of target shape, the edge power can be computed. To implement this, a new way of computing the normals needs to be considered, since the normal orientation for each

## 5 Conclusion and Outlook

pixel on the target edge in the current CDA is computed by the vector between the target center and pixel location. Regarding this, also the interpolation needs to be adjusted so that the interpolated edge vectors will be computed correctly.

### 5.2.2 Algorithm for Edge Detection

To compute the edge power, the CDA used the steerable filter to compute gradients. The length of the gradients indicates the intensity of an edge. This means that the steerable gradients are capable of detecting edges of any arbitrary object. Using the stimulus and the results of the steerable filter for a specific object as base for the training data and training a neural network on this test data, a machine learning solution can be created which is capable of detecting objects based on the data from the steerable filter.

## References

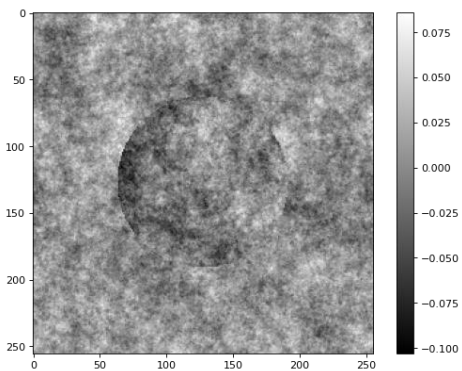
- Das, A. (2022). *Camouflage detection signal discrimination: Theory, methods experiments* (Doctoral dissertation). doi: 10.13140/RG.2.2.10585.80487
- Das, A. (2023). *abhranildas/camouflage-detection: Matlab package for the detection of spatial targets*. Retrieved from <https://github.com/abhranildas/camouflage-detection> (Online; accessed 07.06.2023)
- Das, A., & Geisler, W. (2022). *Camouflage detection: experiments and a principled theory* (Doctoral dissertation). doi: 10.13140/RG.2.2.32016.07683
- Fidler, F., & Wilcox, J. (2021). Reproducibility of Scientific Results. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy* (Summer 2021 ed.). Metaphysics Research Lab, Stanford University. <https://plato.stanford.edu/archives/sum2021/entries/scientific-reproducibility/>.
- Gonzalez, R. C., Woods, R. E., & Eddins, S. L. (2009). *Digital image processing using matlab* (2. ed.). Natick, Mass.: Gatesmark Publ.
- Kovac, J. (2007). How to Encourage and Publish Reproducible Research. , 1273–1276.
- Rougier, N. P., Hinsén, K., Alexandre, F., Arildsen, T., Barba, L. A., Benureau, F. C. Y., ... Zito, T. (2017). Sustainable computational science: the ReScience initiative. *PeerJ Computer Science*, 3, e142.
- Schmittwilken, L., Maertens, M., & Vincent, J. (2023). *stimupy — stimupy*. Retrieved from <https://stimupy.readthedocs.io/en/latest/index.html> (Online; accessed 28.04.2023)
- Stocker, T. C., & Steinke, I. (2017). *Statistik: Grundlagen und methodik*. De Gruyter, Oldenbourg.
- The MathWorks, Inc. (2023). *Find perimeter of objects in binary image - matlab bwperim - mathworks deutschland*. Retrieved from <https://de.mathworks.com/help/images/ref/bwperim.html#buohmjv-6> (Online; accessed 03.05.2023)
- The SciPy community. (2023). *scipy.ndimage.generic\_filter — scipy v1.10.1 manual*. Retrieved from <https://docs.scipy.org/doc/scipy/reference/generated/>

## References

- `scipy.ndimage.generic.filter.html` (Online; accessed 19.05.2023)
- Vandewalle, P., Kovačević, J., & Vetterli, M. (2019). Reproducible research in signal processing: What, why, and how. *IEEE Signal Processing Magazine*, 26(3).
- Zhi, X., & Jiang, S. (2017). Correlation and convolution image filtering application analysis. In *Proceedings of the 7th international conference on education, management, information and mechanical engineering (EMIM 2017)*. Atlantis Press. Retrieved 2023-04-15, from <http://www.atlantis-press.com/php/paper-details.php?id=25879248> doi: 10.2991/emim-17.2017.35

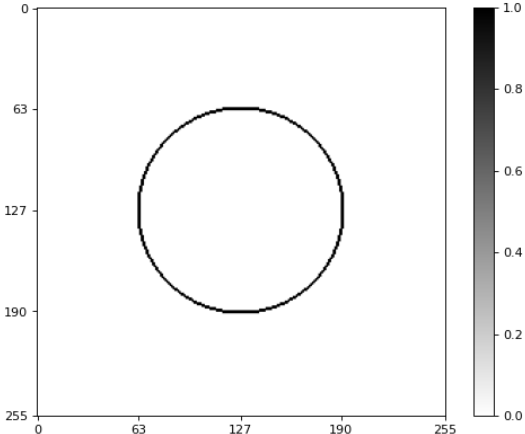
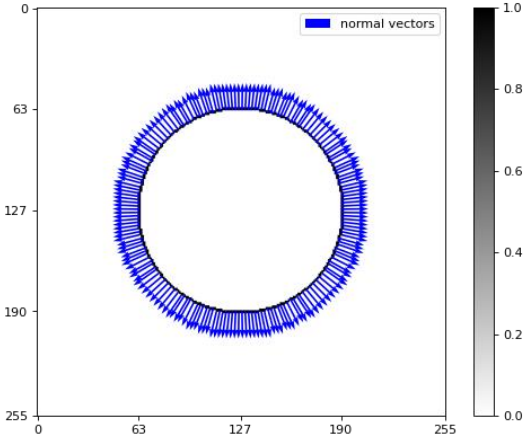
# Appendix

## Appendix A: Documentation of CDA

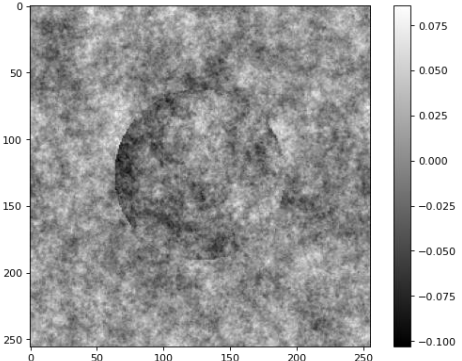
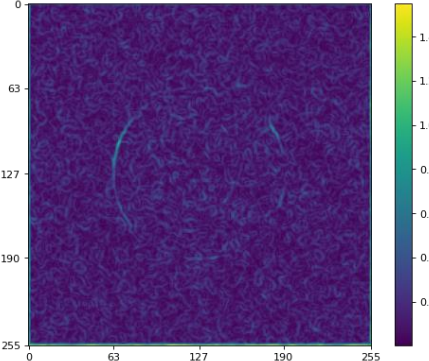
<b>Function Name</b>	edge_power	
<b>Description</b>	compute the edge power for a stimulus. the edge power is a numerical value that represents the visibility of the target in an image.	
<b>Input</b>	<ul style="list-style-type: none"> <li>• <b>stim:</b> numpy.ndarray two-dimensional stimulus.</li> <li>• <b>target_radius:</b> int radius of the target.</li> <li>• <b>n_edge:</b> int amount of interpolation steps.</li> <li>• <b>kernel_size:</b> numpy.array kernel size for steerable filter.</li> </ul>	
<b>Output</b>	<ul style="list-style-type: none"> <li>• <b>edge_power:</b> float edge power value for the given stimulus.</li> </ul>	
	Visualized Input	Visualized Output
	<ul style="list-style-type: none"> <li>• <b>stim:</b></li> </ul>  <ul style="list-style-type: none"> <li>• <b>target_radius:</b> 64</li> <li>• <b>n_edge:</b> 1000</li> <li>• <b>kernel_size:</b> [1, 3]</li> </ul>	<ul style="list-style-type: none"> <li>• <b>edge_power:</b> 9.3214</li> </ul>

Appendix

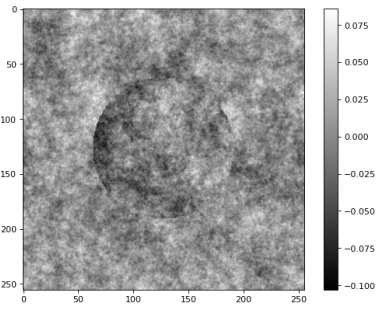
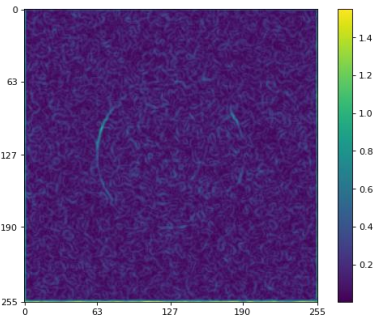
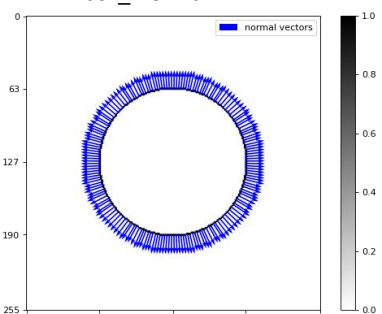
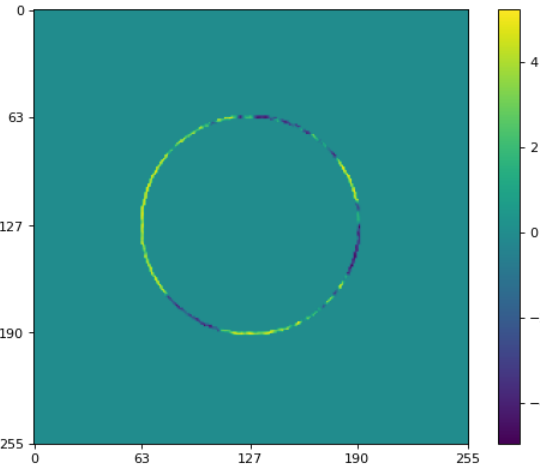
<b>Function Name</b>	theta_field	
<b>Description</b>	Computes an array which contains the polar angle (theta) of each point of a stimulus with respect to the target's center.	
<b>Input</b>	<ul style="list-style-type: none"> <li>• <b>stim_size</b>: int Size of stimulus, where height = width = stim_size.</li> <li>• <b>target_center</b>: np.ndarray center of the stimulus.</li> </ul>	
<b>Output</b>	<ul style="list-style-type: none"> <li>• <b>theta_field</b>: np.ndarray two-dimensional array containing polar angles with respect to a target center.</li> </ul>	
	Visualized Input	Visualized Output
	<ul style="list-style-type: none"> <li>• stim_size: 256</li> <li>• target_center: [127 127]</li> </ul>	

<b>Function Name</b>	circular_target_mask
<b>Description</b>	computes a binary mask which represents the border of circular target. The target is allocated in the center of the stimulus.
<b>Input</b>	<ul style="list-style-type: none"> <li>• <b>stim_size</b>: int Size of stimulus, where height = width = stim_size.</li> <li>• <b>target_radius</b>: int radius of circular target.</li> </ul>
<b>Output</b>	<ul style="list-style-type: none"> <li>• <b>mask_edge</b>: numpy.ndarray two-dimensional array containing 0 and 1 that represents the circular target.</li> <li>• <b>mask_normal</b>: numpy.ndarray two-dimensional array containing normal vectors for the boundary of the target.</li> </ul>
<b>Visualized Input</b>	<b>Visualized Output</b>
<ul style="list-style-type: none"> <li>• stim_size: 256</li> <li>• target_radius: 64</li> </ul>	<ul style="list-style-type: none"> <li>• <b>mask_edge</b>:   </li> <li>• <b>mask_normal</b>:   </li> </ul>

Appendix

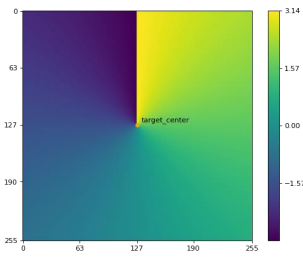
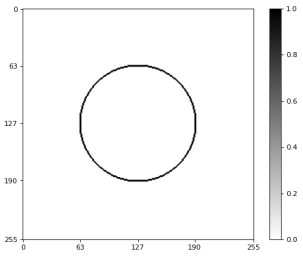
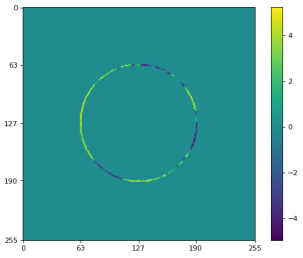
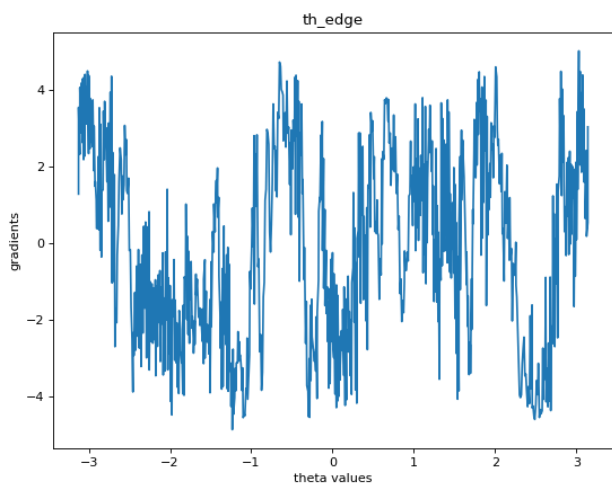
<b>Function Name</b>	steerable_gradient	
<b>Description</b>	Computes the steerable gradients for a given stimulus by applying the steerable filter with a provided kernel size.	
<b>Input</b>	<ul style="list-style-type: none"> <li>• <b>stim:</b> np.ndarray stimulus as two-dimensional numpy array</li> <li>• <b>kernel_size:</b> np.ndarray kernel_size for steerable gradient</li> </ul>	
<b>Output</b>	<ul style="list-style-type: none"> <li>• <b>stim_grad:</b> np.ndarray numpy array containing gradients for each point of the stimulus</li> </ul>	
	Visualized Input	Visualized Output
	<ul style="list-style-type: none"> <li>• <b>stim:</b></li> </ul>  <ul style="list-style-type: none"> <li>• <b>kernel_size:</b> [1, 3]</li> </ul>	<ul style="list-style-type: none"> <li>• <b>stim_grad:</b></li> </ul>  <p>normally, <b>stim_grad</b> is a two-dimensional array that contains gradients (1x2 vectors) for each point of the stimulus. To visualize it, the image above shows the length of each gradient.</p>

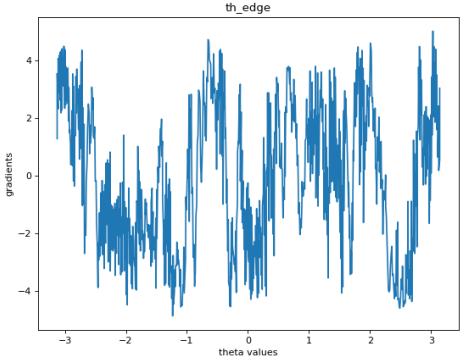
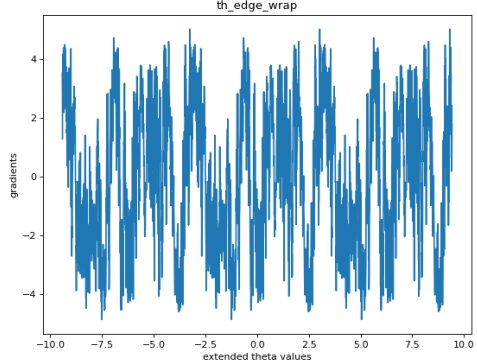


<b>Function Name</b>	normalize_gradients
<b>Description</b>	Normalize the steerable gradients of the stimulus and project them onto the mask normal.
<b>Input</b>	<ul style="list-style-type: none"> <li>• <b>stim:</b> np.ndarray two-dimensional stimulus.</li> <li>• <b>stim_grad:</b> np.ndarray steerable gradients for every point of stimulus.</li> <li>• <b>mask_normal:</b> np.ndarray mask normal vectors from stimulus.</li> </ul>
<b>Output</b>	<ul style="list-style-type: none"> <li>• <b>target_edge_field:</b> numpy.ndarray normalized normal vectors</li> </ul>
<div style="display: flex; justify-content: space-around;"> <span>Visualized Input</span> <span>Visualized Output</span> </div>	
<div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <ul style="list-style-type: none"> <li>• <b>stim:</b></li> </ul>  <ul style="list-style-type: none"> <li>• <b>stim_grad:</b></li> </ul>  <ul style="list-style-type: none"> <li>• <b>mask_normal:</b></li> </ul>  </div> <div style="width: 45%;"> <ul style="list-style-type: none"> <li>• <b>target_edge_field</b></li> </ul>  </div> </div>	

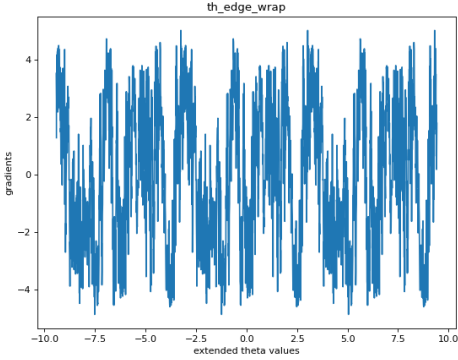
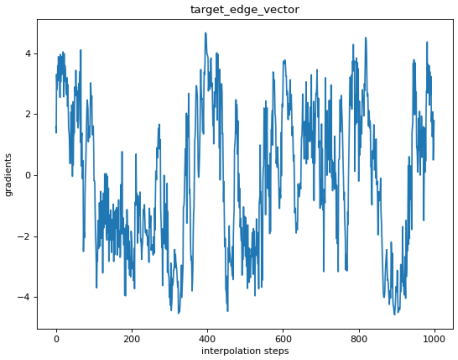
Appendix

<b>Function Name</b>	filter_and_sort_theta_edge
<b>Description</b>	filter out theta's and normal's that don't lie on on the edge of the target and sort the values ascending by theta.
<b>Input</b>	<ul style="list-style-type: none"> <li>• <b>theta_field</b>: np.ndarray polar angles for each point of stimulus.</li> <li>• <b>target_edge_field</b>: np.ndarray normalized gradients.</li> <li>• <b>mask_edge</b>: np.ndarray mask for the edge of the target.</li> </ul>
<b>Output</b>	<ul style="list-style-type: none"> <li>• <b>th_edge</b>: numpy.ndarray sorted array of thetas and gradients that lie on the edge of the target.</li> </ul>

Visualized Input	Visualized Output
<ul style="list-style-type: none"> <li>• <b>theta_field</b>:   </li> <li>• <b>mask_edge</b>:   </li> <li>• <b>target_edge_field</b>:   </li> </ul>	<p>graphical visualization:</p>  <p>excerpt of the corresponding array:  <code>[[ -3.1259, -3.1257, -3.1103 ...  3.1259,  3.1415,  3.1415]</code>  <code>[ 1.2840,  3.5388,  2.3194 ...  0.1833,  0.5443,  3.0325]]</code></p>

<b>Function Name</b>	wrap_and_create_unique_edge_matrix	
<b>Description</b>	<p>wraps th_edge on each side of theta. This means, it copies the theta value and their corresponding gradients and shift the theta value by <math>-2 * \pi</math> and <math>+2 * \pi</math>. This function then removes all duplicated thetas from a numpy array of shape (2,n), where n is the amount of gradients, while th_edge[0] contains the theta values and th_edge[1] their corresponding gradients.</p> <p>If there are duplicates, the mean of the gradients will be computed.</p>	
<b>Input</b>	<ul style="list-style-type: none"> <li>th_edge: np.ndarray 'table' that contains the theta values and their corresponding gradients.</li> </ul>	
<b>Output</b>	<ul style="list-style-type: none"> <li>th_edge_wrap: numpy.ndarray array that contains unique theta values with their corresponding mean gradients.</li> </ul>	
	<b>Visualized Input</b>	<b>Visualized Output</b>
	<p>graphical visualization:</p>  <p>excerpt of the corresponding array:  <pre>[[ -3.1259, -3.1257, -3.1103 ...  3.1259,  3.1415,  3.1415]  [ 1.2840,  3.5388,  2.3194 ...  0.1833,  0.5443,  3.0325]]</pre> </p>	<p>graphical visualization:</p>  <p>excerpt of the corresponding array:  <pre>[[ -9.4091, -9.4089, -9.3935 ...  9.4091,  9.4247]  [ 1.2840,  3.5388,  2.3194 ...  0.1833,  1.7884]]</pre> </p>

Appendix

<b>Function Name</b>	interpolate	
<b>Description</b>	interpolate the theta values between $-\pi$ and $\pi$ in $n\_edge$ steps to get the gradients of every point on the boundary of the circular target in, so that every point has the same distance to its neighbors.	
<b>Input</b>	<ul style="list-style-type: none"> <li>• <b>th_edge_wrap</b>: numpy.ndarray array that contains unique theta values with their corresponding mean gradients.</li> <li>• <b>n_edge</b> : int amount of interpolation steps</li> </ul>	
<b>Output</b>	<ul style="list-style-type: none"> <li>• <b>target_edge_vector</b>: numpy.ndarray array containg edge vectors</li> </ul>	
	Visualized Input	Visualized Output
	graphical visualization:	
	 <p>excerpt of the corresponding array:</p> <pre>[[ -9.4091 -9.4089 -9.3935 ...  9.4091  9.4247]  [ 1.2840  3.5388  2.3194 ...  0.1833  1.7884]]</pre>	 <p>excerpt of the corresponding array:</p> <pre>[ 1.5855  1.3827  3.3025 ...  0.49742  1.1429  1.7884]</pre>