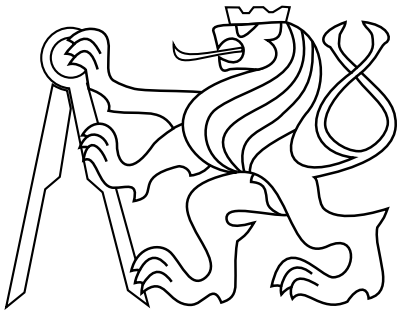
Czech Technical University in Prague

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 Unmanned Aerial Vehicles (UAVs) 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

**BDD** Behavior-Driven Development

**BOA** Business Objective Action

**GPS** Global Positioning System

**LiDAR** Light Detection and Ranging

**UAV** Unmanned Aerial Vehicle

# 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

**Definition**

Behavior-Driven Development (BDD) is a software development methodology that focuses on defining the behavior of a system in terms of business requirements. It encourages collaboration between developers, testers, and business stakeholders to ensure that the system meets the desired functionality and provides value to the end users.

### Business-Objective-Action

**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

### Cecil-Mono

### Roslyn

# 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 distinct custom attributes for defining data preparation methods and for utilizing prepared data within tests.
  - Ensure attributes for data preparation methods specify setup and teardown processes, supporting reuse across multiple test cases.
  - Design attributes to allow the injection parameterized data, enabling differentiation for tests requiring similar preparation but with varying input parameters.
  - Provide mechanisms for conditional test execution based on the prepared data state to ensure test reliability and flexibility.
- **Automated Test Data Handling:**
  - Develop a mechanism to automate the execution of data preparation methods before and after tests, ensuring a consistent and reliable test environment.
  - Ensure seamless preparation and injection of both simple and complex test data structures to support diverse testing scenarios.
- **Data Preparation Store:**
  - Develop a centralized store to maintain mappings of data preparation instances associated with test methods, enabling efficient and organized data management.
  - Ensure thread-safe access to support concurrent test executions in multi-threaded environments.
- **Integration with NUnit Test Lifecycle:**
  - Integrate custom behaviors into the test execution lifecycle to include steps for data preparation and cleanup.
  - Ensure that custom lifecycle hooks are compatible with parallel test execution to maintain consistency in multi-threaded scenarios.
- **Attribute Count Tracking:**
  - Develop a mechanism to track the execution of custom attributes and ensure all necessary data preparations are completed before test execution begins.
  - Avoid redundancy in data preparation by coordinating attribute behavior for tests with multiple attributes applied.
- **Service Provider Utilization:**
  - Use a service provider pattern to manage dependencies and services required for data preparation.

  - Support integration with dependency injection frameworks.
- **Interface Usage:**
  - Define clear and reusable interfaces for data preparation and cleanup logic to promote consistency and scalability across the framework.
  - Ensure interfaces allow flexibility for developers to implement custom preparation methods tailored to specific testing requirements.

**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.
- **Documentation and User Guidance:**
  - Provide comprehensive documentation, including user guides, examples, and FAQs, to simplify adoption and usage of the framework.
  - Include in-line comments within the codebase to explain key implementation details and design choices.
  - Offer troubleshooting steps and recommendations for common integration or usage challenges.

## 4.2 Framework Architecture

The architecture of the framework is carefully designed to ensure scalability, maintainability, and seamless integration with existing unit testing workflows.

The framework architecture is divided into two main components. The first component focuses on data preparation, while the second encompasses the test case, which includes the tests themselves. Except for specific framework elements, the test case retains the same components as those used in NUnit testing. The framework extends the functionality of the test case by defining whether it should be executed, specifying the framework services required for the test case, and identifying the prepared data to be used for individual tests.

Data preparation represents an additional feature compared to standard testing and

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

requires developers to become familiar with its implementation when using the framework. The design of the data preparation component is structured to support both class-level and method-level test data preparation. The primary task for developers is to create a class associated, for example, with a specific tested method. The class does not require a constructor unless the developer requires dependencies on specific services. Instead, it only needs to define methods for setting up the data and for cleaning up the prepared data after testing.

This architectural approach ensures that the framework provides added value through structured data preparation while maintaining compatibility with known NUnit-based test case practices. The framework's design is intended to be intuitive and easy to use, requiring minimal effort to adapt to the new data preparation features. The architecture is designed to be scalable and extensible, allowing developers to add new features and functionalities as needed. The framework's modular design ensures that each component can be easily replaced or extended without affecting the overall functionality of the system.

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* is designed to facilitate the management of test data and the configuration of test frameworks. Almost every attribute in this layer serves as a directive indicating that a particular test case or method requires the preparation and use of specific data. The attributes provide a structured way to manage data setup and cleansing and ensure that the necessary data conditions are met before and after test execution. The main reason to use attributes is to allow the programmer to extend the functionality of the NUnit framework and to ensure that there is no significant change in the process of how the tests are created and that the programmer does not need to spend too much effort to extend the knowledge of how the tests are created. The result is that testing will be almost the same as before the extension of the framework and that there will be no need to learn new procedures or approaches to testing.

### Attributes for Data Preparation

The *Attributes for Data Preparation* layer is designed to manage the setup and cleanup of test data within testing. The attributes serve as directives that indicate specific data requirements for test cases or methods. Provide a structured way to prepare, call, and remove prepared test data. Ensure that the test environment is properly configured and that the conditions for test data are met before, during, and after test execution. This layer provides a unified approach to data setup manipulation that enables precise control over data preparation management, which is essential to maintain robustness and maintainability of test procedures.

The attributes for data preparation for a class or method are different, but the data preparation structure is the same. This structure ensures a uniform approach to data preparation management, which is key to maintaining the robustness and maintainability of test practices. Instead of forcing programmers to implement the data preparation structure for each test method separately, it allows them to simply use the prepared data techniques and ensure that they are used correctly. Although the data preparation structure is the same for class-prepared and method-prepared data, there are differences in how these attributes are used and how they affect test scenarios.Attributes for preparing data for a class allow you to

define methods for preparing data that are necessary for testing the whole class. The attributes ensure that the data preparation methods are properly configured and initialized according to the test scenario. The data preparation method attributes, allow more precise control over the preparation of data specific to a particular test scenario. All attributes support parameterized data inputs, allowing resolution for tests that have similar preparation requirements but different input parameters. The addition of parameters ensures that the data preparation is appropriate to the specific test requirements and that the data preparation can be dynamically changed according to the needs of the test.

The data preparation structure is created by the programmer independently. Consists of identification for which class or method the programmer wants to prepare the data to what the given attribute will help and enables the programmer to arrange the given structure to the class or method.

#### Testing Attributes

Test attributes help define the test scenarios in which test data needs to be prepared and used. Each of the attributes provides unique properties that allow for different data processing configurations. The base test attribute determines if the framework is to be used at all for a given test case. Other attributes are dedicated to calling data preparation for different tests as well as subsequent data rollback. By defining the attributes for a test, it is possible to specify more precisely how the data should be prepared and used. In this way it is possible to adapt the data processing to the needs of the test and ensure that the data is prepared correctly. Ensure that the necessary services are configured and initialized according to the unique requirements of each test scenario.

The attributes are divided into two main types. The first is an attribute that identifies that a specific test data is to be used for a given test method or class. Recognizable by the *For* expression. The second type of attribute allows to call the prepared data directly without knowing which method or class is used in the test. The design allows the programmer to easily access the prepared data during test execution. Enables precise selection of data sources needed for a given test.

In the above cases, we are still discussing calls to prepared data that have no parameters or don't need them. An extended version of these attributes are attributes that allow the use of parameters to define the preparation and manipulation of test data. The attributes allow more precise control over how the data is manipulated based on the specified parameters. Such a structure ensures a systematic approach to test data management. The attributes containing the *Params* expression in the name.

#### Attributes of prepared data

Interfaces provide a guaranteed direction and enforce adherence to a specified structure; however, they can limit a programmer's flexibility. To ensure that developers retain sufficient freedom while allowing prepared data to conform to the structure they envision, attributes were introduced. The attributes, designed to facilitate data preparation for tested methods or functions, indicate whether the operation involves the deployment of prepared data or its removal.

The attributes are defined before the method responsible for handling the data. Their primary benefit lies in granting developers the ability to incorporate input parameters, enabling precise customization of data preparation and manipulation processes. The approach

ensures data preparation and processing processes remain consistent with testing requirements, while allowing developers to tailor them to the specific needs of individual test scenarios.

- **Interface Layer**

The *Interface Layer* consists of the interfaces that cover the preparation and management of test data and related services. The Layer ensures consistency and modularity in the way data is prepared for test scenarios, supporting a structured approach.

Interfaces for preparing data for testing are designed for the class and method levels. These interfaces provide non-parametric methods for data staging and subsequent data cleansing, i.e the method to be run before and after the test.

Interfaces for preparing data for testing are designed for the class and method levels. These interfaces provide non-parametric methods for data staging and subsequent data cleansing, i.e the method to be run before and after the test.

In addition to data preparation, the interfaces layer contains mechanisms for managing test services. The interfaces extend the test case and add additional functionality to the framework. One of them is used to register and initialize services in the container within the test framework. The interface provides a structured resource that the classes for preparing the data accept and can use according to the developer's thinking. Another interface is for defining the databases to be used in the test case.

TODO

- **Data Handling Layer**

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

The core of the *Data Handling Layer* is a system that automates the execution of methods for deploying prepared data and cleaning. The handler enforces a deterministic order of execution, ensuring that all preparatory logic is executed before the test is run and that cleanup operations are then reliably performed. In addition, the layer contains exception handling mechanisms that guarantee proper cleanup even in situations where tests fail unexpectedly.

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

- **Integration Layer**

*Integration Layer* aims to increase the efficiency of attribute execution and promote modularity through service-oriented design. The `Attribute Count Storage` is critical in ensuring proper execution of data preparation. The `Attribute Count Storage` monitors the invocation of attributes applied to test methods, preventing duplicate execution of methods for preparing and cleaning test data when multiple attributes are stored on top of each

other. This mechanism ensures that all preparatory logic runs as expected while maintaining efficiency.

To further isolate the preparatory logic, is used the `Provider Store` framework, which acts as a service provider. This component supports dependency injection principles, allowing developers to dynamically register and load services in the test. This approach allows the framework to seamlessly integrate with existing dependency injection frameworks, such as `Microsoft.Extensions.DependencyInjection`.

## 4.3 NUnit Integration

The framework achieves seamless integration with NUnit by extending its lifecycle through the use of custom hooks. We facilitate this integration by implementing the `ITestAction` interface, which provides NUnit attributes with methods that enable the execution of logic both prior to and following the execution of a test.

Methods within the `ITestAction` interface, such as `BeforeTest` and `AfterTest`, are overridden to introduce steps for preparing data before test execution, as well as performing post-test cleanup and managing other services invoked and instantiated by the framework. This approach ensures that the data preparation and teardown processes are smoothly incorporated into the test execution flow without disrupting the inherent structure of NUnit's testing lifecycle.

For enhanced control over test behavior, the test targets are specified within the attributes, ensuring that the custom logic is applied to individual test methods. Moreover, the architecture is designed to support extensibility, enabling the targeting of entire test classes or namespaces, thereby allowing developers to apply bulk-processing logic as needed.

### Parallel Execution Support

TODO 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.

# 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 [1].

## 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 [3]. You can also cite multiple papers at once like this [1], [2].

## 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 $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^3$ |
| $\mathbf{x}_{(n)} = \mathbf{x}^\intercal \hat{\mathbf{e}}_n$ | $\mathrm{n}^{\mathrm{th}}$ 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 |
| $\dot{x}, \ddot{x}, \dddot{x}, \ddddot{x}$ | $1^{\mathrm{st}}$, $2^{\mathrm{nd}}$, $3^{\mathrm{rd}}$, and $4^{\mathrm{th}}$ 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

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

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 Light Detection and Ranging (LiDAR) and Global Positioning System (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.

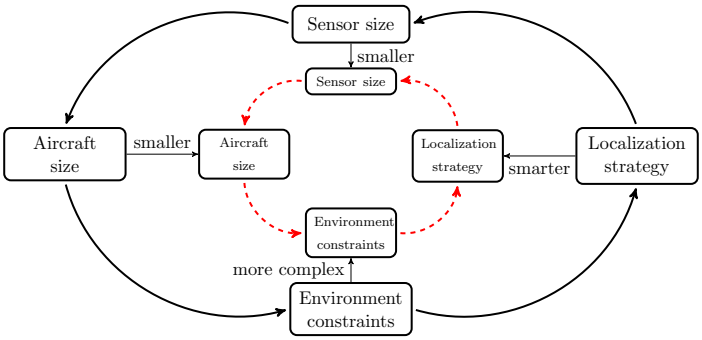



Figure 11.1: Example of a 2D diagram using tikz *PGFPlots*.

## 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.

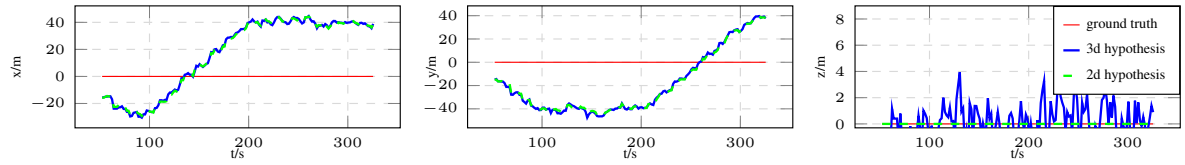



Figure 11.2: Example of a 2D plot using *PGFPlots*.

## 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.

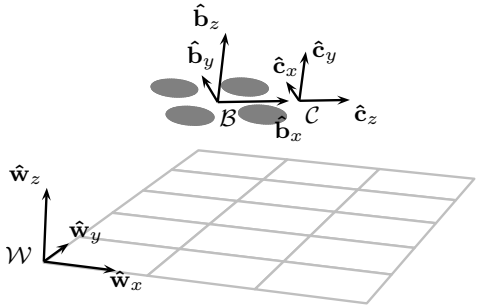



Figure 11.3: Depiction of the used coordinate systems. The image was drawn using *Sketch*.

## 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., [1]. You can also define some nice citation boxes, such as this one:

[1] 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.

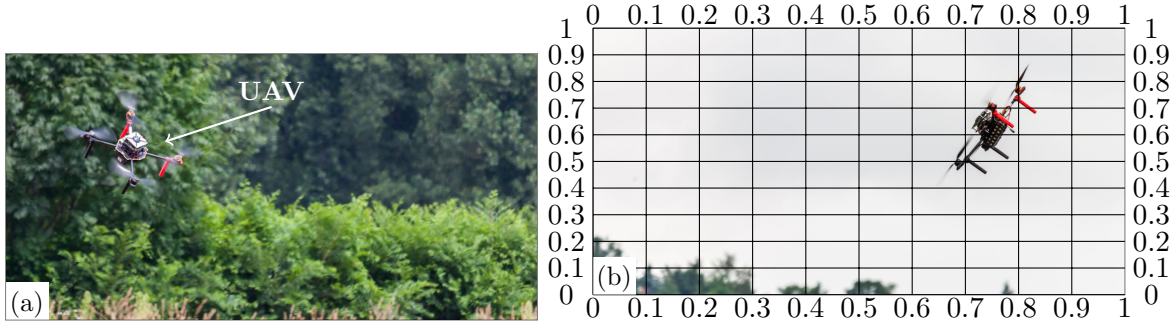



Figure 11.5: Example of using Tikz for image overlays. (a) shows a final product, (b) shows a grid useful for nailing down the coordinates.

## 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

[1] 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.

[2] 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.

[3] 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