BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

ÚSTAV AUTOMATIZACE A INFORMATIKY

AUTOMATIC FORMS VALIDATION WITH RPA

AUTOMATICKÁ VALIDACE FORMULÁŘŮ METODOU RPA

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR — Bc. Filip Rákoczy

AUTOR PRÁCE

SUPERVISOR — Ing. Ladislav Dobrovský

VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF MECHANICAL ENGINEERING

Assignment Master's Thesis

Institut:	Institute of Automation and Computer Science
Student:	**Bc. Filip Rákoczy**
Degree programm:	Applied Computer Science and Control
Branch:	no specialisation
Supervisor:	**Ing. Ladislav Dobrovský**
Academic year:	2022/23

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Master's Thesis:

Automatic Forms Validation with RPA

Brief Description:

Open source tool SikuliX is often being used for automation of user interactions. Current solution used in company Thermofisher scientific which uses scripts in Jython (variant of Python) is deemed insufficient for variety of reasons. It has been decided to implement custom solution. Such solution should be better scalable, lowering maintenance cost, easily extensible with support of artificial intelligence and machine learning. The focus should be on possibility of compatibility with current scripts and also high reliability.

Master's Thesis goals:

Research of current solution (SikuliX).
Design and implementation of new solution in Python or C#.
Implementation of all methods used in current scripts.
Integration of OpenCV or other advanced computer vision framework.
Create a standalone solution including runtimes to be used without requiring installation.
Verification of implemented solution in practice (regression test automation system in TFS).
Design and possibly implement extensions for use of artificial inteligence.

Recommended bibliography:

IVANČIĆ, Lucija, Dalia SUŠA VUGEC a Vesna BOSILJ VUKŠIĆ. Robotic Process Automation: Systematic Literature Review. In: DI CICCIO, Claudio, Renata GABRYELCZYK, Luciano GARCÍA-BAÑUELOS, Tomislav HERNAUS, Rick HULL, Mojca INDIHAR ŠTEMBERGER, Andrea KŐ a Mark STAPLES, ed. Business Process Management: Blockchain and Central and Eastern Europe Forum [online]. Cham: Springer International Publishing, 2019, 2019-08-26, s. 280-295 [cit. 2022-10-19]. Lecture Notes in Business Information Processing. ISBN 978-3-030-30428-7. Dostupné z: doi:10.1007/978-3-030-30429-4_19.

OpenCV: documentation [online]. [cit. 2022-10-19]. Dostupné z: https://docs.opencv.org/4.x/

Deadline for submission Master's Thesis is given by the Schedule of the Academic year 2022/23

In Brno,

L. S.

doc. Ing. Radomil Matoušek, Ph.D. Director of the Institute	doc. Ing. Jiří Hlinka, Ph.D. FME dean

ABSTRACT

The following text deals with the issue of automated testing of user interfaces of Windows applications. Specifically, this work proposes a standalone software solution written in Python that replaces the existing SikuliX software for testing user interfaces of electron microscopes at Thermo Fisher Scientific. This work discusses the benefits and implementation details of this new solution and suggests further possible ways to extend this software. The results are tested on various applications, including the xT microscope control application developed by Thermo Fisher scientific.

ABSTRAKT

Následující text se zabývá otázkou automatizovaného testování uživatelských rozhraní aplikací pro Windows. Konkrétně tato práce navrhuje samostatné softwarové řešení napsané v jazyce Python, které nahrazuje existující software SikuliX pro testování uživatelských rozhraní elektronových mikroskopů v Thermofisher Scientific. Tato práce diskutuje o výhodách a implementačnách detailech tohoto nového řešení a jsou navrženy další možné způsoby rozšíření tohoto softwaru. Výsledky jsou testovány na různých aplikacích, včetně aplikace pro řízení mikroskopu xT od společnosti Thermofisher scientific.

KEYWORDS

RPA testing, automation testing, user interface validation, image recognition, automation testing framework, template matching, feature extracting methods, robot process automation

KLÍČOVÁ SLOVA

RPA testování, automatizované testování, validace uživatelského rozhraní, rozpoznávání obrazu, automatizační testovací framework, hledání vzorů, metody extrahování rysů, robotická automatizace procesů

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

2023

BIBLIOGRAPHIC CITATION

RÁKOCZY, Filip. *Automatic forms validation with RPA*. Brno, 2023. Available at: `https://www.vut.cz/studenti/zav-prace/detail/149350`. Master's Thesis. Brno University of Technology, Faculty of Mechanical Engineering, Institute of Automation and Computer Science, Supervised by Ing. Ladislav Dobrovský

AUTHOR'S DECLARATION

I hereby declare that this diploma thesis is my original work, which I have created independently under the guidance of my thesis supervisor using professional literature and other information sources that are cited in the bibliography.

Brno, on the 20th of May 2023 .....

Filip Rákoczy

ACKNOWLEDGEMENT

I would like to express my appreciation to the supervisor of this thesis, Ing. Ladislav Dobrovský, for his professional guidance and friendly cooperation. I would also like to extend my heartfelt thanks to my dear friends, who diligently dedicated their time and effort to proofread and correct the grammar and language of this thesis. Furthermore, I am immensely grateful to my loving partner and my dear family for their unwavering support and encouragement. Thank you.

CONTENTS

1 INTRODUCTION **15**
2 Overview - Image recognition, RPA and SikuliX **17**
2.1 SikuliX 17
2.1.1 SikuliX - architecture and usage 18
2.1.2 The integration with TFS's testing framework 19
2.2 RPA 20
2.3 Image recognition 21
2.3.1 OpenCV 22
2.3.2 Template matching 23
2.3.3 Feature extraction based methods 26
2.3.4 SIFT 27
2.4 TFS and Microscope overview 28
2.4.1 Microscope structure 29
2.4.2 Microscope controls - User interface 30
3 Custom software solution design **31**
3.1 Software environment - Python 31
3.1.1 Type-hinting and conventions 32
3.2 Standalone solutions 33
3.2.1 PyInstaller 33
3.2.2 cx_Freeze 33
3.2.3 Python embeddable package 34
3.2.4 Embeddable package solution 34
3.3 Migration to Python 3 35
3.4 Choosing user input and screen-grab libraries 36
3.4.1 User input benchmarking 36
3.5 Image recognition engine - OpenCV 38
3.5.1 Template Matching implementation 40
3.5.2 SIFT implementation 40
3.5.3 Template Matching vs. SIFT - comparison 41
3.5.4 Benchmarks of template matching modes 46
3.6 Custom software - features and architecture 49
3.6.1 Functionality 52
4 Discussion **57**
5 Conclusion **59**
BIBLIOGRAPHY **61**
SYMBOLS AND ABBREVIATIONS **67**
LIST OF FIGURES **69**

LIST OF TABLES **71**

1 INTRODUCTION

Automated testing is an integral part of software development that helps ensure the reliability and functionality of production software. It involves utilizing software to run tests on applications without human intervention. The recent trend in the process of developing enterprise applications is the creation of automated workflows that run autonomously in the cloud. This is a part of a so-called CI/CD pipeline that automatically builds, tests, and integrates code changes within a shared repository, delivers these changes to a production-ready environment, and finally deploys the code changes directly to customers [1][2].

It is common practice to test the application in layers of complexity that comprise the so-called testing pyramid. The pyramid consists of unit testing, integration testing, and functional user interface (UI) tests [3]. It's these first two layers of the pyramid that can be tested easily during deployment, as they mainly focus on the back-end application logic and are less time-consuming. Functional UI testing sits at the top of this pyramid because of the time and complexity constraints that such thorough testing requires. Hence, it is often done separately using techniques such as robot process automation (RPA). The advantage of this type of testing is that it directly simulates user interaction with the application, which ensures the correct interaction of all underlying software layers.

RPA is a technology that uses software robots (sometimes referred to as digital workers) or artificial intelligence that mimic human actions by using the keyboard and mouse controls to interact with and test application functionality [4][5]. Tools that allow such testing include Selenium [6] for web application testing and SikuliX [7] for desktop applications.

Thermo Fisher Scientific (TFS) uses the aforementioned SikuliX framework for testing as part of the application development process. Many custom Python scripts that aim to simulate customer workflows were created using the SikuliX libraries.

The aim of this thesis is to replace the SikuliX [7] framework with a new custom solution because of some of the shortcomings and complications that the SikuliX framework entails. I describe these in detail later in this work.

The new custom Python solution, under the internal work name FililiX, replaces all SikuliX functionality that the existing scripts used with a somewhat similar structure. FililiX aims for easy extendability with new functionality, simple deployment without the need to install any additional software on the tested Windows machines, and features up-to-date frameworks, mainly OpenCV for image recognition, while also preserving SikuliX's original syntax.

It is of great importance that the syntax remains largely unchanged due to the large number of existing workflow-testing scripts that rely on this layer of software. Moreover, it is required that the custom software solution be a standalone software that does not require the installation of any dependencies for its operation.

I describe the design of the new solution, discuss the methods and concepts of its inner workings and possible feature extensions, and finally verify its functionality in the regression test automation system in TFS. Furthermore, I discuss RPA software in general, selected methods of image recognition and computer vision, and give an overview of an electron microscope, its UI, and the advantages it provides in comparison with a traditional microscope.

2 Overview - Image recognition, RPA and SikuliX

2.1 SikuliX

Current functional UI testing at Thermo Fisher scientific utilizes the SikuliX framework for many tasks that include image recognition and mouse or keyboard control among others.

SikuliX is a widely used automation tool that allows you to operate anything you see on a computer screen using image recognition and keyboard or mouse actions. It is based on the idea that "What You See Is What You Script", meaning that it allows the user to utilize screenshots of GUI elements the visual content of which is used to identify and interact with UI elements on the screen. SikuliX uses OpenCV, a computer vision library, to locate images on the screen and can simulate mouse clicks, drags, keystrokes, and other user actions. It can also perform basic text recognition (OCR) [7].
SikuliX supports various scripting languages such as, Python, Ruby, JavaScript, and Java, and features an IDE that allows you to edit and run code scripts and provides an easy interface for capturing screenshots. SikuliX works on Windows, Mac, and Linux, provided that the system has a valid 64-bit Java installation. It also requires a real screen connected to the system, as it cannot operate on headless machines [8].

SikuliX supports Python scripts via Jython [9] - an implementation of Python in Java that provides Python with access to classes written in Java via a Java virtual machine and, on the other hand, it allows Java to leverage the rich set of modules and packages available in Python.

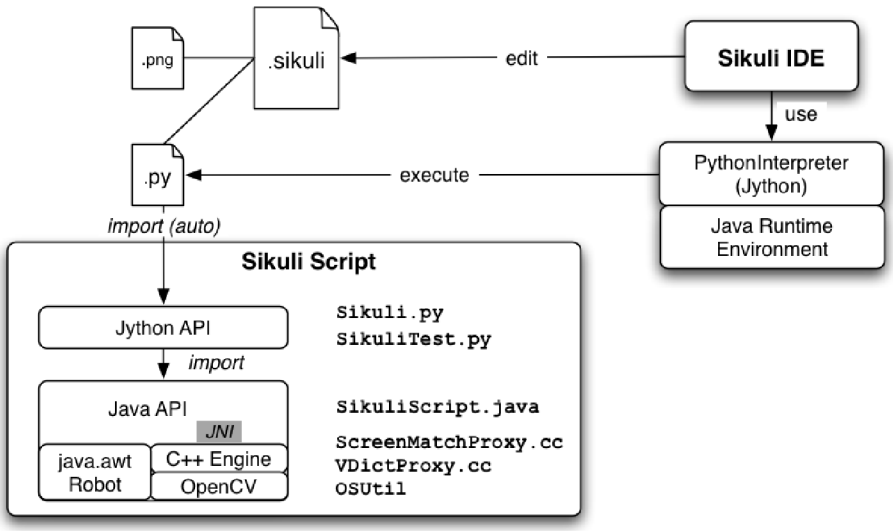


Fig. 1: Diagram of Sikuli IDE system design architecture taken from [8]

Scripts created in the SikuliX IDE have the .sikuli extension and allow users to use any scripting language supported by SikuliX to write scripts while having access to the full functionality of Java and its libraries from those scripts.

SikuliX is useful for automating tasks that are difficult to achieve using other tools, such as testing applications or web pages that are dynamic or complex, automating workflows that involve multiple applications or windows (such as copying data from one program to another) or automating repetitive tasks in routine app workflows (such as filling out forms). SikuliX is flexible in adjusting the image recognition parameters (such as similarity threshold) to deal with different resolutions, colors, fonts, etc. to handle dynamic GUIs.

2.1.1 SikuliX - architecture and usage

In this section, I give an overview of Sikuli's software architecture, focusing on classes relevant to TFS' testing. SikuliX relies on OpenCV for its image recognition tasks, namely for template matching, which serves as the main method for localizing an image in the screen or a part of the screen (a region).

The *Region* class represents the core of Sikuli's logic. A *Region* stands for a rectangular area on the screen and is defined by the location of its upper left corner (x, y) and its height and width (w, h) in pixels. The location of a region's upper left corner specifies the location in a global coordinate system whose origin is located in the upper left corner of the primary display.

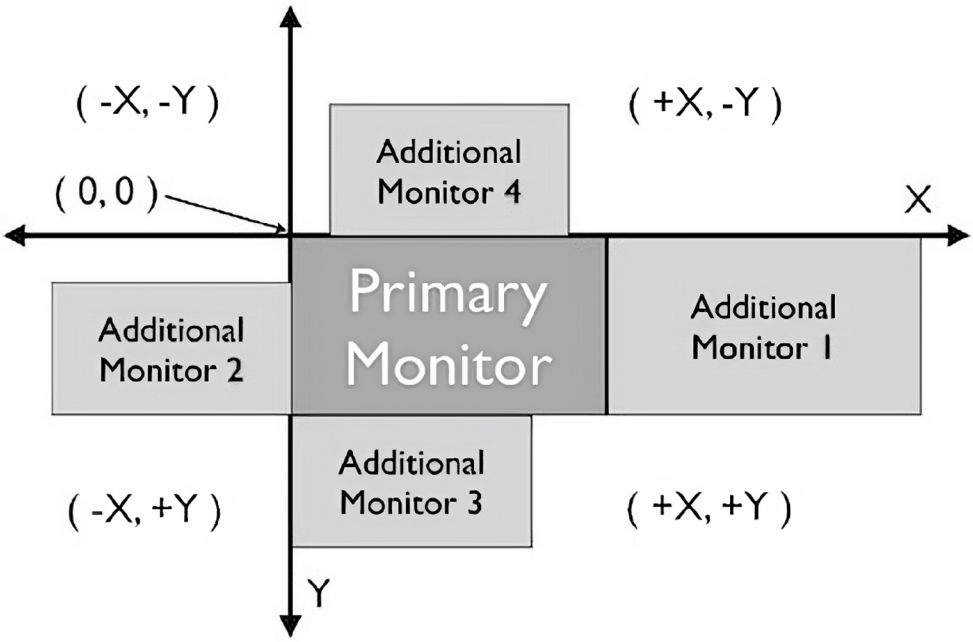


Fig. 2: Coordinate system of multi-monitor setups. Taken from [7]

A region object has no information about the contents of the screen within that region since it simply represents an area on the screen.

The *Screen* class serves as a representation of a physical monitor. More precisely, a *Screen* object is a *Region* object with the position $(0, 0)$ that has a size corresponding to the resolution of the monitor and some additional information since *Screen* inherits from *Region*. Its main purpose is to aid in the capture of screenshots and help manage open windows on different screens. It can provide information about the screen's resolution and orientation and can differentiate individual monitors based on their implicit index.

'An image that the user wants to locate on the screen is characterized using the *Pattern* class. A *Pattern* contains an image, represented as a 3-D matrix, along with the similarity attribute. Similarity is a floating-point number ranging from 0 to 1 that specifies the minimum required confidence level of an image search operation. The result of such a search operation is a *Match* object.

The *Match* class extends the *Region* class and is an encapsulation of any successful image recognition "find" operation. A match object contains the dimensions of the image used to search and the position information (x, y) where the image was found on screen, the score (similarity) that characterizes the matches' confidence in the search operation, and a target location that can be used to operate mouse actions on. Sikuli allows the user to specify a click point for a pattern object using the target offset method, which shifts the default click location from the Patterns' center. The same functionality is implemented in the *Match* object; therefore, a *Pattern* with a set target offset will yield a match whose target will be offset by the same amount. Since *Match* inherits from *Region*, all methods from the *Region* class can be used on a *Match* object.

Sikuli also features optical character recognition (OCR), that can be used to extract text from an image and convert it to a *string* and vice versa [7]

2.1.2 The integration with TFS's testing framework

Since the core of SikuliX and Sikuli IDE logic is written in Java, it requires the installation of the Java Runtime Environment (JRE). Alternatively, SikuliX can be setup with a standalone Jython package, which is an attractive option for the testing of TFS's microscopes since it is not necessary to install JRE on each individual tested machine. Despite SikuliX's flexibility, there are a few setbacks that stem from the underlying design choices.

The current testing framework requires the execution of hundreds of test scripts (written in Python) in succession as part of a test plan that can take hours to execute. An instance of Java Virtual Environment (JVM) has to be running in the background throughout these tests, and in the case where a single individual

script causes an error, the whole test plan can be interrupted as a result. Empirical testing showed the vulnerability to these crashes during long run-times.

The best course of action would then be to restart the JVM repeatedly for each individual test. This turns out to be impractical, because of the long startup and initialization times of standalone Jython and the JVM. Furthermore, Jython only supports Python 2 scripts, and there is not any work at this time towards migrating to Python 3, making future compatibility an issue and possibly limiting the prospect of extending the software. This complication, as well as the fact that an instance of JVM (or Jython) has to be running alongside the Python interpreter during execution, complicates the extendability of Sikuli, which is why the development of a replacement for Sikuli was called for.

2.2 RPA

Robotic process automation (RPA) is a broad term for a technology that aims to automate repetitive, routine, and rule-based human tasks by using software robots (workers) that interact with information systems or applications in general through existing graphical user interfaces (GUIs) or communicate with them using their Application Programming Interface (API) [10].

RPA technology can interact with the user interface of multiple applications and communicate with other systems. A large benefit of an RPA system is its capability to interconnect applications that are not designed to work in unison, acting as a middleman between them, manipulating data, responding to events, and essentially integrating the existing systems. A good example of such an integration system is Make [11]. It leverages access to a wide range of public APIs and gives users access to tools that can leverage the API information. For example, it is possible to connect Make with the back-end of an online store and setup automatic ordering of materials from suppliers after an order is placed.

According to a systematic literature review conducted by Ivančić et al. [4], UI testing is one of the most common topics addressed by academic research on RPA. UI testing is the process of form validation using robots that perform the intended tasks on the GUIs of target systems. The goal of UI testing is to detect and resolve any errors or defects that may arise during development and ensure the reliability and accuracy of enterprise solutions, where manual testing is very costly and time-consuming.

In order to enable the virtual workers to interact with an application's GUI, the software robots have to acquire information about the location of individual UI elements on the screen. This can be achieved by using a code-based approach or a vision-based system.

The code-based approach uses back-end information about the names or IDs of individual UI controls to uniquely identify them. Once identified, information about their position can be derived from their placement in the UI source tree hierarchy or directly from control's properties. On Windows, tools like Accessibility Insights [12] can be used to navigate properties of an application's UI without access to its source code. RPA tools such as Robot Framework [13] utilize this data for testing (Robot Framework also supports vision-based testing via pyautogui [14]).

Vision-based UI testing uses image recognition techniques to identify and interact with the UI elements of the application under test. An image of a correctly displayed UI control is compared and located inside a screenshot of the monitor's screen. Image recognition can be based on pixel comparison, optical character recognition (OCR), or machine learning algorithms. Vision-based UI testing therefore does not rely on access to the source code.

Each approach has their own strengths and weaknesses, and choosing one over the other depends on the specifics of the individual use case. Code-based testing is generally more precise and computationally efficient than vision-based UI testing, as it can access the internal state and logic of the application. However, it can be harder to implement and is less flexible as it can only work with applications that expose their UI elements through code or API, and it is more prone to false negatives than vision-based UI testing, as it can miss visual defects or changes that are not reflected in code.

2.3 Image recognition

In general, image recognition is a type of technology that allows machines to identify and categorize objects within a digital image or video. It is seemingly very straightforward for humans to recognize shapes, faces, and other objects with a very high level of confidence and within a wide range of varying light and viewing conditions with little to no effort. Translating this general object recognition task into a definite procedure that can be understood by a computer turns out to be very difficult. And it's precisely the varying conditions that make the recognition task much harder for a computer to understand. Selecting an appropriate image recognition technique for the UI control recognition task is integral to correct inference.

The recent development of deep neural networks (DNNs) allowed for perhaps the best attempt at a truly general machine image recognition system. Neural networks, however, are very impractical for UI testing because of the fact that every addition to the set of images that testers need the DNN to recognize would require retraining of the neural network, which is computationally very demanding and time-consuming. Techniques that do not account for such great variance in viewing

conditions are often simpler and may be better suited for UI testing since an exact copy of UI controls is easily obtainable. Feature extraction or template matching are great candidates for this task.

In this chapter, I discuss the techniques that are commonly used for these tasks, provide the relevant context, and discuss their benefits and shortcomings in relation to UI testing.

2.3.1 OpenCV

OpenCV (Open Source Computer Vision Library) is free and open-source computer vision and machine learning software designed for real-time image processing and computer vision applications. It provides a wide range of functionalities, such as image processing, face detection, object tracking, video analysis, and machine learning. OpenCV also provides support for popular machine learning libraries such as *TensorFlow, Keras,* and *PyTorch*. Image recognition applications that involve cameras for image acquisition can utilize OpenCV's camera calibration tools, which assist with the process of determining the camera's intrinsic and extrinsic imaging parameters. Camera calibration is important in computer vision applications such as 3D reconstruction, augmented reality, and robotics.

OpenCV can be used with Python using the *opencv-python* package [15] which is a wrapper of the core OpenCV logic implemented in C/C++. This package contains pre-built binaries of OpenCV that are compatible with most platforms and can therefore run on a wide array of hardware, including microcontrollers and integrated chips, which is particularly useful for computer vision applications. OpenCV is also compatible with other languages outside of Python, including Java, C++, and Matlab.

One of the most important applications of OpenCV is image recognition. It implements feature extraction image recognition methods such as SURF (Speeded-Up Robust Features), SIFT (Scale-Invariant Feature Transform), and ORB (Oriented FAST and Rotated BRIEF). SIFT is an algorithm that detects scale-invariant keypoints in an image, describes them using local histograms of gradient orientations, and matches them with keypoints in another image. SURF is a variant of SIFT that is faster and more efficient but less robust to rotation and scale changes. ORB is a binary descriptor that is similar to SURF but more efficient and robust to noise.

OpenCV provides various algorithms for template matching, such as the *matchTemplate()*, *matchShapes()*, and *templateMatching()* methods. These algorithms take an input image and a template image and return a match score or a set of matching points. The match score represents how well the template image matches a region in the input image.

2.3.2 Template matching

A digital color image is represented by multiple matrices of values, usually ranging from 0 to 255 (for 8-bit images), that represent the brightness of each of the colors at a given point in the image. A smaller image (the template, also referred to as a convolution kernel) can be located inside a larger image (the scene) by translating the kernel's position inside the scene. The core idea of template matching is based on convolution.

Two-dimensional discrete convolution is a mathematical operation that calculates the value of an output by multiplying individual pixel values for all translations of two matrices and summing the results. The value at each particular point inside the resulting matrix (heat map) corresponds to the sum of the products at a given translation point.

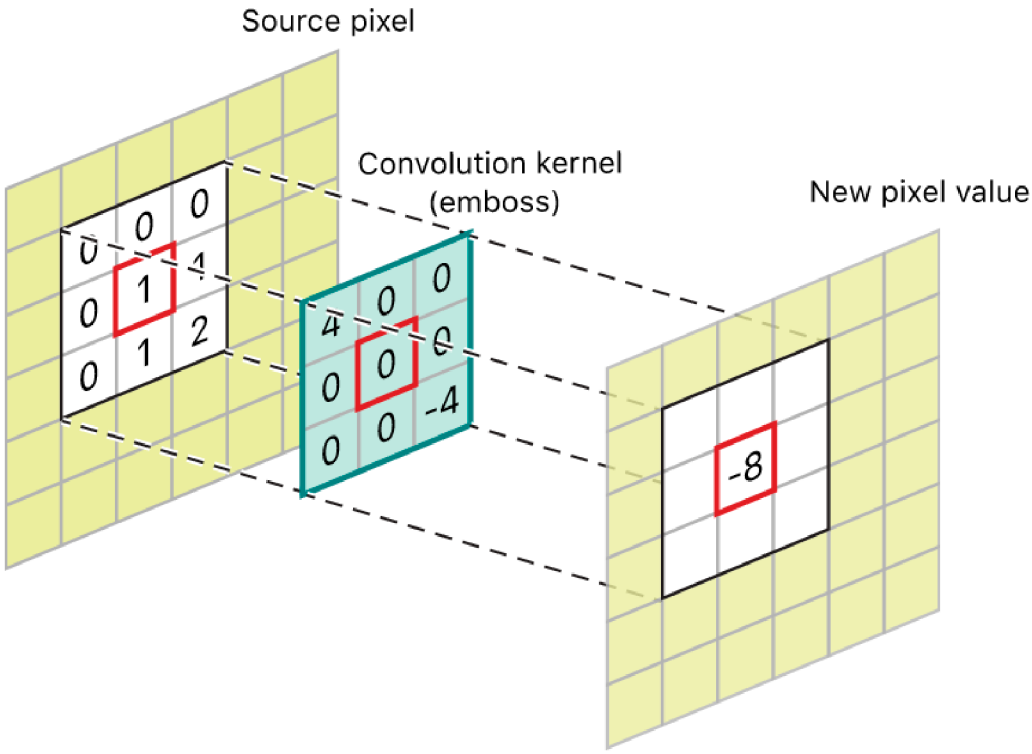


Fig. 3: Discrete 2D convolution, taken from [16]

Let the original image and the template be denoted *I(x,y)* and *T(x,y)* respectively. The resulting convoluted matrix *R(x,y)* can be expressed mathematically as [17]:

$$R(x,y) = \sum_{m=1}^{M} \sum_{n=1}^{N} T(m,n) \cdot I(x-m, y-n) \tag{1}$$

There are several ways referred to as modes of template-matching image comparison. They differ in the way in which the individual values of the resulting image are

calculated, effectively replacing the multiplication operator of the convolution equation with a different expression. The industry-standard and widely used OpenCV framework implements three of these match modes in two variants: absolute and normalized (listed further) between the 0 to 1 range representing similarity[18].

Normalized squared difference:

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{2}$$

Normalized cross-correlation:

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{3}$$

Normalized correlation coefficient:

$$R(x,y) = \frac{\sum_{x',y'}(T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}} \tag{4}$$

Where:

$$\begin{gathered} T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} T(x'',y'') \\ I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} I(x+x'',y+y'') \end{gathered} \tag{5}$$

The dimensions of the resulting matrix are dependent upon factors that include convolution stride (distance between two consecutive positions of the kernel), zero padding (number of zeros concatenated at the beginning and at the end of a matrix's axis), as well as kernel and scene shape [19]. The summation is done over the entire template and/or the image patch: $x' = 0...w-1, y' = 0...h-1$. For a template size $(w \times h)$ and scene size $(W \times H)$ assuming stride equal to one and no zero padding, the $R(x,y)$ matrix's size is: $(W-w+1)\times(H-h+1)$ [20]. Algorithm time complexity assuming no optimizations and a non-separable kernel is $\mathcal{O}(WHwhD)$ where D is matrix dimensionality ($D = 3$ for RGB images) [21].

The position of the template in the scene can be found by locating the global extreme inside the $R(x,y)$ matrix (maximum for the correlation modes and minimum in the case of the squared difference mode since the smallest difference represents the largest similarity and therefore the best match) [18].

The case where no matches are found within the scene, can be distinguished by a very low value of the global extreme and can be cut off using a threshold. This similarity checking has the added benefit of finer user control since a lower threshold will result in a more general search that also matches patterns similar

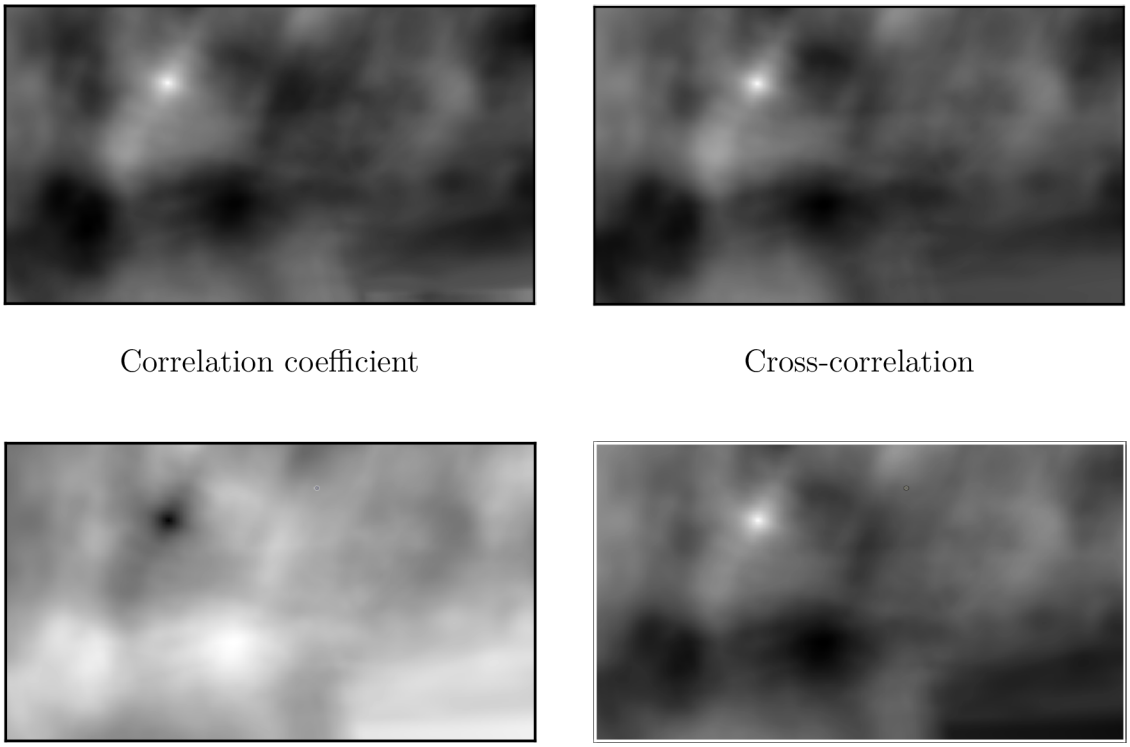


Fig. 4: $R(x, y)$ matrices for different individual modes.

to the template. On the other hand, a higher threshold value will result in a more strict search.

The previous approach, however, is limited to only finding a single instance of the template inside the scene. In the more general case where more than one template occurrence can exist in the search space, the search for a global extreme has to be replaced by a minimum similarity value search that filters out match locations from the $R(x, y)$ matrix.

The most noticeable limitation of template matching is the fact that even relatively slight differences between the template and the scene containing the template will yield unreliable results. Furthermore, template matching does not provide a straightforward way of achieving scale and rotationally invariant template searching behavior apart from simply executing the search with modified templates, which significantly impacts performance. There are techniques that involve Fast Fourier Transforms and other complex image processing techniques that allow for scale and rotation-invariant searching in a reasonable time frame [22][23]. Despite these disadvantages, template matching makes for a very compelling candidate for UI testing applications where obtaining a precise copy of UI controls in the correct orientation is not an issue.

2.3.3 Feature extraction based methods

Feature extraction is the process of finding patterns, called features, inside an image that help identify it. Features are extracted by applying a low level image processing operation that examines the local neighborhood of a pixel to determine whether a feature exists at that location. For example, a corner detector can find pixels that have high gradients in two orthogonal directions.

The selected features need to be described in a manner that is invariant to changes in scale and rotation, etc., these properties are collectively referred to as affine invariant features [24]. An object that describes these features is called a feature descriptor. It is a vector that encapsulates the appearance of the feature in relation to its' surroundings.

Feature extraction methods take an input image and return a set of keypoints and descriptors, which are used to describe the distinctive features in the image. The feature descriptor takes advantage of the defining feature of an affine transformation, which is that it preserves lines and parallelism, meaning that the relative direction of the defining features of an object remains unchanged. Images can therefore be compared by measuring the similarity between their defining features' descriptors. Methods for comparing the generated descriptors with the corresponding features in another image include algorithms such as FLANN (Fast Approximate Nearest Neighbor), the k-d tree algorithm, or brute force. [25]

Once the corresponding features are matched between an image and the scene, a suitable transformation matrix can be computed that specifies the position and orientation correction necessary to align the image within the scene [26].

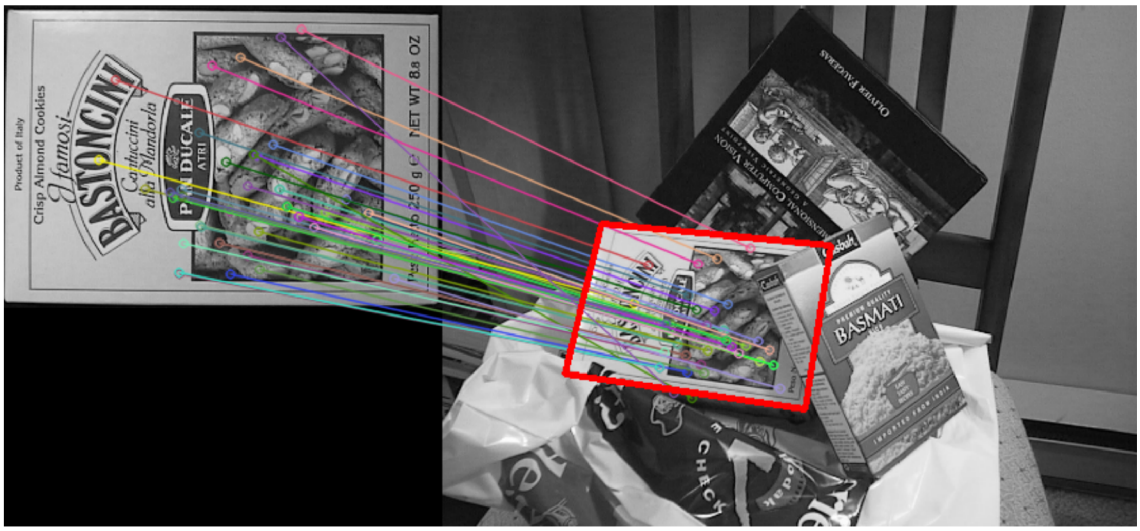


Fig. 5: Matched features between an image and template. Taken from [26]

2.3.4 SIFT

Scale-invariant feature transform (SIFT) is a feature extraction-based method invented by David Lowe in 1999 [27]. SIFT finds the appropriate feature descriptors (keypoints) by comparing images processed with Gaussian blur derived from the original image. An image with Gaussian blur applied is subtracted from another, less blurred version of the original image. This operation then acts as a band-pass filter for features sized in between the two levels of blur since each of the images acts as a low-pass filter for their respective blur strengths. Keypoints are then the minima and maxima of the differences of Gaussians [28] at multiple scales. Interestingly, neuroscience research has shown that object recognition in primates makes use of intermediate complexity features that are mostly invariant to changes in scale, location, and illumination [29] resembling the differences of Gaussians technique.

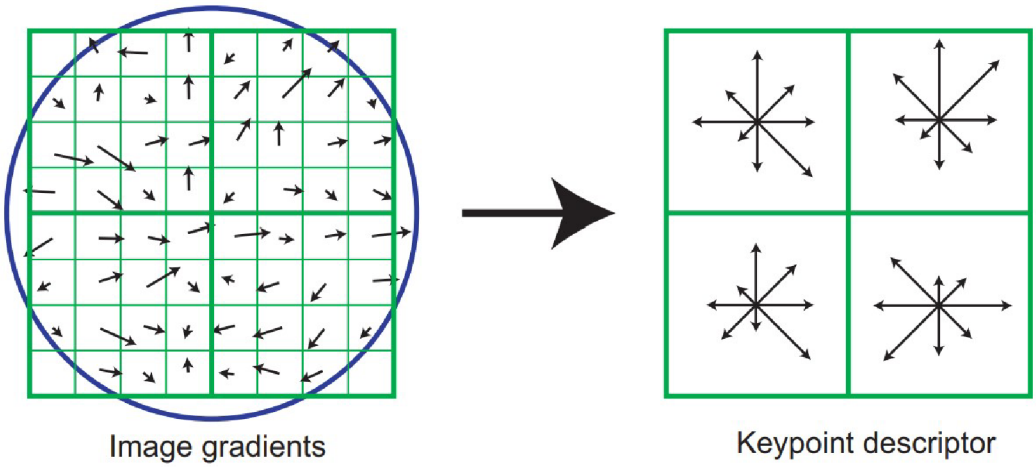


Fig. 6: Keypoint descriptor creation using Gaussian gradient histograms. Taken from [30]

After finding suitable points of interest, the SIFT algorithm assigns keypoint descriptors to each feature location. A keypoint descriptor is created by calculating gradient magnitudes in the neighborhood around a keypoint and assigning weight to these gradients based on their relative position using a Gaussian window. These are then aggregated into "orientation histograms," summarizing the features of the surroundings of a local region [30].

Each individual keypoint inside the scene, characterized by its descriptor, can then be matched to the corresponding keypoints in the template using a nearest neighbor classifier algorithm such as the best-bin-first algorithm (k-d tree modification) [31]. These are then classified using Lowe's ratio test, which filters out the features that stand out from the rest by their distance metric, which repre-

sents the closeness between the descriptors of the scene and the template. Matched features can then be used to reconstruct the position of the template in the scene.

The disadvantages of the SIFT algorithm in relation to UI testing include the fact that there isn't a reliable way of predicting SIFT's keypoint locations. Therefore, the reconstructed bounding box that represents the template's position inside the scene based on those keypoints will include slight variations in size, shape, and location. This slight difference, however small it may be, can cause an undesirable variance in behavior, which may lead to repeatability problems. Moreover, SIFT does not use color information and only relies on the template's distinctive features for matching. This means that low-contrast, feature-less images that only differ in color and/or luminosity are indistinguishable for SIFT.

2.4 TFS and Microscope overview

Thermo Fisher Scientific is a global leader in the field of scientific instrumentation, reagents, and solutions, with a focus on advancing research and innovation. One of their most prominent products is the electron microscope, which enables scientists to observe and analyze the structure and function of materials and tissues at nanoscale.

An ordinary light microscope has a theoretical maximum resolution capability of $200nm$ which corresponds to half the wavelength of blue light. An electron microscope uses a beam of high-energy electrons to create an image. The concept of using electrons for imaging stems from Louis de Broglie, who in 1924 first came up with the hypothesis of the wave properties of particles. The main benefit of an electron microscope is better resolution. Using de Broglie's wave equation:

$$\lambda = \frac{h}{p} = \frac{h}{mv} \tag{6}$$

Where λ is the wavelength of a particle, h is Planck's constant, and p the momentum of a particle. The speed of an electron v accelerated by a difference in electric potential V can be expressed as:

$$eV = \frac{1}{2}mv^2 \Rightarrow v = \sqrt{\frac{2eV}{m}} \tag{7}$$

Combining equations 1 and 2 yields:

$$\lambda = \frac{h}{\sqrt{2emV}} \tag{8}$$

Because the velocity of an electron in an electron microscope is close to the speed of light, relativistic mass increases have to be accounted for by dividing with the Lorentz factor γ resulting in:

$$\lambda = \frac{h}{\sqrt{2e\frac{m}{\gamma}V}} = \frac{hc}{\sqrt{eV(2mc^2 + eV)}} \tag{9}$$

$$\gamma = \frac{1}{1 - \frac{v^2}{c^2}} \tag{10}$$

Substituting constants and a typical accelerating voltage $V = 200kV$ yields a theoretical maximum resolution of $\lambda = 2.51pm$. This wavelength is therefore five orders of magnitude smaller than the wavelength of blue light. However, in practice, the resolution is limited to about ~$0.1nm$ (as of 2023 for commercial TEM microscopes) due to the imaging limitations of the optical system, predominantly because of spherical aberration [32][33].

2.4.1 Microscope structure

A scanning electron microscope (SEM) is a type of electron microscope that uses a beam that is scanned across the surface of the sample, similar to a CRT TV, and produces a high-resolution image.

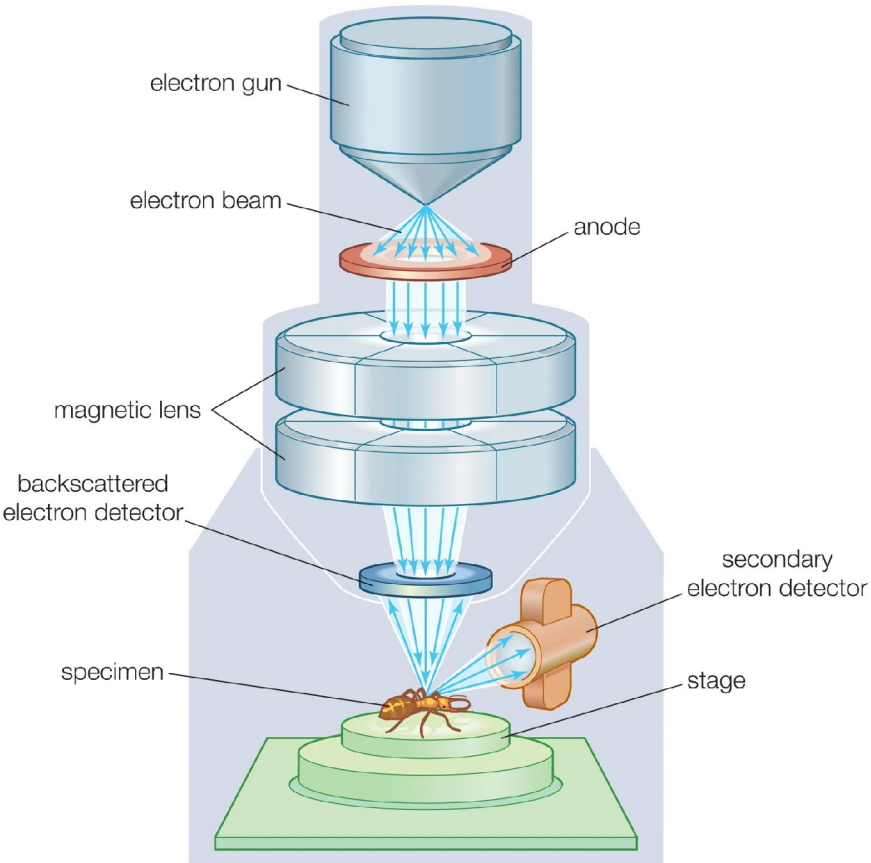


Fig. 7: Structure of a scanning electron microscope. Taken from [34]

It consists of a source of electrons called an electron gun, which usually utilizes a heated filament to generate a stream of electrons called a beam. The electrons are accelerated by an electric field inside the beam tube toward the optical system. Then a set of electromagnetic lenses focus and deflect the beam in a desired way, illuminating a spot on the sample surface. The sample is placed on a stage that can position it in the desired location with the specified orientation. The beam is continuously scanned across the sample, and data is gathered via sensors. The interaction between the electrons and the samples' surface causes an emission of high-energy electromagnetic radiation and other electrons that can be detected by the sensors. Most common among the electron sensors are the back-scattered electron detector located in the beam-accelerating tube and the secondary electron detector positioned in the sample chamber. The emitted electromagnetic radiation can be used to identify material composition.

2.4.2 Microscope controls - User interface

The microscope is equipped with a multitude of sensors, actuators, valves, and other equipment that collectively serve to operate the microscope and can all be controlled via the Microscope Control xT application. The layout of the application is illustrated in the color-coded image below.

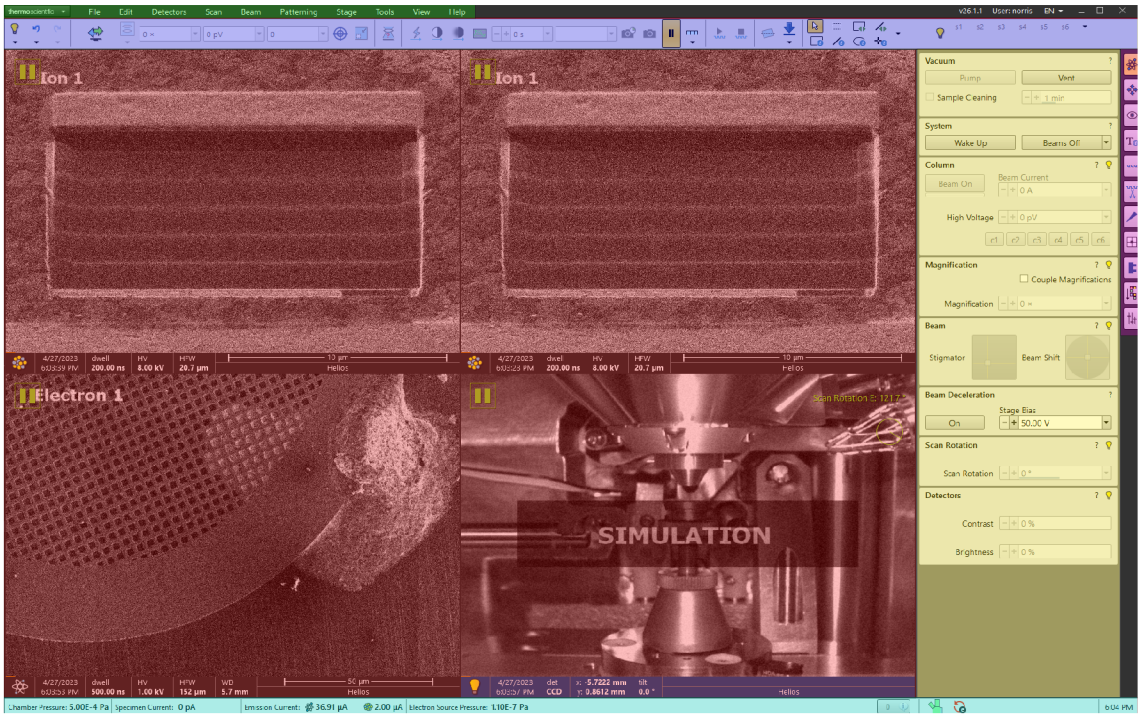


Fig. 8: xT application: Green - menu bar, Blue - Toolbar, Red - Imaging quadrants, Yellow - Pages, Pink - Page Controls, Cyan - Status bar [34]

3 Custom software solution design

Over the course of TFS's SikuliX-based testing, a large number of automation scripts have been created in order to automate the previously manual testing procedures. Along with the testing scripts, a collection of screenshots of UI controls and other features in the xT application have been captured and systematically saved in order to be used as templates for SikuliX's image recognition system. It is desirable that both the scripts and captured images stay the same because of their vast quantity. This requires that the new custom solution maintains SikuliX's syntax while replacing the inner workings of the underlying technology.

In order to appropriately replace the SikuliX software, the custom solution, internally referred to as FililiX, has to fulfill the multiple constraints that are based on the requirements for the software and compatibility with TFS's testing framework technology currently in use. In this chapter, I walk through the implementation specifics and detail the testing performed that guided the reasoning behind certain design decisions.

3.1 Software environment - Python

Part of this thesis's assignment is to design a solution in either Python or C#. The aforementioned testing scripts that used SikuliX were all written in Python which therefore creates a need for a Python interpreter anyway. As a result developing the custom software solution in Python is an obvious choice. Moreover, Python has one of the largest collections of open-source libraries (called packages) among all programming languages in existence. Python has also become the world standard for machine learning, data analysis, automation, and computer vision, which can be beneficial for future software extensions.

Python is an interpreted cross-platform language, which means that it does not need to be compiled before running, which has its own benefits and shortcomings. On the upside, the Python interpreter lets users run and test their code on the fly, and with tools such as Jupyter Notebook, [35] Python has become a very popular option for scripting. However, an interpreted language also has some drawbacks. A compiled language converts the source code into machine code that can be directly executed by the computer, whereas an interpreted language translates the source code into intermediate code that is then executed by the interpreter. This adds an extra layer of processing that slows down the performance of the program. A compiled language also performs type checking, syntax checking, and more during compilation, making it less likely that a run-time error will occur.

The performance drawbacks of Python can, however, be neglected in my case because all computation-heavy tasks (mainly image recognition) are executed using the Python wrapper of the OpenCV library. Meaning that whenever an OpenCV function is called, what is actually executed is the underlying C/C++ code, resulting in negligible performance loss.

3.1.1 Type-hinting and conventions

Python is a dynamically typed programming language, which means that the interpreter performs type checking only during runtime and that the variable is allowed to change its type over its lifetime. Despite this capability, Python has introduced type hints in PEP 484 [36] with the goal of bringing static type checking to code analysis. Type hints are essentially optional static variable types that aid with enhancing readability. They guide the user during development and enforce stricter manipulation with object types.

FililiX features type hints, leveraging their many benefits. These include better code clarity that results from providing a signature that clarifies the assumptions about function arguments and the return type. Type hints also improve the coding experience since most modern IDEs rely on type hints to statically analyze code and help uncover errors during development, and they also aid with code completion. It is, however, important to emphasize that type hints have no run-time effect and are not enforced [36][37] by Python's interpreter.

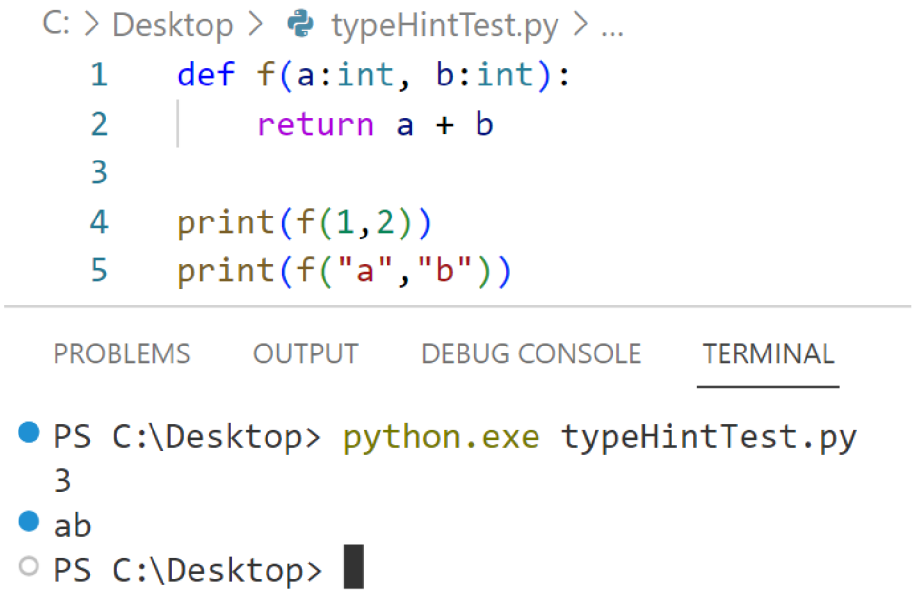


Fig. 9: Type hints do not enforce runtime variable types. Code runs without errors and yields "3" and "ab"

The source code for FililiX follows the conventions set by the PEP-8 style guide [38] for code layout, indentation, naming styles (camel case to preserve SikuliX syntax), doc-strings, and more.

3.2 Standalone solutions

A standalone (sometimes referred to as portable) software solution is a program that can run independently of other software and does not require installation on the target device or any additional components or libraries to function. A great example are operation system images, or ISO files. Because of their very nature, they have to be portable in order to transfer OS images to a new system.

To accomplish this, all FililiX's dependencies have to be gathered in one directory. That also means packaging Python itself, which by default has to be installed on the device before use. To avoid Python installation, a few options were explored, including PyInstaller [39], cx_Freeze [40] and a Python embedded package [41].

3.2.1 PyInstaller

PyInstaller is a package in Python that can be used to create standalone executable files from Python code. It has a built-in mechanism for automatically detecting and including required packages and files and wrapping them into a self-contained executable file that can be distributed and executed on any Windows platform, making it a great solution for sharing Python applications. Because of the nature of .exe files, an application packaged into a single executable in this manner also hides the application's implementation details from the user, which may be useful for security reasons when distributing to clients.

However, there are also some disadvantages. One of them is the size of the resulting executable files. Since PyInstaller bundles the Python interpreter and any required packages, the resulting executable file can be quite large. Another disadvantage is that PyInstaller may not work correctly with some Python code since proper packaging requires an analysis of the code to determine the required packages and files. The analysis process may not detect all necessary dependencies correctly, resulting in errors or missing functionality in the finalized executable. Furthermore, some multiprocessing functionality currently cannot be used with "frozen" executables, meaning binaries produced by packages like PyInstaller and cx_Freeze.

3.2.2 cx_Freeze

Cx_Freeze is a tool similar to PyInstaller with a couple of notable differences. Firstly, cx_Freeze does not support compiling into single-file executables, where

all of the libraries for your application are embedded in one executable file. Instead, it creates multiple executables and extracts their dependencies to separate folders outside of the .exe file, which produces considerable performance improvements but is not quite on par with the non-packaged software.

Cx_Freeze also suffers from the same disadvantages as PyInstaller, mainly the fact that all build dependencies may not be correctly included in the package, which causes broken functionality. Diagnosing missing packages and manually including them can be very difficult and time-consuming, which is the most common reason why developers avoid this approach.

3.2.3 Python embeddable package

The embedded distribution of Python is a minimalist Python environment intended to be a part of another application as opposed to being directly accessed by users. Upon extraction, the integrated distribution is nearly completely isolated from the end-user's system. It does not inherit any environmental variables, system registry settings, or installed packages. The Python standard library is included in precompiled and optimized .pyc files within a ZIP file, while python3.dll, python.exe and pythonw.exe are all provided separately. However, pip and the Python documentation are not included.

Embedded Python can be integrated into another application by running a process calling python.exe with the necessary command-line arguments. Dependent packages can be located in directories other than python.exe's root folder because there is an opportunity to specify the directory search path before launching the application. Native code applications often require some form of scripting language, and the embedded Python distribution can be used for this purpose. Simply extracting the embedded distribution to a sub-directory of the application installation is enough to provide a loadable Python interpreter. [41]

3.2.4 Embeddable package solution

FililiX is part of an embeddable Python package for several reasons. Firstly, it is not suitable to create multiple executables for each of FililiX's files since this would only complicate the logic of the regression test system that executes them. Therefore, cx_Freeze is not very suitable.

Using PyInstaller is an attractive option since it can be compiled into a single executable that handles all logic. However, this would call for a restructuring of the software since all functions would have to be accessible via a single file, just like an independent application, and would have to be called appropriately using the passed arguments. FililiX is rather meant to be treated as a collection of tools that can be accessed individually, just like a Python package. Compiling the en-

tire application into a single executable would also considerably increase its size, and the launch time of the executable itself is slower than executing a python.exe directly.

To evaluate the startup time difference, a simple Hello_World.py program was created and then compiled into a single executable using PyInstaller. Startup times were compared between the executable and a direct Python startup. Differences were measured using Windows PowerShell's Measure command. The executable took $2.5 \pm 0.3s$ to execute, while launching the file directly took $0.36 \pm 0.02s$.

Furthermore, in TFS's use case, the potential added benefit of hiding the application's implementation using the executable is counterproductive since the software is only used for internal testing. To install the pip package manager, the provided get-pip.py script was executed from the embedded interpreter. A package directory, Lib/site-packages, was created inside the Python root folder. This directory then had to be configured as an import site by adding the path to python311._pth file. After that, the Python environment is completely setup and can be used just like a normal Python distribution without any lack of functionality.

3.3 Migration to Python 3

All existing SikuliX-based testing scripts have been developed in Python 2 because SikuliX provides Python scripting capability via the Jython project that only supports this older version of Python for which the official support was discontinued in 2020. The upgrade to Python 3 was called for because of the lack of support, availability, and quantity of third-party libraries and upgrading provides increased security, and easier maintenance. Python 3 comes with changes to the language's syntax, and it is therefore necessary to convert all existing scripts to support the new syntax. To accomplish this, Python's official 2to3 code translation tool was used. This program reads Python 2.x source code and applies a series of so-called fixers to transform it into valid Python 3.x code[42]. A fixer is simply a pair of strings where the former represents the original code syntax and the latter is its substitute.

This tool was used on the entire project folder containing all directories with scripts that are recursively traversed for Python source code, all of which was translated. Part of the transition to Python 3 also involved the replacement of all Java libraries that these scripts used (they were supported via Jython). All the functionalities that these libraries provided, mainly clipboard manipulation and multi-monitor management, were built into FililiX.

3.4 Choosing user input and screen-grab libraries

The SikuliX software used built-in Java libraries to interact with the mouse and keyboard as well as create screen captures for its image recognition algorithms. Python, on the other hand, does not come with this capability out of the box. Therefore, an appropriate package or packages that provide this functionality are needed.

PyAutoGUI [14] is an open-sourced cross-platform library that provides support for GUI automation tasks. It includes basic controls for the mouse and keyboard and other relevant features, including the ability to capture screenshots.

Another option would be to use more low-level packages such as the *mouse* [43] and *keyboard* [44] libraries to control user inputs and the *Pillow* [45] library for capturing screenshots. These libraries are effectively the sole Python package options in their separate domains, neglecting smaller or no longer supported projects. The obvious advantage of using low-level libraries is the associated performance gain that stems from their simplicity. However, they are not as feature-rich, and they lack some fringe capabilities that *PyAutoGUI* provides.

3.4.1 User input benchmarking

In order to evaluate the potential performance benefits, a comparative benchmark was performed. The benchmark consists of a simple Python script that repeatedly performs mouse and keyboard actions, captures screenshots using *PyAutoGUI*, and compares the execution time with one of the specialized libraries on the same tasks. The differences were evaluated using the built-in timeit module.

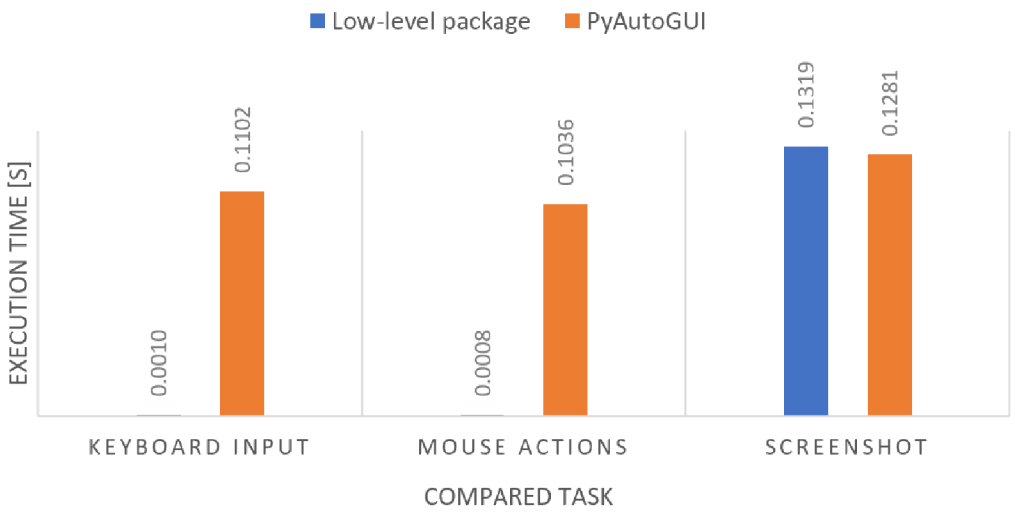


Fig. 10: Average execution time comparison between *PyAutoGUI* and low level low-level packages

The same testing was done for 100 consecutive actions for each task type. However, in the case of mouse testing, a delay between the individual actions had to be introduced in order for the individual clicks to be registered separately. Empirical testing revealed that this limit varies between operating systems. Specifically on Windows, the minimum necessary delay is about 0.03s which was implemented in the test.

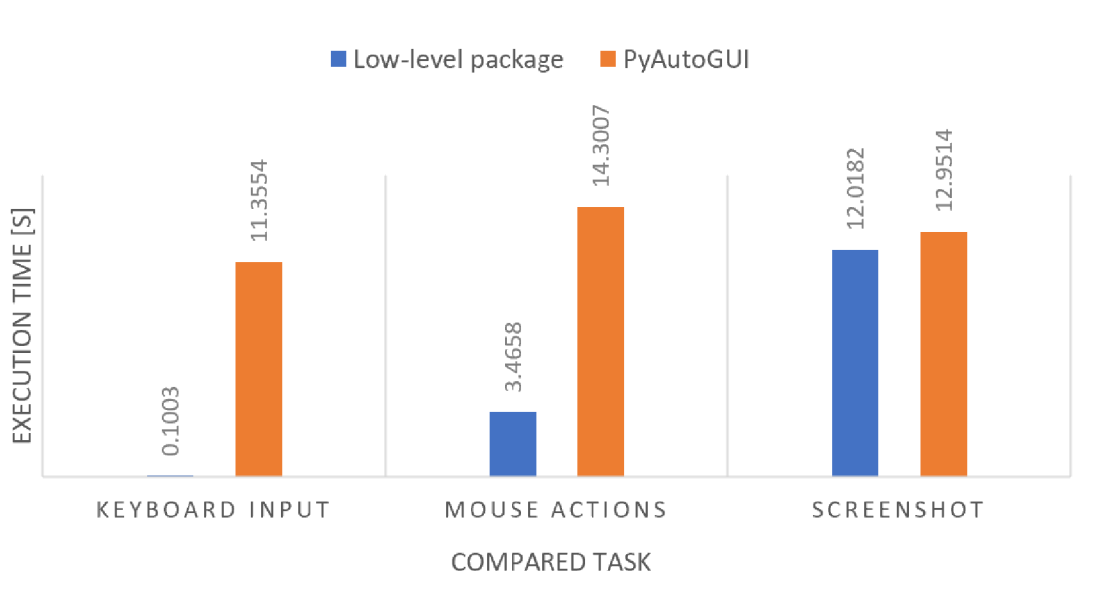


Fig. 11: Average execution time comparison for 100 consecutive actions

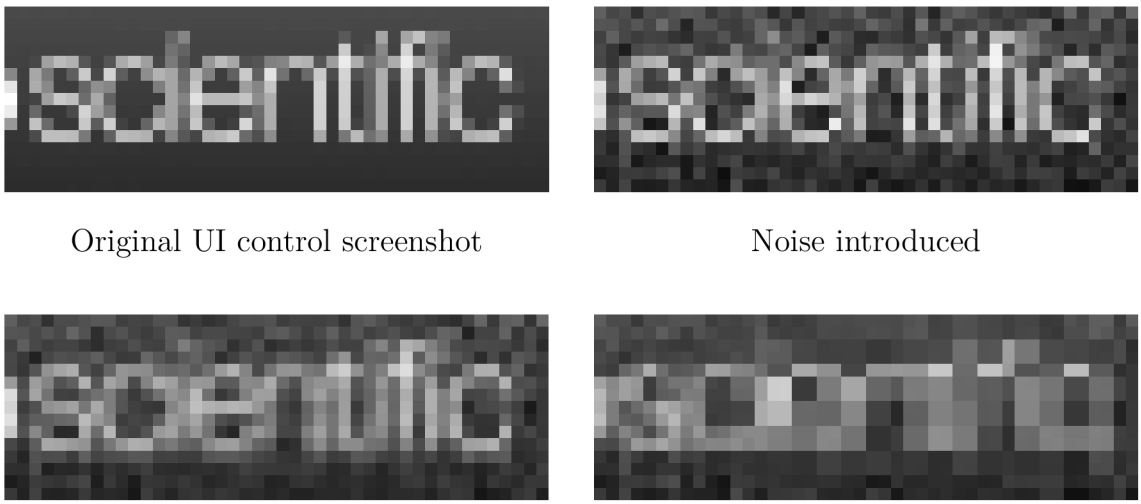
The testing showed a very high performance discrepancy when tested on individual mouse and keyboard actions. Using the lower-level libraries led to an improvement of 11265% for keyboard inputs and 12790% for mouse actions (about a 100x improvement or about 0.1s). The differences are less profound in the consecutive mouse-clicks test (413%) because of the implemented delay, while the keyboard inputs test showed similar improvements of about 11321%. Real-world performance on mouse actions would most likely fall in between the two extremes. However, the difference in time necessary to capture a screenshot is negligible. This is mainly because *PyAutoGUI* itself uses *Pillow* internally for capturing screenshots, yielding very similar results. OpenCV also has the capability to capture screenshots. However, it also uses the *Pillow* library to do so, and therefore similar results would be expected, which is why OpenCV's image capture was not included in the testing.

Because testing procedures at TFS consist of up to thousands of keyboard and mouse actions, the increased execution time at this volume adds up to several minutes of unnecessary wait time. Due to this large performance gap, it has been decided that FililiX will not utilize the*PyAutoGUI* library and will instead use the individual lower-level packages. Moreover, the empirically determined minimum consecutive mouse action delay is implemented to avoid any unregistered clicks.

3.5 Image recognition engine - OpenCV

FililiX utilizes the OpenCV library for all of its image recognition tasks and image manipulation tasks, such as color space conversions. In order to replace SikuliX's functionality, the replacement software solution has to be able to consistently locate a template inside a screenshot of the monitor's content. In order to choose the best method to do so in the context of UI testing, a comparison of two feature extraction methods, SIFT and template matching, was performed in order to evaluate their reliability and inference time. Both methods were implemented using the OpenCV library.

To benchmark and evaluate the best of these methods for the purpose of UI testing, the recognition task has to reflect the real use case. The existing collection of screenshots that are used at TFS for testing accurately reflect the shape, color, and orientation of the searched UI controls. The color information is particularly important because some UI controls can only be visually accurately differentiated by their background color because they are either feature-less (a blank color image) or their features are indistinguishable without the additional color information. However, their resolution may differ since screenshots of all UI controls are captured from a maximized window, no matter the size of the UI control. Meaning that large UI controls will be in high resolution compared to small icons or small-font text screenshots. Furthermore, the template and screenshot may slightly differ due to noise caused by image compression that occurs when capturing a screenshot. The test is therefore designed to evaluate accuracy and reliability on a small target in a large scene (which represents the worst-case scenario) with progressively increasing amounts of artificially introduced image noise and blur. Enlarged versions of the target images are shown below with various degrees of degradation.



Original UI control screenshot

Noise introduced

Noise and blur

Noise and strong blur

Fig. 12: Template screenshots with various degrees of degradation

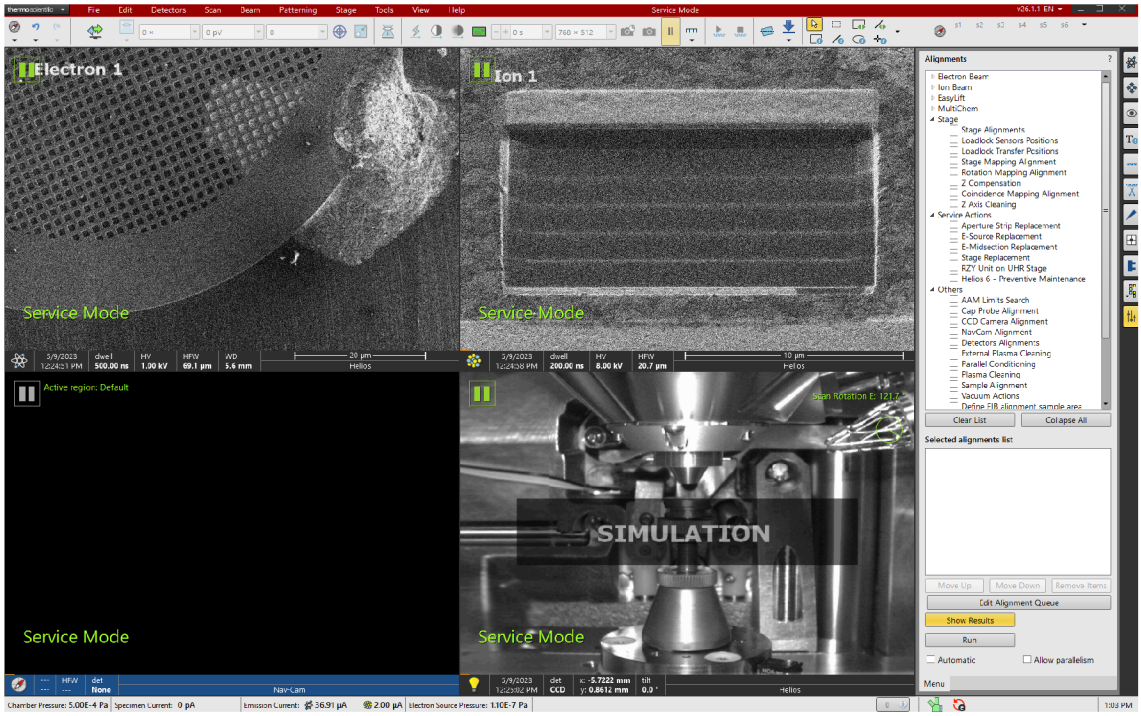


Fig. 13: Image of the searched scene

The degraded images are then used as templates for both the template matching and SIFT algorithms and are searched for in the scene. Both the template matching algorithm and SIFT were implemented using the suggestions in OpenCV's documentation [46][18].

3.5.1 Template Matching implementation

```
# Load Images
img = cv.imread('Resource/scene.png', cv.IMREAD_COLOR)
img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
template = cv.imread('Resource/template.png', cv.IMREAD_COLOR)
template = cv.cvtColor(template, cv.COLOR_BGR2RGB)
h, w, _ = template.shape

method = ['cv.TM_CCOEFF_NORMED']
for meth in method:
    method = eval(meth)

    # Apply template Matching
    res = cv.matchTemplate(img, template, method)
    min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)

    # If the method is TM_SQDIFF or TM_SQDIFF_NORMED, take minimum
    if method in [cv.TM_SQDIFF, cv.TM_SQDIFF_NORMED]:
        top_left = min_loc
    else:
        top_left = max_loc
```

Fig. 14: Template matching implementation using OpenCV in Python

The above code snippet illustrates the search procedure. First, the image and the template are loaded into memory and they are color converted to the RGB color format. Next, for the chosen mode (correlation coefficient), the template matching algorithm is executed via the *matchTemplate()* method, and the extremes from the resulting matrix are extracted using *minMaxLoc()*. The correct extreme is then selected based on the matching mode and the location of the template is identified.

3.5.2 SIFT implementation

The SIFT algorithm was implemented using the FLANN (Fast Library for Approximate Nearest Neighbors) library, which contains a collection of algorithms for fast nearest neighbor search optimized primarily for large datasets.

```python
# Load the icon and the image to search in
icon = cv2.imread('Resource/template.png')
icon = cv2.cvtColor(icon, cv2.COLOR_BGR2RGB)
image = cv2.imread('Resource/scene.png')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
w, h = icon.shape[1], icon.shape[0]

# Initialize SIFT detector and FLANN matcher
sift = cv2.SIFT_create()
flann = cv2.FlannBasedMatcher({'algorithm': 0, 'trees': 5}, {'checks': 50})

# Find keypoints and descriptors in the icon and the image
keypoints_icon, descriptors_icon = sift.detectAndCompute(icon, None)
keypoints_image, descriptors_image = sift.detectAndCompute(image, None)

# Match descriptors between icon and image
matches1 = flann.knnMatch(descriptors_icon, descriptors_image, k=2)

# Filter matches using Lowe's ratio test
good_matches1 = []
for m, n in matches1:
    if m.distance < 0.7 * n.distance:
        good_matches1.append(m)
good_matches1 = sorted(good_matches1, key= lambda x: x.distance)
```

Fig. 15: SIFT implementation using OpenCV in Python

First, the scene and the template are loaded into memory and color converted to the RGB color space, just like before. Subsequently, instances of the SIFT detector and FLANN matcher are created, specifying the algorithm details (defaults). Next, keypoints and their descriptors are found for the image and template respectively, using the *detectAndCompute()* method. Matches are created using the knnMatch method (k-nearest neighbors), and the matches are filtered using Lowe's ratio test. For simplicity, in this case we can use the assumption that the template is not rotated, stretched, or deformed in any way inside the scene. Assuming this, the position of the match can be calculated as the relative shift between the feature's coordinates inside the template and the scene. However, this assumption is not applicable in general, which is why a more sophisticated transformation is required to calculate the correct positioning in the general case.

3.5.3 Template Matching vs. SIFT - comparison

There isn't any clear way of comparing the reliability of the two methods. A confidence value can be extracted from the template matching algorithm that can be viewed as a reliability constant, which in this case varied from 0.99 to 0.74. With

regards to the SIFT matcher, a good metric for the confidence of the match may be the best distance ratio in the Lowe's ratio test as well as the number of features that passed the ratio test.

Lowe's ratio test is used to filter out the most significant features from an image by comparing the euclidean distance of the closest feature with the second closest in the descriptor space. If the closest feature is at least 0.7 times closer, the feature is marked as significant. The ratio of the closest to the second closest represents the measure of distinctness of the feature [27]. Therefore, the most distinct feature has the lowest Lowe's ratio.

The below table lists the confidence of the template matching algorithm (correlation coefficient mode) along with the Lowe's ratio and number of features accepted with the Lowe's ratio criterion.

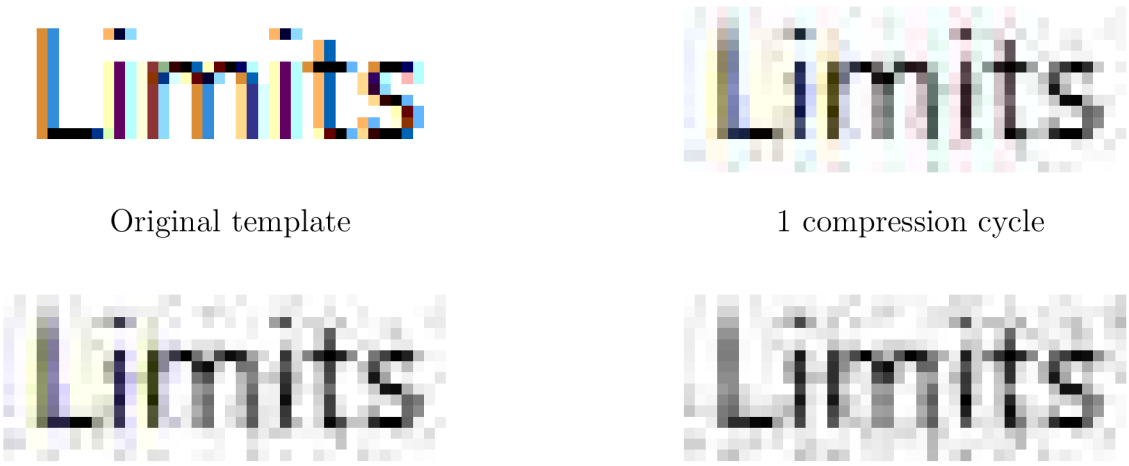
Tab. 1: Scientific template - resulting values of select search parameters

Item	**TM: Confidence**	**SIFT: Lowe ratio**	**SIFT: Features**
Original	0.99	0.09	2
Noise	0.94	0.24	2
Noise + blur	0.89	0.26	2
Noise + strong blur	0.74	0.68	1

Perhaps surprisingly, both the template matching and SIFT algorithms managed to successfully identify and locate the template inside the scene in all cases, despite the very high distortion. This may be because the chosen template itself is relatively unique in that there is no other text with both a similar font and background color in the scene. Also, we may notice the low number of features that SIFT relies on to localize the template, which is caused by its relatively small resolution. In the last case, the Lowe's ratio was very close to the threshold value of 0.7.

To further stress test and find the recognition capability of both algorithms, a new template was created with a slightly different distortion scheme that does not introduce artificial grain. Rather, it uses cumulative lossy compression, meaning that lossy compression algorithms are applied to the previously compressed image, amplifying the loss of detail. This better reflects the real distortions that may occur due to compression that occurs when capturing and processing a screen grab.

Black text with a white background was deliberately chosen as a template this time because of its greater similarity with other text in the scene, making the recognition task more difficult. We can notice that the image noise gets progressively more and more pronounced, and any color information is quickly suppressed by the compression.



Original template

1 compression cycle

5 compression cycles

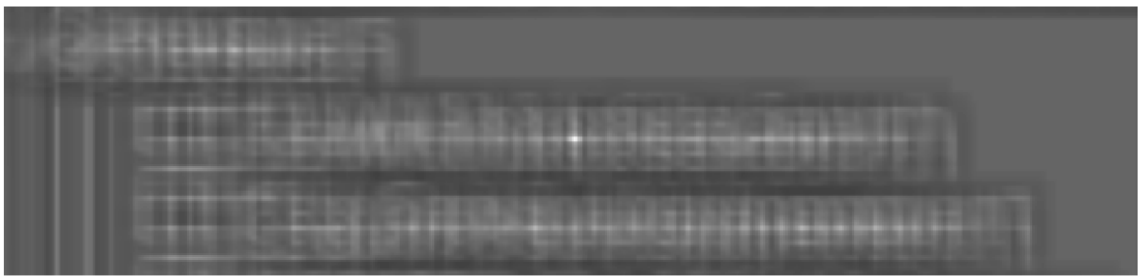
15 compression cycles

Fig. 16: Template screenshots with various degrees of compression

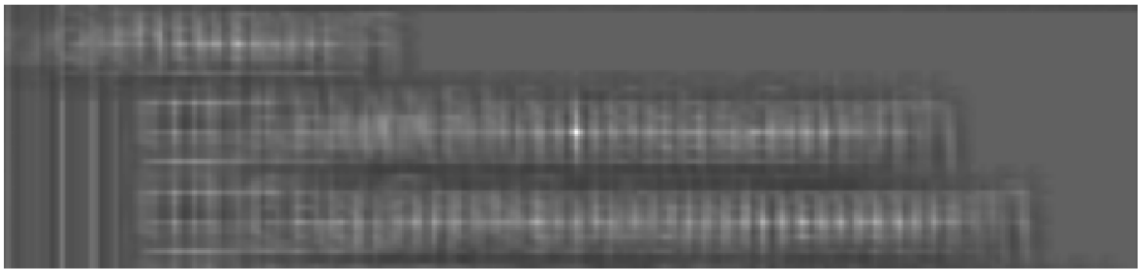
Tab. 2: Limits template - resulting values of select search parameters

Cycles	TM: Confidence	SIFT: Lowe ratio	SIFT: Features
Original	0.77	0.34	3
1	0.90	0.49	1
5	0.89	0.60	1
15	0.87	-	0

The TM algorithm managed to successfully identify the template in all cases, while SIFT did not in the last case. Interestingly, the template matching algorithm seems to benefit from the added compression, as suggested by the confidence value.



Original template's heat map



Compressed template's heat map

Fig. 17: Heat maps showing the perceived increase in confidence

When looking closer at the created heat maps (resulting matrices), it is clear that the compressed image contains more contrast between the brightest point (location of the template) and the rest of the image. To assess the contrast difference, the values from the matrices were sorted in descending order and compared. In the original template's heat map, the maximum value of 0.77 is followed by the second highest value of 0.59, which translates to a contrast difference of 0.18, while the compressed template's heat map maximum was 0.9 and the second highest value was also 0.59, corresponding to a contrast difference of 0.31.

The compressed picture therefore increased the contrast between the maximum value and the second highest value by 72 %. The cause of this is unclear and has been replicated during further testing. However, it can safely be concluded that image compression does not diminish the inference capability of the template matching algorithm, quite the contrary, in fact.

On the other hand, SIFT struggled as more and more noise was introduced in the templates, increasing the Lowe's ratio and decreasing the number of features considered significant by the algorithm. The image with the highest cumulative compression could not be identified at all because no features passed the Lowe's ratio test and therefore were not considered significant enough by the algorithm.

One might speculate that increasing the threshold value of the Lowe's ratio and therefore making the "good enough feature cut-off" value more lenient would

increase the algorithm's sensitivity. While it is true that more feature descriptors pass the threshold, they do not aid in the recognition capability and cause a very rapid increase in false-positives. Hence, the increased compression seriously hurts the recognition capability of SIFT.

Furthermore, benchmarks designed to assess and compare the inference time were designed. Both SIFT and Template Matching algorithms were run with varying screen resolutions ranging from fullHD up to 4K and template sizes of 40x40, 600x400, and 1920x1080.

Tab. 3: Template Matching inference times [s]

Scene resolution	**Small template**	**Middle template**	**Large Template**
FullHD	0.15	0.17	0.15
QHD	0.26	0.28	0.26
4k	0.57	1.42	0.82

Tab. 4: SIFT inference times [s]

Scene resolution	**Small template**	**Middle template**	**Large Template**
FullHD	0.33	0.44	1.06
QHD	0.64	0.84	1.8
4k	1.23	1.61	3.13

It is apparent that the SIFT's inference time is linearly proportional to scene resolution and also proportional to template size. In other words, the bigger the combined resolution of the scene and the template, the longer the inference time. However, SIFT's inference time can vary drastically based on image content. As the detail inside the image increases, the number of extracted features also grows, making the inference time longer. In testing, the performance gain on a nearly feature-less fullHD image can be up to 10x compared to a feature-rich image. This effect, however, does not apply to small scenes or templates. In practical UI testing, however, this effect does not incur any benefit over template matching because the large feature-less areas of images can be cropped out, making TM the faster option.

There are two competing factors in TM inference times. Let us assume a scene with a fixed 4K resolution and a varying resolution of the template. On the one hand, the number of comparisons at each translation point grows as template resolution increases. On the other hand, the number of translation points decreases as there are not as many positions where the image may fit inside the scene. Therefore, a worst-case scenario for template matching occurs at a particular ratio of scene to template

size. This occurs for the middle template size and the 4K scene. However, even in this worst case scenario, template matching still outperforms SIFT.

The results of this testing suggest that the best way forward in terms of both recognition robustness and inference time is to use template matching. Compared to SIFT, it also has the benefit of a direct and somewhat accurate reliability metric in the form of the confidence value. This would be very difficult to reproduce using the SIFT algorithm because of the above-mentioned dependence of the number of extracted feature descriptors on both image size and, mainly, image content. Moreover, the SIFT algorithm is color-blind and relies solely on the image's details, while template matching can differentiate feature-less images based on their color information alone. FililiX therefore uses template matching for its image recognition tasks because of all the listed advantages.

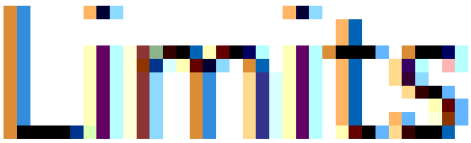
3.5.4 Benchmarks of template matching modes

OpenCV's implementation of template matching features three pairs of image matching modes, each pair consisting of a normed and not normed mode variant. For the purposes of FililiX, only the normed versions are used, as they provide the confidence value in an easily interpretable manner. To choose the right matching mode, a testing script was designed following OpenCV's documentation [18] that measures the mode's inference time and the confidence values. A certain amount of confidence value stability is also important to ensure predictable behavior when match rejection thresholds are set.

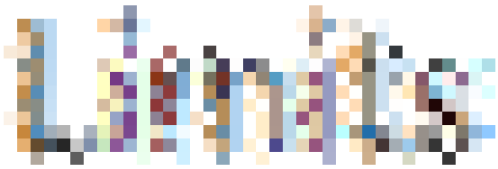
To again push this method to its limits, a new set of images was designed using image distortion since the TM algorithm had successfully identified all previous test images.

```python
# Load images
img = cv.imread('Resource/scene.png', cv.IMREAD_COLOR)
img2 = img.copy()
template = cv.imread('Resource/limits61.png', cv.IMREAD_COLOR)
h, w, _ = template.shape

# Test modes
methods = ['cv.TM_CCOEFF_NORMED', 'cv.TM_CCORR_NORMED', 'cv.TM_SQDIFF_NORMED']
for meth in methods:
    img = img2.copy()

    # Timer start
    start = time.time()

    # Apply template Matching
    res = cv.matchTemplate(img, template, eval(meth))
    min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)

    # If the method is TM_SQDIFF or TM_SQDIFF_NORMED, take minimum
    if meth in ['cv.TM_SQDIFF', 'cv.TM_SQDIFF_NORMED']:
        top_left = min_loc
        similarity = 1 - min_val
    else:
        top_left = max_loc
        similarity = max_val

    # Timer end
    end = time.time()

    # Identify bottom right corner
    bottom_right = (top_left[0] + w, top_left[1] + h)
    cv.rectangle(img, top_left, bottom_right, 200, 2)

    # Print results and save image
    print(f"Time: {end-start:.3f}, Similarity: {similarity:.3f}")
    name = meth.replace("cv.", "")
    cv.imwrite(f"Results/{name}.jpg", img)
```

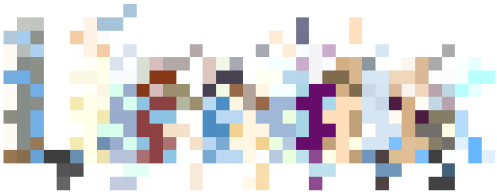
Fig. 18: Template Matching modes comparison script



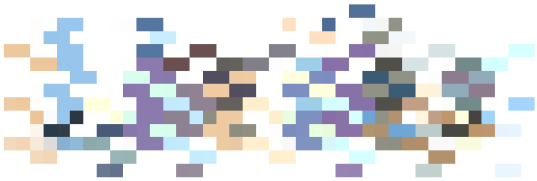
1. Original



2. Mosaic blur 30 %



3. Mosaic blur 60 %



4. Mosaic blur 80 %

Fig. 19: Template screenshots with various degrees of distortion

Similarly to previous test the results of the test are shown in the tables below.

Image	Correlation coefficient	Cross-correlation	Squared difference
Limits	✓	✓	✓
	✓	✓	✓
	✓		Down
			Items

Fig. 20: Success and false-positive matrix

The correlation-coefficient mode was the most successful at correctly identifying the template and seems to have the highest tolerance to image distortion. Interestingly, once this mode failed, it identified a false-positive that color-wise looks nothing like the original template. The squared difference mode had perhaps the most believable false-positives but was more prone to them compared to the other methods.

Tab. 5: TM modes inference times [s]

Image no.	Correlation coefficient	Cross-correlation	Squared difference
1.	0.136	0.129	0.138
2.	0.140	0.130	0.134
3.	0.135	0.129	0.130
4.	0.136	0.129	0.134

Tab. 6: TM modes similarities

Image no.	Correlation coefficient	Cross-correlation	Squared difference
1.	1.00	1.00	1.00
2.	0.64	0.98	0.95
3.	0.40	0.97	0.94
4.	0.41	0.97	0.94

Clearly, the time necessary to identify the image within the scene remains more or less constant for each of the images and is very similar from mode to mode. Therefore, inference time is not a factor worth considering.

The confidence values tell a different story. The values for the correlation coefficient mode lie in a reasonable range from 0.4 to 1, and this mode managed to identify all but the last image. However, the confidence values for the other two modes barely change at all, making the difference between a successful match and a failed find very tight, which is counter-intuitive. Moreover, these modes failed to identify the last two images, making their recognition worse. It is for this reason that FililiX uses template matching with the cross-correlation mode, as it is the most reliable and gives representative similarity values.

3.6 Custom software - features and architecture

FililiX is structured as a Python package, meaning that all source files are contained within a directory with an *__init__.py* file which tells Python to treat the directory as a package. Individual classes are extracted into their separate files. The following diagram shows the class structure of FililiX.

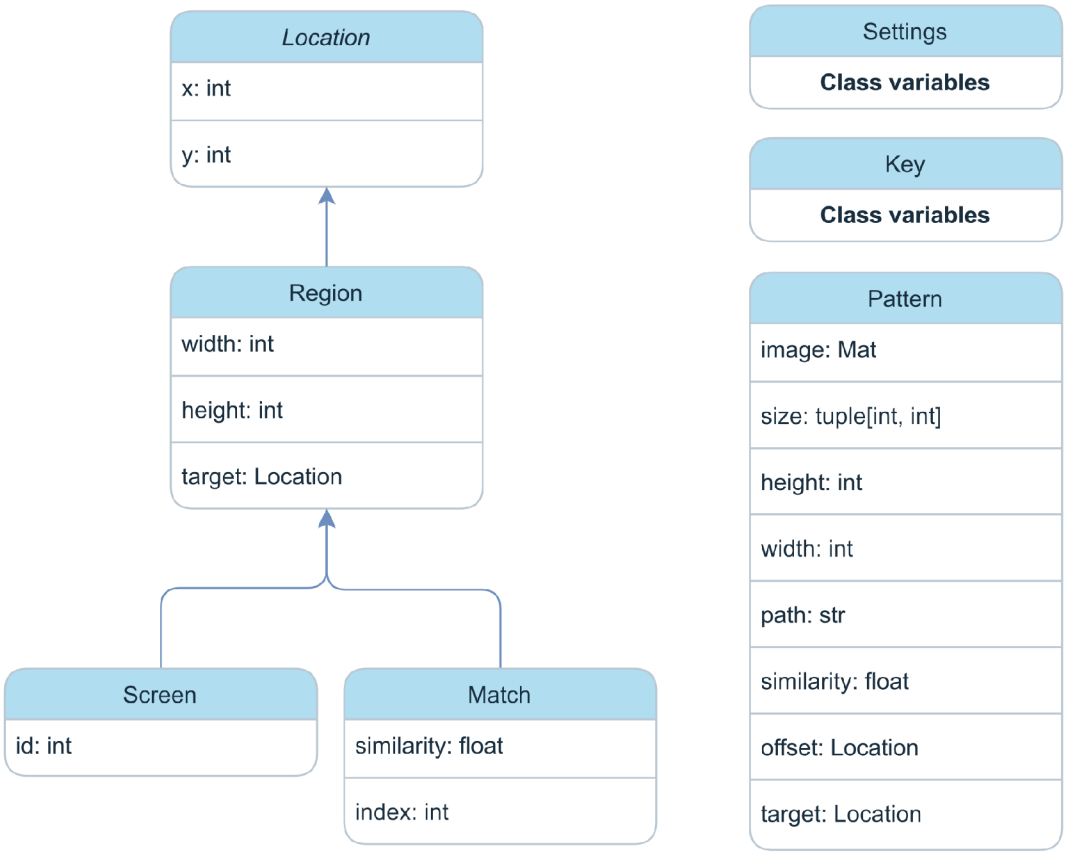


Fig. 21: FililiX class diagram

The *Location* class serves as an abstraction for a point on the screen, with the origin of the coordinate system in the upper left corner of the main monitor. The *Location* objects can be added to one another, and the individual coordinates can be set individually for each axis. Furthermore, a location can be offset using the *offset()* method.

The *Region* class inherits from the *Location* class. It represents a rectangular area on the screen and is defined by its upper left corner and its height, width, and target, which is a location object serving as an abstraction for a point within the *Region* where mouse actions should be performed. By default, the target object coordinates are set to the object's center. It is also the case that the *Region* object, just like in SikuliX's case, is unaware of the content within the *Region*'s bounds. It only represents a rectangular area on screen, which is useful for a variety of reasons, among which is the fact that the *Region* can mark off the area in which an image search should be executed, which brings significant performance gains by decreasing the searched area. The *Region* class features methods designed to manipulate the *Region*'s size, allowing it to grow and shrink or change individual dimensions.

A *Region* object can span multiple monitors, which was not the case in SikuliX's *Region* class, which was confined to a single monitor.

The *Screen* and *Match* classes both inherit from the *Region* class, and because of Python's multi-level inheritance, both of these classes have access to all the *Region* and *Location* methods. *Screen* extends the region with a unique ID that differentiates each monitor and can help with the management of windows. The *Screen* that represents the primary monitor in a multi-monitor setup is referred to as "SCREEN" in capital letters. However, it should be noted that it is not the case that the monitor with an ID equal to zero is the primary monitor. On Windows, any monitor can be set as primary, which may be confusing to users.

The *Match* class extends the *Region* class with a similarity value and an index that is used for categorization in operations that yield more than one *Match*. Unlike the replaced solution, this *Match* class does not contain the matched image itself. It only contains information about the location of the match (in global coordinates).

The *Pattern* class is independent of the other classes. It contains the searched image itself and its path, along with similarity and target information.

FililiX contains several helper classes that accommodate helper functions and settings. The Settings class understandably contains the software preferences, which are represented as class variables. The Key class stores the key-code *string* values of individual keys, also represented as class variables, allowing the user to access the key-codes using the dot notation. There are also several other classes that include other helper functions that mimic SikuliX's syntax, but these are not in any way interlinked with the business logic of the rest of the application.

All the above-mentioned classes are then interconnected using the fililix.py file, which contains function implementations that allow the user to call individual FililiX functionality using static method syntax (*click()*, *rightClick()*, etc.). These methods are also callable via the classes that implement them, meaning that calling *Region.click()* contains the reference to the implementation inside the fililix.py file and executes it, which is therefore the equivalent of calling *click(region=Region())*.

The individual functions implemented in fililix.py can be called using a different set of parameters, and the functions change behavior dependent upon the type of the received argument. This is achieved by checking the argument's type in a switch-case statement. This perhaps seems like a cumbersome way of handling different argument types, and it seems like the use of duck typing would be more intuitive. However, checking the argument type is not avoidable since one of the supported types is a *string*. Meaning that even if the implementations of other variable types were inside their respective classes, this could not be the case with *string*. In other words, implementing *click(pattern)* as *pattern.click()* is possible, while this

is not the case for *click("path/to/image.png")* since *"path/to/image.png".click()* is not a valid syntax.

This structure also means that the fililix.py file has to import all the classes to manipulate them, and at the same time, the individual classes have to import fililix.py in order to call the function implementations, resulting in an import loop. This import loop, however, does not represent a circular dependency. Therefore, it can be avoided by importing the fililix module directly inside the functions that call fililix.py implementations.

3.6.1 Functionality

FililiX's functionality is a subset of the SikuliX feature set, with notable performance improvements stemming from the technology used and extra capabilities mentioned later in the text.

FililiX can control the mouse and simulate mouse actions such as moving the mouse to a position indicated by an absolute coordinate or relatively by a set amount on the x and y axes using the *mouseMove()* function. Mouse clicks of all three standard mouse buttons (right, middle, and left) can be simulated using the click functions. Alternatively, a click can be separated into a mouseDown action followed by a mouseUp function with a set amount of time between them, simulating a long press of a mouse button.

Mouse actions are combined with image recognition, allowing the user to specify a UI control on which the mouse action should be performed. A sequence of adequate steps is executed based on argument type to find and act upon the given UI control. The process is outlined in detail in the following diagram.

First, the click method is called via the pattern object, which invokes the click function in fililix.py. In general, the click function accepts any of these five argument types: *Pattern*, *string*, *Match*, *Region* and *Location* collectively referred to as PSMRL. The find function is then called with the PSMRL argument (pattern in this case), a screenshot of the entire monitor is captured (since no search region was specified), and OpenCV's *matchTemplate()* is executed with the monitor screenshot as the scene and pattern's image as the template. Finally, a *Match* object is created with the location of the template in the screenshot (None is returned if no match is found). The mouse is then moved to the coordinates of the found match using the *mouseMove()* function, and the click action is performed using *mouse.click()*. More elaborate actions, such as the drag-and-drop operation, can be performed in a similar manner by combining the outlined operations in a variety of ways.

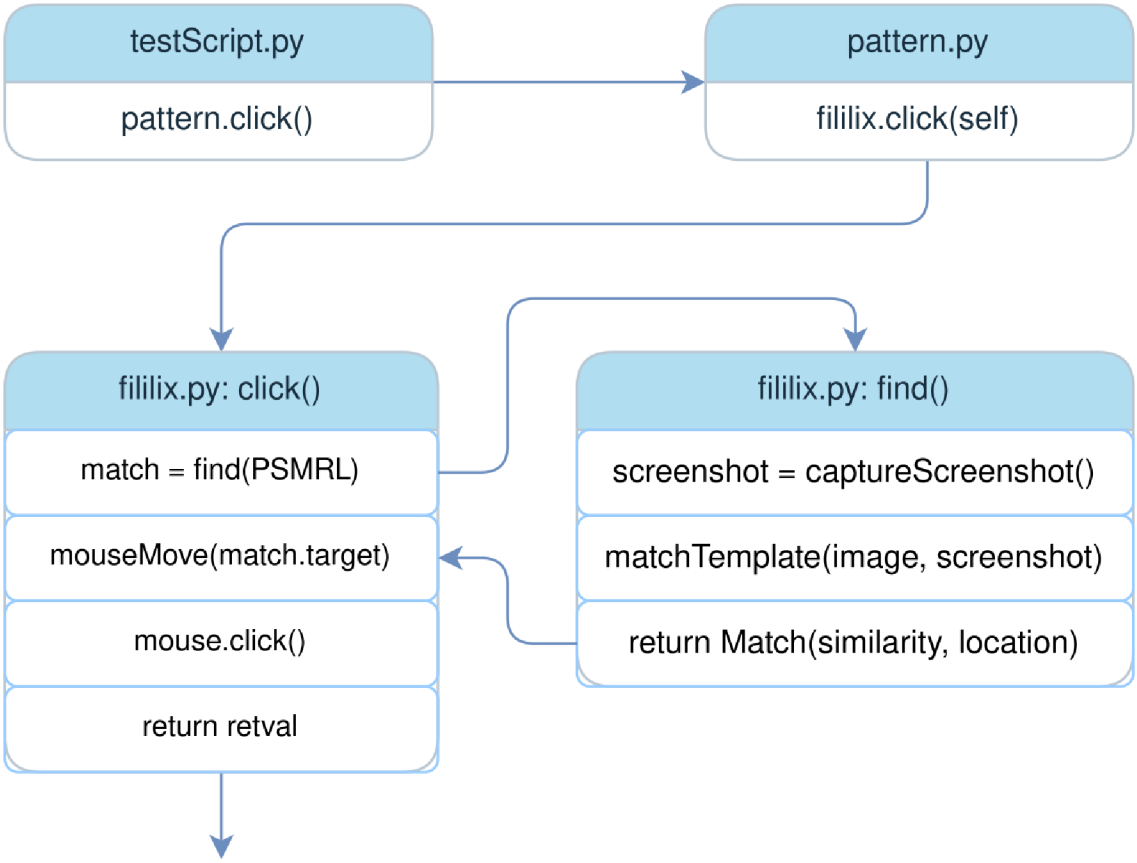


Fig. 22: FililiX class diagram

Similarly, the *hover()* function uses the same internal process to locate the template inside the screen and then moves the mouse to that location, leaving out the mouse click.

Scrolling is enabled via the *wheel()* function, which accepts a PSMRL object along with the desired wheeling direction and number of wheel steps to scroll. The wheel function first performs a hover operation to move the cursor over the target and then controls the mouse wheel for scrolling.

The basic image processing functionality is encapsulated in the find function, which serves as a basic template localization tool as described above. In the case where more than one UI control is to be located inside the scene, the *findAll()* function is used. Just like in the *find()* function, *matchTemplate()* is called and a heat map is generated. The find function extracts the maximum (or minimum) from the heat map that represents the best-found match. Instead, the *findAll()* function sets a similarity threshold (based on Settings or a passed PSMRL object) for the heat map, and all points above this threshold are considered individual Matches. The area around a *Match* also contains pixels with raised similarity values that may be above the set threshold. Therefore, more calculations are necessary

in order to filter out the ghost matches that would be generated around the match location.
Also, more than one image variant can be searched for using the *findBestList()* function, which returns a single *Match* object (or None if no matches are found) with the highest similarity value out of all PS (*Pattern* or *string*) objects in the list. Internally, the created *Match* objects are not sorted in any particular order. However, they are indexed by the order in which they appeared in the PS list, making it easy to identify which of the PS objects were found in the scene.

The *exist()* function waits for a set amount of time and checks whether a certain image has appeared on screen. In other words, the exist function is basically recursively calling the find operation at fixed intervals whose duration is set in the Settings using the waitScanRate option.

The *waitVanish()* function is essentially an inverted exist operation that waits for the disappearance of an image from the screen. Both the *waitVanish()* and *exist()* functions can be configured to wait an indefinite amount of time by passing infinity as their timeout value.

FililiX features several functions for keyboard control, the most basic of which is the *type()* function, which simply types a given *string* one letter at a time. The type function automatically recognizes whether a given *string* contains special keywords, such as a combination of keys CTRL+ALT+DELETE, and presses them simultaneously, which is required for the keyboard shortcut to work. This function also accepts modifier keys besides just the given *string*. These keys are pressed and held down while the text is being typed (e.g., the SHIFT key).

A keyboard key press can be separated into a press and release of a key using the *keyDown()* and *keyUp()* functions, which allow the user to perform mouse actions while a given key is being held down.

FililiX provides other utility functions that do not directly control user inputs but provide useful functionality that is beneficial in this context. The *getClipboard()* utility accesses and pastes the contents of the clipboard (limited to only pure text as of now). The *isLockOn()* function checks whether the caps lock key is currently pressed. The self-explanatory function *getMouseLocation()* captures the current cursor position and outputs it as a *Location* object. FililiX also has access to information about the number of currently connected monitors and their resolutions.

Furthermore, a handful of extra functions have been implemented, notably the ability of a find operation to intelligently search for multiple variants of a given image. For example, there might be a UI control that differs in shape based on software versions. For instance, the start button on Windows machines has changed throughout their releases. FililiX searches the directory indicated by the path of the image file and looks for file names with a previously agreed-upon format that in-

dicates that the file is a variant of the same UI control using a regular expression. Then, if there are such variants, the find operation uses them all as templates in the *findBestList()* function to identify the UI control.

The above-mentioned functionalities are not a complete record of all functions implemented in FililiX, but they represent the core functionality that this software provides. The left-out functionality is usually trivial in nature and mostly relates to the manipulation of PSMRL objects, such as changing their size, adding them together, getting information about their position, and so on. All of the listed functions were tested, and their functionality was successfully verified on multiple Windows machines and in the regression testing system in TFS.

4 Discussion

The proposed custom software was subject to multiple constraints because of the nature of the assignment, which guided many design decisions, most notably the portable software solution and the mirrored syntax that led to unusual naming conventions. The preserved syntax meant dealing with the differences between Java and Python, especially method overloading, which is not a native design pattern in Python. However, these constraints were not in any way limiting as far as the performance and reliability of the software are concerned.

Method overloading was resolved using the match-case statement (favored over multiple or single dispatch[47][48] packages since neither of them are thread-safe). Multi-threading is not yet implemented in FililiX, but future extensions may do so, including the proposed and implemented intelligent image variant search that would benefit from simultaneous variant searching.

Regarding the maintenance and upgradeability of Python and the chosen packages, the embedded Python can simply be updated by downloading a fresh copy of embeddable Python and then installing Pip and the necessary packages as listed in the created requirements.txt file. Packages with a large and active developer base were selected to ensure their longer-term support. On top of that, the package selection was biased towards smaller, lighter packages over large, feature-rich libraries (with the exception of OpenCV) to maximize reliability and maintain a reasonable package size at the expense of some extra functionality. This bias stems from the assignment constraints that emphasize the reliability and portability of the solution.

Other functionality that may be implemented includes the SIFT recognition method, which may be of use in some edge cases, e.g., searching for a pattern in an actual microscope image, taking advantage of SIFT's greater flexibility to variance in distortion and orientation. In combination with other methods, this would allow for automatic testing of the microscope's imaging system, which as of yet is not part of automated regression testing.

Moreover, template matching that utilizes the cross-correlation mode could be implemented since it is as of yet the only match mode that supports masked search. When comparing the template to the scene, the masked search ignores certain parts of the image, indicated by a zero value in the mask. This is particularly useful for templates with transparent (and therefore constantly changing according to the background) areas. A classical template matching search on such an image produces low confidence values, or the template is rejected completely.

5 Conclusion

The first part of this thesis focuses on a theoretical overview of the technologies used. The previously used software solution, SikuliX, is outlined, highlighting its advantages and disadvantages, especially in relation to TFS's regression testing. Also, its basic philosophy for visual-based UI testing is described, and a description of a tight selection of SikuliX's functionality is given. I explain robot process automation, the technology that it entails, as well as its utility in relation to UI testing of software applications or automation in general. The theoretical part contains a brief overview of the structure of an electron microscope, its advantages over a classical microscope, and a description of the xT microscope control application and its UI layout.

The majority of the theory is concerned with image recognition techniques relevant to the developed software, namely template matching and SIFT. An outline of the reasons why the selected methods are well suited for UI testing is described in the introduction. The basic mechanisms of both above mentioned methods are described, including their respective advantages and disadvantages. The mathematical background of template matching, which lies in the convolution operation, is also mentioned.

The main goal of this work was to design replacement software for SikuliX, which is what the rest of the work is concerned with. The custom software solution was designed in Python, as this is the language that the existing scripts were written in, and therefore a Python interpreter was needed anyway. Moreover, Python is considered the standard language for applied computer science, such as machine learning and computer vision, which are areas relevant to this work.

The assignment required the software to work without the need to install it on tested devices. Research on multiple appropriate ways of achieving this goal was conducted, and the specifics of each approach were compared and subsequently benchmarked against one another. The most fitting of which, Python embedded, was selected because of its performance, ease of deployment, and official support from the developers of Python. The existing scripts were migrated from Python 2 to Python 3, which allows access to the newest packages and libraries Python has to offer.

Libraries that control user inputs for the custom software were selected based on their reliability, size, active developer base, and performance, which were assessed in the conducted benchmarks. The selected image recognition methods were also tested against one another for their precision, inference time, and consistency. The testing showed that template matching is the better option in the context of UI testing as it is the more robust and faster of the two methods. Next, various

matching modes in their normed form were benchmarked, again seeking the most reliable and robust option. The correlation coefficient method was selected for the consistency of its similarity values, which accurately represented the confidence of the match operation.

All functionality used by the TFS's regression testing system was implemented and tested using the selected methods and technologies. Additionally, because FililiX's functions are documented using doc-strings, tools such as the built-in pydoc[49] or Sphinx[50] modules can be used to automatically generate documentation in the HTML format, making it easier to get FililiX's users up to speed. Extra functionality was implemented, and other possible extensions of the software were proposed. The benefits of this solution stem from the retired technologies that were no longer supported (Jython), the easy extendability of the software, its portability between devices, and performance improvements, especially in start-up time.

BIBLIOGRAPHY

[1] Red Hat, I. *CI/CD pipeline* [online]. May 2022 [cit. 2023-04-09]. Available at: https://www.redhat.com/en/topics/devops/what-cicd-pipeline.

[2] GitHub. *Continuous Integration and Continuous Delivery (CI/CD) Fundamentals* [online]. 2020 [cit. 2023-04-09]. Available at: https://resources.github.com/ci-cd/.

[3] Software, S. *Guide to UI Testing with Continuous Integration* [online]. 2023 [cit. 2023-04-09]. Available at: https://smartbear.com/resources/ebooks/guide-to-ui-testing-with-continuous-integration/.

[4] Ivančić, L., Suša Vugec, D. and Vuksic, V. *Robotic Process Automation: Systematic Literature Review*. 1st ed. August 2019. 280–295 p. ISBN 978-3-030-30428-7. Available at: https://www.researchgate.net/publication/335400552_Robotic_Process_Automation_Systematic_Literature_Review.

[5] Wikipedia. *Robotic process automation — Wikipedia, The Free Encyclopedia* [online]. 2023 [cit. 2023-05-05]. Available at: http://en.wikipedia.org/w/index.php?title=Robotic%20process%20automation&oldid=1145180413.

[6] SeleniumHQ. *Selenium* [online]. Selenium, may 2023 [cit. 2023-05-05]. Available at: https://github.com/SeleniumHQ/selenium.

[7] Raimund Hocke. *SikuliX Documentation — SikuliX 2.x+ documentation* [online]. 2014 [cit. 2023-04-05]. Available at: https://sikulix-2014.readthedocs.io/en/latest/index.html.

[8] Raimund Hocke. *How Sikuli Works — SikuliX 2.x+ documentation* [online]. April 2022 [cit. 2023-04-18]. Available at: https://sikulix-2014.readthedocs.io/en/latest/devs/system-design.html.

[9] Jim Hugunin. *Jython* [online]. April 2022 [cit. 2023-04-18]. Available at: https://www.jython.org/index.

[10] Wewerka, J. and Reichert, M. *Robotic Process Automation – A Systematic Literature Review and Assessment Framework* [online]. arXiv, 2020 [cit. 2023-05-17]. DOI: 10.48550/arXiv.2012.11951. Available at: http://arxiv.org/abs/2012.11951.

[11] Celonics. *Make | Work the way you imagine* [online]. 2023 [cit. 2023-04-14]. Available at: https://www.make.com/en.

[12] Microsoft Inc.. *Accessibility Insights* [online]. 2023 [cit. 2023-04-22]. Available at: https://accessibilityinsights.io.

[13] Robot Framework Foundation. *Robot Framework* [online]. 2023 [cit. 2023-04-22]. Available at: https://robotframework.org/.

[14] Sweigart, A. *PyAutoGUI* [online]. April 2023 [cit. 2023-04-22]. Original-date: 2014-07-17T23:30:38Z. Available at: https://github.com/asweigart/pyautogui.

[15] andrey senyaev and asmorkalov. *Opencv-python: Wrapper package for OpenCV python bindings.* [online]. 2023 [cit. 2023-05-06]. Available at: https://github.com/opencv/opencv-python.

[16] Inc., A. *2D Convolution ilustration* [online]. 2023 [cit. 2023-04-12]. Available at: https://developer.apple.com/documentation/accelerate/blurring_an_image.

[17] Brunelli, R. *Template Matching Techniques in Computer Vision: Theory and Practice | Wiley.* 1st ed. Hoboken, New Jersey: Wiley, March 2009. ISBN 978-0-470-51706-2. Available at: https://www.wiley.com/en-gb/Template+Matching+Techniques+in+Computer+Vision%3A+Theory+and+Practice-p-9780470517062.

[18] Team, O. *OpenCV: Object Detection* [online]. 2023 [cit. 2023-04-03]. Available at: https://docs.opencv.org/3.4/df/dfb/group__imgproc__object.html#ga586ebfb0a7fb604b35a23d85391329be.

[19] Dumoulin, V. and Visin, F. *A guide to convolution arithmetic for deep learning* [online]. arXiv, january 2018 [cit. 2023-04-14]. DOI: 10.48550/arXiv.1603.07285. ArXiv:1603.07285 [cs, stat]. Available at: http://arxiv.org/abs/1603.07285.

[20] Team, O. *OpenCV: Template Matching* [online]. 2023 [cit. 2023-04-12]. Available at: https://docs.opencv.org/3.4/d4/dc6/tutorial_py_template_matching.html.

[21] Kriz, T. Corner-turn complexity properties of polynomial transform 2D convolution methods. In: Kriz, T., ed. *ICASSP '81. IEEE International Conference on Acoustics, Speech, and Signal Processing.* 1981, vol. 6, p. 319–322. DOI: 10.1109/ICASSP.1981.1171285.

[22] Lin, Y.-H. and Chen, C.-H. Template matching using the parametric template vector with translation, rotation and scale invariance. *Pattern Recognition.* 1st ed. 2008, vol. 41, no. 7, p. 2413–2421. DOI: https://doi.org/10.1016/j.patcog.2008.01.017. ISSN 0031-3203. Available at: https://www.sciencedirect.com/science/article/pii/S0031320308000344.

[23] Hashemi, N. S., Roya Babaei Aghdam, Atieh Sadat Bayat Ghiasi and Parastoo Fatemi. Template Matching Advances and Applications in Image Analysis. *American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS)*. 1st ed. 2016, vol. 26, no. 3, p. 91–108. ISSN 2312-4402. Available at: `https://core.ac.uk/reader/235050100`.

[24] Davies, E. Chapter 5 - Edge detection. In: Davies, E., ed. *Computer Vision (Fifth Edition)*. Fifth Editionth ed. Academic Press, 2018, p. 119–145. DOI: https://doi.org/10.1016/B978-0-12-809284-2.00005-8. ISBN 978-0-12-809284-2. Available at: `https://www.sciencedirect.com/science/article/pii/B9780128092842000058`.

[25] Nixon, M. S. and Aguado, A. S. 4 - Low-level feature extraction (including edge detection). In: Nixon, M. S. and Aguado, A. S., ed. *Feature Extraction and Image Processing for Computer Vision (Fourth Edition)*. Fourth Editionth ed. Academic Press, 2020, p. 141–222. DOI: https://doi.org/10.1016/B978-0-12-814976-8.00004-X. ISBN 978-0-12-814976-8. Available at: `https://www.sciencedirect.com/science/article/pii/B978012814976800004X`.

[26] Jakubović, A. and Velagić, J. Image Feature Matching and Object Detection Using Brute-Force Matchers. In: Jakubović, ed. *2018 International Symposium ELMAR*. Zadar: IEEE, 2018, vol. 1, p. 83–86. DOI: 10.23919/ELMAR.2018.8534641. ISBN 9789531842440.

[27] Lowe, D. Object recognition from local scale-invariant features. In: Lowe, D. G., ed. *Proceedings of the Seventh IEEE International Conference on Computer Vision*. 1999, vol. 2, p. 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410.

[28] Misra, S. and Wu, Y. Chapter 10 - Machine learning assisted segmentation of scanning electron microscopy images of organic-rich shales with feature extraction and feature ranking. In: Misra, S., Li, H. and He, J., ed. *Machine Learning for Subsurface Characterization*. Gulf Professional Publishing, 2020, p. 289–314. DOI: https://doi.org/10.1016/B978-0-12-817736-5.00010-7. ISBN 978-0-12-817736-5. Available at: `https://www.sciencedirect.com/science/article/pii/B9780128177365000107`.

[29] Perrett, D. I. and Oram, M. W. Visual recognition based on temporal cortex cells: viewer-centred processing of pattern configuration. *Zeitschrift Fur Naturforschung. C, Journal of Biosciences*. 1st ed. 1998, vol. 53, 7-8, p. 518–541. DOI: 10.1515/znc-1998-7-807. ISSN 0939-5075.

[30] Lowe, D. G. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*. Springer. november 2004, vol. 60,

no. 2, p. 91–110. DOI: 10.1023/B:VISI.0000029664.99615.94. ISSN 1573-1405. Available at: https://doi.org/10.1023/B:VISI.0000029664.99615.94.

[31] Beis, J. S. and Lowe, D. G. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In: IEEE. *Proceedings of IEEE computer society conference on computer vision and pattern recognition*. San Juan, Puerto Rico: IEEE Comput. Soc, 1997, p. 1000–1006. DOI: 10.1109/CVPR.1997.609451. ISBN 9780818678226. Available at: https://ieeexplore.ieee.org/abstract/document/609451/.

[32] Utah, T. U. of. *Introduction to Electron Microscopy - Advanced Microscopy - Imaging Facilities - The University of Utah* [online]. 2011 [cit. 2023-04-20]. Available at: https://advanced-microscopy.utah.edu/education/electron-micro/.

[33] Kirkland, E. J. *The Transmission Electron Microscope*. Boston, MA: Springer US, 2010. 5–27 p. ISBN 978-1-4419-6533-2. Available at: https://doi.org/10.1007/978-1-4419-6533-2_2.

[34] Savile Bradbury, Brian J. Ford and David C. Joy. *Scanning electron microscope (SEM) | Definition, Images, Uses, Advantages, & Facts | Britannica* [online]. March 2023 [cit. 2023-04-20]. Available at: https://www.britannica.com/technology/scanning-electron-microscope.

[35] Jupyter, P. *Project Jupyter* [online]. 2023 [cit. 2023-05-06]. Available at: https://jupyter.org.

[36] Guido van Rossum, Jukka Lehtosalo and Łukasz Langa. *PEP 484 – Type Hints |peps.python.org* [online]. September 2014 [cit. 2023-05-05]. Available at: https://peps.python.org/pep-0484/.

[37] Besbes, A. *12 Beginner Concepts About Type Hints To Improve Your Python Code* [online]. February 2023 [cit. 2023-05-05]. Available at: https://towardsdatascience.com/12-beginner-concepts-about-type-hints-to-improve-your-python-code-90f1ba0ac49.

[38] Guido van Rossum, Barry Warsaw and Nick Coghlan. *PEP 8 – Style Guide for Python Code |peps.python.org* [online]. July 2001 [cit. 2023-05-05]. Available at: https://peps.python.org/pep-0008/.

[39] Pyinstaller. *GitHub - PyInstaller - Freeze Python programs into stand-alone executables* [online]. 2023 [cit. 2023-05-19]. Available at: https://github.com/pyinstaller/pyinstaller.

[40] Duarte, M. *Cx_Freeze* [online]. May 2023 [cit. 2023-04-18]. Available at: https://github.com/marcelotduarte/cx_Freeze.

[41] Foundation, P. S. *4. Using Python on Windows* [online]. 2023 [cit. 2023-05-17]. Available at: https://docs.python.org/3/using/windows.html.

[42] Python Software Foundation. *2to3 — Automated Python 2 to 3 code translation* [online]. 2023 [cit. 2023-05-19]. Available at: https://docs.python.org/3/library/2to3.html.

[43] BoppreH. *Mouse: Hook and simulate mouse events on Windows and Linux* [online]. 2023 [cit. 2023-05-06]. Available at: https://github.com/boppreh/mouse.

[44] BoppreH. *Keyboard: Hook and simulate keyboard events on Windows and Linux* [online]. 2023 [cit. 2023-05-06]. Available at: https://github.com/boppreh/keyboard.

[45] Clark (Alex), J. A. *Pillow: Python Imaging Library (Fork)* [online]. 2023 [cit. 2023-05-06]. Available at: https://python-pillow.org.

[46] Team, O. *OpenCV: Feature Matching* [online]. 2023 [cit. 2023-05-06]. Available at: https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html.

[47] Rocklin, M. *Multipledispatch: Multiple dispatch* [online]. 2023 [cit. 2023-05-16]. Available at: http://github.com/mrocklin/multipledispatch/.

[48] Lukasz Langa. *PEP 443 – Single-dispatch generic functions |peps.python.org* [online]. May 2013 [cit. 2023-05-06]. Available at: https://peps.python.org/pep-0443/.

[49] Foundation, P. S. *Pydoc — Documentation generator and online help system* [online]. 2023 [cit. 2023-05-17]. Available at: https://docs.python.org/3/library/pydoc.html.

[50] Brandl, G. *Sphinx: Python documentation generator* [online]. 2023 [cit. 2023-05-17]. Available at: https://www.sphinx-doc.org/.

SYMBOLS AND ABBREVIATIONS

RPA - Robot process automation
TFS - Thermo Fisher scientific
TM - Template matching
SIFT - Scale Invariant Feature Transform
CI/CD - Continuous integration and continuous delivery
UI - User interface
GUI - Graphical user interface
OCR - Optical character recognition
IDE - Integrated development environment
JVM - Java Virtual Machine
API - Application programming interface
DNN - Deep neural network
SURF - Speeded-Up Robust Features
ORB - Oriented FAST and rotated BRIEF
FLANN - Fast approximate nearest neighbor
TEM - Transmission electron microscope
SEM - Scanning electron microscope
CRT - cathode-ray tube
PEP - Python enhancement proposals
OS - Operation system
ISO - optical disc image
RGB - Red green blue
PSMRL - Pattern, string, Match, Region and Location
PS - Pattern, string

LIST OF FIGURES

1 Diagram of Sikuli IDE system design architecture taken from [8] . . . 17
2 Coordinate system of multi-monitor setups. Taken from [7] 18
3 Discrete 2D convolution, taken from [16] 23
4 $R(x, y)$ matrices for different individual modes. 25
5 Matched features between an image and template. Taken from [26] . 26
6 Keypoint descriptor creation using Gaussian gradient histograms. Taken from [30] 27
7 Structure of a scanning electron microscope. Taken from [34] 29
8 xT application: Green - menu bar, Blue - Toolbar, Red - Imaging quadrants, Yellow - Pages, Pink - Page Controls, Cyan - Status bar [34] 30
9 Type hints do not enforce runtime variable types. Code runs without errors and yields "3" and "ab" 32
10 Average execution time comparison between *PyAutoGUI* and low level low-level packages 36
11 Average execution time comparison for 100 consecutive actions. 37
12 Template screenshots with various degrees of degradation 39
13 Image of the searched scene 39
14 Template matching implementation using OpenCV in Python 40
15 SIFT implementation using OpenCV in Python. 41
16 Template screenshots with various degrees of compression 43
17 Heat maps showing the perceived increase in confidence. 44
18 Template Matching modes comparison script 47
19 Template screenshots with various degrees of distortion 48
20 Success and false-positive matrix 48
21 FililiX class diagram 50
22 FililiX class diagram 53

LIST OF TABLES

1 Scientific template - resulting values of select search parameters . 42
2 Limits template - resulting values of select search parameters .. 43
3 Template Matching inference times [s] 45
4 SIFT inference times [s] 45
5 TM modes inference times [s] 49
6 TM modes similarities 49