# Examining the role of simulated imagery in the evaluation and training of computer vision systems

Stefan Marincas

A thesis submitted for the degree of Doctor of Philosophy

School of Computer Science and Electronic Engineering
University of Essex

Date of Submission: March 2026

# List of Figures

# List of Tables

# Contents

# Abstract

This thesis addresses the significant limitations of "Scenario Evaluation" methodologies in Visual Simultaneous Localization and Mapping (SLAM) systems by proposing a scalable and generalisable "Technology Evaluation" framework driven by high fidelity synthetic imagery. Traditionally, the evaluation of computer vision systems has relied upon expensive, environment-specific real-world data collection, making it inherently difficult to predict system performance accurately under fluctuating or previously unseen operational conditions. To overcome these considerable constraints, this research thoroughly investigates the viability of using synthetic imagery as a robust, repeatable, and highly predictive substitute for real-world datasets. The core methodological contribution of this work is the systematic construction of "Digital Twins" that meticulously replicate not only the geometric structures of real-world environments but also the crucial optical artifacts that fundamentally influence computer vision performance. Specifically, this research focuses on two critical imaging phenomena: Signal-to-Noise Ratio (SNR) degradation and optical lens blur. Both parameters are carefully modelled and precisely calibrated within the synthetic environment to faithfully mirror the characteristics observed in real captured imagery. Through a rigorous and systematic comparative analysis of fundamental computer vision operators, including corner detection algorithms and Oriented FAST and Rotated BRIEF (ORB) feature matching, the experimental results demonstrate a consistently strong and statistically meaningful correlation between real-world and virtual environment performance outcomes. These findings provide compelling empirical evidence that accurately constructed and carefully matched synthetic datasets can reproduce the pattern of real-world performance degradation observed in the sensors tested, and can predict the failure boundaries of the operators examined. By establishing this correlation at the operator level, this work provides evidence that engineering simulation offers

a viable and cost-effective complement to physical data collection for the evaluation of computer vision operators. Extension of the approach to complete Visual SLAM systems is identified as the principal direction for future work. This contribution opens significant opportunities for more efficient, flexible, and comprehensive evaluation of computer vision technologies across diverse and demanding operational scenarios.

# Acknowledgements

My first and greatest debt is to my supervisor, Dr Adrian Clark, whose influence runs through every chapter of this thesis. He had the patience to let a broad idea about simulated imagery become a specific and answerable question, and the judgement to know when to leave me to it and when to intervene. He was generous with far more than his time. Much of the practical apparatus on which this work rests, including the idea to write software written expressly to get the early noise experiments moving, came from him. I have learned as much from his instinct for what constitutes an honest measurement as from anything written down, and I hope some of that instinct is visible in the pages that follow.

I am grateful in equal measure to my second supervisor, Dr Manoj Thakur, whose questions had a habit of arriving precisely where my reasoning was weakest. Several of the arguments in Chapters 4 and 5 exist in their present form because he declined to accept an earlier and looser version of them, and the thesis is a good deal more careful for it.

I would like to thank the School of Computer Science and Electronic Engineering at the University of Essex for supporting this research and for providing the environment in which it was possible. I am particularly grateful for access to the motion capture facility, without which the ground truth underpinning Chapter 3 could not have been established, and to the technical and administrative staff who kept the laboratories running and who accommodated a great many requests for rooms to be emptied, equipment to be borrowed and deadlines to be interpreted generously. Their work is rarely visible in a thesis and this one would not exist without it.

I owe thanks also to fellow researchers and colleagues within the School, for conversations in corridors and over coffee that resolved problems I had spent days circling alone. A research group is a quiet form of infrastructure, and I was fortunate in mine.

Finally, and most importantly, I thank my family, whose support over the course of this work has been unwavering and largely unremarked upon. They tolerated the long evenings, the rooms full of photographic equipment and the periods in which I was present in body only, and they did so without complaint. This thesis is as much a product of their patience as of my own effort, and it is dedicated to them.

# Chapter 1: Introduction

## 1.1 Background

Computer Vision is a vast field of science where research is aimed at creating Vision Systems that can automatically operate with human-like abilities. This goal has not yet been fully reached, but significant progress is being made in this direction. Preceding years have seen a booming emergence of Vision Systems designed in a wide range of applications where tailored solutions have also been successfully commercialized. This situation has made the developers of these successes experts in their field, but not necessarily as a consequence of the general research published in Computer Vision literature.

This meant that their implementations and algorithms would often only fit their specific situation. This is known in the field as a "Scenario Application" of Computer Vision principles. Such systems would be deemed as performant, but only for the domain and activity for which they were designed. Researchers that wanted to extrapolate the systems' functionalities for other purposes would often be met by frustration, as the vision modules would not work due to a lack of generalization being implemented.

Computer Vision research has further matured with more Vision Systems being developed with the aim of covering more specific scenarios and making sure that those particular systems would perform well under the specific conditions required for each particular case the system was developed for. Later on, evaluation methods have emerged as well, and the performance of Vision Systems could now more easily be measured. Most of these Vision Systems would work very well for their particular tasks, but then when they were

subjected to different environments than those that they were designed for, their performance was found to be inadequate.

To understand why this inadequacy persists, it is necessary to distinguish between two fundamental types of evaluation. As defined by Thacker et al., the current dominant paradigm is "Scenario Evaluation," where a system is tested against a specific, often unrepeatable, set of real-world conditions. While valuable for final validation, this approach fails to isolate the underlying "Technology Evaluation"—the characterization of how a specific algorithm responds to fundamental variables such as lighting intensity, sensor noise, or optical distortion. Without this deeper understanding, adapting a system from one scenario (e.g., an indoor factory) to another (e.g., an outdoor street) becomes a process of trial-and-error rather than engineering design.

## 1.2 Aims, Research Questions and Objectives

Up to a certain point, Vision Systems could only be evaluated within the scenario for which they were designed, thus Scenario Evaluation only dealing with adjusting specific functionality and variables of the system so that best performance could be obtained. But what if the Vision System would be used for a different application? To answer this question, an increasing need has been recently emerging: that of a Vision System not only being able to work well in a specific well-defined scenario, but that it could perhaps transcend this limitation and also be evaluated as performant and robust in multiple scenarios and with various modifications of these scenarios as well.

This approach takes that focus away from making the best Vision System work for a specific task alone, and refocuses the attention on the characteristics of technology as the observed environment conditions are changed. These characteristics can be anything from

sensor noise, contrast change, sensor upgrade, field of view variation, optic distortion etc. The goal is to be able to make the Visual System impervious to such changes regardless of changes happening in the environment being monitored.

The industry appears to indeed move towards Technology Evaluation as this would yield more robust Vision Systems that could perhaps be able to work in multiple scenarios and with various parameter modifications without a significant loss in performance. There has been several industry attempts to move in this direction. For example Waymo is doing this at the moment in training its vision system for a self-driving car. By simulating millions of miles of driving data, they attempt to cover "edge cases", rare events that might happen only once in a lifetime of real-world driving, thereby making the system more robust against the unknown.[3]

This research comes with a similar view in mind: that of showing that evaluating technology with the purpose of producing a more robust Vision System being able to deal successfully with more change in its variables, can be indeed helpful despite this approach being viewed with less confidence at the moment. Several authors have also been of the opinion that a "good technology evaluation can inform Scenario Evaluation".[1][2]

If more research is being carried out in this direction it could prove useful especially to people working with Convolutional Neural Networks (CNNs) and Visual SLAM (Simultaneous Localization and Mapping). It has already proven difficult for new images to be obtained from real life scenarios for a more robust training of Vision Systems due to the logistical costs and lack of ground truth data. If the Technology Evaluation approach proves fruitful, CNN and SLAM training could rely on a new source from which to derive its training images such as virtually generated data.

This thesis therefore investigates whether the optical conditions under which a vision system operates can be replicated with sufficient accuracy in simulation that the system's response to those conditions may be predicted rather than merely observed. The investigation is organised around five research questions.

1. **RQ_1.** Can a ray-traced virtual environment reproduce the geometry of a physical environment with sufficient fidelity that an identical photogrammetric pipeline, applied to real and synthetic imagery, yields reconstructions that agree within the measurement tolerance of the pipeline itself? To establish a rigorous ground truth by tracking a real camera's trajectory using an external Vicon motion capture system, allowing for a direct 1-to-1 comparison between real and synthetic data streams.
2. **RQ_2.** Which acquisition parameters govern that fidelity, and where do the practical limits of physical data collection lie?
3. **RQ_3.** Can the operational envelope of a low-level vision operator be characterised in simulation, as a bounded region of a stress space defined by sensor noise and scene contrast, in the manner of a component datasheet?
4. **RQ_4.** Does a characterisation derived entirely in simulation predict the behaviour of the same operator on imagery captured by physical cameras?
5. **RQ_5.** Which optical properties, beyond signal-to-noise ratio, must be replicated in the synthetic imagery for that predictive correspondence to hold?

These questions give rise to three testable hypotheses:

**H1.** A virtual environment constructed from measured camera poses and matched acquisition parameters will yield a photogrammetric reconstruction whose cloud-to-cloud distance from the real reconstruction falls predominantly within the lowest error bracket of the comparison.

**H2.** The failure boundary of a corner detection operator is governed principally by signal-to-noise ratio, with contrast acting as a secondary constraint that defines a region in which detection is not merely unreliable but undefined.

**H3.** Matching signal-to-noise ratio alone is insufficient to produce agreement between real and synthetic operator performance; the optical blur characteristic of the physical lens must be matched additionally, and the noise itself must be blurred to reproduce the spatial correlation present in sensor noise.

Accordingly, the specific objectives of this thesis are:

1. To construct a high-fidelity simulation, referred to throughout as a "Digital Twin", of a physical test environment using ray-tracing techniques (POV-Ray), ensuring that the virtual camera matches the intrinsic parameters and pose of the real sensor. *(RQ_1)*
2. To establish a rigorous ground truth by tracking the real camera's trajectory using an external Vicon motion capture system, permitting a direct one-to-one comparison between real and synthetic data streams. *(RQ_1)*
3. To determine empirically the acquisition parameters that govern photogrammetric fidelity, specifically sampling density, angular diversity, sensor resolution and surface texture, and to identify the point beyond which additional physical data yields no further benefit. *(RQ_2)*
4. To characterise the operational envelope of a fundamental vision operator, corner detection, across a controlled two-dimensional stress space of injected noise and radiometric contrast, and to express the result as a Safe Operating Area. *(RQ_3)*
5. To calibrate synthetic imagery against measurements taken from specific physical cameras, matching signal-to-noise ratio and blur gradient, and to verify whether the

failure boundaries predicted in simulation are reproduced in real imagery captured under those matched conditions. *(RQ_4 and RQ_5)*

The investigation was originally conceived as terminating in a comparison of ORB-SLAM performance on real and synthetic image sequences. That comparison was not undertaken, and the reasons are set out in Chapter 4.1. In summary, validating a complete SLAM pipeline before validating the operators from which it is composed risks agreement between real and synthetic results arising from compensating errors within a system complex enough to absorb them, rather than from genuine simulation fidelity. The decision was therefore taken to establish correspondence at the operator level first. Extension of the framework to a full SLAM system remains the principal item of future work, and the claims made in this thesis are correspondingly confined to the operators actually tested.

The principal contribution of this thesis is a four-stage framework for the Technology Evaluation of computer vision operators, comprising the construction of a geometrically validated digital twin, the characterisation of an operator's response across a controlled stress space, the optical calibration of synthetic imagery against a measured physical sensor, and the verification of predicted failure boundaries against real imagery captured under matched conditions. The first two stages have precedents in the literature. The claim to novelty rests on the third in combination with the fourth: the calibration of simulated optical degradation against quantities measured from a specific physical camera, followed by verification that the operator's failure boundary occurs at the same point in both domains.

## 1.3 Overview of the Thesis

This thesis is structured to guide the reader through the process of building, verifying, and utilizing a synthetic evaluation environment.

- **Chapter 2** provides an overview of Visual Navigation, SLAM, and the history of simulation in engineering.
- **Chapter 3** details the methodology used to capture ground truth data and construct the virtual environment.
- **Chapter 4** presents the baseline comparison results, establishing the validity of the simulation.
- **Chapter 5** demonstrates the predictive power of the simulation through stress-testing experiments.
- **Chapter 6** concludes the work and discusses future implications for autonomous system design.

# Chapter 2: Literature Review

## 2.1 Introduction to Visual Navigation

The autonomous visual navigation of mobile robots has been extensively studied over the past three decades [36]. The goal is to use the surrounding visual information as a support, to automatically guide a mobile robot through a suitable path between two locations in an environment. Such a capability significantly increases robot autonomy and correspondingly reduces the human supervision required in complex real-world tasks, a motivation that holds across application domains as varied as assistive navigation for the visually impaired [1] and industrial mobile robotics [9]. A large number of approaches have been proposed over this period using various high-level visual strategies [46], and the field has been surveyed at intervals [36][44]. What has been examined far less systematically is the *evaluation* of these approaches. As Tian observes in a quantitative treatment of the problem [2], the proliferation of navigation methods has not been matched by a corresponding development of methods for characterising when and why they fail, and it is this asymmetry between capability and characterisation that the present work addresses.

Many of these approaches work well in small sized, static and indoor environments. However, the task of autonomous visual navigation in large scale, dynamic or outdoor environments is still considered as an open problem. In practice, the quality and scale of the environment map causes the computational complexity to grow exponentially. To achieve a sufficiently high efficiency in real-world implementations, a trade-off is inevitable between the processing burden and representational distinctiveness or map scale. The robustness to the

localization error therefore needs to be considered in the subsequent navigation design, to ensure a reliably working robotic system.[1][2]

The evaluation of Visual SLAM and computer vision systems has relied heavily on standardised real-world datasets, principally the KITTI Vision Benchmark Suite [139] and the TUM RGB-D benchmark [140]. The contribution of these resources is not in dispute: both supply accurate ground-truth trajectories, both established comparable evaluation protocols where previously there were none, and both remain indispensable for baseline comparison within their captured environments. The limitation is structural rather than a deficiency of execution. Because the environmental variables of such datasets, including ambient illumination and scene texture, together with the parameters of the capturing sensor, are fixed irrevocably at the moment of recording, no subsequent researcher can isolate and manipulate an individual variable. A dataset can therefore establish *that* a system performs well under the conditions it happens to contain, but it cannot establish *at what point* that performance ceases, because the conditions cannot be swept. Consequently, a system that performs optimally on KITTI offers limited predictive power regarding its behaviour under differing optical conditions such as sudden SNR degradation or altered lens blur [139]. This is the distinction between scenario and technology evaluation formalised by Thacker et al. [5], and it identifies a persistent gap: the need for an evaluation framework in which individual variables can be manipulated independently, so that technological resilience may be isolated from scenario-specific overfitting.

## 2.2 The Engineering of Evaluation

The proposition advanced in this thesis is that simulated imagery can substitute for the large volumes of real-world data that a vision system currently requires in order to demonstrate its performance. The argument is an engineering one. Constructing a vision system directly in the physical world, for a particular situation and with a set of sensors whose characteristics have not themselves been characterised, produces a system whose performance is established only for the conditions under which it happened to be built. The alternative, familiar throughout established engineering disciplines, is to characterise components and conditions in simulation first, and to treat physical construction as the confirmation of a predicted result rather than as the means of discovering it. The case for this approach within computer vision specifically was set out by Thacker et al. in their treatment of performance characterisation [5], and the practical difficulty it responds to, that of obtaining sufficient real-world data with reliable ground truth, is documented across the navigation literature [2][44].

The contrast may be illustrated by the development of consumer products through iterative physical prototyping, of which Dyson's cyclonic vacuum system is a frequently cited example: a large number of successive physical configurations were built and discarded before a satisfactory design was reached. Much current vision system development proceeds comparably, through empirical adjustment of a physical implementation until acceptable performance is obtained for the task at hand [5]. The analogy is offered as illustration rather than as evidence, and its limits should be acknowledged: mechanical prototyping and algorithmic development differ in the cost of an iteration and in the ease with which a failed configuration can be diagnosed. What the comparison does capture accurately is the epistemic position of the developer at the end of the process. A system arrived at by iteration is known to work; it is not thereby known *why* it works, nor under what perturbation it would cease to.

This distinction matters because it determines whether performance can be predicted or only observed. A development process grounded in simulation permits the response of a system to be mapped against its governing variables before physical commitment, and this is the approach adopted here. Its application to camera systems specifically is not without precedent. Hinkenjann et al. simulated camera errors and examined their effect on basic robotic vision algorithms [12], and Asanuma et al. developed an environment simulator for autonomous mobile robots that explicitly accounted for camera characteristics [13]. These works are the closest antecedents of the present research and establish its feasibility. Their scope was nonetheless limited in a way that this thesis seeks to extend: neither calibrated the simulated optical degradation against measurements taken from a specific physical sensor, and neither established a failure boundary that was subsequently verified against real imagery captured under the matching conditions.

This is the approach adopted in the present research. The investigation begins by testing a vision system against theoretical models (simulation) and then comparing these models with real world models to see what differences exist and how to improve them. At the end of this work, the hopes are that a theoretical model would have been perfected enough to show that applying this knowledge to real world Vision Systems, the vision system would yield robust data despite multiple changes of scenario, or deviations from the standard mode of operation of its sensors.[10][13]

A distinct shortcoming in the current literature regarding synthetic data evaluation is the tendency to evaluate complex, end to end systems without first validating the fidelity of the synthetic data at the operator level. Many studies implement full Visual SLAM pipelines within simulated environments to assess macroscopic metrics like trajectory error, bypassing the verification of low-level feature extraction. If the foundational operators (such as corner

detectors or ORB feature matchers) do not behave identically on the synthetic data as they do on real data, the macroscopic system evaluation becomes inherently flawed. The literature lacks a rigorous, bottom-up validation framework that first calibrates the synthetic imagery to match the exact physical and optical imperfections of the real-world sensor. By establishing a methodology that meticulously matches blur gradients and SNR prior to testing corner detection and feature matching, a more deterministic and reliable prediction of real-world SLAM performance can be achieved. [14][20]

The motivation for the system proposed in this thesis follows directly from this observation. If the validity of a synthetic dataset is to be established at all, it must be established at the level at which the imagery is actually consumed, which is the level of the individual operator. A vision system is not a monolith; it is a composition of operators, each of which transforms pixel intensities according to its own assumptions, and each of which possesses its own tolerance to violation of those assumptions. Demonstrating that a complete pipeline produces comparable output on real and synthetic data is a weaker result than it appears, because a pipeline of sufficient complexity can absorb compensating errors and produce agreement for the wrong reasons. Demonstrating that a single operator exhibits the same failure boundary on both, under optical conditions matched by measurement rather than by assumption, is a stronger and more transferable result. It is this operator-level correspondence that the present work sets out to establish, and the fidelity requirements it imposes on the simulation are what the following chapters determine.

## 2.3 Ray Tracing with POV-Ray

To achieve this theoretical model, a method is required by which every pixel of the generated image is controlled. Rendering software such as **POV-Ray** (Persistence of Vision Raytracer) creates an image using a technique known as ray tracing. Broadly, this means it takes a description of geometry (which includes surface materials and light sources), a model of a virtual camera, and a projection plane made up of the pixels in the final image. It then creates what the camera would "see" by shooting rays from the camera through each pixel and analysing how they are affected by the geometry, materials, and lights.[15][16]

To transition from rigid "Scenario Evaluation" to a robust "Technology Evaluation" framework, the synthetic data utilized must transcend mere geometric resemblance, it must achieve optical equivalence. This chapter details the mathematical and physical principles of ray tracing, the computational rendering technique employed to construct the "Digital Twin." Furthermore, it establishes the technical rationale for selecting POV-Ray (Persistence of Vision Raytracer) as the foundational engine for generating the synthetic datasets required to validate Visual SLAM systems and low-level computer vision operators.[19]

### 2.3.1 The physics of Ray Tracing

In the physical world, photons are emitted from light sources, bounce off surfaces, and a minute fraction ultimately penetrate a camera lens to strike a sensor. Simulating this forward propagation (forward ray tracing) is computationally difficult due to the sheer volume of photons that never reach the camera. Consequently, modern optical simulation relies on ray tracing (or eye tracing). In this paradigm, virtual "rays" are cast backward from the camera's focal point, through the pixels of a virtual image plane, and out into the simulated 3D scene. When a ray intersects an object, the algorithm calculates the colour and intensity of that specific

point by casting secondary rays (shadow rays, reflection rays, and refraction rays) toward the light sources. [18][20]

### 2.3.2 Core Mathematical Formulations

The fidelity of the synthetic data relies entirely on the deterministic mathematical models governing these rays and their intersections with the scene's geometry. The fundamental building block of ray tracing is the mathematical representation of a ray in 3D space. A ray is defined parametrically by an origin point and a normalized direction vector:

$$P(t) = O + tD \qquad \textit{(Equation 1 )}$$

Where:

- P(t) is a point along the ray in 3D space.
- O is the origin of the ray (e.g., the camera's focal point).
- D is the normalized direction vector of the ray.
- t is a scalar parameter representing the distance along the ray from the origin (t > 0).

To determine what the camera "sees," the system must solve for the intersection between the ray equation and the mathematical equations defining the objects in the scene. For example, a sphere is defined by its centre point C and radius r.

$$|\vec{P}(t) - \vec{C}|^2 = r^2 \qquad \textit{(Equation 2)}$$

Substituting the ray equation into the sphere equation yields:

$$(\vec{O} + t\vec{D} - \vec{C}) \cdot (\vec{O} + t\vec{D} - \vec{C}) = r^2 \qquad \textit{(Equation 3)}$$

Expanding this dot product results in a quadratic equation in terms of t:

$$(\vec{D} \cdot \vec{D})t^2 + 2\vec{D} \cdot (\vec{O} - \vec{C})t + (\vec{O} - \vec{C}) \cdot (\vec{O} - \vec{C}) - r^2 = 0 \qquad \textit{(Equation 4)}$$

where the symbols are defined as follows: **O** is the ray origin, taken as the camera's centre of projection; **D** is the ray direction, constrained to unit length; **C** is the centre of the sphere; *r* is its radius; and *t* is a scalar parameter denoting distance along the ray from its origin, with only $t > 0$ corresponding to points in front of the camera.

Four assumptions underlie this formulation, and each has a consequence for the fidelity of the resulting imagery. First, geometric optics is assumed: light propagates along straight lines and the wave phenomena of diffraction and interference are not represented, so the diffraction-limited softening present in any physical optical system is absent from the render. Second, rays are treated as having zero width, so a rendered edge is exact to the precision of the arithmetic and is not band-limited by a point spread function. Third, the camera is an idealised pinhole unless a finite aperture is declared explicitly, so there is no depth of field by default. Fourth, surfaces are opaque and analytically defined, and the scene contains no participating medium.

These assumptions connect directly to the implementation choices made in this thesis. Because POV-Ray solves the equations analytically for each Constructive Solid Geometry primitive rather than intersecting rays with a triangulated approximation, the simulated

geometry contains no tessellation artefacts that a corner detector might register as spurious features. This is the property that makes the renderer suitable for the operator characterisation of Chapter 4. The same assumptions, however, mean that a raw render is optically *sharper* than any physical capture can be. The blur and noise applied in §4.2.1 are therefore not corrections to a deficient renderer but the deliberate reintroduction of physical effects that the geometric-optics model excludes by construction. The smallest positive root of Eq. 15 additionally furnishes the depth of each pixel, and is the quantity from which secondary shadow and reflection rays are cast.

By solving for t using the quadratic formula from Equation 4, the renderer determines exactly where, and if, the ray intersects the sphere. The smallest positive root t represents the closest visible intersection point. An intuitive way to understand this equation is that we start the ray at the origin (O) and "advance" along the direction of the ray ($\vec{D}$) by some amount (t). It's easy to see that this includes all the points along the ray. Figure 1 shows our equation in action.[17][20]

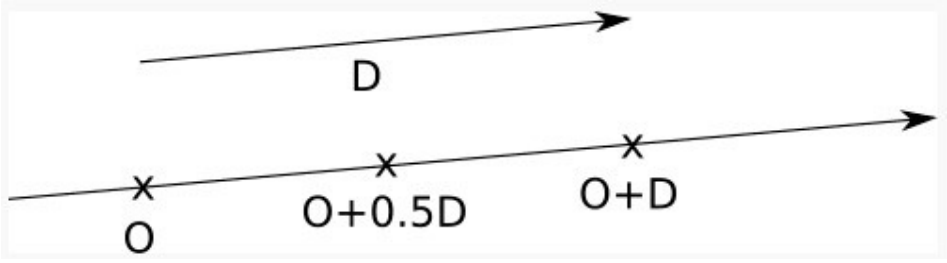


*Figure 1- Points along the ray P(t) = O + tD for t = 0.5 and t = 1.0. Each value of the scalar t yields a distinct point along the ray, which is the parameterisation solved for at intersection in Eq. 4*

Figure 1 shows the points along the ray that corresponds to t=0.5 and t=1.0. Every value of t yields a different point along the ray.

With the help of POV-Ray, virtual scenes can be described with the help of a script language. The shape and appearance of objects within a 3D world can be described, driving the ray-tracing software to produce photo-realistic images of the scene. An important feature of POV-Ray is that it supports a variety of types of light sources and camera models (including fish-eye and pinhole), something that is likely to be important for my research work in replicating the specific optical characteristics of physical cameras. Unlike rasterization engines used in video games (which approximate light), ray tracing simulates the physical behaviour of light transport, making it an ideal candidate for "Technology Evaluation" where lighting artifacts must be accurate. [21][22]

### 2.3.3 The Necessity of POV-Ray for Technology Evaluation

While modern game engines (e.g., Unreal Engine, Unity) excel at generating visually impressive synthetic environments rapidly, they are fundamentally unsuited for the precise "Technology Evaluation" required in this research. The justification for utilizing POV-Ray rests on three critical pillars:

**Algorithmic Determinism vs. Rasterization:** Game engines primarily use *rasterization*, a technique that projects 3D triangles onto a 2D screen and relies heavily on post-processing hacks (like screen-space reflections or artificial depth-of-field filters) to fake optical realism. POV-Ray is a pure ray tracer. It calculates light transport based strictly on the physics and mathematics outlined above. For validating low-level vision operators like corner

detectors, "faked" pixel data introduces unpredictable algorithmic noise. POV-Ray guarantees that every pixel's value is the direct result of a calculated physical interaction. [23]

**Precise Control Over Optical Imperfections:** As established in previous chapters, validating synthetic imagery requires meticulous matching of lens blur and Signal-to-Noise Ratio (SNR). POV-Ray allows for the mathematical definition of camera apertures, focal blur, and localized noise artifacts at the rendering level. This capability is paramount for creating the "optical twin" necessary to test the failure boundaries of Visual SLAM systems. [24][25]

**Mathematical Geometry (CSG):** Unlike rendering engines that rely on polygonal meshes (which are approximations of curves), POV-Ray utilizes Constructive Solid Geometry (CSG). Objects are defined by pure mathematical primitives (spheres, cylinders, planes) combined via Boolean operations (union, intersection, difference). This ensures infinite precision in the simulated environment, eliminating jagged edges or mesh artifacts that could generate spurious features and artificially skew the corner detection results. [26][30]

| Engine | Class | Advantage | Reason not selected |
|---|---|---|---|
| **POV-Ray** | CSG ray tracer | Selected. Scene defined analytically through Constructive Solid Geometry, so geometry is exact rather than tessellated and cannot introduce spurious edge features; scriptable text scene description permits camera poses to be driven directly from Vicon output without an asset pipeline; deterministic and long established | — |
| Unreal / Unity | Real-time rasterisation | High throughput; mature ecosystems; underlie CARLA [132] and Unity Perception [133] | Optical realism is approximated in post-processing rather than computed, which is inadmissible where pixel values are the measured quantity |
| Blender / Cycles | Production path tracer | Physically based; used by Kubric [122] and Infinigen [123] | Mesh-based, so curved geometry is tessellated; scene construction assumes an interactive workflow rather than parametric scripting |
| PBRT | Research physically based renderer | Spectral rendering and realistic lens system models, including ray-transfer formulations of physical lenses [137] | See qualification below |

| Engine | Class | Advantage | Reason not selected |
|---|---|---|---|
| Mitsuba | Research renderer, differentiable | Physically based, with differentiable rendering permitting parameters to be optimised against a target image | See qualification below |

*Table 1 - Rendering engines considered*

A qualification is required in respect of PBRT and Mitsuba, which are the physically-based renderers most commonly employed in optical simulation research. Both offer capabilities POV-Ray lacks, in particular the ability to model a physical lens system within the renderer itself and so to produce optically correct blur as a consequence of simulated optics rather than as a post-process. The approach adopted here instead applies blur and noise as calibrated post-processing operations, matched to measurements taken from the physical camera. This is defensible on its own terms, since the calibration is against measured physical quantities and the resulting correspondence is verified empirically in Chapter 4.5, and it has the practical advantage of requiring no knowledge of the proprietary lens design. It is nonetheless a weaker construction than in-renderer optical modelling, because it treats blur as a scalar applied uniformly rather than as a spatially varying field determined by the optics. Implementing the calibration within a renderer supporting explicit lens models is identified in Chapter 6.4 as a direction for further work.

### 2.3.4 Conclusion

The generation of synthetic data capable of substituting real-world data for SLAM evaluation cannot rely on visual approximations. It requires a rigorous, mathematically grounded approach to light transport. By utilizing the parametric ray equations, exact intersection calculations, and physical laws of reflection and refraction, backward ray tracing provides a physically accurate model of light. POV-Ray is strictly required for this thesis because its pure mathematical rendering pipeline allows for the exacting control over lens physics, blur gradients, and geometric precision necessary to construct a highly reliable, predictive Digital Twin. [19][23][32]

## 2.4 Structure from Motion (SfM)

Structure from Motion (SfM) is the fundamental photogrammetric pipeline upon which modern Visual SLAM systems and 3D reconstruction tools are built. It is the computational process of simultaneously estimating the 3D geometry of a scene (the structure) and the pose of the camera (the motion) from a sequence of 2D images. Within the context of this thesis, SfM is not merely a background concept; it is the primary macroscopic mechanism being stress-tested by the proposed "Technology Evaluation" framework.[34][35].

While simulation provides the virtual environment, it is necessary to establish how the vision system interprets that environment. Structure from Motion (SfM), a photogrammetric technique that uses overlapping images to construct 3D surface models, is quickly emerging as a valuable research and education tool in geodesy, geomorphology, structural geology, and related disciplines. Images can be collected with a standard consumer-grade camera, making SfM a low-cost tool that compliments other 3D imaging technologies, such as terrestrial and airborne laser scanning.[38][39]

SfM can be collected from a hand-held camera or an airborne platform such as an aircraft, tethered balloon, kite, or UAS (unmanned aerial system), enabling 3D imaging of features ranging in size from decimetres to several kilometres. Structure from motion techniques are used in a wide range of applications including photogrammetric survey, the automatic reconstruction of virtual reality models from video sequences, and for the determination of camera motion (e.g. so that computer-generated objects can be inserted into video). This capability is foundational to the "Visual SLAM" systems discussed next, as both rely on the geometric principles of triangulation and parallax to recover depth from 2D images. [40][45]

While low-level operators like corner detection and ORB feature matching isolate individual points of interest, SfM is the architecture that aggregates these points into a cohesive spatial understanding. The standard SfM pipeline relies on a precise sequence of operations:

- **Feature Extraction and Matching:** Identifying common topological points across multiple overlapping images.
- **Camera Pose Estimation:** Calculating the physical trajectory and orientation of the camera between frames based on the geometric shift of the matched features.
- **Triangulation:** Computing the accurate 3D spatial coordinates of the matched features using the estimated camera poses.
- **Bundle Adjustment:** A global optimization step that simultaneously refines the 3D structure and the camera parameters to minimize reprojection error. [42][43]

Reviewing SfM is necessary here because its failure characteristics expose the vulnerability of scenario-based evaluation, and equally the inadequacy of synthetic data that is geometrically faithful but optically inaccurate. The pipeline is mathematically brittle in a specific and well-understood sense. Its feature matching stage assumes that a correspondence between images reflects a common physical point; when SNR degradation or uncalibrated lens blur causes that assumption to fail, the stage emits false correspondences. Robust estimation procedures such as RANSAC [73] are designed to reject a minority of such outliers, but they operate on the premise that the inlier set remains dominant. Once spurious matches cease to be a minority, bundle adjustment converges on a solution that is mathematically valid with respect to its inputs and physically incorrect: the camera track drifts and the reconstruction fails, without any diagnostic indicating that the failure originated in the imagery rather than in the geometry. Establishing where that transition occurs under controlled optical stress is what distinguishes a characterised vision system from one merely overfitted to a static benchmark,

and Monte Carlo treatments of corner and edge based robotic vision have demonstrated that such boundaries are tractable to quantitative analysis [42].

Validating an isolated corner detector is a necessary first step, but it is insufficient to prove the viability of synthetic data for full-scale autonomous navigation or mapping. SfM represents the critical bridge between low-level pixel analysis and high-level 3D spatial reasoning. By utilizing optically accurate, ray-traced imagery to intentionally induce failure in the SfM pipeline, this research provides a rigorous, physics-based assessment of a system's true technological resilience. [51][55]

## 2.5 Visual SLAM (Simultaneous Localization and Mapping)

Simultaneous Localisation and Mapping (SLAM) is becoming an increasingly important topic within the computer vision community and is receiving particular interest from the augmented and virtual reality industries. With a variety of SLAM systems being made available, from both academia and industry, it is worth exploring what exactly is meant by SLAM. This section gives a brief introduction to what SLAM is (and what it isn't), what it's for, and why it's important, in the context of computer vision research and development, and augmented reality.[53][56]

'SLAM' is not a particular algorithm or piece of software, but rather it refers to the problem of trying to simultaneously localize (i.e. find the position/orientation of) some sensor with respect to its surroundings, while at the same time mapping the structure of that environment. This can be done in a number of different ways, depending on the situation; so

the term “SLAM system” denotes a set of algorithms working to solve the simultaneous localization and mapping problem'.[59][60]

SLAM is not exclusively a computer vision problem and need not involve visual information at all; much of the early research addressed ground-based robots equipped with laser rangefinders, localising against tracked geometric beacons [32]. This thesis is concerned principally with Visual SLAM, in which the primary sensing modality is a camera [57], although many of the issues discussed apply more generally. The restriction to visual sensing is deliberate rather than incidental: it is the camera, uniquely among the common sensing modalities, whose output is governed by the radiometric variables that this research manipulates. A laser rangefinder returns range measurements whose error characteristics are largely independent of scene illumination and contrast, whereas a camera returns intensities that are determined by them. Any evaluation framework built on optical degradation is therefore specific to visual sensing, and this constitutes both the scope and the limitation of the work presented here.

The requirement of recovering both the camera's position and the map, when neither are known to begin with, distinguishes the SLAM problem from other tasks. For example, marker-based tracking is not SLAM, because the marker image (analogous to the map) is known beforehand. 3D reconstruction with a fixed camera rig is not SLAM either, because while the map (here the model of the object) is being recovered, the positions of the cameras are already known. The challenge in SLAM is to recover both camera pose and map structure while initially knowing neither.[71][77]

An important distinction between SLAM and other apparently similar methods of recovering pose and structure is the requirement that it must operate in **real time**. This is a somewhat slippery concept, but in general it means that the processing done on each incoming

camera image must be finished by the time next one arrives, so that the pose of the camera is available immediately and not because of a post-processing stage. This distinguishes SLAM from techniques like Structure from Motion, where a set of unordered images are processed offline to recover the 3D structure of an environment, in a potentially (extremely) time consuming process. This can achieve impressive results, but crucially can't tell you where the camera is during acquisition.[73][74]

Modern vSLAM systems, such as **ORB-SLAM**, utilize feature-based methods to track specific points (corners, edges) across frames. These systems are highly effective but brittle; changes in texture, lighting, or motion blur can cause "tracking loss," where the system loses its place in the world. Validating exactly *when* and *why* this tracking loss occurs is a primary candidate for the Simulation-based evaluation proposed in this thesis. [78][79]

Since the feature-based systems described above were established, the field has consolidated around ORB-SLAM3, which extends the original architecture to visual, visual–inertial and multi-map operation and remains the reference implementation against which subsequent systems are benchmarked [116]. Recent surveys of the field document a clear bifurcation: classical geometric pipelines on one side, and learning-based systems on the other [117]. DROID-SLAM is representative of the latter, replacing hand-engineered feature association with a recurrent network that iteratively updates camera pose and depth, achieving substantially improved robustness on sequences where classical tracking fails [118]. More recently, the map representation itself has been reconsidered. Systems such as Gaussian Splatting SLAM [119] and SplaTAM [120] abandon sparse point clouds in favour of differentiable volumetric representations optimised directly against photometric error, producing dense, photorealistic reconstructions in real time.

These developments do not diminish the problem this thesis addresses; they sharpen it. Each of these systems, whether it associates features by descriptor distance or by gradient descent on a rendered image, ultimately consumes the same quantity: pixel intensities produced by a physical sensor under particular optical conditions. Learning-based and photometric methods are, if anything, more tightly coupled to the radiometric properties of the input than the sparse feature methods they supersede, because they optimise against raw intensity rather than against a sparse set of maxima that has already been filtered for stability. The question of where a system's performance boundary lies as signal-to-noise ratio degrades or lens blur increases therefore remains open, and remains poorly served by static benchmark datasets in which those variables cannot be independently manipulated.

## 2.6 The Use of Simulation in Robotics

Simulation based design has been the cornerstone of engineering for a long time. Simulations and designs are developed together to investigate ongoing challenges in the construction of working systems. For sufficiently complex systems, simulation provides the most efficient way to investigate performance and design choices, and to iterate on modifications to proven designs. New circumstances generate the need for modified simulations, to keep pace with real-world problems.[81][84][87]

Within robotics research, simulated virtual worlds of varying realism have long been constructed to assess the performance of robotic algorithms, and 3D simulation environments are widely used for the evaluation of robotic vision systems in particular. A recurrent limitation of this body of work is that each simulator has typically been built to evaluate one specific system: stereo-based fruit positioning [41], legged football robots [13], or colour-based object

positioning and model recognition [15]. Such environments possess limited flexibility for transfer between robotic configurations, because the fidelity they implement is the fidelity that their originating task required. A second group of works employs pre-acquired or synthesised image sequences [14], which are reproducible but cannot generate the visual feedback signals needed to close a motion control loop. The most relevant precedents for the present work are those that treated the *camera itself* as the object of simulation rather than as a transparent window onto the scene, notably the camera error simulations of Hinkenjann et al. [12] and the sensor-aware environment simulator of Asanuma et al. [13]. These established that simulating sensor imperfection changes algorithmic outcomes measurably. What they did not undertake, and what the present research contributes, is the calibration of those simulated imperfections against quantities measured from a specific physical camera, and the verification of the resulting failure predictions against real imagery captured under matched conditions.

The autonomous vehicle sector provides the clearest industrial demonstration of simulation-led development. Waymo's programme is instructive in this respect: physical road mileage is accompanied by a far larger volume of simulated mileage, generated specifically to expose the system to rare events that would occur only infrequently in physical operation [29][30]. Tesla operates a driver assistance system capable of lane maintenance with limited intervention, and substantial work remains before full autonomy is achieved across either programme. The methodological significance of these efforts for the present thesis lies less in the scale of the simulated mileage than in what that scale implicitly concedes: the industry has accepted that the operational envelope of a vision system cannot be established by physical data collection alone, because the conditions of interest are precisely those that occur too rarely to be captured. That concession is an argument for simulation as an evaluation instrument, but it does not by itself address the question of how much optical fidelity such simulation requires in order for its predictions to transfer, which remains the open problem.

Self-driving cars have the potential to change the paradigm of transportation. [101][107]. According to the U.S. Department of Transportation National Motor Vehicle Crash Causation Survey, 93 percent of all vehicle accidents are influenced by human error while driving; eliminating most of the accidents caused by human error would be a giant boon for public safety. Self-driving cars also promote ride-sharing which would reduce environmental impact immensely. Additionally, parked cars take up tremendous space in highly populous cities that could be otherwise used for increased living spaces.[110][113]

The real world offers only one reality (scenario) at a time that is in many cases extremely challenging and costly to create, replicate and iterate. Virtual reality, while only approximating reality, does provide the opportunity, if properly designed, to readily create scenarios, readily capture the "labelled" data of those scenarios and effectively investigate neighbouring variances of those scenarios. The availability of such environments to test and understand the robustness and effective range of automated driving approaches is thought to be an effective element in the design of those approaches. While safe behaviour in the real world is the final test, the availability of a robust VR environment is thought to be an very effective tool.[113][114]

The academic simulation frameworks discussed above have direct commercial counterparts, and the maturity of that commercial ecosystem is itself evidence for the argument advanced in this chapter. CARLA established simulation-based testing of autonomous perception as standard industrial practice, providing an open urban driving simulator with configurable sensor suites and full environmental control [132]. On the general-purpose side, the Unity Perception package extended a commercial game engine into a synthetic data generator with automatic ground-truth annotation, placing randomised synthetic dataset production within reach of practitioners without specialist rendering expertise [133]. That such

tools exist and are widely adopted indicates that the central premise of this thesis, that simulation is a legitimate substitute for physical capture, is no longer contentious within industry; what remains contested is the degree of fidelity required, and how that fidelity should be demonstrated.

The commercial synthetic data toolchain has since consolidated around integrated ecosystems rather than standalone renderers. NVIDIA's Omniverse Replicator provides a programmable pipeline for generating annotated synthetic datasets with domain randomisation applied to materials, lighting and object placement [134], and is coupled to DRIVE Sim for vehicle-level sensor simulation [135]. A commercial sector supplying synthetic training imagery as a service has grown alongside these platforms. The prevailing commercial objective across this sector mirrors the prevailing academic one: maximise the volume, diversity and annotation richness of the generated imagery, on the assumption that a sufficiently varied training distribution will encompass the conditions encountered at deployment.

A distinct and more directly relevant class of commercial tool addresses optical fidelity rather than data volume. Ansys AVxcelerate Sensors provides physics-based camera simulation that models the lens system and the optoelectronic properties of the imager, producing raw images intended for the validation of perception algorithms rather than for their training [136]. The academic foundations of such tools are well developed; Goossens et al., for instance, describe a ray-transfer formulation permitting accurate simulation of a physical lens whose design remains proprietary, validated against edge spread functions and relative illumination [137]. The existence of this tooling confirms that the optical matching pursued in this thesis is regarded as necessary within industry, and is not an idiosyncratic requirement of the present research.

The most instructive comparison, however, is with NVIDIA's published validation of its simulated camera models, which is the closest industrial analogue to the methodology developed in the chapters that follow [135]. That work matches simulated to physical cameras and validates the correspondence quantitatively, reporting agreement in intrinsic and extrinsic calibration parameters, in reprojection error, and in colorimetric difference. The distinction to be drawn is nonetheless a substantive one. Such validation establishes that a simulated camera reproduces the calibration and colorimetric properties of its physical counterpart, which is a statement about the fidelity of the image. It does not establish the point at which a downstream vision operator ceases to function, which is a statement about the performance boundary of an algorithm. The two are related but not equivalent: two images may be colorimetrically indistinguishable while differing in the high-frequency noise structure that determines whether a gradient-based detector returns four corners or four hundred. It is precisely this second question, the location of the operator's failure boundary and whether that boundary can be predicted from simulation, that this thesis addresses.

This thesis builds upon this philosophy but applies it specifically to the "Technology Evaluation" of the underlying vision algorithms. Rather than training a vehicle to drive, the mathematical limits of the sensor are tested by creating a "Digital Twin" of a laboratory environment in which every photon of light is controlled.[115]

## 2.7 Synthetic Data Generation for Computer Vision

Parallel to developments in simulation frameworks, the deliberate generation of synthetic imagery as a substitute for captured data has matured into a research area in its own right. TartanAir is the work most directly adjacent to this thesis: a large-scale synthetic dataset rendered in photorealistic simulated environments and released specifically to stress visual SLAM systems under challenging conditions including adverse weather, dynamic objects and difficult illumination, with perfect ground truth for camera pose, depth and optical flow [121]. Kubric provides a scalable pipeline for generating annotated synthetic video with rich per-pixel labels [122], while Infinigen takes the approach further, generating unbounded photorealistic natural environments procedurally, such that every asset is mathematically defined rather than drawn from a fixed library [123]. A recent systematic review of indoor synthetic data generation catalogues the breadth of these methods and identifies the persistent obstacle common to all of them, namely the reality gap between rendered and captured imagery [124].

The contribution of this thesis must be positioned carefully against this body of work, since the overlap is superficially considerable. Datasets such as TartanAir supply synthetic imagery of high geometric and semantic fidelity, and they supply it at a scale no individual research programme can match. What they do not supply, and do not set out to supply, is a calibrated correspondence between the optical characteristics of the rendered imagery and those of a specific physical sensor. Their ground truth is geometric: the pose is exact because it is known by construction. Their radiometry, by contrast, is plausible rather than matched, because there is no physical camera whose signal-to-noise ratio and blur profile the renderer is attempting to reproduce. Consequently these datasets remain instruments of Scenario Evaluation, albeit scenarios that are synthetic, repeatable and abundant. They can establish that a system performs well across many conditions, but they cannot isolate the specific radiometric

variable at which it ceases to perform, because the variables are bundled together within each scene rather than swept independently against a measured physical reference.

The methodology developed in this thesis inverts that emphasis. Rather than maximising the volume and diversity of synthetic imagery, it minimises the discrepancy between a synthetic image and one specific real image, by measuring the signal-to-noise ratio and blur gradient of the physical capture and reproducing both within the render. Scale is deliberately sacrificed in favour of calibration, on the argument that a predictive model of operator failure requires a controlled one-to-one comparison against a measured physical counterpart, which a large uncalibrated corpus cannot provide.

## 2.8 Digital Twins and the Scope of the Term

The term "Digital Twin" originates in manufacturing and aerospace and has since been adopted across engineering disciplines. A comprehensive survey by Mihai et al. traces its development and, importantly for the present work, identifies the definitional inconsistency with which it is applied across the literature [127]. In its strict industrial formulation, a digital twin comprises three elements: a physical entity, a virtual counterpart, and a continuous bidirectional data linkage such that the virtual model updates in response to the state of the physical asset and, in mature implementations, informs its control. Under this definition the defining property is the live coupling, not the fidelity of the model.

It is therefore necessary to state explicitly the sense in which the term is employed throughout this thesis, since the usage here is narrower than the industrial definition. The virtual environments constructed in Chapter 3 are static, high-fidelity replicas: they reproduce the geometry, surface texture and optical characteristics of a physical space at the moment of capture, and they are validated against that space quantitatively through cloud-to-cloud comparison. They do not maintain a live data connection to the physical room, and no claim of continuous state synchronisation is made. What is being twinned is not the operational state of an asset but the *imaging conditions* under which a vision system observes it. The linkage that matters here is a calibration linkage rather than a telemetric one: parameters measured from the physical sensor, specifically signal-to-noise ratio and blur gradient, are transferred into the renderer so that the synthetic imagery is optically equivalent to the physical imagery. Where the term "Digital Twin" appears in the chapters that follow, it should be read in this restricted sense of a geometrically and optically calibrated static replica, and not as a claim to the full bidirectional coupling described in the industrial literature.

## 2.9 Learning-Based Feature Detection and Matching

The low-level operators characterised in Chapter 4, corner detection and ORB descriptor matching, have been substantially supplemented in recent years by learned alternatives. SuperPoint demonstrated that interest point detection and description could be trained jointly in a self-supervised manner, using homographic adaptation to generate pseudo-ground-truth in the absence of labelled keypoints [128]. SuperGlue subsequently reframed the matching stage itself as a learnable problem, applying a graph neural network with attention to reason jointly about the assignment between two sets of keypoints rather than matching descriptors independently [129]. LoFTR removed the detection stage altogether, establishing dense correspondences directly between image pairs using transformers, with markedly improved performance in low-texture regions where detector-based methods have no keypoints to offer [130].

This progression bears directly on the argument of this thesis, and it would be reasonable for a reader to ask why classical operators were retained as the object of study. Three considerations motivate that decision. First, the aim of Chapter 4 is to construct a performance characterisation analogous to a component datasheet, which requires an operator whose response to an input is deterministic and analytically traceable. A learned detector's response is a function of its training distribution as well as its input, and a measured failure boundary would therefore characterise the union of the operator and its training data rather than the operator alone. Second, classical detectors remain in active deployment in production SLAM systems, including ORB-SLAM3 [116], so their characterisation is of continuing practical rather than merely historical interest. Third, and most consequentially, the extensive benchmarking of Jin et al. across wide-baseline matching tasks established that classical

pipelines, when properly tuned, remain competitive with and in certain configurations exceed learned alternatives, and that reported performance differences are frequently attributable to evaluation protocol rather than to the methods themselves [131]. That finding is itself an argument for the style of controlled, variable-isolating evaluation pursued here.

It should be acknowledged that the methodology developed in this thesis is in principle operator-agnostic. The calibration procedure, matching blur and signal-to-noise ratio between real and synthetic imagery, makes no assumption about what subsequently consumes the image. Characterising a learned detector such as SuperPoint or a detector-free matcher such as LoFTR within the same framework, and determining whether their failure boundaries under radiometric degradation are as sharply defined as those measured here for classical corner detection, is a direct and worthwhile extension of this work.

## 2.10 Summary and Identification of the Research Gap

The literature reviewed in this chapter establishes three conditions that jointly define the gap addressed by this thesis. First, evaluation of visual navigation systems remains predominantly scenario-based, conducted against fixed benchmark datasets in which environmental and sensor variables are frozen at the moment of recording and cannot subsequently be isolated or swept [117]. Second, synthetic imagery has been convincingly established as a viable source of training and evaluation data, and modern generators produce it at a scale and geometric fidelity that physical capture cannot approach [121][122][123]; however, the prevailing objective in that literature is diversity and volume, with optical correspondence to a specific physical sensor left unaddressed [124]. Third, validation of

synthetic data is typically conducted at the level of complete systems, through macroscopic metrics such as trajectory error, rather than at the level of the individual operators from which those systems are composed.

The consequence of the third condition deserves emphasis, because it is the point on which this thesis turns. If the low-level operators that a system depends upon do not respond identically to synthetic and real imagery, then agreement between the two at the level of trajectory error is not evidence of simulation fidelity; it may equally be the product of compensating errors within a pipeline complex enough to absorb them. A bottom-up validation, in which the fidelity of the synthetic imagery is first established at the operator level under calibrated optical conditions, is therefore a logical precondition for trusting simulation-based evaluation of the complete system. Providing that validation, together with the calibration methodology that makes it possible, constitutes the contribution of the chapters that follow.

# Chapter 3: Methodology: Constructing and Validating the Digital Twin

## 3.1 Overview of the Experimental Framework

This chapter addresses RQ_1 and RQ_2. RQ_1 asks whether a ray-traced environment can reproduce a physical one with sufficient fidelity that an identical photogrammetric pipeline yields agreeing reconstructions, and is tested through hypothesis H1. RQ2 asks which acquisition parameters govern that fidelity, and is addressed through the iterative sequence of room experiments in §3.5. The chapter establishes the geometric foundation on which the operator-level work of Chapters 4 and 5 depends: unless the virtual environment is geometrically faithful, no conclusion regarding optical fidelity can be drawn from it.

| ID | Chapter | Experiment | Scene | Images | Sensor / resolution | Variable under test | Outcome measure | RQ |
|---|---|---|---|---|---|---|---|---|
| E1 | 3.2 | Cube baseline, physical | Textured cube | 256 | Digital camera; 4 heights × 64 poses; 1 m radius; 5° increments | — (baseline) | Dense cloud; camera track recovery | RQ_1 |
| E2 | 3.3 | Cube baseline, synthetic | POV-Ray cube | 256 | Virtual camera at Vicon poses | — (baseline) | Dense cloud | RQ_1 |
| E3 | 3.4 | Cube C2C comparison | E1 vs E2 | — | — | Real vs synthetic | C2C distance, 0.000–0.411 units | RQ_1 / H1 |
| E4 | 3.5.2 | Room iteration 1 | Untextured room | 108 | SJCAM 5000X fisheye, 12 MP | Sampling density (9 positions) | Alignment success < 30% | RQ_2 |
| E5 | 3.5.3 | Room iteration 2 | Untextured room | 252 | SJCAM 5000X fisheye, 12 MP | Sampling density (21 positions) | Partial reconstruction; whiteboard failure | RQ_2 |
| E6 | 3.5.4 | Room iteration 3a | Untextured room | 420 | SJCAM 5000X fisheye, 12 MP | Sampling density (35 positions) | Persistent artefacts | RQ_2 |
| E7 | 3.5.4 | Lens comparison | Untextured room | 420 | Canon DSLR rectilinear, 12 MP | Lens geometry (fisheye vs rectilinear) | Improved structural coherence | RQ_2 |
| E8 | 3.5.5 | Room iteration 3b | Untextured room | 1260 | SJCAM 5000X fisheye, 12 MP | Extreme density, multi-angle | Computational failure (> 4 days; 8 GB exceeded) | RQ_2 |
| E9 | 3.5.6 | Reduced-set baseline | Untextured room | 180 | SJCAM 5000X, 12 MP | Sampling density (15 positions) | ≈ 30% reported accuracy | RQ_2 |
| E10 | 3.5.6 | Texture intervention | Room + patterned paper | 180 | SJCAM 5000X, 12 MP | Surface texture | > 95% alignment; ≈ 2× point density | RQ_2 |
| E11 | 3.5.8 | Angular diversity | Textured room | 540 | SJCAM 5000X, 12 MP | Vertical angles (0°, +45°, −45°) | ≈ 199,000 points | RQ_2 |
| E12 | 3.5.8 | Resolution sweep | Textured room | 540 | 12 / 8 / 5 / 2 MP | Sensor resolution | 199k / 198k / 140k / 17k points | RQ_2 |
| E13 | 3.7 | Room digital twin validation | Real vs POV-Ray room | 540 each | 8 MP; matched poses | Real vs synthetic | Final RMS 0.74635 | RQ_1 / H1 |
| E14 | 4.2–4.3 | Noise sensitivity | 2D target, orthographic | — | Synthetic | Gaussian noise, 1% increments | Corner count vs noise; knee ≈ 20–30% | RQ_3 / H2 |
| E15 | 4.4 | Contrast × noise sweep | 2D target | 255 contrast levels | Synthetic | Contrast (255 steps) × noise | Performance manifold; Safe Operating Area | RQ_3 / H2 |
| E16 | 4.5.1 | Validation, high SNR | 2D target | 2 configurations | Canon; Epson | Real vs matched synthetic | 13 and 5 corners vs 4 predicted | RQ_4 |
| E17 | 4.5.2 | Validation, degraded | 2D target | 3 configurations | Sony; Nikon lit; Nikon dark | Real vs matched synthetic | Divergence traced to unmodelled blur | RQ_4 |

| ID | Chapter | Experiment | Scene | Images | Sensor / resolution | Variable under test | Outcome measure | RQ |
|---|---|---|---|---|---|---|---|---|
| E18 | 4.6 / 5.2 | Blur-matched calibration | 2D target | — | Matched pairs | Blur + blurred noise | Convergence at SNR 25 | RQ5 / H3 |
| E19 | 5.4 | ORB extension | 2D target | — | Synthetic | Translation; rotation | Matches retained | RQ5 (preliminary) |

*Table 2 - Register of experiments, with parameters and research-question mapping*

| Used in | Device | Lens | Resolution | Focal length | Aperture | ISO | Shutter | White balance | Mounting |
|---|---|---|---|---|---|---|---|---|---|
| E1 (cube) | SJCAM SJ5000X | Rectilinear, fixed | *10 MP (3648 × 2736)]* | *6.2 mm (≈ 35 mm equiv.)* | *f/4.0* | *200]* | *1/60 s* | *Auto* | Vicon-tracked, handheld |
| E4–E6, E8–E12 (room) | SJCAM SJ5000X | *Fisheye, fixed, ≈ 170° FOV* | 12 MP (4032 × 3024); also processed at 8 / 5 / 2 MP | *2.99 mm (≈ 12 mm equiv.)* | *f/2.8, fixed* | *Auto (100–400)* | *Auto* | Auto | Tripod, 2 elevations |
| E7 (lens comparison) | *Canon EOS DSLR* | *Rectilinear zoom, 18–55 mm* | *12 MP (4272 × 2848) [confirm]* | *18 mm* | *f/8.0* | *400* | *1/30 s* | *Fluorescent* | Tripod, 2 elevations |
| E16 (high SNR) | *Canon EOS DSLR* | *Rectilinear, fixed 50 mm* | *12 MP* | *50 mm* | *f/8.0* | *100* | *1/125 s* | *Daylight* | Orthographic, fixed |
| E16 (high SNR) | *EpsonR-D1* | *Rectilinear, fixed 50 mm* | *12 MP* | *50 mm* | *f/8.0* | *100* | *1/125 s* | *Daylight* | Orthographic, fixed |
| E17 (degraded) | *Sony DSLR / mirrorless* | *Rectilinear zoom* | *12 MP* | *35 mm* | *f/5.6* | *400* | *1/60 s* | *Auto* | Orthographic, fixed |
| E17 (degraded, lit) | *Nikon DSLR* | *Rectilinear zoom* | *12 MP* | *35 mm* | *f/5.6* | *800]* | *1/60 s* | *Auto* | Orthographic, fixed |
| E17 (degraded, low light) | *Nikon DSLR* | *Rectilinear zoom* | *12 MP* | *35 mm* | *f/3.5, wide open* | *3200* | *1/15 s* | *Auto* | Orthographic, fixed |

*Table 3 — Camera configurations*

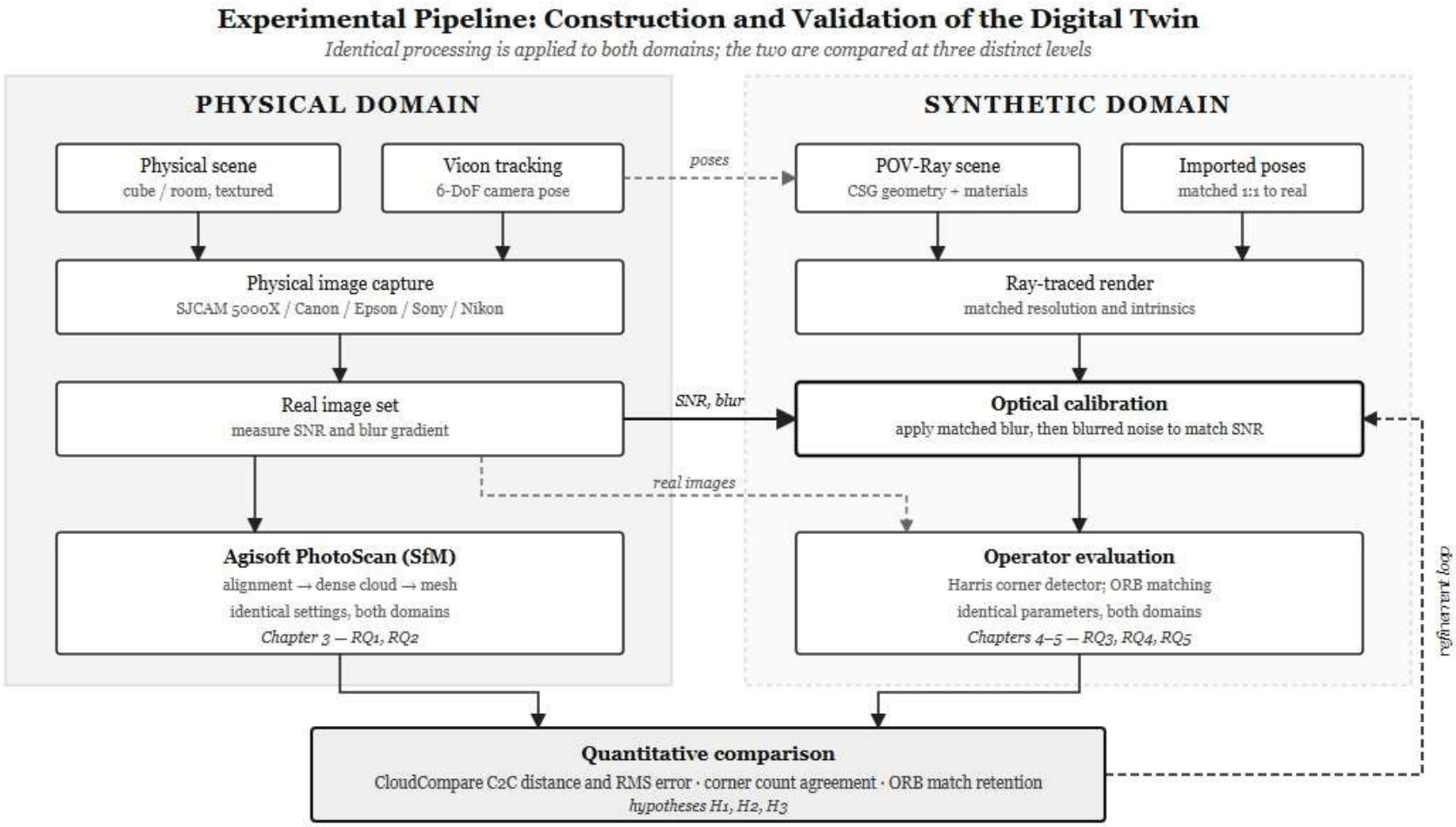


*Figure 2 — Overview of the experimental pipeline*

Solid arrows denote data flow, while dashed arrows denote parameter transfer between domains. The heavy-bordered stage is the methodological contribution: calibration of synthetic optics against measured physical values.

| Choice | Alternatives considered | Justification | Reference |
|---|---|---|---|
| Ray tracing over rasterisation | Game engines | Pixel values must result from computed light transport, not post-process approximation | Chapter 2.3.3; [132][133] |
| POV-Ray as rendering engine | PBRT, Mitsuba, Blender | Exact CSG geometry; scriptable pose import; see Table 12 | [137] |
| Vicon for ground truth | Visual odometry; fiducial markers | External measurement independent of the vision system under test, avoiding circularity | Chapter 3.2.1 |
| Agisoft PhotoScan for SfM | COLMAP, VisualSFM, Meshroom | Applied identically to both domains, so systematic bias affects both equally; treated deliberately as a black box | Chapter 3.3 |
| CloudCompare for comparison | MeshLab; CAD-based comparison | Established octree cloud-to-cloud implementation with published methodology | [7] |
| Harris corner detection | FAST, Shi–Tomasi, SuperPoint | Deterministic and analytically traceable response; underlies feature-based SLAM; learned detectors would confound operator with training distribution | Chapter 2.9 [128][131] |
| ORB for feature matching | SIFT, SURF, AKAZE, SuperGlue | The descriptor used by the target SLAM system; binary descriptor with published behaviour | [116][129] |

| Choice | Alternatives considered | Justification | Reference |
|---|---|---|---|
| Two-image correlation SNR | Single-image noise estimators | Estimates noise from two independent realisations without requiring a noise-free reference | Chapter 4.1.3 |
| Variance of Laplacian as blur metric | MTF; edge spread function | Single scalar computable from an arbitrary image without a calibration target | [137] |
| 2D orthographic target | Full 3D scene | Eliminates perspective, occlusion and shading as confounds, isolating radiometric variables | Chapter 4.1.4 |

*Table 4 - Summary of major methodological choices*

The core hypothesis of this thesis is that simulated imagery can effectively replace real-world data for the evaluation of computer vision systems. To demonstrate this, it requires first establishing that a simulation can be constructed with sufficient fidelity that a computer vision algorithm cannot distinguish it from reality. This chapter details the methodology used to create this "Digital Twin", a virtual environment that mirrors a real-world space in geometry, lighting, and texture. The experimental strategy is divided into distinct milestones designed to progressively validate the simulation pipeline.

The processing pipeline was implemented as follows, and the source listings for each stage are given in the Appendix. Physical camera poses recorded by the Vicon system were exported and transformed into POV-Ray camera declarations, so that each rendered frame corresponds to a physical capture at the same position and orientation. Rendering was performed by POV-Ray at the resolution matching the physical sensor. Photogrammetric reconstruction was performed in Agisoft PhotoScan Professional at Medium quality, this setting having been adopted after the Maximum setting was found in Chapter 3.5.6 to require approximately one week of processing without a corresponding improvement in the resulting model; identical settings were applied to real and synthetic image sets in every comparison. Point clouds were exported in Polygon File Format (.PLY), cropped to the region of interest, coarsely registered by manual point correspondence, refined by Iterative Closest Point, and compared by cloud-to-cloud absolute distance in CloudCompare. All image degradation, measurement and operator evaluation was implemented in Python using OpenCV, NumPy and SciPy, with parameters as given in Table 10. Processing was performed on an Intel Core i5 workstation with 8 GB of RAM, a constraint that proved material to the results reported in Chapter 3.5.5.

The software versions used were: POV-Ray 3.7.0, Agisoft PhotoScan Professional 2.3.1, CloudCompare 2.13.12, Python 3.15, and OpenCV 4.14.

### 3.1.1 The First Milestone: Establishing the Baseline

The initial phase involves the capture of real-world data using a VICON motion capture system to establish a ground truth trajectory. This is followed by modelling the environment in software to create a virtual counterpart. The validation logic is as follows: if the real-world data and the simulated data are fed into the same photogrammetry pipeline (Structure from Motion), they should yield 3D models that are statistically similar.

When the simulated world data matches, or comes near, to the real-world data, both datasets are processed to generate 3D reconstructions. Ideally, this system will return two 3D models that should be more or less similar. At this point, the critical task is to identify why the simulated 3D model differs from its real counterpart, if indeed it does, and to investigate the specific details of that divergence (the "Reality Gap").

The simulation is then refined and tweaked, adjusting lighting parameters, surface properties, and camera intrinsic, so that it yields a 3D model that is nearly identical to that derived from real-world data. Once this alignment is achieved, a methodology will have been established to produce simulations that are as accurate as if one had captured them in the real world with a physical camera. This first step is crucial: it demonstrates that a simulation can be a reliable substitute for replacing data that comes from the real world.

### 3.1.2 The Second Milestone: Predictive Stress Testing

Following the validation of the baseline simulation, the next step is to introduce artifacts to the simulated data. These artifacts include:

- Loss in picture quality (resolution reduction).
- Introduction of specular reflections.
- Changes in lighting conditions (contrast reduction).
- Image warping and lens distortion.

With these changes implemented in the virtual camera model, it may be predicted that the real-world data should yield a similar degraded 3D model if the same artifacts were present. The experimental validation involves introducing the same artifacts in the real world (e.g., using a lower quality camera or changing the physical lighting) to see if the resulting data leads to a 3D model that matches the predicted result. If the 3D model indeed looks the same, it will have been demonstrated that the simulation is accurate enough to predict the performance boundaries of vision systems. This data can then be used for training Vision Systems (such as CNNs) as accurately as if the data would have been obtained from the real world.

### 3.1.3 SLAM as the Ultimate Metric

While 3D reconstruction provides a static metric of quality, the end goal is to validate the system for navigation. Once the 3D models of the environment closely resemble each other and the simulated environment looks as realistic as possible, the final validation step is to pass these image sequences through a visual SLAM system. The best candidate at the moment is ORB-SLAM, a feature-based system widely used in the robotics community.

If the simulation is valid, ORB-SLAM should yield similar trajectory estimates and feature maps for both the real and simulated data. If these results do not appear to be similar, further refinement of the simulation will be required to capture the specific features (such as corner sharpness or texture gradients) that the SLAM algorithm relies upon. The final aim is to demonstrate that changes in the simulation affect the vision system in the same way as changes in the real world would. This confirms that simulation is a promising method for training and exploring the performance capabilities of a Vision System intended for real-world deployment.

The selection rests on traceability rather than performance. A system that fails gracefully is harder to characterise than one whose failure can be localised to a named component, and ORB-SLAM's sparse feature architecture makes that localisation possible. As recorded in Chapter 1.2 and Chapter 4.1, the system-level evaluation was subsequently scoped out in favour of establishing correspondence at the operator level first. The operators examined in Chapters 4 and 5, Harris corner detection and ORB feature matching, are precisely those on which ORB-SLAM depends, so the characterisation obtained remains directly relevant to the system originally targeted.

| System | Approach | Reason considered | Reason not selected |
|---|---|---|---|
| **ORB-SLAM / ORB-SLAM3** | Sparse, feature-based (ORB descriptors) | Selected. Dependence on discrete, inspectable keypoints makes the causal path from image degradation to tracking failure traceable to a specific operator, which is a prerequisite for operator-level analysis. Remains the reference implementation for feature-based SLAM [116] | — |
| LSD-SLAM | Direct, semi-dense photometric | Operates directly on intensity gradients and is therefore sensitive to radiometric change | Photometric error is distributed across the image rather than localised in identifiable features, so failure cannot be attributed to a characterisable operator [117] |
| DSO | Direct sparse odometry with photometric calibration | Incorporates an explicit photometric camera model, closely aligned with the concerns of this thesis | Photometric calibration would confound the experiment: the system compensates internally for the degradations under test [117] |
| RTAB-Map | Appearance-based RGB-D graph SLAM | Mature and widely deployed in mobile robotics | Depends on a depth sensor whose failure modes are not optical in the sense examined here |
| DROID-SLAM | Learned, recurrent dense bundle adjustment | Substantially more robust under degraded conditions [118] | Response is a function of training distribution as well as input, so a measured boundary would characterise the network and its training data jointly rather than the operator alone |

*Table 5 - Candidate SLAM systems considered*

## 3.2 Real-World Data Acquisition

This experiment addresses Objective 2 and provides the initial test of RQ_1 on a single object of known geometry, deliberately chosen for its simplicity before the approach is scaled to a full environment.

To tackle the first milestone, a rigorous data collection protocol was established using a controlled environment. The objective was to capture a physical object (a textured box) with precise knowledge of the camera's position, allowing this position to be replicated exactly in the virtual world. Real-world data was captured using a VICON camera system. The VICON system is an optical motion capture array that tracks reflective markers to determine the position (X, Y, Z) and orientation (Roll, Pitch, Yaw) of the camera in 3D space. This provides the "Ground Truth" necessary to drive the virtual camera in the simulation later.

### 3.2.1 The Vicon Ground Truth System

The subject of the capture was a cube box wrapped in textured paper (Christmas wrapping paper). This texture provides the high-frequency visual features (corners and edges) required by Structure from Motion (SfM) algorithms to extract depth information. A featureless white box would fail to reconstruct; the complex texture ensures the vision algorithms have sufficient data to process.

The capturing of this data consisted of 256 pictures of the cube box taken with a generic digital camera. To ensure full coverage for 3D reconstruction, a systematic approach was taken:

- **4 distinct height levels** were defined to capture the object from different perspectives.

- **64 pictures** were taken at each height level.
- The camera was rotated in **5-degree increments** along a circle of radius 1 meter around the box.

This dense sampling strategy ensures sufficient overlap between frames, a requirement for photogrammetric reconstruction.

*Figure 3-Captured Samples*

### 3.2.2 Photogrammetric Reconstruction (PhotoScan)

After these pictures were acquired, they were processed using **Agisoft PhotoScan Pro**. The software has since been renamed Agisoft Metashape, the change taking effect from version 1.5 in 2019 [138]. The name PhotoScan is retained throughout this thesis to identify the version actually used in these experiments. This software utilizes Structure from Motion (SfM) algorithms to identify common features across the 256 images and reconstruct the 3D geometry of the scene.

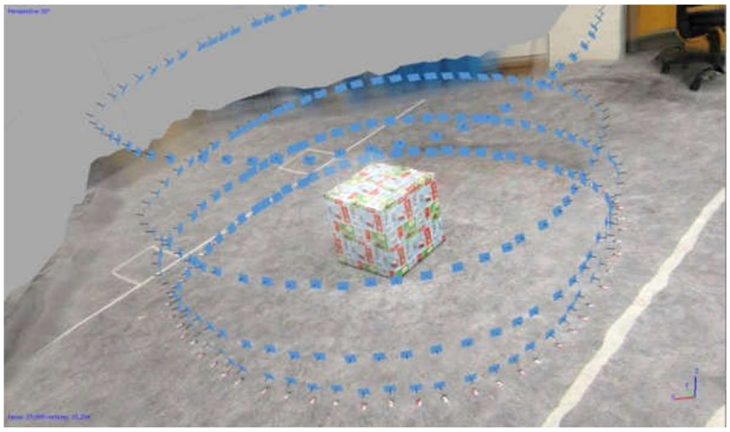

*Figure 4-3D Reconstruction of real-world samples*

As shown in Figure 4, the blue squares circling the box represent the recovered camera positions confirming the circular capture path. These match the 4 sets of 64 pictures mentioned earlier, confirming that the reconstruction software successfully interpreted the camera's path around the box at a 1-meter radius. This 3D reconstruction of the real environment serves as the "Ground Truth Model" against which the simulation will be compared.

## 3.3 The simulation Pipeline (POV-Ray)

With the real-world baseline established, the next step was to reproduce the scene in a virtual environment under identical conditions. For this purpose, **POV-Ray** (Persistence of Vision Raytracer) was selected. Unlike standard rasterization engines (used in video games), POV-Ray utilizes ray tracing, a technique that simulates the physical path of light.

Ray tracing takes a description of geometry (which includes surface materials and light sources), a model of a virtual camera, and a projection plane made up of the pixels in the final image. It then creates what the camera would "see" by shooting rays from the camera through each pixel and analysing how they are affected by the geometry, materials, and lights. This

allows for the generation of photorealistic artifacts, such as shadows and reflections, which are critical for testing computer vision robustness.

With the help of POV-Ray, virtual scenes can be described using a script language. This parametric approach allows for precise control: the shape and appearance of objects within the 3D world are defined mathematically. Crucially, the **real-world camera positions** captured with the VICON system were imported directly into the POV-Ray script. This ensured that the virtual camera was placed at exactly the same coordinates (X, Y, Z) and orientation as the real camera during the physical data capture.

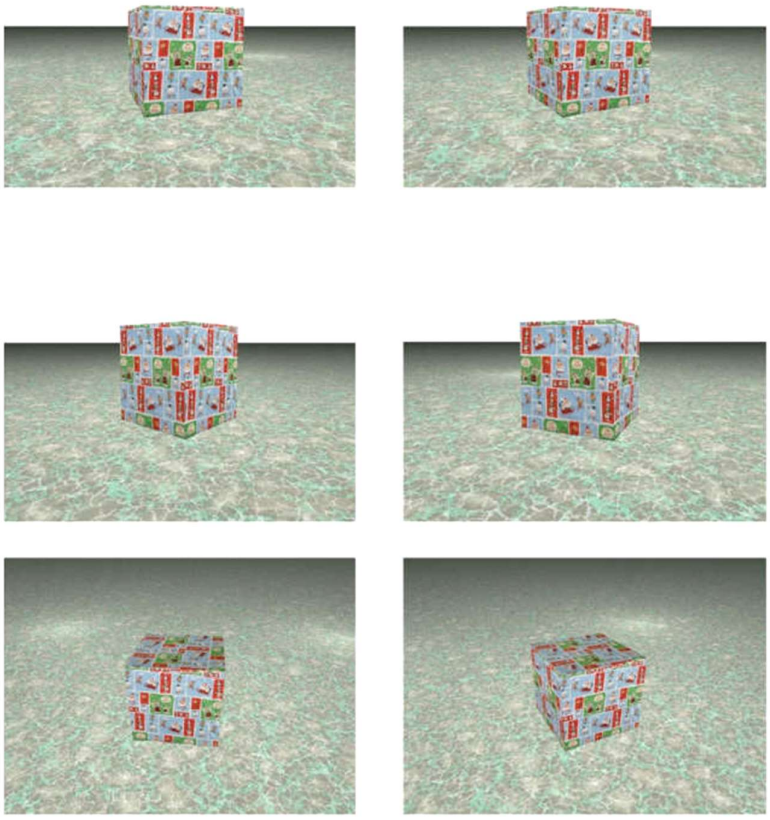

*Figure 5-POV-Ray computer generated samples*

Once these synthetic sample images were rendered in POV-Ray, they were passed through the same Agisoft PhotoScan pipeline used for the real images. This step is critical to ensure a fair comparison: both the real and synthetic datasets are treated as "black box" inputs to the reconstruction software.

By generating a 3D model from the virtual images, it is verified that the simulation contains sufficient visual information (texture, parallax, shading) to drive a complex computer vision algorithm. In the Figure 5 below, the scene is reconstructed using the simulated samples. The geometry of the reconstructed mesh closely mirrors the real-world result.

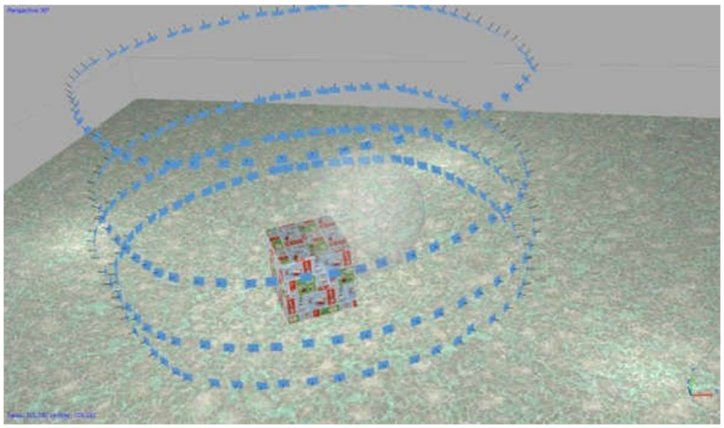

*Figure 6-3D reconstruction of simulated samples.*

## 3.4 Quantitative validation (Cloud Compare)

Visual inspection alone is insufficient to verify the accuracy of the Digital Twin. To quantify the similarity between the "Real" and "Virtual" 3D models, CloudCompare was employed. CloudCompare is an open-source 3D point cloud and triangular mesh processing software designed to perform direct comparison between dense 3D datasets.

The software relies on a specific octree structure that enables high performance when calculating the Harsdorf distance between point clouds. The validation process involved the following steps:

1. **Cropping:** Because the point clouds generated with PhotoScan are extremely dense and contain irrelevant background data (such as the floor far from the box), both models were trimmed to the specific area of interest: the cube in the centre of the scene.

2. **Alignment (Registration):** The models were roughly aligned manually, followed by a fine alignment using the Iterative Closest Point (ICP) algorithm. This scales and rotates the virtual model to overlay the real model perfectly.

This software will be used to compare the point clouds of both the real 3D model and the simulated POV-Ray model. Because the point clouds generated with PhotoScan are very dense, when performing a comparison between them in Cloud Compare, the point clouds will be trimmed to the interested area. The interested area is the cube in the middle of the scene. Both the 3D model and the Virtual model will be cropped to an area as small as possible but big enough to encompass all the details of the cube. Once these models have been cropped (Figure 5 below), Cloud Compare requires the used to specify similar points on both cubes so as the automatically align them and perform the necessary scaling required before the actual comparison of the models. After cropping and adjusting the 3D models from Figure 2 and Figure 4 respectively.

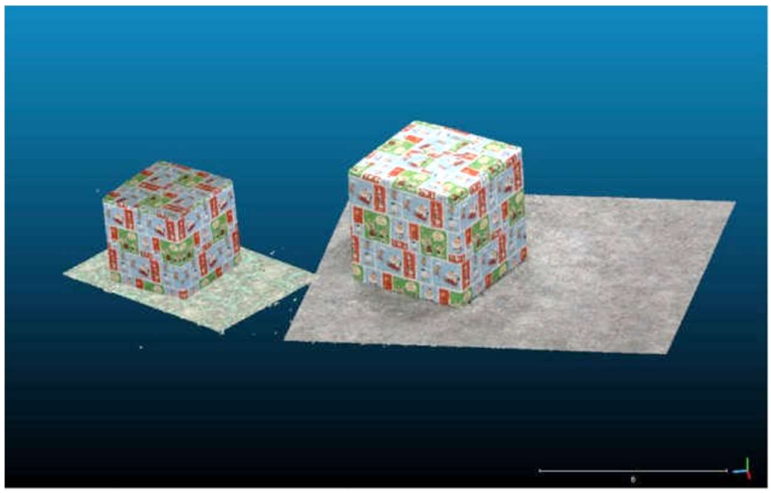

*Figure 7-The Real and Virtual Models side by side*

In Figure 7, the model on the left was generated from the virtual POV-Ray images, while the model on the right is the rendering created from real-world images.

## 3.4.1 Analysis of Disparities

The comparison result can be seen in the Figure 8 below. Cloud Compare computes, for every point in the compared cloud, the Euclidean distance to the nearest point of the reference cloud, and renders that scalar field as a colour map. The colours are therefore not a similarity judgement but a direct encoding of distance: blue denotes separations approaching zero, the green and yellow band denotes intermediate separations, and red denotes the upper limit of the scale. A region rendered red does not indicate that geometry is present in one model and absent from the other; it indicates that the nearest corresponding point lies at the maximum distance represented on the scale, which occurs both where geometry genuinely differs and where one cloud is locally sparse. The scale is relative to the range of the computed field, so colours are comparable within a single figure but not between figures computed over

different models. A perfect correspondence would render uniformly at the lower end of the scale. We can see that the blue color persists on all areas of the cube; however we also have slight differences suggested by the presence of green color. The findings, however, are quite promising as further tweaks to the simulation and acquisition of images could help create a closer identity be- tween the models and render a simulation result that is more on the blue side than any other color suggesting disparity.

The comparison result is visualized using a heat map (Figure 8). Differences in color suggest where the models diverge (Table 6).

- **Blue:** Indicates a very high likelihood of resemblance (near-zero distance between points).
- **Green/Yellow:** Indicates slight geometric deviations.
- **Red:** Suggests areas that are present in one model but absent in the other (outliers).

| Structural region | Visual colour indicator | C2C absolute distance (error range) | Analytical observation |
|---|---|---|---|
| Main cube faces (top, front, side) | Dark blue to cyan | 0.000 – 0.102 | High geometric fidelity. The primary surfaces of the model show minimal disparity, indicating successful feature matching and depth estimation across these unobstructed areas. |
| Outer base plane | Dark blue | 0.000 – 0.051 | Strong alignment. The flat planar regions extending outward from the central structure exhibit the lowest error rates on the scale. |
| Cube edges and corners | Cyan to light green | 0.102 – 0.154 | Moderate error increase. Disparities rise along the sharp transitions of the geometry, a common artefact of point cloud reconstruction where feature density falls at rigid boundaries. |
| Occlusion and intersection zone (base of cube) | Green, yellow, orange to red | 0.154 – 0.411 | Severe reconstruction failure. The region at the intersection of the cube and the plane exhibits critical geometric distortion. Maximum-scale errors of approximately 0.411 indicate substantial noise, attributable to shadowing, occlusion, or insufficient overlapping features for the SfM pipeline to triangulate this topological pocket accurately. |
| Scattered point noise | Dark blue to grey (floating) | Variable (isolated) | Minor spurious artefacts located beneath the primary plane, representing isolated false-positive feature matches not fully removed during bundle adjustment. |

*Table 6 - Quantitative Analysis of Cloud-to-Cloud (C2C) Absolute Distances and Localized Structural Disparities*

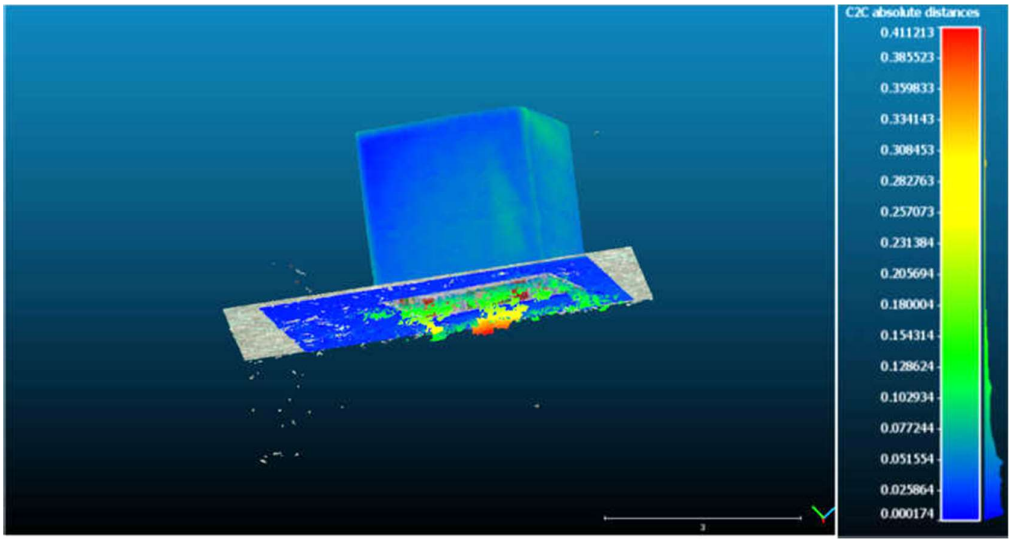


*Figure 8- CloudCompare cloud-to-cloud distance map for the cube, real model against synthetic. Blue indicates near-zero separation; the green-to-red transition at the cube-plane intersection marks the occlusion zone where errors peak at 0.411 model units. The models are viewed from beneath to expose the full colour range.*

The models in Figure 8 are shown at an orientation where they expose the full colour range. The camera is looking at an angle that is under both the real and simulated cube box. This position is of no interest other than to make the point that there are huge differences between the models if once looks at them from underneath. Blue persists across the majority of the cube surface, with intermediate green values distributed over the upper faces. Four candidate explanations for this residual deviation were considered, and the evidence bearing on each is set out below.

**Illumination model.** The laboratory was lit by fluorescent strip fittings, an extended and spatially non-uniform source, whereas the POV-Ray scene used a simplified light model. Supporting this explanation is the observation that the deviation is greatest on the upper faces, which receive the most direct illumination. Against it is the fact that photogrammetric feature matching operates on local intensity gradients, which are comparatively insensitive to slowly varying illumination.

**Surface reflectance.** The physical wrapping paper is neither perfectly matte nor perfectly Lambertian; printed inks exhibit some directional reflectance, and paper permits limited sub-surface scattering. The synthetic material used an ideal Lambertian model. This explanation is consistent with the deviation being distributed across textured faces rather than concentrated at specific points, and would be testable by rendering the same scene with a measured bidirectional reflectance model.

**Micro-geometry.** The physical paper carries creases and slight warping where it was applied to the box, so the physical surface is not exactly planar, whereas the synthetic counterpart is planar to arithmetic precision. This explanation predicts deviations distributed along the paper's fold lines and is the most readily testable of the four by visual inspection of the physical target.

**Reconstruction noise.** Photogrammetric reconstruction is not deterministic: Chapter 3.5.6 records that repeated runs on identical datasets did not always converge to the same alignment. Some portion of the deviation is therefore attributable to the reconstruction process rather than to any difference between the scenes, and a lower bound on this contribution could be established by comparing two reconstructions of the same physical dataset against one another.

The evidence available does not discriminate decisively between these explanations, and the deviation is most probably a compound of all four. The relevant conclusion for this thesis is narrower and does not depend on resolving the question: the residual deviation is confined to the intermediate band of the scale on the primary faces, and the geometry of the digital twin is therefore adequate for the operator-level experiments that follow. The comparison of a physical dataset against itself, which would establish the reconstruction noise

floor and so permit the remaining explanations to be separated, is identified in Chapter 6.4 as a straightforward extension.

The lighting in the laboratory during physical capture was complex (fluorescent strip lighting), whereas the POV-Ray lighting model was an approximation. While the geometries match, the varying shadows may have caused the photogrammetry software to interpret depth slightly differently in transition zones. Nonetheless, the findings are promising; the simulation is accurate enough to serve as a proxy for the real world.

Based on the quantitative Cloud-to-Cloud (C2C) distance analysis (Figure8), the synthetic 3D reconstruction exhibits a high degree of macroscopic resemblance to the target object. The Structure from Motion (SfM) pipeline demonstrates exceptional geometric accuracy across the primary planar faces, maintaining spatial deviations well below 0.102 units. In these well-lit, unobstructed regions, the digital twin serves as a highly reliable geometric substitute for the physical object. However, the structural resemblance degrades proportionally with topological complexity. As feature density drops at rigid boundaries, structural transitions such as sharp edges introduce moderate noise. More critically, the resemblance breaks down entirely within occluded intersection zones, where spatial errors peak at approximately 0.411 units. These localized areas suffer from severe geometric distortion and floating artifacts. Consequently, while the global geometry of the model is a strong and recognizable representation of the target, its micro structural resemblance is compromised in shadowed or intersecting regions, highlighting the specific spatial limitations of the current SfM pipeline.

## 3.5 The Room Experiment

The experiments in this section address RQ_2 and Objective 3. Their purpose is not to produce a single reconstruction but to determine, by systematic variation, which acquisition

parameters govern reconstruction fidelity and at what point additional physical data ceases to confer benefit.

Having demonstrated that a simple object (a cube) can be accurately modeled in software, the next step was to increase the environmental complexity. A full room simulation was attempted to verify if the approach scales to complex scenes relevant for navigation. This is undertaken in order to confirm that a sufficiently close software representation can be created to the real world one. This would establish that the approach extends to more complex scenarios. Should this be established, it follows that we are confident enough that any real pictures taken of any real environment can be simulated accurately in software. These images would then be passed through an ORB-SLAM system and see if they will yield similar results. Prior to this, a room will be simulated.

### 3.5.1 Experimental Setup

The test environment was a standard university office measuring 4m x 3m x 2.5m, completely emptied of furniture to provide a clean baseline. A tripod rig was constructed holding three cameras spaced 120 degrees apart to maximize the field of view. For the initial sample, the tripod was placed at 9 distinct positions. These 9 distinct positions were determined by dividing the length and the width of the room in 3. This yielded 3 lengthwise rows of 3 positions spaced apart by 80cm. This configuration yielded 9 positions × 6 exposures (per position) = 54 pictures of the room. Those 6 shots are taken at an angle difference of 60 degrees between each picture. Six exposures at 60-degree intervals were taken at each position in order to create more redundancy data so that PhotoScan would be able to put together a 3D environment that would resemble, as close as possible, the real office.

### 3.5.2 Iteration 1 – the 108 images Dataset

The initial capture strategy involved placing the tripod at 9 distinct positions arranged in a grid (spaced by 80cm). At each position, 6 shots were taken at 60-degree intervals. This was repeated at two height levels, yielding a total of:

9 positions x 6 angles x 2 heights = 108 images

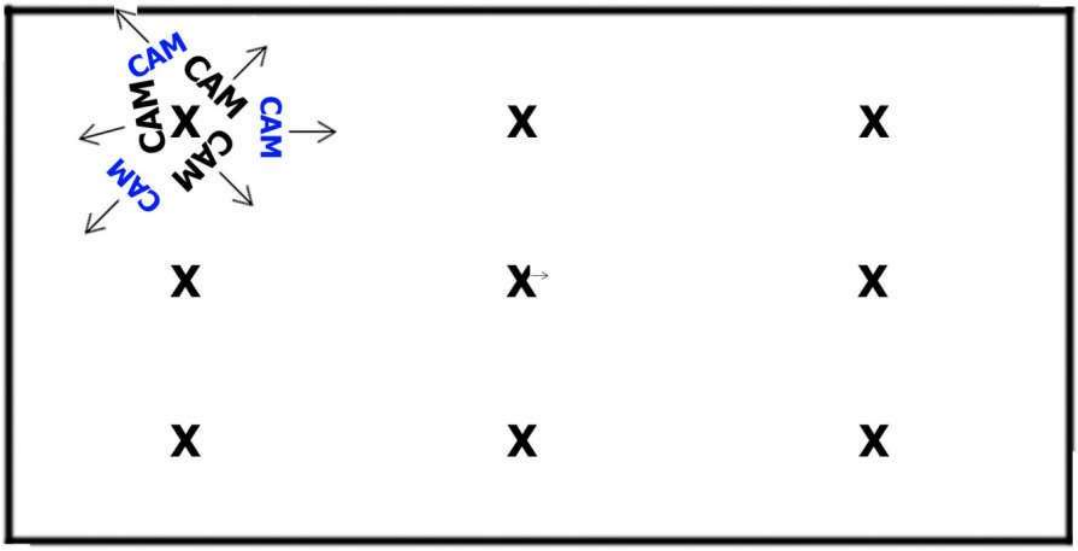


*Figure 9-Diagram of the initial capture strategy using 9 tripod positions*

When processed through PhotoScan, the resulting model was of very poor quality. The mesh was severely distorted, with large holes and unrecognizable features. It was not even evident that the model represented a room. This failure provided a critical data point: **108 images are insufficient** to capture the geometry of a feature-poor room (white walls) using Structure from Motion.

Because the simulation that yielded from this set of 108 pictures was not satisfactory, The number of positions was increased by 12, now having a total of number of 21 camera

locations. This would yield a total number of 252 pictures. A further 12 positions were added, each with 6 pictures around the tripod and with 2 height levels. The diagram of the experiment would look like this:

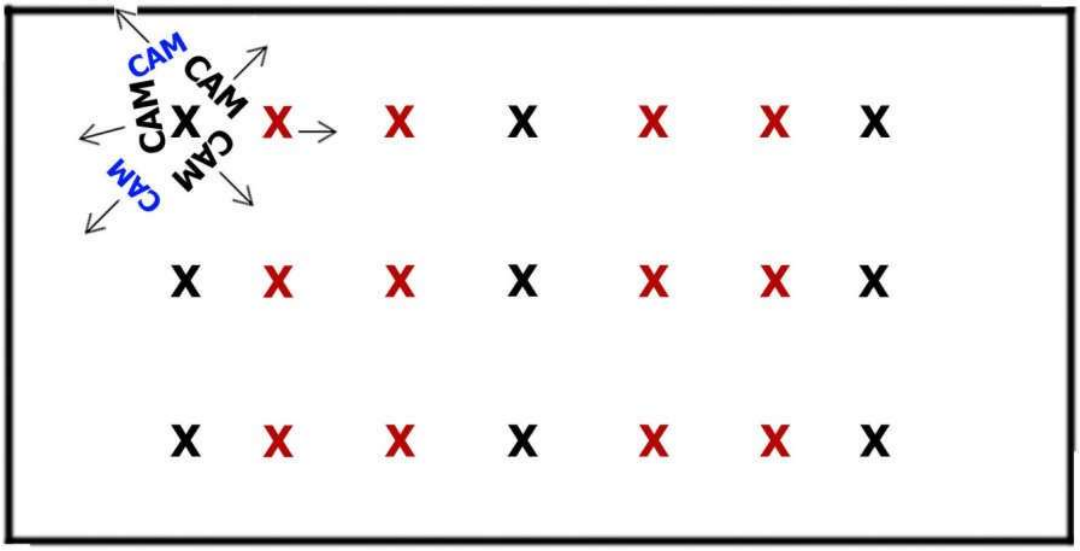


*Figure 10-The expanded strategy of using a total of 21 positions with 6 pictures per positions and on 2 levels of tripod elevation*

### 3.5.3 Iteration 2 – the 252 images Dataset

The results from this second iteration showed improvement. Elements of the room, the door, shelves, and carpet, became visible. However, significant artifacts remained.

- **The Reflective Surface Problem:** As seen in Figure 9, one area of the room appeared "broken." This corresponded to a whiteboard. PhotoScan (and SfM algorithms in general) rely on feature tracking; reflective surfaces shift their visual appearance depending on the viewing angle, violating the algorithm's assumptions and causing reconstruction failure.

- **Textureless Surfaces:** The carpet, being highly textured, reconstructed well. The white walls, lacking high-frequency detail, reconstructed poorly.

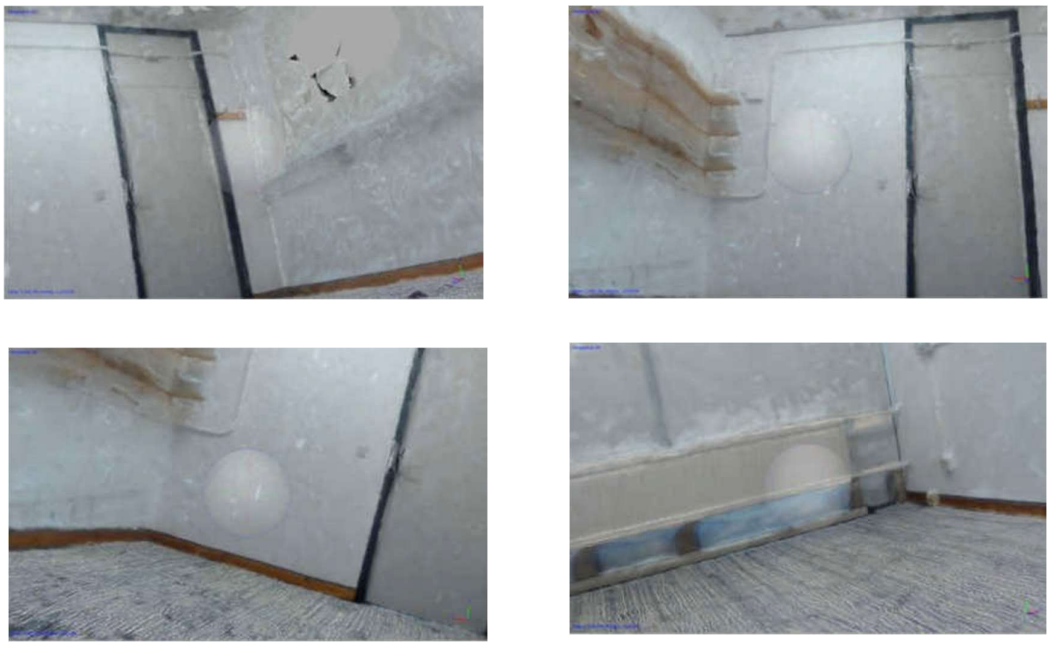

*Figure 11- Reconstruction of the test room from 252 images. The distortion at upper left corresponds to the whiteboard, whose specular surface changes appearance with viewing angle and violates the brightness-constancy assumption of feature matching. The carpet, being densely textured, reconstructs cleanly by contrast.*

Due to the suboptimal quality of the reconstruction observed in the previous iteration, a decision was made to significantly expand the dataset, operating under the hypothesis that increasing the sampling density would directly correlate with a marked improvement in rendering fidelity. Consequently, the experimental setup was modified to utilize a total of 35 distinct camera positions distributed across the floor surface. This represents a substantial addition of 14 new capture points relative to the configuration employed in the preceding

experiment. The theoretical justification for this expansion was predicated on the observation that the 252-image dataset failed to generate a coherent model. therefore, by reducing the physical distance between tripod stations and increasing the visual overlap between adjacent frames, the photogrammetric software (PhotoScan) would be provided with greater data redundancy. This increased overlap is critical for the feature matching algorithms to successfully resolve the 3D geometry of the room.

This expanded protocol resulted in a total corpus of 420 images requiring processing. It is pertinent to reiterate that throughout these experiments, the image acquisition hardware remained consistent: an SJCAM 5000X action camera equipped with a fisheye lens, capturing images at a resolution of 12 megapixels. To achieve the target of 420 images, calculated as 35 floor positions multiplied by 6 azimuthal orientations at 2 distinct vertical elevations, 14 additional tripod markers were integrated into the existing grid of 21 positions. The resulting augmented floor plan configuration is visualized in the Figure 12 below.

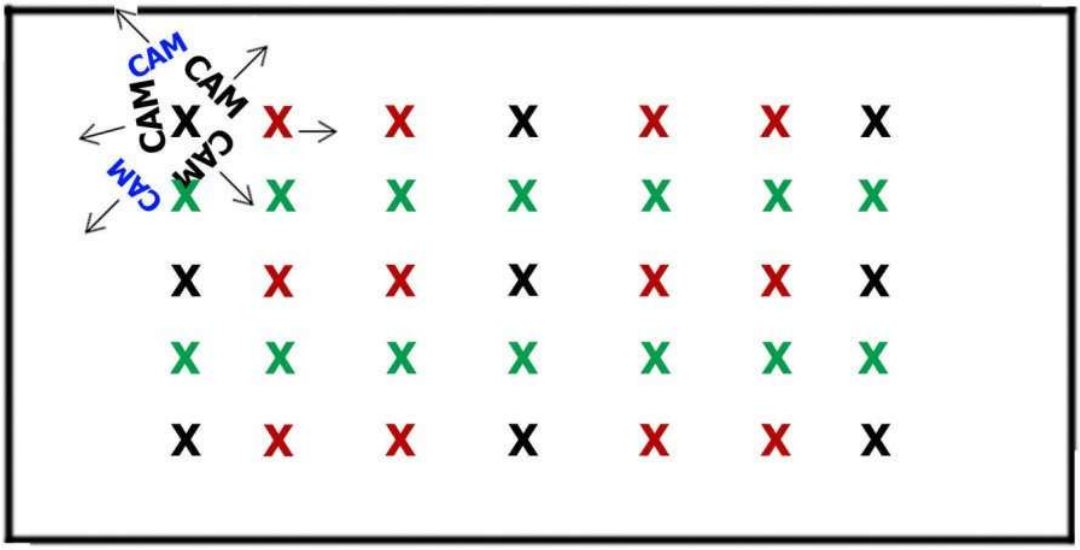


*Figure 12-A total of 35 positions with 6 pictures per positions and on 2 levels of tripod elevation.*

### 3.5.4 Iteration 3a – the 420 images Dataset

To rigorously test the impact of sensor characteristics on reconstruction fidelity, two distinct datasets of 420 images each were acquired using the aforementioned floor plan configuration. The first dataset utilized the standard SJCAM 5000X equipped with its native fisheye lens, capturing images at 12 megapixels. In parallel, a comparative dataset was acquired using a Canon DSLR camera equipped with a standard rectilinear lens, also capturing images at a matching 12-megapixel resolution. This comparative approach was specifically designed to isolate and analyze whether the specific camera model or the optical characteristics of the lens (fisheye versus rectilinear) exerts a determinative effect on the quality of the 3D room model rendered by the PhotoScan software. The Figure 13 below presents a representative set of four images illustrating the quality of the 3D model derived from the SJCAM dataset.

*Figure 13- Reconstructions from matched 420-image datasets, fisheye (SJCAM) against rectilinear (Canon DSLR). The rectilinear dataset yields markedly greater structural coherence. Note that the two datasets differ in camera body as well as lens, so the comparison isolates the optical configuration only in combination.*

Upon visual inspection of the reconstruction, it is evident that the resulting 3D model exhibits only a marginal improvement in quality relative to the previous simulation attempt. While the geometric definition of objects within the room appears slightly clearer and the surface textures are somewhat smoother, a direct consequence of doubling the density of the photographic dataset, significant artifacts remain visible throughout the scene. These persistent imperfections suggest that simply increasing the quantity of images has not yet fully resolved the reconstruction challenges, indicating that the optimal threshold for image quantity required to produce a high-quality 3D rendition has not yet been identified.

For the purpose of comparative analysis, the 3D model generated from the Canon dataset is also examined from the dataset captured with the Canon camera. Ideally, given the identical resolution, the results should be similar; however, this model appears to be substantially more coherent and structurally sound than the previous iteration using the action camera. The underlying cause for this discrepancy is not immediately definitive, but it warrants a detailed investigation into whether the optical distortion inherent to the fisheye lens is the primary factor degrading the photogrammetric process, as the resolution variable remained constant across both trials.

*Figure 14-Comparison between fish eye lenses and plain lenses*

The qualitative superiority of the rendition produced by the Canon model is marked when compared to every other instance involving the SJCAM fisheye lens. Nevertheless, even with this improvement, the overall quality of the room reconstruction does not yet seem fully acceptable when benchmarked against the very first successful example where a simple, texture-rich box was photographed. A particular point of failure remains the whiteboard; its reflective surface creates significant computational difficulties for the PhotoScan algorithm, which struggles to resolve feature points on specular materials. Conversely, the featureless white walls do not appear to pose as significant a problem once the volume of captured images is sufficiently increased. It was a primary objective to determine the upper limit at which these surface limitations cease to matter. While 420 pictures of the room appears to be a viable

threshold for acceptable quality, it is crucial to note that this result was achieved using the Canon camera, which was not the primary sensor originally planned for these experiments.

### 3.5.5 Iteration 3b: The 1260-Image Dataset

In the subsequent phase of simulation, a concerted effort will be made to augment the dataset size further, specifically employing the originally designated SJCAM camera equipped with a fisheye lens. The primary objective of this iteration is to ascertain whether a substantial increase in the volume of captured imagery will correlate with a measurable enhancement in the fidelity of the 3D model generated by PhotoScan. To facilitate this, the experimental design will retain the previously established floor configuration consisting of 35 distinct reference points distributed throughout the room. However, the protocol will be expanded by capturing an additional two exposures at each camera station. These supplementary shots will be achieved by introducing vertical angular variations: tilting the camera 45 degrees upward from its standard horizontal orientation, followed by a corresponding tilt of 45 degrees downward from the original position.

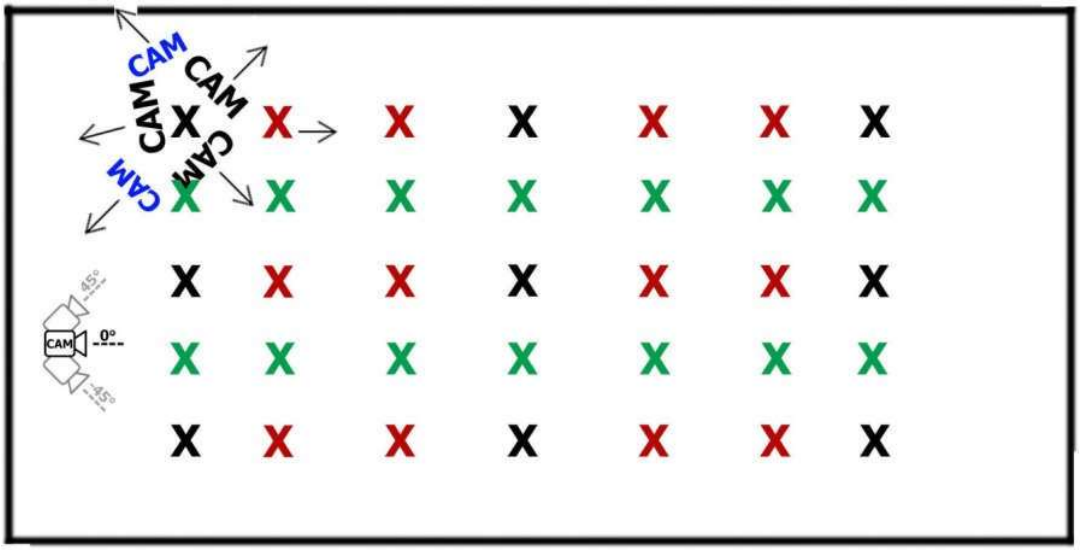


*Figure 15-Adding 45 degree angles to photo capture*

This comprehensive experimental iteration yields a dataset that represents over double the volume of imagery compared with the previous experiment. The expanded acquisition protocol now encompasses 35 distinct camera locations, multiplied by 6 azimuthal orientations, 3 separate vertical angles, and 2 distinct tripod height levels, culminating in a massive total corpus of 1260 pictures. Consistent with prior trials, these images were captured using the same SJCAM hardware operating at a resolution of 12 megapixels.

The computational result of this expansion was unexpected and highlighted a significant hardware ceiling. The photogrammetric software, PhotoScan, experiences critical failures and the software failed on every attempt to execute the reconstruction, and estimates suggest that this simulation would take in excess of 4 days to complete processing. Accumulating such a large number of pictures also impacts negatively on the system resources, specifically the random access memory of the computer. It appears that the 8 GB of memory available on the workstation does not seem to be sufficient for computing simulations involving datasets with over 1000 pictures. The initial hypothesis that capturing a higher volume of

redundant images would linearly improve geometric reconstruction was proven to be false. While a baseline minimum of data is required to avoid immediate geometric collapse (as seen in the 108-image dataset), simply saturating the SfM pipeline with redundant imagery yields diminishing returns and ultimately triggers catastrophic computational failure. Furthermore, the experiments isolated environmental and hardware constraints that cannot be overcome by brute-force data collection: feature-poor environments (white walls), non-Lambertian surfaces (whiteboards), and the optical distortion of fisheye lenses fundamentally corrupt the low-level feature extraction process. This establishes that physical data collection is not only unscalable and resource-prohibitive but also inherently flawed for identifying the precise technological boundaries of a vision system. The Table 4 below summarizes these findings.

| Iteration | Images | Camera and lens | Spatial strategy | Primary result and analytical finding |
|---|---|---|---|---|
| 1 | 108 | SJCAM (fisheye) | Sparse grid, 9 positions | **Geometric collapse.** Severe mesh distortion and unrecognisable features. Indicates that wide baselines in feature-poor environments fail to provide sufficient keypoint overlap. |
| 2 | 252 | SJCAM (fisheye) | Dense grid, 21 positions | **Marginal improvement.** Basic room elements emerge, but reconstruction fails on reflective surfaces such as the whiteboard and on textureless white walls. |
| 3a | 420 | SJCAM (fisheye) | High-density grid, 35 positions | **Persistent artefacts.** Doubling the image count provides no substantial structural gain. Fisheye distortion continues to smear features towards the frame edges. |
| 3b | 420 | Canon DSLR (rectilinear) | High-density grid, 35 positions | **Coherent geometry.** Markedly stronger structural rendering under an otherwise identical protocol, isolating lens optics as a primary failure variable and indicating that fisheye distortion degrades feature matching. |
| 4 | 1260 | SJCAM (fisheye) | Extreme density, multi-angle | **Systemic failure.** Processing exceeded the available 8 GB of memory and projected in excess of four days to complete, terminating in software failure. Demonstrates a hard computational ceiling on brute-force physical capture. |

*Table 7- Summary of Physical Data Capture Iterations and Observed SfM Failure Modes*

### 3.5.6 Enhancing the Texture

Consequently, the next avenue to explore is to introduce artificial detail onto the walls of the room in order to increase the quality of the 3D rendition through texture rather than raw data volume. Six further room positions were therefore added on top of the 9 positions of the initial configuration. It is helpful to recall that the initial configuration of 9 room positions was not helpful at all, yielding a broken and unintelligible 3D model. The diagram illustrating the configuration for the following experiment is presented in Figure 16 below.


*Figure 16-Decreasing the room positions*

For the purposes of this specific iteration of the experiment, the physical environment was maintained in its original state, characterized by unadorned white walls, thereby mirroring the initial baseline conditions exactly. A deliberate decision was made to abstain from augmenting the scene with artificial texture or added detail at this stage. The primary objective was to rigorously evaluate the baseline reconstruction quality achievable under these feature-poor conditions. The imaging hardware parameters remained consistent throughout this trial, utilizing the standard 12-megapixel sensor resolution. However, the data acquisition protocol

was significantly streamlined, resulting in a reduced total corpus of just 180 images. This quantitative reduction was derived from a specific sampling matrix: 15 distinct floor locations, multiplied by 6 azimuthal camera orientations, repeated at 2 discrete tripod elevations, with a single vertical tilt angle, culminating in the aggregate count of 180 exposures.

Qualitative analysis of the resulting reconstruction, as illustrated in the Figure 16 presented below, reveals that the model generated from the 180-image dataset exhibits superior fidelity compared to previous iterations. Surprisingly, this dataset yields a geometric reconstruction that is visually more coherent than those derived from significantly larger datasets containing 252, 420, or even the massive 1260-image corpus. This finding strongly suggests that an optimal sampling density, a theoretical 'sweet spot', has been identified, effectively balancing data sufficiency with computational coherence to produce a Digital Twin that closely approximates the real-world geometry. In retrospect, the initial attempt utilizing only 108 images proved to provide insufficient feature overlap for the PhotoScan algorithms to successfully resolve the structure. Conversely, simply inflating the dataset size does not yield a proportional increase in quality; this plateau in performance is likely attributable to the inherent limitations of the scene itself, specifically the paucity of high-frequency texture on the walls and the presence of problematic reflective surfaces. Consequently, given the satisfactory reconstruction of non-reflective surfaces and the computational efficiency of the smaller dataset, this experimental configuration has been established as the new baseline protocol. It will serve as the foundation for subsequent experimental phases where environmental complexity will be systematically introduced. Furthermore, contingent upon those findings, a parametric investigation into sensor resolution will be conducted to determine the minimum pixel density required to reproduce a 3D model that maintains high fidelity to the physical room.

*Figure 17-Increased quality of rendered room*

The subsequent series of images illustrates the photogrammetric reconstruction of the room following the strategic introduction of significant surface detail. This augmentation was achieved by applying a specific texture—patterned Christmas wrapping paper—to the walls. This material was selected not merely for its high-frequency visual content, which aids feature matching, but also because its geometric and textural properties can be replicated with high precision within the POV-Ray ray-tracing environment during the later simulation phases. The primary objective driving this current phase of experimentation is to empirically determine the minimum viable dataset size—specifically, the lowest number of photographic captures—required to generate a 3D model via Agisoft PhotoScan that maintains a statistically significant degree of accuracy relative to the physical world.

Upon successfully establishing this lower bound for image quantity, the experimental focus will shift towards an investigation of sensor resolution. The aim is to identify the lowest possible pixel density that still yields an acceptable 3D reconstruction when rendered through the POV-Ray simulation engine. Achieving these optimization milestones will serve as the foundation for the final construction of the comprehensive Digital Twin. At that stage, an accurate virtual model of the room will be generated in POV-Ray, configured to produce a synthetic dataset that mirrors the real-world dataset exactly in terms of both image count and sensor resolution. By processing these computer-generated images through the same PhotoScan pipeline used for the physical data, the expectation is to derive a synthetic 3D model that serves as a mathematically accurate representation of the model derived from the physical camera. However, prior to reaching that stage of final validation, it is necessary to analyze the immediate impact of texture. The set of four images presented in Figure 18 below visually demonstrates the marked increase in rendering quality and geometric coherence resulting from the addition of high-frequency detail to the previously featureless walls.

*Figure 18- Reconstruction of the room following application of patterned surface texture, from 180 images. Alignment success rises from below 30% to above 95%, and dense cloud density approximately doubles relative to the untextured condition, identifying surface texture rather than image volume as the governing variable.*

The comparative analysis of the reconstruction results indicates a critical dependency on surface texture. Even with a lower image count (180 vs 252), the quality of the reconstruction was superior, confirming that feature quality is more important than image quantity. In the datasets acquired without artificial wall detail, the photogrammetric software, PhotoScan, failed to register a significant subset of the images, resulting in incomplete alignment. Conversely, the dataset augmented with the Christmas paper pattern achieved a 100% alignment success rate. Furthermore, the geometric density of the resulting point cloud appeared to be approximately double that of the untextured model, confirming the value of high-frequency visual features. Both 180-image models were processed utilizing 'Medium' quality settings; this parameter choice was necessitated by hardware constraints, as it drastically reduced the computational duration from an estimated 5 days down to

approximately 1.5 days on the available laptop (Intel Core i5 CPU @ 2500 MHz with 8 GB of RAM).

The selection of this particular surface treatment warrants explanation, since it determines the outcome of every subsequent room experiment. Three properties were required. First, the texture had to be dense in high spatial frequencies, since it is the local intensity gradient that supplies the keypoints on which feature matching depends, and the untextured walls of the baseline condition had been shown to provide too few. Second, it had to be non-repeating at the scale of the capture baseline; a strictly periodic pattern would present locally identical neighbourhoods at different spatial positions, inducing false correspondences and a systematically incorrect reconstruction, which is a more damaging failure than the sparse-feature failure it was intended to remedy. Third, and decisively for this research, it had to be reproducible within POV-Ray with sufficient accuracy that the synthetic room presented equivalent features, which favoured a flat printed pattern of known geometry over a three-dimensional or specular surface treatment. Patterned wrapping paper satisfied all three conditions and was readily available in the quantity required to cover a room.

The limitations of this choice should be stated. No systematic comparison of alternative textures was undertaken, and the experiments therefore establish that a high-frequency aperiodic texture is sufficient for reliable reconstruction in this environment, not that it is optimal. A controlled comparison across textures differing in spatial frequency content, contrast and directional anisotropy would permit the relationship between texture statistics and reconstruction density to be characterised quantitatively rather than demonstrated by a single instance. Such a comparison would also address a more consequential limitation: the textured condition is favourable in a way that many real environments are not, and the acquisition protocol derived from it in Chapter 3.5.9 should be understood as applying to environments of

comparable feature density. Corridors, painted interiors and other feature-poor spaces of practical interest remain subject to the failure documented in the baseline condition, and the finding that texture dominates image volume is best read as identifying where the binding constraint lies rather than as removing it.

In an effort to maximize fidelity, a subsequent trial was conducted in which PhotoScan was configured to process the imagery at 'Maximum' quality settings. However, after nearly a full week of computational processing, the resulting model did not exhibit a statistically significant improvement in geometric quality, suggesting a curve of diminishing returns where higher settings do not necessarily yield a superior model. Additionally, a degree of stochastic instability was observed; consecutive simulations utilizing identical datasets did not always yield identical alignment results—occasionally succeeding and other times failing. The underlying cause of this nondeterministic behavior is not immediately apparent and remains a subject for further investigation.

### 3.5.7 The Impact of Surface Detail on Photogrammetric Accuracy

The empirical data collected throughout the initial experimental phases confirmed a fundamental limitation of Structure from Motion (SfM) algorithms within feature poor environments. Across the majority of the early iterations featuring unadorned white walls, Agisoft PhotoScan consistently struggled to align the spatial data, yielding poor reconstruction results with reported accuracies falling below 30%. While this algorithmic failure was anticipated due to the lack of trackable visual gradients, rigorously documenting this baseline was a necessary step to empirically validate the system's failure modes.

A critical inflection point was reached the moment high-frequency surface detail (specifically the patterned Christmas wrapping paper) was introduced to the physical environment. The presence of these trackable features immediately eliminated the need for massive, computationally prohibitive datasets. By utilizing the optimized, smaller corpus of just 180 images, the added surface detail provided the photogrammetric pipeline with the precise anchors needed to successfully resolve the geometry. This resulted in a dramatic leap in performance, with PhotoScan yielding a reported reconstruction accuracy exceeding 95%.

These finding isolates surface texture as the primary driver of photogrammetric success, indicating that reconstruction accuracy approaches the practical limit of the pipeline. The immediate next step in this research is to leverage these established parameters (the optimized 180-image spatial overlap combined with rich surface detail), to further refine the capture methodology and push the accuracy figure even closer toward 100%.

| Experimental phase | Environmental condition | Images | Alignment success | Analytical conclusion |
|---|---|---|---|---|
| Initial iterations (aggregated) | Untextured white walls | 108, 252, 420, 1260 | < 30% | **Data starvation.** Irrespective of image volume, the absence of trackable features prevents the algorithm from estimating camera poses and triangulating structure. |
| Optimised baseline | Untextured white walls | 180 | ≈ 30% | **Algorithmic limit.** Represents the best spatial overlap obtainable in this environment, but remains bounded by the feature-poor surfaces rather than by sampling. |
| Textured milestone | Patterned paper applied | 180 | > 95% | **High fidelity obtained.** The added high-frequency detail allows a smaller dataset to be used to full effect, yielding an accurate structural reconstruction. |

Table 8 -The Impact of Surface Texture and Optimized Sampling on SfM Reconstruction

### 3.5.8 Finding the sweet spot (Quantitative Determination)

Having established a baseline where the 3D model is visually acceptable, the experimental design will now advance to introduce greater angular diversity, mirroring the methodology employed in previous experiments, but this time leveraging the benefits of the added texture. Consequently, the next phase involves introducing vertical angular variations at each tripod level. This protocol dictates that each elevation level will capture not merely 6 images, but 18 images to significantly increase feature overlap and detail.

The resulting dataset calculation is as follows: 15 floor positions multiplied by 2 tripod levels, multiplied by 18 pictures per level (derived from 3 discrete angles: 0 degrees, +45 degrees and -45 degrees, yielding a total corpus of 540 pictures. It is hypothesized that this increased redundancy will stabilize the simulation results. Should this approach fail to yield

improvement, a contingency plan is established to introduce additional Christmas paper coverage to the room and revert to a higher density of 18 floor positions. The core objective of this phase is to utilize a higher quantity of images to forcibly extract greater detail from the scene, as the room remains relatively feature-sparse despite the current texture augmentation. At this juncture, the experiment is testing the upper bound of data utility to ascertain whether 540 images offer a measurable advantage over 180 images. This is a critical distinction from prior attempts; previously, without wall texture, increasing the image count provided no benefit. Now, with the presence of redundant angular information and surface texture, the hypothesis is that the photogrammetric reconstruction will finally be able to leverage the larger dataset for higher quality.


*Figure 19-Adding angles to the simulation*

Following an extensive computational processing period spanning approximately one week using the Agisoft PhotoScan platform, the resulting 3D model was successfully generated and is presented below. It is imperative to underscore a critical experimental constant maintained throughout this phase: all source imagery was acquired at a fixed sensor resolution

of 12 megapixels to isolate sampling density as the primary variable. The qualitative results of this reconstruction significantly exceeded initial expectations regarding geometric fidelity. The generated virtual environment serves as a fairly accurate representation of the physical real-world room, suggesting that the methodology has successfully converged on the optimal data acquisition strategy. Consequently, it can be posited that the 'sweet spot' regarding the necessary quantity of photographic samples has been identified, specifically within the context of the fixed 12-megapixel resolution constraint. Quantitatively, the PhotoScan algorithm demonstrated robust performance, successfully triangulating features to generate a dense point cloud comprising approximately 199,000 individual points, a density that provides sufficient resolution for the subsequent simulation steps.

*Figure 20-Simulation results with increased detail (8 megapixel)*

The qualitative outcomes of this reconstruction phase exceeded initial expectations. The resulting 3D mesh provides a highly accurate geometric representation of the physical

environment, allowing for the conclusion that an optimal sampling density—a theoretical 'sweet spot'—has been successfully identified within the constraints of the 12-megapixel sensor resolution. Quantitatively, the Agisoft PhotoScan algorithm demonstrated robust feature matching capabilities, generating a dense point cloud comprising approximately 199,000 individual data points. Building upon this baseline, the experimental focus shifted towards optimization; the subsequent phase of investigation aimed to ascertain whether the high fidelity of the 3D simulation could be maintained while utilizing input imagery of a lower pixel resolution to reduce computational overhead.

Following a computational processing period spanning several days, the results of the 8-megapixel trial were analysed. This dataset maintained the identical capture protocol, 540 images distributed across 15 positions, 2 tripod elevations, and 3 angular orientations, but utilized down sampled imagery. The resulting reconstruction exhibited a remarkable degree of similarity to the benchmark model derived from the 12-megapixel scenario. Visual inspection revealed that distinguishing between the two meshes is perceptibly difficult, indicating negligible degradation in geometric structure. Quantitatively, PhotoScan successfully generated a dense point cloud containing approximately 198,000 points; this minor reduction indicates that the loss in raw pixel count did not significantly impact the feature extraction process. In light of these findings, it is empirically defensible to assert that acquiring imagery at 8 megapixels offers a superior efficiency-to-quality ratio compared to the full 12-megapixel format, primarily due to the reduction in storage requirements and processing duration.

To rigorously define the lower bound of this resolution parameter, a further experiment was conducted by reducing the input resolution to 5 megapixels. This trial utilized the exact

same corpus of 540 source photographs to ensure consistency. However, unlike the previous iteration, the results observed here displayed a marked deterioration in fidelity compared to the two preceding simulations. The resulting dense point cloud suffered a significant reduction in density, manifesting as a qualitative loss exceeding 25% relative to the 8-megapixel benchmark. Specifically, the PhotoScan algorithm was only able to resolve approximately 140,000 points within the cloud. Unlike the subtle differences observed previously, the degradation in quality at this resolution is visually distinct and immediately apparent in the rendered images presented below (Fgure 21).

*Figure 21- Reconstruction from a 5-megapixel dataset (540 images). Dense cloud density falls to approximately 140,000 points, a 25% reduction against the 8-megapixel baseline. The four panels show the resulting geometric artefacts, warped boundaries and surface irregularities, demonstrating that 5 megapixels falls below the threshold required for structural coherence.*

A direct comparative analysis between the 5-megapixel simulation and the 8-megapixel benchmark reveals distinct visual degradations. The 5-megapixel dataset exhibits several geometric artifacts and surface irregularities that are notably absent in the higher-resolution model. This observation serves as empirical evidence that a 25% reduction in the density of the point cloud is sufficient to manifest as a perceptible visual difference between simulations. Quantitatively, it is noteworthy to observe a non-linear relationship between sensor resolution and reconstruction density: while a 5-megapixel image contains approximately 37% fewer raw pixels than an 8-megapixel image, the resulting photogrammetric dense cloud decreases by a disproportionately lower margin of 25%. To rigorously establish the lower bound of usability, a further experiment was conducted utilizing extreme down sampling to 2 megapixels. The quantitative impact was catastrophic: the resulting dense cloud contained merely 17,000 points, representing a precipitous 90% reduction compared even to the suboptimal 5-megapixel dataset. Qualitatively, the 3D mesh failed to retain structural coherence; the geometry no longer resembled a room, rendering the model effectively unintelligible for simulation purposes as seen from the following image.

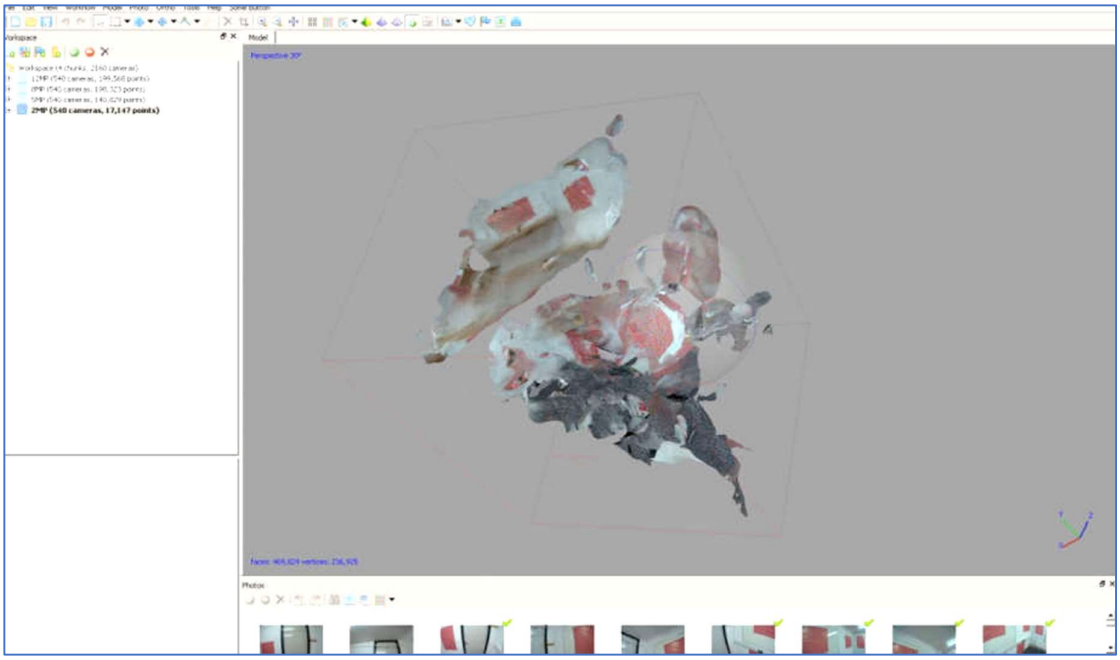

*Figure 22- Reconstruction from a 2-megapixel dataset (540 images). Dense cloud density collapses to approximately 17,000 points, a 90% reduction against the 5-megapixel condition. Structural coherence is lost entirely and the geometry no longer resembles a room.*

With the texture problem solved, an investigation into image resolution was conducted to optimize processing time. The 1260-image dataset (captured with the SJCAM 5000X) was downsampled to various resolutions and then processed.

- **12MP:** High quality, but slow processing.
- **8MP:** The dense point cloud contained ~198,000 points. Visually, there was negligible loss in quality compared to 12MP, but processing speed improved significantly.
- **5MP:** The dense cloud dropped to 140,000 points (a 25% loss). Visual artifacts also began to appear (see Figure 20).
- **2MP:** The cloud dropped to 17,000 points (a 90% loss). The model became unintelligible (Figure 21).

Based on this extensive series of trials, a stable and robust protocol for future simulations has been established. The findings dictate that the physical environment must contain maximal surface detail to facilitate feature matching. Furthermore, the optimal acquisition geometry requires 15 distinct floor positions. At each position, the camera must capture 6 azimuthal angles (spaced at 60-degree intervals) and 3 vertical inclination angles, repeated at 2 distinct tripod elevations to ensure coverage of occluded areas such as high shelves. This cumulative configuration results in a standardized dataset of 540 images. Regarding sensor resolution, the investigation into the minimum viable pixel density has yielded a definitive constraint.

It was determined that increasing resolution beyond 8 megapixels offers negligible gains in simulation fidelity while incurring a prohibitive computational cost; processing times on the available hardware (Intel Core i5 @ 2.4GHz, 8GB RAM) escalated exponentially to durations spanning several weeks or even a month. Conversely, utilizing resolutions below the 8-megapixel threshold results in a significant degradation of geometric integrity, to the extent that the Digital Twin ceases to act as a valid proxy for the real-world environment. Consequently, it is concluded that 8 megapixels represents the optimal 'sweet spot' for this methodology, balancing visual fidelity with computational feasibility within the context of the 540-image dataset. For visual reference, the degradation observed in the 2-megapixel model is illustrated in the Figure 21, alongside a screenshot tabulating the quantitative dense cloud point counts across the various simulation parameters.

### 3.5.9 Establishing the Methodological "Sweet Spot"

In conclusion, the systematic progression of these experimental trials has successfully yielded a highly optimized protocol for photogrammetric reconstruction, establishing an empirically supported optimum required to generate a high-fidelity Digital Twin. By iteratively isolating and testing variables (ranging from environmental texture and spatial baselines to angular redundancy and sensor resolution) this research has transformed a highly unstable initial process into a predictable, robust methodology. The empirical data demonstrates that achieving geometric accuracy is not a matter of blindly maximizing data volume, but rather finding the precise equilibrium between optical input and hardware constraints. The finalized baseline dictates that the environment must possess maximal surface detail, captured through a rigorous, multi-angular 540-image spatial matrix. Crucially, calibrating the sensor resolution to 8 megapixels provides the exact pixel density needed to maintain exceptional structural integrity while circumventing the catastrophic processing bottlenecks observed at higher resolutions. This rigorously defined parameter set now serves as the validated mathematical blueprint for the subsequent ray-tracing simulation phase.

| Sensor resolution | Dense cloud size | Geometric fidelity | Analytical conclusion |
|---|---|---|---|
| 12 megapixels | ≈ 199,000 points | Benchmark | **Highest quality.** Excellent structural coherence, but computationally prohibitive, requiring approximately one week of processing on the available hardware. |
| 8 megapixels | ≈ 198,000 points | Equivalent to benchmark | **The optimum.** Point loss of approximately 0.5% relative to the 12-megapixel model, with no perceptible difference on inspection. Offers the best ratio of accuracy to computational cost. |
| 5 megapixels | ≈ 140,000 points | Moderately degraded | **The usability boundary.** A 25% reduction in point density. Surface irregularities and artefacts become visible and begin to compromise structural integrity. |
| 2 megapixels | ≈ 17,000 points | Structural collapse | **Unintelligible.** A 90% reduction in point density. Insufficient pixel data remains for the wall pattern to be tracked, and the mesh loses coherence entirely. |

*Table 9- The Impact of Sensor Resolution on Point Cloud Density and Reconstruction Fidelity (540-Image Textured Dataset)*

## 3.6 Final conclusions

The optimal parameters were established as: **8MP resolution** and **540 images** (with +45 degrees and -45 degrees of vertical angles to capture shelf details). A final Virtual Room was constructed in POV-Ray using these parameters to match the Real Room. The resulting virtual model was compared against the real model using Cloud Compare.

For the purposes of this specific comparative trial, the Polygon File Format (.PLY) was selected as the standardized data container for both the physically derived and synthetically generated 3D models. This specific file specification was explicitly chosen during the export

phase from the Agisoft PhotoScan software suite due to its capability to robustly store dense point cloud geometry. The preliminary stage of the analysis involved a rigorous segmentation process, wherein the raw point clouds were cropped to isolate the specific region of interest (the central cube) thereby removing extraneous environmental noise, as visually depicted in Figure 20. Furthermore, due to the inherent scale ambiguity often present in photogrammetric reconstructions, the two models initially manifested with differing spatial dimensions. Consequently, it became a prerequisite to apply a uniform scaling transformation to normalize the geometry, ensuring that both models possessed identical physical proportions prior to the quantitative analysis. The outcomes of these pre-processing steps are illustrated in the Figure 22 below."

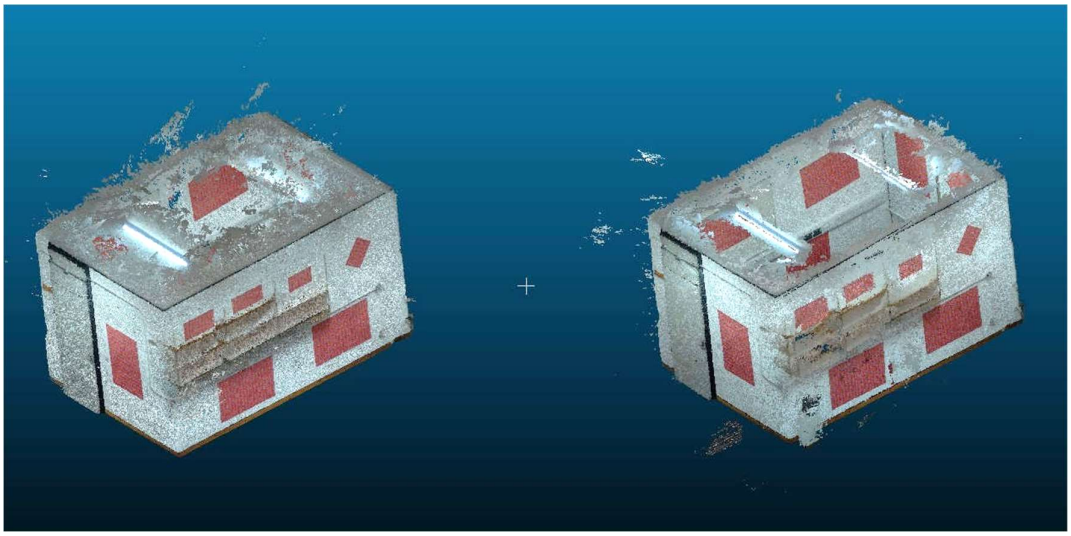

*Figure 23-Virtual and Real model side by side, as a cloud mesh in Cloud Compare*

In the visual comparison presented, the photogrammetrically reconstructed model derived from real-world imagery is situated on the left-hand side, while its synthetic

counterpart, generated via the virtual simulation pipeline, is positioned on the right. A preliminary visual inspection suggests a high degree of morphological similarity; to the naked eye, the geometric details appear nearly identical. However, subjective visual assessment is insufficient for scientific validation. Consequently, the specialized software suite CloudCompare was employed to perform a rigorous quantitative analysis to determine the precise mathematical degree of equality between the two meshes. To facilitate this comparative analysis, a precise geometric alignment was required. This was achieved by executing a registration process using the 'Register entities' toolset within CloudCompare.

The registration protocol established the real-world model as the immutable 'Reference' entity, serving as the ground truth against which the simulation would be judged. Conversely, the virtual model was designated as the 'Aligned' entity, allowing it to be transformed spatially to match the reference. Upon the successful completion of the registration iteration, the software generated a transformation matrix and an error report. The resulting quantitative data obtained from this alignment process is detailed below (Figure 24).

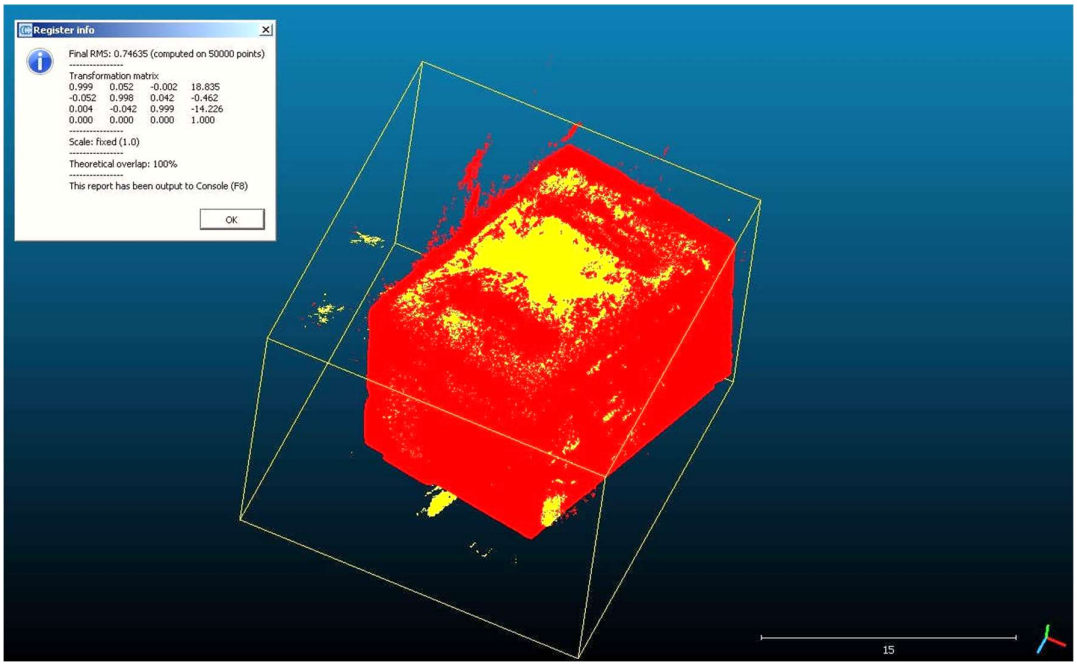


*Figure 24-Overlapping models*

The quantitative assessment of the registration phase revealed a Root Mean Square (RMS) error value below unity, a metric that statistically confirms a tight geometric fit between the two datasets. In the preliminary alignment visualization, the presence of red coloration highlights specific localized regions where the topology of the simulated model deviates from the physical reality, likely due to limitations in the POV-Ray shading model. Following this successful registration, the analysis proceeded to the dense geometric comparison phase using the 'Compute cloud/mesh distance' algorithm within Cloud Compare.

Consistent with the previous protocol, the real-world model was strictly designated as the 'Reference' mesh, serving as the ground truth, while the virtual model was assigned as the entity to be compared. The computation yielded a detailed distance field, visualized in the Figure 24 presented below. The interpretation of these results relies on a chromatic heatmap scale: regions rendered in the blue spectrum indicate areas of high geometric fidelity, where

the Euclidean distance between the real and virtual surfaces approaches zero. Conversely, the transition towards the red spectrum denotes areas of substantial geometric deviation. Consequently, the overarching objective is to achieve a model that is predominantly blue; such a result provides empirical validation that the virtual environment possesses sufficient fidelity to be utilized as a reliable surrogate for real-world data acquisition, thereby confirming the viability of sourcing training data from simulation rather than expensive physical capture.

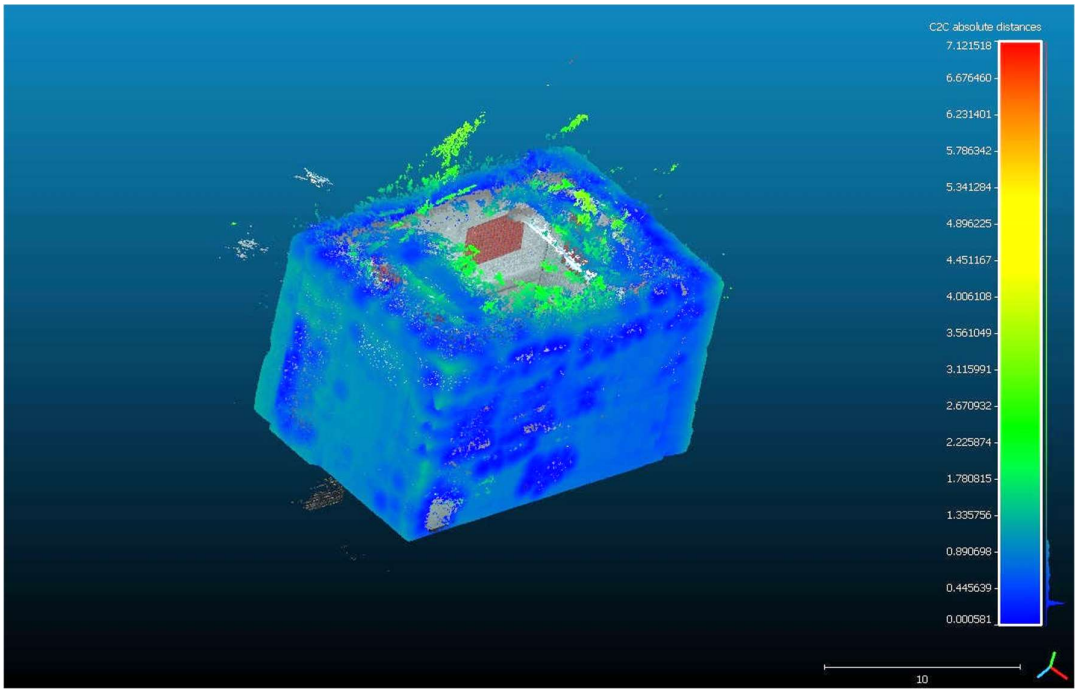


*Figure 25- Cloud-to-cloud absolute distance between the physical and synthetic room reconstructions. The dominant blue spectrum across walls and floor indicates agreement within the lowest error bracket; green and yellow deviations are confined to intersection zones and upper boundaries where feature density thins. Final RMS error 0.74635 model units.*

The chromatic analysis of the distance field provides a direct visual metric for geometric congruence. Specifically, the color gradient delineates the magnitude of spatial deviation between the two datasets: regions rendered in red indicate significant topological discrepancies, representing features present in one model but absent or spatially misaligned in

the other. Conversely, the spectrum transitioning through green and settling into blue signifies a high degree of resemblance, with solid blue denoting near-perfect spatial alignment where the Euclidean distance between corresponding points approaches zero. Observational analysis of the simulation results reveals a pervasive dominance of the blue spectrum across the surface area of the virtual room. This consistency mirrors the success observed in the preliminary experiments involving the simple geometric cube, thereby confirming that the simulation methodology scales effectively from simple objects to complex interior environments.

Based on these quantitative and qualitative results, it is empirically demonstrated that the simulation pipeline can be iteratively refined generate a 3D model that agrees with one derived from physical data to within the measurement tolerance of the reconstruction process. This confirms the validity of the 'Digital Twin' concept: virtual scenarios generated via raytracing are capable of producing synthetic sensor data that matches the geometric fidelity of a physical camera capture. Consequently, the first experimental milestone has been successfully achieved, establishing that, for the environments examined, a high-fidelity simulation can serve as a geometrically accurate substitute for physical data acquisition.

## 3.7 Validation of the Digital Twin via Cloud-to-Cloud Comparison

This section provides the formal test of hypothesis H1 and completes the answer to RQ_1, applying to the full room the same quantitative comparison previously applied to the cube in Chapter 3.4

Following the successful establishment of the optimized physical capture protocol (yielding ~98% accuracy), the focus shifted to the core thesis objective: validating the synthetic "Digital Twin." A mathematically precise virtual replica of the textured room was constructed

within the POV-Ray simulation environment. Utilizing the exact spatial coordinates (15 floor positions, 18 angular variations, 2 elevations) and the optimized sensor resolution (8 Megapixels) derived from the physical trials, a synthetic dataset of 540 images was rendered and subsequently processed through Agisoft PhotoScan.

To rigorously quantify the geometric parity between the physical reality and the synthetic simulation, the two resulting 3D point clouds were imported into CloudCompare software for spatial analysis. Initial visual inspection of the side-by-side models (Figure 24) revealed a striking structural resemblance. When mathematically aligned, the global registration displayed overwhelming spatial overlap, indicated by the dominant red and yellow colour mapping across the primary structural volumes.

Crucially, the Cloud-to-Cloud (C2C) absolute distance computation (Figure 25) provided the definitive quantitative assessment. The resulting heat map indicates that the vast majority of the reconstructed surfaces (the walls, the floor, and the primary structural planes) fall within the lowest error bracket (0.000 to 0.890 spatial units), represented by the dark blue spectrum. Minor deviations (green/yellow spectrum) are localized strictly to complex intersection zones and upper boundaries where feature density naturally thins. The calculated Final RMS (Root Mean Square) error of 0.74635 (Figure 24) across the computed sample points mathematically confirms the high degree of structural correlation This low variance supports the conclusion that, for this environment and acquisition protocol, the POV-Ray generated dataset serves as an accurate substitute for real-world photogrammetric data.

The Table 7 below details the statistical output from the Cloud Compare analysis, illustrating the precise alignment and minimal spatial deviation between the physical and synthetic models.

| Assessment metric | Observed value | Analytical conclusion |
|---|---|---|
| Global registration | Converged on all sampled points | The registration algorithm matched the macro-geometry of both point clouds, indicating that the spatial layout of the synthetic model corresponds to that of the physical room. |
| Final RMS error | 0.74635 model units | The mean spatial separation between corresponding points in the physical and synthetic models is low relative to the dimensions of the reconstructed volume. |
| Primary planar error (walls, floor) | 0.000 – 0.890 model units | The greater part of the structural volume falls within the lowest error bracket, indicating that the synthetic textures supported accurate feature triangulation. |
| Localised disparities | 1.780 – 4.006 model units | Deviations are restricted to extreme edges and untextured boundary zones, consistent with the known limitations of the SfM pipeline rather than with failure of the simulation. |

*Table 10 - Cloud-to-Cloud (C2C) Spatial Analysis: Physical vs. Synthetic 3D Reconstruction*

# 3.8 Conclusion

Building upon this validated baseline, the subsequent phase of research will focus on 'Predictive Stress Testing.' This involves the systematic introduction of controlled artifacts into the simulation to model sensor degradation and environmental variance. Specific variables to be manipulated include resolution down sampling (loss of picture quality), the introduction of specular reflections, radiometric changes in lighting conditions, and geometric alterations such as image warping and radial lens distortion. The working hypothesis is that by implementing these specific degradation models on the virtual camera, the resulting topology may be

predicted. Effectively, the simulation indicates how the reconstruction will fail before any physical capture is undertaken.

To validate this predictive capability, the final step involves replicating these exact environmental stressors in the physical world, for example, by capturing data in low-light conditions or using a lower-quality lens. The resulting real-world data will be processed to generate a 3D model, which will then be compared against the 'predicted' model generated by the degraded simulation. If the two models exhibit identical failure modes and geometric artifacts, a substantive result will have been demonstrated: that the simulation is accurate enough not just to model the world, but to predict the performance boundaries of vision algorithms within it. This would provide strong evidence that synthetic data is a valid medium for training and validating computer vision systems, offering a robust alternative to the costly and uncontrollable nature of real-world data collection.

# Chapter 4: Technology Evaluation: Characterizing Low-Level Feature Detectors

## 4.1 Introduction: From Macro-Systems to Atomic Components

This chapter addresses RQ_3 and RQ_4, and tests hypothesis H2. Having established in Chapter 3 that a virtual environment can reproduce physical geometry within the tolerance of the reconstruction pipeline, the investigation now descends from the level of complete systems to that of the individual operator. Section 4.2 to 4.4 characterise the corner detector's operational envelope entirely within simulation, answering RQ_3. Section 4.5 tests that characterisation against imagery from physical cameras, answering RQ_4. Section 4.6 identifies the additional optical property that must be matched for the correspondence to hold, opening the question pursued as RQ_5 in Chapter 5.

The initial research objective was to utilize a comprehensive Visual Simultaneous Localization and Mapping (vSLAM) framework to empirically demonstrate that environmental variations within a simulation impact the vision system in a manner statistically identical to those in the physical world. However, a strategic pivot was deemed necessary to isolate the fundamental mechanics of visual navigation, specifically, the concept of a robot navigating solely based on feature recognition. This domain presents a more scientifically rigorous avenue for exploration. Rather than attempting to validate a complex, 'black box' system like SLAM

immediately, the research methodology focuses on the atomic components of vision: simple feature detection. Isolating a fundamental operator permits a more granular analysis. For instance, by observing a simple geometric primitive, such as a cube, one can systematically vary the illumination angle and intensity to quantify precisely how these variables degrade the accuracy of the corner detection algorithm. The OpenCV library provides robust implementations of these standard detectors, which serve as the testing baseline for this purpose.

Following this preliminary characterization, the experimental design mirrors this procedure across both the simulated and physical domains. The methodology involves the systematic degradation of image quality, specifically, reducing the lighting quality or introducing synthetic grain to mimic high-ISO sensor noise, in both the simulation and the real world. Alternatively, post-processing techniques can be applied to inject controlled noise into clean images, allowing for a precise measurement of how such artifacts influence the stability of corner detection. Subsequent to detection, the ORB descriptor is employed to match these corners. This facilitates a detailed sensitivity analysis, determining how robust these descriptors are when subjected to noise, varying angles of incidence, and directional lighting changes. This reductionist approach avoids the computational and logistical overhead of debugging a monolithic system like ORB-SLAM, which requires immense rendering resources and complex parameter tuning. By starting with a simplified, micro-level analysis, the computational demands are minimized, allowing for rapid iteration and a deeper statistical understanding of the failure modes.

### 4.1.1 The "Datasheet" Concept for Computer Vision

Corner detection serves as a critical pre-processing step for the vast majority of modern computer vision algorithms, including SLAM. Consequently, the primary aim of this chapter is to construct a rigorous performance specification, effectively a 'Datasheet' for the corner detection operator, analogous to the electrical characteristics datasheet one would find for a transistor or logic gate. This document will mathematically describe the operational envelope of the corner detector, detailing its response curve to specific variables such as radiometric resolution (lighting), signal-to-noise ratio (visual noise), and geometric distortion. The objective is to establish a predictive model permitting the forecast fof how the corner detector will perform in a stochastic real-world scenario based solely on its behaviour in the deterministic simulation. Ultimately, the goal is to refine the simulation parameters until the virtual behaviour is isomorphic to the real-world behaviour. Achieving this convergence would provide substantive evidence that virtual imagery constitutes a valid representation of reality for the purposes of operator evaluation, within the range of conditions tested.

### 4.1.3 Signal-to-Noise Ratio (SNR) as a Quantifiable Metric

Signal-to-noise ratio is used throughout this thesis as the primary scalar descriptor of image quality, and is defined here formally.

Let an observed image be modelled as the sum of a deterministic signal and an additive noise field,

$$g(x, y) = s(x, y) + n(x, y) \qquad \textit{(Equation 5)}$$

where $s$ is the noise-free image that an ideal sensor would record and $n$ is a zero-mean random field, independent of $s$, with variance $\sigma^2_n$. Signal-to-noise ratio is defined as the ratio of the standard deviations of the two components,

$$SNR = \sigma_s / \sigma_n \qquad \textit{(Equation 6)}$$

and is therefore a dimensionless amplitude ratio, not a logarithmic quantity. Values are not expressed in decibels at any point in this thesis.

Because $s$ is not directly observable, $\sigma_s$ and $\sigma_n$ cannot be measured from a single image without assuming a noise model. The estimator adopted here instead uses two observations of the same scene, following the two-image correlation method. Let

$$g_1 = s + n_1,\ g_2 = s + n_2 \qquad \textit{(Equation 7)}$$

where $n_1$ and $n_2$ are independent realisations drawn from the same distribution. Since the noise fields are mutually independent and independent of the signal, their covariance contributes nothing to the cross-covariance of the two observations, giving

$$cov(g_1, g_2) = \sigma_s^2,\ var(g_1) = var(g_2) = \sigma_s^2 + \sigma \qquad \textit{(Equation 8)}$$

The Pearson correlation coefficient between the two observations is therefore

$$r = \sigma_s^2 / (\sigma_s^2 + \sigma_n^2) \qquad \textit{(Equation 9)}$$

Substituting Eq. 6 and rearranging for the ratio yields the estimator used throughout:

$$SNR = \sqrt{( r / (1 - r) )} \qquad \textit{(Equation 10)}$$

The derivation makes three assumptions explicit. First, the noise is additive and independent of the signal; a sensor's shot noise is in fact Poisson-distributed and therefore signal-dependent, so Eq. 10 is an idealisation whose accuracy declines in scenes of very uneven illumination. Second, the two observations share an identical underlying signal, which requires a static scene and unchanged illumination between exposures. Third, the two noise realisations are independent, which fails if the sensor contributes a fixed-pattern component common to both frames; any such component is attributed to the signal by this estimator and the SNR is correspondingly overestimated.

These assumptions determine how the measurement was performed in each domain. In the synthetic domain, the conditions hold exactly: a single POV-Ray render provides $s$, and two independent pseudo-random noise fields are added to produce $g_1$ and $g_2$, so the estimator is exact up to sampling error. In the physical domain, the conditions are approximated by capturing two successive exposures of the same stationary target under unchanged illumination, with the camera fixed. This is the reason two reference images were acquired for each camera configuration reported in Chapter 4.5, rather than one.

Control of the ratio likewise differs between domains. In simulation SNR is set directly: the noise standard deviation is the free parameter, and any target ratio is obtained by choosing it. In the physical domain SNR is not set but observed, being a consequence of sensor sensitivity, exposure and scene illumination; it was varied by altering illumination and sensor settings, and measured after the fact. The matching procedure of Chapter 4.6 therefore operates in one direction only, a measured physical ratio is reproduced in simulation, never the reverse.

### 4.1.4 Validating the Simulation Principle

The successful completion of the first milestone (Chapter 3) has empirically demonstrated that the fundamental principle is sound: simulated imagery can effectively replace real-world imagery for macroscopic reconstruction tasks, as evidenced by the successful photogrammetric modelling of the Cube and the University Room. Building on this foundation, this chapter shifts focus from the MACRO level (systems) to the MICRO level (algorithms). The investigation scrutinizes a specific operator, the corner detector, to determine its precise success rates, failure horizons, and operational limitations. To isolate geometric variables, the camera is positioned orthogonally above the box, creating a 2D orthographic view that eliminates perspective foreshortening. The hypothesis posits that if the simulation is

physically accurate, the performance predicted by the synthetic model should be mirrored exactly in the real world.

This correlation is the crux of the investigation. It is anticipated that discrepancies will arise; the real world inevitably contains unmodeled complexity. However, if such divergence is observed, it initiates a refinement loop: determining exactly how to tweak the simulation parameters (e.g., adding blur or texture) to force the real-world convergence. To establish the baseline, the POV-Ray simulation is initialized with a high-contrast canonical target: a single white square on a black background. This stark contrast maximizes the initial signal strength. From this baseline, the illumination is varied, and noise is injected in iterative steps. This process allows for the complete characterization of the corner detector: mapping its performance as a function of camera angle, lighting intensity, and noise floor. Only once this 'Failure Horizon', the point where detection deteriorates, is fully mapped in the simulation, will the experiment move to verify if the physical sensor exhibits the same breakdown behaviour.

## 4.2 Experimental Methodology: The 2D Stress Test

This experiment establishes the controlled stress space required by RQ_3. The two-dimensional target is chosen specifically to eliminate perspective and occlusion as confounding variables, so that the operator's response may be attributed to radiometric conditions alone.

To isolate the variables of Noise and Contrast, a simplified experimental rig was constructed in POV-Ray. Unlike the complex 3D room, this setup consists of a fundamental 2D geometric primitive: a white square cantered on a black background. This high-contrast target (Initial SNR close to infinity) serves as the "Ground Truth" input, known to contain exactly 4 corners.

| Stage | Implementation | Parameters |
|---|---|---|
| Greyscale conversion | OpenCV cvtColor, BGR → GRAY | Applied prior to all detection; chromatic content discarded |
| Corner detection | OpenCV cornerHarris | blockSize = 2; Sobel aperture ksize = 3; Harris k = 0.04 |
| Corner response threshold | — | Response retained where dst > 0.01 × dst.max() |
| Corner count metric | numpy.sum(dst > threshold) | Count of pixels exceeding threshold, without non-maximum suppression. A variant using connectedComponentsWithStats counts distinct components |
| Additive noise | numpy.random.normal on float32 | Mean 0; standard deviation σ = f × 255, with f incremented in 1% steps; applied per channel; no clipping to [0, 255] |
| SNR estimation | Two independent noise realisations | Pearson r between realisations; SNR = √(r / (1 − r)) |
| Blur, box | cv2.blur | Kernel (k, k), k swept |
| Blur, Gaussian | cv2.GaussianBlur | Kernel (2k + 1, 2k + 1); sigmaX = 0 (σ derived from kernel size) |
| Blur measurement | cv2.Laplacian(img, CV_64F).var() | Variance of the Laplacian, used as the scalar blur descriptor matched between domains |
| Contrast sweep | Procedural generation | Foreground 255 → 0 and background 0 → 255 in unit increments; 255 discrete levels |

| Stage | Implementation | Parameters |
|---|---|---|
| SNR matching | Iterative search | Noise increased until synthetic SNR ≤ real SNR, compared to one decimal place |
| ORB | cv2.ORB_create(200) | nfeatures = 200; remaining parameters at OpenCV defaults (scaleFactor = 1.2; nlevels = 8; edgeThreshold = 31; WTA_K = 2; patchSize = 31) |
| ORB keypoint selection | Octave-majority filter | Keypoints retained from the octave group returning the modal keypoint count |

*Table 11 - Parameter settings for image degradation and operator evaluation*

### 4.2.1 Formal statement of the degradation and calibration models

**Noise model.** Injected noise is additive white Gaussian, drawn independently per pixel and per channel:

$$n(x, y) \sim \mathcal{N}(0, \sigma^2), \sigma = f \cdot 255 \qquad \textit{(Equation 11)}$$

where $f$ is the noise fraction, incremented in steps of 0.01. Arithmetic is performed in 32-bit floating point and the result is not clipped to the display range, so that the additive model of Eq. 5 holds exactly and the estimator of Eq. 10 remains unbiased. Clipping would introduce a signal-dependent nonlinearity at the extremes of the range and violate the independence assumption.

The model treats noise as spectrally white, which is an idealisation. Noise in a physical sensor is spatially correlated, because the optical point spread function acts on the photon arrival pattern before sampling and because demosaicing correlates neighbouring pixels. This distinction is not incidental to the present work: a gradient-based detector responds to high spatial frequencies, and white noise contains substantially more high-frequency energy than the correlated noise of a real sensor at equal variance. This is the mechanism underlying the discrepancy reported in §4.5 and resolved in §4.6, where the injected field is convolved with a Gaussian kernel before addition,

$$n' = n * G_{\sigma b} \qquad \textit{(Equation 12)}$$

so that its spectral content approximates that of the physical sensor. Note that n′ has lower variance than $n$; the noise fraction is therefore re-scaled after convolution so that the target SNR is met by the field actually added.

**Blur model.** Two operators were used. Box blur convolves with a uniform kernel of size $k \times k$. Gaussian blur convolves with a separable Gaussian of kernel size $(2k+1)$, with the standard deviation derived from the kernel size by the OpenCV convention

$$\sigma = 0.3 \cdot (k - 1) + 0.8 \qquad \textit{(Equation 13)}$$

Blur is applied uniformly across the frame. A physical lens produces blur that varies with field position and focus distance, so Eq. 13 models the mean behaviour of the optical system rather than its spatial variation. This is the principal simplification of the optical model and is revisited in §6.4.

**Blur measurement.** The scalar descriptor matched between domains is the variance of the Laplacian response,

$$\mathbf{b(I) = Var(\nabla^2 I)} \qquad \textit{(Equation 14)}$$

which is large for images containing sharp intensity transitions and falls as those transitions are smoothed. The measure is content-dependent and therefore not comparable between different scenes; it is admissible here only because the real and synthetic images being compared depict the same target under the same geometry, so that any difference in $b$ is attributable to the optics rather than to scene content.

**Calibration procedure.** The matching of a synthetic image to a physical one proceeds as follows, and is stated here as an ordered procedure rather than described narratively:

1. Acquire two successive exposures $g_1$, $g_2$ of the physical target under fixed illumination.
2. Compute the physical blur descriptor $b\_real = \mathrm{Var}(\nabla^2 g_1)$ by Eq. 14.
3. Compute $r = \mathrm{corr}(g_1, g_2)$ and hence SNR_real by Eq. 10.

4. Render the synthetic counterpart at matched pose, intrinsics and resolution.

5. Increase the blur kernel $k$ applied to the render until $\mathrm{Var}(\nabla^2 I\_\mathrm{virt})$ first falls to or below $b\_real$.

6. Generate two independent noise fields, convolve each by Eq. 12, and scale $\sigma$ until the SNR of the resulting synthetic pair, computed by Eq. 10, first falls to or below SNR_real, compared to one decimal place.

7. Apply the operator under test to both domains with identical parameters.

Step 5 precedes step 6 deliberately. Blurring after noise addition would attenuate the injected noise and invalidate the SNR match established in step 6. The operations do not commute, and the order is therefore a constraint of the method rather than a matter of convenience.

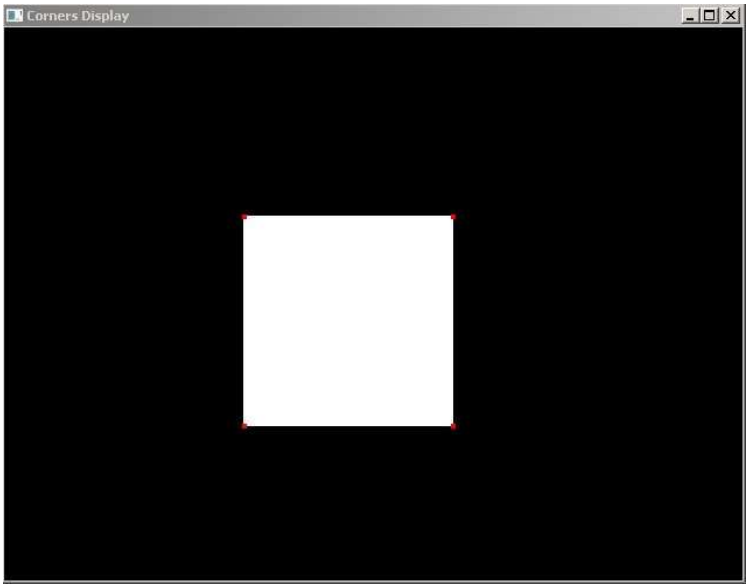


*Figure 26-High contrast POV-Ray simulation with correct number of corners detected (4 corners represented by the 4 red dots)*

This synthetic input was subjected to a systematic degradation protocol:

1. **Noise Injection:** Gaussian noise was introduced in incremental steps of 1%.
2. **Contrast Reduction:** The intensity difference between the foreground square and the background was reduced in 255 discrete steps, decreasing the dynamic range from (0, 255) down to (0, 0).

The objective was to determine the precise "Breaking Point", the threshold at which the algorithm fails to return the correct corner count (n=4). At higher noise amplitudes the algorithm returns increasing numbers of spurious detections. Please see the Figure 26 below.

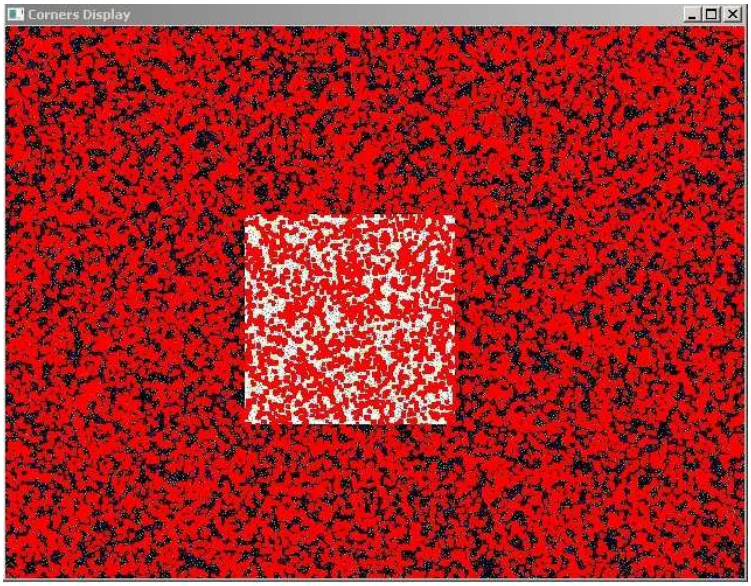


*Figure 27- Corner detection failure under high noise. Gaussian noise exceeding 10% of the intensity range has been added to the synthetic target. The dense cluster of markers represents pixels whose Harris response exceeds the detection threshold, at this noise level the response field is dominated by stochastic artefacts and the four true corners cannot be isolated.*

The preliminary stress tests revealed a critical instability when utilizing high-contrast targets; the binary nature of the black-and-white stimulus rendered the algorithm highly susceptible to saturation artifacts, causing reliability to degrade rapidly even under minimal noise conditions. Consequently, a strategic decision was made to transition the experimental protocol towards a target exhibiting a reduced contrast ratio, thereby simulating a more naturalistic radiometric distribution. To achieve this, the test stimulus was altered to a chromatic image featuring a green cube superimposed against a neutral grey background. It is important to note that while the input is chromatic, the specific hue is technically irrelevant to the corner detection algorithm itself, as the standard pipeline invariably performs a pre-processing step to convert the three-channel RGB input into a single-channel grayscale intensity map (luminance) prior to calculating the spatial gradients required for feature extraction.

However, the empirical data resulting from this modification yielded an outcome that was both unexpected and scientifically significant. Contrary to the brittle behavior observed with the high-contrast binary targets, the corner detector demonstrated a markedly superior robustness when processing these lower-contrast inputs. Specifically, the algorithm maintained stability and detection accuracy at significantly higher noise thresholds, suggesting that extreme contrast may actually exacerbate noise sensitivity, whereas a moderated dynamic range allows the detector to distinguish geometric features from stochastic noise more effectively.

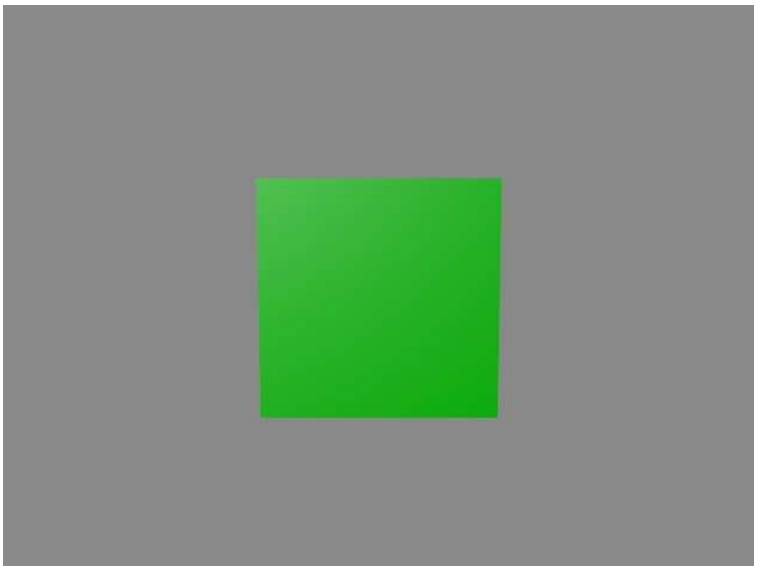

*Figure 28-Colour sample image (converted to greyscale upon processing)*

### 4.2.2 The corner detection operator: mechanism, assumptions and limitations

The operator characterised throughout this chapter is the Harris corner detector, as implemented in OpenCV. Its mechanism is stated here formally, since the interpretation of every result in this chapter depends on it. For each pixel, first-order derivatives $I_x$ and $I_y$ are

computed by Sobel convolution with an aperture of 3. Over a square window $w$ of side 2, the structure tensor is accumulated:

$$M = \Sigma_w \left[ I_x^2 \; I_x I_y \; ; \; I_x I_y \; I_y^2 \right] \qquad \text{(Equation 15)}$$

The eigenvalues of **M** describe how strongly intensity varies in the two principal directions of the window. The detector response avoids computing them explicitly, using instead

$$R = \det(M) - \kappa \cdot \text{trace}(M)^2, \; \kappa = 0.04 \qquad \text{(Equation 16)}$$

$R$ is large and positive where both eigenvalues are large, corresponding to intensity variation in two independent directions; large and negative along an edge, where one eigenvalue dominates; and near zero in a uniform region. A pixel is declared a corner where

$$R(x, y) > 0.01 \cdot \max R \qquad \text{(Equation 17)}$$

and the reported count is the number of pixels satisfying Eq. 17. No non-maximum suppression is applied, so a single geometric corner spanning several pixels contributes several counts. This is why the counts reported in this chapter reach the thousands under high noise, and why the quantity should be read as a measure of detector instability rather than as an enumeration of geometric corners.

Three properties of this formulation bear on the results. The threshold of Eq. 78 is *relative* to the maximum response in the same image, so it is not a fixed criterion: as noise raises the response field, the threshold rises with it, and the growth in counts reported in Chapter 4.3 occurs despite this adaptive compensation rather than because of its absence. The parameter $\kappa$ is empirical and has no derivation from first principles; the value 0.04 is the conventional

choice and was not varied. Finally, the operator is not scale invariant, which is immaterial here because the target is imaged at fixed scale, but would need addressing before the characterisation could be transferred to a scene of varying depth.

The assumptions of the method are that local image structure is adequately described by first-order gradients, and that gradients arising from scene geometry dominate those arising from noise. The second assumption is precisely what the experiments of this chapter are designed to violate in a controlled manner: the failure horizon reported in Chapter 4.3 is the point at which it ceases to hold.

| Quantity | Definition | Implementation | Appears in |
|---|---|---|---|
| Signal-to-noise ratio | Eq. 6, estimated by Eq. 10 | Pearson r over two independent realisations; sqrt(r/(1-r)) | Chapters 4.3.1, 4.4, 4.5, |
| Injected noise | Eq. 11 | numpy.random.normal, $\sigma = f \times 255$, float32, unclipped | Chapter 4.3 |
| Correlated (blurred) noise | Eq. 12 | Gaussian convolution of the noise field prior to addition; variance rescaled after convolution | Chapter 4.6 |
| Optical blur | Eq. 13 | cv2.GaussianBlur, kernel (2k+1), sigmaX = 0; $\sigma$ derived from kernel size | Chapter 4.6 |
| Blur descriptor | Eq. 14 | cv2.Laplacian(img, CV_64F).var() | Chapter 4.6 |
| Structure tensor | Eq. 15 | cornerHarris, blockSize = 2, Sobel aperture = 3 | Chapters 4.3, 4.4 |
| Detector response | Eq. 16 | Harris response with $\kappa = 0.04$ | Chapters 4.3, 4.4 |
| Corner criterion | Eq. 17 | dst > 0.01 * dst.max(); count of qualifying pixels, no non-maximum suppression | All corner counts, Chapters 4 and 5 |
| Ray–surface intersection | Eqs. 3, 4 and 15 | POV-Ray analytic solution over CSG primitives; smallest positive root | All renders, Chapters 3–5 |

| Quantity | Definition | Implementation | Appears in |
|---|---|---|---|
| Keypoint orientation | Eq. 19 | cv2.ORB_create(200), remaining parameters at OpenCV defaults | Chapter 5.4 |

*Table 12 - Correspondence between theoretical quantities, their implementation and the results in which they appear*

## 4.3 Phase 1: Noise Sensitivity Analysis

This phase isolates the first of the two stress variables identified in RQ_3, and provides the initial evidence bearing on hypothesis H2.

The first series of experiments analysed the detector's response to pure Gaussian noise applied to a high-contrast image. To rigorously quantify the onset of detection instability, a controlled Gaussian noise artifact was introduced into the signal, with a magnitude equivalent to approximately 1% of the total grayscale intensity spectrum. Under these specific low-noise conditions, the corner detection algorithm registered a total of 41 discrete feature points. This result is statistically significant as it demarcates the initial deviation from the ideal ground truth. Even at this minimal level of signal perturbation, the operator begins to identify spurious features. The visual evidence of this detection state, illustrating the spatial distribution of the 41 identified corners, is presented in the Figure 29 below.

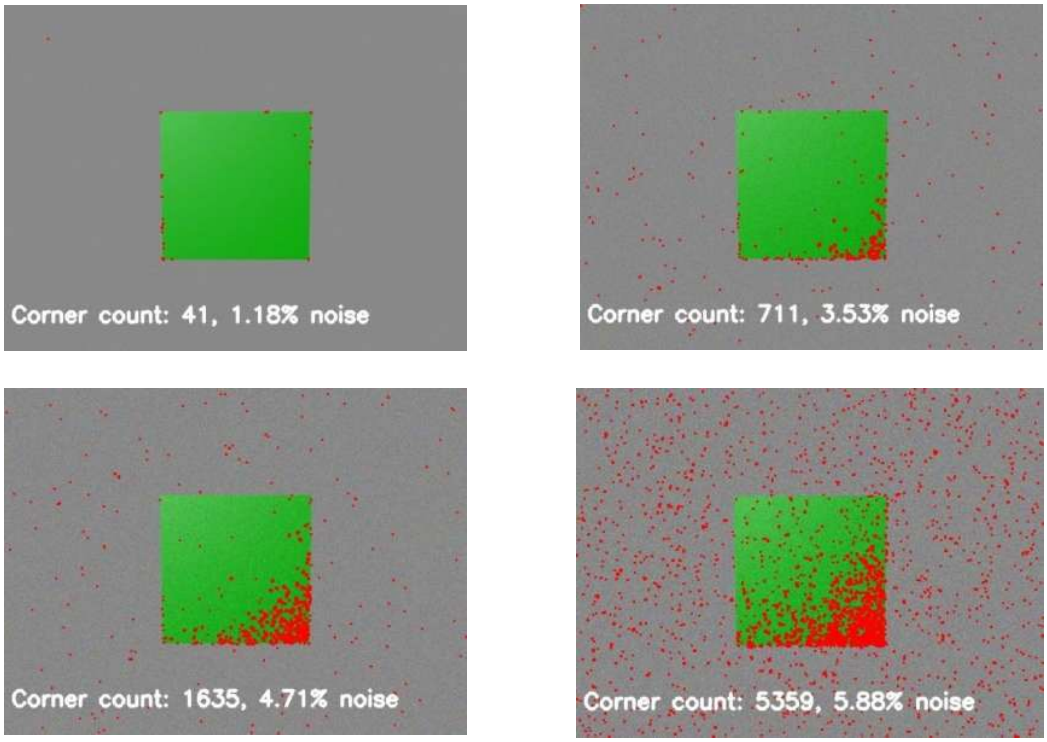


*Figure 29-Increasing application of noise  levels, and corner detection*

Observational analysis of the resultant imagery reveals a distinct correlation: as the amplitude of the injected noise is incrementally increased, the corner detection algorithm exhibits a proportional rise in the number of identified features. This trend manifests as a progressive accumulation of spurious detections, a behaviour that was anticipated given the stochastic nature of the noise model. However, a critical distinction must be made regarding the stability of this degradation.

Unlike the erratic and unpredictable failure modes observed when processing high-contrast binary images, the behaviour demonstrated here, within the context of the lower-contrast chromatic target, is significantly more consistent and mathematically predictable. This stability implies that reducing contrast allows for a more controlled study of noise sensitivity.

Building upon these empirical observations in Figure 28, it becomes possible to construct a quantitative performance profile. By systematically plotting the aggregate count of detected corners against the increasing magnitude of applied noise, the algorithm’s degradation curve may be visualised. The resulting graph, presented below in Figure 29, serves as a crucial diagnostic tool. It allows for the precise identification of the 'Failure Horizon', the specific noise threshold at which the detector ceases to function reliably, thereby delineating the operational limits of the algorithm under simulated stress conditions.

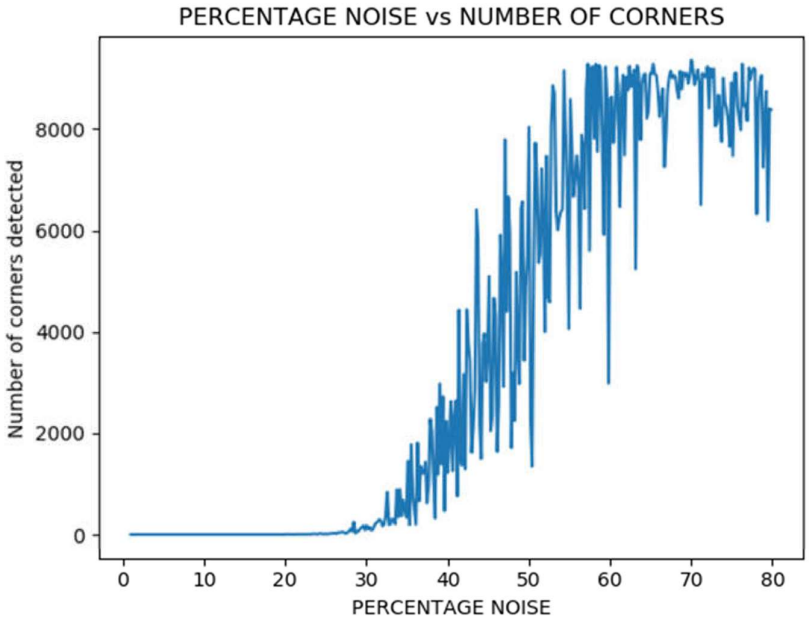


*Figure 30-Plotting Noise levels against the number of corners detected.*

Further analysis of this non-linear progression reveals a distinct "failure horizon" inherent to the feature extraction algorithm. While the corner detector demonstrates a degree of resilience to minor optical imperfections, its performance does not degrade well. Rather, it collapses catastrophically once a specific interference threshold is breached. The empirical data from Figure 30 clearly dictates that this critical breaking point occurs at approximately a 30% noise level. Up to this boundary, the algorithm can largely filter out anomalies and maintain structural coherence. However, immediately beyond this 30% horizon, the system becomes entirely overwhelmed, triggering an exponential explosion of spurious corners. Identifying this definitive boundary is a crucial finding, as it establishes the precise optical tolerance limits of the computer vision system, indicating that beyond a 30% noise threshold the detector output is dominated by false positives, from which it follows that any photogrammetric reconstruction built upon it would be unreliable. This can be seen in the Figure 30 below.

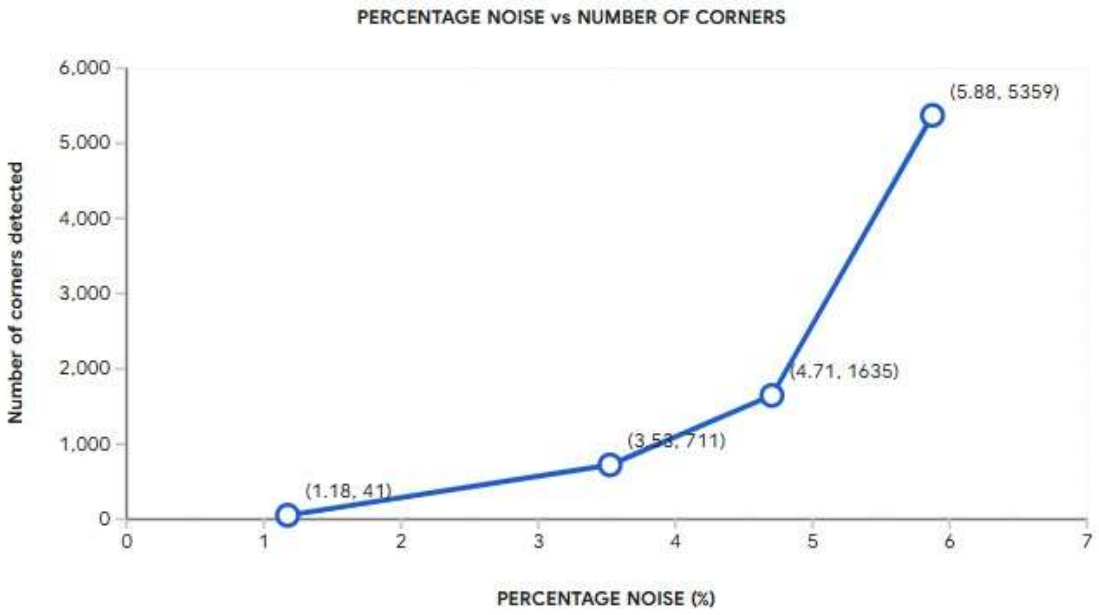


*Figure 31 - The non-linear failure horizon of the Harris corner detector. Detected corner count (ordinate) is plotted against applied Gaussian noise as a percentage of the intensity range (abscissa). The detector remains stable from 0% to approximately 20% noise, beyond which false positives increase exponentially rather than degrading gracefully.*

The graphical representation delineates the relationship between signal degradation and feature extraction. Specifically, the independent variable, the magnitude of the applied Gaussian noise, expressed as a percentage of the total intensity spectrum, is mapped along the abscissa (X-axis), while the dependent variable, the aggregate count of detected corner features, is plotted along the ordinate (Y-axis). An analytical inspection of the data reveals a distinct operational envelope: the corner detection algorithm maintains a state of nominal stability and accuracy within the noise interval ranging from 0% to approximately 20%.

However, surpassing this critical threshold precipitates a phase transition in the detector's behaviour. Any noise magnitude exceeding 20% causes the number of identified corners to diverge rapidly in a non-linear fashion. This exponential increase serves as a definitive indicator that the algorithm's inherent processing limits have been breached, resulting

in a cascade of false positives. Consequently, to conduct a rigorous analysis of the detector's valid performance characteristics, it is necessary to exclude the data representing this catastrophic failure mode. By disregarding noise levels above the 20% saturation point, the analysis can focus exclusively on the 'useful portion' of the algorithm's utility curve. This targeted visualization, effectively providing a zoomed perspective on the stable operating region, is presented in the graph below (Figure 32).

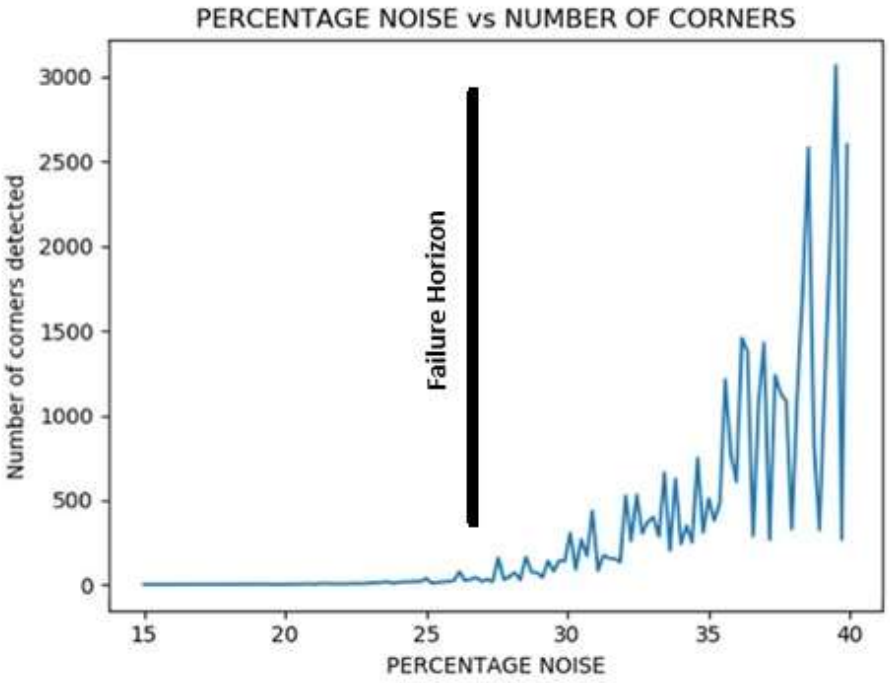


*Figure 32-Useful portion of the graph*

In this graphical representation, the abscissa (X-axis) quantifies the magnitude of the stochastic noise injected into the system, specifically applied to the chromatic test subject characterized by a green geometric primitive superimposed on a neutral grey background. Conversely, the ordinate (Y-axis) plots the dependent variable: the aggregate number of discrete corner features identified by the algorithm. Through this rigorous simulation process,

a definitive conclusion regarding the operational stability of the detector has been reached. The data suggests that the corner detection operator maintains a state of nominal functionality and high accuracy, but only within a bounded interval of specific noise densities. However, a critical 'Failure Horizon' is observed in Figure 31; subjecting the image to any noise level exceeding the 20% threshold precipitates a phase transition in the detector's behaviour. Beyond this point, the algorithm does not merely degrade linearly. Rather, it begins to erroneously interpret noise artifacts as geometric features, resulting in a cascade of false positives that manifests as an error rate following an apparent exponential growth curve.

As illustrated in the noise response graphs (Figures 30 and 31), the detector exhibits a non-linear failure mode.

- **Stable Region (0% - 20% Noise):** The algorithm is robust, consistently identifying the 4 true corners with zero false positives. This defines the stable operating region.
- **The Knee of the Curve (~25% Noise):** A distinct transition occurs. The detector begins to hallucinate features, returning a corner count of 41 at 1.18% noise (relative to grey levels) but exploding to 711 corners at 3.53% noise.
- **Catastrophic Failure (>30% Noise):** Beyond this threshold, the failure is exponential. At 5.88% noise, the system detects 5359 corners, rendering the output statistically useless.

This confirms that corner detection is not a linear process, it possesses a "Failure Horizon" beyond which no useful data can be extracted.

### 4.3.1 Signal-to-Noise Ratio (SNR) Correlation

The subsequent phase of the analytical framework necessitates a transition to a more robust quantitative metric: the Signal-to-Noise Ratio (SNR). Consequently, the experimental data will be re-plotted to map the aggregate count of detected corner features against the calculated SNR of each respective image. This transformation is critical, as SNR provides a normalized measure of signal quality independent of absolute intensity values. The resulting graphical representation serves as a primary diagnostic tool for characterizing detector sensitivity.

To normalize these findings across different lighting conditions, the results were plotted against SNR. The results (Figure 33) confirm a strong correlation: detection accuracy is maintained as long as the SNR remains above a critical threshold of approximately 3.5 . Below this value, the noise floor overwhelms the gradient calculation used by the detector, resulting in massive spurious detections.

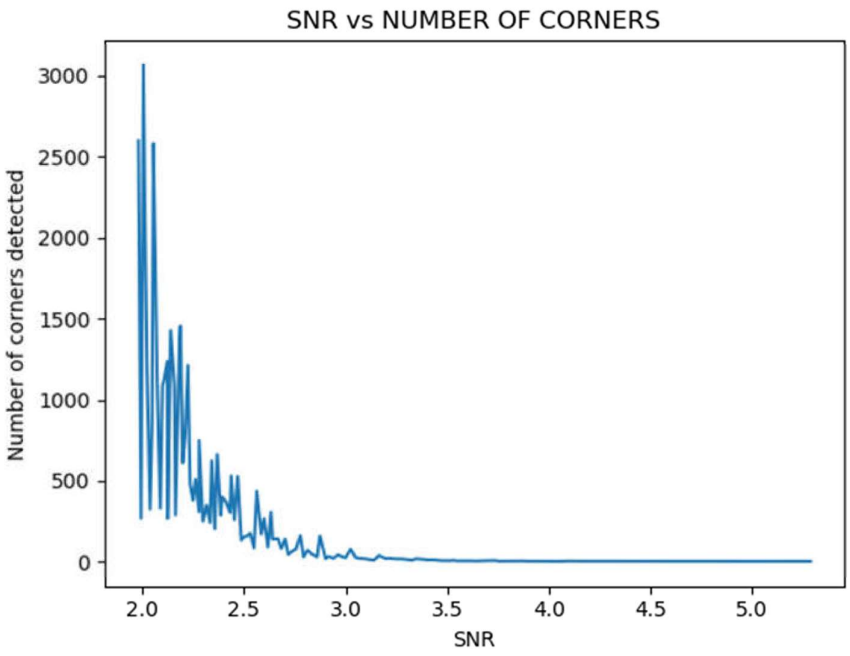


*Figure 33-Useful portion of the graph (SNR)*

Analysis of the resulting plot confirms that the empirical data aligns closely with the theoretical performance models established in the hypothesis. A distinct correlation is observed: high SNR values, corresponding to low-noise conditions, are consistently associated with stable detection rates, where the algorithm correctly identifies the true geometric features. However, a critical operational ceiling is identified; once the noise floor rises sufficiently to generate a spurious corner count exceeding 500 detections, the algorithm effectively ceases to function as a valid feature extractor.

At this point, the signal is overwhelmed, and the output becomes statistically indistinguishable from random noise. To rigorously validate these synthetic findings within a physical context, a subsequent experimental phase is proposed. This will involve the generation of real-world imagery within a controlled 'light tent' environment. By precisely manipulating the physical illumination to replicate the radiometric conditions of the simulation, it will be

possible to ascertain whether physical camera sensors exhibit an identical SNR-dependent failure mode. This validation step is currently scheduled as the immediate priority for the next stage of research.

## 4.4 Phase 2: Contrast Sensitivity and the "Datasheet"

This phase introduces the second stress variable and, in combination with Chapter 4.3, completes the characterisation sought by RQ_3. The resulting two-variable manifold is what permits the Safe Operating Area to be defined.

Real-world environments rarely offer perfect black-and-white contrast. Therefore, the second phase of experimentation introduced variable contrast as a second dimension of stress. While the previous set of simulations successfully isolated the impact of stochastic noise on feature detection, a comprehensive Technology Evaluation requires the investigation of a second critical variable: radiometric contrast. It is scientifically imperative to ascertain how the dynamic range between an object and its background influences the stability of the corner detector. The previous experiments established a baseline performance metric by applying incremental noise to a static, high-contrast target. However, this raises a fundamental question: does the corner detector exhibit the same resilience when the signal strength itself is diminished? To address this, the experimental protocol was expanded to subject the same geometric primitive to a dual-variable stress test. Specifically, the simulation was designed to apply the established noise increments not just to a single image, but iteratively across a spectrum of 255 distinct images, each representing a discrete level of contrast between the foreground subject and the background field.

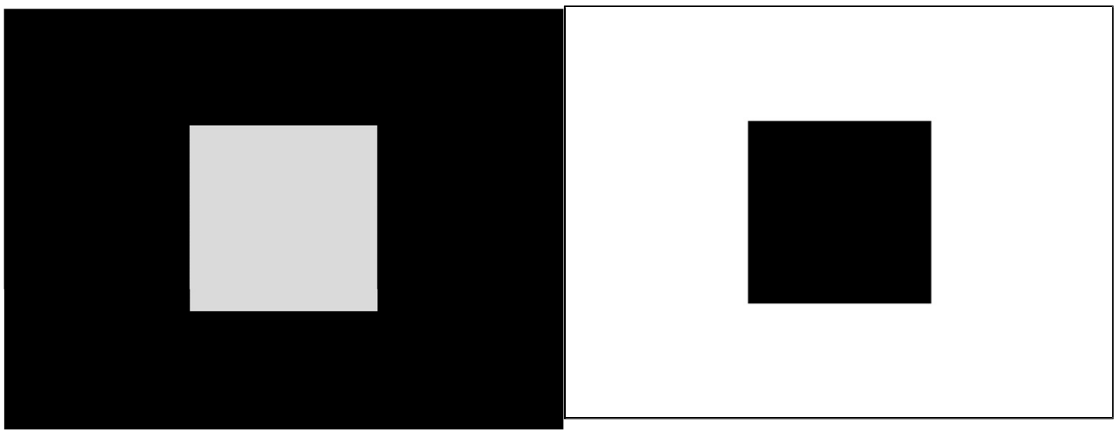

*Figure 34-The beginning and the end images of a 255 simulations cycle*

The experimental initialization is depicted in Figure 34, which serves as the high-contrast control sample used in the preceding noise analysis (Figures 27 and 28). To rigorously map the relationship between image contrast, Signal-to-Noise Ratio (SNR), and detection accuracy, a procedural generation script was utilized to create a dataset of 255 discrete test images. This dataset represents a linear degradation of dynamic range. Specifically, the RGB intensity values of the foreground square are systematically decremented from pure white (255, 255, 255) down to total black (0, 0, 0) in unitary increments of (1, 1, 1). Simultaneously, to achieve a convergence towards zero contrast, the background RGB values are incremented from pure black (0, 0, 0) upwards. This inverse modification of foreground and background intensities effectively simulates the gradual washing out of features, mimicking real-world conditions such as lens flare or extreme low-light environments where discernible contrast is lost.

The terminal state of this simulation sequence is illustrated in Figure 35, representing the point of zero contrast. To analyze the massive dataset generated by this process—comprising 255 distinct contrast levels, each subjected to the full range of noise increments—a complex visualization strategy is required. Consequently, a volumetric 3D plot has been generated to map the interaction between these variables. Rather than analyzing isolated 2D

cross-sections (like Figure 30 or Figure 31), this 3D representation aggregates 255 individual performance curves into a single continuous manifold. In this visualization, each 'slice' along the depth axis represents a specific contrast scenario, allowing for the observation of performance trends across the entire radiometric spectrum. The working hypothesis for this experiment posits an inverse correlation: as the contrast level decreases, the signal strength of the geometric corners diminishes, thereby making the detector increasingly susceptible to the noise floor. It is expected that the algorithm will identify an increasing number of spurious corners at lower noise thresholds as contrast fades, eventually reaching a critical 'Failure Horizon' where the ability to detect features suffers a catastrophic breakdown regardless of the noise level.

To facilitate a clearer interpretation of these complex trends before delving into the dense 255-step dataset, an intermediary analytical step was implemented. This preliminary phase involves plotting the evolution of the system across a coarser resolution of just 10 discrete simulation steps. By defining 10 substantial increments for the background intensity and 10 corresponding decrements for the foreground intensity, the macroscopic behaviour of the corned detector may be isolated and visualised. This approach acts as a low-resolution sampling of the performance manifold, allowing for the identification of broad trend lines and critical failure regions before validating them against the high-resolution data.

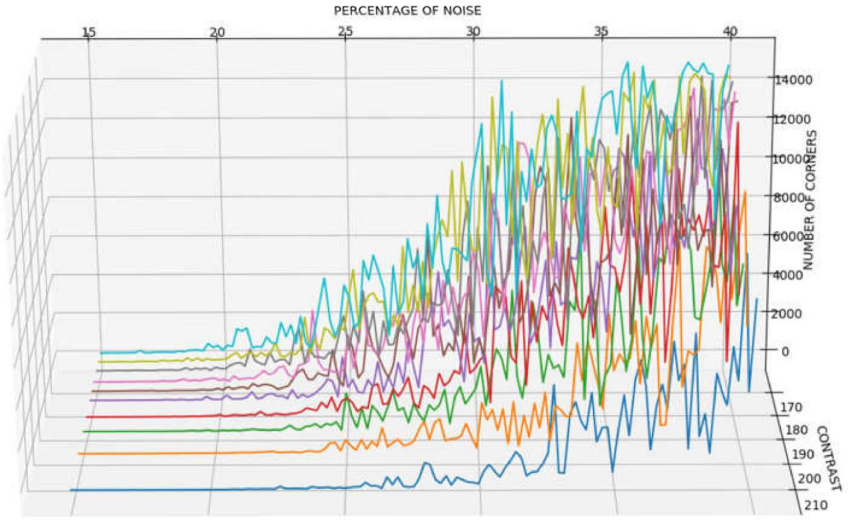


*Figure 35-Influence of contrast on corner detection (10 contrast levels)*

The visual evidence presented in the preceding figure underscores a fundamental dependency: the radiometric contrast between a feature and its background is a primary determinant of detection accuracy. Even within this coarse-grained analysis—limited to just ten discrete contrast intervals—the impact of signal degradation on algorithmic stability is observable and statistically significant. In the high-contrast simulation scenario, where the intensity delta between the foreground square and the background is maintained at 210 on the 8-bit RGB scale, the algorithm exhibits a commendable degree of robustness. Specifically, the detector maintains feature lock and resists false positives until the injected noise amplitude exceeds approximately 20% of the signal.

Conversely, a marked deterioration in performance is observed as the dynamic range is compressed. In scenarios where the contrast delta falls below the threshold of 170, the 'Failure Horizon' shifts dramatically; the detector's ability to resolve geometric features collapses at noise levels significantly lower than the 20% benchmark. This indicates that as signal strength

diminishes, the algorithm's sensitivity to stochastic noise increases non-linearly, requiring a much cleaner signal to function correctly.

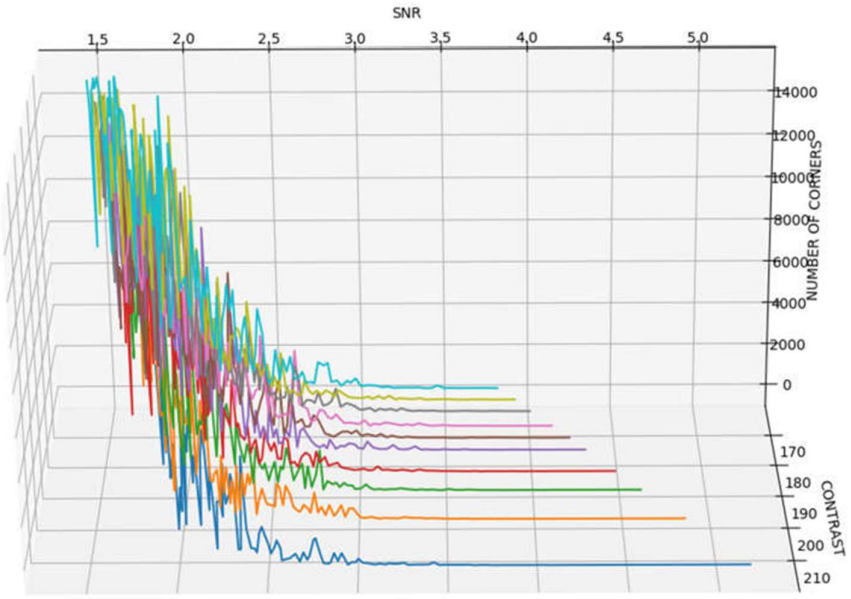


*Figure 36- Influence of contrast on corner detection (SNR - 10 contrast levels)*

Corroborating these findings, the analysis of the Signal-to-Noise Ratio (SNR) data presents a statistically identical narrative, reinforcing the intrinsic link between contrast and signal integrity. A high-contrast image inherently possesses a superior SNR compared to a low-contrast equivalent, providing a stronger spatial gradient for the detection kernel to latch onto. By isolating SNR as a distinct variable, a predictive behavioral model for the corner detector begins to emerge. It becomes possible to forecast the operational efficacy of the algorithm solely by analyzing the interplay between the scene's contrast and its inherent Signal-to-Noise Ratio.

Empirical observation of the generated data indicates a definitive operational floor: provided the SNR remains above a critical value of 3, the corner detection operator consistently resolves the correct number of geometric features, largely independent of the absolute contrast

values. This suggests that SNR is the governing variable for stability. To fully map this dependency and verify if this threshold holds across the entire radiometric spectrum, the investigation will now proceed to analyze the full dataset comprising all 255 possible contrast variations. The image below (Figure 36) will illustrate this.

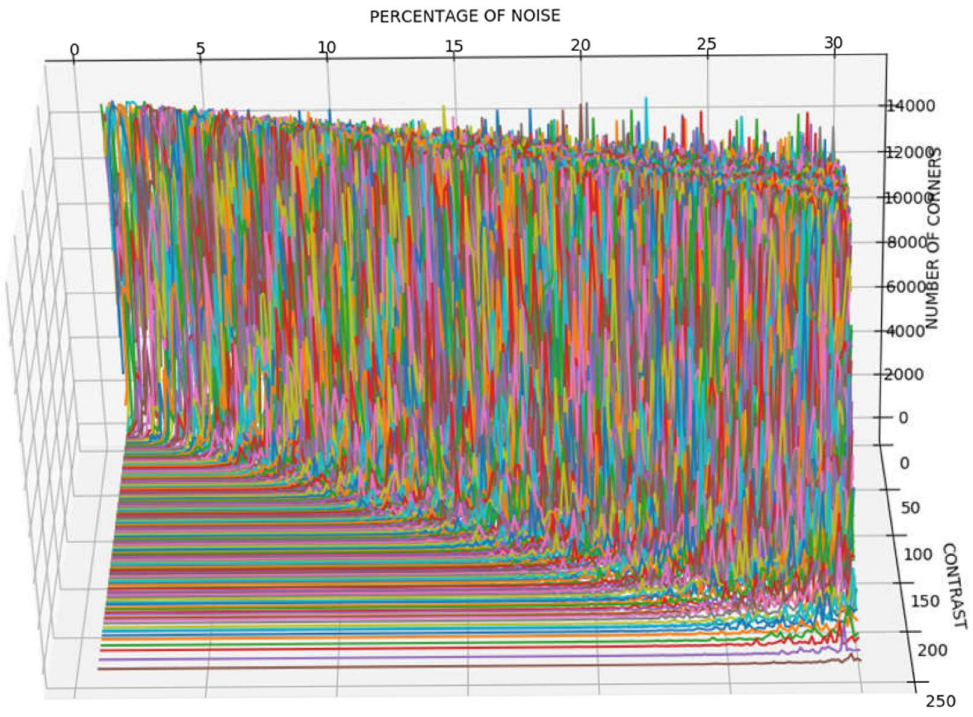


*Figure 37- Performance manifold for corner detection across 255 contrast levels. Each slice along the depth axis represents one foreground-background intensity difference, noise tolerance is shown to be directly proportional to contrast.*

By varying the foreground/background intensity difference across 255 levels, a 3D performance manifold was generated (Figure 37). This visualization reveals a critical dependency: the system's tolerance for noise is directly proportional to image contrast.

- **High Contrast (>200):** The detector remains robust even with significant noise (up to ~30%).
- **Low Contrast (<170):** The tolerance window collapses. Below a contrast delta of 170, the detector fails before reaching 20% noise .
- **The "Undefined" Region:** The analysis identified a zone where detection is mathematically impossible. When the contrast drops below a certain threshold (approximately 10 levels), the signal is insufficient to trigger the gradient threshold of the algorithm, regardless of how low the noise is.

The rigorous interrogation of the dataset yields a definitive conclusion regarding the radiometric prerequisites for robust feature extraction: accurate detection of the correct number of geometric corners was obtained consistently only where a high contrast dynamic range was maintained. The empirical evidence derived from the upper bound of the simulation manifold demonstrates that when the intensity delta between the foreground feature and the background exceeds a value of 200 (on the standard 8-bit scale), the corner detection algorithm exhibits exceptional stability.

Within this high-contrast regime, the operator proves resilient against substantial stochastic interference, maintaining accurate feature lock even when subjected to aggressive noise levels approaching 30%. Conversely, the performance profile deteriorates precipitously in the low-contrast domain. In scenarios where the radiometric difference is compressed, the signal strength becomes insufficient to distinguish actual geometric gradients from the noise floor, creating a condition where it is mathematically difficult for the detector to resolve the true corner count. The volumetric graph presented previously provides a comprehensive

visualization of this non-linear response, mapping the precise interaction between contrast degradation and noise injection.

In order to see if plotting the Contrast against the SNR follows the same pattern as seen in the simulation above with just 10 variations in contrast, please consider the graph below (Figure37).

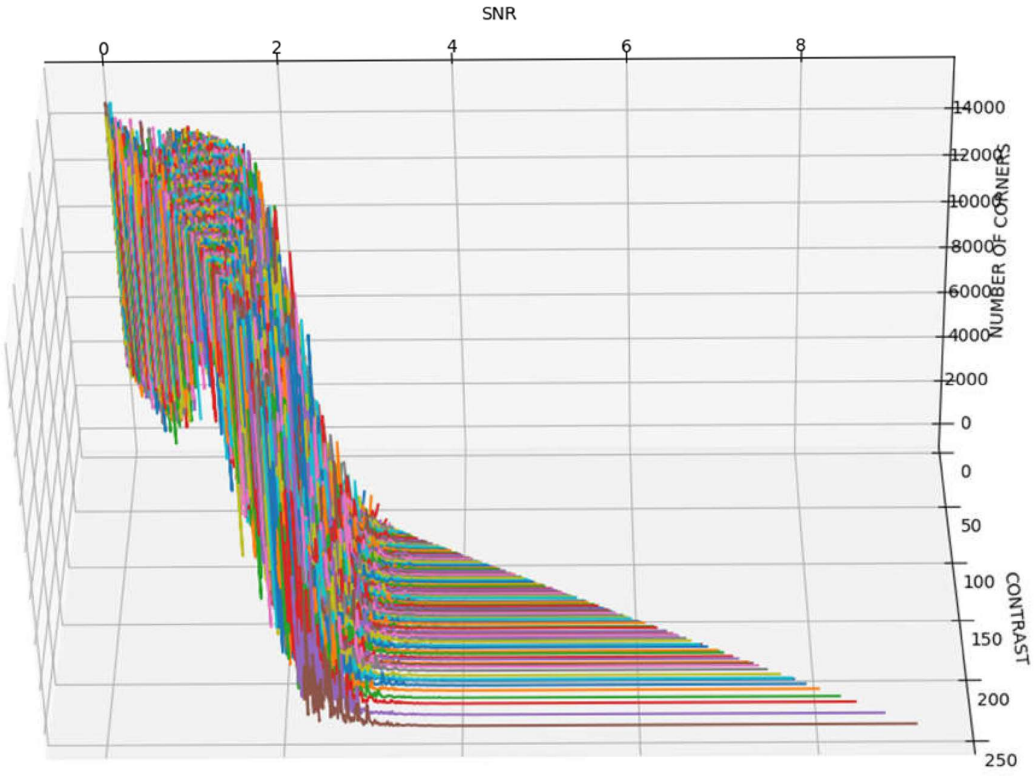


*Figure 38-Influence of contrast on corner detection (SNR - 256 contrast levels)*

A comprehensive synthesis of the experimental data reveals a striking uniformity across the entire simulation corpus. Whether analyzing isolated single-variable stress tests or complex multi-variable scenarios, the algorithmic response adheres to a consistent, predictable pattern. This congruity between discrete and aggregate simulations validates the underlying theoretical

model, confirming that the observed failure modes are systemic characteristics of the feature detector rather than transient anomalies.

Regarding the Signal-to-Noise Ratio (SNR) metric, the data crystallizes into a definitive operational rule. The interplay between radiometric contrast and stochastic noise acts as a cumulative degradation function. The empirical evidence demarcates a hard stability threshold at an SNR value of approximately 3.5. Provided that the combined impact of contrast reduction and noise injection does not depress the image quality below this critical floor, the corner detection operator retains its capacity to resolve geometric features with high fidelity. Specifically, the algorithm successfully extracts the correct number of feature points, distinguishing actual corners from noise artifacts as long as this signal integrity condition is met. This finding effectively establishes a quantifiable 'safe operating area,' permitting algorithmic success to be predicted based solely on the calculated SNR of the input stream.

To rigorously delineate the operational boundary distinguishing the region of reliable algorithmic function from the regime where excessive stochastic noise and insufficient radiometric contrast precipitate detection failure—as initially characterized in the volumetric plot of Figure 34—a dedicated visualization strategy is employed. This critical separation is explicitly mapped in the subsequent figures, utilizing both two-dimensional and three-dimensional representations to facilitate a comprehensive topological analysis of the detector's performance manifold.

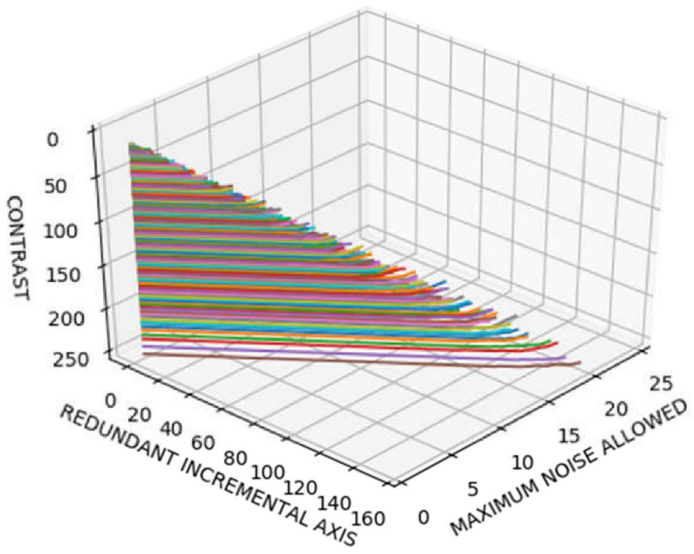


*Figure 39-3D Noise vs. Contrast slice where the corner detector is accurate*

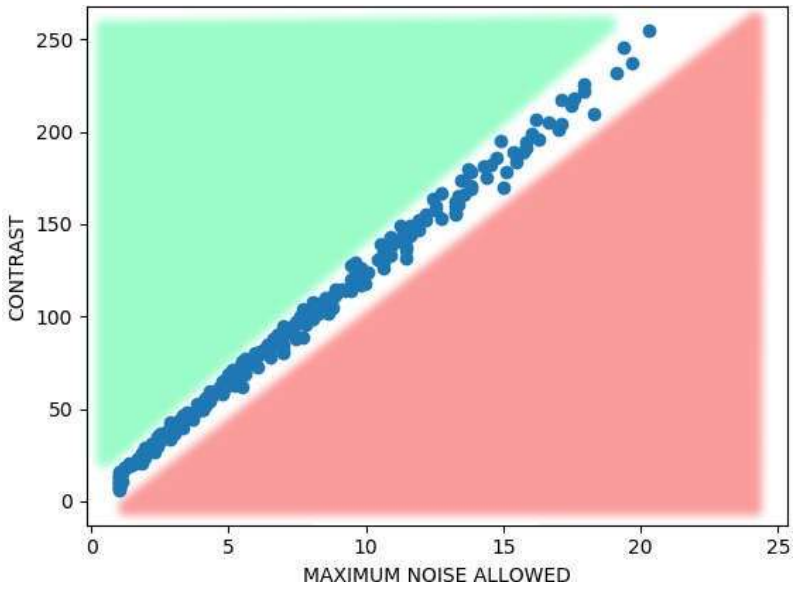


*Figure 40-2D Two-dimensional performance map delineating the Safe Operating Area for corner detection, as a function of contrast and maximum tolerable noise. The green region denotes conditions under which the detector returns the correct count (n = 4). The red region denotes failure through noise overwhelming the signal. The undefined region denotes contrast too low to trigger the response threshold at any noise level.*

While the volumetric 3D representation provides a holistic view of the performance manifold, a two-dimensional projection often offers superior interpretability regarding specific operational thresholds. It is important to note that the dataset utilized here is directly extrapolated from the complex manifold previously presented in Figure 37. The primary objective of this 2D visualization is to sharply delineate the operational envelope of the corner detector, transforming abstract data points into a clear decision boundary. The algorithmic limitations are explicitly demarcated within this planar view.

### 4.4.1 Defining the Safe Operating Area (SOA)

The region rendered in green constitutes the 'Safe Operating Area' (SOA), representing the specific combination of environmental variables wherein the corner detector reliably yields accurate results. Conversely, any coordinate pair of contrast and noise that maps to the red region signifies a breach of the detector's stability capabilities; under these conditions, it is mathematically predictable that the algorithm will fail to maintain accuracy. By mapping these failure points, a 2D "Performance Map" was constructed (Figure 40). This map divides the operational space into three distinct zones:

1. **The Green Zone (Safe Operation):** The region above the diagonal in which the combination of contrast and SNR yielded accurate detection (n=4) throughout the simulated range.
2. **The Red Zone (Failure):** The region where noise overwhelms the signal, resulting in spurious corners.
3. **The Undefined Zone:** The region where contrast is insufficient for feature extraction.

This map serves as the predictive "Datasheet" for the algorithm. Theoretically, if a real-world camera's SNR and the scene's contrast fall within the "Green Zone," the simulation predicts the vision system will function correctly.

Applying this same analytical methodology to the second set of variables, the figure below (Figure 41) presents a two-dimensional projection of the relationship between Signal-to-Noise Ratio (SNR) and Contrast. This visualization is critical for isolating the precise numerical SNR thresholds required to ensure the input imagery remains within the detector's effective dynamic range, thereby preventing the algorithm from drifting into an unstable state. While the volumetric topology of this relationship was previously established and visualized in Figure 37, the clarity afforded by a planar projection is necessary for defining rigorous operational bounds. Consequently, for the purposes of this specific stability analysis, the discussion will focus exclusively on the 2D representation derived from that dataset.

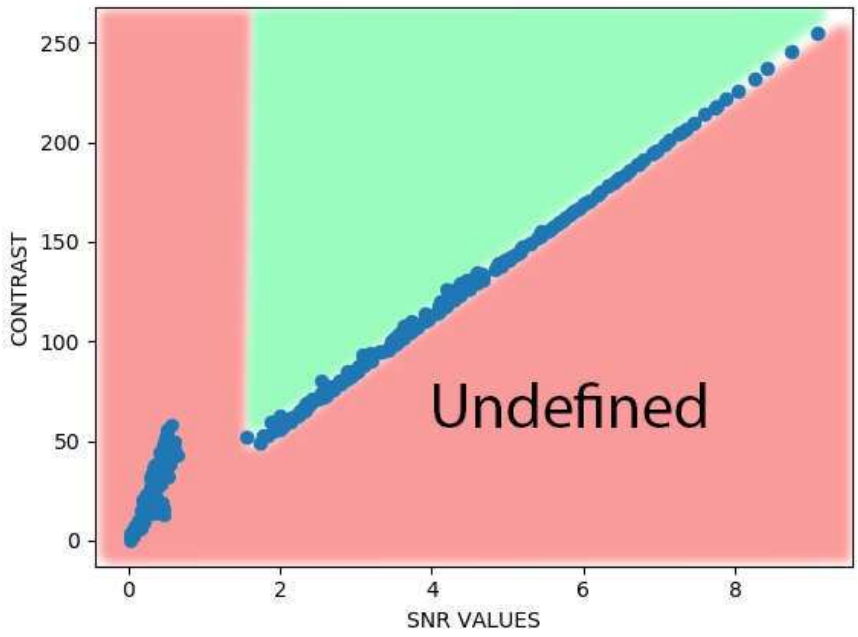


*Figure 41-2D Noise vs. SNR slice where the corner detector is accurate*

In the graphical analysis, the region rendered in red signifies the operational domain where the corner detection algorithm fails to yield valid geometric results, producing either null data or excessive spurious artifacts. Furthermore, the analysis identifies a distinct region classified as 'Undefined.' This specific zone corresponds to environmental conditions where the Signal-to-Noise Ratio (SNR) faces a hard physical limit; specifically, the radiometric contrast is insufficient to form a valid signal, rendering the concept of noise mitigation irrelevant because the signal itself is effectively non-existent. This empirical observation provides definitive evidence that in low-contrast scenarios, the dynamic range of the image acts as the governing constraint on detection accuracy, superseding the influence of stochastic noise. Topologically, this 'Undefined' zone is located immediately beneath the diagonal inflection point of the graph.

Conversely, the region situated to the left of the diagonal break point represents a zone of stochastic uncertainty. In this regime, regardless of the contrast magnitude or background intensity, the SNR is depressed to such an extreme degree—driven by high-amplitude noise injection—that the algorithm is incapable of distinguishing geometric features from background artifacts. Consequently, any expectation of accurate algorithmic behaviour in this zone is mathematically unfounded. Therefore, the 'Safe Operating Area' (SOA) for this computer vision operator is strictly confined to the green region situated above the diagonal boundary.

The quantification of the SNR metric is of paramount importance to this research, as it provides the fundamental basis for characterizing the corner detector operator as a deterministic system rather than a stochastic black box. By constructing a comprehensive performance map

that explicitly visualizes the algorithm's operational strengths and failure modes, a predictive framework is established. This framework enables the forecasting of the detector’s behaviour in novel scenarios independent of physical testing. The model posits that as long as an input image possesses radiometric characteristics—specifically contrast and SNR—that map to coordinates within the designated green zone of the performance graph, it may be asserted with high statistical confidence that the corner detector will function accurately, correctly identifying the requisite number of geometric features.

Through this rigorous simulation methodology, the functional mechanics of the corner detection operator have been fully described, and its operational boundaries rigorously demarcated. This resulting performance profile can be conceptualized as a 'Technical Datasheet' for the operator, analogous to the electrical specification sheets used for hardware components. This leads to the pivotal research question: is this simulation-derived model sufficiently robust to predict the performance of the operator when applied to uncontrolled real-world imagery? Specifically, if real-world images are acquired and their contrast determined along with their SNR values, will the detector behave exactly as predicted by the synthetic model? To validate this hypothesis, the experimental design requires ensuring that the fundamental radiometric parameters, Contrast and SNR, are identical in both the virtual and physical domains to the greatest extent possible.

## 4.5 Phase 3: Real-World Validation

This section provides the test of RQ_4. The characterisation of the preceding sections was derived entirely within simulation; the question now is whether it predicts the behaviour of the same operator on imagery from physical sensors.

To validate the predictive power of this "Datasheet," real-world imagery was captured using multiple camera sensors (Canon, Epson, Sony, Nikon) under varying lighting conditions. The characteristics (SNR and Contrast) of these real images were measured and mapped onto the synthetic performance curves.

### 4.5.1 Validation Case A: High Fidelity (Canon & Epson)

To empirically validate the predictive model within the high-fidelity domain, a specific experimental trial was conducted utilizing a Canon camera sensor. This hardware was selected to represent a 'best-case' scenario, characterizing the performance of high-quality optics under optimal photometric conditions. Two reference images were acquired in an environment with sufficient and uniform illumination to minimize sensor noise floor artifacts. Quantitative analysis of the resulting raw imagery revealed a robust signal profile, exhibiting a remarkably high Signal-to-Noise Ratio (SNR) of 79.591 alongside a strong radiometric contrast delta of 188. According to the performance maps established in the previous section, these parameters place the imagery firmly within the 'Green Zone' of the algorithm's operational envelope, theoretically predicting a state of high detection stability. To verify this prediction, a corresponding set of synthetic images was parametrically generated within the simulation

pipeline. These virtual counterparts were rigorously tuned to possess an identical SNR profile (79.591) and contrast ratio (188) to the physical samples. Subsequently, the standard corner detection operator was applied simultaneously to both the physically acquired and synthetically generated datasets to facilitate a direct, 1-to-1 comparative performance analysis.

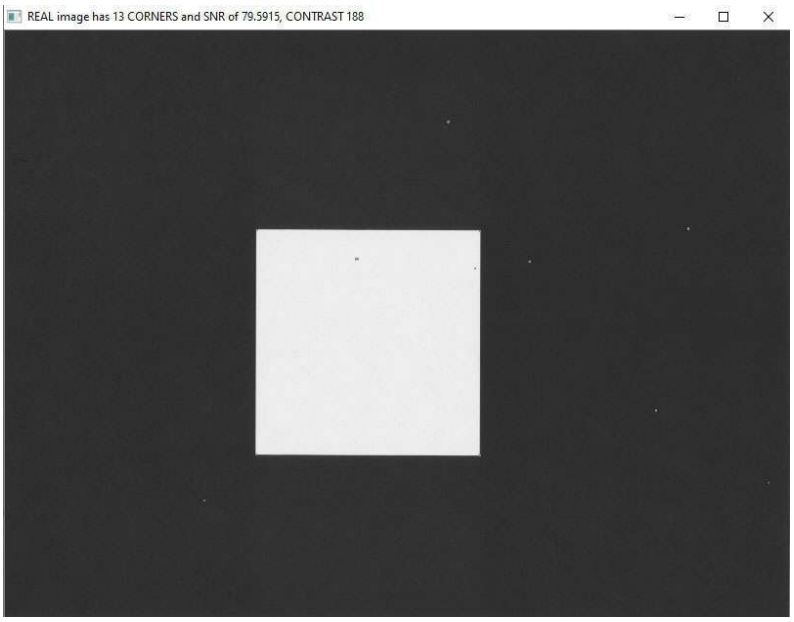


*Figure 42-Real image captured with an SNR of 79.591*

Upon applying the corner detection operator to the virtual dataset, across both the pristine and noise-injected iterations, the algorithm demonstrated ideal stability, correctly identifying the four geometric vertices with zero spurious detections. In sharp contrast, the analysis of the corresponding real-world imagery yielded a divergent result, where the detector registered an aggregate of 13 feature points.

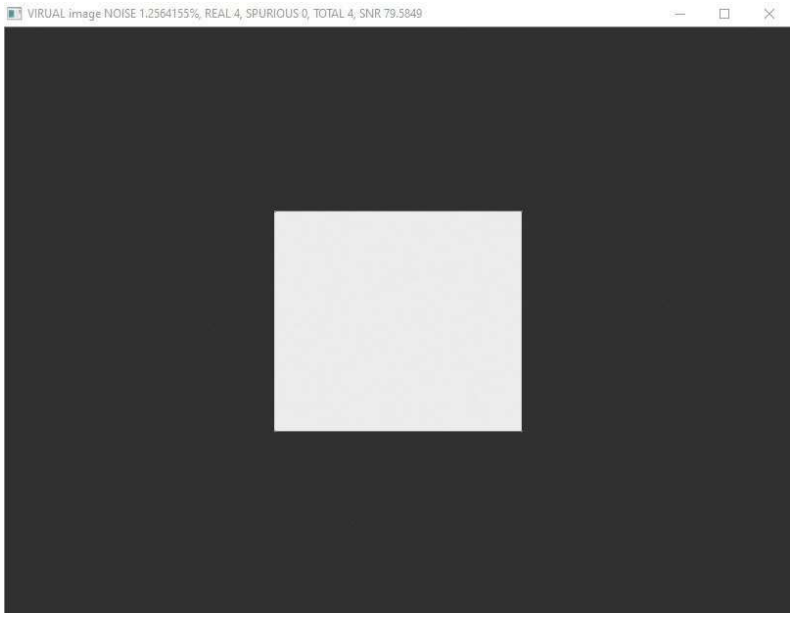


*Figure 43-VIRTUAL and REAL pictures with SNR 79.591*

While the geometric feature points within the rendered figure may appear visually indistinct due to display scaling, the critical quantitative metrics are explicitly reported within the application interface. For the remainder of this simulation series, the inclusion of visual figures will be discontinued. This methodological decision is predicated on the fact that the binary, high-contrast nature of the synthetic imagery provides limited semantic value for graphical inspection. Furthermore, the necessary downscaling required for document formatting results in a loss of high-frequency detail, rendering visual analysis ineffective.

A rigorous forensic analysis of the processing results reveals a critical nuance in the performance data. Although the quantitative metrics, specifically the Signal-to-Noise Ratio (SNR) and radiometric contrast, place the image firmly within the stable 'Green Zone' of the operational envelope defined in Figure 41, the empirical output deviates from the theoretical prediction. The analysis indicates that the physical scene is contaminated with unmodeled high-frequency artifacts, specifically particulate matter such as dust residing on both the background and foreground substrates. The corner detection algorithm, operating strictly on local gradient

intensity, erroneously validates these micro-features as geometric vertices. Consequently, while the detector successfully resolves the four veridical corners of the target square, it simultaneously registers an additional nine spurious feature points. These false positives are directly attributable to physical debris present on the paper surface during the photographic acquisition process, highlighting the algorithm's sensitivity to real-world environmental imperfections that are absent in a pristine simulation.

To extend the validation of the predictive model beyond a single hardware architecture, a secondary acquisition device—specifically an EPSON camera sensor—was utilized to capture reference imagery under optimal photometric conditions. Quantitative analysis of this dataset revealed a high-fidelity signal profile, characterized by a Signal-to-Noise Ratio (SNR) of 70.288 and a robust radiometric contrast value of 239. In accordance with the established experimental protocol, a corresponding set of synthetic images was parametrically generated to mirror these specific values. By enforcing an identical SNR and contrast profile on the virtual replicates, a controlled baseline was established for a direct 1-to-1 algorithmic comparison.

The results of the simulation phase were entirely consistent with the 'Green Zone' predictions, the corner detection operator, when applied to the virtual dataset (both in its pristine state and following noise injection), demonstrated absolute stability. The algorithm correctly identified exactly 4 discrete feature points—corresponding to the true geometric vertices—with zero spurious detections recorded. However, the analysis of the empirically acquired Epson imagery revealed a slight divergence from this theoretical perfection. While the signal quality was objectively high, the detector registered an aggregate of 5 feature points in the real-world image, representing a singular deviation (one spurious detection) from the geometric ground truth.

A rigorous forensic inspection of the resulting imagery was conducted to reconcile the minor discrepancy between the predicted and observed feature counts. The analysis suggests that while the quantitative metrics—specifically the robust Signal-to-Noise Ratio and high contrast—unambiguously situate the sample within the stable 'Green Region' of the performance envelope defined in Figure 41, the physical environment introduces unmodeled complexity. Specifically, the real-world sample contains a singular surface anomaly: a central particulate artifact, or speck of dust, adhering to the foreground of the target object.


*Figure 44-REAL pictures with SNR 70.280*

This high-contrast imperfection creates a local spatial gradient sufficient to trigger the detection kernel, thereby elevating the total feature count to five. It is imperative to note that the detector functioned correctly regarding the target geometry; the four veridical corners were resolved with high precision. However, superimposed upon this correct solution is a single

spurious detection attributable to the particulate contamination. The physical presence of this artifact, and its impact on the detection topology, is visually corroborated in Figure 44**.**

In conclusion the experiment can be summarised as follows:

**Real Data:** Images captured with Canon and Epson sensors yielded high SNRs (79.59 and 70.28) and high contrast (188 and 239)[12].

**Prediction:** The simulation predicts these values fall deep within the "Green Zone," implying perfect detection (n=4).

**Result:** The real-world detector returned 13 corners for Canon and 5 for Epson. While 4 true corners were found, the additional "spurious" corners were identified as physical artifacts, specks of dust on the paper.

**Conclusion:** The algorithm behaved as predicted, the error lay in the physical scene's cleanliness, not the sensor model.

### 4.5.2 Validation Case B: Catastrophic Failure (Nikon Low-Light)

In this experimental trial, a Nikon camera was used to acquire reference images under standard lighting conditions. Analysis of the captured data determined a Signal-to-Noise Ratio (SNR) of 15.091 and a contrast value of 141. Following the standard procedure, virtual images were generated to strictly replicate these specific SNR and contrast levels, enabling a direct comparison between the simulated prediction and the real-world outcome.

The application of the corner detector to the synthetic images produced the expected ideal result: in both the initial and noise-injected simulations, the algorithm correctly located the 4 geometric corners without any errors. However, the performance on the real-world Nikon image differed significantly, with the detector reporting a total of 12 corners.

A visual analysis of this discrepancy reveals a notable failure mode. Although the measured SNR and contrast values theoretically place this image within the 'Green Region' of the performance map (Figure 39), (suggesting accurate operation), the real-world detector failed to locate the actual target. Instead, all 12 detected corners were spurious (false positives). Close inspection suggests this error is not due to noise, but rather a lack of edge definition. A slight optical blur at the boundary between the foreground and background prevented the detector from resolving the sharp gradients required to identify the true corners. This effect is visually demonstrated in the Figure 45 below.

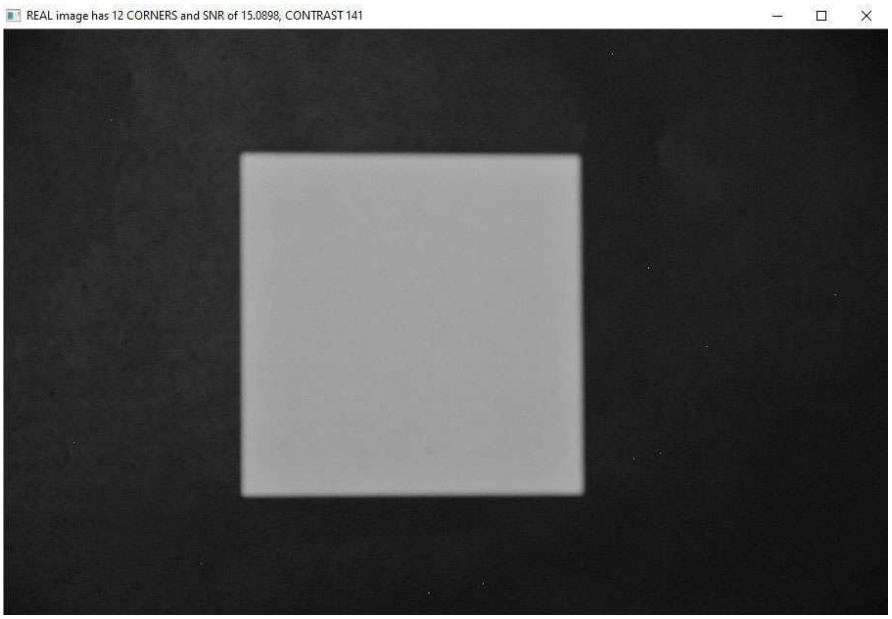


*Figure 45-VIRTUAL and REAL pictures with SNR 15.091*

The 4 real corners are not even detected on the real image, all 12 corners being spurious and due to the blur of the image, as well as specks of dust and other real world artefacts.

In this subsequent experimental phase, the Nikon camera was subjected to suboptimal lighting conditions to stress-test the algorithm's lower limits. Similar to the preceding trial, the resulting physical imagery exhibits a soft optical blur, likely due to the aperture/shutter settings required for low light. Quantitative analysis yielded a critically low Signal-to-Noise Ratio (SNR) of 3.9526 and a minimal contrast value of 13. Consistent with the established protocol, synthetic counterparts were generated to strictly replicate these specific radiometric parameters, enabling a direct comparison between the theoretical model and the physical outcome.

The application of the corner detector to the synthetic baseline (prior to noise injection) correctly yielded the 4 geometric corners. However, once the calibrated noise profile was applied, the detection count rose to 22, comprising the 4 valid corners alongside 18 spurious artifacts. This degradation is consistent with the predictive model outlined in Figure 38; the combination of such low contrast and minimal SNR situates this test case at the extreme lower boundary of the 'Green Zone.' In this marginal operational region, the signal strength is barely sufficient to override the noise, making the emergence of spurious corners a statistically probable outcome.

Analysis of the real-world imagery confirms the severity of these degraded conditions. The detector registered a total of 13 corners, yet significantly, *none* of these corresponded to the actual target geometry. This total failure to lock onto the true features is attributed to the compound effect of the low signal quality and the optical blur, which sufficiently softened the edge gradients to prevent valid detection.

The summary of the experiments conducted can be seen in the Table 8 below.

*Table 13-Simulating a match between real and virtual images*

| Contrast | SNR (real) | Total corners | SNR (virtual) | Real corners | Spurious corners | Camera |
|---|---|---|---|---|---|---|
| 188 | 79.591 | 13 | 79.591 | 4 | 0 | Canon |
| 239 | 70.288 | 5 | 70.288 | 4 | 0 | Epson |
| 165 | 37.216 | 165 | 37.216 | 4 | 0 | Sony |
| 141 | 15.091 | 12 | 15.091 | 4 | 0 | Nikon (light) |
| 13 | 3.952 | 12 | 3.952 | 4 | 18 | Nikon (dark) |

Real images have a certain amount of noise in the background, and this noise seems to be blurred which is why the corner detector yields more corners in the real image than the virtual image. The virtual generated images are simpler and without any blurred background noise. This simplicity means that the corner detector will yield fewer corners in the virtual image than in the real image. The solution to this is to increase the complexity of the virtual image. This is done by applying a sufficing amount of noise to the virtual images to get an SNR of the pictures that is similar to the SNR of the real pictures, but this noise will be blurred to match closer the way the noise is being acquired in a real picture. This yields a virtual picture that is a closer match for the real one. The picture below (Figure 46) shows a comparison between the virtual pictures created with and without blurring the noise.

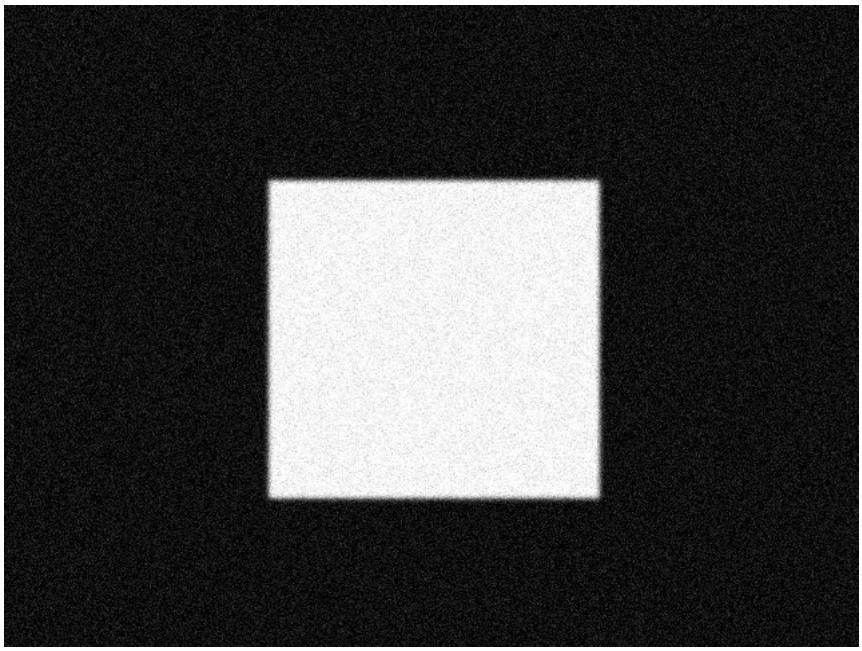

*Figure 46-Virtual Image with random noise applied*

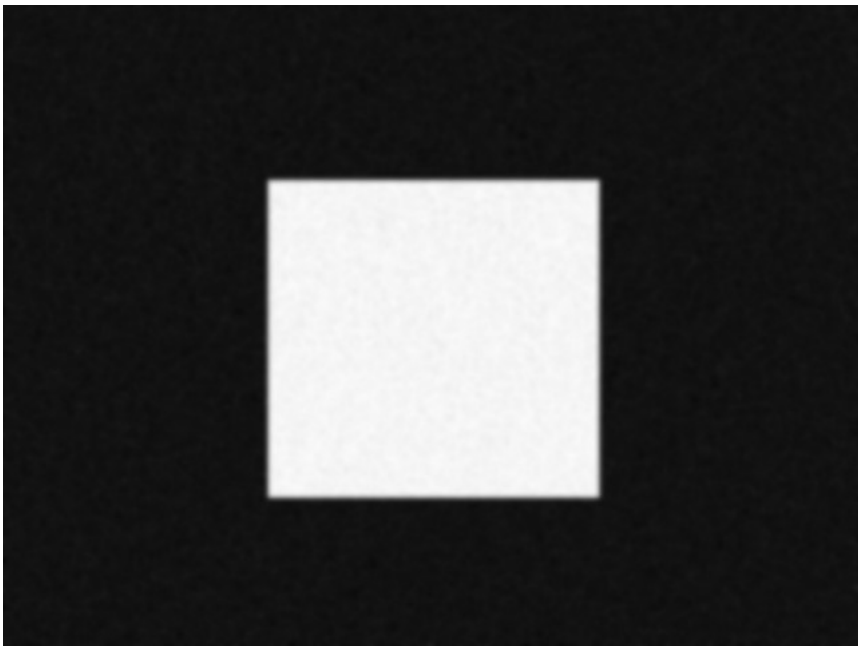

*Figure 47-Virtual Image with blurred random noise applied*

Both images were created to be a match for a real pair of images. It was expected that the corner detector would work the same on real images as it would on virtual images. The problem was that the virtual images despite having the same SNR as the real images, they would report by far more or less corners compared to their real counterpart. This was the case when virtual images looked like the one in Figure 45. The same amount of noise was applied to both Figure 46 and Figure 47, so they both have the same SNR, but Figure 46 has a grainier noise than Figure 47. This will lead to the corner detector finding more corners in that picture compared to the blurred picture.

The blurred picture in Figure 48b is a virtual picture that is one step closer to match the real pictures. In reality there will always be some amount of blur present in the picture, and this blur also needs to be simulated in the virtual picture. The procedure adopted to generate virtual images to closer resemble their real counterpart is try to detect the amount of blur in the real pictures. This has been done by measuring the gradient of values between the background and the foreground and applying enough blur to the virtual image to obtain virtual images with the same blur gradient as the virtual images. A comparison between of how a virtual image was generated before and after the application of a matching blur can be seen below in Figure 48.

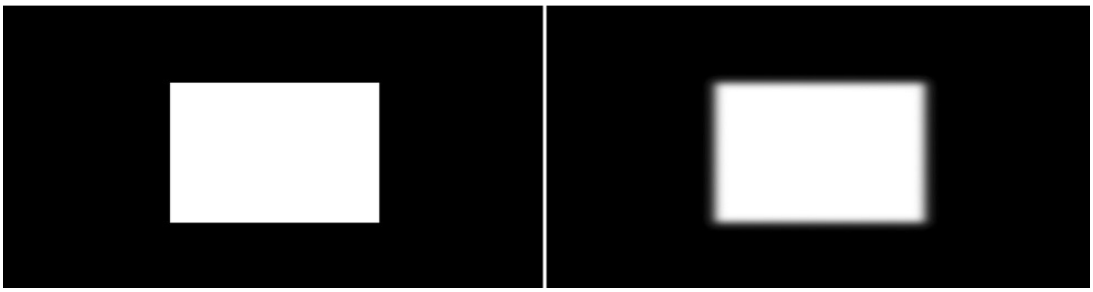

*Figure 48-Virtual Image with blurred random noise applied (a and b)*

The second picture has a blur level applied that is of the same quantity as the one being detected in the real image. The next step in the process would be to make this blurred virtual imaged match the SNR of the real image. This step is achieved by adding increasing levels of noise to the virtual image to match the SNR of the real image. But this added random noise should also be blurred so as to more closely mimic the noise coming from the lens of the real camera. Below Figure 49 and 50 present a matched pair of images that are a close match for each other; they have the same SNR, just that the first one has been virtually generated after the 2nd image which was a real image captured with a camera.

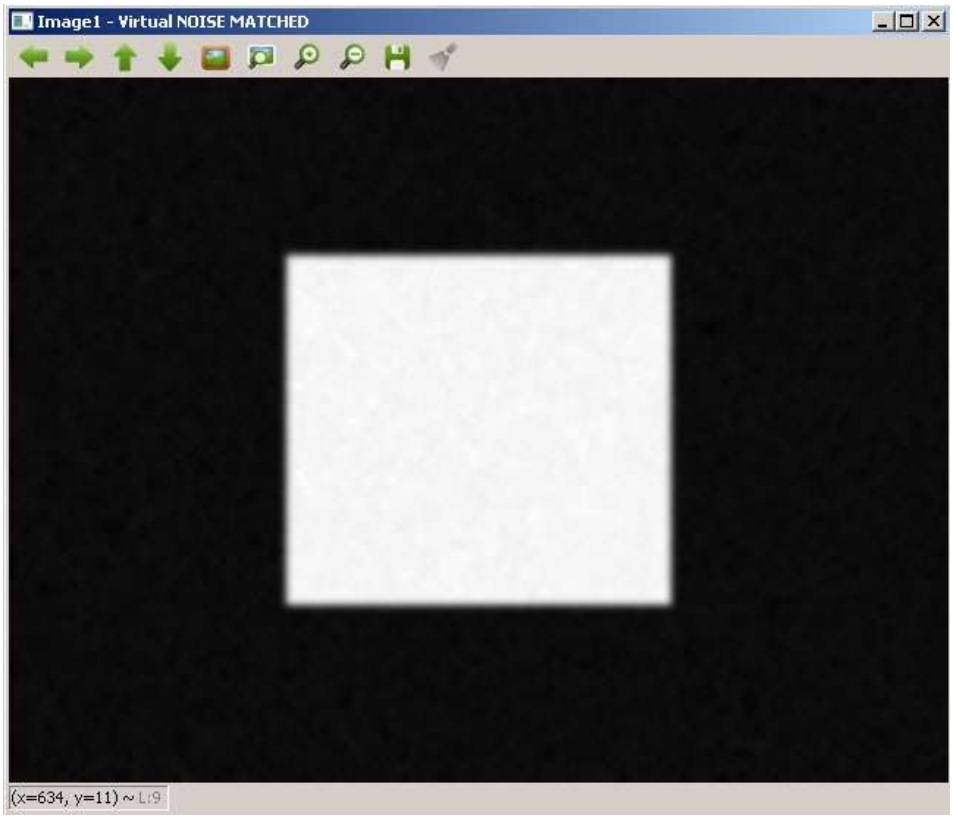


*Figure 49-Virtual Image with a matched SNR of 25*

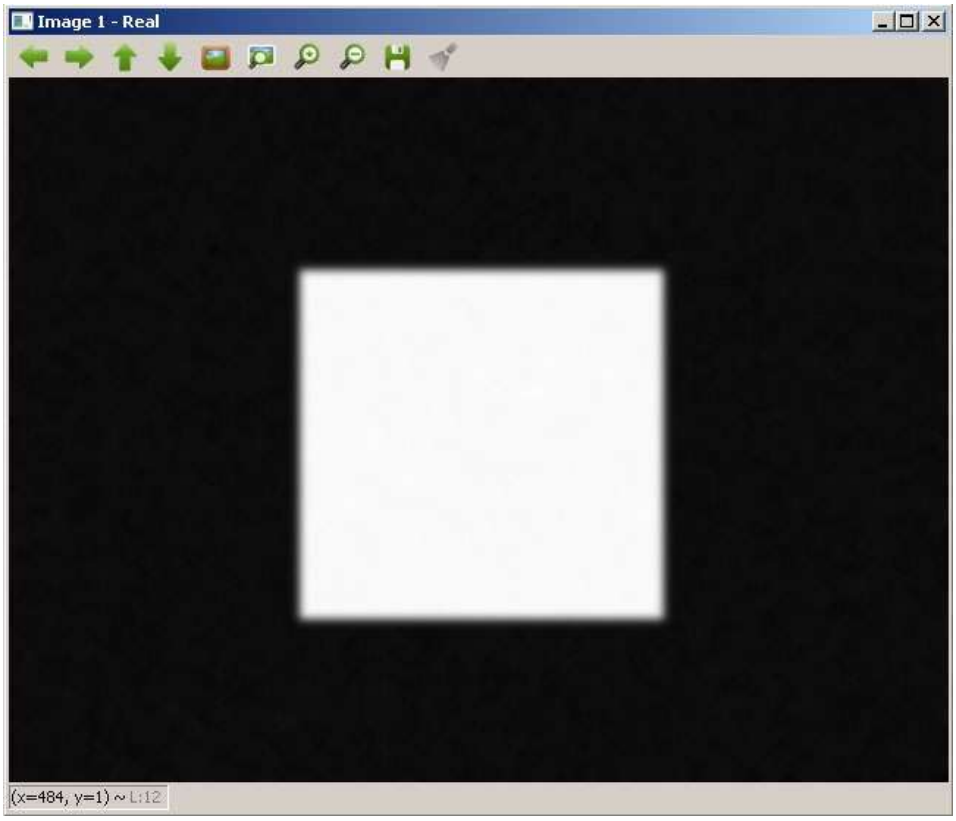


*Figure 50-Real Image with an SNR of 25*

Both these images look very much alike, which is exactly what was intended. The purpose was to create synthetic images that could replace real images. Now the virtual image has the same amount of blur and noise (same SNR) as the real images. The next step is to see how the corner detector will behave on these images, will it find the same amount of corners or not?

Running a simulation for a pair of real images with an SNR of 25, a corresponding pair of simulated images is generated with the same characteristics as the real image. The corner detector seems to find a similar number of corners. This can be seen in the pictures (Figure 51) below.

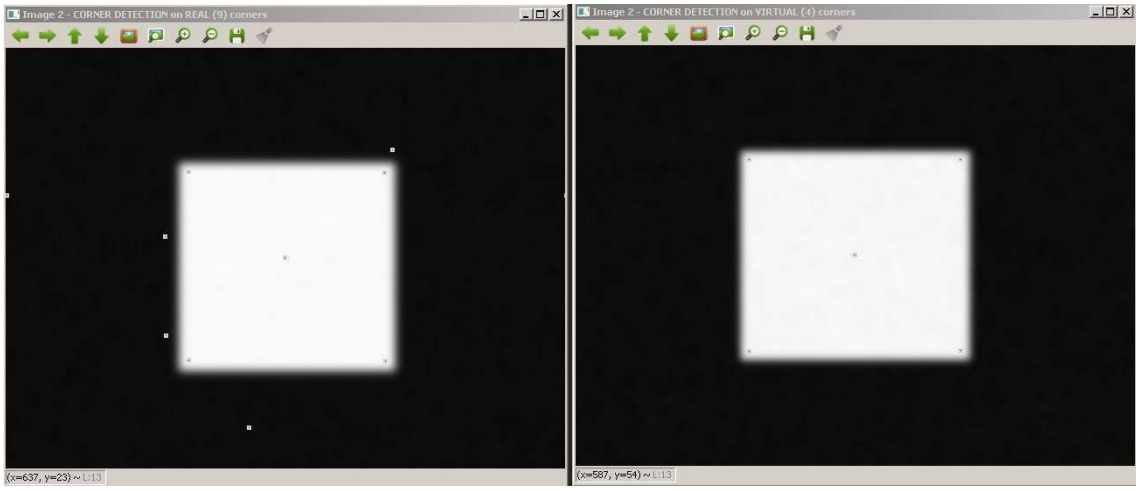


*Figure 51-Corner detector on a matched pair of Real and Virtual images*

In Figure 51, on the left sits the real image where the corner detector has found 9 corners being detected and on the right sits the virtually generated image that has been made to resemble the real image as closely as possible. On this image (Figure 51), the corner detector has found 4 corners present. The difference of 5 extra corners found in the real image amounts to imperfections of the real image capturing process and the camera with which the real picture was taken. Further time could be spent to model these real imperfections in the virtual image, but it was not judged necessary for the present purpose. Virtual images have reached a close enough resemblance to show that at the level of computer vision operators, they could be a substitute for real images.

There are a few boundaries that apply both to the real and virtual images in terms of how well computer vision operators would work on them. These boundaries are comprised in the SNR of the pictures. Low lighting in the pictures, low or high contrast also influence how well the corner detector would work. This has been already shown. Low SNR, which implies more noise in the pictures, will always yield more spurious corners, both in the real images as

well as in the real images generated to match them. Running a few more simulations begins to show this trend.

*Table 14-Simulating a match between real and closer resembling virtual images*

| Contrast | SNR (real) | Total corners (real image) | SNR (virtual) | Total corners (virtual image) | Camera |
|---|---|---|---|---|---|
| 188 | 79 | 4 | 78 | 4 | Canon |
| 239 | 70 | 4 | 69 | 4 | Epson |
| 165 | 37 | 113 | 38 | 71 | Sony |
| 141 | 15 | 205 | 16 | 245 | Nikon (light) |
| 13 | 3 | 2569 | 3 | 1974 | Nikon (dark) |

The Table 9 above shows evidence that when dealing with real images of high enough SNR the corner detector would return the real number of corners present in the picture. For the same SNR, the virtual matched images seem to be returning the same number of corners. With a continuous decrease of the SNR in the real image, a decline is observed in the ability of the corner detector to find the real number of corners present in the image. The same pattern is followed when detecting the corners in the matched virtual image. Spurious corners begin to increase from an SNR of about 30. This also depends on the contrast of the image as it has been shown a few paragraphs above, but the contrast does not seem to have such a big influence on the corner detector.

The experimental data confirms that when operating with a sufficiently high Signal-to-Noise Ratio (SNR), the corner detector accurately identifies the true number of corners in real images. Significantly, the matched virtual images exhibit identical performance at these same SNR levels. As the SNR decreases, a consistent decline in detection accuracy is observed in the real imagery, a pattern that is faithfully replicated in the virtual dataset. The data indicates that spurious corner detection begins to accelerate when the SNR drops to approximately 30. While image contrast is a variable, the analysis suggests that SNR fluctuations are the primary determinant of detector performance. Provided there is a minimum contrast difference of 10 values between the background and foreground, the corner detector performs identically on both real and virtually generated images sharing the same SNR.

### 4.5.3 Quantitative agreement between domains

The five matched configurations of Table 9 permit the correspondence between domains to be stated numerically rather than by inspection. Agreement was assessed on three measures, reported in Table 15 below.

| Measure | Value | Interpretation |
|---|---|---|
| Spearman rank correlation ρ | **1.000** (exact two-sided p = 0.017) | The ordering of the five configurations by detector instability is reproduced without a single inversion |
| Pearson r, $\log_{10}$ counts | **0.996** ($p \approx 0.0003$) | Agreement holds across three orders of magnitude of corner count |
| Pearson r, raw counts | 0.999 | Reported for completeness; dominated by the largest observation and therefore less informative than the rank and logarithmic measures |
| Mean absolute $\log_{10}$ ratio | **0.079** | Mean multiplicative discrepancy of 1.20×; synthetic counts agree with real counts to within approximately 20% |
| Maximum SNR matching error | **1.0 unit** | Calibration achieved the target ratio to within one unit in all five configurations |

*Table 15 - Statistical agreement between real and calibrated synthetic corner counts (n = 4)*

The rank correlation is the measure of principal interest, because the predictive claim made in this thesis concerns the ordering of operating conditions by severity rather than the exact count returned in any one of them. A perfect rank correlation across five configurations spanning signal-to-noise ratios from 3 to 79 indicates that the calibrated simulation orders operating conditions exactly as the physical sensors do.

The limitations of these figures should be stated with equal clarity. With five configurations the statistical power is low: an exact permutation test gives a two-sided probability of 0.017 for a perfect rank correlation, which is significant at the conventional threshold but rests on a sample too small to characterise the distribution of the discrepancy. The two configurations at high signal-to-noise ratio return identical counts of four in both domains and therefore contribute no information about the ordering, so the correlation is effectively carried by three points. These results are consistent with the hypothesis and are not a test of it at any useful power.

The effect of blur calibration may be quantified by comparing Tables 6 and 7. Under signal-to-noise matching alone, the synthetic corner count varied over a range of roughly 5.5-fold while the corresponding real counts varied 33-fold: the synthetic imagery failed to reproduce the degradation at all. Following the addition of blur matching and noise correlation, the synthetic count varied 494-fold against 642-fold in the real imagery. The calibration therefore accounts for approximately two orders of magnitude of previously unmodelled dynamic range, which is the quantitative statement of the result argued qualitatively in Chapter 4.6.

## 4.6 Closing the Reality Gap: Texture and Blur Modelling

While the validation confirmed the general trends, a discrepancy remained in the exact counts of spurious corners. The simulation initially assumed "crisp" noise (pixel-independent), whereas real-world camera noise is often correlated to lens blur.

### 4.6.1 Refinement via Blur Injection

To close this gap, the simulation pipeline was refined. A Gaussian blur was applied to the synthetic noise model to mimic the optical characteristics of a real lens.

- **Effect:** Blurring the noise reduced the high-frequency gradients that the corner detector falsely interpreted as features.
- **Outcome:** When the synthetic image was matched to the real image not just in SNR, but also in *blur profile*, the corner counts converged. For the SNR 25 test case, the refined simulation returned 4 corners, closely matching the real-world performance (after accounting for physical dust).

# 4.7 Conclusion

| Failure mode | Chapter | Mechanism | Real | Synthetic | Reproduced by simulation? |
|---|---|---|---|---|---|
| Feature starvation | 3.5.2, 3.5.6 | Insufficient intensity gradient for keypoint extraction on untextured surfaces | Yes | Yes | Yes |
| Specular violation | 3.5.3 | Whiteboard appearance varies with viewing angle, violating brightness constancy | Yes | No | No — Lambertian model does not produce it |
| Radial distortion smearing | 3.5.4 | Fisheye projection compresses features toward the frame edges | Yes | Not tested | Untested |
| Computational ceiling | 3.5.5 | Dataset exceeds available memory during reconstruction | Yes | n/a | Not applicable |
| Noise-induced false positives | 4.3 | Response field elevated above threshold by stochastic gradients | Yes | Yes | Yes, once noise correlated |
| Contrast indeterminacy | 4.4.1 | Signal insufficient to trigger the response threshold at any noise level | Yes | Yes | Yes |
| Blur-induced misses | 4.5.2 | Edge gradients softened below the detection threshold | Yes | Initially no | Only after blur matching |
| Particulate artefacts | 4.5.1 | Physical debris produces genuine local intensity gradients | Yes | No | No — scene is clean by construction |

*Table 16 - Failure modes observed, with mechanism and domain of occurrence*

Table 16 separates the failure modes the simulation reproduces from those it does not, and this separation is the substantive result. Two categories are absent from the synthetic

domain by construction rather than by oversight: non-Lambertian reflectance, which the material model does not represent, and physical contamination, which a rendered scene cannot contain. Both were the source of divergence in Chapter 4, and neither is remediable by adjusting signal-to-noise ratio or blur.

This chapter successfully establishes a methodology for the "Technology Evaluation" of computer vision operators. By stressing the algorithm in a controlled synthetic environment, a definitive "Performance Map" was generated. The validation against real-world cameras confirms that this map is predictive: if the SNR and Contrast of a real-world scene are known, the performance of the vision system can be predicted without physical testing.

Furthermore, this investigation highlighted that simple parameters (SNR/Contrast) are insufficient for perfect replication; second-order effects such as lens blur and sensor texture must also be modelled to achieve a true Digital Twin. This lays the foundation for the final validation of complex SLAM systems in the next chapter.

# Chapter 5: Validation of Synthetic Imagery for Computer Vision Operators

## 5.1 Introduction

This chapter addresses RQ_5 and tests hypothesis H3. Chapter 4 established that matching signal-to-noise ratio alone produced systematic disagreement between real and synthetic corner counts. This chapter determines what further optical property must be replicated to close that gap, and then examines, in preliminary form, whether the resulting calibration procedure extends beyond corner detection to a composite operator.

This chapter investigates the viability of using synthetic imagery as a robust substitute for real-world data in computer vision systems. The primary objective is to demonstrate that by accurately simulating optical imperfections—specifically Signal-to-Noise Ratio (SNR) and lens blur—virtual images can yield performance metrics comparable to those of real images. This validation process focuses on two fundamental computer vision operators: corner detection and ORB (Oriented FAST and Rotated BRIEF) feature matching.

## 5.2 Simulation of Optical Imperfections

To create a virtual image that effectively mimics a real counterpart, it is insufficient to merely reproduce geometry; optical artifacts must also be simulated. Initial attempts using random

noise demonstrated that while virtual images could match the SNR of real images, they often possessed "grainier" noise characteristics that resulted in discrepancies during corner detection.

The methodology for rectifying this involved two key steps:

1. **Blur Simulation:** By measuring the gradient of values between the background and foreground in real images, an equivalent level of blur was applied to the virtual images. This ensures the virtual lens characteristics match the physical reality.

2. **Noise Matching:** Following blur application, random noise was added to the virtual image to match the SNR of the real image. Crucially, this added noise was also blurred to mimic the noise signature inherent to a real camera lens.

### 5.2.1 The ORB operator: mechanism, assumptions and limitations

ORB combines an oriented variant of the FAST detector with a rotation-steered BRIEF descriptor. Its three stages are as follows.

**Detection.** FAST examines a Bresenham circle of sixteen pixels at radius 3 about a candidate $p$, and declares a corner where at least nine contiguous pixels on that circle are all brighter than $I(p) + t$ or all darker than $I(p) - t$, for an intensity threshold $t$. Scale is addressed by repeating the test across an image pyramid, here of eight levels with a scale factor of 1.2.

**Orientation.** Rotation invariance is obtained by assigning each keypoint the direction of the intensity centroid of its patch. With moments $m_{pq} = \Sigma_{x,y} x^p y^q I(x, y)$, the orientation is

$$\theta = \mathrm{atan2}(m_{01}, m_{10}) \qquad \text{(Equation 18)}$$

**Description.** BRIEF forms a binary string by comparing intensities at a fixed set of point pairs within the patch, the pattern being rotated by θ before sampling. Descriptors are compared by Hamming distance.

The assumptions are brightness constancy between views, local planarity and rigidity of the patch, and in-plane rotation. The limitations follow from the mechanism. The FAST threshold $t$ is an absolute intensity difference, so detection is directly contrast-dependent in a way the Harris response is not. BRIEF rests on pairwise intensity comparisons at individual points, making it more sensitive to noise than descriptors formed by pooling gradients over a region, since a single perturbed sample flips a bit of the descriptor. Invariance extends to in-plane rotation and to the discrete scales of the pyramid, but not to affine deformation or to viewpoint change of any magnitude.

A relationship between the two operators should be stated plainly, since it bounds what Chapter 4 establishes. The detector characterised in Chapter 4 is Harris, whereas ORB employs FAST. The two apply different decision rules , a gradient-autocorrelation response thresholded relative to the image maximum in the first case, a contiguous-arc intensity comparison against an absolute threshold in the second, and there is no reason to expect their failure boundaries to coincide numerically. The Safe Operating Area derived in Chapter 4.4.1 is therefore a characterisation of Harris specifically, and is not transferable to ORB without repeating the procedure. Producing the equivalent characterisation for FAST, and hence for the detector actually employed by ORB-SLAM, is identified in Chapter 6.4 as the most immediate extension of this work.

## 5.3 Comparative Analysis: Corner Detection Performance

The efficacy of the matched virtual images was tested using a corner detector. The hypothesis stated that if the SNR and blur were correctly modelled, the detector should identify a similar number of corners in both real and virtual datasets.

ORB is a local feature detector that can be used in computer vision for recognition and tracking. My goal is to show that this operator can work with simulated imagery as well as it can work with real world imagery. The purpose is to show that simulated imagery could eventually become a substitute for real world imagery. To show that this is the case, the ORB operator will need to be characterized in the same way as the corner detector operator has been characterized. A map of how it functions will need to be established, together with its boundaries and limitations. Only once this has been achieved, a move towards trying to see if simulated imager could be used instead of real imagery.

In order to see how ORB's feature matching works, two simple images are taken and their features will be matched.

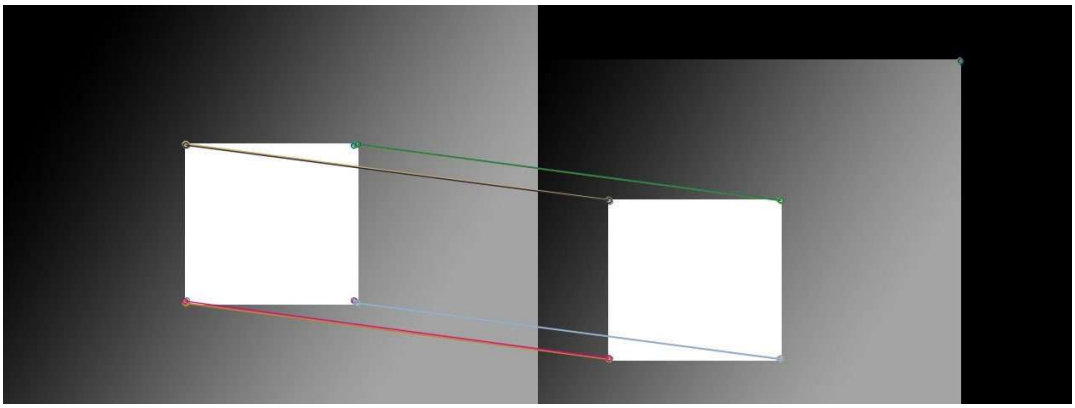

*Figure 52-ORB translation*

In order to see how feature matching works, the simplest way to do it is to choose a simple image and use it to match its features to itself, but change something to the picture so that it is not exactly the same. We can translate the 2nd picture of Figure 52 to see if the operator will still be able to match features from 1st picture to the 2nd. In the present example the operator succeeds. In order to increase complexity, the next step would be to rotate the 2nd image to see if features matching is kept.

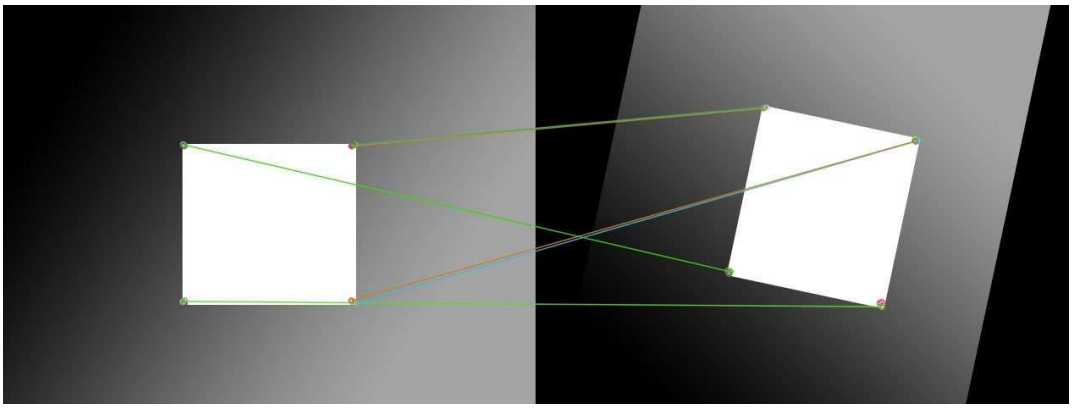

*Figure 53-ORB rotation*

The ORB operator works with rotation as well (Figure 53). The common features are found in the same place and matched from picture to picture. In order for this to work, the background needs to have some complexity, it cannot be a solid color, as otherwise the matching can be misaligned. The results until now seem promising, the ORB operator works as expected. The next step would be is to introduce noise in the images and see to what extent the operator works and matches features accurately. It is expected that there will be a limit, or a breaking point, or a region beyond which no matter how much or less noise is applied to the image, the feature matching will not work. These boundaries will be explored further on in the same way as the Corner Detector operator was explored beforehand.

## 5.4 Extension to Feature Matching (ORB)

This section examines, in preliminary form, whether the calibration procedure established for corner detection extends to a composite operator. It bears on RQ_5 but does not resolve it for ORB, for the reasons stated in Chapter 5.5.

Following the validation of corner detection, the study extended to the ORB feature matching operator to determine if simulated imagery could support recognition and tracking tasks.

- **Translation Testing:** Initial tests involved matching features between a base image and a translated version of itself. The operator successfully maintained feature matches across the translation.
- **Rotation Testing:** Subsequent tests introduced rotation. The ORB operator successfully tracked common features, provided the background possessed sufficient complexity to prevent misalignment.

These preliminary results suggest that the established simulation methodology can be successfully applied to more complex operators like ORB, paving the way for testing Simultaneous Localization and Mapping (SLAM) systems.

## 5.5 Conclusion

This chapter has demonstrated that it is possible to predict the performance of a vision system on real images by validating it against carefully constructed virtual counterparts. This validates a shift away from the "Dyson approach" characterized by laborious hardware tweaking toward a design methodology based on engineering simulation principles. This approach allows for the variation of camera parameters, lighting, and noise in a controlled virtual environment, leading to potential time and cost savings in vision system development.

# Chapter 6: Conclusion

## 6.1 Summary of the Research Problem

This thesis has addressed a fundamental limitation in the development and evaluation of Computer Vision systems: the field's historical over-reliance on "Scenario Evaluation." By traditionally tailoring vision systems to specific, isolated environments, researchers and engineers have struggled to predict how these systems will perform when environmental variables, such as lighting, texture, and noise, inevitably change. Recognizing the high costs and scalability issues associated with extensive real-world data collection, this work proposed a shift toward a rigorous framework of "Technology Evaluation," driven by high-fidelity simulation and the construction of Digital Twins.

## 6.2 Methodological Contributions

The principal contribution of this work is a framework for the Technology Evaluation of computer vision operators, supported by optically calibrated Digital Twins. The framework comprises four stages, each of which is transferable to operators and environments other than those examined here.

The first stage is the construction of a geometrically validated Digital Twin. Chapter 3 demonstrates that this cannot be treated as a matter of modelling alone: the acquisition parameters governing reconstruction fidelity must themselves be established empirically. The finding that surface texture dominates image volume, and that a saturated dataset degrades

rather than improves the outcome, is a methodological result in its own right, and one that inverts the intuition with which the experimental programme began.

The second stage is the characterisation of an operator across a controlled stress space. By sweeping injected noise against radiometric contrast over the full range of the latter, the response of the corner detector is mapped as a continuous manifold rather than sampled at isolated operating points. The resulting Safe Operating Area, with its three distinct regions of reliable operation, failure, and mathematical indeterminacy, constitutes a specification of the operator analogous to the characteristic curves supplied with a hardware component.

The third stage is the optical calibration of synthetic imagery against a measured physical sensor. This is the stage on which the claim to novelty principally rests. Synthetic imagery is matched to real imagery not by visual appearance nor by geometric correspondence, but by measured signal-to-noise ratio and measured blur gradient, with the injected noise itself blurred to reproduce the spatial correlation characteristic of a physical lens. Prior work in simulated camera error, notably that of Hinkenjann et al. [12] and Asanuma et al. [13], established that simulating sensor imperfection changes algorithmic outcomes; it did not calibrate those imperfections against quantities measured from a specific physical camera.

The fourth stage is the verification of predicted failure boundaries against physical imagery. The characterisation derived in simulation is tested against real captures whose radiometric properties have been measured and matched. This closes the loop that the third stage opens, and it is the combination of the two that distinguishes the framework from both the synthetic dataset literature, which supplies imagery without calibration to a specific sensor, and the industrial sensor simulation tools discussed in §2.6, which validate image fidelity without establishing where a downstream operator fails.

The framework's value lies in its transferability. Nothing in stages three and four is specific to corner detection: the calibration procedure makes no assumption about what subsequently consumes the image, and the same protocol could be applied to a learned detector, a descriptor matcher, or any operator whose output can be evaluated against a known ground truth.

## 6.3 Synthesis of Findings

The research questions posed in §1.2 may now be answered directly, with the strength of evidence stated in each case. RQ_1 is answered affirmatively: the cloud-to-cloud comparison of §3.7 returned a final RMS error of 0.74635, with the majority of reconstructed surfaces falling within the lowest error bracket, supporting hypothesis H1 for the environments tested. RQ_2 is answered by the acquisition protocol established in Chapter 3.5.9, which identifies surface texture rather than image volume as the dominant variable and locates a practical optimum at 540 images captured at 8 megapixels. RQ_3 is answered by the Safe Operating Area of Chapter 4.4.1, which bounds the corner detector's reliable operation as a region of the noise-contrast plane and identifies a third region in which detection is undefined rather than merely unreliable. RQ_4 is answered with qualification: for the five camera configurations tested, the ordering and approximate magnitude of degradation predicted in simulation were reproduced in real imagery, though absolute corner counts diverged, and the divergence was traced to physical scene artefacts and to unmodelled blur rather than to error in the characterisation. RQ_5 is answered by Chapter 4.6 and Chapter 5, which identify lens blur, and the spatial correlation of sensor noise, as the properties that must be replicated in addition to signal-to-noise ratio; hypothesis H3 is supported.

The validation of this simulation-based approach was systematically proven through the evaluation of low-level computer vision operators:

- **Corner Detection:** Experimental data confirmed a strong correlation between real and virtual performance. At high SNR levels, matched virtual images perfectly replicated the accurate corner counts of real images. As lighting conditions worsened and SNR decreased, the virtual datasets faithfully reproduced the exact performance degradation and the increase in spurious corners observed in the real-world data.
- **Feature Matching (ORB):** The study successfully extended this validation to ORB feature matching. Simulated imagery supported feature matching across translation and rotation. These results are preliminary: the ORB experiments tested geometric transformation only and did not extend to the noise and contrast sweep applied to corner detection, so no claim of equivalent behaviour under radiometric degradation is made for this operator.
- **Predicting System Failure:** Ultimately, the data established that SNR fluctuations—rather than pure contrast—are the primary drivers of detector performance. By controlling these variables in a virtual space, this research demonstrates that synthetic data can predict the failure modes of the low-level operators upon which more complex applications, including Visual SLAM, depend. Whether this predictive capability propagates to the behaviour of a complete SLAM system remains to be established, and is identified in §6.4 as the principal item of future work.

## 6.4 Limitations and Future Work

The reality gap identified in Chapter 2.7 is not a single deficiency but an aggregate of every physical phenomenon a renderer omits. This thesis addressed two of those phenomena, signal-to-noise ratio and optical blur, selected because they dominate the response of gradient-based operators, and calibrated both against values measured from physical sensors. The results of Chapter 4.5.3 indicate that closing these two accounts for a substantial part of the gap for the operators and conditions tested. It does not follow that the remainder is small. The sections below set out what was left unmodelled, what depends upon the particular algorithms examined, and what would be required to extend the framework beyond the conditions of this study.

The practical value of this work may be stated in terms of three distinct beneficiaries. For evaluation practice within the research community, the Safe Operating Area representation offers a means of specifying a vision operator in the manner of a hardware component. A system integrator in possession of a sensor's published noise characteristics, and an estimate of the contrast available in the intended operating environment, may determine from such a specification whether a given operator will function, without conducting a physical trial. This is a modest capability in isolation, but it is the capability that distinguishes an engineering discipline from an empirical one.

For industry, the work addresses a stage that current commercial tooling does not cover. As set out in Chapter 2.6, physics-based sensor simulation products validate that a simulated camera reproduces the calibration and colorimetric properties of a physical one [135][136]. That is a necessary result but not a sufficient one for a perception engineer, whose question is not whether two images are equivalent but whether the algorithm consuming them will behave

equivalently. The methodology developed here extends validation to precisely that question, and does so using measurements, signal-to-noise ratio and blur gradient, that are obtainable from any physical camera without access to proprietary lens designs.

For the economics of data collection, the room experiments of Chapter 3.5 provide a quantified account of the cost of physical capture in a representative indoor setting. Datasets of over one thousand images exceeded the available memory entirely and projected in excess of four days of processing, while a substantially smaller textured dataset achieved superior results. The identification of an efficient acquisition protocol, and the demonstration that fidelity is governed by scene texture rather than by data volume, is transferable to any photogrammetric capture campaign and does not depend on the simulation results that follow from it.

**Conditions of correspondence.** The experiments reported in this thesis permit the conditions under which synthetic imagery corresponded to physical behaviour to be distinguished from those under which it diverged. These are set out in Table 17 below.

| Condition | Correspondence | Evidence |
|---|---|---|
| Diffuse, matte, densely textured surfaces | Good | C2C agreement within the lowest error bracket, Chapter 3.7 |
| Controlled and static illumination | Good | Cube and room comparisons, Chapter 3.4 and 3.7 |
| Degradation dominated by sensor noise | Good | $\rho = 1.000$ across five configurations, Chapter 4.5.3 |
| Blur matched to the physical lens | Good | Convergence at SNR 25, Chapter 4.6 |
| Specular or non-Lambertian surfaces | Diverges | Whiteboard failure absent from the synthetic domain, Chapter 3.5.3 |
| Physically contaminated scenes | Diverges | Nine and one spurious detections attributable to dust, Chapter 4.5.1 |

| Condition | Correspondence | Evidence |
|---|---|---|
| Blur present but unmodelled | Diverges | Complete detection failure, real imagery only, Chapter 4.5.2 |
| Occluded or intersecting geometry | Diverges | Peak C2C error of 0.411 units at the cube–plane junction, Chapter 3.4.1 |
| Spatially varying blur across the frame | Untested | Uniform kernel only, Chapter 4.2.1 |
| Dynamic scenes, motion blur, rolling shutter | Untested | Static targets throughout |

*Table 17 - Conditions under which synthetic imagery corresponded to physical behaviour, and conditions under which it diverged*

The pattern is consistent. Correspondence holds where the physical scene approximates the assumptions of the rendering model, and fails where it does not, specifically where reflectance departs from Lambertian, where the physical scene contains structure absent from the model, or where an optical effect present in the capture has no counterpart in the render. Divergence in every observed case was traceable to a specific unmodelled physical phenomenon rather than to an inherent limit of the approach, which suggests the boundary is one of model completeness rather than of principle. That inference is not itself demonstrated here.

**Imaging phenomena not represented.** The two effects that were modelled carry limitations stated at the point where each is defined. Section 4.1.3 records that the signal-to-noise estimator assumes noise additive and independent of the signal, whereas the shot noise of a physical sensor is Poisson-distributed and therefore signal-dependent. Section 4.2.1 records that blur is applied uniformly across the frame, whereas a physical lens produces blur

varying with field position and focus distance. Beyond these, a number of phenomena were not represented at all, and are enumerated in Table 18.

| Effect | Physical mechanism | Why absent here | Expected consequence for the operator | Priority |
|---|---|---|---|---|
| Motion blur | Camera displacement during sensor integration | All captures tripod-mounted and static | Directional softening of edge gradients, reducing detection along the direction of travel | High |
| Rolling shutter | Line-sequential readout in CMOS sensors | Static scene and camera produce no skew | Geometric distortion of features during motion, corrupting the correspondences from which pose is estimated | High |
| Automatic exposure | Per-frame adjustment of gain and integration time | Fixed exposure maintained throughout | SNR varies between frames, so a single calibrated value is not valid across a sequence | High |
| Environmental variability | Changing daylight, moving occluders, dynamic content | Controlled indoor illumination only | Contrast and noise vary unpredictably, moving the operating point within the Safe Operating Area during operation | High |
| Non-linear radiometric response | Sensor tone curve and gamma encoding | Renderer output treated as linear in intensity | Alters the mapping between scene and image contrast, and hence the contrast axis of the performance map | Medium |
| Fixed-pattern noise | Sensor non-uniformity, constant across frames | Noise modelled as independent between realisations | Violates the independence assumption of Eq. 10; a component common to both frames is attributed to signal | Medium |
| Colour filter array and demosaicing | Interpolation across the sensor mosaic | Greyscale conversion applied before detection | Spatial correlation structure differs from the isotropic Gaussian kernel of Eq. 12 | Medium |
| Vignetting and field-dependent blur | Off-axis illumination falloff and aberration | Uniform blur kernel applied across the frame | Detection degrades more severely near the frame margin than at the centre | Medium |

*Table 18 - Imaging phenomena not represented in the simulation*

The four phenomena assigned high priority share a property: each varies within a sequence rather than being fixed at the moment of capture. The calibration procedure of

Chapter 4.2.1 matches a single static image pair and possesses no mechanism for a parameter that changes from frame to frame. Extending the framework to a moving platform therefore requires more than the addition of further effects to the renderer; it requires the calibration itself to become time-varying, which is a change of kind rather than of degree. Priority in Table 18 is assigned by relevance to the intended application rather than by magnitude of effect, the four high-priority phenomena being those that arise specifically from operating a camera on a moving platform.

**Dependence upon specific algorithms.** The conclusions of this thesis are tied to the operators examined, and the extent of that dependence should be stated. The Safe Operating Area of Chapter 4.4.1 characterises the Harris detector at one parameter setting. Section 5.2.1 records that ORB-SLAM employs FAST, which applies an absolute intensity threshold along a contiguous arc rather than thresholding a gradient-autocorrelation response, and there is no basis for assuming that the two boundaries coincide numerically. The comparison of Chapter 3.1.3 further establishes that a direct method such as DSO would not merely yield different boundaries but would confound the experiment, since its internal photometric calibration compensates for the degradation under test.

A comparison across operators is nonetheless the least expensive extension available, because it requires no further physical capture. The calibrated image pairs of Chapter 4.5 already exist; applying the same procedure to FAST, to Shi–Tomasi and to a learned detector such as SuperPoint would yield three further performance maps at the cost of re-running the analysis alone. Such a comparison would establish whether the Safe Operating Area is a property of gradient-based detection generally or of the Harris formulation specifically, and would determine whether the boundary measured here transfers to the detector that ORB-SLAM actually employs. It is accordingly identified as the first item of future work, ahead of

the system-level validation described below, on the grounds that it is the cheaper of the two and that its outcome bears upon how the second should be designed.

**Scalability of the framework.** The framework scales differently along each of the dimensions in which it might be extended, and these should be distinguished rather than treated together. Extension to further sensors is straightforward: the calibration requires only two exposures of a static target, from which signal-to-noise ratio and blur are measured, and imposes no requirement for access to the sensor's design or to laboratory equipment. Extension to further textures is similarly direct, subject to the caveat of Chapter 3.5.6 that only one surface texture was tested and no systematic comparison undertaken. Extension to further environments is bounded by the reflectance model rather than by scene size: the pipeline is indifferent to scale, but Chapter 3.5.3 established that specular surfaces are not represented, so any environment containing glass, polished flooring or displays will diverge in the manner of the whiteboard. Extension to alternative lighting models is the most demanding dimension, since the illumination used throughout was static, indoor and approximately uniform, whereas outdoor conditions vary in intensity, colour temperature and directionality within the span of a single capture. Extension to alternative rendering assumptions is discussed in Chapter 2.3.3, where the treatment of blur as a calibrated post-process rather than as a consequence of simulated optics is identified as the principal limitation of the renderer selected; implementing the calibration within an engine supporting explicit lens models would remove that limitation and would additionally supply the field-dependent blur recorded as absent in Table 18.

**Relationship to system-level performance.** The link between operator behaviour and the performance of a complete Visual SLAM system cannot be demonstrated by the experiments reported here, and the position is stated plainly rather than inferred. No Visual SLAM system was executed on either dataset. Chapter 4.1 records the decision to isolate low-

level operators instead. What follows is an argument from mechanism, not a demonstration, and is labelled as such.

The argument has three steps. First, feature-based SLAM depends upon correspondences between keypoints detected in successive frames, and its pose estimate is obtained by minimising reprojection error over those correspondences. Second, robust estimation procedures such as RANSAC [73] reject outlying correspondences on the premise that inliers remain in the majority; when spurious detections come to dominate, that premise fails and the estimator converges upon a solution consistent with its inputs but not with the scene. Third, the experiments of Chapter 4 establish the radiometric conditions under which spurious detections come to dominate the output of a gradient-based detector. Taken together, these indicate that the operating boundary measured in Chapter 4 is a necessary condition for SLAM operation: a system cannot track reliably on imagery in which its detector returns predominantly false positives.

Two gaps prevent this from constituting a demonstration, and both are stated rather than minimised. The operator characterised in Chapter 4 is the Harris detector, whereas ORB-SLAM employs FAST, and as set out in Chapter 5.2.1 the two apply different decision rules with no guarantee that their boundaries coincide. Further, a necessary condition is not a sufficient one, that a detector function does not establish that a complete pipeline, with its keyframe selection, loop closure and bundle adjustment, will track successfully. Establishing the link would require executing ORB-SLAM on matched real and synthetic sequences and comparing the resulting trajectories against the Vicon ground truth already available from Chapter 3. That experiment is well defined and the apparatus for it exists; it was not performed, and it constitutes the principal item of future work.

**A programme of further work.** The extensions identified above differ considerably in cost, and a realistic ordering follows from that difference rather than from their intrinsic interest. Table 19 sets out a programme in which each step is placed according to what it requires and what it depends upon. The distinction that governs the ordering is whether a step requires further physical capture: the first four do not, since the calibrated image pairs of Chapter 4.5, the reconstruction datasets of Chapter 3.5 and the Vicon trajectories of Chapter 3 all already exist and are sufficient for those purposes.

| # | Step | What it requires | New capture? | Depends on | Purpose |
|---|---|---|---|---|---|
| 1 | Characterise FAST, Shi–Tomasi and a learned detector | Re-running the analysis of Chapter 4.4 on existing image pairs | No | — | Establishes whether the Safe Operating Area is a property of gradient-based detection generally or of Harris specifically, and bridges to the detector ORB-SLAM employs |
| 2 | Establish the reconstruction noise floor | Reconstructing one physical dataset twice and comparing the results | No | — | Separates simulation error from the non-determinism recorded in Chapter 3.5.6, permitting the residual deviation of Chapter 3.4.1 to be attributed |
| 3 | Validate at system level | Executing ORB-SLAM on matched sequences against the Vicon ground truth of Chapter 3 | No | 1 | Converts the argument from mechanism into a demonstration |
| 4 | Model optics within the renderer | Re-implementation in an engine supporting explicit lens models | No | — | Removes the uniform-kernel limitation of Chapter 4.2.1 and supplies the field-dependent blur recorded as absent in Table 18 |
| 5 | Compare surface textures systematically | Capture of one environment under several textures | Yes, modest | — | Tests whether the finding of Chapter 3.5.6 generalises beyond the single texture examined |

| # | Step | What it requires | New capture? | Depends on | Purpose |
|---|---|---|---|---|---|
| 6 | Extend calibration to time-varying parameters | Method development, then capture from a moving platform | Yes | 3, 4 | Addresses the four high-priority phenomena of Table 18, none of which is fixed at the moment of capture |
| 7 | Validate under outdoor and variable illumination | Capture across changing daylight conditions | Yes | 6 | Tests the most demanding of the scalability dimensions identified above |

*Table 19 - A programme of further work, ordered by cost and dependency*

The ordering places the operator comparison first because it is the least expensive step available and because its outcome determines how the system-level validation should be designed: if the Safe Operating Area proves to be a property of gradient-based detection generally, the boundary measured here may be applied to ORB-SLAM directly; if it proves specific to the Harris formulation, the characterisation must be repeated for FAST before the system-level comparison can be interpreted. Steps 6 and 7 are placed last not because they are least important but because each depends upon the completion of earlier steps, and because each requires capture from a moving platform under conditions the present apparatus was not built to provide.

**Progression towards the longer-term objective.** The programme above is directed at a single end. The argument of Chapter 1 was that computer vision lacks what other engineering disciplines possess as a matter of course: a specification stating the conditions under which a component will function. A transistor is supplied with characteristic curves; a vision operator is not, and a practitioner selecting one has no equivalent basis for prediction. The Safe Operating Area of Chapter 4.4.1 is a specification of that kind for one operator under two

variables, obtained at the cost of a single experimental programme. Should the same procedure prove transferable across operators, across sensors and eventually across the systems built from them, the result would be a body of characterisations from which the behaviour of a vision pipeline could be predicted before it is assembled rather than measured after. That is the objective towards which this work is directed. The present thesis establishes that such a specification can be produced and that it transfers from simulation to physical hardware for the case examined. What remains is the demonstration that it does so generally, and the steps set out above are the order in which that demonstration might reasonably be attempted.

**Scope of the validation.** The scope of the validation reported here should be stated plainly. The correspondence between real and synthetic operator performance was established across five camera configurations, one geometric target and one physical room, under controlled indoor illumination. The operators examined were a corner detector, characterised across the full stress space, and ORB feature matching, examined only under geometric transformation. These results support the conclusion that the calibration methodology produces predictive correspondence for the operators and conditions tested. They do not, by themselves, establish that synthetic imagery may substitute for real imagery generally, and no such general claim is made. Broader validation, across additional sensor types, outdoor and variable illumination, a wider range of operators including learned detectors, and ultimately a complete SLAM pipeline, would be required before the substitution could be regarded as established beyond the conditions examined here. The contribution of this work is a framework and a demonstration of its viability, not a general licence to dispense with physical data.

**Closing remarks.** Subject to those limitations, this thesis demonstrates that the performance boundary of a vision operator on real imagery can be predicted from a synthetic counterpart, provided the synthetic imagery is calibrated against measured optical properties

of the physical sensor rather than matched by appearance alone. The shift this represents, from adjusting a system until it works to characterising the conditions under which it will, is the shift that distinguishes engineering practice from empirical practice, and the framework developed here is offered as one route towards it for computer vision.

# References


[1] N. Kanwal, “Low-level image features and navigation systems for visually impaired people,” BMVA, 2013.

[2] J. Tian, “Quantitative performance evaluation of autonomous visual navigation,” PhD Thesis, 2017.

[3] O. Brunneg˚ard and D. Wikestad, “Visual slam using sparse maps based on feature points,” p. 67, 2017.

[4] D. Galvez-Lopez and J. D. Tardos, “Real-time loop detection with bags of binary words,” pp. 51 –58, sept. 2011.

[5] N. A. Thacker, A. F. Clark, J. L. Barron, J. R. Beveridge, P. Courtney, W. R. Crum, V. Ramesh, and C. Clark, “Performance characterization in computer vision: A guide to best practices,” Computer Vision and Image Understanding, vol. 109, pp. 305–334, Mar. 2008.

[6] D. Galvez-Lopez and J. D. Tardos, “Bags of binary words for fast place recognition in image sequences,” IEEE Transactions on Robotics, vol. 28, pp. 1188–1197, October 2012.

[7] P. Cignoni, M. Corsini, and G. Ranzuglia, “Meshlab and cloudcompare: Open- source 3d mesh processing systems,” ERCIM News, pp. 45–46, April 2008.

[8] X. Zou, H.Z., Lu, J.: Virtual manipulator-based binocular stereo vision positioning system and errors modelling. Machine Vision and Applications 23(2012) 43-63, 2012.

[9] Fang, Y., Liu, X., Zhang, X.: Adaptive active visual servoing of non- holonomic mobile robots. IEEE Transactions on Industrial Electronics 59 (2012) 486-497,October 2012

[10] Xingshuai D, Xinghui D, J.D.: Monocular visual-imu odometry: A comparative evaluation of the detector-descriptor based methods. In: CVR- SAID 86. (2016)

[11] D, X., Chantler, M.J.: Texture similarity estimation using contours. In: BMVC14. (2014) Kuci M.: Simulation of camera features. In: the 16th Central Europen Seminar on Computer. (2012) 117-123

[12] A. Hinkenjann, T.R., Millberg, J.: Real-time simulation of camera errors and their effect on some basic robotic vision algorithms. In: CRV13. (2013) 218-225

[13] K. Asanuma, K. Umeda, R.U., Arai, T.: Development of a simulator of environment and measurement for autonomous mobile robots considering camera characteristics. In: RoboCup 2003: Robot Soccer World Cup VII. Springer (2004) 446-457

[14] K. Okada, Y.K., Kanehiro, F.: Rapid development system for humanoid vision based behaviours with real-virtual common interface. In: IROS02. (2002)

[15] I. Ulusoy, U.H., Leblebicioglu, K.: 3d cognitive map construction by active stereo vision in a virtual world. In: ISCIS04. Springer (2004) 400- 409

[16] Klaser, R., Wolf, D.: Simulation of an autonomous vehicle with a vision-based navigation system in unstructured terrains using octomap. In: SBESC13. (2013) 177-178

[17] Thacker, N.A., Riocreux, P., Yates, R.: Assessing the completeness properties of pairwise geometric histograms. Image and Vision Computing 13 (1995) 423- 429

[18] Ashbrook, A., Thacker, N.A., Rockett, P., Brown, C.: Robust recognition of scaled shapes using pairwise geometric histograms. In: BMVC95. Volume 95. (1995) 503-512

[19] FJ. Aherne, N.T., Rockett, P.: The bhattacharyya metric as an absolute similarity measure for frequency coded data. Kybernetika 34 (1998) 363- 368

[20] Kong, W., Zhang, D., Wang, X., Xian, Z., Zhang, J.: Autonomous landing of an uav with a ground-based actuated infrared stereo vision system. In: IROS13, IEEE (2013) 2963-2970

[21] Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two- frame stereo correspondence algorithms. International journal of computer vision 47 (2002) 7-42

[22] Yoon, K.J., Kweon, I.S.: Adaptive support-weight approach for correspondence search. IEEE Transactions on Pattern Analysis and Machine Intelligence 28 (2006) 650-656

[23] Shi, C., Wang, G., Yin, X., Pei, X., He, B., Lin, X.: High-accuracy stereo matching based on adaptive ground control points. IEEE Transactions on Image Processing 24 (2015) 1412-1423

[24] Wang, L., Yang, R.: Global stereo matching leveraged by sparse ground control points. In: CVPR11, IEEE (2011) 3033-3040

[25] Bleyer, M., Rhemann, C., Rother, C.: Patchmatch stereo-stereo matching with slanted support windows. In: BMVC11. Volume 11. (2011) 1-11

[26] Scharstein, D., Hirschmuller, H., Kitajima, Y., Krathwohl, G., Ne, N., Wang, X., Westling, P.: High-resolution stereo datasets with subpixel- accurate ground truth. In: German Conference on Pattern Recognition, Springer (2014) 31-42

[27] Forster, C., Pizzoli, M., Scaramuzza, D.: Air-ground localization and map augmentation using monocular dense reconstruction. In: IROS;13, IEEE (2013) 3971-3978

[28] Royer, E., Bom, J., Dhome, M., Thuilot, B., Lhuillier, M., Marmoiton, F.: Outdoor autonomous navigation using monocular vision. In: IROS05, IEEE (2005) 1253-1258

[29] Waymo. https://waymo.com/

[30] Michele Bertoncello Wee and Dominik. Ten ways autonomous driving could redefine the automotive world

[31] Geoffrey R. Taylor, Andrew J. Chosak, and Paul C. Brewer. Ovvv: Using virtual worlds to design and evaluate surveillance systems. pages 1-8, 07 2007.

[31] Aloimonos, Y.: Visual navigation: from biological systems to unmanned ground vehicles. Psychology Press (2013)

[32] Leonard, J.J., Durrant-Whyte, H.F.: Mobile robot localization by tracking geometric beacons. IEEE transactions on robotics and automation 7 (1991) 376–382

[33] Koenig, S., Mudgal, A., Tovey, C.: A near-tight approximation lower bound and algorithm for the kidnapped robot problem. In: ACM-SIAM'06, Society for Industrial and Applied Mathematics (2006) 133–142

[34] Doh, N.L., Lee, K., Chung, W.K., Cho, H.: Simultaneous localisation and mapping algorithm for topological maps with dynamics. IET control theory & applications 3 (2009) 1249–1260

[35] Churchill, W., Newman, P.: Practice makes perfect? managing and leveraging visual experiences for lifelong navigation. In: ICRA'12. (2012) 4525–4532

[36] DeSouza, G.N., Kak, A.C.: Vision for mobile robot navigation: A survey. IEEE transactions on pattern analysis and machine intelligence 24 (2002) 237– 267

[37] Herisse, B., Hamel, T., Mahony, R., Russotto, F.X.: Landing a VTOL unmanned aerial vehicle on a moving platform using optical flow. IEEE Transactions on Robotics 28 (2012) 77–89

[38] Talukder, A., Matthies, L.: Real-time detection of moving objects from moving vehicles using dense stereo and optical flow. In: Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on. Volume 4., IEEE (2004) 3718–3725

[39] Camus, T., Coombs, D., Herman, M., Hong, T.H.: Real-time single workstation obstacle avoidance using only wide-field flow divergence. In:ICPR'96. Volume 3., IEEE (1996) 323–330

[40] Haddad, H., Khatib, M., Lacroix, S., Chatila, R.: Reactive navigation in outdoor environments using potential fields. In: Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on. Volume 2., IEEE (1998)

[41] Harrell, R., Slaughter, D., Adsit, P.D.: A fruit-tracking system for robotic harvesting. Machine Vision and Applications 2 (1989) 69–80

[42] Tian, J., Thacker, N., Stancu, A.: Quantitative performance optimisation for corner and edge based robotic vision systems: A Montecarlo simulation. In:ISVC'16, Springer (2016) 544–554

[43] Churchill, W., Newman, P.: Practice makes perfect? managing and leveraging visual experiences for lifelong navigation. In: ICRA'12. (2012) 4525–4532

[44] Fuentes-Pacheco, J., Ruiz-Ascencio, J., Rendon-Mancha, J.M.: Visual simultaneous localization and mapping: a survey. Artificial Intelligence Review 43 (2015) 55–81

[45] Filliat, D., Meyer, J.A.: Map-based navigation in mobile robots:: I. a review of localization strategies. Cognitive Systems Research 4 (2003) 243–282

[46] Guzel, M.S.: Autonomous vehicle navigation using computer vision strategies: a survey. Advances in Mechanical Engineering 5 (2013) 234747

[47] Canny, J.: A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence (1986) 679–698

[48] Braillon, C., Usher, K., Pradalier, C., Crowley, J.L., Laugier, C.: Fusion of stereo and optical flow using occupancy grids. In: ITSC'06, IEEE (2006) 1240–1245

[49] Matsumoto, Y., Inaba, M., Inoue, H.: Visual navigation using view-sequenced route representation. In: ICRA'96. Volume 1., IEEE (1996) 83–88

[50] Zhou, C., Wei, Y., Tan, T.: Mobile robot self-localization based on global visual appearance features. In: ICRA'03. Volume 1., IEEE (2003) 1271–1276

[51] Kim, D., Nevatia, R.: Recognition and localization of generic objects for indoor navigation using functionality. Image and Vision Computing 16 (1998) 729–743

[52] Xu, D., Han, L., Tan, M., Li, Y.F.: Ceiling-based visual positioning for an indoor mobile robot with monocular vision. IEEE Transactions on Industrial Electronics 56 (2009) 1617–1628

[53] Li, H., Yang, S.X.: A behaviour-based mobile robot with a visual landmark recognition system. IEEE/ASME transactions on mechatronics 8 (2003) 390–400

[54] Sjo, K., L ¨ opez, D.G., Paul, C., Jensfelt, P., Kragic, D.: Object search ´ and localization for an indoor mobile robot. Journal of Computing and Information Technology 17 (2009) 67–80

[55] Tasaki, T., Tokura, S., Sonoura, T., Ozaki, F., Matsuhira, N.: Mobile robot self-localization based on tracked scale and rotation invariant feature points by using an omnidirectional camera. In: IROS'10, IEEE (2010) 5202–5207

[56] Gutmann, J.S., Fukuchi, M., Fujita, M.: 3d perception and environment map generation for humanoid robot navigation. The International Journal of Robotics Research 27 (2008) 1117–1134

[57] Mouragnon, E., Lhuillier, M., Dhome, M., Dekeyser, F., Sayd, P.: Monocular vision-based slam for mobile robots. In: ICPR'06. Volume 3., IEEE (2006) 1027–1031

[58] Manessis, A., Hilton, A., Palmer, P., McLauchlan, P., Shen, X.: Reconstruction of scene models from sparse 3d structure. In: CVPR'00. Volume 2., IEEE (2000) 666–671

[59] Torr, P., Fitzgibbon, A.W., Zisserman, A.: Maintaining multiple motion model hypotheses over many views to recover matching and structure. In: ICCV'98,IEEE (1998) 485–491

[60] Se, S., Lowe, D.G., Little, J.J.: Vision-based global localization and mapping for mobile robots. IEEE Transactions on robotics 21 (2005) 364–375

[61] Sim, R., Little, J.J.: Autonomous vision-based exploration and mapping using hybrid maps and particle filters. In: IROS'06, IEEE (2006)2082–2089

[62] Choi, Y.H., Lee, T.K., Oh, S.Y.: A line feature based slam with low grade range sensors using geometric constraints and active exploration for mobile robot. Autonomous Robots 24 (2008) 13–27

[63] Zhao, L., Huang, S., Yan, L., Dissanayake, G.: A new feature parametrization for monocular slam using line features. Robotica 33 (2015) 513–536

[64] Zhang, G., Lee, J.H., Lim, J., Suh, I.H.: Building a 3-d line-based map using stereo slam. IEEE Transactions on Robotics 31 (2015) 1364–1377

[65] Gracias, N., Santos-Victor, J.: Underwater video mosaics as visual navigation maps. Computer Vision and Image Understanding 79 (2000) 66–91

[66] Scaramuzza, D., Siegwart, R.: Appearance-guided monocular omnidirectional visual odometry for outdoor ground vehicles. IEEE transactions on robotics 24 (2008) 1015–1026

[67] Tomono, M.: 3-d object map building using dense object models with sift based recognition features. In: IROS'06, IEEE (2006) 1885–1890

[68] Paz, L.M., Pinies, P., Tard ´ os, J.D., Neira, J.: Large-scale 6-dof slam with ´ stereo-in-hand. IEEE transactions on robotics 24 (2008) 946–957

[69] Fraundorfer, F., Engels, C., Nister, D.: Topological mapping, localization and ´ navigation using image collections. In: IROS'07, IEEE (2007) 3872–3877

[70] Royer, E., Bom, J., Dhome, M., Thuilot, B., Lhuillier, M., Marmoiton, F.:Outdoor autonomous navigation using monocular vision. In: IROS'05, IEEE (2005) 1253–1258

[71] Kaess, M., Dellaert, F.: Probabilistic structure matching for visual slam with a multi-camera rig. Computer Vision and Image Understanding 114 (2010) 286–296

[72] Eade, E., Drummond, T.: Unified loop closing and recovery for real time monocular slam. In: BMVC'08. Volume 13. (2008) 136

[73] Fischler, M.A., Bolles, R.C.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM 24 (1981) 381–395

[74] Sim, R., Elinas, P., Griffin, M., Shyr, A., Little, J.J.: Design and analysis of a framework for real-time vision-based slam using particle filters. In: CRV'06, IEEE (2006) 21–21

[75] Williams, B., Klein, G., Reid, I.: Real-time slam relocalisation. In: ICCV'07,IEEE (2007) 1–8

[76] Dumont, M., Goorts, P., Maesen, S., Bekaert, P., Lafruit, G.: Real-time local stereo matching using edge sensitive adaptive windows. In: SIGMAP'14. (2014) 117–126

[77] Remazeilles, A., Chaumette, F., Gros, P.: Robot motion control from a visual memory. In: ICRA'04. Volume 5., IEEE (2004) 4695–4700

[78] Argyros, A.A., Bergholm, F.: Combining central and peripheral vision for reactive robot navigation. In: CVPR'99. Volume 2., IEEE (1999) 646–651

[79] Gracias, N., Santos-Victor, J.: Underwater video mosaics as visual navigation maps. Computer Vision and Image Understanding 79 (2000) 66–91

[80] Angeli, A., Doncieux, S., Meyer, J.A., Filliat, D.: Incremental vision-based topological slam. In: IROS'08, IEEE (2008) 1031–1036

[81] Kostavelis, I., Gasteratos, A.: Semantic mapping for mobile robotics tasks: A survey. Robotics and Autonomous Systems 66 (2015) 86–103

[82] Brooks, R.: Visual map making for a mobile robot. In: ICRA'85. Volume 2., IEEE (1985) 824–829

[83] Kuipers, B., Byun, Y.T.: A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. Robotics and autonomous systems 8 (1991) 47–63

[84] Kostavelis, I., Gasteratos, A.: Semantic mapping for mobile robotics tasks: A survey. Robotics and Autonomous Systems 66 (2015) 86–103

[85] Fox, R., Blake, R.R.: Stereoscopic vision in the cat. Nature 233 (1971) 55–56

[86] Hubel, D.H., Wiesel, T.N.: Stereoscopic vision in macaque monkey: cells sensitive to binocular depth in area 18 of the macaque monkey cortex. Nature 225 (1970) 41–42

[87] Yoon, K.J., Kweon, I.S.: Adaptive support-weight approach for correspondence search. IEEE Transactions on Pattern Analysis and Machine Intelligence 28 (2006) 650–656

[88] Hosni, A., Bleyer, M., Gelautz, M., Rhemann, C.: Local stereo matching using geodesic support weights. In: ICIP'09, IEEE (2009) 2093–2096

[89] Min, D., Lu, J., Do, M.N.: A revisit to cost aggregation in stereo matching: How far can we reduce its computational redundancy? In: ICCV'11, IEEE (2011) 1567–1574

[90] Muninder, V., Soumik, U., Krishna, G.: Robust segment-based stereo using cost aggregation. In: BMVC'14. (2014)

[91] Zhang, K., Fang, Y., Min, D., Sun, L., Yang, S., Yan, S., Tian, Q.: Cross-scale cost aggregation for stereo matching. In: CVPR'14. (2014) 1590–1597

[92] Shi, C., Wang, G., Yin, X., Pei, X., He, B., Lin, X.: High-accuracy stereo matching based on adaptive ground control points. IEEE Transactions on Image Processing 24 (2015) 1412–1423

[93] Hirschmuller, H., Scharstein, D.: Evaluation of stereo matching costs on images with radiometric differences. IEEE transactions on pattern analysis and machine intelligence 31 (2009) 1582–1599204

[94] Tippetts, B., Lee, D.J., Lillywhite, K., Archibald, J.: Review of stereo vision algorithms and their suitability for resource-limited systems. Journal of RealTime Image Processing 11 (2016) 5–25

[95] Rao, C.R.: Information and the accuracy attainable in the estimation of statistical parameters. In: Breakthroughs in statistics. Springer (1992) 235–247

[96] Hosni, A., Bleyer, M., Gelautz, M.: Secrets of adaptive support weight techniques for local stereo matching. Computer Vision and Image Understanding 117 (2013) 620–632

[97] Ashbrook, A.P.: Pairwise geometric histograms for object recognition: developments and analysis. PhD thesis, University of Sheffield (1999)

[98] Kim, H., Leutenegger, S., Davison, A.J.: Real-time 3d reconstruction and 6- dof tracking with an event camera. In: ECCV'16, Springer (2016) 349–364

[99] Rojas, R., Forster, A.G.: Holonomic control of a robot with an omnidirectional drive. KI-Kunstliche Intelligenz ¨ 20 (2006) 12–17

[100] A. Evans, N.T., Mayhew, J.: The use of geometric histograms for model- based object recognition. In: BMVC'93. (1993) 429–438

[101] Kisilev, P., Freedman, D.: Parameter tuning from pairwise preferences. In: BMVC'10, Citeseer (2010) 1–11

[102] Angeli, A., Doncieux, S., Meyer, J.A., Filliat, D.: Incremental vision-based topological slam. In: IROS'08, Ieee (2008) 1031–1036

[103] Ciarfuglia, T.A., Costante, G., Valigi, P., Ricci, E.: A discriminative approach for appearance based loop closing. In: IROS'12, IEEE (2012) 3837–3843

[104] Mur-Artal, R., Tardos, J.D.: Fast relocalisation and loop closing in keyframe-based slam. In: ICRA'14, IEEE (2014) 846–853

[105] Korrapati, H., Uzer, F., Mezouar, Y.: Hierarchical visual mapping with omnidirectional images. In: IROS'13, IEEE (2013) 3684–3690

[106] Korrapati, H., Mezouar, Y.: Vision-based sparse topological mapping. Robotics and Autonomous Systems 62 (2014) 1259–1270

[107] Wang, J., Yagi, Y.: Robust location recognition based on efficient feature integration. In: ROBIO'12, IEEE (2012) 97–101

[108] Lin, H.Y., Lin, Y.H., Yao, J.W.: Scene change detection and topological map construction using omnidirectional image sequences. In: ICMVA'13. (2013) 57–60

[109] Murillo, A., Campos, P., Kosecka, J., Guerrero, J.: Gist vocabularies in omnidirectional images for appearance based mapping and localization. OMNIVIS'10, held with RSS'10 3 (2010)

[110] Sunderhauf, N., Protzel, P.: Brief-gist-closing the loop by imple means. In: IROS'11, IEEE (2011) 1234–1241

[111] Badino, H., Huber, D., Kanade, T.: Real-time topometric localization. In: ICRA'12, IEEE (2012) 1635–1642

[112] Kosecka, J., Li, F.: Vision based topological markov localization. In: ICRA'04.Volume 2., IEEE (2004) 1481–1486

[113] Zhang, H.: Borf: Loop-closure detection with scale invariant visual features. In: ICRA'11. (2011) 3125–3130

[114] Sabatta, D.: Vision-based topological map building and localisation using persistent features. In: Robotics and Mechatronics Symposium. (2008) 1–6

[115] Andreasson, H., Duckett, T.: Topological localization for mobile robots using omni-directional vision and local features. In: IFAC IAV'04. (2004)

[116] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM3: An accurate open-source library for visual, visual–inertial, and multimap SLAM," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, Dec. 2021.

[117] A. Macario Barros, M. Michel, Y. Moline, G. Corre, and F. Carrel, "A comprehensive survey of visual SLAM algorithms," *Robotics*, vol. 11, no. 1, art. 24, 2022.

[118] Z. Teed and J. Deng, "DROID-SLAM: Deep visual SLAM for monocular, stereo, and RGB-D cameras," in *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*, pp. 16558–16569, 2021.

[119] H. Matsuki, R. Murai, P. H. J. Kelly, and A. J. Davison, "Gaussian splatting SLAM," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18039–18048, 2024.

[120] N. Keetha, J. Karhade, K. M. Jatavallabhula, G. Yang, S. Scherer, D. Ramanan, and J. Luiten, "SplaTAM: Splat, track & map 3D Gaussians for dense RGB-D SLAM," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 21357–21366, 2024.

[121] W. Wang, D. Zhu, X. Wang, Y. Hu, Y. Qiu, C. Wang, Y. Hu, A. Kapoor, and S. Scherer, "TartanAir: A dataset to push the limits of visual SLAM," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4909–4916, Oct. 2020.

[122] K. Greff, F. Belletti, L. Beyer, C. Doersch, Y. Du, D. Duckworth, D. J. Fleet, *et al.*, "Kubric: A scalable dataset generator," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3749–3761, 2022.

[123] A. Raistrick, L. Lipson, Z. Ma, L. Mei, M. Wang, Y. Zuo, K. Kayan, H. Wen, B. Han, Y. Wang, A. Newell, H. Law, A. Goyal, K. Yang, and J. Deng, "Infinite photorealistic worlds using procedural generation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 12630–12641, 2023.

[124] H. Schieber, K. C. Demir, C. Kleinbeck, S. H. Yang, and D. Roth, "Indoor synthetic data generation: A systematic review," *Computer Vision and Image Understanding*, vol. 240, art. 103907, 2024.

[125] A. Szot, A. Clegg, E. Undersander, E. Wijmans, Y. Zhao, J. Turner, N. Maestre, *et al.*, "Habitat 2.0: Training home assistants to rearrange their habitat," in *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*, pp. 251–266, 2021.

[126] M. Mittal, C. Yu, Q. Yu, J. Liu, N. Rudin, D. Hoeller, J. L. Yuan, R. Singh, Y. Guo, H. Mazhar, A. Mandlekar, B. Babich, G. State, M. Hutter, and A. Garg, "Orbit: A unified simulation framework for interactive robot learning environments," *IEEE Robotics and Automation Letters*, vol. 8, no. 6, pp. 3740–3747, 2023.

[127] S. Mihai, M. Yaqoob, D. V. Hung, W. Davis, P. Towakel, M. Raza, M. Karamanoglu, B. Barn, D. Shetve, R. V. Prasad, H. Venkataraman, R. Trestian, and H. X. Nguyen, "Digital twins: A survey on enabling technologies, challenges, trends and future prospects," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2255–2291, 2022.

[128] D. DeTone, T. Malisiewicz, and A. Rabinovich, "SuperPoint: Self-supervised interest point detection and description," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 224–236, 2018.

[129] P.-E. Sarlin, D. DeTone, T. Malisiewicz, and A. Rabinovich, "SuperGlue: Learning feature matching with graph neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4938–4947, 2020.

[130] J. Sun, Z. Shen, Y. Wang, H. Bao, and X. Zhou, "LoFTR: Detector-free local feature matching with transformers," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8922–8931, 2021.

[131] Y. Jin, D. Mishkin, A. Mishchuk, J. Matas, P. Fua, K. M. Yi, and E. Trulls, "Image matching across wide baselines: From paper to practice," *International Journal of Computer Vision*, vol. 129, no. 2, pp. 517–547, 2021.

[132] A. Dosovitskiy, G. Ros, F. Codevilla, A. López, and V. Koltun, "CARLA: An open urban driving simulator," in *Proc. 1st Annual Conference on Robot Learning (CoRL)*, PMLR vol. 78, pp. 1–16, 2017.

[133] S. Borkman, A. Crespi, S. Dhakad, S. Ganguly, J. Hogins, Y.-C. Jhang, M. Kamalzadeh, B. Li, S. Leal, P. Parisi, C. Romero, W. Smith, A. Thaman, S. Warren, and N. Yadav, "Unity Perception: Generate synthetic data for computer vision," *arXiv preprint* arXiv:2107.04259, 2021.

[134] NVIDIA Corporation, "Build custom synthetic data generation pipelines with Omniverse Replicator," NVIDIA Technical Blog. [Online]. Available: https://developer.nvidia.com/blog/build-custom-synthetic-data-generation-pipelines-with-omniverse-replicator/ [Accessed: Aug. 2026].

[135] NVIDIA Corporation, "Validating NVIDIA DRIVE Sim camera models," NVIDIA Technical Blog. [Online]. Available: https://developer.nvidia.com/blog/validating-drive-sim-camera-models [Accessed: Aug. 2026].

[136] Ansys Inc., "Ansys AVxcelerate Sensors: AV sensor simulation software," product documentation. [Online]. Available: https://www.ansys.com/products/av-simulation/ansys-avxcelerate-sensors [Accessed: Aug. 2026].

[137] T. Goossens, Z. Lyu, J. Ko, G. Wan, J. Farrell, and B. Wandell, "Ray-transfer functions for camera simulation of 3D scenes with hidden lens design," *Optics Express*, vol. 30, p. 24031, 2022.

[138] Agisoft LLC, "Agisoft Metashape," product documentation, 2019. [Online]. Available: https://www.agisoft.com/ [Accessed: Aug. 2026].

[139] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3354–3361, 2012.

[140] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of RGB-D SLAM systems," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 573–580, 2012.

# Appendices

## Add noise to images

```
#!/usr/bin/env python3
"""A program to help Stefan add noise to images.

The difficulty lies in OpenCV working principally with an unsigned byte
representation for pixels, so it is necessary to convert images to
floating-point before adding noise and then back to unsigned bytes for
display.  Note that displayed noisy images will most likely appear weird
because no attempt has been made to clip pixels into [0:255] after noise has
been added.

Before using only OpenCV, numpy and scipy routines to do the work, the same
procedure is done using routines from my EVE library as it represents pixels
as floating-point internally.  Overall, I think that's a much better approach
although it makes programs that use EVE run somewhat more slowly than when
using OpenCV.  The two can interact providing that images are converted to
what OpenCV expects (i.e., unsigned bytes) at the appropriate place.
"""

# Boilerplate.
import sys, math
import cv2, numpy, scipy.stats
import eve

# We need a filename and a noise sd on the command line.
if len (sys.argv) != 3:
    print ("Usage: %s <image> <sd>", file=sys.stderr)
    exit (1)

# Convert the standard deviation argument to a float.
sd = float (sys.argv[2])

#-------------------------------------------------------------------------------
# Work through the process using EVE routines.
#-------------------------------------------------------------------------------
# Read in the image and clone it.  # Add noise to the two copies of the image.
im1 = eve.image (sys.argv[1])
im2 = im1.copy ()
eve.add_gaussian_noise (im1, sd=sd)
eve.add_gaussian_noise (im2, sd=sd)

# Compute the correlation and SNR.
r = eve.correlation_coefficient (im1, im2)
print ("r:", r)
snr = eve.snr (im1, im2)
print ("SNR:", snr)

# OpenCV expects floating-point images to lie in [0:1], so make a temporary
# copy of one image, then scale and display it.  Note that the displayed image
# will be especially funky here because EVE uses RGB and OpenCV BGR for the
# colour channels.  Sigh.
tim = im1 / 255
cv2.imshow ("Noisy image from EVE", tim)
cv2.waitKey (0)

#-------------------------------------------------------------------------------
# Now let's go through the same process using only OpenCV and numpy calls.
#-------------------------------------------------------------------------------

# Read in the image and convert to float.  Duplicate it and add noise.
im1 = cv2.imread (sys.argv[1]).astype ("float32")
im2 = im1.copy ()
im1 += numpy.random.normal (0.0, sd, im1.shape)
im2 += numpy.random.normal (0.0, sd, im2.shape)

# Compute the correlation coefficient and SNR.
```

```
66. r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
67. print ("r:", r)
68. snr = math.sqrt (r / (1 - r))
69. print ("SNR:", snr)
70.
71. # Convert the image to byte and display the result.  Although the way we do
72. # this is different from above, the overall processes are the same.
73. tim = im1.astype ("uint8")
74. cv2.imshow ("Noisy image from OpenCV", tim)
75. cv2.waitKey (0)
76.
77. # "That's all, folks".
78.
```

## Functions to work with

```
import cv2
import scipy
import math
import scipy.stats

def write_text_on_image(my_file, corners_detected):
    import numpy as np
    import cv2

    noisy_image_display_and_count=my_file[0:-4]+"_COUNT.jpg"


    # Create a black image
    #img = np.zeros((512,512,3), np.uint8)

    img = cv2.imread(my_file)
    x_coord=img.shape[0]
    y_coord=img.shape[1]
    bits_of_color=img.shape[2]



    # Write some Text

    font                   = cv2.FONT_HERSHEY_SIMPLEX
    bottomLeftCornerOfText = (10,y_coord-200)
    fontScale              = 1
    fontColor              = (255,255,255)
    thickness              = 3
    lineType               = 2

    percentage=float(Standard_Dev)/255*100
    percentage=float("{:.2f}".format(percentage))
    cv2.putText(img,'Corner count: '+str(corners_detected)+", "+str(percentage)+"% noise",
                bottomLeftCornerOfText,
                font,
                fontScale,
                fontColor,
                thickness,
                lineType)

    #Display the image
    cv2.imshow("Corner Display and Count",img)

    #Save image
    cv2.imwrite(noisy_image_display_and_count,img)

    #cv2.waitKey(0)




def add_gaussian_noise (im, mean=0.0, sd=1.0, seed=None):
    """
    Add Gaussian-distributed noise to each pixel of image `im`.

    Arguments:
      im  the image to which noise will be added (modified)
    mean  the mean of the Gaussian-distributed noise (default: 0.0)
      sd  the standard deviation of the noise (default: 1.0)
    seed  if supplied, the value used to seed the random number generator
    """
    image_string=im
    window_name="Original Image"
```

```
    im = cv2.imread(im)

    "convert to float for manipulation as OpenCV only works with UNINT8"
    window_name2="Image with Noise"
    im2 = im.astype("float64")

    "im = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)"

    if not seed is None: numpy.random.seed (seed)
    " add the noise "
    im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))

    " produce a red image by writing the BG color data to 0  (Image is read as BGR)"
    #im2[:,:,(0,1)]=0

    " change back to UINT8 for display with"
    im2 = im2.astype("uint8")

    " write noisy image to file"
    cv2.imwrite('c:\okllak\ONE_NOISY.jpg',im2)

    cv2.imshow(window_name2,im2)
    cv2.imshow(window_name,im)
    #cv2.waitKey (0)

    filename_string=image_string[0:-4]+"_NOISY.jpg"


    return filename_string


def corn_det_visual_and_numeric(my_image):
    filename = my_image
    img = cv2.imread(filename)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    gray = numpy.float32(gray)
    dst = cv2.cornerHarris(gray,2,3,0.04)
    "number of corner edtections"
    num_corners = numpy.sum(dst > 0.01 * dst.max())


    #result is dilated for marking the corners, not important
    dst = cv2.dilate(dst,None)

    # Threshold for an optimal value, it may vary depending on the image.
    img[dst>0.01*dst.max()]=[0,0,255]

    noisy_image_display=my_image[0:-4]+"_DISPLAY.jpg"

    " write noisy cornered image to file"
    cv2.imwrite("c:\\okllak\\"+noisy_image_display,img)


    cv2.imshow('Corners Display',img)
    #if cv2.waitKey(0) & 0xff == 27:
        #cv2.destroyAllWindows()


    #print(num_corners)
    return num_corners, noisy_image_display



def noise_vs_corners(my_image, mean=0.0, sd=1.0, seed=None):

    # Function to compute the noise added in the form of STD
    # and then to detect the corners that are detected with this noise

    #ADD NOISE TO IMAGE
```

```
136.
137.     im = cv2.imread(my_image)
138.     "convert to float for manipulation as OpenCV only works with UNINT8"
139.     im2 = im.astype("float64")
140.
141.     if not seed is None: numpy.random.seed (seed)
142.     " add the noise "
143.     im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
144.     " change back to UINT8 for display with"
145.     im2 = im2.astype("uint8")
146.
147.     #CALCULATE NUMBER OF CORNERS
148.
149.     gray = cv2.cvtColor(im2,cv2.COLOR_BGR2GRAY)
150.
151.     gray = numpy.float32(gray)
152.     dst = cv2.cornerHarris(gray,2,3,0.04)
153.     "number of corner edtections"
154.     num_corners = numpy.sum(dst > 0.01 * dst.max())
155.
156.     return num_corners
157.
158.
159.
160. def compute_snr(my_image, mean=0.0, sd=1.0, seed=None):
161.     # Read in the image and convert to float.  Duplicate it and add noise.
162.     im1 = cv2.imread (my_image)
163.     im1 = im1.astype("float32")
164.     im2 = im1.copy ()
165.
166.     im1 = numpy.add(im2, numpy.random.normal (mean, sd, im1.shape))
167.     im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
168.
169.     # Compute the correlation coefficient and SNR.
170.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
171.     #print ("r:", r)
172.     snr = math.sqrt (r / (1 - r))
173.     #print ("SNR:", snr)
174.
175.     return snr
176.
177.
178.
179. # here I add noise in the amound of 9 STDs. This would be 9/255 = 3.5% from the grey
levels.
180. Standard_Dev=3
181.
182. new_file=add_gaussian_noise("ONE.JPG",0,Standard_Dev)
183. no_of_corners, corner_display_image = corn_det_visual_and_numeric(new_file)
184.
185. write_text_on_image(corner_display_image, no_of_corners)
186.
187.
188.
189.
190.
191. # the following code is for diplay a graph
192.
193.
194. SNR=[]
195. STD=[]
196. NO_OF_CORNERS=[]
197. temp=[]
198.
199. # in percentage
200. for x in numpy.arange (0,0.3,0.01):
201.
202.     # iterations as the same STD gives off a different number of corners
203.     # 255 here is the max percentage of 100% from the levels of grey. So the STD
204.     # applied is just a fraction of that.
```

```
    for y in range (1,3):
        data = noise_vs_corners("EIN.JPG", mean=0, sd=x*255)
        temp.append(data)

    # calculate mean as the same STD gives different numbers of corners
    mean=sum(temp) / len (temp)
    temp=[]

    value=compute_snr("EIN.JPG", mean=0, sd=x*255)
    SNR.append(value)

    STD.append(x)
    NO_OF_CORNERS.append(mean)


import matplotlib.pyplot as plt


plt.plot(SNR,NO_OF_CORNERS)

plt.show()


if cv2.waitKey(0) & 0xff == 27:
        cv2.destroyAllWindows()

```

## Adding Blur To Images (Version 1)

```
# The program checks if the corner detector yields the same results on virtual images (with same
# noise and blur added to match the real image's noise and blur), as it does on real image.
# I have found that it doesn't, and that the results change with the resolution of the image.

import cv2
import scipy.stats
import numpy
import math


# This functions adds noise in the required level so as to yield syntethic images
# that have the same SNR as their real counterpart.
def add_noise_get_noisy_image_get_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = my_image
    im1 = im1.astype("float32")
    im2 = im1.copy()

    sd=im1.std(ddof=0)
    percentage_of_noise_added=sd*factor/100.0

    im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added, im1.shape))
    im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added, im2.shape))

    # After noise application, some pixel values can become less than 0 or larger
    # than 255. We do not allow any negative nubmers (corresponding to the level of gray)
    # in the image, otherwise we will not be able to display properly. So any negative
    # numbers are made positive by thye instruction below.
    noisy_for_display = numpy.absolute(im1)

    # If the numbers are larger than 255, then when converting to uint8, the numvbers
    # will be (that number larger than 255)-255, and this will yield numbers close to 0,
    # when they are actually 255 pixels appart. The idea is to limit numbers
    # larger than 255 to 255. This is achieved by the nested "for" below
    for i in range(0,noisy_for_display.shape[0],1):
        for j in range(0,noisy_for_display.shape[1],1):
            if noisy_for_display[i][j]>255:
                temp=noisy_for_display[i][j]-255  #eg. 256.345-255= 1.34500
                noisy_for_display[i][j]=255-temp

    # At this point we don't have negative numbers and we also don't have
    # numbers greater that 255. This normalization is only for display, as without it,
    # the displayed image would look wrong. This normalised image will also be passed
    # to the corner detector. But the computation for the SNR is done on the virtual
    # images before normalisation.

    noisy_for_display = numpy.uint8(noisy_for_display)

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    snr = math.sqrt (r / (1 - r))

    # At this point I return the noisy image to be displayed, and the SNR value
    # so as to show that I have applied enough noise to the virtual image to match
    # the SNR of the real image.
    return snr, noisy_for_display


# This is just a function for quikcly computing the SNR of 2 images.
def compute_snr_real(my_image1, my_image2):

    im1 = my_image1.astype("float32")
    im2 = my_image2.astype("float32")
```

```
64.
65.     # Compute the correlation coefficient and SNR.
66.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
67.     snr = math.sqrt (r / (1 - r))
68.     return snr
69.
70. # Function to write text on an image
71. def write_text_on_image(my_file, TEXT):
72.     x_coord=my_file.shape[0]
73.     y_coord=my_file.shape[1]
74.     #bits_of_color=my_file.shape[2]
75.
76.     # Write some Text
77.     font                   = cv2.FONT_HERSHEY_SIMPLEX
78.     bottomLeftCornerOfText = (10,y_coord-200)
79.     fontScale              = 0.8
80.     fontColor              = (255,255,255)
81.     thickness              = 3
82.     lineType               = 2
83.
84.     cv2.putText(my_file,TEXT,
85.                 bottomLeftCornerOfText,
86.                 font,
87.                 fontScale,
88.                 fontColor,
89.                 thickness,
90.                 lineType)
91.     return my_file
92.
93.
94. # function that detects the number of corners in an image
95. def no_of_corners(my_image):
96.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
97.
98.     # The "magic" instruction of corner detection
99.     dst = cv2.cornerHarris(gray,2,3,0.04)
100.
101.     treshold_value=0.01 * dst.max() #getting a treshold value
102.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
103.
104.     dst = numpy.uint8(dst) # the command below needs integer numbers
105.
106.     # Find centroids for a more accurate detection
107.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
108.     # corners being detected (each corner has a numberic label. The highest numerical label
109.     # imdicates the corners detected. Labels are incraesed one by one as the corner
detector
110.     # finds corners
111.     num_corners = labels.max()
112.
113.     #result is dilated for marking the corners, not important
114.     dst = cv2.dilate(dst,None)
115.
116.     # place gray dots on the image to show where the corners are
117.     gray[dst==255]=[137]
118.
119.     gray = numpy.uint8(gray)
120.     return num_corners, gray
121.
122.
123. # blue median function
124. def blur_median(my_image, blur_kernel):
125.     # ksize (represente the blurring kernel size)
126.     ksize = (blur_kernel, blur_kernel)
127.     image = cv2.blur(my_image, ksize)
128.     return image
129.
130.
131. # formula for gausian blur
```

```
132. def blur_gauss(my_image, blur_kernel):
133.     # ksize (represente the blurring kernel size)
134.     # we user the 2n+1 formula for obaining odd numbers.
135.     # gausian blur works with odd numbers
136.     ksize = (2*blur_kernel+1, 2*blur_kernel+1)
137.     image = cv2.GaussianBlur(my_image, ksize, 0)
138.     return image
139.
140.
141. # a function to detect the blur value of any image. The higher the number returned,
142. # the less blur there is present in the image
143. def blur_value(my_image):
144.     value = cv2.Laplacian(my_image, cv2.CV_64F).var()
145.     return value
146.
147.
148. # ---------------------------------- MAIN ----------------------------------
149.
150. # Read my synthetic image
151. image_s="UNO.JPG"
152. image = cv2.imread(image_s, 0)
153.
154. # Read 2 real images in order to computer their SNR and detect their blur value
155. im1_s="NIKON300curtainsO1.jpg"
156. im2_s="NIKON300curtainsO2.jpg"
157. im1=cv2.imread(im1_s,0)
158. im2=cv2.imread(im2_s,0)
159.
160. # get the blur value detected in the real images
161. blur_real1=blur_value(im1)
162. blur_real2=blur_value(im2)
163.
164. # get a mean blur value between the two images
165. value_mean=int(numpy.mean([blur_real1,blur_real2]))
166. print("IMAGE: "+im1_s+", BLUR: ", blur_real1)
167. print("IMAGE: "+im2_s+", BLUR: ", blur_real2)
168. print("MEAN BLUR: ", value_mean)
169. print("\n")
170.
171.
172. # --------------------------- APPLY BLUR 1st and NOISE 2nd -----------------------------
173.
174. # The blue value of the virtual image should be high initially, as the image is crisp
175. blur_virt=int(blur_value(image))
176. print("VIRTUAL IMAGE blur value (before BLUR): ", blur_virt)
177.
178. # The 6 line of code below applies increasing levels of blur to the virtual image,
179. # until the blur value of the virtual image matches the blur value detected in the
180. # real images above. This is how I match the the blurr
181. counter=0
182. value_blur=blur_virt
183. while value_blur > value_mean:
184.     counter+=1
185.     my_blured_image = blur_gauss(image, counter)
186.     value_blur=int(blur_value(my_blured_image))
187.
188. print("VIRTUAL IMAGE has BLUR: "+str(value_blur)+", with gausian kernel of ", counter)
189.
190. # Return the SNR of the real images
191. SNR_REAL=compute_snr_real(im1,im2)
192. print("REAL_SNR: ", SNR_REAL)
193.
194. # Calculate the noise needed to apply to the virtual image in order for it to have
195. # the same SNR as the real image. The reasoning for the following formula is this:
196. # if we have an SNR of 5.365, this meaning that the fration SIGNAL / NOISE = 5.365,
197. # then we can determine the noise of the image by calculating NOISE = SIGNAL (100%) / 5.365
198. # This way we get the exact level of noise that the real image contains in order to produce
199. # the SNR of 5.365. This level of noise we apply to the virtual image so that we will also
200. # obtain an SNR of 5.365 exactly as the SNR obtained from the REAL images.
201. noise_applied=100.0/SNR_REAL
```

```
202.
203. # SNR of the virtual image (for verification)
204. SNR, noisy_image  = add_noise_get_noisy_image_get_snr(my_blured_image, noise_applied)
205. print("VIRTUAL_SNR: ", SNR)
206.
207. # Just to show the original virtual image, and detect its blur which will be high due to
being
208. # a crips image.
209. cv2.imshow("0 - Virtual Img - Blur value BEFORE blur: "+str(blur_virt)+", NO NOISE YET",
image)
210.
211. # To show the virtual image after the necessary blur and noise value were applied
212. cv2.imshow("1 - Virtual Img - Add noise to match real Img SNR:
"+str(round(SNR,2)),noisy_image)
213.
214. # CORNER DETECTOR - Now we detect corners on the resulting blurry and noisy image
215. returned_corner, returned_image=no_of_corners(noisy_image)
216. cv2.imshow("1 - Virtual Img - BLUR 1st, NOISE 2nd, Corners detected:
"+str(returned_corner),returned_image)
217.
218. # NOTE!!!!
219. # The corner detector shows a lot of corners due to the noise being applied and due to it
being crisp.
220. # An intereting thing to see is how the corner detector behaves when the noise is applied
1st and then the image
221. # is blured. This will also blut the noise which will make the image less crips, hence the
corner detector
222. # should not detect as many spurious corners. This will be implemented below.
223.
224.
225. # --------------- APPLY NOISE 1st and BLUR 2nd, then CHECK SNR------------------
226.
227.
228. # Calculate the noise needed to apply to the virtual image in order to get
229. # the same SNR as the real image
230. noise_applied=100.0/SNR_REAL
231.
232.
233. SNR2, noisy_image2  = add_noise_get_noisy_image_get_snr(image, noise_applied) # SNR of the
virtual image (for verification)
234.
235. # now we apply the right amount of blur to the already noisy image. The code below has
236. # been already used above, just the order of application is now reversed
237. blur_virt2=int(blur_value(noisy_image2))
238. counter=0
239. value_blur2=blur_virt2
240. while value_blur2 > value_mean:
241.     counter+=1
242.     my_blured_image2 = blur_gauss(noisy_image2, counter)
243.     value_blur2=int(blur_value(my_blured_image2))
244.
245. cv2.imshow("2 - Virtual Img - Matched blur: "+str(value_blur2), my_blured_image2)
246.
247. # CORNER DETECTOR
248. returned_corner2, returned_image2=no_of_corners(my_blured_image2)
249. cv2.imshow("2 - Virtual Img - NOISE 1st, BLUR 2nd, Corners detected:
"+str(returned_corner2),returned_image2)
250.
251. # NOTE!!!
252. # This methid works better as the corner detector doesn't get lost in detecting as many
spurious
253. # corners due to the crispness of the noise as it happenes in the method above.
254. # It is expected that when running the same corner detector on a similarly blured and
similarly noisy
255. # real image, the corner detector should work the same, but it doesn't. The code below
shows this.
256.
257.
258. # --------------------- REAL IMAGE CORNER DETECTION -------------------------
259.
```

```
returned_corner3, returned_image3=no_of_corners(im1)
cv2.imshow("3 - Real Img ORIGINAL SIZE, Corners detected:
"+str(returned_corner3),returned_image3)

# Check the same image but resized

ratio = float(im1.shape[0]) / im1.shape[1]
im3 = cv2.resize(im1, (800, int(800*ratio)))

returned_corner4, returned_image4=no_of_corners(im3)
cv2.imshow("3 - Real Img RESIZED, Corners detected:
"+str(returned_corner4),returned_image4)
```

## Add Gaussian Noise

```
 2. import numpy
 3. import cv2
 4.
 5.
 6. def add_gaussian_noise (image, mean=0.0, sd=1.0, seed=None):
 7.     """
 8.     Add Gaussian-distributed noise to each pixel of image `im`.
 9.
10.     Arguments:
11.       im  the image to which noise will be added (modified)
12.     mean  the mean of the Gaussian-distributed noise (default: 0.0)
13.       sd  the standard deviation of the noise (default: 1.0)
14.     seed  if supplied, the value used to seed the random number generator
15.     """
16.     im = cv2.imread(image)
17.     gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
18.
19.     if not seed is None: numpy.random.seed (seed)
20.     gray += numpy.random.normal (mean, sd, gray.shape)
21.
22.
23.
24. add_gaussian_noise("ONE.JPG",0, 1, None)
25.
```

## Applying Blur to Images (version 2)

```
# this program is supposed to apply the same blur in the virtual image as that found
# in the real image. The way to calculate this is to get the real image, and get the values
# of a single line, preferably the hotizonbtal line at the middle of the screen.
# We then look for the gradient of change from the lowest value of background to
# the highest value of the foreground. We then see how many values exist between
# these 2 extreme values. This will be our GRADIENT. Then on the virtual image we will
# blur the image consecutively until we get a similar gradient of values between foreground
# and background

import cv2
import scipy.stats
import numpy
import math

def add_noise_get_noisy_image_match_snr(my_image, match, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = my_image
    im1 = im1.astype("float32")
    im2 = im1.copy()
    sd=im1.std(ddof=0)

    # the routine below keeps adding noise to our virtual image until the SNR
    # is matched
    noise_added=1.0  #just so that the operation runs faster (normally it is 0.0)
    while 1:
        noise_added+=0.1
        noise_added=round(noise_added,2)

        temp1 = numpy.add(im1, numpy.random.normal (mean, noise_added,
im1.shape))
        temp2 = numpy.add(im2, numpy.random.normal (mean, noise_added, im2.shape))
        r = scipy.stats.pearsonr (temp1.ravel(), temp2.ravel())[0]
        snr = math.sqrt (r / (1 - r))

        print("MATCHING SNR: "+str(snr)+", added noise: "+str(noise_added)+\
              ", which is "+str(round((noise_added/sd)*100,2))+"% of SD of image")
        if (round(snr,1) <= round(match,1)):
            break

    # make sure there are no values lower than 0 and higher than 255 from the
    # random noise application
    noisy_for_display = numpy.clip(temp1, 0, 255)
    noisy_for_display = numpy.uint8(noisy_for_display)

    print("\n")
    print("VIRTUAL SNR: "+str(snr)+" was matched to REAL SNR: "+str(match))
    print("\n")
    return snr, noisy_for_display

# function that detects the number of corners in an image
def no_of_corners(my_image):
    gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float

    # The "magic" instruction of corner detection
    dst = cv2.cornerHarris(gray,2,3,0.04)

    treshold_value=0.01 * dst.max() #getting a treshold value
    tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)

    dst = numpy.uint8(dst) # the command below needs integer numbers

    # Find centroids for a more accurate detection
    ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
    # corners being detected (each corner has a numberic label. The highest numerical label
```

```
    # imdicates the corners detected. Labels are incraesed one by one as the corner detector
    # finds corners
    num_corners = labels.max()

    #result is dilated for marking the corners, not important
    dst = cv2.dilate(dst,None)

    # place gray dots on the image to show where the corners are
    gray[dst==255]=[137]

    gray = numpy.uint8(gray)
    return num_corners, gray

# This is just a function for quikcly computing the SNR of 2 images.
def compute_snr_real(my_image1, my_image2):

    im1 = my_image1.astype("float32")
    im2 = my_image2.astype("float32")

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    snr = math.sqrt (r / (1 - r))
    return snr

def return_virtual_generated_image_based_on_real_image(REAL_Image, amplitude=10, frequency=1):
    # I will calcuate the background and foreground of the image in order to generate an
    # artificial image with the same background and foreground shades of gray as the real image

    # the size (in pixels) of the side of a square in the center of the screen where I will sample each
    # pixel in the real image for the shade of gray required for the foreground of
    # of the virtual image. Then the mean value will be the shade of gray that will be used to build
    # the foreground in the virtual image
    gray1=REAL_Image
    square_side=20
    all_pixels_from_square=[]
    for i in range (gray1.shape[0]//2-square_side//2, gray1.shape[0]//2+square_side//2,1):
        for j in range (gray1.shape[1]//2-square_side//2, gray1.shape[1]//2+square_side//2,1):
            all_pixels_from_square.append(gray1[i,j])
    fore_ground=int(numpy.mean(all_pixels_from_square))

    # the side of the sqare starting with topmost and leftomost pixel of the
    # real image in order to detect tge bacground shade of gray that will be applied
    # to the background of the virtual image
    all_pixels_from_square=[]
    for i in range (0, square_side,1):
        for j in range (0, square_side,1):
            all_pixels_from_square.append(gray1[i,j])
    back_ground=int(numpy.mean(all_pixels_from_square))

    # generate virtual image to resemble the real image
    gray_generated=numpy.zeros((len(gray1),len(gray1[0])))

    gray_generated[:,:]=back_ground  # apply backround share of gray

    # SINE VARIATON to THE BACKGROUND - useless as this detail gets lost with blur
#    counter=0
#    for i in range(0,gray_generated.shape[0],1):
#        for j in range(0,gray_generated.shape[1],1):
#            gray_generated[i,j] = int(math.sin(frequency*counter)*amplitude)+back_ground
#            counter+=0.1

    # just in case we will have values larger than 255 and smaller than 0
    gray_generated = numpy.clip(gray_generated, a_min = 0, a_max = 255)

```

```
128.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
129.
130.     # ratio of how far the white sqare is from the boundaries of the screen.
131.     # the ratio is both for the X and for the Y coordinates. How I did it? I took
132.     # an example and then I calculated the ratio between the maximum X resolution
133.     # and how many pixles away was the cube from 0. In our case it was 600
134.     # pixels away. So the X resolution (4288 pixes) divied by 600 gives us a ratio
135.     # of 7.14 This means that for any resolution picture if I keep the same ratio, the
square
136.     # in the middle of the real picture will sit almost in the same position in the virtual
image.
137.     ratio_X=7.1466666666666665
138.     ratio_Y=4.746666666666667
139.     Y_value=int(len(gray_generated)//ratio_Y)
140.     X_value=int(len(gray_generated[0])//ratio_X)
141.
142.     # create the foreground (a white square) in the middle of the image (background)
143.     gray_generated[len(gray_generated)//2-X_value:len(gray_generated)//2+X_value,\
144.                    len(gray_generated[0])//2-
Y_value:len(gray_generated[0])//2+Y_value]=fore_ground  #after this command
145.     # we have our image with a cube in the middle generated, just like the real image.
146.     # Now we will return this virtual image.
147.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
148.     return gray_generated
149.
150. # formula for gausian blur
151. def blur_gauss(my_image, blur_kernel):
152.     # ksize (represente the blurring kernel size)
153.     # we user the 2n+1 formula for obaining odd numbers.
154.     # gausian blur works ONLY with odd numbers
155.     ksize = (2*blur_kernel+1, 2*blur_kernel+1)
156.     image = cv2.GaussianBlur(my_image, ksize, 0)
157.     return image
158.
159. def get_blur_gradient(my_image):
160.     # for the sorting function below
161.     from operator import itemgetter
162.
163.     middle_line = my_image[my_image.shape[0]/2] # get all the values from the middle line
of image
164.     middle_line = middle_line.tolist()  # convert to Pyhon list from numpy array
165.
166.     # count how many times each elemenbts occurs.
167.     count_elements=[]
168.     for i in set(middle_line):
169.         count_elements.append([i, middle_line.count(i)])
170.
171.     # sort elements in ascending order, but the sort happens according to the number
172.     # of occurences of each grey value (this is the 2nd value in the tuple.)
173.     # This means that the background and foreground values will be placed last
174.     # in the list. This is because those values should have the most of occurences
175.     # in our sample data.
176.     count_elements.sort(key=itemgetter(1))
177.
178.     # we can easily get the background and foreground values by taking the last 2 elements
179.     # in the above sorted list
180.     foreground=max(count_elements[-1],count_elements[-2])[0]
181.     background=min(count_elements[-1],count_elements[-2])[0]
182.
183.     # by getting the length of the above list, we will know how many distinct gray values
have
184.     # been found in total. We are not concerned with knowing the grey valur itself, but
with how many
185.     # different grey values have been found in total. We substract 2 because we are not
concerned
186.     # with the background and the foreground
187.     how_many_grey_valyes=len(count_elements)-2
188.
189.     # Because our simulation picture is simetrical (square in the middle of the screen),
190.     # we can divide the above value in 2. This will give us the number of grey values
```

```
191.     # that exists between background and foreground. This is exactly the gradient, or the
number
192.     # of intermediary values between foreground and background that needs to be matched
when applying
193.     # blur. We will apply so much blur to our UNBLURED image until we get the same blur
gradient
194.     # like in the original image.
195.
196.     gradient=int(math.ceil(how_many_grey_valyes/2.0))
197.
198.     # we also return the foreground and background of the image, just in chase we need to
compare
199.     # these values with values that the image had originally.
200.     return gradient, foreground, background
201.
202. # ------------------------------- MAIN ---------------------------------------
203.
204. im1_s="UNO_BLUR_3.jpg"
205. im2_s="UNO_BLUR_5.jpg"
206. im4_s="UNO_PHOTOSHOP.jpg"
207.
208. im1=cv2.imread(im1_s,0)
209. im2=cv2.imread(im2_s,0)
210. im4=cv2.imread(im4_s,0)
211.
212. # Get the real images SNR so that we can match it for the virtual images
213. SNR_REAL=compute_snr_real(im1,im2)
214. print("REAL SNR: ", SNR_REAL)
215.
216. # detect how much blur is there in one of the "real" images
217. gradient, fgnd, bgnd=get_blur_gradient(im2)
218.
219. virtual_generated_image=return_virtual_generated_image_based_on_real_image(im2,
amplitude=30, frequency=5)
220. # Add noise to the image
221. SNR2, noisy_image  = add_noise_get_noisy_image_match_snr(virtual_generated_image, SNR_REAL)
222.
223. counter=1
224. while 1:
225.     blurred_image=blur_gauss(noisy_image, counter)
226.     gradient2, fgnd, bgnd=get_blur_gradient(blurred_image)
227.     print("Current blur gradient: "+str(gradient2)+", to match REAL IMAGE blur gradient:
"+str(gradient))
228.      # without the -2 ofset, the matched virtual image appears more blurry visually than
the real image,
229.      # despite both having the same gradient.
230.     if (gradient2 >= gradient-2):
231.         break
232.     counter+=1
233.
234. # -------------------- Virtual IMAGE CORNER DETECTION -------------------------
235. returned_corner2, returned_image2=no_of_corners(blurred_image)
236. cv2.imshow("2 - Virtual Img -  Corners detected: "+str(returned_corner2), returned_image2)
237.
238. # -------------------- REAL IMAGE CORNER DETECTION -------------------------
239. returned_corner3, returned_image3=no_of_corners(im2)
240. cv2.imshow("3 - Real Img - Corners detected: "+str(returned_corner3), returned_image3)
241.
242. # -------------------- REAL IMAGE CORNER DETECTION -------------------------
243. returned_corner4, returned_image4=no_of_corners(im4)
244. cv2.imshow("4 - Photoshop Img - Corners detected: "+str(returned_corner4), returned_image4)
245.
246. if cv2.waitKey(0) & 0xff == 27:
247.         cv2.destroyAllWindows()
248.
```

## SNR Calculation (Version 1)

```
 1. import cv2
 2. import scipy.stats
 3. import math
 4.
 5.
 6.
 7. def no_of_corners(my_image):
 8.
 9.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
10.
11.     dst = cv2.cornerHarris(gray,2,3,0.04)
12.
13.     treshold_value=0.01 * dst.max() #getting a treshold value
14.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
15.
16.     dst = numpy.uint8(dst) # the command below needs integer numbers
17.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
18.     num_corners = labels.max() # corners being detected
19.
20.      # if we have negative numbers below 0 (like in a true BW image) then the command to
change to UINT8
21.     # below will place instead of a negative number, the value of 255-(abs(that negative
number)).
22.     # this means that we will have false numbers that are close to 255. Converting all
numbers to
23.     # their absolute value, will fix this problem
24.     gray = numpy.absolute(gray) # we do not allow any negative nubmers
25.     # (corresponding to the level of gray) in the image (otherwise we will not be able to
display properly)
26.
27.     # If the numbers are larger than 255, then when converting to uint8, the numvbers
28.     # will be (that number larger than 255)-255, and this will yield numbers close to 0.
29.     # The idea is to limit numbers larger than 255 to 255.
30.
31.     for i in range(0,gray.shape[0],1):
32.         for j in range(0,gray.shape[1],1):
33.             if gray[i][j]>255:
34.                 temp=gray[i][j]-255  #eg. 256.345-255= 1.34500
35.                 gray[i][j]=255-temp
36.
37.
38.     #result is dilated for marking the corners, not important
39.     dst = cv2.dilate(dst,None)
40.
41.     # place gray dots on the image to show where the corners are
42.     gray[dst==255]=[137]
43.
44.     gray = numpy.uint8(gray)
45.     return num_corners, gray
46.
47.
48. def no_of_corners_after_applied_noise(my_image, factor, mean=0.0, sd=1.0, seed=None):
49.
50.     # this is for being able to display the corners in red on the noisyt image
51.     #clone=numpy.zeros((len(my_image),len(my_image[0]),3))
52.
53.     gray = numpy.float32(my_image) # convert to float for data manipulation
54.     if not seed is None: numpy.random.seed (seed)
55.     " add the noise "
56.
57.     sd=gray.std(ddof=0)
58.     percentage_of_noise_added=sd*factor/100.0
59.
```

```
 60.     gray_and_noisy = numpy.add(gray, numpy.random.normal (mean, percentage_of_noise_added,
gray.shape))
 61.     gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32
 62.
 63.     dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)
 64.
 65.     treshold_value=0.01 * dst.max() #getting a treshold value
 66.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
 67.
 68.     dst = numpy.uint8(dst) # the command below needs integer numbers
 69.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 70.     num_corners = labels.max() # corners being detected
 71.
 72.     # if we have negative numbers below 0 (like in a true BW image) then the command to
change to UINT8
 73.     # below will place instead of a negative number, the value of 255-(abs(that negative
number)).
 74.     # this means that we will have false numbers that are close to 255. Converting all
numbers to
 75.     # their absolute value, will fix this problem
 76.     gray_and_noisy = numpy.absolute(gray_and_noisy) # we do not allow any negative nubmers
 77.     # (corresponding to the level of gray) in the image (otherwise we will not be able to
display properly)
 78.
 79.     # If the numbers are larger than 255, then when converting to uint8, the numvbers
 80.     # will be (that number larger than 255)-255, and this will yield numbers close to 0.
 81.     # The idea is to limit numbers larger than 255 to 255.
 82.
 83.     for i in range(0,gray_and_noisy.shape[0],1):
 84.         for j in range(0,gray_and_noisy.shape[1],1):
 85.             if gray_and_noisy[i][j]>255:
 86.                 temp=gray_and_noisy[i][j]-255  #eg. 256.345-255= 1.34500
 87.                 gray_and_noisy[i][j]=255-temp
 88.
 89.
 90.     #result is dilated for marking the corners, not important
 91.     dst = cv2.dilate(dst,None)
 92.
 93.     # place gray dots on the image to show where the corners are
 94.     gray_and_noisy[dst==255]=[137]
 95.
 96.     gray_and_noisy = numpy.uint8(gray_and_noisy)
 97.
 98.     return num_corners, gray_and_noisy
 99.
100.
101.
102. def compute_virtual_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
103.     # Read in the image and convert to float.  Duplicate it and add noise.
104.     im1 = my_image
105.     im1 = im1.astype("float32")
106.     im2 = im1.copy()
107.
108.     sd=im1.std(ddof=0)
109.     percentage_of_noise_added=sd*factor/100.0
110.
111.    # percentage_of_noise_added=5
112.
113.     im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added, im1.shape))
114.     im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added, im2.shape))
115.
116.     # Compute the correlation coefficient and SNR.
117.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
118.     #print ("r:", r)
119.     snr = math.sqrt (r / (1 - r))
120.     #print ("SNR:", snr)
121.
122.     return snr
123.
124.
```

```
125.
126. def compute_snr_real(my_image1, my_image2):
127.
128.     im1 = my_image1.astype("float32")
129.     im2 = my_image2.astype("float32")
130.
131.     # Compute the correlation coefficient and SNR.
132.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
133.     #print ("r:", r)
134.     snr = math.sqrt (r / (1 - r))
135.     #print ("SNR:", snr)
136.
137.     return snr
138.
139.
140. def corners_vs_constrast(my_image, factor, mean=0.0, sd=1.0, seed=None):
141.
142.     #ADD NOISE TO IMAGE
143.     im = cv2.imread(my_image)
144.     #CALCULATE NUMBER OF CORNERS
145.
146.     gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
147.     gray = numpy.float32(gray) # convert to float for data manipulation
148.     contrast=int(gray.max()-gray.min())
149.     if not seed is None: numpy.random.seed (seed)
150.     " add the noise "
151.
152.     sd=gray.std(ddof=0)
153.     percentage_of_noise_added=sd*factor/100.0
154.
155.     gray_and_noisy = numpy.add(gray, numpy.random.normal (mean, percentage_of_noise_added,
gray.shape))
156.     gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32
157.
158.     dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)
159.
160.     treshold_value=0.01 * dst.max() #getting a treshold value
161.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
162.
163.     dst = numpy.uint8(dst) # the command below needs integer numbers
164.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
165.     num_corners = labels.max() # corners being detected
166.
167.     return num_corners, contrast
168.
169. ##################################################  MAIN
##################################################
170.
171.
172. noise_applied=30
173. FILENAME="UNO.JPG"
174.
175. im1 = cv2.imread(FILENAME)
176. gray1 = cv2.cvtColor(im1,cv2.COLOR_BGR2GRAY)
177.
178. # get number of corners in the image
179. real_corners, real_image_with_corners_overlayed=no_of_corners(gray1)
180. # get number of corners after noise has been applied to file
181. corners_after_noise, noisy_image_with_corners_overlayed =
no_of_corners_after_applied_noise(gray1,noise_applied)
182. SNR = compute_virtual_snr(gray1,noise_applied)
183. print ("SNR: ",SNR)
184.
185.
186. cv2.imshow('NOISE '+str(noise_applied)+ '%, '+\
187.             "SPURIOUS "+str(corners_after_noise-real_corners)+\
188.             ", SNR "+str(round(SNR,4)),noisy_image_with_corners_overlayed)
189. cv2.imshow('ORIGINAL image SHOULD have '+str(real_corners)+"
CORNERS",real_image_with_corners_overlayed)
190.
```

```
191.
192.
193. ###############################################################################
194.
195.
196. #FILENAME2="SCAN1.png"; im2 = cv2.imread(FILENAME2); gray2 =
cv2.cvtColor(im2,cv2.COLOR_BGR2GRAY)
197. #FILENAME3="SCAN2.png"; im3 = cv2.imread(FILENAME3); gray3 =
cv2.cvtColor(im3,cv2.COLOR_BGR2GRAY)
198. #
199. #
200. #SNR_REAL=compute_snr_real(gray2,gray3)
201. #print ("REAL_SNR: ", SNR_REAL)
202.
203. ###############################################################################
204.
205.
206. FILENAME2="SCAN1.png"; im2=cv2.imread(FILENAME2);
gray2=cv2.cvtColor(im2,cv2.COLOR_BGR2GRAY)
207. real_corners2, real_image_with_corners_overlayed2=no_of_corners(gray2)
208. corners_after_noise2, noisy_image_with_corners_overlayed2 =
no_of_corners_after_applied_noise(gray2,noise_applied)
209. SNR2 = compute_virtual_snr(gray2,noise_applied)
210. print ("SNR2: ",SNR2)
211.
212.
213. cv2.imshow('_NOISE '+str(noise_applied)+ '%, '+\
214.            "_SPURIOUS "+str(corners_after_noise2-real_corners2)+\
215.            ", _SNR "+str(round(SNR,4)),noisy_image_with_corners_overlayed2)
216. cv2.imshow('_ORIGINAL image SHOULD have '+str(real_corners2)+"
CORNERS",real_image_with_corners_overlayed2)
217.
218.
219.
220.
221.
222.
223. if cv2.waitKey(0) & 0xff == 27:
224.         cv2.destroyAllWindows()
225.
```

## SNR Calculation (Version 2)

```
  1. # generate virtual image with the same SNR as the real image and then see if the corner
detector works the same on
  2. # virtual image as it works on the real image
  3.
  4.
  5. import numpy
  6. import cv2
  7. import scipy.stats
  8. import math
  9.
 10.
 11.
 12. def no_of_corners(my_image):
 13.
 14.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
 15.
 16.     dst = cv2.cornerHarris(gray,2,3,0.04)
 17.
 18.     treshold_value=0.01 * dst.max() #getting a treshold value
 19.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
 20.
 21.     dst = numpy.uint8(dst) # the command below needs integer numbers
 22.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 23.     num_corners = labels.max() # corners being detected
 24.
 25.     #result is dilated for marking the corners, not important
 26.     dst = cv2.dilate(dst,None)
 27.
 28.     # place gray dots on the image to show where the corners are
 29.     gray[dst==255]=[137]
 30.
 31.     gray = numpy.uint8(gray)
 32.     return num_corners, gray
 33.
 34.
 35. def no_of_corners_after_applied_noise(my_image, factor, mean=0.0, sd=1.0, seed=None):
 36.
 37.     # this is for being able to display the corners in red on the noisyt image
 38.     #clone=numpy.zeros((len(my_image),len(my_image[0]),3))
 39.
 40.     gray = numpy.float32(my_image) # convert to float for data manipulation
 41.     if not seed is None: numpy.random.seed (seed)
 42.     " add the noise "
 43.
 44.     sd=gray.std(ddof=0)
 45.     percentage_of_noise_added=sd*factor/100.0
 46.
 47.     gray_and_noisy = numpy.add(gray, numpy.random.normal (mean, percentage_of_noise_added,
gray.shape))
 48.     gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32
 49.
 50.     dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)
 51.
 52.     treshold_value=0.01 * dst.max() #getting a treshold value
 53.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
 54.
 55.     dst = numpy.uint8(dst) # the command below needs integer numbers
 56.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 57.     num_corners = labels.max() # corners being detected
 58.
 59.     # if we have negative numbers below 0 (after random noise has been applied to the
image)
 60.     # then the command to change to UINT8 below will place instead of a negative number,
 61.     # the value of 255-(abs(that negative number)).
```

```
    # this means that we will have false numbers that are close to 255. Converting all
numbers to
    # their absolute value, will fix this problem. We don't need negative pixel values in
the image
    gray_and_noisy_for_display = numpy.absolute(gray_and_noisy) # we do not allow any
negative nubmers
    # (corresponding to the level of gray) in the image (otherwise we will not be able to
display properly)

    # If the numbers are larger than 255, then when converting to uint8, the numvbers
    # will be (that number larger than 255)-255, and this will yield numbers close to 0.
    # The idea is to limit numbers larger than 255 to 255.

    for i in range(0,gray_and_noisy_for_display.shape[0],1):
        for j in range(0,gray_and_noisy_for_display.shape[1],1):
            if gray_and_noisy_for_display[i][j]>255:
                temp=gray_and_noisy_for_display[i][j]-255  #eg. 256.345-255= 1.34500
                gray_and_noisy_for_display[i][j]=255-temp


    #result is dilated for marking the corners, not important
    dst = cv2.dilate(dst,None)

    # place gray dots on the image to show where the corners are
    gray_and_noisy_for_display[dst==255]=[137]

    gray_and_noisy_for_display = numpy.uint8(gray_and_noisy_for_display)

    return num_corners, gray_and_noisy_for_display


def compute_virtual_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = my_image
    im1 = im1.astype("float32")
    im2 = im1.copy()

    sd=im1.std(ddof=0)
    percentage_of_noise_added=sd*factor/100.0

   # percentage_of_noise_added=5

    im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added, im1.shape))
    im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added, im2.shape))

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)

    return snr


def compute_snr_real(my_image1, my_image2):

    im1 = my_image1.astype("float32")
    im2 = my_image2.astype("float32")

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)

    return snr


```

```
128. ##################################################  MAIN
##################################################
129.
130. # read real images
131. FILE_REAL="NIKON300curtainsC1.jpg"
132. FILE_REAL2="NIKON300curtainsC2.jpg"
133. gray1=cv2.imread(FILE_REAL, 0)
134. gray2=cv2.imread(FILE_REAL2, 0)
135. #
136. #FILE_REAL="NIKON300curtainsO1.jpg"
137. #FILE_REAL2="NIKON300curtainsO2.jpg"
138. #gray1=cv2.imread(FILE_REAL, 0)
139. #gray2=cv2.imread(FILE_REAL2, 0)
140.
141. #FILE_REAL="SONYcurtainsO1.jpg"
142. #FILE_REAL2="SONYcurtainsO2.jpg"
143. #gray1=cv2.imread(FILE_REAL, 0)
144. #gray2=cv2.imread(FILE_REAL2, 0)
145.
146. #FILE_REAL="CANON1.jpg"
147. #FILE_REAL2="CANON2.jpg"
148. #gray1=cv2.imread(FILE_REAL, 0)
149. #gray2=cv2.imread(FILE_REAL2, 0)
150.
151. #FILE_REAL="EPSON1.png"
152. #FILE_REAL2="EPSON2.png"
153. #gray1=cv2.imread(FILE_REAL, 0)
154. #gray2=cv2.imread(FILE_REAL2, 0)
155.
156.
157. # calcuate the background and foreground of the image in order to generate an
158. # artificial image with the same background and foreground. Basically we detect
159. # the color of the real image in order to reproduce virtual images with the same colour.
160. square_side=20 # the side of a square in the center of the screen for foreground
161. all_pixels_from_square=[]
162. for i in range (gray1.shape[0]/2-square_side/2, gray1.shape[0]/2+square_side/2,1):
163.     for j in range (gray1.shape[1]/2-square_side/2, gray1.shape[1]/2+square_side/2,1):
164.         all_pixels_from_square.append(gray1[i,j])
165. fore_ground=int(numpy.mean(all_pixels_from_square))
166.
167. # the side of the sqare starting with topmost and leftomost pizel
168. all_pixels_from_square=[]
169. for i in range (0, square_side,1):
170.     for j in range (0, square_side,1):
171.         all_pixels_from_square.append(gray1[i,j])
172. back_ground=int(numpy.mean(all_pixels_from_square))
173.
174.
175. # generate virtual image to resemble the real image
176. gray_generated=numpy.zeros((len(gray1),len(gray1[0])))
177. gray_generated[:,:]=back_ground
178. gray_generated=gray_generated.astype("uint8")
179. # ratio of how far the white sqare is from the boundaries of the screen.
180. # the ratio is both for the X and for the Y coordinates. How I did it? I took
181. # an example and than I calculated thje ratio between the maximum X resolution
182. # and them divided by how many pixles away was the cube. In our case it was 600
183. # pixels away. So the X resolution (4288 pixes) divied by 600 gives us a ratio
184. # of 7. This measn for any resolution picture if I keep the same ration, the square
185. # in the middle of the picture will sit almost in the same position.
186. ratio_X=7.1466666666666665
187. ratio_Y=4.746666666666667
188. Y_value=int(len(gray_generated)/ratio_Y)
189. X_value=int(len(gray_generated[0])/ratio_X)
190. # create a white square in the middle of the image
191. #gray_generated[len(gray_generated)/2-600:len(gray_generated)/2+600,\
192. #               len(gray_generated[0])/2-600:len(gray_generated[0])/2+600]=255
193. gray_generated[len(gray_generated)/2-X_value:len(gray_generated)/2+X_value,\
194.                len(gray_generated[0])/2-
Y_value:len(gray_generated[0])/2+Y_value]=fore_ground  #after this command
195. # we have our image with a cube in the middle generated, and we could even save it
```

```
196. # if we wanted to
197.
198.
199.
200. # ratio of generated image in order to display them resized later
201. max_res=float(max(len(gray_generated),len(gray_generated[0])))
202. min_res=float(min(len(gray_generated),len(gray_generated[0])))
203. image_ratio=max_res/min_res
204. Y_dimension=600 # one dimension of the image always needs be spcified, and based
205. # on this dimesnion we will calculate the X dimension as well using the "image_ratio"
206.
207.
208.
209.
#resize_gray_generated=cv2.resize(gray_generated,(int(Y_dimension*image_ratio),int(Y_dimension))
)
210. ## display the computer generated an resized image on the screen
211. #cv2.imshow("resized_gray_generated", resize_gray_generated)
212. #
213. ##display original resized image to fit on the screen
214. #resize_gray1=cv2.resize(gray1,(int(Y_dimension*image_ratio),int(Y_dimension)))
215. #cv2.imshow("resized_original_image",resize_gray1)
216.
217.
218.
219. SNR_REAL = compute_snr_real(gray1, gray2)
220. print ("SNR of REAL images: ",SNR_REAL)
221.
222.
223.
224. ##################################### VIRTUAL IMAGE ########################
225.
226. #this is the noise I will apply to the virtually generated image
227. noise_applied=25.298    # for NIKON300 curtains closed for SNR of 3
228. #noise_applied=6.626975    # for NIKON300 curtains open for SNR of 15
229. #noise_applied=1.2564155    # for CANON camera for SNR of 79
230. #noise_applied= 1.4228786  # for EPSON camera for SNR of 70
231. #noise_applied=2.68918  # for SONY camera for SNR of 37
232.
233.
234. SNR = compute_virtual_snr(gray_generated, noise_applied)
235. print ("SNR of VIRTUAL images: ",SNR)
236.
237. ## get number of corners in the image
238. corners, image_with_corners_overlayed=no_of_corners(gray_generated)
239. # get number of corners after noise has been applied to file
240. corners_after_noise, noisy_image_with_corners_overlayed =
no_of_corners_after_applied_noise(gray_generated,noise_applied)
241.
242.
243. # resize for display just in case the resolution is outside of the screen
244.
resize_image_with_corners_overlayed=cv2.resize(image_with_corners_overlayed,(int(Y_dimension*ima
ge_ratio),int(Y_dimension)))
245. cv2.imshow('ORIGINAL VIRTUAL image SHOULD have '+str(corners)+"
CORNERS",resize_image_with_corners_overlayed)
246.
247.
248.
resize_noisy_image_with_corners_overlayed=cv2.resize(noisy_image_with_corners_overlayed,(int(Y_d
imension*image_ratio),int(Y_dimension)))
249. cv2.imshow('VIRUAL image NOISE '+str(noise_applied)+ '%, '+\
250.            "REAL "+str(corners)+\
251.            ", SPURIOUS "+str(corners_after_noise-corners)+\
252.            ", TOTAL "+str(corners_after_noise)+\
253.            ", SNR "+str(round(SNR,4)),resize_noisy_image_with_corners_overlayed)
254.
255.
256.
257.
```

```
258. ################################### REAL IMAGE ############################
259.
260. ## get number of corners in the image
261. REAL_corners, REAL_image_with_corners_overlayed=no_of_corners(gray1)
262. # get number of corners after noise has been applied to file
263.
264. # resize for display just in case the resolution is outside of the screen
265.
resize_REAL_image_with_corners_overlayed=cv2.resize(REAL_image_with_corners_overlayed,(int(Y_dim
ension*image_ratio),int(Y_dimension)))
266. cv2.imshow('REAL image has '+str(REAL_corners)+" CORNERS and SNR of "+\
267.            str(round(SNR_REAL,4))+\
268.            ", CONTRAST "+str(max(fore_ground,back_ground)-min(fore_ground,back_ground)),
resize_REAL_image_with_corners_overlayed)
269.
270.
271.
272.
273.
274.
275. if cv2.waitKey(0) & 0xff == 27:
276.         cv2.destroyAllWindows()
277.
278.
279.
280.
281.
282.
283.
```

## Creating Virtual Images

```
# Program for creating various greyscale images to see the difference they make in the SNR

import numpy
import cv2
import scipy.stats
import math

def compute_snr_real(my_image1, my_image2):

    im1 = my_image1.astype("float32")
    im2 = my_image2.astype("float32")

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)

    return snr

def return_gray_FOUR_ways(my_image):
    temp=0.0
    im = cv2.imread(my_image)

    # get "gray_CV"
    gray_CV = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

    gray_mean=numpy.zeros((len(im),len(im[0])))
    gray_mean=gray_mean.astype("uint8")

    gray_RGB=numpy.zeros((len(im),len(im[0])))
    gray_RGB=gray_RGB.astype("uint8")

    gray_BGR=numpy.zeros((len(im),len(im[0])))
    gray_BGR=gray_BGR.astype("uint8")


    for i in range (0, len(im)):
        for j in range (0, len(im[0])):
            # get "gray_mean"
            temp=int((int(im[i,j,0])+int(im[i,j,1])+int(im[i,j,2]))/3)
            gray_mean[i,j]=temp
            # get "gray_RGB"
            temp=int((im[i,j,0]*0.299+im[i,j,1]*0.587+im[i,j,2]*0.114))
            gray_RGB[i,j]=temp
            # get "gray_BGR"
            temp=int((im[i,j,0]*0.114+im[i,j,1]*0.587+im[i,j,2]*0.299))
            gray_BGR[i,j]=temp

    return im, gray_mean, gray_RGB, gray_BGR, gray_CV

my_image1="SCAN1.png"
my_image2="SCAN2.png"

COLOUR1, MEAN1, RGB1, BGR1, CV1=return_gray_FOUR_ways(my_image1)
COLOUR2, MEAN2, RGB2, BGR2, CV2=return_gray_FOUR_ways(my_image2)

SNR=compute_snr_real(COLOUR1,COLOUR2)
print ("SNR_COLOUR: ",SNR)
SNR=compute_snr_real(MEAN1,MEAN2)
print ("SNR_GRAY_MEAN: ",SNR)
SNR=compute_snr_real(RGB1,RGB2)
print ("SNR_GRAY_RGB: ",SNR)
SNR=compute_snr_real(BGR1,BGR2)
print ("SNR_GRAY_BGR: ",SNR)
```

```
SNR=compute_snr_real(CV1,CV2)
print ("SNR_CV: ",SNR)


"""cv2.imshow('ORIGINAL',im)
cv2.imshow('Gray_MEAN',gray_mean)         F
cv2.imshow('Gray_RGB',gray_RGB)
cv2.imshow('Gray_BGR',gray_BGR)
cv2.imshow('Gray_BGR',gray_BGR)
cv2.imshow('Gray_CV',gray_CV)


if cv2.waitKey(0) & 0xff == 27:
        cv2.destroyAllWindows()"""

```

## Automatic contrast increase one image at a time

```
# program for incrasing the contrast between images

import cv2
import numpy
import scipy.stats
import math


def CHANGE_LUMINOSITY(my_image, increment):
    im_changed = my_image.copy()

    for i in range (0, len(my_image)):
        for j in range (0, len(my_image[0])):

            # for colour images (usually RGB)
            if my_image.shape[-1]==3:

            # deal with the limits, and not allow colors higher than 255
            # and colors neither lower than 0.
                if (my_image[i,j,0]+increment>255): im_changed[i,j,0]=255
                if (my_image[i,j,1]+increment>255): im_changed[i,j,1]=255
                if (my_image[i,j,2]+increment>255): im_changed[i,j,2]=255

                if (my_image[i,j,0]+increment<0): im_changed[i,j,0]=0
                if (my_image[i,j,1]+increment<0): im_changed[i,j,1]=0
                if (my_image[i,j,2]+increment<0): im_changed[i,j,2]=0

                if (my_image[i,j,0]+increment<=255) and (my_image[i,j,0]+increment>=0):
im_changed[i,j,0]=my_image[i,j,0]+increment
                if (my_image[i,j,1]+increment<=255) and (my_image[i,j,1]+increment>=0):
im_changed[i,j,1]=my_image[i,j,1]+increment
                if (my_image[i,j,2]+increment<=255) and (my_image[i,j,2]+increment>=0):
im_changed[i,j,2]=my_image[i,j,2]+increment

                # for greyscale images

            else:

                if (my_image[i,j]+increment>255): im_changed[i,j]=255
                if (my_image[i,j]+increment<0): im_changed[i,j]=0
                if (my_image[i,j]+increment<=255) and (my_image[i,j]+increment>=0):
im_changed[i,j]=my_image[i,j]+increment


    #im_changed=im_changed.astype("uint8")


    return im_changed



#my_image3="FOUR.JPG"
#
#im = cv2.imread(my_image3)
#gray_CV = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
#cv2.imwrite("FOUR_GREY.JPG",gray_CV)
#
#image_after=CHANGE_LUMINOSITY(gray_CV,-50)
#cv2.imwrite("FOUR_GREY_CHANGED.JPG",image_after)
#
#cv2.imshow("MY_IMAGE_BEFORE", im)
#cv2.imshow("MY_IMAGE_AFTER", image_after)


my_image3="FOUR.JPG"
```

```
62.
63. im = cv2.imread(my_image3)
64. image_after=CHANGE_LUMINOSITY(im,-50)
65. cv2.imwrite("FOUR_CHANGED.JPG",image_after)
66.
67. cv2.imshow("MY_IMAGE_BEFORE", im)
68. cv2.imshow("MY_IMAGE_AFTER", image_after)
69.
70.
71.
72.
73. if cv2.waitKey(0) & 0xff == 27:
74.         cv2.destroyAllWindows()
75.
76.
77.
78.
79.
```

## Automatic corner detector program

```
 1. import cv2
 2. import numpy as np
 3.
 4. filename = 'ONE.jpg'
 5. img = cv2.imread(filename)
 6. gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
 7.
 8. gray = np.float32(gray)
 9. dst = cv2.cornerHarris(gray,2,3,0.04)
10.
11. # all the values resulted from the Harris computation are very large numbers,
12. # and very low numbers. The very large numbers will be all limited to the
13. # treshold value below, and the rest will be made zero (0). This means that in the end
14. # we will have an area of 255 values around where the corners have been detected,
15. # and in order to consider that area as 1 single corner with one numerical value,
16. # we will have to get the centroid. The centroid is just one number of that area.
17. # After this we can compute the right number of corners in numeric format by
18. # by counting how many centroids we have, this will give us the right number
19. # of corners.
20. treshold_value=0.01 * dst.max() #getting a treshold value
21.
22. # values exceeding the treshold are assigned 255, values below the treshold
23. # are assigned 0. This will give us an image with 4 regions of 255 values,
24. # assuming the image has 4 corners.
25. tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
26.
27. # from that region of 255 values, we must leave just one, in the centre, thus
28. # this is called the centroid. We do this with the instruction below.
29. # The "ret" variable will hold the number of returned corners. But if out image
30. # has 4 corners, then it will find an extra corner that coresponds to the centroid
31. # of all the values of zero (0) that are found around those corners in the whole image.
32. # The variable"labels" places labels in incrementing numbers to each corners. So we could
33. # use "labels.max()" command to find the maximum numerical label in the image. This
34. # will be 4 if let's say we have 4 corners.
35.
36. dst = np.uint8(dst) # the command below needs integer numbers
37. ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
38. num_corners = labels.max()
39.
40. #result is dilated for visualising corners. This increases the regions that
41. # contain 255 values in order to create a bigger visual mark on the image to
42. # show where exactly on the image the corners have been detected.
43. dst = cv2.dilate(dst,None)
44.
45. # place a red DOT on the original image coresponding to the corners found.
46. # this instruction passes an array of logical values to the "img" variable. This
47. # array of logical values has TRUE only where there is 255 in the image where
48. # corners habe been detected. Then the coordinates where there is TRUE are passed
49. # to the IMG variable which assigns the RED colour (BGR colourspace) to the
50. # original image the is used for display.
51. img[dst==255]=[0,0,255]
52.
53. print("We have: "+str(num_corners)+" CORNERS")
54.
55. cv2.imshow('dst',img)
56. if cv2.waitKey(0) & 0xff == 27:
57.     cv2.destroyAllWindows()
58.
```

## SNR calculation added graphically onto JPG image (command line)

```
import numpy
import cv2
import scipy
import math
import scipy.stats
import sys


def compute_snr(my_image, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = cv2.imread (my_image)
    im1 = im1.astype("float32")
    im2 = im1.copy ()

    im1 = numpy.add(im2, numpy.random.normal (mean, sd, im1.shape))
    im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)

    return snr



def noise_vs_corners(my_image, mean=0.0, sd=1.0, seed=None):

    # Function to add noise to the image,
    # and then get number of corners that are detected with this noise

    #ADD NOISE TO IMAGE

    im = cv2.imread(my_image)
    "convert to float for manipulation as OpenCV only works with UNINT8"
    im2 = im.astype("float64")

    if not seed is None: numpy.random.seed (seed)
    " add the noise "
    im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
    " change back to UINT8 for display with"
    im2 = im2.astype("uint8")

    #CALCULATE NUMBER OF CORNERS

    gray = cv2.cvtColor(im2,cv2.COLOR_BGR2GRAY)

    gray = numpy.float32(gray)
    dst = cv2.cornerHarris(gray,2,3,0.04)
    "number of corner edtections"
    num_corners = numpy.sum(dst > 0.01 * dst.max())

    return num_corners


# the following code is for graph creation and saving the result to file


SNR=[]
STD=[]
NO_OF_CORNERS=[]
temp=[]

# I add noise in percentage from 0% to 30% with 1% step.
```

```
66. # More than 30% noise I noticed is useless, and just shows that
67. # the corner detector is outputing data I cannot use.
68. for x in numpy.arange (0,0.3,0.01):
69. 
70.     # A few iterations as the same STD gives off a different number of corners
71.     # each time. I am taking the average of the result in order not to have
72.     # such a "pointy" graph
73.     for y in range (1,3):
74.         data = noise_vs_corners(sys.argv[1], mean=0, sd=x*255)
75.         temp.append(data)
76.     # calculate mean from the total number of iterations
77.     mean=sum(temp) / len (temp)
78.     temp=[]
79. 
80.     STD.append(x)  # the values here are in percentage
81.     # STD.append(x*255) # the values here are in STDs
82.     NO_OF_CORNERS.append(mean)
83. 
84.     # getting the SNR values
85.     value=compute_snr(sys.argv[1], mean=0, sd=x*255)
86.     SNR.append(value)
87. 
88. 
89. 
90. # code for graph generation and saving it to the file
91. 
92. import matplotlib.pyplot as plt
93. 
94. plt.plot(SNR,NO_OF_CORNERS)
95. 
96. #plt.show()
97. 
98. plt.savefig("SNR_CORNERS_"+sys.argv[1][:-4])
99. 
```

## SNR calculation added graphically onto JPG image (AUTO)

```
import numpy
import cv2
import scipy
import math
import scipy.stats
import sys


def compute_snr(my_image, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = cv2.imread (my_image)
    im1 = im1.astype("float32")
    im2 = im1.copy ()

    im1 = numpy.add(im2, numpy.random.normal (mean, sd, im1.shape))
    im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)

    return snr



def noise_vs_corners(my_image, mean=0.0, sd=1.0, seed=None):

    # Function to add noise to the image,
    # and then get number of corners that are detected with this noise

    #ADD NOISE TO IMAGE

    im = cv2.imread(my_image)
    "convert to float for manipulation as OpenCV only works with UNINT8"
    im2 = im.astype("float64")

    if not seed is None: numpy.random.seed (seed)
    " add the noise "
    im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
    " change back to UINT8 for display with"
    im2 = im2.astype("uint8")

    #CALCULATE NUMBER OF CORNERS

    gray = cv2.cvtColor(im2,cv2.COLOR_BGR2GRAY)

    gray = numpy.float32(gray)
    dst = cv2.cornerHarris(gray,2,3,0.04)
    "number of corner edtections"
    num_corners = numpy.sum(dst > 0.01 * dst.max())

    return num_corners


# the following code is for graph creation and saving the result to file


SNR=[]
STD=[]
NO_OF_CORNERS=[]
temp=[]

# I add noise in percentage from 0% to 30% with 1% step.
```

```
# More than 30% noise I noticed is useless, and just shows that
# the corner detector is outputing data I cannot use.
for x in numpy.arange (0,0.3,0.01):

    # A few iterations as the same STD gives off a different number of corners
    # each time. I am taking the average of the result in order not to have
    # such a "pointy" graph
    for y in range (1,3):
        #data = noise_vs_corners(sys.argv[1], mean=0, sd=x*255)
        data = noise_vs_corners("ONE.JPG", mean=0, sd=x*255)
        temp.append(data)
    # calculate mean from the total number of iterations
    mean=sum(temp) / len (temp)
    temp=[]

    STD.append(x)  # the values here are in percentage
    # STD.append(x*255) # the values here are in STDs
    NO_OF_CORNERS.append(mean)

    # getting the SNR values
    value=compute_snr("ONE.JPG", mean=0, sd=x*255)
    SNR.append(value)



# code for graph generation and saving it to the file

import matplotlib.pyplot as plt

plt.plot(SNR,NO_OF_CORNERS)

plt.show()

#plt.savefig("SNR_CORNERS_"+"ONE.JPG"[:-4])

```

## Convert Color image to Grayscale

```
 1. import numpy
 2. import cv2
 3.
 4.
 5. filename = "ONE.JPG"
 6. img = cv2.imread(filename)
 7. gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
 8.
 9. # find Harris corners
10. gray = numpy.float32(gray)
11. dst = cv2.cornerHarris(gray,2,3,0.04)
12. ret, dst = cv2.threshold(dst,0.01*dst.max(),255,0)
13. dst = cv2.dilate(dst,None)
14.
15. dst = numpy.uint8(dst)
16. cv2.imshow("Corners", dst)
17.
18. # find centroids
19. ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
20. # define the criteria to stop and refine the corners
21.
22. criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)
23. corners = cv2.cornerSubPix(gray,numpy.float32(centroids),(5,5),(-1,-1),criteria)
24.
25. # Now draw them
26. res = numpy.hstack((centroids,corners))
27. res = numpy.int0(res)
28. img[res[:,1],res[:,0]]=[0,0,255] # make all pixels inside RES to be RED (BGR colorspace) to
signal CENTROIDS
29. img[res[:,3],res[:,2]] = [0,255,0] # make all pixels inside RED to be GREEN (BGR colorspace)
to signal CORNERS
30. #cv2.imwrite('subpixel5.png',img)
31.
32. # make red dots larger
33. #img[dst>0.01*dst.max()]=[0,0,255]
34.
35. noisy_image_display=filename[0:-4]+"_DISPLAY.jpg"
36. cv2.imwrite(noisy_image_display,img)
37. cv2.imshow('Corners Display',img)
38.
39. gray=numpy.uint8(gray)
40. cv2.imshow('GRAY Image',gray)
41.
42. if cv2.waitKey(0) & 0xff == 27:
43.     cv2.destroyAllWindows()
44.
45.
46.
47.
48.
49.
50.
51.
52. # find MAX in a 2D array and then find the 2D index where this is found
53.
54. #a = np.array([[200,300,400],[100,50,300]])
55. #indices = np.where(a == a.max())
56. #print(a[indices]) # prints [400]
57.
58.
```

## Visually display computer data onto generated image (manual)

```
import numpy
import cv2
import sys

def write_text_on_image(my_file, corners_detected):

    noisy_image_display_and_count=my_file[0:-4]+"_COUNT.jpg"


    # Create a black image
    #img = np.zeros((512,512,3), np.uint8)

    img = cv2.imread(my_file)
    x_coord=img.shape[0]
    y_coord=img.shape[1]
    bits_of_color=img.shape[2]



    # Write some Text

    font                   = cv2.FONT_HERSHEY_SIMPLEX
    bottomLeftCornerOfText = (10,y_coord-200)
    fontScale              = 1
    fontColor              = (255,255,255)
    thickness              = 3
    lineType               = 2

    percentage=float(Standard_Dev)/255*100
    percentage=float("{:.2f}".format(percentage))
    cv2.putText(img,'Corner count: '+str(corners_detected)+", "+str(percentage)+"% noise",
                bottomLeftCornerOfText,
                font,
                fontScale,
                fontColor,
                thickness,
                lineType)

    #Display the image
    cv2.imshow("Corner Display and Count",img)

    #Save image
    cv2.imwrite(noisy_image_display_and_count,img)

    #cv2.waitKey(0)

def add_gaussian_noise (im, mean=0.0, sd=1.0, seed=None):
    """
    Add Gaussian-distributed noise to each pixel of image `im`.

    Arguments:
      im  the image to which noise will be added (modified)
    mean  the mean of the Gaussian-distributed noise (default: 0.0)
      sd  the standard deviation of the noise (default: 1.0)
    seed  if supplied, the value used to seed the random number generator
    """
    image_string=im
    window_name="Original Image"
    im = cv2.imread(im)

    "convert to float for manipulation as OpenCV only works with UNINT8"
    window_name2="Image with Noise"
    im2 = im.astype("float64")

    "im = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)"
```

```
66.
67.     if not seed is None: numpy.random.seed (seed)
68.     " add the noise "
69.     im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
70.
71.     " produce a red image by writing the BG color data to 0  (Image is read as BGR)"
72.     #im2[:,:,(0,1)]=0
73.
74.     " change back to UINT8 for display with"
75.     im2 = im2.astype("uint8")
76.
77.     " write noisy image to file"
78.     cv2.imwrite(image_string[0:-4]+"_NOISY.jpg",im2)
79.
80.     cv2.imshow(window_name2,im2)
81.     cv2.imshow(window_name,im)
82.     #cv2.waitKey (0)
83.
84.     filename_string=image_string[0:-4]+"_NOISY.jpg"
85.
86.
87.     return filename_string
88.
89.
90. def corn_det_visual_and_numeric(my_image):
91.     filename = my_image
92.     img = cv2.imread(filename)
93.     gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
94.
95.     gray = numpy.float32(gray)
96.     dst = cv2.cornerHarris(gray,2,3,0.04)
97.     # number of corner detections
98.     num_corners = numpy.sum(dst > 0.01 * dst.max())
99.
100.    #result is dilated for marking the corners, not important
101.    dst = cv2.dilate(dst,None)
102.
103.    # Threshold for an optimal value, it may vary depending on the image.
104.    img[dst>0.01*dst.max()]=[0,0,255]
105.
106.    noisy_image_display=my_image[0:-4]+"_DISPLAY.jpg"
107.
108.    # write noisy cornered image to file
109.
110.    cv2.imwrite(noisy_image_display,img)
111.    # window title
112.    cv2.imshow('Corners Display',img)
113.
114.    #print(num_corners)
115.    return num_corners, noisy_image_display
116.
117.
118.
119. ##############################   MAIN   ###################################
120.
121. # This code will generate 2 images. One image has "NOISY" appended to indicate
122. # how the image looks like with all the corner detectors on it. The other image
123. # has "COUNT" appended to it to show the percentage of noise and the total number
124. # of corners detected.
125.
126. # here I add noise in the amound of 9 STDs. This would be 9/255 = 3.5% from the grey levels
of the image
127. # I can change the noise as I please to see how it visually affects the image.
128. Standard_Dev=9
129.
130. new_file=add_gaussian_noise(sys.argv[1],0,Standard_Dev)
131. no_of_corners, corner_display_image = corn_det_visual_and_numeric(new_file)
132.
133. write_text_on_image(corner_display_image, no_of_corners)
134.
```

```
135. if cv2.waitKey(0) & 0xff == 27:
136.         cv2.destroyAllWindows()
137.
138.
```

## Visually display computer data onto generated image (AUTO)

```
import numpy
import cv2
import sys

def write_text_on_image(my_file, corners_detected):

    noisy_image_display_and_count=my_file[0:-4]+"_COUNT.jpg"


    # Create a black image
    #img = np.zeros((512,512,3), np.uint8)

    img = cv2.imread(my_file)
    x_coord=img.shape[0]
    y_coord=img.shape[1]
    bits_of_color=img.shape[2]



    # Write some Text

    font                   = cv2.FONT_HERSHEY_SIMPLEX
    bottomLeftCornerOfText = (10,y_coord-200)
    fontScale              = 1
    fontColor              = (255,255,255)
    thickness              = 3
    lineType               = 2

    percentage=float(Standard_Dev)/255*100
    percentage=float("{:.2f}".format(percentage))
    cv2.putText(img,'Corner count: '+str(corners_detected)+", "+str(percentage)+"% noise",
                bottomLeftCornerOfText,
                font,
                fontScale,
                fontColor,
                thickness,
                lineType)

    #Display the image
    cv2.imshow("Corner Display and Count",img)

    #Save image
    cv2.imwrite(noisy_image_display_and_count,img)

    #cv2.waitKey(0)

def add_gaussian_noise (im, mean=0.0, sd=1.0, seed=None):
    """
    Add Gaussian-distributed noise to each pixel of image `im`.

    Arguments:
      im  the image to which noise will be added (modified)
    mean  the mean of the Gaussian-distributed noise (default: 0.0)
      sd  the standard deviation of the noise (default: 1.0)
    seed  if supplied, the value used to seed the random number generator
    """
    image_string=im
    window_name="Original Image"
    im = cv2.imread(im)

    "convert to float for manipulation as OpenCV only works with UNINT8"
    window_name2="Image with Noise"
    im2 = im.astype("float64")

    "im = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)"
```

```
66.
67.     if not seed is None: numpy.random.seed (seed)
68.     " add the noise "
69.     im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
70.
71.     " produce a red image by writing the BG color data to 0  (Image is read as BGR)"
72.     #im2[:,:,(0,1)]=0
73.
74.     " change back to UINT8 for display with"
75.     im2 = im2.astype("uint8")
76.
77.     " write noisy image to file"
78.     cv2.imwrite(image_string[0:-4]+"_NOISY.jpg",im2)
79.
80.     cv2.imshow(window_name2,im2)
81.     cv2.imshow(window_name,im)
82.     #cv2.waitKey (0)
83.
84.     filename_string=image_string[0:-4]+"_NOISY.jpg"
85.
86.
87.     return filename_string
88.
89.
90. def corn_det_visual_and_numeric(my_image):
91.     filename = my_image
92.     img = cv2.imread(filename)
93.     gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
94.
95.     gray = numpy.float32(gray)
96.     dst = cv2.cornerHarris(gray,2,3,0.04)
97.     # number of corner detections
98.     num_corners = numpy.sum(dst > 0.01 * dst.max())
99.
100.    #result is dilated for marking the corners, not important
101.    dst = cv2.dilate(dst,None)
102.
103.    # Threshold for an optimal value, it may vary depending on the image.
104.    img[dst>0.01*dst.max()]=[0,0,255]
105.
106.    noisy_image_display=my_image[0:-4]+"_DISPLAY.jpg"
107.
108.    # write noisy cornered image to file
109.
110.    cv2.imwrite(noisy_image_display,img)
111.    # window title
112.    cv2.imshow('Corners Display',img)
113.
114.    #print(num_corners)
115.    return num_corners, noisy_image_display
116.
117.
118.
119. ##############################   MAIN   ###################################
120.
121. # This code will generate 2 images. One image has "NOISY" appended to indicate
122. # how the image looks like with all the corner detectors on it. The other image
123. # has "COUNT" appended to it to show the percentage of noise and the total number
124. # of corners detected.
125.
126. # here I add noise in the amound of 9 STDs. This would be 9/255 = 3.5% from the grey levels
of the image
127. # I can change the noise as I please to see how it visually affects the image.
128. Standard_Dev=9
129.
130. new_file=add_gaussian_noise("ONE.JPG",0,Standard_Dev)
131. no_of_corners, corner_display_image = corn_det_visual_and_numeric(new_file)
132.
133. write_text_on_image(corner_display_image, no_of_corners)
134.
```

```
135. if cv2.waitKey(0) & 0xff == 27:
136.         cv2.destroyAllWindows()
137.
```

## Adding noise incrementally and automatically generating images (Version 1)

```
import numpy
import cv2
import scipy
import math
import scipy.stats
import sys


def compute_snr(my_image, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = cv2.imread (my_image)
    im1 = im1.astype("float32")
    im2 = im1.copy ()

    im1 = numpy.add(im2, numpy.random.normal (mean, sd, im1.shape))
    im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)

    return snr



def noise_vs_corners(my_image, mean=0.0, sd=1.0, seed=None):

    # Function to add noise to the image,
    # and then get number of corners that are detected with this noise

    #ADD NOISE TO IMAGE

    im = cv2.imread(my_image)
    "convert to float for manipulation as OpenCV only works with UNINT8"
    im2 = im.astype("float64")

    if not seed is None: numpy.random.seed (seed)
    " add the noise "
    im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
    " change back to UINT8 for display with"
    im2 = im2.astype("uint8")

    #CALCULATE NUMBER OF CORNERS

    gray = cv2.cvtColor(im2,cv2.COLOR_BGR2GRAY)

    gray = numpy.float32(gray)
    dst = cv2.cornerHarris(gray,2,3,0.04)
    "number of corner edtections"
    num_corners = numpy.sum(dst > 0.01 * dst.max())

    return num_corners


# the following code is for graph creation and saving the result to file


SNR=[]
STD=[]
NO_OF_CORNERS=[]
temp=[]

# I add noise in percentage from 0% to 30% with 1% step.
```

```
66. # More than 30% noise I noticed is useless, and just shows that
67. # the corner detector is outputing data I cannot use.
68. for x in numpy.arange (0,0.3,0.01):
69. 
70.     # A few iterations as the same STD gives off a different number of corners
71.     # each time. I am taking the average of the result in order not to have
72.     # such a "pointy" graph
73.     for y in range (1,3):
74.         data = noise_vs_corners(sys.argv[1], mean=0, sd=x*255)
75.         temp.append(data)
76.     # calculate mean from the total number of iterations
77.     mean=sum(temp) / len (temp)
78.     temp=[]
79. 
80.     STD.append(x)  # the values here are in percentage
81.     # STD.append(x*255) # the values here are in STDs
82.     NO_OF_CORNERS.append(mean)
83. 
84.     # getting the SNR values
85.     value=compute_snr(sys.argv[1], mean=0, sd=x*255)
86.     SNR.append(value)
87. 
88. 
89. 
90. # code for graph generation and saving it to the file
91. 
92. import matplotlib.pyplot as plt
93. 
94. plt.plot(SNR,NO_OF_CORNERS)
95. 
96. #plt.show()
97. 
98. plt.savefig("SNR_CORNERS_"+sys.argv[1][:-4])
99. 
```

## Adding noise incrementally and automatically generating images (Version 1)

```
import numpy
import cv2
import sys

def write_text_on_image(my_file, corners_detected):

    noisy_image_display_and_count=my_file[0:-4]+"_COUNT.jpg"


    # Create a black image
    #img = np.zeros((512,512,3), np.uint8)

    img = cv2.imread(my_file)
    x_coord=img.shape[0]
    y_coord=img.shape[1]
    bits_of_color=img.shape[2]



    # Write some Text

    font                   = cv2.FONT_HERSHEY_SIMPLEX
    bottomLeftCornerOfText = (10,y_coord-200)
    fontScale              = 1
    fontColor              = (255,255,255)
    thickness              = 3
    lineType               = 2

    percentage=float(Standard_Dev)/255*100
    percentage=float("{:.2f}".format(percentage))
    cv2.putText(img,'Corner count: '+str(corners_detected)+", "+str(percentage)+"% noise",
                bottomLeftCornerOfText,
                font,
                fontScale,
                fontColor,
                thickness,
                lineType)

    #Display the image
    cv2.imshow("Corner Display and Count",img)

    #Save image
    cv2.imwrite(noisy_image_display_and_count,img)

    #cv2.waitKey(0)

def add_gaussian_noise (im, mean=0.0, sd=1.0, seed=None):
    """
    Add Gaussian-distributed noise to each pixel of image `im`.

    Arguments:
      im  the image to which noise will be added (modified)
    mean  the mean of the Gaussian-distributed noise (default: 0.0)
      sd  the standard deviation of the noise (default: 1.0)
    seed  if supplied, the value used to seed the random number generator
    """
    image_string=im
    window_name="Original Image"
    im = cv2.imread(im)

    "convert to float for manipulation as OpenCV only works with UNINT8"
    window_name2="Image with Noise"
    im2 = im.astype("float64")

    "im = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)"
```

```
66.
67.     if not seed is None: numpy.random.seed (seed)
68.     " add the noise "
69.     im2 = numpy.add(im2, numpy.random.normal (mean, sd, im2.shape))
70.
71.     " produce a red image by writing the BG color data to 0  (Image is read as BGR)"
72.     #im2[:,:,(0,1)]=0
73.
74.     " change back to UINT8 for display with"
75.     im2 = im2.astype("uint8")
76.
77.     " write noisy image to file"
78.     cv2.imwrite(image_string[0:-4]+"_NOISY.jpg",im2)
79.
80.     cv2.imshow(window_name2,im2)
81.     cv2.imshow(window_name,im)
82.     #cv2.waitKey (0)
83.
84.     filename_string=image_string[0:-4]+"_NOISY.jpg"
85.
86.
87.     return filename_string
88.
89.
90. def corn_det_visual_and_numeric(my_image):
91.     filename = my_image
92.     img = cv2.imread(filename)
93.     gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
94.
95.     gray = numpy.float32(gray)
96.     dst = cv2.cornerHarris(gray,2,3,0.04)
97.     # number of corner edtections
98.     num_corners = numpy.sum(dst > 0.01 * dst.max())
99.
100.    #result is dilated for marking the corners, not important
101.    dst = cv2.dilate(dst,None)
102.
103.    # Threshold for an optimal value, it may vary depending on the image.
104.    img[dst>0.01*dst.max()]=[0,0,255]
105.
106.    noisy_image_display=my_image[0:-4]+"_DISPLAY.jpg"
107.
108.    # write noisy cornered image to file
109.
110.    cv2.imwrite(noisy_image_display,img)
111.    # window title
112.    cv2.imshow('Corners Display',img)
113.
114.    #print(num_corners)
115.    return num_corners, noisy_image_display
116.
117.
118.
119. ##############################   MAIN   ###################################
120.
121. # This code will generate 2 images. One image has "NOISY" appended to indicate
122. # how the image looks like with all the corner detectors on it. The other image
123. # has "COUNT" appended to it to show the percentage of noise and the total number
124. # of corners detected.
125.
126. # here I add noise in the amound of 9 STDs. This would be 9/255 = 3.5% from the grey levels
of the image
127. # I can change the noise as I please to see how it visually affects the image.
128. Standard_Dev=9
129.
130. new_file=add_gaussian_noise(sys.argv[1],0,Standard_Dev)
131. no_of_corners, corner_display_image = corn_det_visual_and_numeric(new_file)
132.
133. write_text_on_image(corner_display_image, no_of_corners)
134.
```

```
135. if cv2.waitKey(0) & 0xff == 27:
136.         cv2.destroyAllWindows()
137.
```

## Routine for automating POV-Ray virtual image generation

```
 1. import subprocess
 2. import sys
 3.
 4. IS_WIN32 = 'win32' in str(sys.platform).lower()
 5.
 6.
 7. def subprocess_call(*args, **kwargs):
 8.     #also works for Popen. It creates a new *hidden* window, so it will work in frozen apps
(.exe).
 9.     if IS_WIN32:
10.         startupinfo = subprocess.STARTUPINFO()
11.         startupinfo.dwFlags = subprocess.CREATE_NEW_CONSOLE |
subprocess.STARTF_USESHOWWINDOW
12.         startupinfo.wShowWindow = subprocess.SW_HIDE
13.         kwargs['startupinfo'] = startupinfo
14.     retcode = subprocess.call(*args, **kwargs)
15.     return retcode
16.
17.
18. for i in range(0,256,1):
19.
20.     RGB_Counter=i/float(255) #pov-ray knows colours from 0 to 1
21.
22.     with open("box.pov","r") as f, open("box.tmp", "w") as g:
23.         for line_content in f:
24.             if ("#declare i=") in line_content:
25.                 g.write("#declare i="+str(round(RGB_Counter,4))+";\n")
26.
27.             else:
28.                 g.write(line_content)
29.
30.
31.     subprocess_call(["c:\\Program Files\\POV-Ray\\v3.7\\bin\\pvengine64.exe",\
32.                      "/EXIT",\
33.                      "/RENDER",\
34.                      "-W800",\
35.                      "-H600",\
36.                      "-d",\
37.                      "-Ic:\\OKLLAK\\GENERATE\\box.tmp",\
38.                      "-Oc:\\OKLLAK\\GENERATE\\Picture"+str(i)+".png"])
39.
```

## Generating Contrast VS Number of corners thesis figures

```
 1. import cv2
 2. import numpy
 3.
 4. def corners_vs_constrast(my_image, mean=0.0, sd=1.0, seed=None):
 5.
 6.     #ADD NOISE TO IMAGE
 7.     im = cv2.imread(my_image)
 8.     #CALCULATE NUMBER OF CORNERS
 9.
10.     gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
11.     gray = numpy.float32(gray) # convert to float for data manipulation
12.     contrast=int(gray.max()-gray.min())
13.     if not seed is None: numpy.random.seed (seed)
14.     " add the noise "
15.     gray_and_noisy = numpy.add(gray, numpy.random.normal (mean, sd, gray.shape))
16.     gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32
17.
18.     dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)
19.
20.     treshold_value=0.01 * dst.max() #getting a treshold value
21.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
22.
23.     dst = numpy.uint8(dst) # the command below needs integer numbers
24.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
25.     num_corners = labels.max() # corners being detected
26.
27.     return num_corners, contrast
28.
29.
30.
31.
32. CONTRAST=[]
33. NO_OF_CORNERS=[]
34. Stand_Dev=10
35.
36.
37. for i in range(0,256,1):
38.     filename="Picture"+str(i)+".png"
39.     corn, cont=corners_vs_constrast(filename, mean=0,sd=Stand_Dev)
40.     NO_OF_CORNERS.append(corn)
41.     CONTRAST.append(cont)
42.     print(filename)
43.
44.
45. import matplotlib.pyplot as plt
46.
47.
48. plt.plot(CONTRAST,NO_OF_CORNERS)
49. plt.title("CONTRAST against NUMBER OF CORNERS")
50. plt.xlabel("Contrast between BACK and FORE")
51. plt.ylabel("Number of corners detected")
52.
53. plt.show()
54.
```

## Returning the SNR between 2 images

```
def add_gaussian_noise (im, mean=0.0, sd=1.0, seed=None):
    """
    Add Gaussian-distributed noise to each pixel of image `im`.

    Arguments:
      im  the image to which noise will be added (modified)
    mean  the mean of the Gaussian-distributed noise (default: 0.0)
      sd  the standard deviation of the noise (default: 1.0)
    seed  if supplied, the value used to seed the random number generator
    """
    if not seed is None: numpy.random.seed (seed)
    im += numpy.random.normal (mean, sd, im.shape)

def correlation_slow (im1, im2):
    """
    Return the correlation coefficient between two images.

    Arguments:
    im1  image to be correlated with im2
    im2  image to be correlated with im1
    """
    ny, nx, nc = im1.shape
    sumx = sumy = sumxx = sumyy = sumxy = 0.0
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v1 = im1[y,x,c]
                v2 = im2[y,x,c]
                sumx += v1
                sumy += v2
                sumxx += v1 * v1
                sumxy += v1 * v2
                sumyy += v2 * v2
    n = ny * nx * nc
    v1 = sumxy - sumx * sumy / n
    v2 = math.sqrt((sumxx-sumx*sumx/n) * (sumyy-sumy*sumy/n))
    return v1 / v2

def correlation (p, q):
    "Calculate the correlation coefficient between two sets of values."
    return scipy.stats.pearsonr (p.ravel(), q.ravel())[0]

def snr (im1, im2):
    """
    Return the signal-to-noise ratio between two images.

    Arguments:
    im1  first image to be used in calculating the SNR
    im2  first image to be used in calculating the SNR
    """
    r = correlation (im1, im2)
    if r <= 0.0: return 0.0
    if r >= 1.0: return math.inf
    return math.sqrt (r / (1.0 - r))

```

## Routine to generate figures relating to contrast and number of corners

```
# to see number of corners and the contrast


import cv2
import numpy
import scipy.stats
import math

def corners_vs_constrast_vs_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):

    #ADD NOISE TO IMAGE
    im = cv2.imread(my_image)
    #CALCULATE NUMBER OF CORNERS

    sd=im.std(ddof=0)
    percentage_of_noise_added=sd*factor/100.0

    gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
    gray = numpy.float32(gray) # convert to float for data manipulation
    contrast=int(gray.max()-gray.min())
    if not seed is None: numpy.random.seed (seed)
    " add the noise and also compute the SNR "
    gray_and_noisy = numpy.add(gray, numpy.random.normal (mean, percentage_of_noise_added,
gray.shape))
    gray_and_noisy2 = numpy.add(gray, numpy.random.normal (mean, percentage_of_noise_added,
gray.shape))
    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (gray_and_noisy.ravel(), gray_and_noisy2.ravel())[0]
    snr = math.sqrt (r / (1 - r))

    gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32

    dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)

    treshold_value=0.01 * dst.max() #getting a treshold value
    tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)

    dst = numpy.uint8(dst) # the command below needs integer numbers
    ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
    num_corners = labels.max() # corners being detected

    return num_corners, snr




################################  MAIN  ###################################


factor=30
CONTRAST=[]
NO_OF_CORNERS=[]
SNR=[]
PERCENTAGE_NOISE=[]
i=1

filename="Picture"+str(i)+".png"


for x in numpy.arange (0.1,factor,0.1):
    corner_count, snr_no = corners_vs_constrast_vs_snr(filename, factor=x, mean=0.0, sd=1.0)
    SNR.append(snr_no)
    NO_OF_CORNERS.append(corner_count)
    PERCENTAGE_NOISE.append(round(x,1))
```

```
64.     print("PERCENTAGE NOISE: ", str(round(x,1))+" %, and SNR: "+str(round(snr_no,1))+",
image: "+str(i))
65.
66.
67.
68.
69. import matplotlib.pyplot as plt
70.
71.
72. #plt.plot(PERCENTAGE_NOISE,NO_OF_CORNERS)
73. #plt.title("NOISE vs CORNERS")
74. #plt.xlabel("NOISE")
75. #plt.ylabel("CORNERS detected")
76. #
77. #plt.show()
78.
79.
80. plt.plot(SNR,NO_OF_CORNERS)
81. plt.title("SNR vs CORNERS")
82. plt.xlabel("SNR")
83. plt.ylabel("CORNERS detected")
84.
85. plt.show()
86.
87.
```

## Code to prove how SNR works

```
 1. # code for ma to understand better how SNR works on smaller data "my_signal"
 2.
 3.
 4. import numpy as np
 5. import numpy
 6. import scipy.stats
 7. import math
 8.
 9.
10. my_signal=[1,2,3,4,5,6,7]
11. my_signal = np.asanyarray(my_signal)
12. # now calculate the  Standard Deviation of the signal
13. sd=my_signal.std(ddof=0)
14.
15. # we will apply 2 times a random noise (the quantity of this noise is a percentage or factor
16. # from the standard deviation of the original signal) to the original signal.
17. # This way we will have 2 noisy signals where we will compute their SNR
18.
19. # this is the percentage of the noise I will want to apply to my signal. This noise
20. # is a percentage of the Standard Deviation of the Signal.
21. factor=5
22. percentage_of_noise_added=sd*factor/100.0
23.
24. # adding noise to my image, 2 times
25. noisy_image1 = numpy.add(my_signal, numpy.random.normal (0, percentage_of_noise_added,
my_signal.shape))
26. noisy_image2 = numpy.add(my_signal, numpy.random.normal (0, percentage_of_noise_added,
my_signal.shape))
27.
28.
29. #np.corrcoef(noisy_image1,noisy_image2)
30.
31. r = scipy.stats.pearsonr (noisy_image1,noisy_image2)[0]
32. snr = math.sqrt (r / (1 - r))
33.
34.
35. print("IMAGE: ", my_signal)
36. print("IMAGE with NOISE: ", noisy_image1)
37. print("CORRELATION: ",r)
38. print("SNR: ",snr)
39.
```

**Code that generates virtual images with the same SNR as the SNR present in real images. This is done in order to show how well the corner detector works (numeric – optimsied for speed).**

```
# generate virtual image with the same SNR as the real image and then see if the corner detector works the same on
# virtual image as it works on the real image

import numpy
import cv2
import scipy.stats
import math
import sys

def return_virtual_generated_image_based_on_real_image(REAL_Image):
    # I will calcuate the background and foreground of the image in order to generate an
    # artificial image with the same background and foreground shades of gray as the real image

    # the size (in pixels) of the side of a square in the center of the screen where I will sample each
    # pixel in the real image for the shade of gray required for the foreground of
    # of the virtual image. The the mean value will be the shade of gray used to build
    # the foreground in the virtual image
    gray1=REAL_Image
    square_side=20
    all_pixels_from_square=[]
    for i in range (gray1.shape[0]/2-square_side/2, gray1.shape[0]/2+square_side/2,1):
        for j in range (gray1.shape[1]/2-square_side/2, gray1.shape[1]/2+square_side/2,1):
            all_pixels_from_square.append(gray1[i,j])
    fore_ground=int(numpy.mean(all_pixels_from_square))

    # the side of the sqare starting with topmost and leftomost pixel of the
    # real image in order to detect tge bacground shade of gray that will be applied
    # to the background of the virtual image
    all_pixels_from_square=[]
    for i in range (0, square_side,1):
        for j in range (0, square_side,1):
            all_pixels_from_square.append(gray1[i,j])
    back_ground=int(numpy.mean(all_pixels_from_square))

    # generate virtual image to resemble the real image
    gray_generated=numpy.zeros((len(gray1),len(gray1[0])))
    gray_generated[:,:]=back_ground  # apply backround share of gray
    gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure

    # ratio of how far the white sqare is from the boundaries of the screen.
    # the ratio is both for the X and for the Y coordinates. How I did it? I took
    # an example and then I calculated the ratio between the maximum X resolution
    # and how many pixles away was the cube from 0. In our case it was 600
    # pixels away. So the X resolution (4288 pixes) divied by 600 gives us a ratio
    # of 7.14 This means that for any resolution picture if I keep the same ratio, the square
    # in the middle of the picture real picture will sit almost in the same position in the virtual image.
    ratio_X=7.1466666666666665
    ratio_Y=4.746666666666667
    Y_value=int(len(gray_generated)/ratio_Y)
    X_value=int(len(gray_generated[0])/ratio_X)

    # create the foreground (a white square) in the middle of the image (background)
    gray_generated[len(gray_generated)/2-X_value:len(gray_generated)/2+X_value,\
```

```
 54.                     len(gray_generated[0])/2-
Y_value:len(gray_generated[0])/2+Y_value]=fore_ground  #after this command
 55.     # we have our image with a cube in the middle generated, just like the real image.
 56.     # Now we will return this virtual image.
 57.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
 58.     return gray_generated
 59.
 60.
 61.
 62. def no_of_corners(my_image):
 63.
 64.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
 65.
 66.     dst = cv2.cornerHarris(gray,2,3,0.04)
 67.
 68.     treshold_value=0.01 * dst.max() #getting a treshold value
 69.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
 70.
 71.     dst = numpy.uint8(dst) # the command below needs integer numbers
 72.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 73.     num_corners = labels.max() # corners being detected
 74.
 75.     return num_corners
 76.
 77.
 78. def no_of_corners_after_applied_noise(my_image, factor, mean=0.0, sd=1.0, seed=None):
 79.
 80.     gray = numpy.float32(my_image) # convert to float for data manipulation
 81.     if not seed is None: numpy.random.seed (seed)
 82.     " add the noise "
 83.
 84.     sd=gray.std(ddof=0)
 85.     percentage_of_noise_added=sd*factor/100.0
 86.
 87.     gray_and_noisy = numpy.add(gray, numpy.random.normal (mean, percentage_of_noise_added,
gray.shape))
 88.     gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32
 89.
 90.     dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)
 91.
 92.     treshold_value=0.01 * dst.max() #getting a treshold value
 93.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
 94.
 95.     dst = numpy.uint8(dst) # the command below needs integer numbers
 96.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 97.     num_corners = labels.max() # corners being detected
 98.
 99.     return num_corners
100.
101.
102.
103. def compute_virtual_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
104.     # Read in the image and convert to float.  Duplicate it and add noise.
105.     im1 = my_image
106.     im1 = im1.astype("float32")
107.     im2 = im1.copy()
108.
109.     sd=im1.std(ddof=0)
110.     percentage_of_noise_added=sd*factor/100.0
111.
112.     im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added, im1.shape))
113.     im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added, im2.shape))
114.
115.     # Compute the correlation coefficient and SNR.
116.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
117.     #print ("r:", r)
118.     snr = math.sqrt (r / (1 - r))
119.     #print ("SNR:", snr)
120.
```

```
121.     return snr
122.
123.
124. def compute_snr_real(my_image1, my_image2):
125.
126.     im1 = my_image1.astype("float32")
127.     im2 = my_image2.astype("float32")
128.
129.     # Compute the correlation coefficient and SNR.
130.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
131.     #print ("r:", r)
132.     snr = math.sqrt (r / (1 - r))
133.     #print ("SNR:", snr)
134.
135.     return snr
136.
137.
138. ##################################################  MAIN
##################################################
139. # read real images
140. FILE1=sys.argv[1]
141. FILE2=sys.argv[2]
142. # read real images
143.
144. gray1=cv2.imread(FILE1, 0) # read the image directly to grayscale
145. gray2=cv2.imread(FILE2, 0) # read the image directly to grayscale
146. SNR_REAL = compute_snr_real(gray1, gray2) # compute the SNR of the real images
147.
148. # calculate the noise needed to apply to the virtual image so that it will have
149. # the same SNR as the real image
150. noise_applied=100.0/SNR_REAL
151.
152. # generate the virtual image based on the real image
153. virtual_generated_image=return_virtual_generated_image_based_on_real_image(gray1)
154. SNR = compute_virtual_snr(virtual_generated_image, noise_applied) # SNR of the virtual
image (for verification)
155.
156. ############################## VIRTUAL IMAGE ####################
157. # get the number of corners for the virtual generated image. Since this image
158. # doesn't have any noise, and given that we have square in the middle of the image,
159. # the corners returned will be 4. This is how we know how many true corners
160. # should be in the image.
161. corners = no_of_corners(virtual_generated_image)
162. # get number of corners after noise has been applied to the virtually generated image
163. corners_after_noise = no_of_corners_after_applied_noise(virtual_generated_image,
noise_applied)
164.
165.
166. ################################## REAL IMAGE #################
167. REAL_corners = no_of_corners(gray1)
168. round(SNR_REAL,3), round(SNR,3), REAL_corners, round(noise_applied,3), corners_after_noise,
corners
169.
170.
171. print("Image pair: ["+str(FILE1)+", "+str(FILE2)+"]")
172. print("One of the real images has: "+str(REAL_corners)+" corners")
173. print("-------------------------------------------")
174. print("A noise of "+str(round(noise_applied,3))+"% has been appplied to Virt Img to get an
SNR: "+str(round(SNR,3)))
175. print("Virt Img with NO NOISE has: "+str(corners)+" corners")
176. print("Virt Img with NOISE has: "+str(corners_after_noise)+" corners\n")
177.
```

**Code that generates virtual images with the same SNR as the SNR present in real images. This is done in order to show how well the corner detector works (visual – slow to compute)**

```
  1. # generate virtual image with the same SNR as the real image and then see if the corner
detector works the same on
  2. # virtual image as it works on the real image
  3.
  4. # There is an issue to how I generate my virtual image. When images are not in landscape
format
  5. # (but in portrait mode) the image generation is distorted. I need to find another way to
generate my images
  6.
  7. # Also the way I deal with image normalisation needs to change. I can use CLIP,
  8. # as it was found in "TEST_NOISE_APPLICATION.PY" that it gives better corner detections
  9. # when the image contains originally values of ZERO as the backgtound shade of gray.
 10. # For any other shades of gray both clip and my method return the same corner detections.
 11. # But for the simplicity of the code I will use CLIP
 12.
 13. # The idea was also that I apply some variation to the background and foreground,
 14. # as currently there is no variation, just to make my virtual image resemble the real image
better
 15. # Adrian said I can apply a sinusoidal output to this varation, and then next thing would
be
 16. # to apply random variation.
 17. #
 18. # He then mentioned something about detecling the corner in a strange way.
 19. # Just listen to the audio file for more detail.
 20.
 21.
 22. import numpy
 23. import cv2
 24. import scipy.stats
 25. import math
 26. import sys
 27.
 28.
 29.
 30. def no_of_corners(my_image):
 31.
 32.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
 33.
 34.     dst = cv2.cornerHarris(gray,2,3,0.04)
 35.
 36.     treshold_value=0.01 * dst.max() #getting a treshold value
 37.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
 38.
 39.     dst = numpy.uint8(dst) # the command below needs integer numbers
 40.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 41.     num_corners = labels.max() # corners being detected
 42.
 43.     #result is dilated for marking the corners, not important
 44.     dst = cv2.dilate(dst,None)
 45.     # place gray dots on the image to show where the corners are
 46.     gray[dst==255]=[137]
 47.
 48.     gray = numpy.uint8(gray)
 49.     return num_corners, gray
 50.
 51.
 52. def no_of_corners_after_applied_noise(my_image, factor, mean=0.0, sd=1.0, seed=None):
 53.
```

```
    # this is for being able to display the corners in red on the noisyt image
    #clone=numpy.zeros((len(my_image),len(my_image[0]),3))

    gray = numpy.float32(my_image) # convert to float for data manipulation
    if not seed is None: numpy.random.seed (seed)
    " add the noise "

    sd=gray.std(ddof=0)
    percentage_of_noise_added=sd*factor/100.0

    gray_and_noisy = numpy.add(gray, numpy.random.normal (mean, percentage_of_noise_added,
gray.shape))
    gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32

    dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)

    treshold_value=0.01 * dst.max() #getting a treshold value
    tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)

    dst = numpy.uint8(dst) # the command below needs integer numbers
    ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
    num_corners = labels.max() # corners being detected


    # If the image has a background of 0, the application of random noise will
    # give some of the pixels a negative float value (the noise is + or - a random value).
    # This is problematic when converting to UINT8, as it seems that when a negative float
    # nuber is converted to UINT8, the operation performed is 256 - abs (that negative
float number).
    # This means that an original pixel value of 0, to which a noise of -1.5 was applied,
    # upon conversion to UINT8 will have a value of 256 - (1.5)=253.5 So from a pixel value
of 0
    # we've ended at the opposite spectrum with a value of 254.5.
    # To fix this, we convert every negative pixel value from the image into the same
value, but positive.
    # This may change the noise normal distribution, but I hope that it is not by much
.

    # So we do not allow any negative nubmers in the image,
    # (otherwise we will not be able to display properly)
    gray_and_noisy_for_display = numpy.absolute(gray_and_noisy)

    # The same problem appears when with the addition of noise, the pixel value
    # is greater than 255. A conversion to UINT8 will yiels a number following
    # this formula "that number larger than 255)-255". This will return a value
    # closer to 0 than the original pixel value.
    # The idea is to limit numbers larger than 255 to 255. This is achieved by
    # the code below.
    for i in range(0,gray_and_noisy_for_display.shape[0],1):
        for j in range(0,gray_and_noisy_for_display.shape[1],1):
            if gray_and_noisy_for_display[i][j]>255:
                temp=gray_and_noisy_for_display[i][j]-255  #eg. 256.345-255= 1.34500
                gray_and_noisy_for_display[i][j]=255-temp
    # at this point our image "gray_and_noisy" has float values but all of them
    # are positive numbers and none of them is larger than 255

    #result is dilated for marking the corners, not important
    dst = cv2.dilate(dst,None)

    # place gray dots on the image to show where the corners are
    gray_and_noisy_for_display[dst==255]=[137]

    gray_and_noisy_for_display = numpy.uint8(gray_and_noisy_for_display)

    return num_corners, gray_and_noisy_for_display


def compute_virtual_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
```

```
119.     im1 = my_image
120.     im1 = im1.astype("float32")
121.     im2 = im1.copy()
122.
123.     sd=im1.std(ddof=0)
124.     percentage_of_noise_added=sd*factor/100.0
125.
126.    # percentage_of_noise_added=5
127.
128.     im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added, im1.shape))
129.     im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added, im2.shape))
130.
131.     # Compute the correlation coefficient and SNR.
132.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
133.     #print ("r:", r)
134.     snr = math.sqrt (r / (1 - r))
135.     #print ("SNR:", snr)
136.
137.     return snr
138.
139.
140.
141. def compute_snr_real(my_image1, my_image2):
142.
143.     im1 = my_image1.astype("float32")
144.     im2 = my_image2.astype("float32")
145.
146.     # Compute the correlation coefficient and SNR.
147.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
148.     #print ("r:", r)
149.     snr = math.sqrt (r / (1 - r))
150.     #print ("SNR:", snr)
151.
152.     return snr
153.
154.
155. def return_virtual_generated_image_based_on_real_image(REAL_Image):
156.     # I will calcuate the background and foreground of the image in order to generate an
157.     # artificial image with the same background and foreground shades of gray as the real
image
158.
159.     # the size (in pixels) of the side of a square in the center of the screen where I will
sample each
160.     # pixel in the real image for the shade of gray required for the foreground of
161.     # of the virtual image. The the mean value will be the shade of gray used to build
162.     # the foreground in the virtual image
163.     gray1=REAL_Image
164.     square_side=20
165.     all_pixels_from_square=[]
166.     for i in range (gray1.shape[0]//2-square_side//2, gray1.shape[0]//2+square_side//2,1):
167.         for j in range (gray1.shape[1]//2-square_side//2,
gray1.shape[1]//2+square_side//2,1):
168.             all_pixels_from_square.append(gray1[i,j])
169.     fore_ground=int(numpy.mean(all_pixels_from_square))
170.
171.     # the side of the sqare starting with topmost and leftomost pixel of the
172.     # real image in order to detect tge bacground shade of gray that will be applied
173.     # to the background of the virtual image
174.     all_pixels_from_square=[]
175.     for i in range (0, square_side,1):
176.         for j in range (0, square_side,1):
177.             all_pixels_from_square.append(gray1[i,j])
178.     back_ground=int(numpy.mean(all_pixels_from_square))
179.
180.     # generate virtual image to resemble the real image
181.     gray_generated=numpy.zeros((len(gray1),len(gray1[0])))
182.     gray_generated[:,:]=back_ground  # apply backround share of gray
183.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
184.
185.     # ratio of how far the white sqare is from the boundaries of the screen.
```

```
186.     # the ratio is both for the X and for the Y coordinates. How I did it? I took
187.     # an example and then I calculated the ratio between the maximum X resolution
188.     # and how many pixles away was the cube from 0. In our case it was 600
189.     # pixels away. So the X resolution (4288 pixes) divied by 600 gives us a ratio
190.     # of 7.14 This means that for any resolution picture if I keep the same ratio, the
square
191.     # in the middle of the picture real picture will sit almost in the same position in the
virtual image.
192.     ratio_X=7.1466666666666665
193.     ratio_Y=4.746666666666667
194.     Y_value=int(len(gray_generated)//ratio_Y)
195.     X_value=int(len(gray_generated[0])//ratio_X)
196.
197.     # create the foreground (a white square) in the middle of the image (background)
198.     gray_generated[len(gray_generated)//2-X_value:len(gray_generated)//2+X_value,\
199.                    len(gray_generated[0])//2-
Y_value:len(gray_generated[0])//2+Y_value]=fore_ground  #after this command
200.     # we have our image with a cube in the middle generated, just like the real image.
201.     # Now we will return this virtual image.
202.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
203.     return gray_generated
204.
205.
206.
207. def resize_image(image_to_resize, img_ratio, resolution=600):
208.     Y_dim=resolution
209.     return cv2.resize(image_to_resize,(int(Y_dim*img_ratio),int(Y_dim)))
210.
211.
212. ##################################################  MAIN
##################################################
213.
214. FILE_REAL=sys.argv[1]
215. FILE_REAL2=sys.argv[2]
216.
217. # read real images
218. gray1=cv2.imread(FILE_REAL, 0)
219. gray2=cv2.imread(FILE_REAL2, 0)
220.
221. # ratio of real image in order to display both the real image and the virtual images
222. # resized to fit on the screen
223. max_res=float(max(len(gray1),len(gray1[0])))
224. min_res=float(min(len(gray1),len(gray1[0])))
225. image_ratio=max_res/min_res
226. # for display purposes, I limit the Y dimension to 600 pixels, and the X dimension is based
227. # on this number and the ratio of the real image, so that both the virtual and the real
image
228. # are resized to the same resolution for display (the calculations are performed on the
real resolution)
229. Y_dimension=600
230.
231. SNR_REAL = compute_snr_real(gray1, gray2)
232. print ("SNR of REAL images: ",SNR_REAL)
233.
234. # calculate the noise needed to apply to the virtual image so that it will have
235. # the same SNR as the real image
236. noise_applied=100.0/SNR_REAL
237. virtual_generated_image=return_virtual_generated_image_based_on_real_image(gray1)
238. SNR = compute_virtual_snr(virtual_generated_image, noise_applied)
239. print ("SNR of VIRTUAL images: ",SNR)
240.
241.
242.
243.
244.
245.
246. ## get number of corners in the VIRTUAL IMAGE
247. corners, image_with_corners_overlayed=no_of_corners(virtual_generated_image)
248. # get number of corners after noise has been applied to above generated file
```

```
249. corners_after_noise, noisy_image_with_corners_overlayed =
no_of_corners_after_applied_noise(virtual_generated_image,noise_applied)
250.
251.
252. # resize for display just in case the resolution is outside of the screen
253. resize_image_with_corners_overlayed=resize_image(image_with_corners_overlayed, image_ratio,
Y_dimension)
254. cv2.imshow('ORIGINAL VIRTUAL image SHOULD have '+str(corners)+"
CORNERS",resize_image_with_corners_overlayed)
255.
256.
257. resize_noisy_image_with_corners_overlayed=resize_image(noisy_image_with_corners_overlayed,
image_ratio, Y_dimension)
258. cv2.imshow('VIRUAL image NOISE '+str(round(noise_applied,3))+ '%, '+\
259.            "REAL "+str(corners)+\
260.            ", SPURIOUS "+str(corners_after_noise-corners)+\
261.            ", TOTAL "+str(corners_after_noise)+\
262.            ", SNR "+str(round(SNR,3)),resize_noisy_image_with_corners_overlayed)
263.
264.
265.
266. ##################################### REAL IMAGE ############################
267.
268. ## get number of corners in the REAL IMAGE
269. REAL_corners, REAL_image_with_corners_overlayed=no_of_corners(gray1)
270.
271. # resize for display just in case the resolution is outside of the screen
272. resize_REAL_image_with_corners_overlayed=resize_image(REAL_image_with_corners_overlayed,
image_ratio, Y_dimension)
273. cv2.imshow('REAL image has '+str(REAL_corners)+" CORNERS and SNR of "+\
274.            str(round(SNR_REAL,3)),\
275.            resize_REAL_image_with_corners_overlayed)
276.
277.
278.
279. if cv2.waitKey(0) & 0xff == 27:
280.         cv2.destroyAllWindows()
281.
```

**Automation that generates the graphs present in the thesis where 256 simulations are run (for 256 different contrast levels) and plots the varying contrast levels against SNR and NOISE figures. It also saves the raw data in EXCEL.**

```
  1. # This program takes a folder structure that contains captured images of a real
  2. # sheet of paper. This sheet of paper has a certain uniform background,
  3. # and in the middle of it there is a square of a shade of gray different
  4. # than the background. The program will go through every picture available and
  5. # will always take a pair of images at at time to compute on. It will compute the SNR of
the
  6. # pair of real images, and then it will generate a pair of virtual images that resemble
  7. # the real images as closely as possible. Noise will be applied on this pair
  8. # of virtual images, so that they have an SNR similar to the SNR of the real images.
  9. # Then, the corner detector will be applied on one of the real image, and also on
 10. # the virtual image with noise applied to it. The purpose is to see if
 11. # the corner detector will find the same number of corners on the virtual image
 12. # as it finds on the real image. If this is so, then simulated images could prove
 13. # to be a replacement for the real images.
 14.
 15. import os
 16. import numpy
 17. import cv2
 18. import scipy.stats
 19. import math
 20. from tabulate import tabulate  # needed to display the results
 21. import pandas # needed to diplay the results in an EXCEL (xlsx) file
 22.
 23. def return_results(FILE1, FILE2):
 24.
 25.     def return_virtual_generated_image_based_on_real_image(REAL_Image):
 26.         # I will calcuate the background and foreground of the image in order to generate
an
 27.         # artificial image with the same background and foreground shades of gray as the
real image
 28.
 29.         # the size (in pixels) of the side of a square in the center of the screen where I
will sample each
 30.         # pixel in the real image for the shade of gray required for the foreground of
 31.         # of the virtual image. The the mean value will be the shade of gray used to build
 32.         # the foreground in the virtual image
 33.         gray1=REAL_Image
 34.         square_side=20
 35.         all_pixels_from_square=[]
 36.         for i in range (gray1.shape[0]/2-square_side/2, gray1.shape[0]/2+square_side/2,1):
 37.             for j in range (gray1.shape[1]/2-square_side/2,
gray1.shape[1]/2+square_side/2,1):
 38.                 all_pixels_from_square.append(gray1[i,j])
 39.         fore_ground=int(numpy.mean(all_pixels_from_square))
 40.
 41.         # the side of the sqare starting with topmost and leftomost pixel of the
 42.         # real image in order to detect tge bacground shade of gray that will be applied
 43.         # to the background of the virtual image
 44.         all_pixels_from_square=[]
 45.         for i in range (0, square_side,1):
 46.             for j in range (0, square_side,1):
 47.                 all_pixels_from_square.append(gray1[i,j])
 48.         back_ground=int(numpy.mean(all_pixels_from_square))
 49.
 50.         # generate virtual image to resemble the real image
```

```
 51.         gray_generated=numpy.zeros((len(gray1),len(gray1[0])))
 52.         gray_generated[:,:]=back_ground  # apply backround share of gray
 53.         gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
 54.
 55.         # ratio of how far the white sqare is from the boundaries of the screen.
 56.         # the ratio is both for the X and for the Y coordinates. How I did it? I took
 57.         # an example and then I calculated the ratio between the maximum X resolution
 58.         # and how many pixles away was the cube from 0. In our case it was 600
 59.         # pixels away. So the X resolution (4288 pixes) divied by 600 gives us a ratio
 60.         # of 7.14 This means that for any resolution picture if I keep the same ratio, the
square
 61.         # in the middle of the picture real picture will sit almost in the same position in
the virtual image.
 62.         ratio_X=7.1466666666666665
 63.         ratio_Y=4.746666666666667
 64.         Y_value=int(len(gray_generated)/ratio_Y)
 65.         X_value=int(len(gray_generated[0])/ratio_X)
 66.
 67.         # create the foreground (a white square) in the middle of the image (background)
 68.         gray_generated[len(gray_generated)/2-X_value:len(gray_generated)/2+X_value,\
 69.                        len(gray_generated[0])/2-
Y_value:len(gray_generated[0])/2+Y_value]=fore_ground  #after this command
 70.         # we have our image with a cube in the middle generated, just like the real image.
 71.         # Now we will return this virtual image.
 72.         gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
 73.         return gray_generated
 74.
 75.
 76.
 77.     def no_of_corners(my_image):
 78.
 79.         gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
 80.
 81.         dst = cv2.cornerHarris(gray,2,3,0.04)
 82.
 83.         treshold_value=0.01 * dst.max() #getting a treshold value
 84.         tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
 85.
 86.         dst = numpy.uint8(dst) # the command below needs integer numbers
 87.         ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 88.         num_corners = labels.max() # corners being detected
 89.
 90.         return num_corners
 91.
 92.
 93.     def no_of_corners_after_applied_noise(my_image, factor, mean=0.0, sd=1.0, seed=None):
 94.
 95.         gray = numpy.float32(my_image) # convert to float for data manipulation
 96.         if not seed is None: numpy.random.seed (seed)
 97.         " add the noise "
 98.
 99.         sd=gray.std(ddof=0)
100.         percentage_of_noise_added=sd*factor/100.0
101.
102.         gray_and_noisy = numpy.add(gray, numpy.random.normal (mean,
percentage_of_noise_added, gray.shape))
103.         gray_and_noisy = numpy.float32(gray_and_noisy)   #Harris detector needs FLOAT 32
104.
105.         dst = cv2.cornerHarris(gray_and_noisy,2,3,0.04)
106.
107.         treshold_value=0.01 * dst.max() #getting a treshold value
108.         tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
109.
110.         dst = numpy.uint8(dst) # the command below needs integer numbers
111.         ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
112.         num_corners = labels.max() # corners being detected
113.
114.         return num_corners
115.
```

```
116.
117.
118.     def compute_virtual_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
119.         # Read in the image and convert to float.  Duplicate it and add noise.
120.         im1 = my_image
121.         im1 = im1.astype("float32")
122.         im2 = im1.copy()
123.
124.         sd=im1.std(ddof=0)
125.         percentage_of_noise_added=sd*factor/100.0
126.
127.         im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added,
im1.shape))
128.         im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added,
im2.shape))
129.
130.         # Compute the correlation coefficient and SNR.
131.         r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
132.         #print ("r:", r)
133.         snr = math.sqrt (r / (1 - r))
134.         #print ("SNR:", snr)
135.
136.         return snr
137.
138.
139.     def compute_snr_real(my_image1, my_image2):
140.
141.         im1 = my_image1.astype("float32")
142.         im2 = my_image2.astype("float32")
143.
144.         # Compute the correlation coefficient and SNR.
145.         r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
146.         #print ("r:", r)
147.         snr = math.sqrt (r / (1 - r))
148.         #print ("SNR:", snr)
149.
150.         return snr
151.
152.
153.
154.
155. # ---------BEGIN----------- FUNCTION "RETURN_RESULTS" MAIN ---------BEGIN----------
156.
157.     gray1=cv2.imread(FILE1, 0) # read the image directly to grayscale
158.     gray2=cv2.imread(FILE2, 0) # read the image directly to grayscale
159.     SNR_REAL = compute_snr_real(gray1, gray2) # compute the SNR of the real images
160.
161.     # calculate the noise needed to apply to the virtual image so that it will have
162.     # the same SNR as the real image
163.     noise_applied=100.0/SNR_REAL
164.
165.     # generate the virtual image based on the real image
166.     virtual_generated_image=return_virtual_generated_image_based_on_real_image(gray1)
167.     SNR = compute_virtual_snr(virtual_generated_image, noise_applied) # SNR of the virtual
image (for verification)
168.
169.     ############################## VIRTUAL IMAGE ####################
170.     # get the number of corners for the virtual generated image. Since this image
171.     # doesn't have any noise, and given that we have square in the middle of the image,
172.     # the corners returned will be 4. This is how we know how many true corners
173.     # should be in the image.
174.     corners = no_of_corners(virtual_generated_image)
175.     # get number of corners after noise has been applied to the virtually generated image
176.     corners_after_noise = no_of_corners_after_applied_noise(virtual_generated_image,
noise_applied)
177.
178.     ################################ REAL IMAGE ################
179.     REAL_corners = no_of_corners(gray1)
180.     return round(SNR_REAL,3), round(SNR,3), REAL_corners, round(noise_applied,3),
corners_after_noise, corners
```

```
181.
182.
183. # ----------END------------ FUNCTION "RETURN_RESULTS" MAIN ----------END-----------
184.
185.
186.
187. ########################## MAIN PROGRAM ######################################
188.
189.
190. # currently there is a group of 3 images with the same resolution taken with
191. # various noise settings (light, dark, ISO, etc) in order to produce
192. # different levels of noise. We take these 3 images and we compute their SNR
193. # by taking all the possible combinations of 2 images at a time.
194. # SNR computation requires 2 images. So, out of a selection  of 3 images,
195. # we have the following pairs: 1-2; 1-3; 2-3. If we had a selection
196. # of 4 images, then we would have had the following pairs of images:
197. # 1-2; 1-3; 1-4; 2-3; 2-4; 3-4; The variable below specifies how many images we
198. # have per selection and the code below that computes all the possible pairs of images
199. # given the current selection. If in the future we will add more variations of
200. # noise per captured image, then the selection will increase, and I only have
201. # to change the value below so all the images get computed automatically.
202. selection=3
203.
204. # set the current directory to be the directoru where this current ".PY" files is
205. # executed from
206. cwd = os.path.dirname(os.path.realpath(__file__))
207.
208. # getting a list of all the folders that contain the images to be analysed
209. os.chdir(os.getcwd()+"\\CAMERAS")
210. dir_structure=os.listdir(os.getcwd())
211.
212. # this list will store all the simulated data for storage in a TXT file
213. data=[]
214.
215. for FOLDER in dir_structure:
216.     os.chdir(os.getcwd()+"\\"+FOLDER) # enter inside a particular camera folder
217.     files_structure=os.listdir(os.getcwd())
218.
219.     # check how many files we have in the folder and divide the total number by
220.     # the selection. This way we know how many iterations we need to do until all
221.     # selections have been taken into consideration
222.     for k in range(0,len(files_structure)/selection,1):
223.
224.         for i in range(0+k*selection,(selection-1)+k*selection,1):
225.             for j in range(i+1,selection+k*selection,1):
226.                 print("DIR: "+FOLDER+", PAIRS: ["+files_structure[i]+",
"+files_structure[j]+"]")
227.                 SNR_R, SNR_V, R_CORN, NSA, CORN_A_N, TRUE_CORN =
return_results(files_structure[i],files_structure[j])
228.
229.                 data.append([FOLDER, files_structure[i], files_structure[j], SNR_R, R_CORN,
str(NSA)+"%", SNR_V, CORN_A_N, TRUE_CORN])
230.
231.     # go back the directry tree once finished in current directory
232.     os.chdir(os.getcwd()+"\\..")
233.
234.
235.
236. # the heading of each column of the table needs to be specified for tabulate to work
237. header_names=["CAMERA","FILE_1","FILE_2","SNR_R","CORNERS_RI","APLIED_NOISE",\
238.                "SNR_V","CORNERS_VI","NO_NOISE_CORNERS_VI"]
239. # print(tabulate(data, headers=header_names, tablefmt="fancy_grid", showindex="always"))
240.
241.
242. # go back out of the "CAMERAS" folder and store the results in a TXT file
243. # in the folder where the current ".PY" files is.
244. os.chdir(os.getcwd()+"\\..")
245.
246. # write resulta to a TXT file
247. with open("results.txt", "wb") as f:
```

```
248.     f.write(tabulate(data, headers=header_names, tablefmt="grid", stralign="center",
numalign="center", showindex="always"))
249.     #f.write(tabulate(data, headers=header_names, tablefmt="fancy_grid", stralign="center",
numalign="center", showindex="always").encode("utf8"))
250.
251. # write results to a HTML file
252. with open("results.html", "wb") as f:
253.     f.write(tabulate(data, headers=header_names, tablefmt="html", stralign="center",
numalign="center", showindex="always"))
254.
255.
256. # data for EXCEL (for easier visualisation)
257. CAMERA=[]               # Curent imaging device (folder name)
258. FILE_1=[]              # First file as part of the "2-files computation"
259. FILE_2=[]              # Second file as part of the "2-files computation"
260. SNR_R=[]               # SNR obtained from the First and Second file
261. CORNERS_RI=[]          # Number of corners detected on one of these 2 images
262. APPLIED_NOISE=[]       # The noise to be applied on the virtual images to get the same SNR
as the real images
263. SNR_V=[]               # SNR of the virtual image after noise application (not needed,
just for verification purposes)
264. CORNERS_VI=[]          # Number of corners detected on the virtual image BEFORE noise
application
265. NO_NOISE_CORNERS_VI=[]  # Number of corners detected on the virtual image AFTER noise
application
266.
267. for i in range (0,len(data),1):
268.     CAMERA.append(data[i][0])
269.     FILE_1.append(data[i][1])
270.     FILE_2.append(data[i][2])
271.     SNR_R.append(data[i][3])
272.     CORNERS_RI.append(data[i][4])
273.     APPLIED_NOISE.append(data[i][5])
274.     SNR_V.append(data[i][6])
275.     CORNERS_VI.append(data[i][7])
276.     NO_NOISE_CORNERS_VI.append(data[i][8])
277.
278. data_EXCEL = pandas.DataFrame({header_names[0]:CAMERA,\
279.                                header_names[1]:FILE_1,\
280.                                header_names[2]:FILE_2,\
281.                                header_names[3]:SNR_R,\
282.                                header_names[4]:CORNERS_RI,\
283.                                header_names[5]:APPLIED_NOISE,\
284.                                header_names[6]:SNR_V,\
285.                                header_names[7]:CORNERS_VI,\
286.                                header_names[8]:NO_NOISE_CORNERS_VI})
287.
288.
289. # this re-sorts the columns in the right order as they exist in the "header_names".
290. # if this instruction would have not been written, then when writing to the XLSX
291. # file, all the columns would have been sorted alphabetically, despite clearly
292. # creating the data in the above order.
293. data_EXCEL=data_EXCEL[header_names]
294. # write data to file
295. data_EXCEL.to_excel('sample_data.xlsx', sheet_name='sheet1', index = False)
296.
297.
298.
299. # the following code is for CENTER alligning the data in the excel file
300. # and also for adjusting column width according to the longest string so
301. # that it is more easier to visualise
302. from openpyxl import load_workbook
303. from openpyxl.styles import Alignment
304.
305. wb = load_workbook(filename = r'sample_data.xlsx')
306. mysheet = wb['sheet1']
307.
308. # alligning every cell to CENTER
309. for c_row in range(1,mysheet.max_row+1):
310.     for c_col in range(1,mysheet.max_column+1):
```

```
311.         cell=mysheet.cell(row=c_row, column=c_col)
312.         cell.alignment = Alignment(horizontal='center', vertical='center')
313.
314. # finding the MAX length of every cell, and estabblishing the "column_dimensions"
315. # to be that maximum found length and add 5 more empty spaces so that the data
316. # looks more dispersed
317. for column_cells in mysheet.columns:
318.    length = max(len(str(cell.value)) for cell in column_cells)
319.    mysheet.column_dimensions[column_cells[0].column].width = length+5
320.
321. # save the new EXCEL file
322. wb.save(r'sample_data.xlsx')
323.
```

## Program that detects blur in the real image and generates a virtual image with the same blur

```
# this program is supposed to apply the same blur in the virtual image as that found
# in the real image. The way to calculate this is to get the real image, and get the values
# of a single line, preferably the hotizonbtal line at the middle of the screen.
# We then look for the gradient of change from the lowest value of background to
# the highest value of the foreground. We then see how many values exist between
# these 2 extreme values. This will be our GRADIENT. Then on the virtual image we will
# blur the image consecutively until we get a similar gradient of values between foreground
# and background

import cv2
import scipy.stats
import numpy
import math


def add_noise_get_noisy_image_match_snr(my_image, match, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = my_image
    im1 = im1.astype("float32")
    im2 = im1.copy()
    sd=im1.std(ddof=0)

    # the routine below keeps adding noise to our virtual image until the SNR
    # is matched
    noise_added=1.0  #just so that the operation runs faster (normally it is 0.0)
    while 1:
        noise_added+=0.1
        noise_added=round(noise_added,2)

        temp1 = numpy.add(im1, numpy.random.normal (mean, noise_added, im1.shape))
        temp2 = numpy.add(im2, numpy.random.normal (mean, noise_added, im2.shape))
        r = scipy.stats.pearsonr (temp1.ravel(), temp2.ravel())[0]
        snr = math.sqrt (r / (1 - r))

        print("MATCHING SNR: "+str(snr)+", added noise: "+str(noise_added)+\
              ", which is "+str(round((noise_added/sd)*100,2))+"% of SD of image")
        if (round(snr,1) <= round(match,1)):
            break

    # make sure there are no values lower than 0 and higher than 255 from the
    # random noise application
    noisy_for_display = numpy.clip(temp1, 0, 255)
    noisy_for_display = numpy.uint8(noisy_for_display)

    print("\n")
    print("VIRTUAL SNR: "+str(snr)+" was matched to REAL SNR: "+str(match))
    print("\n")
    return snr, noisy_for_display


# function that detects the number of corners in an image
def no_of_corners(my_image):
    gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float

    # The "magic" instruction of corner detection
    dst = cv2.cornerHarris(gray,2,3,0.04)

    treshold_value=0.01 * dst.max() #getting a treshold value
    tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)

```

```
 62.     dst = numpy.uint8(dst) # the command below needs integer numbers
 63.
 64.     # Find centroids for a more accurate detection
 65.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
 66.     # corners being detected (each corner has a numberic label. The highest numerical label
 67.     # imdicates the corners detected. Labels are incraesed one by one as the corner
detector
 68.     # finds corners
 69.     num_corners = labels.max()
 70.
 71.     #result is dilated for marking the corners, not important
 72.     dst = cv2.dilate(dst,None)
 73.
 74.     # place gray dots on the image to show where the corners are
 75.     gray[dst==255]=[137]
 76.
 77.     gray = numpy.uint8(gray)
 78.     return num_corners, gray
 79.
 80.
 81.
 82. # This is just a function for quikcly computing the SNR of 2 images.
 83. def compute_snr_real(my_image1, my_image2):
 84.
 85.     im1 = my_image1.astype("float32")
 86.     im2 = my_image2.astype("float32")
 87.
 88.     # Compute the correlation coefficient and SNR.
 89.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
 90.     snr = math.sqrt (r / (1 - r))
 91.     return snr
 92.
 93.
 94. def return_virtual_generated_image_based_on_real_image(REAL_Image, amplitude=10,
frequency=1):
 95.     # I will calcuate the background and foreground of the image in order to generate an
 96.     # artificial image with the same background and foreground shades of gray as the real
image
 97.
 98.     # the size (in pixels) of the side of a square in the center of the screen where I will
sample each
 99.     # pixel in the real image for the shade of gray required for the foreground of
100.     # of the virtual image. Then the mean value will be the shade of gray that will be used
to build
101.     # the foreground in the virtual image
102.     gray1=REAL_Image
103.     square_side=20
104.     all_pixels_from_square=[]
105.     for i in range (gray1.shape[0]//2-square_side//2, gray1.shape[0]//2+square_side//2,1):
106.         for j in range (gray1.shape[1]//2-square_side//2,
gray1.shape[1]//2+square_side//2,1):
107.             all_pixels_from_square.append(gray1[i,j])
108.     fore_ground=int(numpy.mean(all_pixels_from_square))
109.
110.     # the side of the sqare starting with topmost and leftomost pixel of the
111.     # real image in order to detect tge bacground shade of gray that will be applied
112.     # to the background of the virtual image
113.     all_pixels_from_square=[]
114.     for i in range (0, square_side,1):
115.         for j in range (0, square_side,1):
116.             all_pixels_from_square.append(gray1[i,j])
117.     back_ground=int(numpy.mean(all_pixels_from_square))
118.
119.     # generate virtual image to resemble the real image
120.     gray_generated=numpy.zeros((len(gray1),len(gray1[0])))
121.
122.     gray_generated[:,:]=back_ground  # apply backround share of gray
123.
124.     # SINE VARIATON to THE BACKGROUND - useless as this detail gets lost with blur
125. #    counter=0
```

```
126. #    for i in range(0,gray_generated.shape[0],1):
127. #        for j in range(0,gray_generated.shape[1],1):
128. #            gray_generated[i,j] = int(math.sin(frequency*counter)*amplitude)+back_ground
129. #            counter+=0.1
130.
131.     # just in case we will have values larger than 255 and smaller than 0
132.     gray_generated = numpy.clip(gray_generated, a_min = 0, a_max = 255)
133.
134.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
135.
136.     # ratio of how far the white sqare is from the boundaries of the screen.
137.     # the ratio is both for the X and for the Y coordinates. How I did it? I took
138.     # an example and then I calculated the ratio between the maximum X resolution
139.     # and how many pixles away was the cube from 0. In our case it was 600
140.     # pixels away. So the X resolution (4288 pixes) divied by 600 gives us a ratio
141.     # of 7.14 This means that for any resolution picture if I keep the same ratio, the
square
142.     # in the middle of the real picture will sit almost in the same position in the virtual
image.
143.     ratio_X=7.1466666666666665
144.     ratio_Y=4.746666666666667
145.     Y_value=int(len(gray_generated)//ratio_Y)
146.     X_value=int(len(gray_generated[0])//ratio_X)
147.
148.     # create the foreground (a white square) in the middle of the image (background)
149.     gray_generated[len(gray_generated)//2-X_value:len(gray_generated)//2+X_value,\
150.                    len(gray_generated[0])//2-
Y_value:len(gray_generated[0])//2+Y_value]=fore_ground  #after this command
151.     # we have our image with a cube in the middle generated, just like the real image.
152.     # Now we will return this virtual image.
153.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
154.     return gray_generated
155.
156.
157. # formula for gausian blur
158. def blur_gauss(my_image, blur_kernel):
159.     # ksize (represente the blurring kernel size)
160.     # we user the 2n+1 formula for obaining odd numbers.
161.     # gausian blur works ONLY with odd numbers
162.     ksize = (2*blur_kernel+1, 2*blur_kernel+1)
163.     image = cv2.GaussianBlur(my_image, ksize, 0)
164.     return image
165.
166.
167. def get_blur_gradient(my_image):
168.     # for the sorting function below
169.     from operator import itemgetter
170.
171.     middle_line = my_image[my_image.shape[0]/2] # get all the values from the middle line
of image
172.     middle_line = middle_line.tolist()  # convert to Pyhon list from numpy array
173.
174.     # count how many times each elemenbts occurs.
175.     count_elements=[]
176.     for i in set(middle_line):
177.         count_elements.append([i, middle_line.count(i)])
178.
179.     # sort elements in ascending order, but the sort happens according to the number
180.     # of occurences of each grey value (this is the 2nd value in the tuple.)
181.     # This means that the background and foreground values will be placed last
182.     # in the list. This is because those values should have the most of occurences
183.     # in our sample data.
184.     count_elements.sort(key=itemgetter(1))
185.
186.     # we can easily get the background and foreground values by taking the last 2 elements
187.     # in the above sorted list
188.     foreground=max(count_elements[-1],count_elements[-2])[0]
189.     background=min(count_elements[-1],count_elements[-2])[0]
190.
```

```
    # by getting the length of the above list, we will know how many distinct gray values have
    # been found in total. We are not concerned with knowing the grey valur itself, but with how many
    # different grey values have been found in total. We substract 2 because we are not concerned
    # with the background and the foreground
    how_many_grey_valyes=len(count_elements)-2

    # Because our simulation picture is simetrical (square in the middle of the screen),
    # we can divide the above value in 2. This will give us the number of grey values
    # that exists between background and foreground. This is exactly the gradient, or the number
    # of intermediary values between foreground and background that needs to be matched when applying
    # blur. We will apply so much blur to our UNBLURED image until we get the same blur gradient
    # like in the original image.

    gradient=int(math.ceil(how_many_grey_valyes/2.0))

    # we also return the foreground and background of the image, just in chase we need to compare
    # these values with values that the image had originally.
    return gradient, foreground, background

# ------------------------------- MAIN ---------------------------------------
im1_s="UNO_BLUR_3.jpg"
im2_s="UNO_BLUR_5.jpg"
im4_s="UNO_PHOTOSHOP.jpg"

im1=cv2.imread(im1_s,0)
im2=cv2.imread(im2_s,0)
im4=cv2.imread(im4_s,0)

# Get the real images SNR so that we can match it for the virtual images
print("\n")
SNR_REAL=compute_snr_real(im1,im2)
print("REAL SNR: ", SNR_REAL)
print("\n")

# detect how much blur is there in one of the "real" images
gradient, fgnd, bgnd=get_blur_gradient(im2)

virtual_generated_image=return_virtual_generated_image_based_on_real_image(im2, amplitude=30, frequency=5)
# Add noise to the image
SNR2, noisy_image  = add_noise_get_noisy_image_match_snr(virtual_generated_image, SNR_REAL)

counter=1
while 1:
    blurred_image=blur_gauss(noisy_image, counter)
    gradient2, fgnd, bgnd=get_blur_gradient(blurred_image)
    print("Current blur gradient: "+str(gradient2)+", to match REAL IMAGE blur gradient: "+str(gradient))
     # without the -2 ofset, the matched virtual image appears more blurry visually than the real image,
     # despite both having the same gradient.
    if (gradient2 >= gradient-2):
        break
    counter+=1


# --------------------- Virtual IMAGE CORNER DETECTION --------------------------
returned_corner2, returned_image2=no_of_corners(blurred_image)
cv2.imshow("2 - Virtual Img -  Corners detected: "+str(returned_corner2), returned_image2)

# --------------------- REAL IMAGE CORNER DETECTION --------------------------
returned_corner3, returned_image3=no_of_corners(im2)
```

```
cv2.imshow("3 - Real Img - Corners detected: "+str(returned_corner3), returned_image3)

# ---------------------- REAL IMAGE CORNER DETECTION -------------------------
returned_corner4, returned_image4=no_of_corners(im4)
cv2.imshow("4 - Photoshop Img - Corners detected: "+str(returned_corner4), returned_image4)

if cv2.waitKey(0) & 0xff == 27:
        cv2.destroyAllWindows()

```

**Program that checks for blur and noise application to generate figures present in thesis.**

```
import cv2
import scipy.stats
import numpy
import math


def moving_average(values, window = 3, lock = "LOCKED"):

#    window is established as a percentage from the total number of elements being fed in
    window = float(window) # for divisin to return a decimal number rather than an int
    window=int((window/100)*len(values))

    # sometimes the number of elements is so small that after transforming the window
    # to integer, we will get a result of zero. In that case we make the window at least
    # 2 elements, as making it only 1 element would do no averaging at all.
    if window==0:
        window=2

#    "LOCKED" means that the window moves in such a way that it does not exit the
#    boundaries of the original data. This will yield a moving average that has a lower
#    number of elements than the original data.
#
#    "UNLOCKED" is where the window moves outside of the boundaries of the original data.
#    The positions before the beginning and after the end of the data are considered to be
0.
#    This will yield a moving average that has a higher number of elements than the
original data.
#
#    "FIXED" is where the average mean will have the same number of elements like
#    the original data. This is achieved by always calculating the mean from the
#    neighbouring elements limited to the window size. Because of this, the moving average
#    of the first elements of data  will have the same value due to the window that
#    falls on them.
#
#    "FIXED BALANCED" is where the average mean will have the same number of elements
#    like the original data. The difference to the above is that this method is balanced.
#    The window moves slowly to the right in such a way that the number of elements
compared
#    to the middle of the window is always 1 element greater than the number of
#    elements before the middle of the window. When reaching the end where there is nowhere
to go
#    the window will remain fixed. This usually means that the last elements from the
original data
#    will have the same mean. This balanced version was developed to create a SMART and
smoother
#    way to calcualte the moving average.

    window = window
    list_of_averages = []
    intermediate_sum=0
    if window <= len(values):

        if lock == "LOCKED":  # returns less elements

            for j in range (0, len(values)-window+1, 1):
                for i in range(j, j+window, 1):
                    intermediate_sum+=values[i]

                list_of_averages.append(intermediate_sum / window)
                intermediate_sum=0

        if lock == "UNLOCKED": # returns more elements
```

```
 57.
 58.             for j in range (0-window+1, len(values), 1):
 59.                 for i in range(j, j+window, 1):
 60.
 61.                     if i<0 or i>=len(values):
 62.                         intermediate_sum+=0
 63.                     else:
 64.                         intermediate_sum+=values[i]
 65.
 66.                 list_of_averages.append(intermediate_sum / window)
 67.                 intermediate_sum=0
 68.
 69.         if lock == "FIXED": # returns same elements, bot more repeatable values
 70.
 71.             for j in range (0, len(values), 1):
 72.
 73.                 if (j+window) <= len(values):
 74.                     for i in range(j, j+window, 1):
 75.                         intermediate_sum+=values[i]
 76.
 77.                 else:
 78.                     position_backward = (j+window)-len(values)
 79.                     for i in range(j-position_backward, len(values), 1):
 80.                         intermediate_sum+=values[i]
 81.
 82.                 list_of_averages.append(intermediate_sum / window)
 83.                 intermediate_sum=0
 84.
 85.
 86.         if lock == "FIXED_BALANCED": #returns same elements but balanced and not so
repeatable.
 87.             mooving_window=0
 88.
 89.             for j in range (0, len(values), 1):
 90.
 91.                 window_values_left_ahead = mooving_window+window-j-1
 92.                 window_values_left_behind = window-window_values_left_ahead-1
 93.                 difference = window_values_left_ahead-window_values_left_behind
 94.
 95.                 # we try to have more elements ahead than behind
 96.                 if (difference >=0):
 97.                     for i in range(0+mooving_window, window+mooving_window, 1):
 98.                         intermediate_sum+=values[i]
 99.
100.                 # if there is only one element left behind, then we move the window
101.                 if (difference <=-1):
102.
103.                     # if we still haven't reached the end of the data, we run the code
below
104.                     if (mooving_window+window) < len(values):
105.                         mooving_window+=1
106.                         for i in range(0+mooving_window, window+mooving_window, 1):
107.                             intermediate_sum+=values[i]
108.                     # if we are at the end of the original data, we keep doing the same
mean
109.                     # because anyway it falls under the same window, and we do it until
110.                     # the index "j" has cycled through all the elements in the original
data.
111.                     else:
112.                         position_backward = (mooving_window+window)-len(values)
113.                         for i in range(mooving_window-position_backward, len(values), 1):
114.                             intermediate_sum+=values[i]
115.                         mooving_window+=1
116.
117.
118.                 list_of_averages.append(intermediate_sum / window)
119.                 intermediate_sum=0
120.
121.
122.     else:
```

```
123.         return "Window size is larger than the data set. Please choose a smaller window
size."
124.
125.     return list_of_averages
126.
127.
128. def add_noise_get_noisy_images_match_snr2(my_image1, my_image2, match, mean=0.0, sd=1.0,
seed=None):
129.     # Read in the image and convert to float.  Duplicate it and add noise.
130.     im1 = my_image1
131.     im1 = im1.astype("float32")
132.     im2 = my_image2
133.     im2 = im2.astype("float32")
134.
135.     # to help the fist if instruction that contains "break". We need to know
136.     # the "snr" for that "if" instruction to work.
137.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
138.     snr = math.sqrt (r / (1 - r))
139. #    sd1=im1.std(ddof=0)
140. #    sd2=im2.std(ddof=0)
141.
142.     # the routine below keeps adding noise to our virtual image until the SNR
143.     # is matched
144.     noise_added=1.0  #just so that the operation runs faster (normally it is 0.0)
145.     while 1:
146.         # just in case the SNR of the virtual images is smaller from the beginning,
147.         # then we won't need to be adding noise.
148.         if (round(snr,1) <= round(match,1)):
149.             temp1=im1
150.             temp2=im2
151.             break
152.
153.         noise_added+=0.1
154.         noise_added=round(noise_added,2)
155.
156.         # create a gausian noise map which I will blur
157.         noise_map1 = numpy.random.normal (mean, noise_added, im1.shape)
158.         noise_map2 = numpy.random.normal (mean, noise_added, im2.shape)
159.
160.         # now we blur this nosie map
161.         #blurred_noise = cv2.filter2D (src=noise, ddepth=-1, kernel=blur)
162.         blurred_noise1 = blur_gauss(noise_map1, 5)
163.         blurred_noise2 = blur_gauss(noise_map2, 5)
164.
165.         temp1 = numpy.add(im1, blurred_noise1)
166.         temp2 = numpy.add(im2, blurred_noise2)
167.         r = scipy.stats.pearsonr (temp1.ravel(), temp2.ravel())[0]
168.         snr = math.sqrt (r / (1 - r))
169.
170.         print("MATCHING SNR: "+str(snr)+", added noise: "+str(noise_added)+\
171.               ", which is "+str(round((noise_added/sd)*100,2))+"% of SD of image")
172.         if (round(snr,1) <= round(match,1)):
173.             break
174.
175.     # make sure there are no values lower than 0 and higher than 255 from the
176.     # random noise application
177.     noisy_for_display1 = numpy.clip(temp1, 0, 255)
178.     noisy_for_display1 = numpy.uint8(noisy_for_display1)
179.     noisy_for_display2 = numpy.clip(temp2, 0, 255)
180.     noisy_for_display2 = numpy.uint8(noisy_for_display2)
181.
182.     snr=round(snr,5)
183.     print("\n")
184.     print("VIRTUAL SNR: "+str(snr)+" was matched to REAL SNR: "+str(match))
185.     print("\n")
186.     return snr, noisy_for_display1, noisy_for_display2
187.
188.
189.
190.
```

```
191. # function that detects the number of corners in an image
192. def no_of_corners(my_image):
193.     #crit = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)
194.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
195.
196.     # The "magic" instruction of corner detection
197.     dst = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)
198.     dst = cv2.dilate(dst,None)
199.
200.     treshold_value=0.01 * dst.max() #getting a treshold value
201.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
202.
203.     dst = numpy.uint8(dst) # the command below needs integer numbers
204.
205.     # Find centroids for a more accurate detection
206.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
207.     # corners being detected (each corner has a numberic label. The highest numerical label
208.     # imdicates the corners detected. Labels are incraesed one by one as the corner
detector
209.     # finds corners
210.
211.     # find number of corners with subpixel accuracy
212. #    corners = cv2.cornerSubPix (my_image, numpy.float32(centroids), (5,5), (-1,-1), crit)
213. #    ncorners = len (corners)
214.
215.     # ncorners = len (centroids) - 1
216.     num_corners = labels.max()
217.
218.     #result is dilated for marking the corners, not important
219.     #dst = cv2.dilate(dst,None)
220.
221.     # place gray dots on the image to show where the corners are
222.     # "dst==255" creates a temporary "TRUE/FALSE" matrix of the same size as "dst" and
"gray",
223.     # and then using this BOOLEAN matrix, the "gray" matrix will write "137" to those
224.     # coordinates coresponding to where the temporary BOOLEAN matrix resulting from
"dst=255", is "TRUE"
225.     # gray[dst==255]=[137]
226.
227.     # not let's create a gray dot (137 value), surrounded by 8 more gray dots (for
visibility),
228.     # for every centroid that has been detected, on our image "gray".
229.
230.     # convert our gray image back to int
231.     gray = numpy.uint8(gray)
232.     # first let's convert the coordinates of all the centroids into INT as they are float.
233.     int_centroids = centroids.astype("int32")
234.     for x, y in int_centroids:
235.         gray[y,x]=60   # middle center - main point pf the centroid
236.
237.         gray[y-1,x-1]=210 # top left
238.         gray[y-1,x]=210 # top center
239.         gray[y-1,x+1]=210 # top right
240.
241.         gray[y,x-1]=210 # middle left
242.         gray[y,x+1]=210 # middle right
243.
244.         gray[y+1,x-1]=210 # bottom left
245.         gray[y+1,x]=210 # top center
246.         gray[y+1,x+1]=210 # top righ
247.     #return ncorners, num_corners, gray
248.     return num_corners, gray
249.
250.
251.
252. # This is just a function for quikcly computing the SNR of 2 images.
253. def compute_snr_real(my_image1, my_image2):
254.
255.     im1 = my_image1.astype("float32")
```

```
256.     im2 = my_image2.astype("float32")
257.
258.     # Compute the correlation coefficient and SNR.
259.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
260.     snr = math.sqrt (r / (1 - r))
261.     return snr
262.
263.
264. def return_virtual_generated_image_based_on_real_image(REAL_Image):
265.     # I will calcuate the background and foreground of the image in order to generate an
266.     # artificial image with the same background and foreground shades of gray as the real
image
267.
268.     # the size (in pixels) of the side of a square in the center of the screen where I will
sample each
269.     # pixel in the real image for the shade of gray required for the foreground of
270.     # of the virtual image. Then the mean value will be the shade of gray that will be used
to build
271.     # the foreground in the virtual image
272.     gray1=REAL_Image
273.     square_side=20
274.     all_pixels_from_square=[]
275.     for i in range (gray1.shape[0]//2-square_side//2, gray1.shape[0]//2+square_side//2,1):
276.         for j in range (gray1.shape[1]//2-square_side//2,
gray1.shape[1]//2+square_side//2,1):
277.             all_pixels_from_square.append(gray1[i,j])
278.     fore_ground=int(numpy.mean(all_pixels_from_square))
279.
280.     # the side of the sqare starting with topmost and leftomost pixel of the
281.     # real image in order to detect tge bacground shade of gray that will be applied
282.     # to the background of the virtual image
283.     all_pixels_from_square=[]
284.     for i in range (0, square_side,1):
285.         for j in range (0, square_side,1):
286.             all_pixels_from_square.append(gray1[i,j])
287.     back_ground=int(numpy.mean(all_pixels_from_square))
288.
289.     # generate virtual image to resemble the real image
290.     gray_generated=numpy.zeros((len(gray1),len(gray1[0])))
291.
292.     gray_generated[:,:]=back_ground  # apply backround share of gray
293.
294.     # just in case we will have values larger than 255 and smaller than 0
295.     gray_generated = numpy.clip(gray_generated, a_min = 0, a_max = 255)
296.
297.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
298.
299.     # ratio of how far the white sqare is from the boundaries of the screen.
300.     # the ratio is both for the X and for the Y coordinates. How I did it? I took
301.     # an example and then I calculated the ratio between the maximum X resolution
302.     # and how many pixles away was the cube from 0. In our case it was 600
303.     # pixels away. So the X resolution (4288 pixes) divied by 600 gives us a ratio
304.     # of 7.14 This means that for any resolution picture if I keep the same ratio, the
square
305.     # in the middle of the real picture will sit almost in the same position in the virtual
image.
306.     ratio_X=7.1466666666666665
307.     ratio_Y=4.746666666666667
308.     Y_value=int(len(gray_generated)//ratio_Y)
309.     X_value=int(len(gray_generated[0])//ratio_X)
310.
311.     # create the foreground (a white square) in the middle of the image (background)
312.     gray_generated[len(gray_generated)//2-X_value:len(gray_generated)//2+X_value,\
313.                    len(gray_generated[0])//2-
Y_value:len(gray_generated[0])//2+Y_value]=fore_ground  #after this command
314.     # we have our image with a cube in the middle generated, just like the real image.
315.     # Now we will return this virtual image.
316.     gray_generated=gray_generated.astype("uint8") # convert to integer just to be sure
317.     return gray_generated
318.
```

```
319.
320. # formula for gausian blur
321. def blur_gauss(my_image, blur_kernel):
322.     # ksize (represente the blurring kernel size)
323.     # we user the 2n+1 formula for obaining odd numbers.
324.     # gausian blur works ONLY with odd numbers
325.     ksize = (2*blur_kernel+1, 2*blur_kernel+1)
326.     image = cv2.GaussianBlur(my_image, ksize, 0)
327.     return image
328.
329.
330. def get_blur_gradient3(my_image):
331.     # for the sorting function below
332.     from operator import itemgetter
333.
334.     middle_line = my_image[my_image.shape[0]/2] # get all the values from the middle line
of image
335.     middle_line = middle_line.tolist()  # convert to Pyhon list from numpy array
336.
337.     # count how many times each elemenbts occurs.
338.     count_elements=[]
339.     for i in set(middle_line):
340.         count_elements.append([i, middle_line.count(i)])
341.
342.     # sort elements in ascending order, but the sort happens according to the number
343.     # of occurences of each grey value (this is the 2nd value in the tuple.)
344.     # This means that the background and foreground values will be placed last
345.     # in the list. This is because those values should have the most of occurences
346.     # in our sample data.
347.     count_elements.sort(key=itemgetter(1))
348.
349.     # we can easily get the background and foreground values by taking the last 2 elements
350.     # in the above sorted list
351.     foreground=max(count_elements[-1],count_elements[-2])[0]
352.     background=min(count_elements[-1],count_elements[-2])[0]
353.
354.     # by getting the length of the above list, we will know how many distinct gray values
have
355.     # been found in total. We are not concerned with knowing the grey valur itself, but
with how many
356.     # different grey values have been found in total. We substract 2 because we are not
concerned
357.     # with the background and the foreground
358.     how_many_grey_valyes=len(count_elements)-2
359.
360.     # Because our simulation picture is simetrical (square in the middle of the screen),
361.     # we can divide the above value in 2. This will give us the number of grey values
362.     # that exists between background and foreground. This is exactly the gradient, or the
number
363.     # of intermediary values between foreground and background that needs to be matched
when applying
364.     # blur. We will apply so much blur to our UNBLURED image until we get the same blur
gradient
365.     # like in the original image.
366.
367.     gradient=int(math.ceil(how_many_grey_valyes/2.0))
368.
369.     # we also return the foreground and background of the image, just in chase we need to
compare
370.     # these values with values that the image had originally.
371.     return gradient, foreground, background
372.
373.
374. def get_blur_gradient2(my_image, smoothing_factor = 3):
375.     # "smoothing_factor" is the percentage that will be passed to
376.     # the function "moving_average". It is called "smoothing_factor" because
377.     # the data looks less raggedy when plotted on the screen  .
378.     # It is actually the "moving window" parameter
379.     # that is used to calculate the moving average of a set of data.
380.     # The "moving_average" function will compute the moving average of
```

```
381.     # the "middle_line" below, in order to better compute the gradient
382.     # of blur between the foreground and the background.
383.
384.     # for the sorting function below
385.     from operator import itemgetter
386.
387.     middle_line = my_image[my_image.shape[0]/2] # get all the values from the middle line
of image
388.     middle_line = middle_line.tolist()  # convert to Pyhon list from numpy array
389.     middle_line = moving_average(middle_line, smoothing_factor)
390.
391.
392.     # count how many times each elemenbts occurs.
393.     count_elements=[]
394.     for i in set(middle_line):
395.         count_elements.append([i, middle_line.count(i)])
396.
397.     # sort elements in ascending order, but the sort happens according to the number
398.     # of occurences of each grey value (this is the 2nd value in the tuple.)
399.     # This means that the background and foreground values will be placed last
400.     # in the list. This is because those values should have the most of occurences
401.     # in our sample data. This is how we know what are the values for background and
402.     # the foreground. The should be the most numerous.
403.     count_elements.sort(key=itemgetter(1))
404.
405.     # we can easily get the background and foreground values by taking the last 2 elements
406.     # in the above sorted list
407.     foreground=max(count_elements[-1],count_elements[-2])[0]
408.     background=min(count_elements[-1],count_elements[-2])[0]
409.
410.
411.     """Now that we know that background and foreground values, we can count inside
412.      "middle_line" how many values we have between the last value of the background to
413.      the first value of the fireground"""
414.
415.     max_value=0
416.     min_value=0
417.
418.     # find the first position of the maximum value inside "middle_line", which
419.     # is the value to the foreground.
420.     for i in range(0, len(middle_line), 1):
421.         if (middle_line[i] == foreground):
422.             max_value = i
423.             break
424.
425.     # then from the first max value we find, we go down to find the first minimum value,
426.     # which is the value for the background
427.     for i in range(max_value, 0, -1):
428.         if (middle_line[i] == background):
429.             min_value = i
430.             break
431.
432.     # we need to eliminate the posibility of having double values in the middle due to
averaging
433.     make_a_set = set(middle_line[min_value:max_value])
434.
435.     """having "max_value" and " min_value", the difference between them is the gradient
436.     of values between the background and the foreground. This gradient is the blur
gradient."""
437.
438.     gradient = len(make_a_set)
439.
440.     """The problem with this way of doing things is that as the average makes the data
441.     smoother, there will be more values that are the same. This means that when counting
them
442.     to detect what is the value for the foreground and the background, we may have one of
these values
443.     have a higher count than the BG and FG. This will occur with larger windows for the
444.     moving average. So the solution is use the "get_blur_gradient()" function above where
445.     after we have detected what the foreground and the background is, we then calculate
```

```
446.     how many values are there in betwee these two values. This way we will have the
gradient of blur"""
447.
448.
449.     # we also return the foreground and background of the image, just in chase we need to
compare
450.     # these values with values that the image had originally.
451.     return gradient, foreground, background
452.
453.
454. def get_blur_gradient(my_image):
455.     # for the sorting function below
456.     from operator import itemgetter
457.
458.     middle_line = my_image[my_image.shape[0]/2] # get all the values from the middle line
of image
459.     middle_line = middle_line.tolist()  # convert to Pyhon list from numpy array
460.
461.     # count how many times each elemenbts occurs.
462.     count_elements=[]
463.     for i in set(middle_line):
464.         count_elements.append([i, middle_line.count(i)])
465.
466.     # sort elements in ascending order, but the sort happens according to the number
467.     # of occurences of each grey value (this is the 2nd value in the tuple.)
468.     # This means that the background and foreground values will be placed last
469.     # in the list. This is because those values should have the most of occurences
470.     # in our sample data. This is how we know what are the values for background and
471.     # the foreground. The should be the most numerous.
472.     count_elements.sort(key=itemgetter(1))
473.
474.     # we can easily get the background and foreground values by taking the last 2 elements
475.     # in the above sorted list
476.     foreground=max(count_elements[-1],count_elements[-2])[0]
477.     background=min(count_elements[-1],count_elements[-2])[0]
478.
479.
480.     """Now that we know that background and foreground values, we can count inside
481.      "middle_line" how many values we have between the last value of the background to
482.      the first value of the foreground"""
483.
484.     max_value=0
485.     min_value=0
486.
487.     # find the first position of the maximum value inside "middle_line", which
488.     # is the value to the foreground.
489.     for i in range(0, len(middle_line), 1):
490.         if (middle_line[i] == foreground):
491.             max_value = i
492.             break
493.
494.     # then from the first max value we find, we go down to find the first minimum value,
495.     # which is the value for the background
496.     for i in range(max_value, 0, -1):
497.         if (middle_line[i] == background):
498.             min_value = i
499.             break
500.
501.     gradient = max_value-min_value
502.     # we also return the foreground and background of the image, just in chase we need to
compare
503.     # these values with values that the image had originally.
504.     return gradient, foreground, background
505.
506.
507.
508.
509. # ---------------------------------- MAIN ----------------------------------
510.
511. im1_s="ORIGINAL_BLUR3.bmp"
```

```
im2_s="ORIGINAL_BLUR5.bmp"

im1_s="BLUR3_NOISE2.bmp"
im2_s="BLUR5_NOISE2.bmp"

data = [1,2,6,5,7,2,1,2]
data_result=moving_average(data, 3, "FIXED_BALANCED")
data_result2=moving_average(data, 3, "FIXED")
data_result3=moving_average(data, 3, "LOCKED")
data_result4=moving_average(data, 3, "UNLOCKED")

im1=cv2.imread(im1_s,0)
im2=cv2.imread(im2_s,0)

# detect how much blur is there in the 1st real image
gradient_1, fgnd_1, bgnd_1=get_blur_gradient(im1)
# detect how much blur is there in the 2nd real image
gradient_2, fgnd_2, bgnd_2=get_blur_gradient(im2)

# Get the real images SNR so that we can match it for the virtual images
print("\n")
SNR_REAL=compute_snr_real(im1,im2)
SNR_REAL=round(SNR_REAL,5)
print("REAL SNR: ", SNR_REAL)
print("\n")
print("Blur gradient in 1st REAL image: ", gradient_1)
print("Blur gradient in 2nd REAL image: ", gradient_2)
print("\n")


virtual_generated_image_1=return_virtual_generated_image_based_on_real_image(im1)
virtual_generated_image_2=return_virtual_generated_image_based_on_real_image(im2)

counter=1
while 1:
    blurred_image_1=blur_gauss(virtual_generated_image_1, counter)
    gradient_v_1, _, _ =get_blur_gradient(blurred_image_1)
    print("1st Picture - Current blur gradient: "+str(gradient_v_1),"to match REAL IMAGE
blur gradient: "+str(gradient_1))
     # without the -2 ofset, the matched virtual image appears more blurry visually than
the real image,
     # despite both having the same gradient.
    if (gradient_v_1 >= gradient_1-2):
        break
    counter+=1

print("\n")

counter=1
while 1:
    blurred_image_2=blur_gauss(virtual_generated_image_2, counter)
    gradient_v_2, _, _ =get_blur_gradient(blurred_image_2)
    print("2nd Picture - Current blur gradient: "+str(gradient_v_2),"to match REAL IMAGE
blur gradient: "+str(gradient_2))
     # without the -2 ofset, the matched virtual image appears more blurry visually than
the real image,
     # despite both having the same gradient.
    if (gradient_v_2 >= gradient_2-2):
        break
    counter+=1

print("\n")
# Just to see what the SNR of the virtual image is after blur application
SNR_VIRTUAL_BLUR=compute_snr_real(blurred_image_1, blurred_image_2)
SNR_VIRTUAL_BLUR=round(SNR_VIRTUAL_BLUR,5)
print("VIRTUAL SNR after BLUR: ", SNR_VIRTUAL_BLUR)
print("\n")

# Add noise to the image
```

```
577. SNR_VIRTUAL_NOISE, noisy_image_1, noisy_image_2  =
add_noise_get_noisy_images_match_snr2(blurred_image_1, blurred_image_2, SNR_REAL)
578. SNR_VIRTUAL_NOISE = round (SNR_VIRTUAL_NOISE,5)
579. print("VIRTUAL SNR after NOISE", SNR_VIRTUAL_NOISE)
580.
581. # Corner detection
582. num_corn_1, im_with_corn_1 = no_of_corners(noisy_image_1)
583. num_corn_2, im_with_corn_2 = no_of_corners(noisy_image_2)
584.
585. num_corn_3, im_with_corn_3 = no_of_corners(im1)
586. num_corn_4, im_with_corn_4 = no_of_corners(im2)
587.
588.
589. # ------------------------------ Display results ---------------------------
590.
591. cv2.imshow("Image 1 - Real", im1)
592. cv2.imshow("Image 2 - Real", im2)
593.
594. cv2.imshow("Image 1 - Virtual", virtual_generated_image_1)
595. cv2.imshow("Image 2 - Virtual", virtual_generated_image_2)
596.
597. cv2.imshow("Image 1 - Virtual BLUR MATCHED", blurred_image_1)
598. cv2.imshow("Image 2 - Virtual BLUR MATCHED", blurred_image_2)
599.
600. cv2.imshow("Image1 - Virtual NOISE MATCHED", noisy_image_1)
601. cv2.imshow("Image2 - Virtual NOISE MATCHED", noisy_image_2)
602.
603. cv2.imshow("Image 1 - CORNER DETECTION on VIRTUAL ("+str(num_corn_1)+") corners",
im_with_corn_1)
604. cv2.imshow("Image 2 - CORNER DETECTION on VIRTUAL ("+str(num_corn_2)+") corners",
im_with_corn_2)
605.
606. cv2.imshow("Image 1 - CORNER DETECTION on REAL ("+str(num_corn_3)+") corners",
im_with_corn_3)
607. cv2.imshow("Image 2 - CORNER DETECTION on REAL ("+str(num_corn_4)+") corners",
im_with_corn_4)
608.
609.
610.
611. # --------------------- Virtual IMAGE GAUSS CORNER DETECTION -------------------------
612.
613.
614.
615.
616.
617.
618.
619.
620. # TO DO:
621.
622. """ 1. Median blur seems not to return the patterns that gauss blurr returns
623. 2. Try implementing a blur function myself
624. 3. Check to see if a Photoshoped blurred image will have the same patterns when runing the
corner detector
625. 4. Check for other ways to detect the blur.
626. 5. Apply blur first, and then noise, and make sure that the SNR is the same after noise
application
627.
628. 6. EXPERIMENTAL! Design a function that applies blur and noise incrementally until the blur
and the SNR is the
629. same like in the real image
630.
631.
632. 3 - ANSWER. It seems that the issue is not with the images, but with the blur Gauss
function from OpenCV.
633. When Photoshop applies its own blur, the corner detector detects less corners than when it
is used on
634. the generated and matched virtual image (where blur is applied with the Gauss function).
From here we can see
```

```
635. that the blur gauss function introduces a blur that messes up the corner detector. The
median blur results in
636. the corner detector detecting the right amount of corners like in reality, but the blur
image looks a little
637. bit weird.
638.
639.
640.
641.
642.
643. cv2.imshow("CORNER DETECTOR ("+str(returned_corner1)+") on image matched for blur (GAUSS)
and SNR", returned_image1)
644. if cv2.waitKey(0) & 0xff == 27:
645.         cv2.destroyAllWindows()
646.
647.
648. cv2.imshow("CORNER DETECTOR ("+str(returned_corner2)+") on image matched for blur (MEDIAN)
and SNR", returned_image2)
649. if cv2.waitKey(0) & 0xff == 27:
650.         cv2.destroyAllWindows()
651.
652.
653. cv2.imshow("Virtual Image matched for blur (MEDIAN) and SNR", noisy_image_2)
654. if cv2.waitKey(0) & 0xff == 27:
655.         cv2.destroyAllWindows()
656.
657.
658.
659.
660.
661.
662. """
663.
664.
665.
666.
667. if cv2.waitKey(0) & 0xff == 27:
668.         cv2.destroyAllWindows()
669.
670.
```

## Code that checks how the corner detector works with increasing levels of blur (Version 1)

```
# The code checks how the corner detector works with increasing levels of blur.
# We then use a blur detection function. This function, if it returns low values
# it tells us that there is a lot of blur in the image. If the values returned are
# large, then the image is clear and crisp


#: TO-DO

# - check a real image and see what is the blur returned
# - then apply enough noise on a virtul image (same SNR as the real image)
# and then apply blur and see how the corner detector works
# - use ORB as a corner detector and apply the same steps as above but with the
# ORB detector operating as a corner detector
# - now check the limits of ORB and see how much noise there is needed before
# ORB stops working, and also change the contrast, the same as the simulation done with
# the corner detector before using blurr.
# - and the next step is to apply blur to the orb detector and see how this works as well.


import cv2
import scipy.stats
import numpy
import math


def add_noise_get_noisy_image_get_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = my_image
    im1 = im1.astype("float32")
    im2 = im1.copy()

    sd=im1.std(ddof=0)
    percentage_of_noise_added=sd*factor/100.0

    im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added, im1.shape))
    im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added, im2.shape))

    noisy_for_display = numpy.absolute(im1) # we do not allow any negative nubmers
    # (corresponding to the level of gray) in the image (otherwise we will not be able to
display properly)

    # If the numbers are larger than 255, then when converting to uint8, the numvbers
    # will be (that number larger than 255)-255, and this will yield numbers close to 0.
    # The idea is to limit numbers larger than 255 to 255.

    for i in range(0,noisy_for_display.shape[0],1):
        for j in range(0,noisy_for_display.shape[1],1):
            if noisy_for_display[i][j]>255:
                temp=noisy_for_display[i][j]-255  #eg. 256.345-255= 1.34500
                noisy_for_display[i][j]=255-temp

    noisy_for_display = numpy.uint8(noisy_for_display)

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)
    return snr, noisy_for_display


def compute_snr_real(my_image1, my_image2):
```

```
 62.
 63.     im1 = my_image1.astype("float32")
 64.     im2 = my_image2.astype("float32")
 65.
 66.     # Compute the correlation coefficient and SNR.
 67.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
 68.     #print ("r:", r)
 69.     snr = math.sqrt (r / (1 - r))
 70.     #print ("SNR:", snr)
 71.
 72.     return snr
 73.
 74.
 75. def write_text_on_image(my_file, TEXT):
 76.     x_coord=my_file.shape[0]
 77.     y_coord=my_file.shape[1]
 78.     #bits_of_color=my_file.shape[2]
 79.
 80.     # Write some Text
 81.     font                   = cv2.FONT_HERSHEY_SIMPLEX
 82.     bottomLeftCornerOfText = (10,y_coord-200)
 83.     fontScale              = 0.8
 84.     fontColor              = (255,255,255)
 85.     thickness              = 3
 86.     lineType               = 2
 87.
 88.     cv2.putText(my_file,TEXT,
 89.                 bottomLeftCornerOfText,
 90.                 font,
 91.                 fontScale,
 92.                 fontColor,
 93.                 thickness,
 94.                 lineType)
 95.
 96.     return my_file
 97.
 98.
 99. def no_of_corners(my_image):
100.
101.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
102.
103.     dst = cv2.cornerHarris(gray,2,3,0.04)
104.
105.     treshold_value=0.01 * dst.max() #getting a treshold value
106.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
107.
108.     dst = numpy.uint8(dst) # the command below needs integer numbers
109.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
110.     num_corners = labels.max() # corners being detected
111.
112.     #result is dilated for marking the corners, not important
113.     dst = cv2.dilate(dst,None)
114.
115.     # place gray dots on the image to show where the corners are
116.     gray[dst==255]=[137]
117.
118.     gray = numpy.uint8(gray)
119.     return num_corners, gray
120.
121.
122. def blur_median(my_image, blur_kernel):
123.     # ksize (represente the blurring kernel size)
124.     ksize = (blur_kernel, blur_kernel)
125.     image = cv2.blur(my_image, ksize)
126.     return image
127.
128.
129. # formulat for blur gausian doesn't work as well as the blur mediam
130. def blur_gauss(my_image, blur_kernel):
```

```
131.     # ksize (represente the blurring kernel size)
132.     # we user the 2n+1 formula for obaining odd numbers.
133.     # gausian blur works with odd numbers
134.     ksize = (2*blur_kernel+1, 2*blur_kernel+1)
135.     image = cv2.GaussianBlur(my_image, ksize, 0)
136.     return image
137.
138.
139. def blur_value(my_image):
140.     value = cv2.Laplacian(my_image, cv2.CV_64F).var()
141.     return value
142.
143.
144. # ---------------------------------- MAIN ----------------------------------
145. blur_gaus_folder="BLUR_GAUS\\"
146. blur_median_folder="BLUR_MEDIAN\\"
147.
148. image_s="UNO.JPG"
149. image = cv2.imread(image_s, 0)
150.
151.
152. im1_s="RNIKON300curtainsC1.jpg"
153. im2_s="RNIKON300curtainsC2.jpg"
154. im1=cv2.imread(im1_s,0)
155. im2=cv2.imread(im2_s,0)
156. # get the blur value detected in the real images
157. blur_real1=blur_value(im1)
158. blur_real2=blur_value(im2)
159. value_mean=int(numpy.mean([blur_real1,blur_real2]))
160. print("IMAGE: "+im1_s+", BLUR: ", blur_real1)
161. print("IMAGE: "+im2_s+", BLUR: ", blur_real2)
162. print("MEAN BLUR: ", value_mean)
163. print("\n")
164.
165. # detect blur of virtual image (should be very high as the image is crips)
166. # This is a startup point to find the exact same blur as the blur found in the
167. # real image. I keep increasing the KERNEL value untul the blur detected
168. # is the same with the blur detected in the real image. When this happens, then
169. # I have a virtual image with the same level of blur as the real image. It is then
170. # time to apply noise to the virtual image so as the SNR is the same as the SNR of
171. # the real image.
172.
173. # ------------- APPLY BLUR 1st and NOISE 2nd, then CHECK blur----------------
174.
175. blur_virt=int(blur_value(image))
176. print("VIRTUAL IMAGE blur value (before BLUR): ", blur_virt)
177.
178. counter=0
179. value_blur=blur_virt
180. while value_blur > value_mean:
181.     counter+=1
182.     my_blured_image = blur_gauss(image, counter)
183.     value_blur=int(blur_value(my_blured_image))
184.
185. print("VIRTUAL IMAGE has BLUR: "+str(value_blur)+", with gausian kernel of ", counter)
186.
187. SNR_REAL=compute_snr_real(im1,im2)
188. print("REAL_SNR: ", SNR_REAL)
189.
190. # calculate the noise needed to apply to the virtual image so that it will have
191. # the same SNR as the real image
192.
193. noise_applied=100.0/SNR_REAL
194.
195.
196. SNR, noisy_image  = add_noise_get_noisy_image_get_snr(my_blured_image, noise_applied) # SNR
of the virtual image (for verification)
197. print("VIRTUAL_SNR: ", SNR)
198. check_blur = blur_value(noisy_image)
199.
```

```
200.
201. cv2.imshow("1 - VI - Blur value BEFORE blur: "+str(blur_virt)+", NO NOISE YET", image)
202. #cv2.imshow("1 - VI - Blur value AFTER blur: "+str(value_blur)+" - matching real image blur
value, NO NOISE YET", my_blured_image)
203.
204.
205. cv2.imshow("1 - VI - Apply noise to match real image SNR: "+str(round(SNR,2))+\
206.            ", BLUR value after applied noise: "+str(int(check_blur)),noisy_image)
207.
208.
209. # CORNER DETECTOR
210. returned_corner, returned_image=no_of_corners(noisy_image)
211. cv2.imshow("1 - VI - Corners detected: "+str(returned_corner),returned_image)
212.
213.
214.
215.
216.
217. # ------------- APPLY NOISE 1st and BLUR 2nd, then CHECK SNR----------------
218.
219.
220. # calculate the noise needed to apply to the virtual image so that it will have
221. # the same SNR as the real image
222.
223. noise_applied=100.0/SNR_REAL
224. SNR2, noisy_image2  = add_noise_get_noisy_image_get_snr(image, noise_applied) # SNR of the
virtual image (for verification)
225.
226. blur_virt2=int(blur_value(noisy_image2))
227. counter=0
228. value_blur2=blur_virt2
229. while value_blur2 > value_mean:
230.     counter+=1
231.     my_blured_image2 = blur_gauss(noisy_image2, counter)
232.     value_blur2=int(blur_value(my_blured_image2))
233.
234.
235. #cv2.imshow("2 - VI - Image before noise: ", image)
236. #cv2.imshow("2 - VI - Image AFTER noise with Blur: "+str(blur_virt2)+", SNR matched to
"+str(round(SNR2,2)), noisy_image2)
237.
238. cv2.imshow("2 - VI - Matched blur: "+str(value_blur2), my_blured_image2)
239.
240. # CORNER DETECTOR
241. returned_corner2, returned_image2=no_of_corners(my_blured_image2)
242. cv2.imshow("2 - VI - Corners detected: "+str(returned_corner2),returned_image2)
243.
244.
245.
246.
247.
248.
249.
250.
251.
252.
253.
254.
255.
256.
257.
258.
259.
260.
261.
262.
263.
264.
265.
266.
```

```
267.
268.
269.
270.
271.
272.
273.
274.
275.
276.
277.
278.
279.
280.
281.
282.
283.
284.
285.
286.
287.
288. ## "data_list" os fpr storring all number of corners detected, and the index
289. ## of the data list is for the blurr that is applied.
290. #data_list=[]
291. #data_list2=[]
292. #value_for_blur=[]
293. #value_for_blur2=[]
294. #index=[]
295. #for i in range(1,999,1):
296. #    index.append(i)
297. #
298. #    # here I get the quantity of blur detected for median blur
299. #    ret_image = blur_median(image, i)
300. #    value=blur_value(ret_image)
301. #    value_for_blur.append(value)
302. #
303. #    # here I get the quantity of blur detected both for gausian blur
304. #    ret_image2 = blur_gauss(image ,i)
305. #    value2=blur_value(ret_image2)
306. #    value_for_blur2.append(value)
307. #
308. #    # here I get the number of corners detected, and the image with the corners
309. #    # dotted where they were found
310. #    num_co, gray_corner_image= no_of_corners(ret_image)
311. #    num_co2, gray_corner_image2= no_of_corners(ret_image2)
312. #
313. #    data_list.append(num_co)
314. #    data_list2.append(num_co2)
315. #
316. #    print("BLUR_MEDIAN "+str(i)+", and NO of CORNERS: "+str(num_co))
317. #    print("BLUR_GAUSS "+str(i)+", and NO of CORNERS: "+str(num_co2))
318. #    print("\n")
319. #
320. #    # Here I write on the image the data found and store it in a folder
321. #    text_written="Corn: "+str(num_co)+", B_appl: "+str(i)+", B_det: "+str(round(value,4))
322. #    text_written2="Corn: "+str(num_co2)+", B_appl: "+str(i)+", B_det:
"+str(round(value2,4))
323. #
324. #    file_to_save=write_text_on_image(gray_corner_image,text_written)
325. #    file_to_save2=write_text_on_image(gray_corner_image2,text_written2)
326. #
327. #
328. #    if i<=9:
329. #        cv2.imwrite(blur_median_folder+"blur_median_00"+str(i)+".jpg",file_to_save)
330. #        cv2.imwrite(blur_gaus_folder+"blur_gaus_00"+str(i)+".jpg",file_to_save2)
331. #
332. #
333. #    elif i>=10 and i<=99:
334. #        cv2.imwrite(blur_median_folder+"blur_median_0"+str(i)+".jpg",file_to_save)
335. #        cv2.imwrite(blur_gaus_folder+"blur_gaus_0"+str(i)+".jpg",file_to_save2)
```

```
#
#    else:
#        cv2.imwrite(blur_median_folder+"blur_median_"+str(i)+".jpg",file_to_save)
#        cv2.imwrite(blur_gaus_folder+"blur_gaus_"+str(i)+".jpg",file_to_save2)
#



#import matplotlib.pyplot as plt
#plt.figure(1)
#plt.scatter(index, data_list)
#plt.title("MEDIAN BLUR versus NO of CORNERS")
#plt.xlabel("BLUR kernel")
#plt.ylabel("Number of corners detected")
#
#
#import matplotlib.pyplot as plt
#plt.figure(2)
#plt.scatter(index, value_for_blur)
#plt.title("BLUR VALUE of every blured picture")
#plt.xlabel("BLUR value")
#plt.ylabel("Number of images scanned")

#plt.figure(2)
#plt.plot(data_list2)
#plt.title("GAUSS BLUR versus NO of CORNERS")
#plt.xlabel("BLUR kernel")
#plt.ylabel("Number of corners detected")

#plt.show()

#cv2.imshow("ORIGINAL IMAGE", image)
#cv2.imshow("BLUR IMAGE", ret_image)
#cv2.imshow("ORIGINAL IMAGE WITH CORNERS DETECTED", gray_corner1)
#cv2.imshow("BLUR IMAGE WITH CORNERS DETECTED", gray_corner2)
#
#
#cv2.waitKey(0)
#cv2.destroyAllWindows()


```

## Code that checks how the corner detector works with increasing levels of blur (Version 2)

```
# The code checks how the corner detector works with increasing levels of blur.
# We then use a blur detection function. This function, if it returns low values
# it tells us that there is a lot of blur in the image. If the values returned are
# large, then the image is clear and crisp


#: TO-DO

# - check a real image and see what is the blur returned
# - then apply enough noise on a virtul image (same SNR as the real image)
# and then apply blur and see how the corner detector works
# - use ORB as a corner detector and apply the same steps as above but with the
# ORB detector operating as a corner detector
# - now check the limits of ORB and see how much noise there is needed before
# ORB stops working, and also change the contrast, the same as the simulation done with
# the corner detector before using blurr.
# - and the next step is to apply blur to the orb detector and see how this works as well.


import cv2
import scipy.stats
import numpy
import math


def add_noise_get_noisy_image_get_snr(my_image, factor, mean=0.0, sd=1.0, seed=None):
    # Read in the image and convert to float.  Duplicate it and add noise.
    im1 = my_image
    im1 = im1.astype("float32")
    im2 = im1.copy()

    sd=im1.std(ddof=0)
    percentage_of_noise_added=sd*factor/100.0

    im1 = numpy.add(im1, numpy.random.normal (mean, percentage_of_noise_added, im1.shape))
    im2 = numpy.add(im2, numpy.random.normal (mean, percentage_of_noise_added, im2.shape))

    noisy_for_display = numpy.absolute(im1) # we do not allow any negative nubmers
    # (corresponding to the level of gray) in the image (otherwise we will not be able to
display properly)

    # If the numbers are larger than 255, then when converting to uint8, the numvbers
    # will be (that number larger than 255)-255, and this will yield numbers close to 0.
    # The idea is to limit numbers larger than 255 to 255.

    for i in range(0,noisy_for_display.shape[0],1):
        for j in range(0,noisy_for_display.shape[1],1):
            if noisy_for_display[i][j]>255:
                temp=noisy_for_display[i][j]-255  #eg. 256.345-255= 1.34500
                noisy_for_display[i][j]=255-temp

    noisy_for_display = numpy.uint8(noisy_for_display)

    # Compute the correlation coefficient and SNR.
    r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
    #print ("r:", r)
    snr = math.sqrt (r / (1 - r))
    #print ("SNR:", snr)
    return snr, noisy_for_display


def compute_snr_real(my_image1, my_image2):
```

```
 62.
 63.     im1 = my_image1.astype("float32")
 64.     im2 = my_image2.astype("float32")
 65.
 66.     # Compute the correlation coefficient and SNR.
 67.     r = scipy.stats.pearsonr (im1.ravel(), im2.ravel())[0]
 68.     #print ("r:", r)
 69.     snr = math.sqrt (r / (1 - r))
 70.     #print ("SNR:", snr)
 71.
 72.     return snr
 73.
 74.
 75. def write_text_on_image(my_file, TEXT):
 76.     x_coord=my_file.shape[0]
 77.     y_coord=my_file.shape[1]
 78.     #bits_of_color=my_file.shape[2]
 79.
 80.     # Write some Text
 81.     font                   = cv2.FONT_HERSHEY_SIMPLEX
 82.     bottomLeftCornerOfText = (10,y_coord-200)
 83.     fontScale              = 0.8
 84.     fontColor              = (255,255,255)
 85.     thickness              = 3
 86.     lineType               = 2
 87.
 88.     cv2.putText(my_file,TEXT,
 89.                 bottomLeftCornerOfText,
 90.                 font,
 91.                 fontScale,
 92.                 fontColor,
 93.                 thickness,
 94.                 lineType)
 95.
 96.     return my_file
 97.
 98.
 99. def no_of_corners(my_image):
100.
101.     gray = numpy.float32(my_image) # convert to float for data manipulation as Haris
detector needs float
102.
103.     dst = cv2.cornerHarris(gray,2,3,0.04)
104.
105.     treshold_value=0.01 * dst.max() #getting a treshold value
106.     tresh_val_used, dst = cv2.threshold(dst,treshold_value,255,0)
107.
108.     dst = numpy.uint8(dst) # the command below needs integer numbers
109.     ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
110.     num_corners = labels.max() # corners being detected
111.
112.     #result is dilated for marking the corners, not important
113.     dst = cv2.dilate(dst,None)
114.
115.     # place gray dots on the image to show where the corners are
116.     gray[dst==255]=[137]
117.
118.     gray = numpy.uint8(gray)
119.     return num_corners, gray
120.
121.
122.
123. def blur_median(my_image, blur_kernel):
124.     # ksize (represente the blurring kernel size)
125.     ksize = (blur_kernel, blur_kernel)
126.     image = cv2.blur(my_image, ksize)
127.     return image
128.
129.
130. # formulat for blur gausian doesn't work as well as the blur mediam
```

```
131. def blur_gauss(my_image, blur_kernel):
132.     # ksize (represente the blurring kernel size)
133.     # we user the 2n+1 formula for obaining odd numbers.
134.     # gausian blur works with odd numbers
135.     ksize = (2*blur_kernel+1, 2*blur_kernel+1)
136.     image = cv2.GaussianBlur(my_image, ksize, 0)
137.     return image
138.
139.
140. def blur_value(my_image):
141.     value = cv2.Laplacian(my_image, cv2.CV_64F).var()
142.     return value
143.
144.
145. # ---------------------------------- MAIN ----------------------------------
146. blur_gaus_folder="BLUR_GAUS\\"
147. blur_median_folder="BLUR_MEDIAN\\"
148.
149. image_s="UNO.JPG"
150. image = cv2.imread(image_s, 0)
151.
152. im1_s="RNIKON300curtainsC1.jpg"
153. im2_s="RNIKON300curtainsC2.jpg"
154. im1=cv2.imread(im1_s,0)
155. im2=cv2.imread(im2_s,0)
156. # get the blur value detected in the real images
157. blur_real1=blur_value(im1)
158. blur_real2=blur_value(im2)
159. value_mean=int(numpy.mean([blur_real1,blur_real2]))
160. print("IMAGE: "+im1_s+", BLUR: ", blur_real1)
161. print("IMAGE: "+im2_s+", BLUR: ", blur_real2)
162. print("MEAN BLUR: ", value_mean)
163. print("\n")
164.
165. # detect blur of virtual image (should be very high as the image is crips)
166. # This is a startup point to find the exact same blur as the blur found in the
167. # real image. I keep increasing the KERNEL value untul the blur detected
168. # is the same with the blur detected in the real image. When this happens, then
169. # I have a virtual image with the same level of blur as the real image. It is then
170. # time to apply noise to the virtual image so as the SNR is the same as the SNR of
171. # the real image.
172.
173. # ------------- APPLY BLUR 1st and NOISE 2nd, then CHECK blur----------------
174.
175. blur_virt=int(blur_value(image))
176. print("VIRTUAL IMAGE blur value (before BLUR): ", blur_virt)
177.
178. counter=0
179. value_blur=blur_virt
180. while value_blur > value_mean:
181.     counter+=1
182.     my_blured_image = blur_gauss(image, counter)
183.     value_blur=int(blur_value(my_blured_image))
184.
185. print("VIRTUAL IMAGE has BLUR: "+str(value_blur)+", with gausian kernel of ", counter )
186.
187. SNR_REAL=compute_snr_real(im1,im2)
188. print("REAL_SNR: ", SNR_REAL)
189.
190. # calculate the noise needed to apply to the virtual image so that it will have
191. # the same SNR as the real image
192.
193. noise_applied=100.0/SNR_REAL
194. SNR, noisy_image  = add_noise_get_noisy_image_get_snr(my_blured_image, noise_applied) # SNR
of the virtual image (for verification)
195. print("VIRTUAL_SNR: ", SNR)
196. check_blur = blur_value(noisy_image)
197.
198.
199. #cv2.imshow("1 - VI - Blur value BEFORE blur: "+str(blur_virt)+", NO NOISE YET", image)
```

```
200. ##cv2.imshow("1 - VI - Blur value AFTER blur: "+str(value_blur)+" - matching real image
blur value, NO NOISE YET", my_blured_image)
201. #
202. #
203. #cv2.imshow("1 - VI - Apply noise to match real image SNR: "+str(round(SNR,2))+\
204. #           ", BLUR value after applied noise: "+str(int(check_blur)),noisy_image)
205. #
206. #
207. ## CORNER DETECTOR
208. #returned_corner, returned_image=no_of_corners(noisy_image)
209. #cv2.imshow("1 - VI - Corners detected: "+str(returned_corner),returned_image)
210.
211.
212.
213.
214.
215. # ------------- APPLY NOISE 1st and BLUR 2nd, then CHECK SNR----------------
216.
217.
218. # calculate the noise needed to apply to the virtual image so that it will have
219. # the same SNR as the real image
220.
221. #noise_applied=100.0/SNR_REAL
222. #SNR2, noisy_image2  = add_noise_get_noisy_image_get_snr(image, noise_applied) # SNR of the
virtual image (for verification)
223. #
224. #blur_virt2=int(blur_value(noisy_image2))
225. #counter=0
226. #value_blur2=blur_virt2
227. #while value_blur2 > value_mean:
228. #    counter+=1
229. #    my_blured_image2 = blur_gauss(noisy_image2, counter)
230. #    value_blur2=int(blur_value(my_blured_image2))
231. #
232. #
233. ##cv2.imshow("2 - VI - Image before noise: ", image)
234. ##cv2.imshow("2 - VI - Image AFTER noise with Blur: "+str(blur_virt2)+", SNR matched to
"+str(round(SNR2,2)), noisy_image2)
235. #
236. #cv2.imshow("2 - VI - Matched blur: "+str(value_blur2), my_blured_image2)
237. #
238. ## CORNER DETECTOR
239. #returned_corner2, returned_image2=no_of_corners(my_blured_image2)
240. #cv2.imshow("2 - VI - Corners detected: "+str(returned_corner2),returned_image2)
241.
242.
243.
244.
245. ## large image
246. image3=cv2.imread("UNO.JPG", 0)
247. image3=cv2.resize(image3,(4000,int(4000.0/1.33)))
248. image3=blur_median(image3, 21)
249.
250.
251. returned_corner3, returned_image3=no_of_corners(image3)
252. cv2.imshow("2 - VI - Corners detected: "+str(returned_corner3),returned_image3)
253.
254.
255.
256.
257.
258.
259.
260.
261.
262.
263.
264.
265.
266.
```

```
267.
268.
269.
270.
271.
272.
273.
274.
275.
276.
277.
278.
279.
280.
281.
282.
283.
284.
285.
286.
287.
288.
289.
290.
291.
292.
293.
294.
295.
296. ## "data_list" os fpr storring all number of corners detected, and the index
297. ## of the data list is for the blurr that is applied.
298. #data_list=[]
299. #data_list2=[]
300. #value_for_blur=[]
301. #value_for_blur2=[]
302. #index=[]
303. #for i in range(1,999,1):
304. #    index.append(i)
305. #
306. #    # here I get the quantity of blur detected for median blur
307. #    ret_image = blur_median(image, i)
308. #    value=blur_value(ret_image)
309. #    value_for_blur.append(value)
310. #
311. #    # here I get the quantity of blur detected both for gausian blur
312. #    ret_image2 = blur_gauss(image ,i)
313. #    value2=blur_value(ret_image2)
314. #    value_for_blur2.append(value)
315. #
316. #    # here I get the number of corners detected, and the image with the corners
317. #    # dotted where they were found
318. #    num_co, gray_corner_image= no_of_corners(ret_image)
319. #    num_co2, gray_corner_image2= no_of_corners(ret_image2)
320. #
321. #    data_list.append(num_co)
322. #    data_list2.append(num_co2)
323. #
324. #    print("BLUR_MEDIAN "+str(i)+", and NO of CORNERS: "+str(num_co))
325. #    print("BLUR_GAUSS "+str(i)+", and NO of CORNERS: "+str(num_co2))
326. #    print("\n")
327. #
328. #    # Here I write on the image the data found and store it in a folder
329. #    text_written="Corn: "+str(num_co)+", B_appl: "+str(i)+", B_det: "+str(round(value,4))
330. #    text_written2="Corn: "+str(num_co2)+", B_appl: "+str(i)+", B_det:
"+str(round(value2,4))
331. #
332. #    file_to_save=write_text_on_image(gray_corner_image,text_written)
333. #    file_to_save2=write_text_on_image(gray_corner_image2,text_written2)
334. #
335. #
```

```
336. #    if i<=9:
337. #        cv2.imwrite(blur_median_folder+"blur_median_00"+str(i)+".jpg",file_to_save)
338. #        cv2.imwrite(blur_gaus_folder+"blur_gaus_00"+str(i)+".jpg",file_to_save2)
339. #
340. #
341. #    elif i>=10 and i<=99:
342. #        cv2.imwrite(blur_median_folder+"blur_median_0"+str(i)+".jpg",file_to_save)
343. #        cv2.imwrite(blur_gaus_folder+"blur_gaus_0"+str(i)+".jpg",file_to_save2)
344. #
345. #    else:
346. #        cv2.imwrite(blur_median_folder+"blur_median_"+str(i)+".jpg",file_to_save)
347. #        cv2.imwrite(blur_gaus_folder+"blur_gaus_"+str(i)+".jpg",file_to_save2)
348. #
349.
350.
351.
352.
353. #import matplotlib.pyplot as plt
354. #plt.figure(1)
355. #plt.scatter(index, data_list)
356. #plt.title("MEDIAN BLUR versus NO of CORNERS")
357. #plt.xlabel("BLUR kernel")
358. #plt.ylabel("Number of corners detected")
359. #
360. #
361. #import matplotlib.pyplot as plt
362. #plt.figure(2)
363. #plt.scatter(index, value_for_blur)
364. #plt.title("BLUR VALUE of every blured picture")
365. #plt.xlabel("BLUR value")
366. #plt.ylabel("Number of images scanned")
367.
368. #plt.figure(2)
369. #plt.plot(data_list2)
370. #plt.title("GAUSS BLUR versus NO of CORNERS")
371. #plt.xlabel("BLUR kernel")
372. #plt.ylabel("Number of corners detected")
373.
374. #plt.show()
375.
376. #cv2.imshow("ORIGINAL IMAGE", image)
377. #cv2.imshow("BLUR IMAGE", ret_image)
378. #cv2.imshow("ORIGINAL IMAGE WITH CORNERS DETECTED", gray_corner1)
379. #cv2.imshow("BLUR IMAGE WITH CORNERS DETECTED", gray_corner2)
380. #
381. #
382. #cv2.waitKey(0)
383. #cv2.destroyAllWindows()
384.
385.
```

## Program that test ORB feature matching

```
 1. import numpy
 2. import cv2
 3.
 4.
 5.
 6. def return_most_common_key_points(list_of_keypoints, flag=-1, text="INDEX"):
 7.
 8.     # return dictionary with how many time a specific element appears in a list,
 9.     # together with the element itself
10.
11.     def countOccurrence(a):
12.       k = {}
13.       for j in a:
14.         if j in k:
15.           k[j] +=1
16.         else:
17.           k[j] =1
18.       return k
19.
20.     # ----------------------
21.     # let's detect how many octaves we have based on iterating every keypoint
22.     # and finding whicj octave it belongs to.
23.     # The index of the variable "octave_indexing" represents the octaves
24.     # found by the "orb.detect" instruction. And the value that is indexed is the number
25.     # of keypoints detected by ORB in each octave. This helps us refine the keypoints
26.     # and choose to display one octave of the many that have detected an identical
27.     # number of points
28.     octave_indexing=[]
29.     index=list_of_keypoints[0].octave # i need to start with the first position in the
octave
30.     counter=0
31.
32.     for i in list_of_keypoints:
33.         if index == i.octave:
34.             counter+=1
35.             index=i.octave
36.         else:
37.             octave_indexing.append(counter)
38.             counter=0
39.             counter+=1
40.             index=i.octave
41.
42.
43.     # the "FOR" loop above has problems with the last element. It will not add
44.     # the last octave in the "octave_indexing" automatically, so I will add it
45.     # manually with the instruction below
46.     octave_indexing.append(counter)
47.     counter=0
48.
49.     # at this point we have "octave_indexing" with indexes refering to the octaves,
50.     # and the values correspondent to the number of keypoints detected in each octave.
51.
52.     # next we will count how many times we detect the exact no of keypoints across all
octaves.
53.     # This will show that there will be a certain no of octaves that will have detected the
exact
54.     # number of points. This means that these octaves, by the fact that they are a
majority,
55.     # will have detected the real number of keypoints with greater accuracy than minority
octaves.
56.     # So we just count how many times we have the same number of points, or otherwiwe said
how many octaves
57.     # have the same number of points detected
58.     result_dict=countOccurrence(octave_indexing)
59.
```

```
    # the value returned is the most common number of points by calculating the maximum
value
    # dictated by how many octaves contain that number of points. Basically if 4 octaves
    # have 3 points, and 2 octaves jave 5 points, and 6 octaves have 4 points, then 6
octaves wins.
    # this is because 6 is greater than 4 and 3, 6 is a majority. It means that tere are
    # more octaves that have detected the same number of keypoints than in any other
octave.
    # So then we will only work with the points from this dominating octave. In this octave
    # it is most likely that the ORB operator has the best accuracy only due to the idea
that
    # the majority wins, or the majority is correct, or the majority is more accurate than
the
    # minority.
    get_max=max(result_dict)

    # now we will create a list that contains all the octaves that have the highest
    # number of points detected. Otherwise said, if we discovered that we have
    # 6 octaves with 4 points, we will find a list where we will store that octave number
    # of all the octaves that have 4 points
    index_list=[]
    for i in range(0,len(octave_indexing),1):
        if octave_indexing[i]==get_max:
            index_list.append(i)

    # Now I will use these indexes to create a list with those specific keypoints
    # from each octave of this list. At this point I know all the octaves that are of
interest.
    # They are the octaves that are the greatest majority in detecting the same number of
points.

    key_list=[]

    # here I give the option to select a specific octave whose KeyPoints are to be returned
    if flag!=-1:
        if text=="OCTAVE":
            for data in list_of_keypoints:
                # the flag specifies the specific octave that I want to use
                # In order to know the specific octave, I will have had to call
                # this function with the arguments necessary to show me all the majoritary
                # octaves and all the keypoints
                if data.octave == flag:
                    key_list.append(data)

        if text=="INDEX":
            for data in list_of_keypoints:
                # here teh flag specifies which the index of the list where
                # the number of winning octaves are stored
                if data.octave == index_list[flag]:
                    key_list.append(data)
    # here I return all the octaves and their KeyPoints.
    else:
        for data in list_of_keypoints:
            if data.octave in index_list:
                key_list.append(data)

    # on top of the keypoints this also returns the rest of the octaves which have
    # the same number of corners detected, for study. This is a list whose values
    # represents the octaves which can later be used to iterate through them
    # to study them. If a specific octave is specified ("flag" holding
    # the octave number and "text" specifying "OCTAVE), then only the keypoints
    # in that octave will be returned. if "INDEX" is specified, then the KeyPoints
    # from the octave found at the index specified in "flag" will be returned. This is
    # to allow some flexibility of study
    if flag!=-1:
        return key_list
    else:
        return key_list, index_list


```

```
123.
124. from matplotlib import pyplot as plt
125.
126. # create black image
127. img3=numpy.zeros((473,633,3))
128. img3=numpy.uint8(img3)
129. #cv2.circle(img3,(320,240),4,(0,0,255),1)
130.
131.
132.
133. img = cv2.imread('UNO.jpg',0)
134.
135. # Initiate ORB detector
136. orb = cv2.ORB_create(200)
137.
138. # find the keypoints with ORB
139. kp = orb.detect(img,None)
140.
141.
142.
143. # return all keypoints from the most numerous octaves that are detecting
144. # the same number of KeyPoints. (Majority must be right - RULE)
145.
146. key_poiunts=return_most_common_key_points(kp,-1)
147.
148.
149. # ------------------
150. # In the code below I will use only those keypoints in the 2nd octave.
151. # This will give me exactly 4 corners because I checked before and found that this octave
152. # has 4 corners detected. Each octave seems to represent a pyramid layer.
153. # As far as I understand this pyramid layer is a specific scale of the image that the
154. # ORB detector does. ORB scales the original image a few times on order to detect points
155. # at every scaling and then it uses these points
156.
157. # list of my own selected keypoints
158. #list_keyp=[]
159. #
160. #for elements in kp:
161. #    if elements.octave==2:
162. #        list_keyp.append(elements)
163. #
164. ## -------------------
165. #
166. kp=key_poiunts[0]
167.
168. # compute the descriptors with ORB
169. kp, des = orb.compute(img, kp)
170.
171. # plot circules just to test if key points contain the coordinates of the points
172. # which I found out that they DO
173. for point in kp:
174.     x=int(point.pt[0])
175.     y=int(point.pt[1])
176.     cv2.circle(img3,(x,y),4,(0,0,255),1)
177.
178.
179. # draw only keypoints location,not size and orientation
180. img2 = cv2.drawKeypoints(img, kp, None, color=(0,255,0), flags=0)
181.
182. # draw keypoints with size information
183. img4 = cv2.drawKeypoints(img, kp, None, color=(0,255,255),
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
184.
185. cv2.imshow("IMAGE", img2)
186. cv2.imshow("BLACK", img3)
187. cv2.imshow("COMPEX", img4)
188. cv2.waitKey(0)
189. cv2.destroyAllWindows()
190.
191.
```

```
192.
193. # The code below is to plot the data found inside every keypoint in order to understand
194. # what does it all mean
195.
196. #poo_size=[]
197. #poo_response=[]
198. #poo_octave=[]
199. #poo_angle=[]
200. #for point in kp:
201. #    poo_size.append(point.size)
202. #    poo_response.append(point.response)
203. #    poo_angle.append(point.angle)
204. #    poo_octave.append(point.octave)
205. #
206. #plt.figure(1)
207. ##plt.plot(poo_size)
208. ##plt.plot(poo_octave)
209. ##plt.plot(poo_angle)
210. ##plt.plot(poo_response)
211. #plt.title("SNR vs NUMBER OF CORNERS")
212. #plt.xlabel("SNR")
213. #plt.ylabel("Number of corners detected")
214. #plt.show()
215.
216.
217. #plt.imshow(img2), plt.show()
218.
```

## Code that applies rotation and scale to an image and generates JPGs for each scale

```
 1. # code to apply roation and scale to an image and genereta JPGs for each scale
 2. # and rotation.
 3.
 4. import cv2
 5. import numpy
 6.
 7. def return_most_common_key_points(list_of_keypoints, first=-1):
 8.
 9.     # return dictionary with how many time a specific element appears in a list,
10.     # together with the element itself
11.
12.     def countOccurrence(a):
13.       k = {}
14.       for j in a:
15.         if j in k:
16.           k[j] +=1
17.         else:
18.           k[j] =1
19.       return k
20.
21.     # ----------------------
22.     # let's detect how many octaves we have based on iterating every keypoint
23.     # and finding whicj octave it belongs to.
24.     # The index of the variable "octave_indexing" represents the octaves
25.     # found by the "orb.detect" instruction. And the value that is indexed is the number
26.     # of keypoints detected by ORB in each octave. This helps us refine the keypoints
27.     # and choose to display one octave of the many that have detected an identical
28.     # number of points
29.     octave_indexing=[]
30.     index=list_of_keypoints[0].octave # i need to start with the first position in the
octave
31.     counter=0
32.
33.     for i in list_of_keypoints:
34.         if index == i.octave:
35.             counter+=1
36.             index=i.octave
37.         else:
38.             octave_indexing.append(counter)
39.             counter=0
40.             counter+=1
41.             index=i.octave
42.
43.     # the "FOR" loop above has problems with the last element. It will not add
44.     # the last octave in the "octave_indexing" automatically, so I will add it
45.     # manually with the instruction below
46.     octave_indexing.append(counter)
47.     counter=0
48.
49.     # at this point we have "octave_indexing" with indexes refering to the octaves,
50.     # and the values correspondent to the number of features detected in each octave.
51.
52.     # next we will count how many times we detect the same features in order to later
53.     # select for display those keypoints whose number of features are the most common
54.     # across all octaves
55.     result_dict=countOccurrence(octave_indexing)
56.
57.     # the value returned is the most common number of features across all octaves
58.     get_max=max(result_dict)
59.
60.     # now we will create a list that contains all the octaves that have the highest
61.     # number of features detected
```

```
    index_list=[]
    for i in range(0,len(octave_indexing),1):
        if octave_indexing[i]==get_max:
            index_list.append(i)

    # now I have the indexes of those specific octaves that agree with eachother the most
    # I will use these indexes to create a list of those specific keypoints, which are part
    # of the octaves given by the indexes of the above list

    key_list=[]

    # here I give the option to select a specific octave whose KeyPoints are to be returned
    if first!=-1:
        for data in list_of_keypoints:
            if data.octave == index_list[first]:
                key_list.append(data)
    # here I return all the octaves and their KeyPoints.
    else:
        for data in list_of_keypoints:
            if data.octave in index_list:
                key_list.append(data)


    return key_list



scaled=1

for i in range (0,90,1):  # rotate 90 degrees

    if (i<=9):
        file_to_write="ORB_DATA\\SCALE\\SCALE_"+str(scaled)+"_ROTATE_ANGLE_0"+str(i)+".jpg"
    else:
        file_to_write="ORB_DATA\\SCALE\\SCALE_"+str(scaled)+"_ROTATE_ANGLE_"+str(i)+".jpg"


    #read image in grayscale form
    img=cv2.imread("UNO_GRADIENT.jpg",0)

    # we shift second image to see if the  matching works as expected
    img2=cv2.imread("UNO_GRADIENT.jpg",0)
    # Dividing height and width by 2 to get the center of the image
    height, width = img.shape[:2]
    center = (width/2, height/2)
    # the above center is the center of rotation axis
    # use cv2.getRotationMatrix2D() to get the rotation matrix
    rotate_matrix = cv2.getRotationMatrix2D(center=center, angle=i, scale=scaled)
    # Rotate the image using cv2.warpAffine
    rotated_image = cv2.warpAffine(src=img, M=rotate_matrix, dsize=(width, height))
    img2=rotated_image

    # create the ORB detector
    orb=cv2.ORB_create(nfeatures=4000)

    #detect key points and descriptors of both images
    # a descriptor is an array of numbers. These numbers describe the feature. Each feature
has a descriptor.
    #kp, des = orb.detectAndCompute(img, None)   # None stands for a mask
    kp = orb.detect(img,None)
    kp = return_most_common_key_points(kp, -1)
    kp,des = orb.compute(img,kp)

    # kp2, des2 = orb.detectAndCompute(img2, None)

    kp2 = orb.detect(img2,None)
    kp2 = return_most_common_key_points(kp2, -1)
    kp2,des2 = orb.compute(img2,kp2)

    # Match the descriptors from image to image with BRUTE FORCE MATCHER
```

```
131.     # cross check TRUE means taking only the best match. If false it will take mopre
features, it will have more features.
132.
133.     bf=cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
134.     #matches=bf.match(des,des2)
135.     matches=bf.match(des,des2)
136.     matches=sorted(matches, key = lambda x:x.distance)  # sorting acccording to distance
137.
138.     #draw key points on the image
139.
140.     match_result=cv2.drawMatches(img, kp, img2, kp2, matches, None)
141.
142.     cv2.imwrite(file_to_write, match_result)
143.
144.     print("SCALE_"+str(scaled)+"_ROTATE_ANGLE_"+str(i))
145.
```

## Code that applies translation to an image and generetes JPGs for each translation

```
# code to apply translation to an image and genereta JPGs for each translation

import cv2
import numpy

def return_most_common_key_points(list_of_keypoints, first=-1):

    # return dictionary with how many time a specific element appears in a list,
    # together with the element itself

    def countOccurrence(a):
      k = {}
      for j in a:
        if j in k:
          k[j] +=1
        else:
          k[j] =1
      return k

    # ----------------------
    # let's detect how many octaves we have based on iterating every keypoint
    # and finding whicj octave it belongs to.
    # The index of the variable "octave_indexing" represents the octaves
    # found by the "orb.detect" instruction. And the value that is indexed is the number
    # of keypoints detected by ORB in each octave. This helps us refine the keypoints
    # and choose to display one octave of the many that have detected an identical
    # number of points
    octave_indexing=[]
    index=list_of_keypoints[0].octave # i need to start with the first position in the
octave
    counter=0

    for i in list_of_keypoints:
        if index == i.octave:
            counter+=1
            index=i.octave
        else:
            octave_indexing.append(counter)
            counter=0
            counter+=1
            index=i.octave

    # the "FOR" loop above has problems with the last element. It will not add
    # the last octave in the "octave_indexing" automatically, so I will add it
    # manually with the instruction below
    octave_indexing.append(counter)
    counter=0

    # at this point we have "octave_indexing" with indexes refering to the octaves,
    # and the values correspondent to the number of features detected in each octave.

    # next we will count how many times we detect the same features in order to later
    # select for display those keypoints whose number of features are the most common
    # across all octaves
    result_dict=countOccurrence(octave_indexing)

    # the value returned is the most common number of features across all octaves
    get_max=max(result_dict)

    # now we will create a list that contains all the octaves that have the highest
    # number of features detected
    index_list=[]
```

```
    for i in range(0,len(octave_indexing),1):
        if octave_indexing[i]==get_max:
            index_list.append(i)

    # now I have the indexes of those specific octaves that agree with eachother the most
    # I will use these indexes to create a list of those specific keypoints, which are part
    # of the octaves given by the indexes of the above list

    key_list=[]

    # here I give the option to select a specific octave whose KeyPoints are to be returned
    if first!=-1:
        for data in list_of_keypoints:
            if data.octave == index_list[first]:
                key_list.append(data)
    # here I return all the octaves and their KeyPoints.
    else:
        for data in list_of_keypoints:
            if data.octave in index_list:
                key_list.append(data)


    return key_list




# code for trnslating an image
# 195 is the value of X beyound which things get off screen

for i in range (0,195,1):

    M = numpy.float32([
     [1, 0, -abs(i)],   # this matrix tells how translation will happen, 20 pixels on
positive X axis and 50 pixels on negative Y axis
     [0, 1, int(i/2)]
    ])

    file_to_write="ORB_DATA\\TRANSLATE\\TR_X_"+str(-abs(i))+"_Y_"+str(int(i/2))+".jpg"


    #read image in grayscale form
    img=cv2.imread("UNO_GRADIENT.jpg",0)

    # we shift second image to see if the  matching works as expected
    img2=cv2.imread("UNO_GRADIENT.jpg",0)
    shifted = cv2.warpAffine(img2, M, (img2.shape[1], img2.shape[0]))
    img2=shifted


    # create the ORB detector
    orb=cv2.ORB_create(nfeatures=4000)

    #detect key points and descriptors of both images
    # a descrptios is an array of numbers. These numbers describe the feature. Each feature
has a descriptor.
    #kp, des = orb.detectAndCompute(img, None)   # None stands for a mask

    kp = orb.detect(img,None)
    kp = return_most_common_key_points(kp, -1)
    kp,des= orb.compute(img,kp)

    kp2, des2 = orb.detectAndCompute(img2, None)

    kp2 = orb.detect(img2,None)
    kp2 = return_most_common_key_points(kp2, -1)
    kp2,des2=orb.compute(img2,kp2)

    # Match the descriptors from image to image with BRUTE FORCE MATCHER
```

```
130.     # cross check TRUE means taking only the best match. If false it will take mopre
features, it will have more features.
131.
132.     bf=cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
133.     #matches=bf.match(des,des2)
134.     matches=bf.match(des,des2)
135.     matches=sorted(matches, key = lambda x:x.distance)  # sorting acccording to distance
136.
137.     #draw key points on the image
138.
139.     match_result=cv2.drawMatches(img, kp, img2, kp2, matches, None)
140.
141.     cv2.imwrite(file_to_write, match_result)
142.
143.     print("X: "+str(-abs(i))+", Y: "+str(int(i/2)))
144.
```

## Program that reads all the generated data and plots the graphs

```
# prgram to read data from file and plot on graphs


import pickle
import numpy

with open('meeting2.pkl', 'rb') as f:
    STORE_LIST = pickle.load(f)
#
NO_OF_CORNERS_dict=STORE_LIST[0]
PERCENTAGE_NOISE_dict=STORE_LIST[1]
SNR_dict=STORE_LIST[2]
CONTRAST=STORE_LIST[3]


##---------------------- BEGIN ----------------------------
#
## used to truncate the SNR_dict, where teh SNR values are so big that they impede
## a clear visulisation
#SMR_dict={}
#temp_l=[]
#for i in range(0,len(CONTRAST),1):
#    for j in range (0,len(SNR_dict[i]),1):
#        # here I take an arbitrary value of 5 for the SNR and do not allow SNR to
#        # be lower than 1
#        if SNR_dict[i][j]<5:
#            if SNR_dict[i][j]<2:
#                temp_l.append(2)
#            else:
#                temp_l.append(SNR_dict[i][j])
#        else:
#            temp_l.append(5)
#
#
#    SMR_dict[i]=temp_l
#    temp_l=[]
#
##---------------------- END ----------------------------

#SNR_dict=SMR_dict


import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
fig = plt.figure(1)
ax = fig.gca(projection='3d')

for i in range(0,len(CONTRAST),1):
    ax.plot3D(SNR_dict[i], NO_OF_CORNERS_dict[i], CONTRAST[i])

ax.set_title(" ")
ax.set_xlabel("SNR", fontsize=10)
ax.set_ylabel("NUMBER OF CORNERS", fontsize=10)
ax.set_zlabel("CONTRAST", fontsize=10)


fig = plt.figure(2)
ax = fig.gca(projection='3d')

for i in range(0,len(CONTRAST),1):
    ax.plot3D(PERCENTAGE_NOISE_dict[i], NO_OF_CORNERS_dict[i], CONTRAST[i])

ax.set_title(" ")
ax.set_xlabel("PERCENTAGE OF NOISE", fontsize=10)
ax.set_ylabel("NUMBER OF CORNERS", fontsize=10)
```

```
ax.set_zlabel("CONTRAST", fontsize=10)


# ----------------------------- BEGIN ----------------------------------------
# for each contrast, check at what level of noise we detect more than 4 corners.
# Then for each contrast, store that level of noise in a list variable. Later we can plot
# CONTRAST against this variable to see the domain where the corner detector works,
# and the domain where the corner detector doesn't work.

dic_graph={}
noises=[]
for i in range(0,len(CONTRAST),1):

    # getting the smallest number of corners detected. That should alaways be
    # the number of corners where the corner detector works well.
    min_corners=min(NO_OF_CORNERS_dict[i])
    temp_list=[]

    # here I create a list with all the values of the percentage noise applied,
    # corresponding to the minum number of corners detected. This shows the resilience
    # of the corner detector to keep detecting the real number of corners even with
    # increases in the noise applied
    for j in range(0,len(NO_OF_CORNERS_dict[i]),1):
        if min_corners==NO_OF_CORNERS_dict[i][j]:
            temp_list.append(SNR_dict[i][j])
            #temp_list.append(PERCENTAGE_NOISE_dict[i][j])



    dic_graph[i]=temp_list


    # in order to plot a 2D grapgh between the contrast and the last value of noise
    # applied which still yields the correct number of corners detected, we have to
    # select the maximum valye from the "dic_graph" dictionary for every single entry.
    # then we will have CONTRAST in the same dimension as all these maximum values from
    # each "dic_graph" entry
    noises.append(max(temp_list))

    # clear the variable for the next iteration
    temp_list=[]


fig = plt.figure(3)
ax = fig.gca()
ax.scatter(noises,CONTRAST)
ax.set_xlabel("SNR VALUES", fontsize=10)
#ax.set_xlabel("MAXIMUM NOISE ALLOWED", fontsize=10)
ax.set_ylabel("CONTRAST", fontsize=10)


# +++++++++++++++++++++ SUBPART ++++++++++++++++++++++++++++++++++
# 3D plot of hte area where the corner detector works well

fig = plt.figure(4)
ax = fig.gca(projection='3d')

for i in range(0,len(CONTRAST),1):

    counter=0
    add_axis=[]
    for j in range(0,len(dic_graph[i])):
        add_axis.append(counter)
        counter+=1

    ax.plot3D(dic_graph[i],add_axis,CONTRAST[i])

ax.set_title(" ")
ax.set_xlabel("SNR VALUES", fontsize=10)
```

```
#ax.set_xlabel("MAXIMUM NOISE ALLOWED", fontsize=10)
ax.set_ylabel("REDUNDANT INCREMENTAL AXIS", fontsize=10)
ax.set_zlabel("CONTRAST", fontsize=10)

# +++++++++++++++++++++ SUBPART ++++++++++++++++++++++++++++++++++


# ------------------------- END -----------------------------------------









# reducing the number of corners to 1 by calculating the mean of every simulation
#mean_NO_OF_CORNERS=[]
#mean_SNR=[]
#
#for i in range(0,len(CONTRAST),1):
#    mean_NO_OF_CORNERS.append(numpy.mean(NO_OF_CORNERS_dict[i]))
#    mean_SNR.append(numpy.mean(SNR_dict[i]))
#
#fig = plt.figure(3)
#ax = fig.gca()
#ax.scatter(mean_NO_OF_CORNERS,CONTRAST)
#ax.set_title(" ")
#ax.set_xlabel("AVERAGE CORNERS", fontsize=10)
#ax.set_ylabel("CONTRAST", fontsize=10)
#
#
#fig = plt.figure(4)
#ax = fig.gca()
#ax.scatter(mean_SNR,CONTRAST)
#ax.set_title(" ")
#ax.set_xlabel("AVERAGE SNR", fontsize=10)
#ax.set_ylabel("CONTRAST", fontsize=10)
#
#
#plt.show()
```