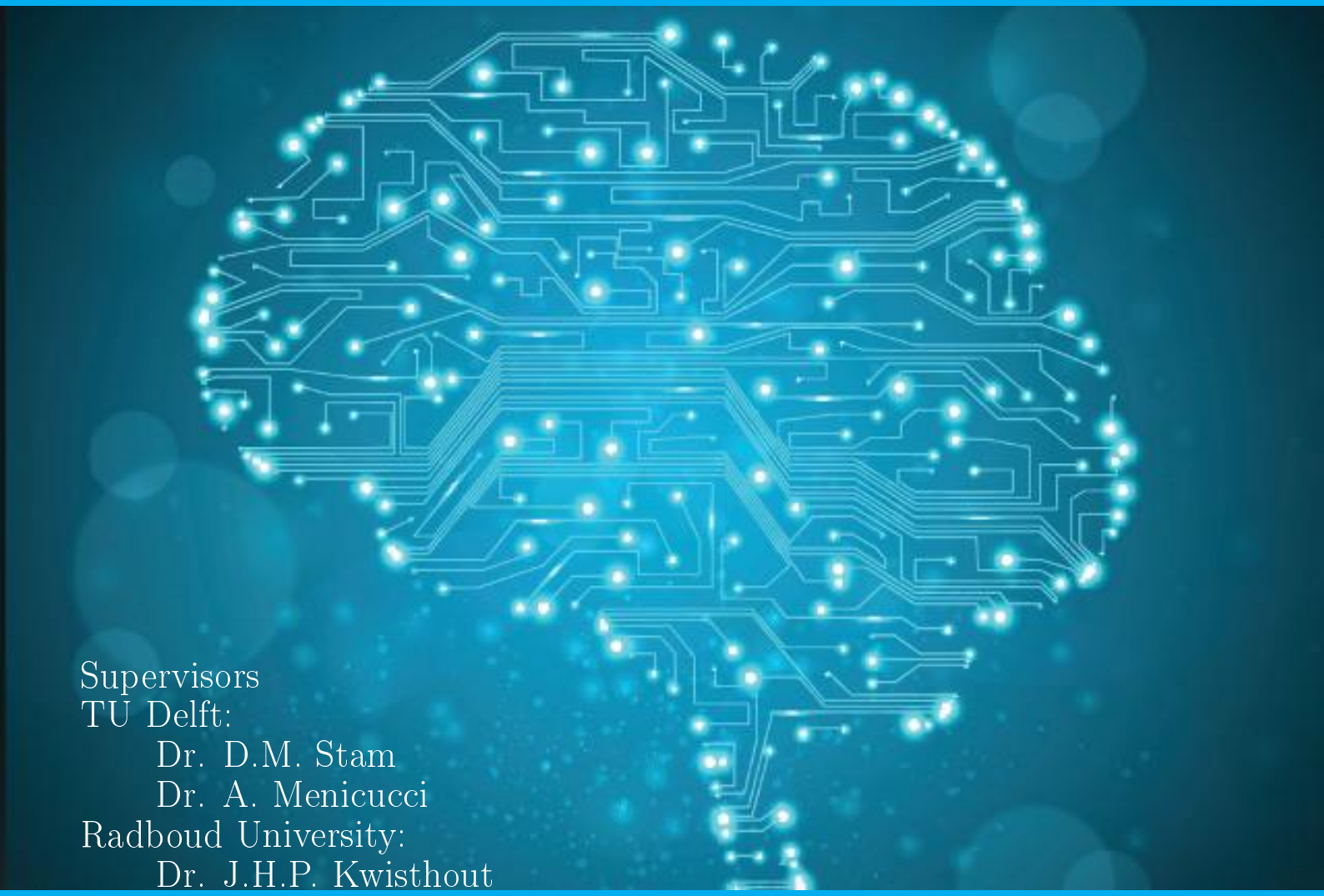
Leveraging brain adaptation to increase radiation resistance of neuromorphic space hardware

AE5810 Thesis Project: baseline report

Akke Toeter



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

TU Delft

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 a tutorial on the creation of a Minimum Dominating Set (MDS) approximation with an accuracy ratio of 16 for MDSs and 32 for minimum total dominating sets.

Chapter 2

Ideas on the realisation of the algorithm by Alipour et al in spiking neural networks using Lava

List of neurons so far discussed:

1. degree_receiver
2. degree_sender
3. r (randomly spiking neuron)
4. w neuron with a spike rate proportional to r + d
5. wta neurons (a circuit connected to all w neurons from the
6. neighbourhood of a given vertex)

2.1. degree_receiver

The weight calculation in the first step can be done by creating a neuron for every node in the graph. Let this neuron be called degree_sender. Furthermore create a neuron for every vertex which is called degree_receiver. The degree_sender neurons are connected to degree_receiver neurons, when the respective vertices are also connected. We could then calculate the degree of a neuron by letting all the degree_senders spike and then counting the spikes in the degree_receiver neuron. Given that we want to keep using this information however, we need to pass this information on to the next part of the process.

2.2. degree_sender

A rate coding for the above situation might be done like this: the degree_receiver neurons have a threshold (henceforth called degree_threshold) higher than the highest degree (but low enough to be time efficient) and the incoming spikes thus make the neuron spike with a frequency that relates to the threshold and the degree. whatever comes downstream of the degree_receiver neuron thus knows about the relative degrees of the vertices.

2.3. r (randomly spiking neuron)

The random number between 0 and 1 that we add thus needs to be added as a randomly spiking neuron that fires with a chance of between 0 and 100 percent per timestep. This chance could be set up at the initialisation phase or maybe introduced by something inherent to loihi or lava (where does randomness in the brain come from anyway.) Given that the nxsdk and the original lava project had a built in stochastically spiking neuron, I'm assuming that it will be implemented in lava as well and until then we can write our own stochastically spiking process. To make the association of the random spiking neuron and the randomly sampled value described in the algorithm more clear, we should create a new neuron called r (for the random spikes) and w

(as the result of d and r in the algorithm) in which the random spikes and the degree_receiver spikes come together.

2.4. w neuron with a spike rate proportional to r + d

We need to find a maximum vertex in the neighbourhood of every vertex. This can be done using winner take all circuits, but we will have to use a separate wta-circuit for every vertex which are driven by the w neurons. This seems excessive, but I'm not sure how to reduce that number of circuits for now. The wta circuit receives the spikes from the relevant w neurons and using inhibitingsynapses, should select the most spiking neuron. This neuron is spiking (once per timestep) towards the w neuron of it's associated vertex and is at the same time inhibiting the degree_receiver neuron. This creates the new spikerate of the w neuron as in the algorithm.

2.5. wta neurons (a circuit connected to all w neurons from the

The runtime of the network is dependent on the precise spike rate implementations. If we choose to use spike rates that are meaningful, they must be distinguishable for every number represented in the network. This means that for example the spikerate representing the degree of the highest degree node and the second highest degree node needs to be different, even after adding the random value. Thus one possibility to achieve this is to use a spike rate close to the degree of the highest node, which would result in a almost constantly spiking w neuron for the node with the highest degree. If this neuron has a degree of 10, then a node with degree one would only spike at time 10. thus we would need to measure this spike rate over a window of at least length 10. For finer measurements, we could introduce longer windows (not sure yet). However, in this example, the window of a number of spikes represents one "communication round" as defined in the distributed algorithms literature. Thus letting the algorithm run multiples of 10 amounts to selecting m in the original algorithm. Given the precise implementation of these ideas there might be some "burn in time" or processes that need to start up, which add a constant amount of time to this, but the main idea should be clear. The wta circuits are self improving because they after silencing the other neurons they adjust the weights of their own input and thus allow the wta to select a different winner. (This needs to be the case and should be possible given the correct fine tuning of the synapse weights, but I have no proof of it working.)

2.6. neighbourhood of a given vertex)

At the end, we can attach measuring neurons that never spike to check on the wta winner neurons at a certain timestep (might need to be a multiple of 10 given the above example)

Bibliography

Acronyms

ALU Arithmatic Logic Unit. 7

ANN Artificial Neural Network. 7

BIOS Basic Input/Output System. 7

BISER Built-in Soft Error Resillience. 7

BIT Binary Digit. 7

BJT Bipolar Junction Transistor. 7

CME Coronal Mass Ejection. 7

CMOS Complementary MetalâĂŞOxideâĂŞSemiconductor. 7

CPU Central Processing Unit. 7

DEC Double-error Correcting Code. 7

DICE Dual Interlocked Storage Cell. 7

DNN Deep Neural Network. 7

DNU Dual-node Upset. 7

DOD Department of Defence. 7

DOT Design Options Structuring Tree. 7

DRAM Dynamic Random Access Memory. 7

ECC Error Correction Code. 7

ELT Enclosed Layout Transistor. 7

ESA European Space Agency. 7

FBD Functional Break-down Diagram. 7

FET Field-Effect Transistor. 7

FFD Functional Flow Diagram. 7

GCD Greatest Common Divisor. 7

GCR Galactic Cosmic Ray. 7

IC Integrated Circuit. 7

INRC Intel Neuromorphic Research Community. 7

LCM Least Common Multiple. 7

LET Linear Energy Transfer. 7

MOS MetalâĂŞOxideâĂŞSemiconductor. 7

MOSFET MetalâĂŞOxideâĂŞSemiconductor Field-Effect Transistor. 7

MPNN Multilayer Perceptron Neural Network. 7

MSN Mission Need Statement. 7

NVM Non-Volatile Memory. 7

POS Project Objective Statement. 7

RAM Random Access Memory. 7

RDT Requirements Discovery Tree. 7

ROM Read Only Memory. 7

SEC Single-error Correcting Code. 7

SEE Single-event Effects. 7

SEGR Single-event Gate Rupture. 7

SEIB Single-event Induced Burnout. 7

SEL Single-event Latch-up. 7

SER Soft-error Rate. 7

SERL Soft-error Resilient Latch. 7

SES Single-event Snapback. 7

SET Single-event Transient. 7

SEU Single-event Upset. 7

SNN Spiking Neural Network. 7

SNU Single-node Upset. 7

SPE Solar Particle Events. 7

SPENVIS Space Environment Information System. 7

SRAM Static Random Access Memory. 7

STDP Spike-Timing-Dependent Plasticity. 7

TLB Translation Lookaside Buffer. 7

TMR Tripple-mode Redundancy. 7

VLSI Very-Large-Scale Integration. 7

WBS Work Break-down Structure. 7

WFD Work Flow Diagram. 7

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.. 7

CPU cache A small hardware memory unit that is faster than the main memory and closer to the Arithmatic Logic Unit.. 7

data cache A cache that is designed to increase the speed with which data is fetched and stored.. 7

formula A mathematical expression. 7

instruction cache A cache that is designed to increase the speed with which instructions are fetched.. 7

latex Is a mark up language specially suited for scientific documents. 7

mathematics Mathematics is what mathematicians do. 7

memory cell A fundamental/basic unit in computing that is used to store information.. 7

n-type semiconductor A semiconductor that is doped with electron donors. 7

p-n junction A boundary interface of two a p-type semiconductor and a n-type semiconductor. 7

p-type semiconductor A semiconductor that is doped with electron acceptors. 7

primary memory A form of memory that is only accessible to the Central Processing Unit which reads instructions from it and executes those instructions.. 7

prompt charge The charge that is collected by means of funnelling.. 7

random-access A memory type that has access times that are independent of its physical location.. 7

unipolar transistors Unipolar transistors are transistors that use either electrons or electron holes as charge carriers and not the combination of the two.. 7

Nomenclature

c	Speed of light in a vacuum inertial frame
h	Planck constant

.1. Appendix /src/__main__.py

```
## 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 (
    create_triangle_free_graph,
    create_triangle_free_planar_graph,
    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)

# Generate a random triangle free graph
# create_triangle_free_graph(False)

# Generate a random triangle free planar graph
# nr_nodes = 7
# edge_probability = 0.85
# seed = 42
# G = create_triangle_free_planar_graph(nr_nodes,
   ↪ edge_probability, seed, False)

## 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 random
import networkx as nx

import matplotlib.pyplot as plt

from src.graph_properties import is_planar, is_triangle_free


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

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

```
            ("f", "g"),
        ]
    )
    return graph


def create_triangle_free_graph(show_graphs):
    seed = 42
    nr_of_nodes = 10
    probability_of_creating_an_edge = 0.85
    nr_of_triangles = 1  # Initialise at 1 to initiate while
        ↪ loop.
    while nr_of_triangles > 0:
        G = nx.fast_gnp_random_graph(
            nr_of_nodes, probability_of_creating_an_edge
        )
        nr_of_triangles = nx.triangles(G, 0)
        print(f"nr_of_triangles={nr_of_triangles}")
        if show_graphs:
            plot_graph(G)
    return G


def create_triangle_free_planar_graph(
    nr_nodes, edge_probability, seed, show_graph
):
    G = nx.Graph()
    G.add_nodes_from(list(range(nr_nodes)))

    # Loop over all edge spaces:
    for u in G.nodes():
        for v in G.nodes():
            # Don't add an edge to a node itself.
            if u != v:

                # Add edge to compute implications.
                G.add_edge(u, v)

                # Check if adding edge yields a triangle free
                    ↪ and planar graph.
                if is_triangle_free(G) and is_planar(G):

                    # Apply probability of creating edge
                    # TODO: Use 1-edge_probability to get bias
                        ↪  against removing edges.
                    # TODO: Use seed to get certain list of
                        ↪ random booleans.
                    if bool(random.getrandbits(1)):
                        G.remove_edge(u, v)
                else:
                    G.remove_edge(u, v)

    # Verify all nodes are connected to graph.
    if not nx.is_connected(G):
        G = add_connections_to_make_graph_connected(G)
```

```
    if show_graph:
        plot_graph(G)
    return G


def add_connections_to_make_graph_connected(G):
    # Loop over all edge spaces:
    for u in G.nodes():
        for v in G.nodes():
            # Don't add an edge to a node itself.
            if u != v:
                if not G.has_edge(u, v):

                    # Add edge to compute implications.
                    G.add_edge(u, v)

                    # Check if adding edge yields a triangle
                    ↪ free and planar graph.
                    if is_triangle_free(G) and is_planar(G):
                        if nx.is_connected(G):
                            return G
                    else:
                        G.remove_edge(u, v)


# 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

```
import random
import networkx as nx


def compute_mtds(input_graph, m=0):
    # As currently no decision has been reached as to what
       ↪ kind of object
    # input_graph actually is, this code will accept both node
       ↪ and edges lists
    # as well as nx.graphs.
    if isinstance(input_graph, nx.Graph):
        graph = nx.Graph()
        graph.add_nodes_from(
            list(input_graph.nodes),
            marks=0,
            random_number=0,
            weight=0,
        )
        graph.add_edges_from(list(input_graph.edges))
    if isinstance(input_graph, tuple):
        graph = nx.Graph()
        graph.add_nodes_from(
            input_graph[0],
            marks=0,
            random_number=0,
            weight=0,
        )
        graph.add_edges_from(input_graph[1])

    for node in graph.nodes:
        graph.nodes[node]["marks"] = 0
        # 1.a Each vertex v_i chooses random float r_i in
           ↪ range 0<r_i<1
        graph.nodes[node]["random_number"] = random.random()
        # 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. (Not
        # happening due to a centralised version)
        graph.nodes[node]["weight"] = (
            graph.degree(node) + graph.nodes[node]["
               ↪ random_number"]
        )

    for node in graph.nodes:
        # 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.
        max_weight = max(
            graph.nodes[n]["weight"] for n in nx.all_neighbors
               ↪ (graph, node)
        )
        for n in nx.all_neighbors(graph, node):
```

```
            if graph.nodes[n]["weight"] == max_weight:
                # 2.b Each vertex v_i adds a mark to that
                    ↪ neighbour vertex
                # v_j_(w_max).
                graph.nodes[n]["marks"] += 1

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

        for node in graph.nodes:
            # 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
            graph.nodes[node]["weight"] = (
                graph.nodes[node]["marks"] + graph.nodes[node
                    ↪ ]["random_number"]
            )
            # 5. Reset marked vertices: for each vertex v_i, (
                ↪ x_i)_k=0
            graph.nodes[node]["marks"] = 0

        for node in graph.nodes:
            # 6.a Each vertex v_i sends w_i to each of its
                ↪ neighbours.
            # 6.b Each vertex v_i gets the index of the
            # neighbouring vertex v_j_(w_max) that has the
                ↪ heighest w_i, with
            # i!=j.
            max_weight = max(
                graph.nodes[n]["weight"] for n in nx.
                    ↪ all_neighbors(graph, node)
            )
            for n in nx.all_neighbors(graph, node):
                if graph.nodes[n]["weight"] == max_weight:
                    # 6.d Each vertex v_i adds a mark to that
                        ↪ neighbour vertex
                    # v_j_(w_max).
                    graph.nodes[n]["marks"] += 1

    dset = []
    for n in graph.nodes.data():
        if n[1]["marks"] > 0:
            dset.append(n[0])

    return dset
```

.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


def add_two(x):
    return x + 2
```

.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:

        """
```

```
        # 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):
    # 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 (
                    (excluded_files is not None)
                    and (file not 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 (
            (excluded_files is not None) and (file not 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 delete_dir_if_exists(dirpath):
    if os.path.exists(dirpath) and os.path.isdir(dirpath):
        shutil.rmtree(dirpath)


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)


def add_two(x):
    return x + 2
```

.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.
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 = (
        "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 = "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 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 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 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},"
        + f" 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
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: Relative filepath as
       ↪ seen from root of
    file that is used 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.

    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."
        )
```