testbench

October 24, 2022

1 AtMonSat Detection Algorithm Test Bench

Datapoints from a dataset are send to the C++ implementation for evaluation and results are recorded.

1.0.1 Settings

Microcontroller / processor frequency (this value is used to select some settings (e.g. baudrate) and define the filename to store the results)

[ ]:
```
microcontroller_frequency = '39'
```

Connection type
A connection can be done to an implementation compiled for the host computer using pipes or to an implementation running on a microcontroller (e.g. STM32H743) using serial communication.

[ ]:
```
# connection_type = 'local'
connection_type = 'remote'
```

Settings for a remote connection
connection_baudrate is the baudrate to use for the communication with the microntroller. This setting must match the baurate defined in the firmware.
connection_port is the interface on the local machine that connects to the remote microcontroller.

[ ]:
```
connection_baudrate = {'298': 3686400,
                       '146': 1843200,
                       '78': 921600,
                       '39': 460800}[microcontroller_frequency]  # this must␣
 ↪match the baudrate of the firmware
connection_port = '/dev/ttyACM0'
```

Settings for a local connection
connection_local_command is a list with the command as the first member.

[ ]:
```
connection_local_command = ['../atmonsat']
```

Dataset to transmit
All datasets are located in the datasets directory. The use_dataset variable defines which dataset to use.

These measurements have been taken from the EPS of a cubesat. TEMP_0 to TEMP_3 are MPPT temperatures TEMP_4 to TEMP_8 are voltage regulator temperatures and TEMP_9 represents the battery temperature.
All temperatures have a resolution of 1°C.

Several ranges have been defined for each dataset. - 'normal': range where no anomaly happens. - 'abnormal': range where one or multiple anomalies have been introduced. - 'experiment': normal + abnormal range.

The use_range variable can be used to select what range of datapoints to use for evaluation.

[ ]:
```
use_dataset = '2022.06.08'
use_range = 'experiment'
```

Number of datapoints
limit_number_of_datapoints limits the number of datapoints from dataset to be transmitted for evaluation. - None: all datapoints will be transmitted - any integer > 0: Maximum number of datapoints to transmit

[ ]:
```
limit_number_of_datapoints = None # <int> or None
```

Threshold hold-off
A hold-off can be activated that suppresses any further detection after an anomaly detection during the number of specified iterations.
threshold_hold_off can take any positive integer value. 0 disables the postprocessing.

[ ]:
```
threshold_hold_off = 60
```

Output directory (where to store the results). A directory will be created inside this output directory for each measurement.

[ ]:
```
output_directory = './results'
```

Create output directory

[ ]:
```
import datetime
import os
timestamp_string = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
output_directory = output_directory + os.path.sep + timestamp_string
os.mkdir(output_directory)
print("Results store in {}".format(output_directory))
```

```
Results store in ./results/20221024-082511
```

Save experiment settings for later reference into meta.json

[ ]:
```
import json

settings = {
    'timestamp': timestamp_string,
```

```
    'dataset': use_dataset,
    'range': use_range,
    'frequency': microcontroller_frequency,
    'connection_type': connection_type,
    'limit': limit_number_of_datapoints,
    'threshold_hold_off': threshold_hold_off,
}

print('Settings:')
for k, v in settings.items():
    print(" - {} = {}".format(k, v))

filename = output_directory + os.path.sep + 'settings.json'
with open(filename, 'w') as file:
    json.dump(settings, file)
    print("Settings saved to '{}'".format(filename))
```

```
Settings:
 - timestamp = 20221024-082511
 - dataset = 2022.06.08
 - range = experiment
 - frequency = 39
 - connection_type = remote
 - limit = None
 - threshold_hold_off = 60
Settings saved to './results/20221024-082511/settings.json'
```

1.0.2 Includes

[ ]:
```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
import datetime
import time
import pickle
import sys

enable_example = False
%run dataset.ipynb
%run protocol.ipynb
```

1.0.3 Logger

Information, warning and errors are getting logged into atmonsat_testbench.txt. The variable display_log is a boolean that defines if the same information should be visualized on screen.

[ ]:
```
display_log = True
```

[ ]:
```
import logging
logging.basicConfig(filename='atmonsat_testbench.txt',
                    format='%(levelname)s:%(message)s',
                    encoding='utf-8',
                    filemode='w',
                    level=logging.INFO)

if display_log:
    logging.getLogger().addHandler(logging.StreamHandler())
```

The following function is getting called by the communication protocol and can be used to visualize data exchange between the host and the target.

[ ]:
```
def comment(s):
    # print(s)
    pass
```

1.0.4 Preparation

Load the specified dataset

[ ]:
```
dataset = Dataset(use_dataset)[use_range]
```

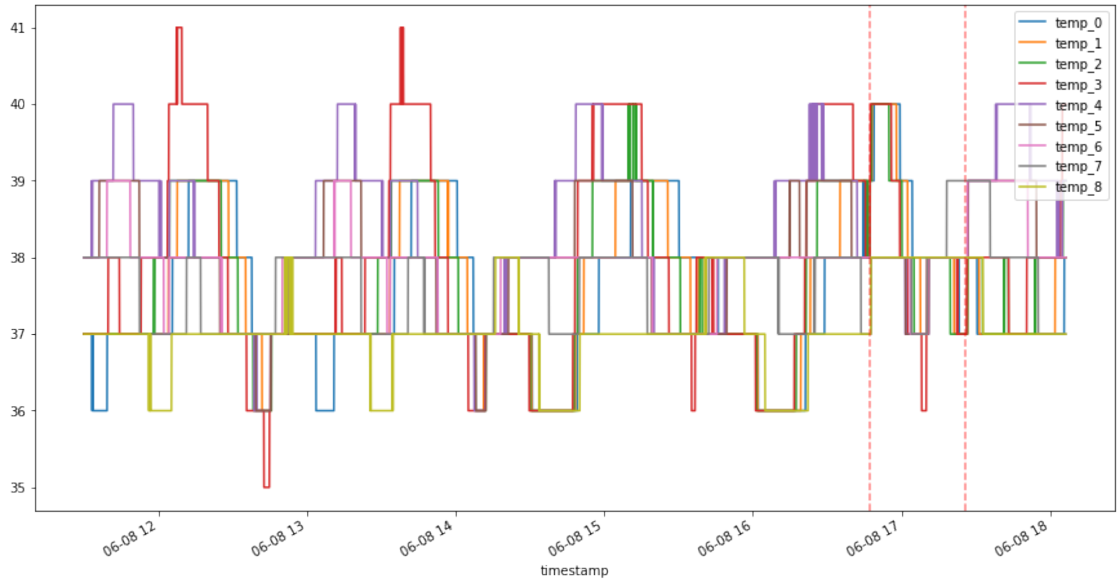
Features
Features to be packed into a datapoint and send to the implementaion. The order of the features must correspond to the ordering of a datapoint in the C++ implementation

[ ]:
```
feature_columns = ['temp_0', 'temp_1', 'temp_2', 'temp_3',
                   'temp_4', 'temp_5', 'temp_6', 'temp_7', 'temp_8']
```

Visualize the selectted range of the dataset for inspection

[ ]:
```
fig, ax = plt.subplots(1, 1)
fig.set_figwidth(15)
fig.set_figheight(8)
dataset.plot(not_columns=['angle', 'sin_of_angle'],
             ax=ax).plot_anomalies(ax=ax)
```

[ ]:
```
<__main__.Dataset at 0x7f5b5787c1c0>
```



Datapoints

Prepare a list of datapoint commands to be send to the implementation.

[ ]:
```
commands = []
for index, row in dataset.dataframe[feature_columns].iterrows():
    if not limit_number_of_datapoints is None:
        if len(commands) > limit_number_of_datapoints:
            break
    commands.append(CommandDatapointINT8(data=row.to_numpy(dtype=np.int8)))
```

Datastore

Datastore is a class that collects incoming distance and detection responses from the implementation.

[ ]:
```
class DataStore(object):

    def __init__(self):
        self.__df = pd.DataFrame(
            columns=['detection_recording_timestamp', 'distance', 'detection'])
        self.__row = 0
        # distance and detection arrive independant of each other: wait for␣
 ↪both, then store the row
        self.__distance_arrived = False
        self.__detection_arrived = False
        self.__recorded_distance = 0
        self.__recorded_detection = False

    @property
```

```
    def dataframe(self):
        return self.__df

    def save(self, filename="distance_detection.csv", sep=";"):
        self.__df.to_csv(filename, sep=sep)

    def __store(self):
        if self.__distance_arrived and self.__detection_arrived:
            # Store received row
            self.__df.loc[self.__row] = [time.time(),
                                         self.__recorded_distance, self.
 ↪__recorded_detection]
            self.__row = self.__row + 1
            self.__distance_arrived = False  # reset flag
            self.__detection_arrived = False  # reset flag

    def store_distance(self, distance):
        # logging.info("Malanobis distance={}".format(distance))
        self.__recorded_distance = distance
        if self.__distance_arrived:
            logging.error(
                "Recevied distance a second time before detection. This should␣
 ↪not happen.")
        self.__distance_arrived = True
        self.__store()

    def store_detection(self, detection):
        # logging.info("Detection={}".format(detection))
        self.__recorded_detection = detection
        if self.__detection_arrived:
            logging.error(
                "Recevied detection a second time before distance. This should␣
 ↪not happen.")
        self.__detection_arrived = True
        self.__store()


datastore = DataStore()
```

ExecutionTimeStore

ExecutionTimeStore is a class that collects incoming execution_time information.

[ ]:
```
class ExecutionTimeStore(object):

    def __init__(self):
        self.__df = pd.DataFrame(columns=['execution_time_recording_timestamp',␣
 ↪'execution_time'])
```

```
        self.__row = 0

    @property
    def dataframe(self):
        return self.__df

    def response_handler_callback_execution_time(self, duration):
        # logging.info("Execution time={}".format(duration))
        self.__df.loc[self.__row] = [time.time(), float(duration*1e-9)]
        self.__row = self.__row + 1

    def save(self, filename="execution_times.csv", sep=";"):
        self.__df.to_csv(filename, sep=sep)

execution_time_store = ExecutionTimeStore()
```

response_handler_callback_remote_exception is a response handler that is getting called when an exception occured in the implementation (local or remote)

[ ]:
```
def response_handler_callback_remote_exception(str):
    text = "A remote exception occured: {}".format(str)
    logging.fatal("Exception={}".format(text))
    raise Exception(text)
```

Communication response handlers

response_handlers is a list that defines all allowed responses from the microcontroller and what handlers to call after reception.

[ ]:
```
response_handlers = [
    CommandInt32(PROTOCOL_COMMAND_ID_INT32,
                 callback=lambda d: logging.info("Got int32={}".format(d))),
    CommandFloat(PROTOCOL_COMMAND_ID_FLOAT,
                 callback=lambda d: logging.info("Got float={}".format(d))),
    CommandFloat(PROTOCOL_COMMAND_ID_MAHALANOBIS_DISTANCE,
                 callback=datastore.store_distance),
    CommandUInt8(PROTOCOL_COMMAND_ID_DETECTION,
                 callback=datastore.store_detection),
    CommandComment(PROTOCOL_COMMAND_ID_COMMENT,
                   callback=lambda d: logging.info("Comment={}".format(d))),
    CommandComment(PROTOCOL_COMMAND_ID_EXCEPTION,␣
 ↪callback=response_handler_callback_remote_exception ),
    CommandUInt32(PROTOCOL_COMMAND_ID_EXECUTION_TIME,␣
 ↪callback=execution_time_store.response_handler_callback_execution_time)
]
```

Connect to the target

Connection can be done on a local target (compiled for the host) or a remote target over serial communication.

```
[ ]: if connection_type == "local":
         connection = ConnectionSubprocess(command=connection_local_command,
                                           response_handlers=response_handlers)
     elif connection_type == "remote":
         connection = ConnectionSerial(port=connection_port,
                                       baudrate=connection_baudrate,
                                       response_handlers=response_handlers)
     else:
         print("Connection type must be 'pipe' or 'serial' ")

     connection.open()
```

Initialize target
This command initializes or resets the implementation. It has been implemented as a command as the initialization code might try to report errors which is a communication from target->host that is only allowed in reply to a command.

```
[ ]: connection.emit(CommandInitialize())
```

```
Comment=target initialized.
```

```
[ ]: True
```

Upload coefficients
Upload coefficients stored in a save_variables.pkl to the implementation. The uploaded coefficients are: - detection_threshold - mahalanobis inverse covariance matrix - mahalanobis mean vector

```
[ ]: filename = 'saved_variables.pkl'
     if os.path.exists(filename):
         with open(filename, 'rb') as file:
             variables_dict = pickle.load(file)
             # logging.info("Threshold: {}".
      ↪format(float(variables_dict['detection_threshold'])))
             # logging.info("Matrix: {}".format(np.
      ↪array(variables_dict['mahalanobis_matrix']).flatten()))
             # logging.info("Mean: {}".format(np.
      ↪array(variables_dict['mahalanobis_mean']).flatten()))

             connection.emit(CommandFloat(id=PROTOCOL_COMMAND_ID_THRESHOLD,
                                          ␣
      ↪data=float(variables_dict['detection_threshold'])))
             connection.
      ↪emit(CommandMatrix(id=PROTOCOL_COMMAND_ID_MAHALANOBIS_INVERSE_COVARIANCE_MATRIX,
                                           data=np.
      ↪array(variables_dict['mahalanobis_matrix']).flatten()))
             connection.emit(CommandMatrix(id=PROTOCOL_COMMAND_ID_MAHALANOBIS_MEAN,
```

```
                                           data=np.
    ↪array(variables_dict['mahalanobis_mean']).flatten()))
```

```
Comment=treshold updated
Comment=mahalanobis inverse covariance matrix has been updated
```

[ ]:
```
# connection.emit(CommandFloat(id=PROTOCOL_COMMAND_ID_THRESHOLD,␣
    ↪data=float(1e6)))
```

Upload hold-off

[ ]:
```
connection.emit(CommandUInt32(id=PROTOCOL_COMMAND_ID_THRESHOLD_HOLD_OFF,␣
    ↪data=threshold_hold_off))
```

```
Comment=hold-off updated
```

[ ]: True

1.0.5 Processing

Take time before sending all datapoints to estimate the total required time.

[ ]:
```
start_time = time.time()
```

Upload the datapoints

[ ]:
```
connection.emit(commands)
```

[ ]: True

Total required time
This is the time of the evaluation + testbench and communication overhead. It gives an uppper bound on the duration.

[ ]:
```
total_evaluation_time = time.time() - start_time
print("Duration of evaluation + communication + testbench: {} [s]".
    ↪format(total_evaluation_time))
```

```
Duration of evaluation + communication + testbench: 141.68548607826233 [s]
```

Store the received data and extend the dataset with responses

[ ]:
```
datastore.save(filename = output_directory + os.path.sep + "detection.csv")
execution_time_store.save(filename = output_directory + os.path.sep +␣
    ↪"execution_time.csv")

missing_values = len(dataset.dataframe.index) - len(datastore.
    ↪dataframe['detection'])
```

```
print("Number of missing detection responses: {} / {}".format(missing_values,␣
  ↪len(dataset.dataframe.index)))
dataset.dataframe['detection'] = list(datastore.dataframe['detection']) +␣
  ↪[0]*missing_values
dataset.dataframe['distance'] = list(datastore.dataframe['distance']) +␣
  ↪[0]*missing_values
dataset.dataframe['detection_recording_timestamp'] = list(datastore.
  ↪dataframe['detection_recording_timestamp']) + [0]*missing_values

missing_values = len(dataset.dataframe.index) - len(execution_time_store.
  ↪dataframe['execution_time'])
print("Number of missing execution_time responses: {} / {}".
  ↪format(missing_values, len(dataset.dataframe.index)))
dataset.dataframe['execution_time'] = list(execution_time_store.
  ↪dataframe['execution_time']) + [0]*missing_values
dataset.dataframe['execution_time_recording_timestamp'] =␣
  ↪list(execution_time_store.dataframe['execution_time_recording_timestamp']) +␣
  ↪[0]*missing_values
```

```
Number of missing detection responses: 8 / 4639
Number of missing execution_time responses: 0 / 4639
```

Save the dataset with the results to CSV

[ ]:
```
filename = output_directory + os.path.sep + "testbench_data.csv"
print("Dataset and results have been stored into: {}".format(filename))
dataset.save_dataframe_as_csv(filename=filename)
```

```
Dataset and results have been stored into:
./results/20221024-082511/testbench_data.csv
```

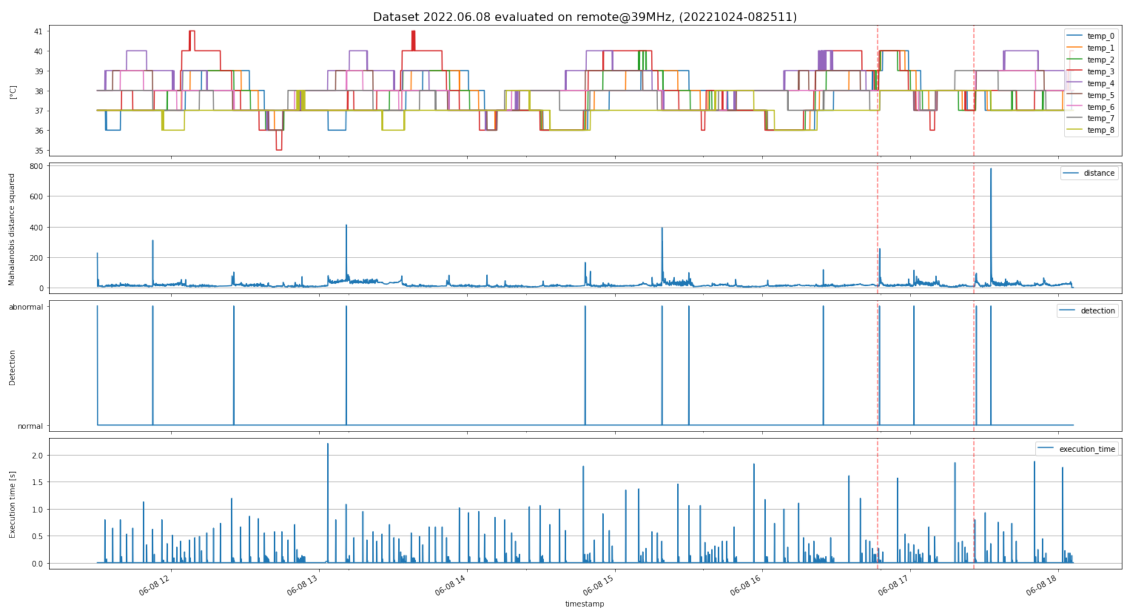
[ ]:
```
<__main__.Dataset at 0x7f5b5787c1c0>
```

1.0.6 Postprocessing

Visualize the dataset and the results

[ ]:
```
ylabel_position = (-0.030, 0.5)
fig, axs = plt.subplots(4, 1, sharex=True)
fig.set_figwidth(26)
fig.set_figheight(15)
axs[0].set_title("Dataset {} evaluated on {}@{}MHz, ({})".format(dataset.
  ↪dataset_name, connection_type, microcontroller_frequency,␣
  ↪timestamp_string),fontsize=16)
axs[0].set_ylabel("[°C]")
axs[0].yaxis.set_label_coords(*ylabel_position)
dataset.plot(columns=feature_columns, ax=axs[0]).plot_anomalies(ax=axs[0])
axs[1].set_ylabel("Mahalanobis distance squared")
```

```
axs[1].yaxis.set_label_coords(*ylabel_position)
dataset.plot(columns=['distance'], ax=axs[1], grid=True).
  ↪plot_anomalies(ax=axs[1])
axs[1].grid(axis='x')
axs[2].set_ylabel("Detection")
axs[2].yaxis.set_label_coords(*ylabel_position)
axs[2].set_yticks([1.0, 0.0], ["abnormal", "normal"])
dataset.plot(columns=['detection'], ax=axs[2]).plot_anomalies(ax=axs[2])
axs[3].set_ylabel("Execution time [s]")
axs[3].yaxis.set_label_coords(*ylabel_position)
dataset.plot(columns=['execution_time'], ax=axs[3], grid=True).
  ↪plot_anomalies(ax=axs[3])
axs[3].grid(axis='x')
fig.subplots_adjust(hspace=0.05)
filename = output_directory + os.path.sep + "testbench_results.png"
plt.savefig(filename)
```



Execution time statistics

```
[ ]: column = dataset.dataframe['execution_time']

execution_time_statistics = {"mean": float(column.mean()),
                "median": float(column.median()),
                "std": float(column.std()),
                "min": float(column.min()),
                "max": float(column.max()),
                "sum": float(column.sum()),
```

```
            "sample_size": len(column)
            }

print("Execution time / datapoint statistics")
for k, v in execution_time_statistics.items():
    print("{} \t= {} [s]".format(k, v))

print("")
print("Description")
print(column.describe())

statistics = {"execution_time": execution_time_statistics}

filename = output_directory + os.path.sep + "execution_time_statistics.json"
with open(filename, 'w') as file:
    json.dump(statistics, file)
    print("Statistics stored to '{}'".format(filename))
```

```
Execution time / datapoint statistics
mean    = 0.022028150032334 56 [s]
median  = 5e-06 [s]
std     = 0.12943470578734137 [s]
min     = 4.000000000000001e-06 [s]
max     = 2.209719 [s]
sum     = 102.18858800000001 [s]
sample_size     = 4639 [s]

Description
count    4639.000000
mean        0.022028
std         0.129435
min         0.000004
25%         0.000005
50%         0.000005
75%         0.000005
max         2.209719
Name: execution_time, dtype: float64
Statistics stored to './results/20221024-082511/execution_time_statistics.json'
```

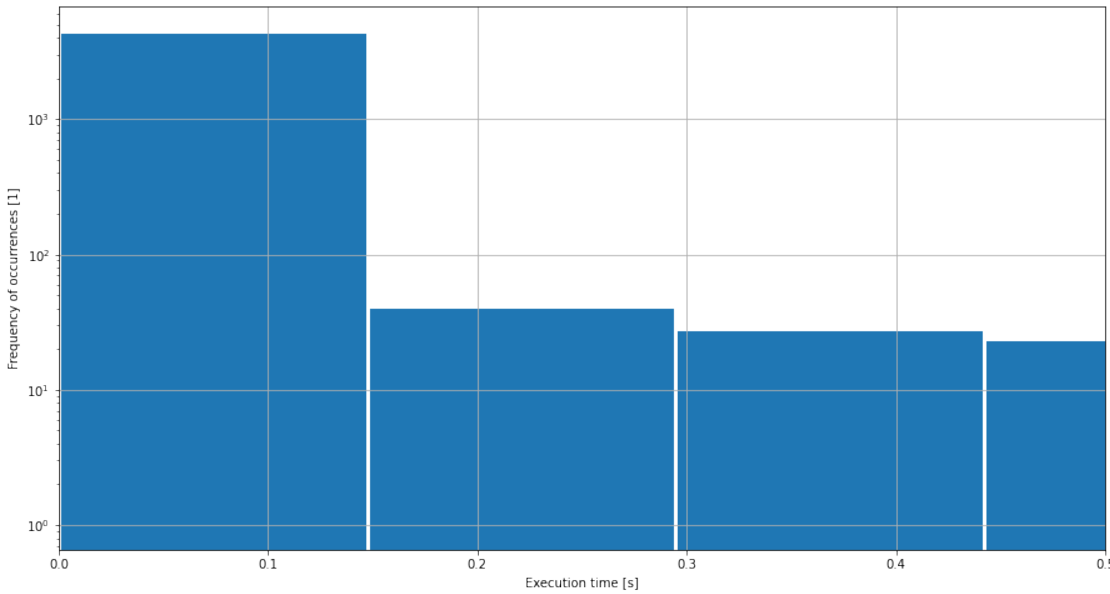
Histogram of excution duration

[ ]:

```
if connection_type == "local":
    ax=column.plot.hist(column=["execution_time"], bins=15, logy=True,␣
↪xlim=[0,0.001], figsize=(15,8), edgecolor='white', linewidth=3, grid=True)
else:
    ax=column.plot.hist(column=["execution_time"], bins=15, logy=True, xlim=[0,␣
↪0.5], figsize=(15,8), edgecolor='white', linewidth=3, grid=True)
```

```
ax.set_xlabel("Execution time [s]")
ax.set_ylabel("Frequency of occurrences [1]")

filename = output_directory + os.path.sep + "testbench_histogram.png"
plt.savefig(filename)
```



Communication and testbench overhead

Calculate time overhead due to communication and testbench

```
[ ]: time_overhead = total_evaluation_time - column.sum()
     print("Total duration: {} [s]".format(total_evaluation_time))
     print("Evaluation duration: {} [s]".format(column.sum()))
     print("Overhead (communication and testbench): {} [s]".format(time_overhead))
```

```
Total duration: 141.68548607826233 [s]
Evaluation duration: 102.18858800000001 [s]
Overhead (communication and testbench): 39.49689807826232 [s]
```

1.0.7 Finalize

Terminate the connection

```
[ ]: connection.close()
```