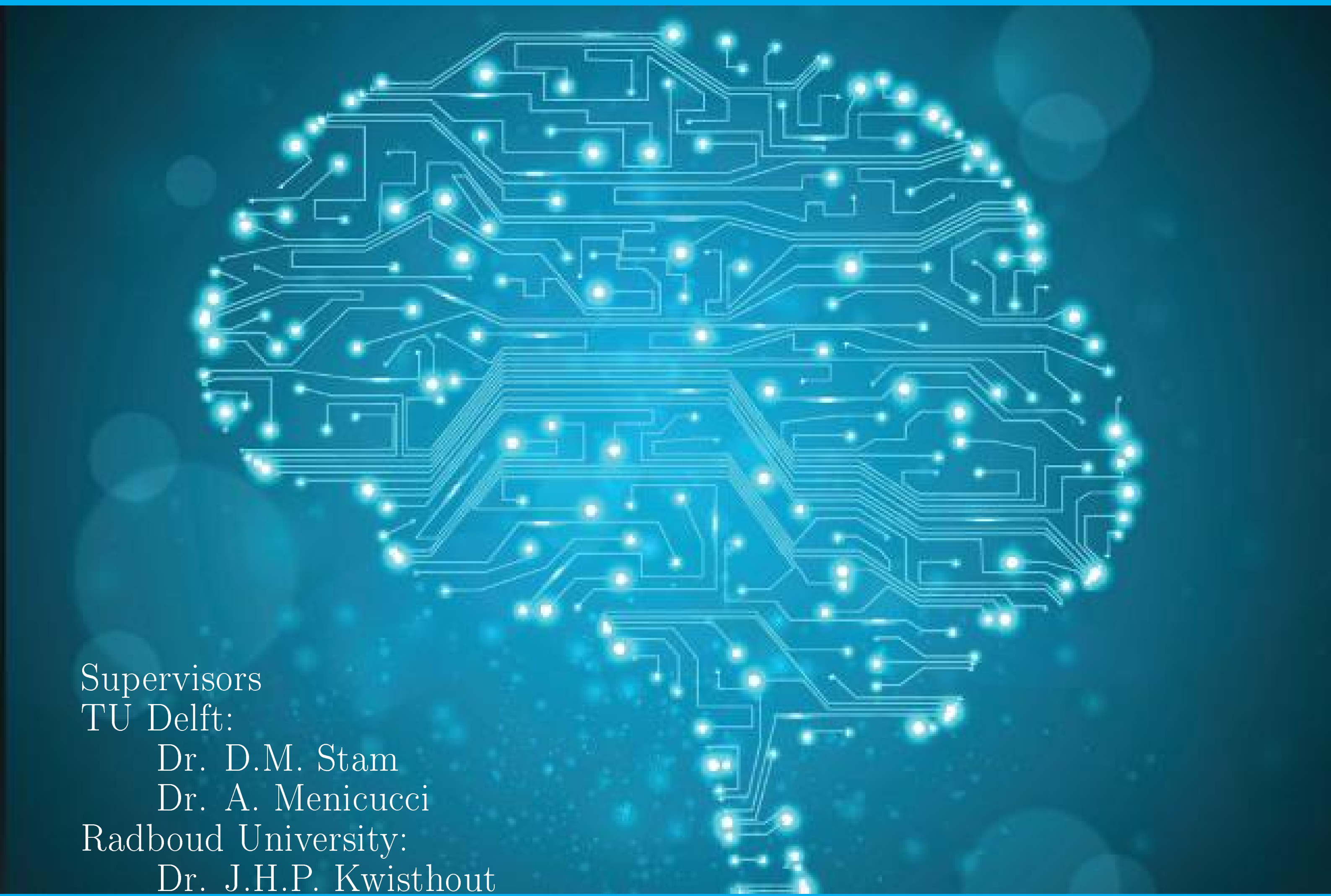
Leveraging brain adaptation to increase radiation resistance of neuromorphic space hardware

AE5810 Thesis Project: baseline report

Akke Toeter



Supervisors
TU Delft:
Dr. D.M. Stam
Dr. A. Menicucci
Radboud University:
Dr. J.H.P. Kwisthout

Leveraging brain adaptation to increase radiation resistance of neuromorphic space hardware

AE5810 Thesis Project: baseline report

by

Akke Toeter

Systems Engineering component of the AE5810 Thesis
at the Delft University of Technology.

Student number:	1507958
Project Planning duration:	September 20, 2020 – September 24, 2021
Version:	0.1

TU Delft

Contents

Chapter 1

Introduction

This document presents the baseline for the AE5810 Thesis Project of the Space Flight Master at the Faculty of Aerospace Engineering of Delft University of Technology and the SOW-MKI92 Research Project of the Master in Artificial Intelligence at the faculty of Social Sciences of Radboud University. Its purpose is to identify the 2-5 most feasible design options that can be used to determine whether the principle of brain adaptation can be leveraged in neuromorphic space hardware.

The baseline report presents the Functional Flow Diagram (FFD) and Functional Break-down Diagram (FBD) in chapter 2 and chapter 3 respectively. These function descriptions of the system that is to be designed, is then used to generate the Requirements Discovery Tree (RDT) in chapter 4. Next, the resource allocation and budget breakdown presented in chapter 5. This is followed by the technical risk assessment in chapter 6. From the RDT, the Design Options Structuring Tree (DOT) is generated in chapter 4. Contingency management is applied in chapter 8. A market analysis is presented in chapter 9, and the sustainable development strategy is presented in chapter 10. To ensure this work is performed with sufficient quality, the reporting and quality control is presented in chapter 11. The baseline is concluded in chapter 12.

Chapter 2

Functional Flow Diagram

To gain insight in the system that is to be designed, a Functional Flow Diagram is generated. This FFD presents the high level functions that the system should be able to perform, in chronological order. These functions are presented in the flow diagram of fig. 2.1.

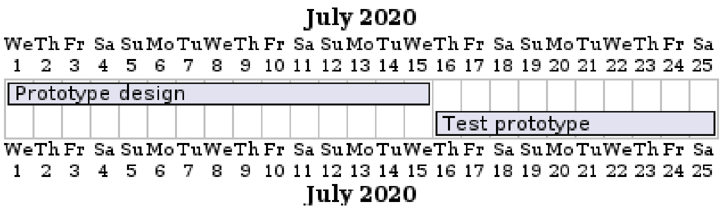


Figure 2.1: A functional flow diagram with the high-level functions of the system that is to be designed.

1. Initialise & start space-related function on neuromorphic architecture without brain adaptation implementation.
2. Initialise & start space-related function on neuromorphic architecture with brain adaptation implementation.
3. Endure modelled space radiation on neuromorphic architecture.
4. Measure space-related function performance without brain adaptation implementation.
5. Measure space-related function performance with brain adaptation implementation.
6. Report any performance difference between with- and without brain adaptation.
7. Determine significance of difference.

From these high level functions, a more detailed functional breakdown diagram is generated.

2.1. Detailed Functional Flow Diagram

1. Initialise & start space-related function on neuromorphic architecture without brain adaptation implementation.
   (a) Boot/initialise neuromorphic architecture
   (b) Load function.
   (c) Load function data.
2. Initialise & start space-related function on neuromorphic architecture with brain adaptation implementation.
   (a) Boot/initialise neuromorphic architecture

   (b) Load brain adaptation.
   (c) Load space-related function.
   (d) Load space-related function data.

3. Render and endure modelled space radiation on neuromorphic architecture.
   (a) Determine simulated space radiation pattern of neuromorphic space architecture.
   (b) Expose neuromorphic architecture to simulated space radiation pattern.
   (c) Complete space-related function.
   (d) Return results of space-related performance.

4. Measure space-related function performance without brain adaptation implementation.
   (a) Retrieve space-related function output.
   (b) Convert space-related function output to score.

5. Measure space-related function performance with brain adaptation implementation.

6. (a) Retrieve space-related function output.
   (b) Convert space-related function output to score.

7. Report difference between the scores of the architectures with- and without brain adaptation.

8. Determine significance of difference.

2.1.1. Description

1. Initialise & start space-related function on neuromorphic architecture without brain adaptation implementation.
   - To run a function on the neuromorphic architecture, it will have to be booted and initialised. The function that will be ran on the neuromorphic architecture is space related, to increase the level of representativeness of this study, in terms of space applications. The initalisation allows for loading the space related function that is to be executed. Additionally, the space related function may require (training) data on which it is ran. For example, a Martian rover that has a function that identifies rocks in its environment, may be partially simulated by loading a (labelled) dataset of Martian images.

     To run the function without brain adaptation implementation, allows for the creation of a baseline to which the brain adaptation performance can be compared.

2. Initialise & start space-related function on neuromorphic architecture with brain adaptation implementation.
   - The intialisation of the brain adaptation implementation can occur, before, during or after the loading of the space related function. Which of these options is selected depends on the more detailed design process.

3. Endure modelled space radiation on neuromorphic architecture.
   - The radiation robustness of the neuromorphic architecture can be tested by exposing the neuromorphic architecture to the radiation that it would experience in a space application. To determine what this radiation is, a relevant space mission and space function are selected. The time, position and orientation of the spacecraft in such a mission is then used to derive the radiation pattern to which the radiation may be exposed. This radiation is pattern is then used to determine to which (simulated) radiation the neuromorphic architecture will be exposed.

4. Measure space-related function performance without brain adaptation implementation.

- The performance of the space related function without brain adaptation implementaiton on the neuromorphic architecture is measured before, during and/or after radiation exposure. Which of these measuring moments are used, is still to be determined by the detailed design process. This measurement then serves as a comparison baseline to put the impact of the brain adaptation implementation into context.

5. Measure space-related function performance with brain adaptation implementation.
6. - Once a baseline for comparison is established, the space related function can be ran again on the neuromorphic hardware, whilst being exposed to radiation. In this second setting, the brain adaptation implementation is used in an attempt to increase the radiation robustness of the neuromorphic architecture. The performance of the space related function is then measured before, during and/or after radiation exposure. Which of these measuring moments are used, is still to be determined by the detailed design process.
7. Report any performance difference between with- and without brain adaptation.
8. - If any performance difference is observed between the space related function with- and without brain adaptation implementation, it will be computed and stored.
9. Determine significance of difference.
10. - An analysis is performed to determine the level of significance of any observed difference.

Chapter 3

Functional Breakdown Diagram

This section presents the functional breakdown diagram of the system that is designed in this thesis project. This FBD is generated using the detailed functional flow diagram of section 2.1. The FBD presents the activities in an hierarchical style.

1. Run space-related function on neuromorphic architecture.
   (a) Initialise neuromorphic architecture.
   (b) Optional: Load brain adaptation implementation.
   (c) Load space related function.
   (d) Load space related function data.
   (e) Run space related function.
   (f) Complete running space related function.
   (g) Retrieve space related function outputs.
   (h) Convert space related function outputs to performance score.
   (i) Report difference between the scores of the architectures with- and without brain adaptation.
   (j) Determine significance of difference.
2. Render and endure modelled space radiation on neuromorphic architecture.
   (a) Generate simulated space radiation pattern of neuromorphic space architecture.
   (b) Expose neuromorphic architecture to simulated space radiation pattern.
   (c) Optional: model architecture-radiation interaction.
   (d) Optional: measure architecture-radiation interaction.

Chapter 4

Requirements Discovery Tree

This section presents an overview of the requirements that are identified within this thesis project. Its purpose consists of listing the requirements that drive the design, identifying killer requirements and presenting an overview of the project requirements. section 4.1 contains the mission need statement of this project. Next, the stakeholder requirements are identified in section 4.2. The top level requirements are presented in section 4.3, and the key requirements are presented in section 4.4. From these combined requirements, the Requirements Discovery Tree is drafted in section 4.5.

4.1. Mission Need Statement

The mission need statement is generated in the project plan phase, and is included in this section again to provide the context of the requirement derivation process. The MSN is:

Increase the radiation robustness of neuromorphic space hardware by leveraging the principle of brain adaptation in neuromorphic hardware.

4.2. Stakeholder Requirements

The following stakeholder requirements are identified:

- **STKH-UNI-01** - The research that is performed shall be reproducible.
- **STKH-UNI-02** - The sustainability of the design concepts shall be taken into account in the design trade-off process.
- **STKH-SPACEBRAINS-01** - Achieve results towards the REACH research by Q2 2022.
- **STKH-SPACEBRAINS-01-a** - Achieve results towards the REACH research by either: 2022-03-01, 2022-04-07, 2022-07-01.
- **STKH-ICONS-01** - Submit paper documenting research results before April 15th, 2022.
- **STKH-ICONS-01-a** - Submit full paper of 6-8 pages, presenting original research, or submit short paper of 3-4 pages that has preliminary results.
- **STKH-ICONS-02** - Upon acceptance for a presentation, submit presentation before July 27th, 2022.
- **STKH-RADBOUD-01** - The research proceedings shall be documented and submitted in a format accepted by Dr. J.H.P. Kwisthout before the SOW-MKI92 Research Project is completed.
- **STKH-RADBOUD-02** - The research for the SOW-MKI92 Research Project shall be performed using at least 28 EC of work.
- **STKH-Delft-01** - The research proceedings shall be documented and submitted in a format accepted by Dr. D.M. Stam and Dr. A. Menicucci before the AE5810 Thesis Project is completed.
- **STKH-Delft-02** - The research for the AE5810 Thesis Project shall be performed using at least 42 EC of work.

4.3. Top Level Requirements

The following requirements for this thesis project are identified:

- **TECH-01** - Technology Readiness Level (TRL) of the used technology shall be at least TRL 4 [-].
- **TEST-01** - Radiation robustness shall be tested in terms of algorithmic performance.
- **TEST-02** - The radiation tests shall be technically and economically feasible.
- **TEST-03** - The radiation tests results shall be generated before September 2022.
- **SUS-01** - Sustainability management shall be integrated in each phase of this project.
- **SUS-02** - Sustainability shall be assessed in the design trade-off process.
- **SAF-01** - All participants involved in testing, integrating and operations shall not be exposed to serious danger.
- **ESA-01** - Throughout this project quarterly reports shall be provided to the SpaceBrains foundation and the European Space Agency (ESA).

4.4. Identification Key Requirements

The key requirements are requirements that can render the project infeasible, requirements that drive the design, and/or requirements that induce high risk to project success. Within this thesis project, the following key requirements are identified:

1. **STKH-SPACEBRAINS-01** - Achieve results towards the REACH research by Q2 2022.
2. **TECH-01** - Technology Readiness Level (TRL) of the used technology shall be at least TRL 4 [-].
3. **TEST-03** - The radiation tests results shall be generated before September 2022.

The **STKH-SPACEBRAINS-01** requirement significantly drives the design space, as physical radiation tests are not deemed feasible within the given timeframe and project constraints. **TEST-03** implies a strict time constraint that induces significant risk to project success as it limits the amount of time available for development and scheduling. **TECH-01** significantly drives the design as it limits the design space to concepts that rely only on technologies of TRL 4 and higher.

4.5. Requirements Discovery Tree

A requirements' discovery tree is currently omitted due to time constraints.

Chapter 5

Resource Allocation and Budget Breakdown

Chapter 6

Technical Risk assessment

Chapter 7

Design Option Structuring Tree

The purpose of the design option tree is to find a feasible design option that can satisfy the requirements. This selection process can be an iterative process in case no feasible design option is found in the initial design option tree. By positioning the design options in a tree format, their hierarchical structure becomes visible. The tree is structured such that the most impactful decisions are selected at the top of the tree, whereas more detailed decisions are made at lower levels of the tree. Several subsections are created to Figure 7.1 presents the design option tree for this thesis. The nodes in the tree represent design options and child leafs represent design options within a parent design option. For example, a choice may be made to use digital neuromorphic hardware, and within that design space, a particular chip may be selected. Multiple (parallel) design options may be selected. For example, a softwarematic and hardwarematic implementation may be chosen.

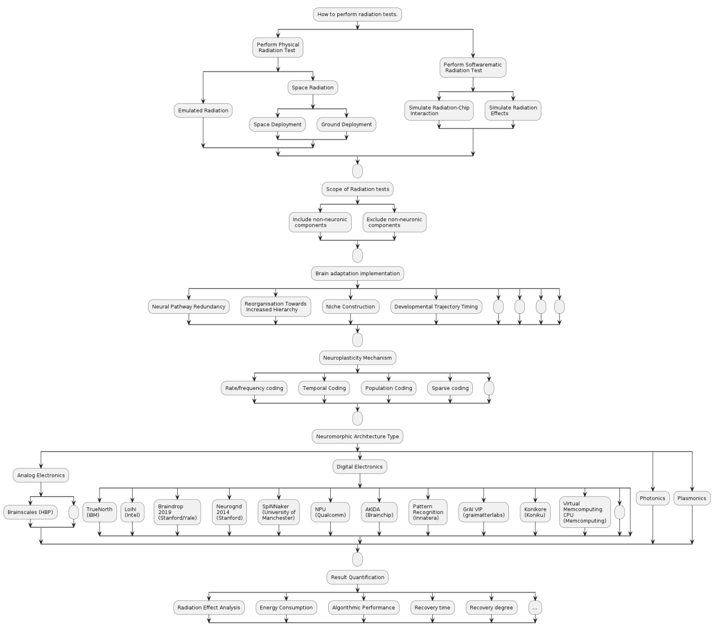


Figure 7.1: A functional flow diagram with the high-level functions of the system that is to be designed.

7.1. Pruning

With the design option tree generated, it can be pruned of options that are considered infeasible. This is done using the knowledge base that was generated in the literature study and using the requirements identified in chapter 4. The pruning process will be performed from top to bottom, which matches from high- to low hierarchical design choices.

7.1.1. Tested Functionality

At the time of writing, no specific function that is used for testing, can be eliminated.

7.1.2. How To Perform Radiation Tests

1. Starting with key requirement: **STKH-SPACEBRAINS-01** and **STKH-ICONS-01**, it is possible to eliminate the physical radiation test design option(along with its children), before the ICONS deadline of April 15th, 2022. Given the full scope of the thesis, and the **TEST-03** requirement which implies a test deadline before September 2022, a physical radiation test is still considered feasible before that time. Hence, instead of a complete termination (red), it is turned orange.

2. Continuing with the **STKH-SPACEBRAINS-01** requirement, it is considered infeasible to do a full simulation of radiation effects on the hardware components, before the ICONS deadline of April 15th, 2022. Hence, also this option will be coloured orange. Most neuromorphic chips in the DOT are proprietary, with many of the chip designs not being publically available. Since that makes it difficult to determine what the radiation effects will be on the hardware components of the chip, and how those effects, such as single-event upsets, would propagate towards influencing neurons and/or synapses. Therefore, this option is not considered feasible before April 15th 2022. It may be possible to contact manufacturers to ask how the radiation influences the neuronal- and synaptic properties. If such research is performed, it may be applied to simulate the neuromorphic hardware-radiation interaction softwarematically with sufficient accuracy to produce meaningful results.

7.1.3. Scope Of Radiation Tests

1. For the same as the last enumerated point of section 7.1.2, including the non-neuromorphic components is not considered feasible for before the ICONS deadline of April 15th, 2022. Hence, this element is also coloured orange.

7.1.4. Brain Adaptation Implementation

At the time of writing, no brain adaptation mechanisms can be eliminated.

7.1.5. Neuroplasticity Mechanism

At the time of writing, no neuroplasticitiy mechanisms can be eliminated.

7.1.6. Neuromorphic Architecture

1. Some neuromorphic architectures can be eliminated based on logistical reasons. Currently, the only direct access within this thesis project is to the Loihi. Furthermore, it may be expected that access to Pattern Recognition Chip by Innaterra may be realised after the ICONS deadline. Similarly, the Spinnaker device may become available later-on in the project. An economic feasibility assessment needs to be made on whether they should be used in physical radiation testing or not.

2. Some of the neuromorphic chip manufacturers have been contacted in the past, these contacts may allow for access to their respective chips for physical radation testing, if the intermediate results at ICONS are promising. Hence, they are kept orange.

7.1.7. Result Quantification

1. Since physical radiation testing is not deemed possible, and since the hardware diagrams of the respective neuromorphic architectures are not available, it is not deemed feasible to include a radiation effect analysis before ICONS. Therefore, this option is turned orange.

7.1.8. Pruned Design Option Tree

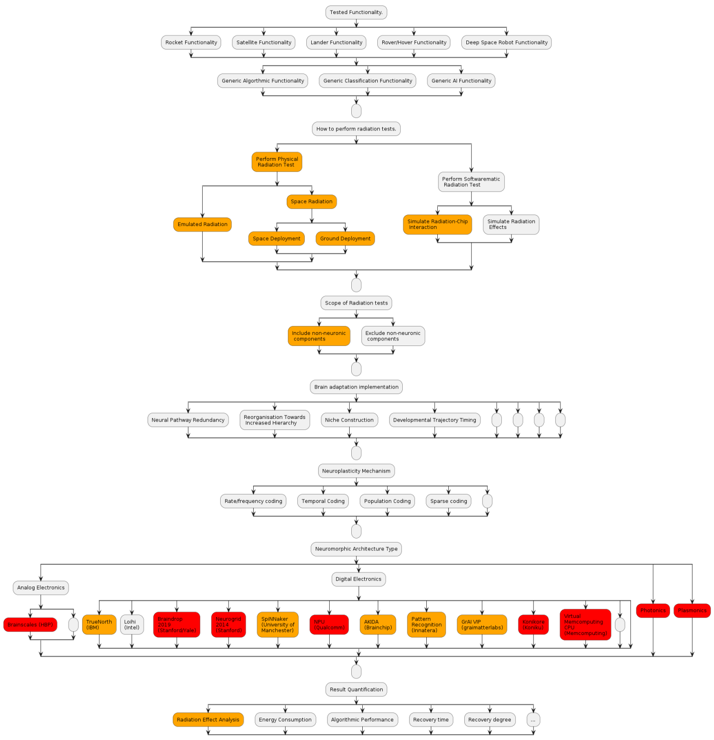


Figure 7.2: A functional flow diagram with the high-level functions of the system that is to be designed.

7.2. Preliminary Design Option Selection

With some design options eliminated, an analysis can be performed to see whether some design options are considered to be more feasible than others. This analysis is started by taking the **STKH-SPACEBRAINS-01** requirement into account, which implies results need to be produced by April 15th. This allows for expressing some design option preferences as listed in section 7.2.1 to section 7.2.7. Taking the full scope of the thesis project into account allows identification of design options that seem most feasible for a follow-up with physical radiation testing.

7.2.1. Tested Functionality Preference

For performing a generic algorithmic functionality for the following two reasons:

1. No additional dependencies such as datasets, panda packages, tensorflow etc. is required. This lowers the probability of allocating time on work that does not directly support the objective of this thesis project.
2. No preprocessing work, such as loading and/or cropping images etc., is required. This increases the amount of time that can be allocated to implementing the brain adaptation and testing its functionality.
3. Graph algorithms are typically used in space applications [**?**].
4. Thorough testing can quickly be set up for graph algorithms.

7.2.2. How To Perform Radiation Tests Preference

Using softwarematic radiation tests that simulate radiation effects.

7.2.3. Scope of Radiation Tests Preference

Excluding non-neural components from radiation effects allows for a complete focus using the expected radiation effects on the neural and synaptic properties, without having to model additional radiation interactions with (other) hardware elements. This increases the feasibility of producing results by April 15th.

7.2.4. Brain Adaptation Implementation Preference

No preference in brain adaptation implementations is expressed at the time of writing.

7.2.5. Neuroplasticity Mechanism Preference

No preference in neuroplasticity mechanisms is expressed at the time of writing.

7.2.6. Neuromorphic Architecture Type Preference

Based on availability and previous experience, a preference is expressed for the Loihi platform for the softwarematic simulation. Insights from this simulation will be used to determine the best way forward towards hardware simulations. Since the Innatera chips are expected to be available for testing in the second quarter of 2022, combined with there relatively low cost, this option is mentioned as a possible suitable candidate for physical radiation testing. The Spinnaker boards appear to have a higher unit cost.

7.2.7. Result Quantification Preference

Based on the vicinity of the ICONS deadline of April 15th, 2022, a preference is expressed for measuring the softwarematic results in terms of algorithmic performance. This is because it forms the most direct measurement that can be used to determine whether the principle of brain adaptation is indeed able to increase the radiation robustness of neuromorphic space hardware.

7.2.8. Preliminary Design Option Tree

The preferences expressed in section 7.2 are coloured green in the preliminary design option tree visualised in fig. 7.3.

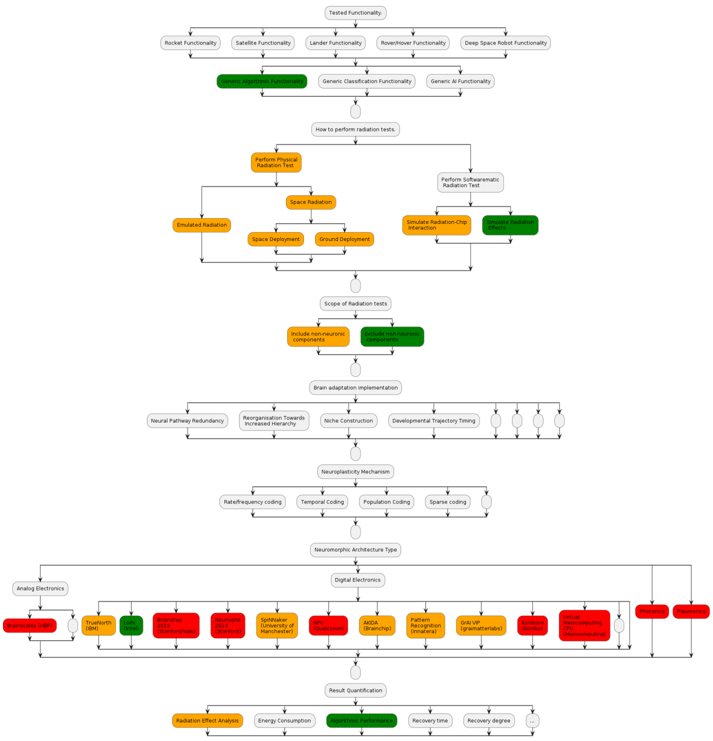


Figure 7.3: todo

7.3. Proposed Design options

To summarise, a distinction between project phases. First a softwarematic test is performed and the results are reported to the ICONS by April 15th 2022. Next, a hardwarematic test is proposed that builds upon the knowledge gained in the softwarematic tests. For the first phase, the following options are proposed to be executed during the midterm: A graph algorithm is proposed to be ran, where the radiation effects on the neuromorphic architecture are simulated as changes in the the neural and synaptic properties of the spiking neural networks. For this first softwarematic test, the radiation effects on non-neural/traditional hardware components (such as a memory bus, CPU etc.) of the neuromorphic architecture are ignored. The Loihi platform will be used for the simulation development and radiation tests. The test results will be interpreted in terms of performance of the graph algorithm. The most feasible brain-adaptation method and neuroplasticitiy mechanism will be determined using hands-on experimentation. For the second tests, a physical radiation test is proposed. For this test, the Innatera chips are currently considered as the most feasible option. Based on the results of the softwarematic testing, a decision can be made to inlcude the traditional hardware components of the respective chips. An alternative option could be to apply extra shielding to those components, whilst saving mass on the non-shielded components though the use of brain adaptation inspired implementations.

Chapter 8

Contingency Management

Chapter 9

Market Analysis

Chapter 10

Sustainable Development Management

Chapter 11

Reporting and Quality Control

The following quality control compliance checklist can currently be generated for this project plan:

- Language Tools Grammar check applied:✔
- Language Tools Spelling check applied:✔
- CI is ran on code base used in generating this Project Plan:✗
  - Python Black Formatting Compliance:✗
  - ShellCheck Compliance:✗
  - Latex Prettier Formatting Compliance:✗
  - Python Unit Test Passing: 2 failed, 7 passed [tests]
  - Python Test Coverage %:✗[%]
  - Shell Unit Test Passing:✗[tests]
  - Shell Unit Test Coverage %:✗[%]
- Manual Quality check:
  1. Citations still have to be compiled correctly.
  2. Reproduction instructions in Appendix A should be updated.
  3. Word wrap should be applied to .uml appendices.

Chapter 12

Conclusion

Bibliography

Acronyms

ALU Arithmatic Logic Unit. 32

ANN Artificial Neural Network. 32

BIOS Basic Input/Output System. 32

BISER Built-in Soft Error Resillience. 32

BIT Binary Digit. 32

BJT Bipolar Junction Transistor. 32

CME Coronal Mass Ejection. 32

CMOS Complementary MetalâĂŞOxideâĂŞSemiconductor. 32

CPU Central Processing Unit. 21, 32

DEC Double-error Correcting Code. 32

DICE Dual Interlocked Storage Cell. 32

DNN Deep Neural Network. 32

DNU Dual-node Upset. 32

DOD Department of Defence. 32

DOT Design Options Structuring Tree. 3, 18, 32

DRAM Dynamic Random Access Memory. 32

ECC Error Correction Code. 32

ELT Enclosed Layout Transistor. 32

ESA European Space Agency. 12, 32

FBD Functional Break-down Diagram. 3, 9, 32

FET Field-Effect Transistor. 32

FFD Functional Flow Diagram. 3, 5, 32

GCD Greatest Common Divisor. 32

GCR Galactic Cosmic Ray. 32

IC Integrated Circuit. 32

INRC Intel Neuromorphic Research Community. 32

LCM Least Common Multiple. 32

LET Linear Energy Transfer. 32

MOS MetalâĂŞOxideâĂŞSemiconductor. 32

MOSFET MetalâĂŞOxideâĂŞSemiconductor Field-Effect Transistor. 32

MPNN Multilayer Perceptron Neural Network. 32

MSN Mission Need Statement. 11, 32

NVM Non-Volatile Memory. 32

POS Project Objective Statement. 32

RAM Random Access Memory. 32

RDT Requirements Discovery Tree. 3, 11, 32

ROM Read Only Memory. 32

SEC Single-error Correcting Code. 32

SEE Single-event Effects. 32

SEGR Single-event Gate Rupture. 32

SEIB Single-event Induced Burnout. 32

SEL Single-event Latch-up. 32

SER Soft-error Rate. 32

SERL Soft-error Resilient Latch. 32

SES Single-event Snapback. 32

SET Single-event Transient. 32

SEU Single-event Upset. 32

SNN Spiking Neural Network. 32

SNU Single-node Upset. 32

SPE Solar Particle Events. 32

SPENVIS Space Environment Information System. 32

SRAM Static Random Access Memory. 32

STDP Spike-Timing-Dependent Plasticity. 32

TLB Translation Lookaside Buffer. 32

TMR Tripple-mode Redundancy. 32

VLSI Very-Large-Scale Integration. 32

WBS Work Break-down Structure. 32

WFD Work Flow Diagram. 32

Glossary

(electron) holes In the context of doped semiconductors, holes are positions where electron acceptors are located, in other words, they are holes at which the electron can go.. 32

CPU cache A small hardware memory unit that is faster than the main memory and closer to the Arithmatic Logic Unit.. 32

data cache A cache that is designed to increase the speed with which data is fetched and stored.. 32

formula A mathematical expression. 32

instruction cache A cache that is designed to increase the speed with which instructions are fetched.. 32

latex Is a mark up language specially suited for scientific documents. 32

mathematics Mathematics is what mathematicians do. 32

memory cell A fundamental/basic unit in computing that is used to store information.. 32

n-type semiconductor A semiconductor that is doped with electron donors. 32

p-n junction A boundary interface of two a p-type semiconductor and a n-type semiconductor. 32

p-type semiconductor A semiconductor that is doped with electron acceptors. 32

primary memory A form of memory that is only accessible to the Central Processing Unit which reads instructions from it and executes those instructions.. 32

prompt charge The charge that is collected by means of funnelling.. 32

random-access A memory type that has access times that are independent of its physical location.. 32

unipolar transistors Unipolar transistors are transistors that use either electrons or electron holes as charge carriers and not the combination of the two.. 32

Nomenclature

c Speed of light in a vacuum inertial frame

h Planck constant

.1. Appendix /src/__main__.py

```python
## Entry point for this project, runs the project code and exports data if
# export commmands are given to the cli command that invokes this script.


## Import used functions.
# Project code imports.
from .create_planar_triangle_free_graph import get_graph
from .neumann import compute_mtds
from .neumann_a_t_0 import compute_mtds_a_t_0
from .arg_parser import parse_cli_args

# Export data import.
from .export_data.export_data import export_data


## Parse command line interface arguments to determine what this script does.
args = parse_cli_args()


## Run main code.
G = get_graph(args, False)
# compute_mtds(G)
compute_mtds_a_t_0(G)


## Run data export code if any argument is given.
if not all(arg is None for arg in [args.l, args.dd, args.sd, args.c2l, args.ec2l]):
    export_data(args)
```

.2. Appendix /src/arg_parser.py

```
# This is the main code of this project nr, and it manages
    ↪ running the code and
# outputting the results to LaTex.
import argparse


def parse_cli_args():
    # Instantiate the parser
    parser = argparse.ArgumentParser(description="Optional app
        ↪ description")

    ## Include argument parsing for default code.
    # Allow user to load a graph from file.
    parser.add_argument(
        "--g",
        dest="graph_from_file",
        action="store_true",
        help="boolean flag, determines whether energy analysis
            ↪ graph is created from file ",
    )

    # Allow user to specify an infile.
    parser.add_argument("infile", nargs="?", type=argparse.
        ↪ FileType("r"))

    # Specify default argument values for the parser.
    parser.set_defaults(
        infile=None,
        graph_from_file=False,
    )

    ## Include argument parsing for data exporting code.
    # Compile LaTex
    parser.add_argument(
        "--l", action="store_true", help="Boolean indicating
            ↪ if code compiles LaTex"
    )

    # Generate, compile and export Dynamic PlantUML diagrams
        ↪ to LaTex.
    parser.add_argument(
        "--dd",
        action="store_true",
        help="A boolean indicating if code generated diagrams
            ↪ are compiled and exported.",
    )
    # Generate, compile and export Static PlantUML diagrams to
        ↪ LaTex.
    parser.add_argument(
        "--sd",
        action="store_true",
        help="A boolean indicating if static diagrams are
            ↪ compiled and exported.",
    )
```

```
# Export the project code to LaTex.
parser.add_argument(
    "--c2l",
    action="store_true",
    help="A boolean indicating if project code is exported
       ↪  to LaTex.",
)

# Export the exporting code, and the project code to LaTex
   ↪ .
parser.add_argument(
    "--ec2l",
    action="store_true",
    help="A boolean indicating if code that exports code
       ↪ is exported to LaTex.",
)

# Load the arguments that are given.
args = parser.parse_args()
return args
```

.3. Appendix /src/create_planar_triangle_free_graph.py

```
# This code creates a planar graph without triangles.
# Planar implies that it fits on a 2D plane, without any edges
    ↪ crossing.
# Triangle free means there are no triangles in the graph.

import networkx as nx

import matplotlib.pyplot as plt


def get_graph(args, show_graph):
    if args.infile and not args.graph_from_file:
        try:
            graph = nx.read_graphml(args.infile.name)
        except Exception as exc:
            raise Exception(
                "Supplied input file is not a gml networkx
                    ↪ graph object."
            ) from exc
    else:
        graph = create_manual_test_graph()
    if show_graph:
        plot_graph(graph)
    return graph


def create_manual_test_graph():
    """
    creates manual test graph with 7 undirected nodes.
    """

    graph = nx.Graph()
    graph.add_nodes_from(
        ["a", "b", "c", "d", "e", "f", "g"],
        x=0,
        color="w",
        dynamic_degree=0,
        delta_two=0,
        p=0,
        xds=0,
    )
    graph.add_edges_from(
        [
            ("a", "b"),
            ("a", "c"),
            # ("b", "c"),
            # ("b", "d"),
            ("c", "d"),
            ("d", "e"),
            ("e", "g"),
            ("b", "e"),
            ("b", "f"),
            ("f", "g"),
        ]
```

```
    )
    return graph


# TODO: move to helper
def plot_graph(G):
    options = {"with_labels": True, "node_color": "white", "
       ↪ edgecolors": "blue"}
    nx.draw_networkx(G, **options)
    plt.show()
    plt.clf()
```

.4. Appendix /src/distributed.py

```
# 1.a Each vertex v_i chooses random float r_i in range 0<r_i
    ↪ <1

# 1.b Each vertex v_i computes d_i. d_i=degree of vertex v_i
# 1.c Each vertex v_i computes weight: w_i=d_i+r_i
# 1.d Each vertex v_i sends w_i to each of its neighbours.

# 2.a Each vertex v_i gets the index of the
# neighbouring vertex v_j_(w_max) that has the heighest w_i,
    ↪ with i!=j.
# 2.b Each vertex v_i adds a mark to that neighbour vertex
    ↪ v_j_(w_max).

# 3 for k in range [0,m] rounds, do:

# 4.a Each node v_i computes how many marks it has received,
    ↪ as (x_i)_k.
# 4.b Each node v_i computes (w_i)_k=(x_i)_k+ r_i

# 5. Reset marked vertices: for each vertex v_i, (x_i)_k=0

# 6.a Each vertex v_i computes d_i. d_i=degree of vertex v_i
# 6.b Each vertex v_i sends w_i to each of its neighbours.
# 6.c Each vertex v_i gets the index of the
# neighbouring vertex v_j_(w_max) that has the heighest w_i,
    ↪ with i!=j.
# 6.d Each vertex v_i adds a mark to that neighbour vertex
    ↪ v_j_(w_max).
```

.5. Appendix /src/neumann.py

```
# This code computes a minimium total dominating set
    ↪ approximation.
import random


def compute_mtds(G, m=1):

    # No directed graphs supported in this algorithm.
    if G.is_directed():
        raise Exception("Parameter graph should be undirected"
            ↪ )

    # 1.a Each vertex v_i chooses random float r_i in range 0<
        ↪ r_i<1
    for node in G.nodes:
        # print(f'label={G.nodes[node]["label"]}')
        G.nodes[node]["rand_val"] = random.uniform(0, 1)
        print(f'rand_val={G.nodes[node]["rand_val"]}')
    # 1.b Each vertex v_i computes d_i. d_i=degree of vertex
        ↪ v_i
    for node in G.nodes:
        G.nodes[node]["degree"] = G.degree(node)
        print(f'degree={G.nodes[node]["degree"]}')
    # 1.c Each vertex v_i computes weight: w_i=d_i+r_i
    for node in G.nodes:
        G.nodes[node]["weight"] = G.nodes[node]["degree"] + G.
            ↪ nodes[node]["rand_val"]
        print(f'weight={G.nodes[node]["weight"]}')

    # 1.d Each vertex v_i sends w_i to each of its neighbours.
    # TODO: paradigm: Either each neuron sends some value to
        ↪ another neuron.
    # OR: each neuron asks/demands some value of each of its
        ↪ neighbours.
    store_index_max_neighbour_weight(G)

    # 2.a Each vertex v_i gets the index of the
    # neighbouring vertex v_j_(w_max) that has the heighest
        ↪ w_i, with i!=j.
    # 2.b Each vertex v_i adds a mark to that neighbour vertex
        ↪  v_j_(w_max).
    for node in G.nodes:
        G.nodes[node]["mark"] = 0
    for node in G.nodes:
        neighbor_with_max_weight = G.nodes[node]["
            ↪ neighbor_with_max_weight"]
        G.nodes[neighbor_with_max_weight]["mark"] = (
            G.nodes[neighbor_with_max_weight]["mark"] + 1
        )

    # 3 for k in range [0,m] rounds, do:
    for _ in range(0, m):
```

```
        # 4.a Each node v_i computes how many marks it has
          ↪ received , as (x_i)_k.
        # 4.b Each node v_i computes (w_i)_k=(x_i)_k+ r_i
        for node in G.nodes:
            G.nodes[node]["weight"] = G.nodes[node]["mark"] +
              ↪ G.nodes[node]["rand_val"]
            print(f'weight={G.nodes[node]["weight"]}')

        # 5. Reset marked vertices: for each vertex v_i, (x_i)
          ↪ _k=0

        # 6.a Each vertex v_i computes d_i. d_i=degree of
          ↪ vertex v_i
        # 6.b Each vertex v_i sends w_i to each of its
          ↪ neighbours.
        # 6.c Each vertex v_i gets the index of the
        # neighbouring vertex v_j_(w_max) that has the
          ↪ heighest w_i, with i!=j.
        # 6.d Each vertex v_i adds a mark to that neighbour
          ↪ vertex v_j_(w_max).


def store_index_max_neighbour_weight(G):
    for node in G.nodes:
        G.nodes[node]["neighbor_with_max_weight"] = None
        max = 0  # Assumes positive weights
        for neighbor in G.neighbors(node):
            if G.nodes[node]["neighbor_with_max_weight"] is
              ↪ None:
                G.nodes[node][
                    "neighbor_with_max_weight"
                ] = neighbor  # G.nodes[neighbor]
                max = G.nodes[neighbor]["weight"]
            elif G.nodes[neighbor]["weight"] > max:
                G.nodes[node][
                    "neighbor_with_max_weight"
                ] = neighbor  # G.nodes[neighbor]
                max = G.nodes[neighbor]["weight"]
        print(
            f'node={node}G.nodes[node]["
              ↪ neighbor_with_max_weight"]={G.nodes[node]["
              ↪ neighbor_with_max_weight"]}'
        )
    return G
```

.6. Appendix /src/neumann_a_t_0.py

```
# This code computes a minimium total dominating set
    ↪ approximation.
import random


def compute_mtds_a_t_0(G, m=2):

    # No directed graphs supported in this algorithm.
    if G.is_directed():
        raise Exception("Parameter graph should be undirected"
            ↪ )

    # 1.a Each vertex v_i chooses random float r_i in range 0<
        ↪ r_i<1
    for node in G.nodes:
        # print(f'label={G.nodes[node]["label"]}')
        G.nodes[node]["rand_val"] = random.uniform(0, 1)
        print(f'rand_val={G.nodes[node]["rand_val"]}')
    # 1.b Each vertex v_i computes d_i. d_i=degree of vertex
        ↪ v_i
    for node in G.nodes:
        G.nodes[node]["degree"] = G.degree(node)
        print(f'degree={G.nodes[node]["degree"]}')
    # 1.c Each vertex v_i computes weight: w_i=d_i+r_i
    for node in G.nodes:
        G.nodes[node]["weight"] = G.nodes[node]["degree"] + G.
            ↪ nodes[node]["rand_val"]
        print(f'weight={G.nodes[node]["weight"]}')

    # 1.d Each vertex v_i sends w_i to each of its neighbours.
    # TODO: paradigm: Either each neuron sends some value to
        ↪ another neuron.
    # OR: each neuron asks/demands some value of each of its
        ↪ neighbours.
    store_index_max_neighbour_weight(G)

    # 2.a Each vertex v_i gets the index of the
    # neighbouring vertex v_j_(w_max) that has the heighest
        ↪ w_i, with i!=j.
    # 2.b Initialise the mark counter x_i
    for node in G.nodes:
        G.nodes[node]["mark"] = 0
    # 2.c Each vertex v_i adds a mark to that neighbour vertex
        ↪  v_j_(w_max).
    for node in G.nodes:
        neighbor_with_max_weight = G.nodes[node]["
            ↪ neighbor_with_max_weight"]
        G.nodes[neighbor_with_max_weight]["mark"] = (
            G.nodes[neighbor_with_max_weight]["mark"] + 1
        )

    # 3 for k in range [0,m] rounds, do:
    for _ in range(0, m):
```

```
    # 4.a Each node v_i computes how many marks it has
      ↪ received , as (x_i)_k.
    # 4.b Each node v_i computes (w_i)_k=(x_i)_k+ r_i
    for node in G.nodes:
        G.nodes[node]["weight"] = G.nodes[node]["mark"] +
           ↪ G.nodes[node]["rand_val"]
        print(f'weight={G.nodes[node]["weight"]}')

    # 5. Reset marked vertices: for each vertex v_i, (x_i)
      ↪ _k=0
    for node in G.nodes:
        G.nodes[node]["mark"] = 0

    # 6.a Each vertex v_i computes d_i. d_i=degree of
      ↪ vertex v_i
    # 6.b Each vertex v_i sends w_i to each of its
      ↪ neighbours.
    # 6.c Each vertex v_i gets the index of the
    # neighbouring vertex v_j_(w_max) that has the
      ↪ heighest w_i, with i!=j.
    store_index_max_neighbour_weight(G)
    # 6.d Each vertex v_i adds a mark to that neighbour
      ↪ vertex v_j_(w_max).
    for node in G.nodes:
        neighbor_with_max_weight = G.nodes[node]["
           ↪ neighbor_with_max_weight"]
        G.nodes[neighbor_with_max_weight]["mark"] = (
            G.nodes[neighbor_with_max_weight]["mark"] + 1
        )

    # Print Minimum Total Dominating Set approximation.
    mtds = []
    for node in G.nodes:
        print(f'node {node} has {G.nodes[node]["mark"]}
           ↪ marks')
        if 0 < G.nodes[node]["mark"]:
            mtds.append(node)
    print(f"mtds={mtds}")


def store_index_max_neighbour_weight(G):
    for node in G.nodes:
        G.nodes[node]["neighbor_with_max_weight"] = None
        max = 0  # Assumes positive weights
        for neighbor in G.neighbors(node):
            if G.nodes[node]["neighbor_with_max_weight"] is
               ↪ None:
                G.nodes[node][
                    "neighbor_with_max_weight"
                ] = neighbor  # G.nodes[neighbor]
                max = G.nodes[neighbor]["weight"]
            elif G.nodes[neighbor]["weight"] > max:
                G.nodes[node][
                    "neighbor_with_max_weight"
                ] = neighbor  # G.nodes[neighbor]
                max = G.nodes[neighbor]["weight"]
```

```
        print(
            f'node={node}G.nodes[node]["
              ↪ neighbor_with_max_weight"]={G.nodes[node]["
              ↪ neighbor_with_max_weight"]}'
        )
    return G
```

.7. Appendix /src/export_data/Hardcoded_data.py

```
# Specify hardcoded output data.
from .latex_export_code import export_code_to_latex
from .latex_compile import compile_latex
from .plantuml_generate import generate_all_dynamic_diagrams
from .plantuml_compile import
   ↪ compile_diagrams_in_dir_relative_to_root
from .plantuml_to_tex import export_diagrams_to_latex


class Hardcoded_data:
    """ """

    def __init__(self):

        # Specify code configuration details
        # TODO: include as optional arguments.
        self.await_compilation = True
        self.verbose = True
        self.gantt_extension = ".uml"
        self.diagram_extension = ".png"

        # Filenames.
        self.main_latex_filename = "report.tex"
        self.export_data_dirname = "export_data"
        self.diagram_dir = "Diagrams"
        self.plant_uml_java_filename = "plantuml.jar"

        # Appendix manager filenames
        self.export_code_appendices_filename = "
           ↪ export_code_appendices.tex"
        self.export_code_appendices_filename_from_root = (
            "export_code_appendices_from_root.tex"
        )
        self.project_code_appendices_filename = "
           ↪ project_code_appendices.tex"
        self.project_code_appendices_filename_from_root = (
            "project_code_appendices_from_root.tex"
        )
        self.automatic_appendices_manager_filenames = [
            self.export_code_appendices_filename,
            self.export_code_appendices_filename_from_root,
            self.project_code_appendices_filename,
            self.project_code_appendices_filename_from_root,
        ]

        self.manual_appendices_filename = "manual_appendices.
           ↪ tex"
        self.manual_appendices_filename_from_root = "
           ↪ manual_appendices_from_root.tex"
        self.manual_appendices_manager_filenames = [
            self.manual_appendices_filename,
            self.manual_appendices_filename_from_root,
        ]
        self.appendix_dir_from_root = "latex/Appendices/"
```

```
# Folder names.
self.dynamic_diagram_dir = "Dynamic_diagrams"
self.static_diagram_dir = "Static_diagrams"

# Specify paths relative to root.
self.path_to_export_data_from_root = f"src/{self.
   ↪ export_data_dirname}"
self.jar_path_relative_from_root = (
    f"{self.path_to_export_data_from_root}/{self.
       ↪ plant_uml_java_filename}"
)
self.diagram_output_dir_relative_to_root = f"latex/
   ↪ Images/{self.diagram_dir}"

# Path related variables
self.append_export_code_to_latex = True
self.path_to_dynamic_gantts = f"{self.
   ↪ path_to_export_data_from_root}/{self.diagram_dir
   ↪ }/{self.dynamic_diagram_dir}"
self.path_to_static_gantts = f"{self.
   ↪ path_to_export_data_from_root}/{self.diagram_dir
   ↪ }/{self.static_diagram_dir}"
```

.8. Appendix /src/export_data/Plot_to_tex.py

```
### Call this from another file, for project 11, question 3b:
### from Plot_to_tex import Plot_to_tex as plt_tex
### multiple_y_series = np.zeros((nrOfDataSeries,
    ↪ nrOfDataPoints), dtype=int); # actually fill with data
### lineLabels = [] # add a label for each dataseries
### plt_tex.plotMultipleLines(plt_tex,single_x_series,
    ↪ multiple_y_series,"x-axis label [units]","y-axis label [
    ↪ units]",lineLabels,"3b",4,11)
### 4b=filename
### 4 = position of legend, e.g. top right.
###
### For a single line, use:
### plt_tex.plotSingleLine(plt_tex,range(0, len(dataseries)),
    ↪ dataseries,"x-axis label [units]","y-axis label [units]",
    ↪ lineLabel,"3b",4,11)

### You can also plot a table directly into latex, see
    ↪ example_create_a_table(..)
###
### Then put it in latex with for example:
###\begin{table}[H]
###     \centering
###     \caption{Results some computation.}\label{tab:
    ↪ some_computation}
###     \begin{tabular}{|c|c|} % remember to update this to
    ↪ show all columns of table
###         \hline
###         \input{latex/project3/tables/q2.txt}
###     \end{tabular}
###\end{table}
import random
from matplotlib import lines
import matplotlib.pyplot as plt
import numpy as np
import os


class Plot_to_tex:
    """ """

    def __init__(self):
        self.script_dir = self.get_script_dir()
        print("Created main")

    # plot graph (legendPosition = integer 1 to 4)
    def plotSingleLine(
        self,
        x_path,
        y_series,
        x_axis_label,
        y_axis_label,
        label,
        filename,
        legendPosition,
```

```
        project_name,
    ):
        """

        :param x_path: param y_series:
        :param x_axis_label: param y_axis_label:
        :param label: param filename:
        :param legendPosition: param project_name:
        :param y_series: param y_axis_label:
        :param filename: param project_name:
        :param y_axis_label: param project_name:
        :param project_name:

        """
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(x_path, y_series, c="b", ls="-", label=label,
           ↪ fillstyle="none")
        plt.legend(loc=legendPosition)
        plt.xlabel(x_axis_label)
        plt.ylabel(y_axis_label)
        plt.savefig(
            os.path.dirname(__file__)
            + f"/../../../latex/{project_name}"
            + "/Images/"
            + filename
            + ".png"
        )

    #           plt.show();

    # plot graphs
    def plotMultipleLines(
        self,
        x,
        y_series,
        x_label,
        y_label,
        label,
        filename,
        legendPosition,
        project_name,
    ):
        """

        :param x: param y_series:
        :param x_label: param y_label:
        :param label: param filename:
        :param legendPosition: param project_name:
        :param y_series: param y_label:
        :param filename: param project_name:
        :param y_label: param project_name:
        :param project_name:

        """
        fig = plt.figure()
```

```
        ax = fig.add_subplot(111)

        # generate colours
        cmap = self.get_cmap(len(y_series[:, 0]))

        # generate line types
        lineTypes = self.generateLineTypes(y_series)

        for i in range(0, len(y_series)):
            # overwrite linetypes to single type
            lineTypes[i] = "-"
            ax.plot(
                x,
                y_series[i, :],
                ls=lineTypes[i],
                label=label[i],
                fillstyle="none",
                c=cmap(i),
            )
            # color

        # configure plot layout
        plt.legend(loc=legendPosition)
        plt.xlabel(x_label)
        plt.ylabel(y_label)
        plt.savefig(
            os.path.dirname(__file__)
            + f"/../../../latex/{project_name}"
            + "/Images/"
            + filename
            + ".png"
        )

        print(f"plotted lines")

    # Generate random line colours
    # Source: https://stackoverflow.com/questions/14720331/how
    ↪ -to-generate-random-colors-in-matplotlib
    def get_cmap(n, name="hsv"):
        """Returns a function that maps each index in 0, 1,
        ↪ ..., n-1 to a distinct
        RGB color; the keyword argument name must be a
        ↪ standard mpl colormap name.

        :param n: param name:  (Default value = "hsv")
        :param name: Default value = "hsv")

        """
        return plt.cm.get_cmap(name, n)

    def generateLineTypes(y_series):
        """

        :param y_series:

        """
```

```python
        # generate varying linetypes
        typeOfLines = list(lines.lineStyles.keys())

        while len(y_series) > len(typeOfLines):
            typeOfLines.append("-.")

        # remove void lines
        for i in range(0, len(y_series)):
            if typeOfLines[i] == "None":
                typeOfLines[i] = "-"
            if typeOfLines[i] == "":
                typeOfLines[i] = ":"
            if typeOfLines[i] == " ":
                typeOfLines[i] = "--"
        return typeOfLines

    # Create a table with: table_matrix = np.zeros((4,4),dtype
    ↪ =object) and pass it to this object
    def put_table_in_tex(self, table_matrix, filename,
    ↪ project_name):
        """

        :param table_matrix: param filename:
        :param project_name: param filename:
        :param filename:

        """
        cols = np.shape(table_matrix)[1]
        format = "%s"
        for col in range(1, cols):
            format = format + " & %s"
        format = format + ""
        plt.savetxt(
            os.path.dirname(__file__)
            + f"/../../../latex/{project_name}"
            + "/tables/"
            + filename
            + ".txt",
            table_matrix,
            delimiter=" & ",
            fmt=format,
            newline="  \\\\ \hline \n",
        )

    # replace this with your own table creation and then pass
    ↪ it to put_table_in_tex(..)
    def example_create_a_table(self):
        """ """
        project_name = "1"
        table_name = "example_table_name"
        rows = 2
        columns = 4
        table_matrix = np.zeros((rows, columns), dtype=object)
        table_matrix[:, :] = ""  # replace the standard zeros
            ↪ with emtpy cell
        print(table_matrix)
```

```
        for column in range(0, columns):
            for row in range(0, rows):
                table_matrix[row, column] = row + column
        table_matrix[1, 0] = "example"
        table_matrix[0, 1] = "grid sizes"

        self.put_table_in_tex(table_matrix, table_name,
           ↪ project_name)

    def get_script_dir(self):
        """returns the directory of this script regardles of
           ↪ from which level the code is executed"""
        return os.path.dirname(__file__)


if __name__ == "__main__":
    main = Plot_to_tex()
    main.example_create_a_table()
```

.9. Appendix /src/export_data/create_dynamic_diagrams.py

```
# Data export imports.
from .plantuml_generate import generate_all_dynamic_diagrams
from .plantuml_compile import
    ↪ compile_diagrams_in_dir_relative_to_root
from .plantuml_to_tex import export_diagrams_to_latex


def create_dynamic_diagrams(args, hd):
    ## PlantUML
    # Generate PlantUML diagrams dynamically (using code).
    if args.dd:
        generate_all_dynamic_diagrams(
            f"{hd.path_to_export_data_from_root}/Diagrams/
                ↪ Dynamic"
        )

        # Compile dynamically generated PlantUML diagrams to
            ↪ images.
        compile_diagrams_in_dir_relative_to_root(
            hd.await_compilation,
            hd.gantt_extension,
            hd.jar_path_relative_from_root,
            hd.path_to_dynamic_gantts,
            hd.verbose,
        )

        # Export dynamic PlantUML text files to LaTex.
        export_diagrams_to_latex(
            hd.path_to_dynamic_gantts,
            hd.gantt_extension,
            hd.diagram_output_dir_relative_to_root,
        )

        # Export dynamic PlantUML diagram images to LaTex.
        export_diagrams_to_latex(
            hd.path_to_dynamic_gantts,
            hd.diagram_extension,
            hd.diagram_output_dir_relative_to_root,
        )
```

.10. Appendix /src/export_data/create_static_diagrams.py

```
# Data export imports.
from .plantuml_generate import generate_all_dynamic_diagrams
from .plantuml_compile import
    ↪ compile_diagrams_in_dir_relative_to_root
from .plantuml_to_tex import export_diagrams_to_latex


def create_static_diagrams(args, hd):

    ## PlantUML
    if args.sd:
        # Compile statically generated PlantUML diagrams to
            ↪ images.
        compile_diagrams_in_dir_relative_to_root(
            hd.await_compilation,
            hd.gantt_extension,
            hd.jar_path_relative_from_root,
            hd.path_to_static_gantts,
            hd.verbose,
        )

        # Export static PlantUML text files to LaTex.
        export_diagrams_to_latex(
            hd.path_to_static_gantts,
            hd.gantt_extension,
            hd.diagram_output_dir_relative_to_root,
        )

        # Export static PlantUML diagram images to LaTex.
        export_diagrams_to_latex(
            hd.path_to_static_gantts,
            hd.diagram_extension,
            hd.diagram_output_dir_relative_to_root,
        )
```

.11. Appendix /src/export_data/export_data.py

```
# Data export imports.
from .Hardcoded_data import Hardcoded_data

from .create_dynamic_diagrams import create_dynamic_diagrams
from .create_static_diagrams import create_static_diagrams
from .latex_export_code import export_code_to_latex
from .latex_compile import compile_latex


def export_data(args):

    hd = Hardcoded_data()

    ## PlantUML
    create_dynamic_diagrams(args, hd)
    create_static_diagrams(args, hd)

    ## Plotting
    # Generate plots.
    # Export plots to LaTex.

    ## Export code to LaTex.
    if args.c2l:
        # TODO: verify whether the latex/{project_name}/
          ↪ Appendices folder exists before exporting.
        # TODO: verify whether the latex/{project_name}/Images
          ↪  folder exists before exporting.
        export_code_to_latex(hd, False)
    elif args.ec2l:
        # TODO: verify whether the latex/{project_name}/
          ↪ Appendices folder exists before exporting.
        # TODO: verify whether the latex/{project_name}/Images
          ↪  folder exists before exporting.
        export_code_to_latex(hd, True)

    ## Compile the accompanying LaTex report.
    if args.l:
        compile_latex(True, True)
        print("")
    print(f"\n\nDone exporting data.")
```

.12. Appendix /src/export_data/helper_bash_commands.py

```
import subprocess


def run_bash_command(await_compilation, bash_command, verbose)
    ↪ :
    if await_compilation:
        if verbose:
            subprocess.call(bash_command, shell=True)
        else:
            subprocess.call(
                bash_command,
                shell=True,
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            )
    else:
        if verbose:
            subprocess.Popen(bash_command, shell=True)
        else:
            subprocess.Popen(
                bash_command,
                shell=True,
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            )
```

.13. Appendix /src/export_data/helper_dir_file_edit.py

```
import os
import shutil
import glob


def file_contains(filepath, substring):
    with open(filepath) as f:
        if substring in f.read():
            return True


def get_dir_filelist_based_on_extension(dir_relative_to_root,
    ↪ extension):
    """

    :param dir_relative_to_root: A relative directory as seen
        ↪ from the root dir of this project.
    :param extension: The file extension that is used/searched
        ↪ in this function.

    """
    selected_filenames = []
    # TODO: assert directory exists
    for filename in os.listdir(dir_relative_to_root):
        if filename.endswith(extension):
            selected_filenames.append(filename)
    return selected_filenames


def create_dir_relative_to_root_if_not_exists(
    ↪ dir_relative_to_root):
    """

    :param dir_relative_to_root: A relative directory as seen
        ↪ from the root dir of this project.

    """
    if not os.path.exists(dir_relative_to_root):
        os.makedirs(dir_relative_to_root)


def dir_relative_to_root_exists(dir_relative_to_root):
    """

    :param dir_relative_to_root: A relative directory as seen
        ↪ from the root dir of this project.

    """
    if not os.path.exists(dir_relative_to_root):
        return False
    elif os.path.exists(dir_relative_to_root):
        return True
    else:
        raise Exception(
```

```
            "Directory relative to root: {dir_relative_to_root
               ↪ } did not exist, nor did it exist."
        )


def get_all_files_in_dir_and_child_dirs(extension, path,
   ↪ excluded_files=None):
    """Returns a list of the relative paths to all files
       ↪ within the some path that match
    the given file extension. Also includes files in child
       ↪ directories.

    :param extension: The file extension that is used/searched
       ↪  in this function. The file extension of the file
       ↪ that is sought in the appendix line. Either ".py" or
       ↪ ".pdf".
    :param path: Absolute filepath in which files are being
       ↪ sought.
    :param excluded_files: Default value = None) Files that
       ↪ will not be included even if they are found.

    """
    filepaths = []
    for r, d, f in os.walk(path):
        for file in f:
            if file.endswith(extension):
                if (excluded_files is None) or (
                    (not excluded_files is None) and (not file
                       ↪  in excluded_files)
                ):
                    filepaths.append(r + "/" + file)
    return filepaths


def get_filepaths_in_dir(extension, path, excluded_files=None)
   ↪ :
    """Returns a list of the relative paths to all files
       ↪ within the some path that match
    the given file extension. Does not include files in
       ↪ child_directories.

    :param extension: The file extension that is used/searched
       ↪  in this function. The file extension of the file
       ↪ that is sought in the appendix line. Either ".py" or
       ↪ ".pdf".
    :param path: Absolute filepath in which files are being
       ↪ sought.
    :param excluded_files: Default value = None) Files that
       ↪ will not be included even if they are found.

    """
    filepaths = []
    current_path = os.getcwd()
    os.chdir(path)
    for file in glob.glob(f"*.{extension}"):
        print(file)
```

```
        if (excluded_files is None) or (
            (not excluded_files is None) and (not file in
               ↪ excluded_files)
        ):
            # Append normalised filepath e.g. collapses b/src
               ↪ /../d to b/d.
            filepaths.append(os.path.normpath(f"{path}/{file}"
               ↪ ))
    os.chdir(current_path)
    return filepaths


def sort_filepaths_by_filename(filepaths):
    # filepaths.sort(key = lambda x: x.split()[1])
    filepaths.sort(key=lambda x: x[x.rfind("/") + 1 :])
    for filepath in filepaths:
        print(f"{filepath}")
    return filepaths


def get_filename_from_dir(path):
    """Returns a filename from an absolute path to a file.

    :param path: path to a file of which the name is queried.

    """
    return path[path.rfind("/") + 1 :]


def delete_file_if_exists(filepath):
    try:
        os.remove(filepath)
    except OSError:
        pass


def convert_filepath_to_filepath_from_root(filepath,
   ↪ normalised_root_path):
    normalised_filepath = os.path.normpath(filepath)
    filepath_relative_from_root = normalised_filepath[len(
       ↪ normalised_root_path) :]
    return filepath_relative_from_root


def append_lines_to_file(filepath, lines):
    with open(filepath, "a") as the_file:
        for line in lines:
            the_file.write(f"{line}\n")


def append_line_to_file(filepath, line):
    with open(filepath, "a") as the_file:
        the_file.write(f"{line}\n")
        the_file.close()
```

```
def remove_all_auto_generated_appendices(hd):

    # TODO: move identifier into hardcoded.
    all_appendix_files = get_all_files_in_dir_and_child_dirs(
        ".tex", hd.appendix_dir_from_root, excluded_files=None
    )
    for file in all_appendix_files:
        if "Auto_generated" in file:
            delete_file_if_exists(file)
```

.14. Appendix /src/export_data/helper_tex_editing.py

```
import os
from .helper_dir_file_edit import (
    append_line_to_file,
    append_lines_to_file,
    convert_filepath_to_filepath_from_root,
    delete_file_if_exists,
    get_filename_from_dir,
)


def code_filepath_to_tex_appendix_filename(
    filename, from_root, is_project_code, is_export_code
):

    # TODO: Include assert to verify filename ends at .py.
    # TODO: Include assert to verify filename doesn't end at .
      ↪ py anymore.
    filename_without_extension = os.path.splitext(filename)[0]

    # Create appendix filename identifier segment
    verify_input_code_type(is_export_code, is_project_code)
    if is_project_code:
        identifier = "Auto_generated_project_code_appendix_"
    elif is_export_code:
        identifier = "Auto_generated_export_code_appendix_"

    appendix_filename = f"{identifier}{
      ↪ filename_without_extension}"
    return appendix_filename


def verify_input_code_type(is_export_code, is_project_code):
    # Create appendix filename identifier segment
    if is_project_code and is_export_code:
        raise Exception(
            "Error, a file can't be both project code, and
              ↪ export code at same time."
        )
    if not is_project_code and not is_export_code:
        raise Exception(
            "Error, don't know what to do with files that are
              ↪ neither project code, nor export code."
        )


def tex_appendix_filename_to_inclusion_command(
  ↪ appendix_filename, from_root):
    # Create full appendix filename.
    if from_root:
        # Generate latex inclusion command for latex
          ↪ compilation from root dir.
        appendix_inclusion_command = (
            f"\input{{latex/Appendices/{appendix_filename}.tex
              ↪ }} \\newpage"
```

```
        )
        # \input{latex/Appendices/Auto_generated_py_App8.tex}
            ↪ \newpage
    else:
        # \input{Appendices/Auto_generated_py_App8.tex} \
            ↪ newpage
        appendix_inclusion_command = (
            f"\input{{Appendices/{appendix_filename}.tex}} \\
                ↪ newpage"
        )
    return appendix_inclusion_command


def create_appendix_filecontent(
    latex_object_name, filename, filepath_from_root, from_root
):
    # Latex titles should escape underscores.
    filepath_from_root_without_underscores =
        ↪ filepath_from_root.replace("_", "\_")
    lines = []
    lines.append(
        f"\{latex_object_name}{{Appendix {
            ↪ filepath_from_root_without_underscores}}}\label{{
            ↪ app:{filename}}}"
    )
    if from_root:
        lines.append(f"\pythonexternal{{latex/..{
            ↪ filepath_from_root}}}")
    else:
        lines.append(f"\pythonexternal{{latex/..{
            ↪ filepath_from_root}}}")
    return lines


def create_appendix_manager_files(hd):
    # Verify target directory exists.
    if not os.path.exists(hd.appendix_dir_from_root):
        raise Exception(
            f"Error, the Appendices directory was not found at
                ↪ :{hd.appendix_dir_from_root}"
        )

    # Delete appendix manager files.
    list(
        map(
            lambda x: delete_file_if_exists(f"{hd.
                ↪ appendix_dir_from_root}{x}"),
            hd.automatic_appendices_manager_filenames,
        )
    )

    # Create new appendix_manager_files
    list(
        map(
            lambda x: open(f"{hd.appendix_dir_from_root}{x}",
                ↪ "a"),
```

```
            hd.automatic_appendices_manager_filenames,
        )
    )

    # Ensure manual appendix_manager_files are created.
    list(
        map(
            lambda x: open(f"{hd.appendix_dir_from_root}{x}",
                ↪ "a"),
            hd.manual_appendices_manager_filenames,
        )
    )


def create_appendix_file(
    hd,
    filename,
    filepath_from_root,
    latex_object_name,
    is_export_code,
    is_project_code,
):
    verify_input_code_type(is_export_code, is_project_code)
    filename_without_extension = os.path.splitext(filename)[0]
    if is_project_code:
        # Create the appendix for the case the latex is
          ↪ compiled from root.
        appendix_filepath = f"{hd.appendix_dir_from_root}/
          ↪ Auto_generated_project_code_appendix_{
          ↪ filename_without_extension}.tex"

        # Append latex_filepath to appendix manager.
        # append_lines_to_file(
        #     f"{hd.appendix_dir_from_root}{hd.
          ↪ project_code_appendices_filename}",
        #     [tex_appendix_filepath_to_inclusion_command(
          ↪ appendix_filepath)],
        # )

        # Get Appendix .tex content.
        appendix_lines_from_root = create_appendix_filecontent
          ↪ (
            latex_object_name, filename, filepath_from_root,
              ↪ True
        )

        # Write appendix to .tex file.
        append_lines_to_file(appendix_filepath,
          ↪ appendix_lines_from_root)
    elif is_export_code:
        # Create the appendix for the case the latex is
          ↪ compiled from root.
        appendix_filepath = f"{hd.appendix_dir_from_root}/
          ↪ Auto_generated_export_code_appendix_{
          ↪ filename_without_extension}.tex"
```

```
        # Append latex_filepath to appendix manager.
        # append_lines_to_file(
        #     f"{hd.appendix_dir_from_root}{hd.
            ↪ export_code_appendices_filename}",
        #     [tex_appendix_filepath_to_inclusion_command(
            ↪ appendix_filepath)],
        # )

        # Get Appendix .tex content.
        appendix_lines_from_root = create_appendix_filecontent
            ↪ (
            latex_object_name, filename, filepath_from_root,
                ↪ True
        )

        # Write appendix to .tex file.
        append_lines_to_file(appendix_filepath,
            ↪ appendix_lines_from_root)
    # TODO: verify files exist


def export_python_project_code(hd, normalised_root_dir,
    ↪ python_project_code_filepaths):
    is_project_code = True
    is_export_code = False
    from_root = False
    for filepath in python_project_code_filepaths:
        create_appendices(
            hd,
            filepath,
            normalised_root_dir,
            from_root,
            is_export_code,
            is_project_code,
        )
        create_appendices(
            hd, filepath, normalised_root_dir, True,
                ↪ is_export_code, is_project_code
        )


def export_python_export_code(hd, normalised_root_dir,
    ↪ python_export_code_filepaths):
    is_project_code = False
    is_export_code = True
    from_root = False
    for filepath in python_export_code_filepaths:
        create_appendices(
            hd,
            filepath,
            normalised_root_dir,
            from_root,
            is_export_code,
            is_project_code,
        )
        create_appendices(
```

```
            hd, filepath, normalised_root_dir, True,
                ↪ is_export_code, is_project_code
        )


def create_appendices(
    hd, filepath, normalised_root_dir, from_root,
        ↪ is_export_code, is_project_code
):
    # Get the filepath of a python file from the root dir of
        ↪ this project.
    filepath_from_root =
        ↪ convert_filepath_to_filepath_from_root(
        filepath, normalised_root_dir
    )
    print(f"from_root={from_root},filepath_from_root={
        ↪ filepath_from_root}")

    # Get the filename of a python filepath
    filename = get_filename_from_dir(filepath)

    # Get the filename for a latex appendix from a python
        ↪ filename.
    appendix_filename = code_filepath_to_tex_appendix_filename
        ↪ (
        filename, from_root, is_project_code, is_export_code
    )

    # Command to include the appendix in the appendices
        ↪ manager.
    appendix_inclusion_command =
        ↪ tex_appendix_filename_to_inclusion_command(
        appendix_filename, from_root
    )
    # if from_root:
    #     print(f'tex_appendix_filename_to_inclusion_command={
        ↪ appendix_inclusion_command}')
    #     exit()

    append_appendix_to_appendix_managers(
        appendix_inclusion_command, from_root, hd,
            ↪ is_export_code, is_project_code
    )

    # Create the appendix .tex file.
    # TODO: move "section" to hardcoded.
    if from_root:  # Appendix only contains files readable
        ↪ from root.
        create_appendix_file(
            hd,
            filename,
            filepath_from_root,
            "section",
            is_export_code,
            is_project_code,
        )
```

```
def append_appendix_to_appendix_managers(
    appendix_inclusion_command, from_root, hd, is_export_code,
      ↪  is_project_code
):
    # Append the appendix .tex file to the appendix manager.
    if is_project_code:
        if from_root:
            # print(f'from_root={from_root}Append to:{hd.
              ↪  project_code_appendices_filename_from_root}')
            append_line_to_file(
                f"{hd.appendix_dir_from_root}{hd.
                  ↪ project_code_appendices_filename_from_root
                  ↪ }",
                appendix_inclusion_command,
            )
        else:
            # print(f'from_root={from_root}Append to:{hd.
              ↪  project_code_appendices_filename}')
            append_line_to_file(
                f"{hd.appendix_dir_from_root}{hd.
                  ↪ project_code_appendices_filename}",
                appendix_inclusion_command,
            )

    if is_export_code:
        if from_root:
            append_line_to_file(
                f"{hd.appendix_dir_from_root}{hd.
                  ↪ export_code_appendices_filename_from_root
                  ↪ }",
                appendix_inclusion_command,
            )
        else:
            append_line_to_file(
                f"{hd.appendix_dir_from_root}{hd.
                  ↪ export_code_appendices_filename}",
                appendix_inclusion_command,
            )
```

.15. Appendix /src/export_data/helper_tex_reading.py

```
from .helper_dir_file_edit import (
    file_contains,
)


def verify_latex_supports_auto_generated_appendices(
    ↪ path_to_main_latex_file):
    # TODO: change verification to complete tex block(s) for
        ↪ appendices.
    # TODO: Also verify related boolean and if statement
        ↪ creations.
    determining_overleaf_home_line = "\def\overleafhome{/tmp}%
        ↪  change as appropriate"
    begin_apendices_line = "\\begin{appendices}"
    print(f"determining_overleaf_home_line={
        ↪ determining_overleaf_home_line}")
    print(f"begin_apendices_line={begin_apendices_line}")

    if not file_contains(path_to_main_latex_file,
        ↪ determining_overleaf_home_line):
        raise Exception(
            f"Error, {path_to_main_latex_file} does not
                ↪ contain:\n\n{determining_overleaf_home_line}\
                ↪ n\n so this Python code cannot export the
                ↪ code as latex appendices."
        )
    if not file_contains(path_to_main_latex_file,
        ↪ determining_overleaf_home_line):
        raise Exception(
            f"Error, {path_to_main_latex_file} does not
                ↪ contain:\n\n{begin_apendices_line}\n\n so
                ↪ this Python code cannot export the code as
                ↪ latex appendices."
        )
```

.16. Appendix /src/export_data/latex_compile.py

```
# Compiles the latex report using the compile script.
import subprocess

from .helper_bash_commands import run_bash_command


def compile_latex(await_compilation, verbose):
    """
    Compiles the LaTex report of this project using its
      ↪ compile script.

    :param await_compilation: Make python wait untill the
      ↪ PlantUML compilation is completed.
    :param project_name: The name of the project that is being
      ↪  executed/ran.
    :param verbose: True, ensures compilation output is
      ↪ printed to terminal, False means compilation is
      ↪ silent.

    Returns:
        Nothing.

    Raises:
        Nothing.
    """

    # Ensure compile script is runnable.
    bash_make_compile_script_runnable_command = f"chmod +x
      ↪ latex/compile_script.sh"
    run_bash_command(
        await_compilation,
          ↪ bash_make_compile_script_runnable_command,
          ↪ verbose
    )

    # Run latex compilation script to compile latex project.
    bash_compilation_command = f"latex/compile_script.sh"
    run_bash_command(await_compilation,
      ↪ bash_compilation_command, verbose)
```

.17. Appendix /src/export_data/latex_export_code.py

```
# runs a jupyter notebook and converts it to pdf
import os

from .helper_tex_editing import (
    create_appendix_manager_files,
    export_python_export_code,
    export_python_project_code,
)

from .helper_tex_reading import (
    verify_latex_supports_auto_generated_appendices,
)

from .helper_dir_file_edit import (
    get_all_files_in_dir_and_child_dirs,
    get_filepaths_in_dir,
    remove_all_auto_generated_appendices,
    sort_filepaths_by_filename,
)


def export_code_to_latex(hd, include_export_code):
    """This function exports the python files and compiled
      ↪ pdfs of jupiter notebooks into the
    latex of the same project number. First it scans which
      ↪ appendices (without code, without
    notebooks) are already manually included in the main latex
      ↪ code. Next, all appendices
    that contain the python code are eiter found or created in
      ↪ the following order:
    First, the __main__.py file is included, followed by the
      ↪ main.py file, followed by all
    python code files in alphabetic order. After this, all the
      ↪ pdfs of the compiled notebooks
    are added in alphabetic order of filename. This order of
      ↪ appendices is overwritten in the
    main tex file.

    :param main_latex_filename: Name of the main latex
      ↪ document of this project number
    :param project_name: The name of the project that is being
      ↪ executed/ran. The number indicating which project
      ↪ this code pertains to.

    """
    script_dir = get_script_dir()
    latex_dir = script_dir + "/../../latex/"
    path_to_main_latex_file = f"{latex_dir}{hd.
      ↪ main_latex_filename}"
    root_dir = script_dir + "/../../"
    normalised_root_dir = os.path.normpath(root_dir)
    src_dir = script_dir + "/../"
```

```
    # Verify the latex file supports auto-generated python
        ↪ appendices.
    verify_latex_supports_auto_generated_appendices(
        ↪ path_to_main_latex_file)

    # Get paths to files containing project python code.
    python_project_code_filepaths = get_filepaths_in_dir("py",
        ↪  src_dir, ["__init__.py"])

    compiled_notebook_pdf_filepaths =
        ↪ get_compiled_notebook_paths(script_dir)
    print(f"python_project_code_filepaths={
        ↪ python_project_code_filepaths}")

    # Get paths to the files containing the latex export code
    if include_export_code:
        python_export_code_filepaths = get_filepaths_in_dir(
            "py", script_dir, ["__init__.py"]
        )

    remove_all_auto_generated_appendices(hd)

    # Create appendix file # ensure they are also deleted at
        ↪ the start of every run.
    create_appendix_manager_files(hd)

    # TODO: Sort main files.
    export_python_project_code(
        hd,
        normalised_root_dir,
        sort_filepaths_by_filename(
            ↪ python_project_code_filepaths),
    )
    if include_export_code:
        export_python_export_code(
            hd,
            normalised_root_dir,
            sort_filepaths_by_filename(
                ↪ python_export_code_filepaths),
        )


def get_compiled_notebook_paths(script_dir):
    """Returns the list of jupiter notebook filepaths that
        ↪ were compiled successfully and that are
    included in the same dias this script (the src directory).

    :param script_dir: absolute path of this file.

    """
    notebook_filepaths = get_all_files_in_dir_and_child_dirs("
        ↪ .ipynb", script_dir)
    compiled_notebook_filepaths = []

    # check if the jupyter notebooks were compiled
    for notebook_filepath in notebook_filepaths:
```

```
        # swap file extension
        notebook_filepath = notebook_filepath.replace(".ipynb"
            ↪ , ".pdf")

        # check if file exists
        if os.path.isfile(notebook_filepath):
            compiled_notebook_filepaths.append(
                ↪ notebook_filepath)
    return compiled_notebook_filepaths


def get_script_dir():
    """returns the directory of this script regardles of from
        ↪ which level the code is executed"""
    return os.path.dirname(__file__)
```

.18. Appendix /src/export_data/plantuml_compile.py

```
# This script automatically compiles the text files
    ↪ representing a PlantUML
# diagram into an actual figure.

# To compile locally manually:
# pip install plantuml
# export  PLANTUML_LIMIT_SIZE=8192
# java -jar plantuml.jar -verbose sequenceDiagram.txt

import os
import subprocess
from os.path import abspath

from .helper_dir_file_edit import
    ↪ get_dir_filelist_based_on_extension
from .plantuml_get_package import got_java_file


def compile_diagrams_in_dir_relative_to_root(
    await_compilation,
    extension,
    jar_path_relative_from_root,
    input_dir_relative_to_root,
    verbose,
):
    """
    Loops through the files in a directory and exports them to
        ↪  the latex /Images
    directory.

    Args:
    :param await_compilation: Make python wait untill the
        ↪ PlantUML compilation is completed. param extension:
        ↪ The filetype of the text file that is converted to
        ↪ image.
    :param jar_path_relative_from_root: The path as seen from
        ↪ root towards the PlantUML .jar file that compiles .
        ↪ uml files to .png files.
    :param verbose: True, ensures compilation output is
        ↪ printed to terminal, False means compilation is
        ↪ silent.
    :param extension: The file extension that is used/searched
        ↪  in this function.
    :param input_dir_relative_to_root: The directory as seen
        ↪ from root containing files that are modified in this
        ↪ function.

    Returns:
        Nothing

    Raises:
        Nothing
    """
    # Verify the PlantUML .jar file is gotten.
```

```
    got_java_file(jar_path_relative_from_root)

    diagram_text_filenames =
       ↪ get_dir_filelist_based_on_extension(
         input_dir_relative_to_root, extension
    )

    for diagram_text_filename in diagram_text_filenames:
        diagram_text_filepath_relative_from_root = (
            f"{input_dir_relative_to_root}/{
               ↪ diagram_text_filename}"
        )

        execute_diagram_compilation_command(
            await_compilation,
            jar_path_relative_from_root,
            diagram_text_filepath_relative_from_root,
            verbose,
        )


def execute_diagram_compilation_command(
    await_compilation,
    jar_path_relative_from_root,
    relative_filepath_from_root,
    verbose,
):
    """
    Compiles a .uml/text file containing a PlantUML diagram to
       ↪ a .png image
    using the PlantUML .jar file.

    Args:
    :param await_compilation: Make python wait untill the
       ↪ PlantUML compilation is completed. param
       ↪ jar_path_relative_from_root:
    :param relative_filepath_from_root: Relative filepath as
       ↪ seen from root of file that is used in this function.
    :param jar_path_relative_from_root: The path as seen from
       ↪ root towards the PlantUML .jar file that compiles .
       ↪ uml files to .png files.
    :param verbose: True, ensures compilation output is
       ↪ printed to terminal, False means compilation is
       ↪ silent.

    Returns:
        Nothing

    Raises:
        Nothing
    """
    # Verify the files required for compilation exist, and
       ↪ convert the paths
    # into absolute filepaths.
    abs_diagram_filepath, abs_jar_path =
       ↪ assert_diagram_compilation_requirements(
```

```
        jar_path_relative_from_root,
            ↪ relative_filepath_from_root
    )

    # Generate command to compile the PlantUML diagram locally
        ↪ .
    print(
        f"abs_jar_path={abs_jar_path}, abs_diagram_filepath={
            ↪ abs_diagram_filepath}\n\n"
    )
    bash_diagram_compilation_command = (
        f"java -jar {abs_jar_path} -verbose {
            ↪ abs_diagram_filepath}"
    )
    print(f"bash_diagram_compilation_command={
        ↪ bash_diagram_compilation_command}")
    # Generate global variable specifying max image width in
        ↪ pixels, in the
    # shell that compiles.
    os.environ["PLANTUML_LIMIT_SIZE"] = "16192"

    # Perform PlantUML compilation locally.
    if await_compilation:
        if verbose:
            subprocess.call(bash_diagram_compilation_command,
                ↪ shell=True)
        else:
            subprocess.call(
                bash_diagram_compilation_command,
                shell=True,
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            )
    else:
        if verbose:
            subprocess.Popen(bash_diagram_compilation_command,
                ↪  shell=True)
        else:
            subprocess.Popen(
                bash_diagram_compilation_command,
                shell=True,
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            )


def assert_diagram_compilation_requirements(
    jar_path_relative_from_root,
    relative_filepath_from_root,
):
    """
    Asserts that the PlantUML .jar file used for compilation
        ↪ exists, and that
    the diagram file with the .uml content for the diagram
        ↪ exists. Throws an
    error if either of two is missing.
```

```
    :param relative_filepath_from_root: Relative filepath as
        ↪ seen from root of file that is used in this function.
    :param output_dir_from_root: Relative directory as seen
        ↪ from root, to which files are outputted.
    :param jar_path_relative_from_root: The path as seen from
        ↪ root towards the PlantUML .jar file that compiles .
        ↪ uml files to .png files.

    Returns:
        Nothing

    Raises:
        Exception if PlantUML .jar file used to compile the .
            ↪ uml to .png files
        is missing.
        Exception if the file with the .uml content is missing
            ↪ .
    """
    abs_diagram_filepath = abspath(relative_filepath_from_root
        ↪ )
    abs_jar_path = abspath(jar_path_relative_from_root)
    if os.path.isfile(abs_diagram_filepath):
        if os.path.isfile(abs_jar_path):
            return abs_diagram_filepath, abs_jar_path
        else:
            raise Exception(
                f"The input diagram file:{abs_diagram_filepath
                    ↪ } does not exist."
            )
    else:
        raise Exception(f"The input jar file:{abs_jar_path}
            ↪ does not exist.")
```

.19. Appendix /src/export_data/plantuml_generate.py

```
# This script generates PlantUML diagrams and outputs them as
    ↪ .uml files.

import os
import subprocess
from os.path import abspath

from .helper_dir_file_edit import
    ↪ create_dir_relative_to_root_if_not_exists
from .helper_dir_file_edit import dir_relative_to_root_exists


def generate_all_dynamic_diagrams(output_dir_relative_to_root)
    ↪ :
    """
    Manages the generation of all the diagrams created in this
        ↪ file.

    Args:
    :param output_dir_relative_to_root: Relative path as seen
        ↪ from the root dir of this project, to which modified
        ↪ files are outputted.

    Returns:
        Nothing

    Raises:
    """
    # Create a example Gantt output file.
    filename_one, lines_one = create_trivial_gantt("
        ↪ trivial_gantt.uml")
    output_diagram_text_file(filename_one, lines_one,
        ↪ output_dir_relative_to_root)

    # Create another example Gantt output file.
    filename_two, lines_two = create_trivial_gantt("
        ↪ another_trivial_gantt.uml")
    output_diagram_text_file(
        filename_two,
        lines_two,
        output_dir_relative_to_root,
    )


def output_diagram_text_file(filename, lines,
    ↪ output_dir_relative_to_root):
    """
    Gets the filename and lines of an PlantUML diagram, and
        ↪ writes these to a
    file at the relative output path.

    Args:
    :param filename: The filename of the PlantUML Gantt file
        ↪ that is being created.
```

```
    :param lines: The lines of the Gantt chart PlantUML code
       ↪ that is being written to file.
    :param output_dir_relative_to_root: Relative path as seen
       ↪ from the root dir of this project, to which modified
       ↪ files are outputted.

    Returns:
        Nothing

    Raises:
        Exception if input file does not exist.
    """
    abs_filepath = abspath(f"{output_dir_relative_to_root}/{
       ↪ filename}")

    # Ensure output directory is created.
    create_dir_relative_to_root_if_not_exists(
       ↪ output_dir_relative_to_root)
    if not dir_relative_to_root_exists(
       ↪ output_dir_relative_to_root):
        raise Exception(
            f"Error, the output directory relative to root:{
               ↪ output_dir_relative_to_root} does not exist."
        )

    # Delete output file if it already exists.
    if os.path.exists(abs_filepath):
        os.remove(abs_filepath)

    # Write lines to file.
    f = open(abs_filepath, "w")
    for line in lines:
        f.write(line)
    f.close()

    # Assert output file exists.
    if not os.path.isfile(abs_filepath):
        raise Exception(f"The input file:{abs_filepath} does
           ↪ not exist.")


def create_trivial_gantt(filename):
    """
    Creates a trivial Gantt chart.

    Args:
    :param filename: The filename of the PlantUML diagram file
       ↪  that is being
    created.

    Returns:
        The filename of the PlantUML diagram, and the lines of
           ↪  the uml content
        of the diagram

    Raises:
```

```
        Nothing
    """
    lines = []
    lines.append("@startuml\n")
    lines.append("[Prototype design] lasts 15 days\n")
    lines.append("[Test prototype] lasts 10 days\n")
    lines.append("\n")
    lines.append("Project starts 2020-07-01\n")
    lines.append("[Prototype design] starts 2020-07-01\n")
    lines.append("[Test prototype] starts 2020-07-16\n")
    lines.append("@enduml\n")
    return filename, lines


def create_another_trivial_gantt(filename):
    """
    Creates a trivial Gantt chart.

    :param filename: The filename of the PlantUML Gantt file
       ↪ that is being created.

    Returns:
        The filename of the PlantUML diagram, and the lines of
           ↪ the uml content
        of the diagram

    Raises:
        Nothing
    """
    lines = []
    lines.append("@startuml\n")
    lines.append("[EXAMPLE SENTENCE] lasts 15 days\n")
    lines.append("[Test prototype] lasts 10 days\n")
    lines.append("\n")
    lines.append("Project starts 2022-07-01\n")
    lines.append("[Prototype design] starts 2022-07-01\n")
    lines.append("[Test prototype] starts 2022-07-16\n")
    lines.append("@enduml\n")

    return filename, lines
```

.20. Appendix /src/export_data/plantuml_get_package.py

```
import os
import subprocess


def check_if_java_file_exists(relative_filepath):
    """
    Safe check to see if file exists or not.

    Args:
    :param relative_filepath: Path as seen from root towards a
       ↪ file.

    Returns:
        True if a file exists.
        False if a file does not exists.

    Raises:
        Nothing
    """
    if os.path.isfile(relative_filepath):
        return True
    else:
        return False


def got_java_file(relative_filepath):
    """
    Asserts if PlantUML .jar file exists. Tries to download is
       ↪ one time if it
    does not exist at the start of the function.

    Args:
    :param relative_filepath: Path as seen from root towards a
       ↪ file.

    Returns:
        True if a file exists.

    Raises:
        Exception if the PlantUML .jar file does not exist
           ↪ after downloading
        it.
    """
    # Check if the jar file exists, curl it if not.
    if not check_if_java_file_exists(relative_filepath):
        # The java file is not found, curl it
        request_file(
            "https://sourceforge.net/projects/plantuml/files/
               ↪ plantuml.jar/download",
            relative_filepath,
        )
    # Check if the jar file exists after curling it. Raise
       ↪ Exception if it is not found after curling.
    if not check_if_java_file_exists(relative_filepath):
```

```
        raise Exception(f"File:{relative_filepath} is not
            ↪ accessible")
    print(f"Got the Java file")
    return True


def request_file(url, output_filepath):
    """
    Downloads a file or file content.

    Args:
    :param url: Url towards a file that will be downloaded.
    :param relative_filepath: The path as seen from the root
        ↪ of this directory, in which files are outputted.

    Returns:
        Nothing

    Raises:
        Nothing
    """
    import requests

    # Request the file in the url
    response = requests.get(url)
    with open(output_filepath, "wb") as f:
        f.write(response.content)


def run_bash_command(bashCommand):
    """
    Unused method. TODO: verify it is unused and delete it.
    :param bashCommand: A string containing a bash command
        ↪ that can be executed.

    """
    # Verbose call.
    # subprocess.Popen(bashCommand, shell=True)
    # Silent call.
    # subprocess.Popen(bashCommand, shell=True, stderr=
        ↪ subprocess.DEVNULL, stdout=subprocess.DEVNULL)

    # Await completion:
    # Verbose call.
    subprocess.call(bashCommand, shell=True)
    # Silent call.
    # subprocess.call(bashCommand, shell=True, stderr=
        ↪ subprocess.DEVNULL, stdout=subprocess.DEVNULL)
```

.21. Appendix /src/export_data/plantuml_to_tex.py

```
from posixpath import abspath
from shutil import copyfile
import shutil
import os.path

from .helper_dir_file_edit import
    ↪ get_dir_filelist_based_on_extension
from .helper_dir_file_edit import
    ↪ create_dir_relative_to_root_if_not_exists


def export_diagrams_to_latex(
    input_dir_relative_to_root, extension,
        ↪ output_dir_relative_to_root
):
    """Loops through the files in a directory and exports them
        ↪ to the latex /Images
    directory.

    :param dir: The directory in which the Gantt charts are
        ↪ being searched.
    :param extension: The file extension that is used/searched
        ↪ in this function. The filetypes that are being
        ↪ exported.
    :param input_dir_relative_to_root: Relative path as seen
        ↪ from the root dir of this project, containing files
        ↪ that modified in this function.
    :param output_dir_relative_to_root: Relative path as seen
        ↪ from the root dir of this project, to which modified
        ↪ files are outputted.

    """
    diagram_filenames = get_dir_filelist_based_on_extension(
        input_dir_relative_to_root, extension
    )
    if len(diagram_filenames) > 0:
        # Ensure output directory is created.
        create_dir_relative_to_root_if_not_exists(
            ↪ output_dir_relative_to_root)

    for diagram_filename in diagram_filenames:
        diagram_filepath_relative_from_root = (
            f"{input_dir_relative_to_root}/{diagram_filename}"
        )

        export_gantt_to_latex(
            diagram_filepath_relative_from_root,
                ↪ output_dir_relative_to_root
        )


def export_gantt_to_latex(relative_filepath_from_root,
    ↪ output_dir_relative_to_root):
    """
```

```
    Takes an input filepath and an output directory as input
      ↪ and copies the
    file towards the output directory.

    :param relative_filepath_from_root: param
      ↪ output_dir_relative_to_root:
    :param output_dir_relative_to_root: Relative path as seen
      ↪ from the root dir of this project, to which modified
      ↪ files are outputted.

    Returns:
        Nothing.

    Raises:
        Exception if the output directory does not exist.
        Exception if the input file is not found.
    """
    if os.path.isfile(relative_filepath_from_root):
        if os.path.isdir(output_dir_relative_to_root):
            shutil.copy(relative_filepath_from_root,
               ↪ output_dir_relative_to_root)
        else:
            raise Exception(
                f"The output directory:{
                   ↪ output_dir_relative_to_root} does not
                   ↪ exist."
            )
    else:
        raise Exception(f"The input file:{
           ↪ relative_filepath_from_root} does not exist.")
```