Description of the scripts data extraction & r datatreatment

data extraction script

Data extraction

To access and manipulate the data stored in the `.bag` files, the C++ library provided by ROS is used. The code to make the data accessible roughly has the following structure:

1. Run functions to subscribe to the topics containing the live streams of the data and write the information contained therein to memory. Both the variables contained in a specific topic, as well as the time stamp, are written to memory as ordered arrays. Below, the generic structure of such a function is provided:

```
void Subscriber::subscriberFunctionForTopic1(const
    message_type::Topic::ConstPtr& pointer_topic1){
   variable1_type variable1_name = pointer_topic1 ->
       topic_member_corresponding_to_variable1;
   variable2_type variable2_name = pointer_topic1 ->
       topic_member_corresponding_to_variable2;
   // etc. for all the variables stored in the topic
   this -> variable1_member_array.push_back(variable1); // here, the
       variable is stored to memory in the form of an ordered array
   this -> variable2_member_array.push_back(variable2);
   // etc. for all the variables stored in the topic
   std_msgs::Header hdr = pointer_name -> header; // to access the time
       stamp, which is a string in the member 'header' of the topic
   int secs = hdr.stamp.sec;
   int nanosecs = hdr.stamp.nsec;
   std::stringstream ss;
   ss << secs << "." << nanosecs;
   std::string time = ss.str();
   this -> time.push_back(time); // here, the time stamp is stored to
       memory as a variable in the form of an array
   }
// etc. for all other topics.
```

   Note that since in the case of some topics, the variables had been grouped as dimensions of member arrays, the above generic structure sometimes had to be adjusted significantly.

2. Access the data from memory to perform different operations with it. The `Subscriber` class, to which the subscriber functions explained above belong, stores the data as members, and contains functions to access these members. These functions have the following generic structure:

```
variable_type & Subscriber::getVariable(){
   return this -> variable_name;
}
// etc. for all other variables from all topics.
```

   The fact that the programme needs to subscribe to the live streams to store the data requires that to access the data, the code must be run for the duration of the live stream in order to gather all the data points. However, the fact that the data is written to memory also means

that it then becomes instantly available in the form of ordered arrays, which can be accessed from other programmes or functions. Currently, two categories of such functions are available: 1) data treatment functions, which perform operations on the data, and 2) writer functions, which write the data to file.

Data treatment

This code allows to perform different data cleaning and treatment operations. Currently, only two such functions are available, but this code will be extended based on the identified needs. The currently available functions are briefly explained below:

1. A function to convert the time stamp from system time to a time stream starting from zero: This enhances visualisation, since the time is output in seconds, and the system time starts at the first initialisation of the rover, so that the time stamp starts at some billion seconds. The function has the structure presented below. It takes the time stamp as a string and outputs a time stream starting from zero and which has the format of a double (long floating point numbers).

```
std::vector<double> CleanData::convertTimeToDouble(std::vector<std::string>
    time){
   std::vector<double> time_double; // convert the array of strings to an
       array of doubles
   for (int i = 0; i < time.size(); i++){
      std::string x = time[i];
      double y = ::atof(x.c_str());
      time_double.push_back(y);
   }
int start_time = time_double[0]; // get the value of the first time stamp,
    which is then substracted from the time stream so that it starts from
    zero and increments from there
std::vector<double> time_double_clean; // generate an array of time stamps
    which starts from zero and increments from there
for (int i = 0; i < time_double.size(); i++){
   double time = time_double[i];
   time_double_clean.push_back(time - start_time);
}
return time_double_clean;
}
```

2. A function to average data to the nearest second by taking the median: This allows for cross – comparisons between variables. Since the time stamp is recorded by the nanosecond, no two data points are recorded at exactly the same point in time, since the different sensors have different publishing rates. Consequently, to compare the different variables, the time stamps need to be homogenised to the nearest second to allow for analyses such as correlation analyses. Furthermore, this method also allows to reduce the data volume for variables which are recorded every couple of nanoseconds. It was also hoped that it would allow to eliminate some measurement errors which led to outliers; however, as is shown in the results section, this was not achieved due to the fact that the corresponding topics did not publish data every second, so that all data points were conserved when applying this method. Below, the function is presented. It takes as an input a data stream and its corresponding stream of time stamps, and outputs an array which contains the averaged data (medians) with the new time stamps (by the second).

```
std::vector<std::vector<float> >
    CleanData::calculateMedian(std::vector<float> data, std::vector<double>
    time_double){
   // initialising
   std::vector<std::vector<float> > medians;
   int startpos = 0;
   int endpos;
   int tag = 0;
   while (tag != 1){
      for (int i = 0; i < time_double.size(); i++){
         int firstint = floor(time_double[startpos] + 0.5);
         int nearestint = floor(time_double[i] + 0.5);
         // when the time stamp is rounded to the next second, start
             calculating the median of the data points until the next
             rounded second is reached
         if (nearestint != firstint && nearestint > firstint){ // Note: for
             one topic, the time stamp does not increase monotonically,
             which is why the second condition is needed. Consequently,
             until the source of this problem is identified, the data points
             corresponding to these 'temporal outliers' are deleted with
             this procedure.
            endpos = i;
            std::vector<float> partialdata;
            // access only the data points covering one second of data and
                store it to a temporary array
            for (int j = startpos; j < endpos; j++){
               partialdata.push_back(data[j]);
            }
            int length = partialdata.size();
            // calculate the median over the temporary array
            float median;
            sort(partialdata.begin(), partialdata.end());
            if (length % 2 == 0){
               median = (partialdata[length / 2 - 1] + partialdata[length /
                   2]) / 2;
            }
            else {
               median = partialdata[length / 2];
            }
            // store the median in a vector which only contains one data
                point per second (the median of all data points within one
                second)
            std::vector<float> point;
            point.push_back(median);
            point.push_back(firstint); // store the new time stamp (by the
                second) in the same array
            medians.push_back(point); // store the subarray consisting of
                median and time stamp in an array which stores the entire
                stream of medians and time stamps
            startpos = endpos;
         }
         // final iteration is different in order not to loose the last
             data point
         else if (i == (time_double.size() - 1)){
            endpos = i;
            std::vector<float> partialdata;
            for (int j = startpos; j <= endpos; j++){
               partialdata.push_back(data[j]);
```

```
                }
                int length = partialdata.size();
                float median;
                sort(partialdata.begin(), partialdata.end());
                if (length % 2 == 0){
                   median = (partialdata[length / 2 - 1] + partialdata[length /
                       2]) / 2;
                }
                else {
                   median = partialdata[length / 2];
                }
                std::vector<float> point;
                point.push_back(median);
                point.push_back(firstint);
                medians.push_back(point);
                tag = 1; // abort while loop
             }
          }
       }
       return medians;
    }
```

Writing to file

This codes allows to write data from memory to file. It can be used either for the raw data, or for the data which has been averaged to the nearest second (see section on data treatment), and also for potentially further treated data, in case the data treatment code is extended. The data is written to `.csv` files. For each topic, an individual file is created, which contains all variables contained in this topic. This choice was made because the variables contained in one topic are published at the same frequency and the same moment in time, so that their time stamps are identical. Each variable is written to a different column. The `.csv` files contain a header row with the names of the variables. The first column corresponds to the time stamp. Below, the generic structure of the code is presented. Note that since the topics are very differently structured, and since the data contained therein has very different formats, ranging from numbers over strings to booleans, the individual writers can differ quite substantially. They take as many variable arguments as the topic contains variables.

```
void Topic1Writer::writer(std::string filename, variable1_type variable1_name,
    variable2_type variable2_name, int length_of_data_stream, std::vector<double>
    time){
   std::ofstream file(filename.c_str());
   if (file.is_open() == false){ // verify that the file could be opened correctly
      std::cout << "File could not be opened" << std::endl;
      throw;
   }
   file << "Time" << ";" << "Variable1_name" << ";" << "Variable2_name" <<
       std::endl; // write the header row
   for (int i = 0; i < length_of_data_stream; i++){
      file << time[i] << ";" << variable1_name[i] << ";" << variable2_name[i] <<
          std::endl; // write a line of data at each time point
   }
}
```

In cases where the data needs to be converted to appropriate units (such as from Kelvin to degree Celsius), this is also done in the corresponding writer function.

r datatreatment script

In this section, the methods used for exploring the data, as well as the insights gained from the data exploration, are presented.

Methods

The explorative data analysis was perfored with R, and was conducted by using data which had been averaged to the nearest second by taking the median (see the section on data treatment), which had been written to `.csv` files by using the code presented in the previous section. The following analytical tools were used:

1. *Time series:* The aim of producing time series graphs was to gain a first impression of what the data from the different sensors looks like, and to identify potential issues which need to be resolved.

2. *Box plots:* The aim of producing box plots was to gain an impression of how the values produced by the different sensors are distributed.

3. *QQ plots:* The aim of producing QQ plots was not so much to identify the degree of normality of the variables' distributions, but rather to gain an improved insight into their distribution. The degree of normality is irrelevant for most data, since it is not supposed to be distributed normally, especially given the short duration of the available data streams (the maximum duration is 30 minutes).

4. *Scatter plots and correlation analyses:* For the variables where this promised to produce potentially interesting insights, scatter plots were produced and correlation analyses were run. This was done mainly for variables which measure the same phenomenon (such as temperature) but are produced by different sensors, and variables were assumed to be co–dependent, such as for instance the speed and acceleration of the rover and its energy consumption.

Lastly, since the analysis of the time series plots has shown that in the case of some data streams, there appear to be measurement errors which lead to unrealistic outliers, which distort analyses such as QQ plots or scatter plots and correlation analyses, a function was created to eliminate such outliers, and which is presented below:

```
remove_outliers <- function(x, na.rm = TRUE, ...) {
   qnt <- quantile(x, probs=c(.25, .75), na.rm = na.rm, ...)
   H <- 1.5 * IQR(x, na.rm = na.rm)
   y <- x
   y[x < (qnt[1] - H)] <- NA
   y[x > (qnt[2] + H)] <- NA
   y
}
```