Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Science

NUnit test framework extension for test data preparation

Master's Thesis

Tadeáš Zribko

Prague, May 2025

Study programme: Open informatics
Branch of study: Software Engineering

Supervisor: Ing. Karel Frajták, Ph.D.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisor.

Swap this for the Thesis Assignment, when you have it from the department.

Declaration

I declare that presented work was developed independently, and that I have listed all sources of information used within, in accordance with the Methodical instructions for observing ethical principles in preparation of university theses.

Date ....

Abstract

The study of autonomous **UAV!**s (**UAV!**s) has become a prominent sub-field of mobile robotics.

Keywords

Abstrakt

Výzkum na poli autonomních bezpilotních prostředků (UAV) se stal významným oborem mobilní robotiky.

Klíčová slova

Abbreviations

Contents

1 Introduction

...existing code...

1.1 Motivation

1.2 Objectives

2 State of the Art

...existing code...

2.1 Current Software Testing Practices

2.2 Existing Frameworks

IntelliTest Framework

AutoFixture Framework

2.3 Research Gaps

3 Preliminaries

...existing code...

3.1 Software Testing Concepts

3.2 Overview of C#

3.3 .NET Testing Ecosystem

3.4 Next Unit Testing Framework

3.5 Screenplay Pattern

Behavior-Driven Development

Business-Objective-Action (BOA)

Definition

The Business-Objective-Action (BOA) pattern is a structured approach to defining and organizing test cases based on business objectives and actions. It helps ensure that tests are aligned with business goals and provide meaningful feedback on the system's behavior.

3.6 Tools and Technologies

4 Framework Design

...existing code...

4.1 Framework Requirements

The primary goal of the framework extension is to enhance the existing unit testing capabilities by integrating advanced data preparation and handling mechanisms. Unit testing is a critical practice in modern software development, ensuring that individual components of an application behave as expected. However, setting up and managing test data can often become a cumbersome process, especially in complex systems where tests depend on intricate data structures, external services, or specific state configurations.

Functional Requirements

- **Custom Data Preparation Attributes:**
  - Implement custom attributes to facilitate specific data setup and teardown processes for each test case.
  - Attributes should allow dynamic injection of test data and enable conditional test execution based on data state.
- **Automated Test Data Handling:**
  - Develop a `TestDataHandler` class to automate the execution of data preparation methods before and after tests, ensuring a consistent test environment.
  - Ensure that both simple and complex test data structures can be seamlessly prepared and injected.
- **Data Preparation Store:**
  - Create a `TestDataPreparationStore` to maintain mappings of data preparation instances associated with test methods, enabling efficient data management.
  - Implement thread-safe access for concurrent test executions in multi-threaded environments.
- **Integration with NUnit Test Lifecycle:**
  - Leverage NUnit's `ITestAction` interface to integrate custom behaviors into the test execution lifecycle, using methods like `BeforeTest` and `AfterTest`.
  - Ensure custom lifecycle hooks are compatible with NUnit's parallel test execution model.
- **Attribute Count Tracking:**
  - Implement a `TestAttributeCountStore` to track the execution of custom attributes and ensure all data preparations are completed before test execution.
  - Prevent redundancy in data preparation execution for tests using multiple attributes.
- **Service Provider Utilization:**
  - Use a service provider pattern to manage dependencies and services required for data preparation.
  - Support integration with dependency injection frameworks.

Non-Functional Requirements

- **Performance Efficiency:**
  - Ensure minimal overhead is introduced during test execution to maintain optimal performance.
  - Optimize data preparation logic to minimize unnecessary computation or resource usage.
- **Modularity and Extensibility:**
  - Design the framework with a modular architecture to facilitate easy maintenance and future enhancements.
  - Provide extension points for developers to customize or extend attributes and lifecycle handling.
- **Compliance with Coding Standards:**
  - Adhere to coding best practices and standards to ensure code quality and readability.
  - Follow SOLID[1] principles for maintainable and testable code.
- **Seamless NUnit Integration:**
  - The extension should integrate smoothly with the existing NUnit framework without disrupting current testing workflows.
  - Support legacy NUnit test cases alongside the new extension.

4.2 Framework Architecture

The architecture of the framework has been carefully designed to ensure scalability, maintainability, and seamless integration with existing unit testing workflows. It is structured into distinct layers, each fulfilling specific responsibilities while adhering to principles of modular design and separation of concerns.The following subsections describe the primary components of the architecture.

Attributes Layer

The *Attributes Layer* provides the interface for specifying and controlling test data preparation. It introduces two primary attributes that enhance test configurability and reusability.

The first attribute, `UsePreparedDataParamsForAttribute`, is derived from `UsePreparedAttribute`.This attribute enables the definition of data preparation and teardown methods at both the class and method levels. By supporting parameterized configuration, it facilitates the reuse of preparation logic across multiple test methods, thus reducing code duplication and improving maintainability.

Complementing this is the `UsePreparedDataParamsAttribute`, which targets individual test methods. Unlike the previous attribute, it allows for precise control over data preparation specific to a particular test scenario. This attribute supports parameterized data injections and conditional logic, enabling dynamic and flexible preparation workflows without impacting the entire test class.

[1]The SOLID principles are a set of five design guidelines aimed at improving the clarity, flexibility, and maintainability of object-oriented software.

Data Handling Layer

The *Data Handling Layer* is responsible for orchestrating the preparation, management, and cleanup of test data. This layer introduces key components to ensure the integrity and consistency of the testing environment.

At the core of this layer is the `TestDataHandler` class, which automates the execution of preparation (*DataUp*) and cleanup (*DataDown*) methods. The handler enforces a deterministic execution order, ensuring that all preparatory logic is executed before a test runs, and cleanup operations are reliably performed afterward. Furthermore, it incorporates exception-handling mechanisms to guarantee proper cleanup even in cases where tests fail unexpectedly.

The `TestDataPreparationStore` complements the handler by providing a centralized repository for managing data preparation instances associated with test methods. This store optimizes test execution through intelligent caching of prepared data, reducing redundant computations and improving performance. Additionally, it is implemented with thread safety to support concurrent test executions in multi-threaded environments.

Integration Layer

The *Integration Layer* focuses on enhancing the efficiency of attribute execution and promoting modularity through service-oriented design. The `TestAttributeCountStore` plays a critical role in ensuring the correct execution of custom attributes. It monitors the invocation of attributes applied to test methods, preventing duplicate executions of *DataUp* and *DataDown* methods when multiple attributes are stacked. This mechanism guarantees that all preparatory logic runs as expected while maintaining efficiency.

To further decouple preparation logic, the framework leverages the `CaseProviderStore`, which acts as a service provider. This component supports dependency injection principles, allowing developers to register and retrieve services dynamically. Such an approach enables the framework to integrate seamlessly with existing dependency injection frameworks, such as `Microsoft.Extensions.DependencyInjection`, while remaining adaptable to diverse test environments.

4.3 NUnit Integration

The framework seamlessly integrates with NUnit by extending its lifecycle through custom hooks. This integration is achieved using NUnit's `ITestAction` interface, which provides methods to execute logic before and after test execution.

The `BeforeTest` and `AfterTest` methods are overridden to inject data preparation steps (*DataUp*) prior to test execution and to perform cleanup (*DataDown*) afterward. This ensures that preparatory and teardown logic integrates seamlessly without disrupting NUnit's native test execution flow.

To provide fine-grained control, the attributes specify `ActionTargets.Test`, indicating that the custom behaviors are applied to individual test methods. However, the architecture also supports extensibility to target entire test classes or namespaces, enabling developers to apply bulk preparation logic when required.

Parallel Execution Support

Given the growing importance of parallelism in modern testing frameworks, the architecture has been designed to ensure reliable behavior in NUnit's *parallel execution mode*. All lifecycle hooks, data preparation stores, and service providers are implemented with thread safety guarantees. This ensures that test data preparation and cleanup operations are executed consistently, even when tests are run concurrently across multiple threads.

Summary

The framework's architecture combines modularity, scalability, and extensibility to deliver a robust solution for advanced test data preparation. By introducing dedicated layers for attributes, data handling, and integration, the design adheres to clean coding principles and fosters maintainability. Furthermore, its seamless integration with NUnit ensures compatibility with existing testing workflows, while support for parallel execution makes it well-suited for modern, high-performance testing environments.

5 Framework Implementation

...existing code...

5.1 Technology Selection

5.2 Implementation of Attributes

5.3 Supporting BDD

5.4 Sample Testing Code

5.5 Framework Integration

6 Experiments and Framework Evaluation

...existing code...

6.1 Test Scenarios

6.2 Framework Comparison

6.3 Framework Evaluation

6.4 Framework Limitations

7 User Guide for the Framework

...existing code...

7.1 Framework Installation

7.2 Project Configuration

7.3 Preparing Test Scenarios

7.4 Running Tests

7.5 Sample Scenario

7.6 Tips and Best Practices

8 Discussion

...existing code...

8.1 Framework Benefits

8.2 Possibilities for Extension

8.3 Framework Limitations

9 Conclusion

...existing code...

9.1 Summary of Key Findings

9.2 Recommendations for Future Work

10 Introduction

First, introduce the reader to the research topic. Start with the most general view and slowly converge to the particular field, sub-field, and the challenges you face. You can cite others' work here [0].

10.1 Related works

This section should contain related state-of-the-art works and their relation to the author's work. We usually cite the original works like this [0]. You can also cite multiple papers at once like this [0], [0].

10.2 Contributions

This section should describe the author's contributions to the field of research.

10.3 Mathematical notation

It is a good practice to define basic mathematical notation in the introduction. See Table 10.1 for an example.

$\mathbf{x}$, $\boldsymbol{\alpha}$	vector, pseudo-vector, or tuple
$\hat{\mathbf{x}}$, $\hat{\boldsymbol{\omega}}$	unit vector or unit pseudo-vector
$\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3$	elements of the *standard basis*
$\mathbf{X}, \boldsymbol{\Omega}$	matrix
$\mathbf{I}$	identity matrix
$x = \mathbf{a}^\intercal \mathbf{b}$	inner product of $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^3$
$\mathbf{x} = \mathbf{a} \times \mathbf{b}$	cross product of $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^3$
$\mathbf{x} = \mathbf{a} \circ \mathbf{b}$	element-wise product of $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^3$
$\mathbf{x}_{(n)} = \mathbf{x}^\intercal \hat{\mathbf{e}}_n$	n^{th} vector element (row), $\mathbf{x}, \mathbf{e} \in \mathbb{R}^3$
$\mathbf{X}_{(a,b)}$	matrix element, (row, column)
x_d	x_d is *desired*, a reference
$\dot{x}, \ddot{x}, \dddot{x}, \ddddot{x}$	1^{st}, 2^{nd}, 3^{rd}, and 4^{th} time derivative of x
$x_{[n]}$	x at the sample n
$\mathbf{A}, \mathbf{B}, \mathbf{x}$	LTI system matrix, input matrix and input vector
SO(3)	3D special orthogonal group of rotations
SE(3)	*SO(3)* $\times \mathbb{R}^3$, special Euclidean group

Table 10.1: Mathematical notation, nomenclature and notable symbols.

11 How to write thesis in LaTeX

11.1 Versioning with git

Write the LaTeX in such a way that it could be versioned by git, which will help when collaborating with other people. This means writing **one sentence per line**. Even when you use third-party platforms, such as the OverLeaf, you can still share the repository through Git.

11.2 Forming paragraphs

A paragraph is formed in LaTeX by an uninterrupted block of non-empty lines. It is recommended to keep a single sentence per line (helps with versioning using git). A new paragraph is started after an empty line.

This is a new paragraph. It is strongly recommended to **avoid** the use of the *newline* (\\) feature of LaTeX for forming paragraphs as it doesn't format the new paragraph properly (no space at beginning of the new paragraph).

11.3 Linguistic anti patterns

Narrative

We recommend to write your thesis in plural form of the first-person narrative in combination with passive tense, e.g.:

- We discourage the use of any other form, and/or
- any other form is discouraged, but **not**
- I discourage you from using the first-person narrative.

Moreover, avoid "instructional" or "teacher"-like style of writing, such as "Now, we multiply the matrix $\mathbf{A}$ by the scalar c to get the scaled matrix $\mathbf{B}$." A better way of writing the same information would be e.g. "Now, the scaled matrix $\mathbf{B}$ is obtained by multiplying the matrix $\mathbf{A}$ by the scalar c."

Pronouns

The use of pronouns (it, this, they) is strongly **discouraged**. Although, pronouns make it easier for you as a writer to form the flow of the text, pronouns also make it much more difficult for the reader to follow the text. The reader is forced to retain more of the context to substitute and understand what the author meant. Moreover, pronouns can easily become vague (there is more than one way how to interpret them) and can become invalid while making editorial changes to the text, i.e., when moving sentences around. A technical text should be written in a way that makes it as easy to read and comprehend as possible and as hard to misunderstand or misinterpret as possible at the same time.

11.4 Mathematical notation with LaTeX

Take care to use the correct mathematical symbols and common ways of denoting mathematical concepts. Use bold fonts to visually distinguish vectors and matrices ($\mathbf{x}$, $\mathbf{A}$) and scalars (k, N).

Common errors

A frequent error, carried over from programming languages, is using the asterisk symbol ($*$) to denote multiplication. The asterisk correctly denotes convolution. Similarly, the cross sign ($\times$) typically denotes the cross product (it can also used for stating dimensions, such as $10\,\mathrm{m} \times 10\,\mathrm{m}$) and thus should not be used for scalar multiplication. In English mathematical notation, **scalar multiplication is typically not denoted at all**.

This custom may sometimes make it unclear whether a sequence of letters denotes multiplication of several scalars or a multi-letter variable, such as

$$T = T0 + coeffmeas, \tag{11.1}$$

where the variables in this hypothetical equation are T, $T0$, $coeff$ and $meas$. For this reason, **avoid using multi-letter variable naming** and strive to denote mathematical variables with single letters optionally a with lower or upper index, or other modifiers (`\hat`, `\bar`, etc.). The equation above could be modified to be

$$T = T_0 + cT_{\text{meas}}. \tag{11.2}$$

If the multiplication is still unclear (e.g. when multiplying many single-letter scalars), the `\cdot` symbol may be used such as

$$P \cdot V = n \cdot R \cdot T. \tag{11.3}$$

Equations

Mathematical equations should be numbered and should be a part of a sentence. For example, a discrete LTI system update is described as

$$\mathbf{x}_{[k+1]} = \mathbf{A}\mathbf{x}_{[k]} + \mathbf{B}\mathbf{u}_{[k]}, \tag{11.4}$$

where $\mathbf{x}_{[k]} \in \mathbb{R}^m$ is the state vector at the sample k, $\mathbf{u}_{[k]} \in \mathbb{R}^n$ is the input vector, $\mathbf{A} \in \mathbb{R}^{m\times m}$ is the main system matrix, and $\mathbf{B} \in \mathbb{R}^{m\times n}$ is the system input matrix. Proper punctuation should be used after the equation, as if it were an ordinary object in the sentence.

Do not put any empty lines before the equation. If the sentence that the equation is a part of continues after the equation (as is the case here), do not put empty lines after the equation either. That would create a new paragraph mid-sentence. For an example of how not to do it, the equation

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{11.5}$$

describes the logistic function often used in machine learning. Observe how a new paragraph is created for the equation and then for this block of text (compare with the proper typesetting above). Not only does this not look correct, it may also cause incorrect page breaking.

11.5 Using footnotes

Do not be afraid to use footnotes for additional information, such as http links[1]. We use footnote links whenever we want to *point* to a website, rather then to cite it as a source. Like with everything, do not overdo it.

11.6 Referencing document elements

LaTeX allows you to dynamically reference to parts of the documents, such as

- figures: Fig. 11.4, Figure 11.4,
- equations: eq. (11.4), (11.4),
- code: Lst. 11.1,
- and any other object that can contain a `\label`.

Check the section in the `document_setup.tex` that contains useful macros for unifying the references:

```
\newcommand{\reffig}[1]{Fig.~\ref{#1}}
\newcommand{\reflst}[1]{Lst.~\ref{#1}}
\newcommand{\refalg}[1]{Alg.~\ref{#1}}
\newcommand{\refsec}[1]{Sec.~\ref{#1}}
\newcommand{\reftab}[1]{Table~\ref{#1}}
\newcommand{\refeq}[1]{\eqref{#1}}
```

Listing 11.1: LaTeX macros for referencing to document elements.

11.7 Abbreviations with Acronym

Abbreviations are handled by the *acronym* package. Example sentence with abbreviations: "**UAV!** is a flying vehicle that commonly uses **LiDAR!** (**LiDAR!**) and **GPS!** (**GPS!**) receiver". Note that the acronyms are only explained once in the document by default. It is good practice to re-explain acronyms used both in the abstract and the rest of the document as the abstract is often presented separately. This can be achieved by resetting the internal status of the acronyms ("forgetting" that they were explained) using the `\acresetall` command after the abstract. Please, read the documentation[2].

11.8 Units of measurements with Siunitx

Typesetting of units has never been more accessible with the Siunitx package. Acceleration is measured in $\mathrm{m\,s^{-2}}$. Gravity accelerates objects at a rate $\approx 9.81\,\mathrm{m\,s^{-2}}$ near the sea level. You can define your units if you want.

11.9 Hyphens and dashes

Hyphens and dashes are the various form of the symbol "-" used in many situations. There are also various ways how to typeset the symbol in LaTeX.

- The *hyphen* is used to compound words, e.g., "the eye-opener". The hyphen is typeset as a single *minus*/*hyphen* character: `-`.

[1] This repository: https://github.com/ctu-mrs/thesis_template.
[2] Acronym package: http://mirrors.ctan.org/macros/latex/contrib/acronym/acronym.pdf

- The *en-dash* is used to specify ranges of values, e.g., "between 2–10". The en-dash is typeset as two consecutive hyphens characters: `--`.
- The *em-dash* is used to separate complex sentences in place of commas, parenthesis and colons — each with its particular rules. The em-dash is typeset as three consecutive hyphens characters: `---`.

Check the https://www.thepunctuationguide.com/ for all the details.

11.10 Double quotation marks

"Double quotes" in English are composed of a pair of opening (") and closing (") symbols. The opening symbol is typeset as two backtick characters: `` ` ` `` (typically below the `Esc` key on the English keyboard), and the closing quotes as two apostrophes: `''`. The LaTeX engine will convert them automatically to the opening and closing symbols. A more robust solution is to use the `csquotes` package and the `\enquote` command which also takes care of nested quoting and other peculiarities.

11.11 2D Diagrams with Tikz

Tikz is a powerful tool for drawing 2D (and 3D) shapes and diagrams. Check the documentation and examples: https://www.overleaf.com/learn/latex/TikZ_package. The benefit of using *Tikz*, instead of some other third-party drawing program, are:

- fonts are the same as in LaTeX,
- you can typeset math in LaTeX,
- you can use references to other parts of your document,
- you can version the image in git,
- the images are easily adjustable while editing your document.

Check Fig. 11.1 for example.

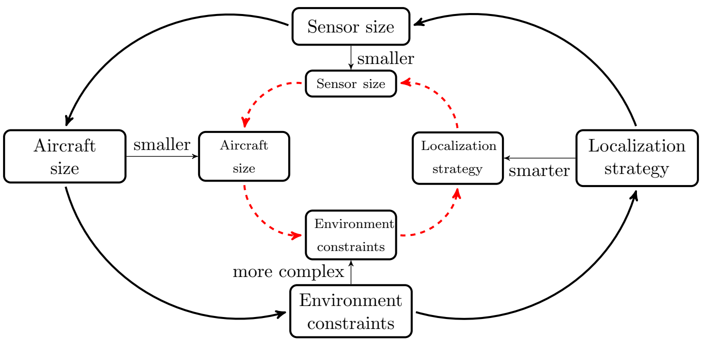

Figure 11.1: Example of a 2D diagram using tikz *PGFPlots*.

11.12 Data plots with PGFPlots

PGFPlots produces nice 2D and 3D data plots from data stored in CSV. The plot parameters can be versioned and easily adjusted by editing the plot definition file.

- Documentation and manual: https://ctan.org/pkg/pgfplots
- Compile the plots individually and then include the pdfs because it can take longer.
- Example located in `fig/plots/example_plot`, see Fig. 11.2.

- You could include the latex file directly. However, it will take longer to compile, and platforms such as Overleaf can have a problem with that.

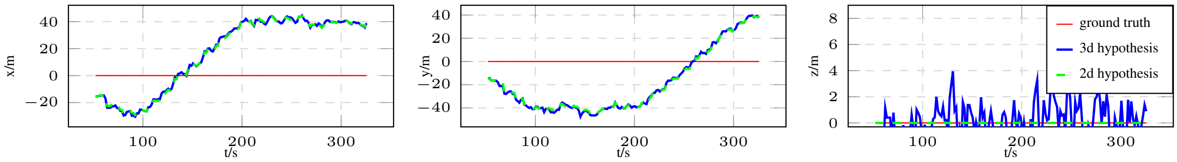

Figure 11.2: Example of a 2D plot using *PGFPlots*.

11.13 3D Plots with Sketch

Sketch is a tool for defining a 3D scene using simple descriptive language. The 3D scene is then converted to *Tikz*, which is later compiled to pdf. The benefits of using *Sketch* are similar to using *Tikz*: LaTeX fonts, versioning using git, and cleanness of the result. See the example image in Fig. 11.3.

- Documentation and manual: http://www.frontiernet.net/~eugene.ressler/
- Cross-compilation from *Sketch* to *pdf* using the `fig/sketch/compile_sketch.sh` script.

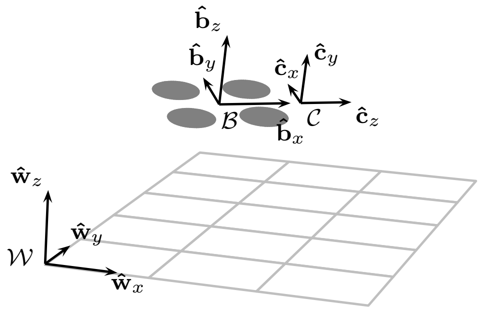

Figure 11.3: Depiction of the used coordinate systems. The image was drawn using *Sketch*.

11.14 Image collages with Subfig

We recommend using the `subfig` package, which provides the `\subfloat` command. It is more versatile than the simpler `subcaption` package. Check Fig. 11.4 for an example.

11.15 Citations with Biblatex

Biblatex is probably the most powerful citation package for LaTeX. It consumes the standard `.bib` file. However, it can sort and filter the citations using the `keywords` tag. Citing references is done using the `cite` command, e.g., [0]. You can also define some nice citation boxes, such as this one:

[0] T. Baca, M. Petrlik, M. Vrba, V. Spurny, R. Penicka, D. Hert, *et al.*, "The MRS UAV System: Pushing the Frontiers of Reproducible Research, Real-world Deployment, and Education with Autonomous Unmanned Aerial Vehicles," *Journal of Intelligent & Robotic Systems*, vol. 102, no. 26, pp. 1–28, 1 May 2021

(a) A UAV, the T650 model.

(b) Another UAV, again, the T650 model.

Figure 11.4: The caption should mention both subfigures, the Fig. 11.4a and the Fig. 11.4b. You can just refer to them as (a) and (b) in the main Figure's caption, but beware, you need to keep it correct as you edit.

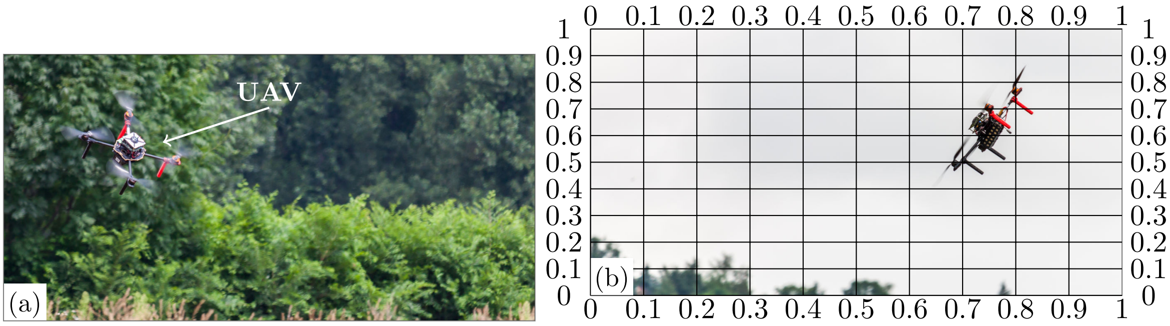

Figure 11.5: Example of using Tikz for image overlays. (a) shows a final product, (b) shows a grid useful for nailing down the coordinates.

11.16 Image overlays with Tikz

Tikz is very useful to create custom image overlays. The overlay can be set such that the image is spanned by Cartesian coordinates $(x, y) \in [0, 1]^2$ Example can be seen in Fig. 11.5.

11.17 General tips

In general, strive to make the paper easy to read and understand, and hard to misunderstand or misinterpret. Here are some more specific tips on how to achieve that (and other general suggestions).

- **Be consistent.** This applies in all contexts. For example, if you decide to use the name "LiDAR", do not mix it with "LIDAR" or "Lidar", do not mix different mathematical notations, ensure your Figures have the same style and use the same graphics for the same concepts, etc.
- After you finish writing or modifying any of:
  - a sentence,
  - a paragraph,
  - a section/chapter,
  - the whole paper/thesis,

  re-read it to make sure that it makes sense, it is coherent and correct, and doesn't contain typos.

- If you're using a LLM-based tool (ChatGPT etc.) for grammar-proofing or even formulation of sentences, **do not just copy-paste its response** to your query. The previous rule applies doubly here. LLMs tend to often produce confident-sounding nonsense, sentences with reformulated duplicated content, or with a slightly changed meaning. They are a good tool to get inspiration to start writing about a subject, for grammar-checking, or for finding alternative, nice-sounding formulations, but they can lie or warp facts — take care when using them!

12 Conclusion

Summarize the achieved results. Can be similar as an abstract or an introduction, however, it should be written in past tense.

13 References

[0] T. Baca *et al.*, "The MRS UAV System: Pushing the Frontiers of Reproducible Research, Real-world Deployment, and Education with Autonomous Unmanned Aerial Vehicles," *Journal of Intelligent & Robotic Systems*, vol. 102, no. 26, pp. 1–28, 1 May 2021.

[0] T. Baca, G. Loianno, and M. Saska, "Embedded Model Predictive Control of Unmanned Micro Aerial Vehicles," in *IEEE International Conference on Methods and Models in Automation and Robotics (MMAR)*, IEEE, 2016, pp. 992–997.

[0] A. Benallegue, A. Mokhtari, and L. Fridman, "High-order sliding-mode observer for a quadrotor UAV," *International Journal of Robust and Nonlinear Control: IFAC-Affiliated Journal*, vol. 18, no. 4-5, pp. 427–440, 2008.

A Appendix A