SIS Project Documentation – v2

The following gives some useful hints and technical details to help you with implementing the project.

Code structure

C/C++

As far as you are concerned, you will be working in C. However, the material provided is actually implemented in C++, but all common C features you have used so far in the labs still work the same way.

You will mostly work in the robot's controller folder `controllers/controller` in which the main source file `controller.cpp` is located. You can edit this file and implement your project in this file directly, but you can also use additional header files in which you can implement specific features (e.g., `kalman.hpp` for the Kalman filter, `odometry.hpp` to compute the odometry, etc...).

Important note on header files*: You are most likely used to implement functions in a `.c/.cpp` file and declare the functions you implemented in the corresponding `.h/.hpp` file, which can then be included in your main source file. To simplify the compilation process for the project, we suggest you to use only one source file (`controller.cpp`), and use header files to implement the different features of the project. Therefore, the header files will look like `.cpp` files since they contain implemented code, but can still be included in your main `controller.cpp` with the usual `#include` directive. This will let you create as many new header files as you need, without worrying about updating the project's Makefile already provided for you.*

We already provide a basic implementation in `controller.cpp` (the main file for the project), as well as (mostly) empty header files in which you can implement the different parts of your project. This should provide you with a basis to get acquainted with the project material and help you structure your work, but **you do not have to follow the suggested structure exactly**.

Here are the source/header files you'll find in the `controllers/controller` folder:

- braitenberg.hpp
- **controller.cpp** (main source file)
- FSM.hpp
- kalman.hpp
- odometry.hpp
- serial.hpp
- signal_analysis.hpp

Scripts

Finally, you can implement Matlab or Python script to analyze the data generated by the robot and store these scripts in the `scripts/matlab` or `scripts/python` folder respectively.

Robot API

An Application Programming Interface (API) is provided to interact with the robot. It can be found in the `libraries/pioneer_interface/pioneer_interface.hpp` file. The provided controller `controllers/controller.cpp` demonstrates how to use it.

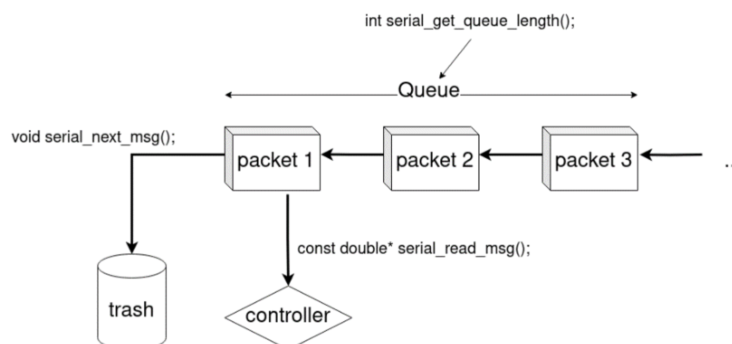
Following functions are available (they all contain an associated docstring, which further details their functionalities), implemented as a class:

```
class Pioneer {
  void init();
  double get_time();
  double get_timestep();
  int step();
  void set_motors_velocity(double left, double right);
  double* get_encoders();
  double* get_proximity();
  double get_light_intensity();
  double* get_imu();
  int serial_get_queue_length();
  const double* serial_read_msg();
  void serial_next_msg();
  double serial_get_signal_strength();
}
```

Also, some relevant dimensions and properties are provided as a structure:

```
struct PioneerInfo{
  double length = 0.508;  // length of the robot [m]
  double width  = 0.497;  // width of the robot [m]
  double height = 0.277;  // height of the robot [m]
  double weight = 12.000; // weight of the robot [kg]
  double wheel_radius = 0.11; // Radius of a wheel [m]
  double axis_length  = 0.4;  // Distance between front/back pair of
wheels[m]
} typedef PioneerInfo;
```

The serial communication works as follows:



Data logging

You will need your robot to log data during its mission. For this, we provide you a set of functions to log data into `csv` files easily. They are implemented in the file `libraries/utils/log_data.hpp`. You can look at the provided controller code to see how to use these functions. The data will be logged in the `controllers/controller/data` folder as `csv` files.

There are two main functions you need to use (here in the case you want to log each time three values per rows):

- `num_cols = init_csv("my_file_name.csv", "col1, col2, col3,");`
- `success = log_csv("my_file_name.csv", num_cols, value1, value2, value3);`

The `init_csv()` function must be called only once, as it opens a new csv file in the data folder next to your controller. It returns the number of columns you specified. **Note the trailing comma after the last column, do not forget to include it!** This function can be called multiple times with different file names to open several different files in parallel.

The `log_csv()` function allows you to log a new line into the designated csv file. You must provide as many arguments as what is specified by `num_cols` here. It returns true upon successful completion or false otherwise (that can happen if you specify a file that does not exist for instance). If the values you want to store have a different type than double, **make sure to cast the values you provide to double when passing them as argument to the function!!** Otherwise, the logged data would not be interpreted correctly.

Before your controller exits, call the `close_csv()` function to close all opened csv files (already called at the end of the controller file).

Kiss FFT library

We provide you with a C library for the FFT algorithm. The example below demonstrates how to use it:

```
#include <math.h>
#include "kiss_fft/kiss_fft.h" // FFT library

#define SIGNAL_LENGTH 1024 // length of the signal to be analyzed

/**
 * @brief KISS FFT USAGE EXAMPLE
 * @param signal_data pointer to the array containing the signal to be analyzed
 */
void kiss_fft_demo(double* signal_data){

    /* fft preparation */
    kiss_fft_cfg cfg = kiss_fft_alloc( SIGNAL_LENGTH, 0, NULL, NULL );

     /* fft variables */
    kiss_fft_cpx cx_in[SIGNAL_LENGTH],  // input signal (time domain)
                 cx_out[SIGNAL_LENGTH]; // output signal (frequency domain)

    // prepare the input (add signal in the 'in' structure of the kiss_fft)
    for (int n=0; n<SIGNAL_LENGTH; n++) {
```

```
      cx_in[n].r = signal_data[n]; // the real part of the signal is the data
      cx_in[n].i = 0.; // set the imaginary part to zero
    }

    // run the fft (the fourier transform is stored in the 'out' structure of the
kiss_fft)
    kiss_fft( cfg , cx_in , cx_out );

    // compute the magnitude of the complex numbers
    double mag[SIGNAL_LENGTH];
    for (int n=0; n<SIGNAL_LENGTH; n++) {
      mag[n] = sqrt(cx_out[n].r*cx_out[n].r + cx_out[n].i*cx_out[n].i);
    }

    // TODO: do something with the magnitude, real, imaginary part of the fft
    // ...

    // free fft memory once done with it
    free(cfg);
}
```

Matrix operations

The following types are defined to implement matrices in C++ (for the Kalman filter typically):

```
#define DIM 3                                          // State dimension (assume 3)

typedef Eigen::Matrix<double,DIM,DIM>   Mat;           // DIMxDIM matrix
typedef Eigen::Matrix<double, -1, -1>   MatX;          // Arbitrary size matrix
typedef Eigen::Matrix<double,DIM,  1>   Vec;           // DIMx1 column vector
typedef Eigen::Matrix<double, -1,  1>   VecX;          // Arbitrary size column vector
```

They can be used as follows:

```
Mat m = Mat::Zero(); // zero matrix
m = Mat::Identity(); // identity matrix

MatX mx(3,2); // arbitrary size matrix (uninitialized)
mx << 1, 2,
      3, 4,
      5, 6; // fill it to be a 3x2 matrix

m.inverse(); // inverse of a matrix
m.transpose(); // transpose of a matrix
m * mx; // matrix multiplication
m + m;  // matrix addition
m - m;  // matrix substraction
mx = 2*mx; // scalar multiplication
```

Kalman Filter

As the implementation of the Kalman filter can be challenging in some cases, we review here some details concerning the prediction step using either an IMU or wheel encoders for a state $\boldsymbol{\mu} = [x\ y\ \theta]^T$, describing the 2D pose of a differential drive robot.

1. IMU prediction:
The IMU measures the acceleration and angular rate for each axis, hence it provides the following measurement (acting as a control input for the prediction step):

$$\boldsymbol{u}_{IMU} = \left[a_x\ a_y\ a_z\ \omega_x\ \omega_y\ \omega_z\right]^T$$

The acceleration has the standard deviation σ_a and the angular rate the standard deviation σ_ω (identical for all axes). Since we work with a 2D pose, we only consider the x and y axis for the acceleration, and the z axis for the angular rate. We need to integrate the acceleration into a velocity, hence we have $v_x = a_x dt$ and $v_y = a_y dt$, with associated standard deviation $\sigma_v = \sigma_a dt$. You will have to keep track of these velocities and keep adding new acceleration measurements to them.
Finally, the input (the integrated accelerations and angular rate) and its associated noise matrix are defined as:

$$\boldsymbol{u} = \left[v_x\ v_y\ \omega_z\right]^T$$
$$\boldsymbol{\Sigma_u} = \begin{bmatrix} \sigma_v^2 & 0 & 0 \\ 0 & \sigma_v^2 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix}$$

Now we can write the matrices for the system model (simple integrator) expressed in the **global frame**:

$$\boldsymbol{A} = \boldsymbol{I}^{(3\times3)}, \qquad \boldsymbol{B}_t = \boldsymbol{T_t}\boldsymbol{I}^{(3\times3)}dt$$

where $\boldsymbol{I}^{(3\times3)}$ is the 3x3 identity matrix and $\boldsymbol{T_t}$ is the rotation matrix that converts the input vector from the local frame to the global frame using the current heading θ_t and is defined as (notice it leaves ω_z untouched, which is what we want):

$$\boldsymbol{T_t} = \begin{bmatrix} cos\theta_t & -sin\theta_t & 0 \\ sin\theta_t & cos\theta_t & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The model noise matrix is finally:

$$\boldsymbol{R}_t = \boldsymbol{B}_t\boldsymbol{\Sigma_u}\boldsymbol{B}_t^T$$

Finally the prediction step of the Kalman Filter can be computed as follows:

$$\boldsymbol{\mu}_t = \boldsymbol{A}\boldsymbol{\mu}_{t-1} + \boldsymbol{B}_t\boldsymbol{u}_t$$
$$\boldsymbol{\Sigma}_t = \boldsymbol{A}\boldsymbol{\Sigma}_{t-1}\boldsymbol{A}^{\boldsymbol{T}} + \boldsymbol{R_t}$$

2. Encoder prediction:
   The odometry allows to directly measure the robot's velocity from the wheel rotations. We therefore have the following input:

$$\boldsymbol{u} = \left[v_x \; v_y \; \omega_z\right]^T$$

   Note that the robot behaves as a differential drive robot, so we can set $v_y = 0$, and therefore has no associated uncertainty. It is not as easy to estimate the standard deviation of the wheel odometry. One can start with an empirical guess and see how the filter performs. You can try to set $\sigma_v = 0.05$ m/s for instance and see how the filter performs. Same goes for the angular rate uncertainty σ_ω. You can assume it to be less accurate than the heading measurements from the gyroscope (wheels slipping, uneven terrain, etc.), so you can assume $\sigma_\omega^{odo} = 10\sigma_\omega^{imu}$. You may want to try different values. Finally, the process covariance matrix is:

$$\boldsymbol{\Sigma_u} = \begin{bmatrix} \sigma_v^2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix}$$

   From there, the exact same derivations follow as for the IMU prediction step.

While the presented prediction steps allow us to use the standard Kalman Filter formulation, it is not completely correct because of the non-linearity introduced by the heading angle θ_t. While this may suffice for the purpose of this project, one should actually employ the **Extended Kalman Filter**. We briefly go through its general implementation in the context of this project, applicable to both the IMU or encoder prediction.

The motion model is (actually same as before but written explicitly this time):

$$\boldsymbol{x}_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t) = \begin{bmatrix} x_{t-1} + \left(\cos\theta_{t-1}\, v_{x,t} - \sin\theta_{t-1}\, v_{y,t}\right)dt \\ y_{t-1} + \left(\sin\theta_{t-1}\, v_{x,t} + \cos\theta_{t-1}\, v_{y,t}\right)dt \\ \theta_{t-1} + \omega_{z,t}\, dt \end{bmatrix}$$

With the same velocity input and associated covariance:

$$\boldsymbol{u}_t = \left[v_{x,t} \; v_{y,t} \; \omega_{z,t}\right]^T, \qquad \boldsymbol{\Sigma_u} = \begin{bmatrix} \sigma_v^2 & 0 & 0 \\ 0 & \sigma_v^2 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix}$$

To perform error propagation, the system needs to be linearized using its first order Taylor expansion. The process matrix therefore becomes

$$\boldsymbol{F}_t(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \triangleq \frac{\partial f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t)}{\partial \boldsymbol{x}_{t-1}} = \begin{bmatrix} 1 & 0 & \left(-\sin\theta_{t-1}\, v_{x,t} - \cos\theta_{t-1}\, v_{y,t}\right)dt \\ 0 & 1 & \left(\cos\theta_{t-1}\, v_{x,t} - \sin\theta_{t-1}\, v_{y,t}\right)dt \\ 0 & 0 & 1 \end{bmatrix}$$

The associated process noise is computed from the velocity standard deviations, similarly to the derivations for the standard Kalman filter:

$$\boldsymbol{R}_t \triangleq \boldsymbol{G}_t \boldsymbol{\Sigma}_{\boldsymbol{u}} \boldsymbol{G}_t^T = \boldsymbol{T}_{t-1} \begin{bmatrix} \sigma_v^2 dt^2 & 0 & 0 \\ 0 & \sigma_v^2 dt^2 & 0 \\ 0 & 0 & \sigma_\omega^2 dt^2 \end{bmatrix} \boldsymbol{T}_{t-1}^T, \quad \text{where} \quad \boldsymbol{G}_t \triangleq \frac{\partial f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t)}{\partial \boldsymbol{u}_t} = \boldsymbol{T}_{t-1} dt$$

where $\boldsymbol{T}_{t-1}$ is the rotation matrix that implements the conversion from the local frame to the global frame using the current heading θ_{t-1}, as defined earlier.

Regarding the update step, for instance in the case where the position in x and y is measured, with $\boldsymbol{z}_t = [x_m \ y_m]^T$, the associated measurement matrix would be

$$\boldsymbol{H}_t(\boldsymbol{x}_t) \triangleq \frac{\partial h(\boldsymbol{x}_t)}{\partial \boldsymbol{x}_t} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

where $h(\boldsymbol{x}_t)$ maps the current state to the expected measurement, thus in this case $h(\boldsymbol{x}_t) = [x_t \ y_t]^T$. Note that the update step is essentially implemented in the same way, whether one uses the standard or extended version of the filter.

You are free to choose/combine/adapt/compare any of the previous options, based on what you feel is more relevant and/or would yield better results. Make sure to provide all the relevant details in your report and justify your choices.

VSCode configuration

To ensure code completion works in VSCode, ensure the file `.vscode/c_cpp_properties.json` is configured appropriately (check in particular that you have the paths indicated in bold):

```
{
    "configurations": [
        {
            "name": "Linux",
            "includePath": [
                "${workspaceFolder}/**",
                "/usr/local/webots/include/controller/c/**",
                "/usr/local/webots/include/controller/cpp/**",
                "/usr/include/eigen3/**"
            ],
            "defines": [],
            "compilerPath": "/usr/bin/gcc",
            "cStandard": "c17",
            "cppStandard": "c++14",
            "intelliSenseMode": "linux-gcc-x64"
        }
    ],
    "version": 4
}
```