Visual-Based Navigation
Solution Exercise Sheet 1
Topic: Introduction to SLAM and Lie Groups

Kamysek, Josef-Maria

April 29, 2021

Part 1: setup, git, merge-requests, cmake, gcc

```
# Snippet 1
set(CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake_modules/" ${
    CMAKE_MODULE_PATH})

# Snippet 2
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

# Snippet 3
set(CMAKE_CXX_FLAGS_DEBUG "-O0 -g -DEIGEN_INITIALIZE_MATRICES_BY_NAN")
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "-O3 -DNDEBUG -g -
    DEIGEN_INITIALIZE_MATRICES_BY_NAN")
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -DNDEBUG")
SET(CMAKE_CXX_FLAGS " -ftemplate -backtrace -limit=0 -Wall -Wextra ${
    EXTRA_WARNING_FLAGS} -march=${CXX_MARCH} ${CMAKE_CXX_FLAGS}")

# Snippet 4
add_executable(calibration src/calibration.cpp)
target_link_libraries(calibration Ceres::ceres pangolin TBB)
```

Snippet 1

With this CMake statement we provide additional modules to the current project. In this case, the additional modules are specified in the /cmake_modules folder, e.g. Eigen.
(Source: https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html)

Snippet 2

The first statement is used to specify the CXX_STANDARD property on all targets. This means it defines which compiler and linker flags should be added to the target to build it with a specific C++ standard. In this case it is C++ 14.
(Source: https://cmake.org/cmake/help/latest/variable/CMAKE_CXX_STANDARD.html)

The second statement is used to initialize the CXX_STANDARD_REQUIRED property on all targets. This means that it ensures that CMake does not fall back to a version that the compiler supports if it cannot find the specified version.
(Source: https://cmake.org/cmake/help/latest/variable/CMAKE_CXX_STANDARD_REQUIRED.html)

The third statement is used to change the flag that sets the language standard. In other words, to ensure broader compatibility across compilers, projects should disable the C++ extensions.
(Source: https://cmake.org/cmake/help/latest/variable/CMAKE_CXX_EXTENSIONS.html)
(Source: https://crascit.com/2015/03/28/enabling-cxx11-in-cmake/)

Snippet 3

CMake creates the following variables by default: None, Debug, Release, RelWithDebInfo, MinSizeRel. These default compilation flags can also be changed, which is precisely what is shown in

these statements. One can set these compilation flags in the variable CMAKE_BUILD_TYPE. As indicated in the provided source, the default values for these flags change with different compilers. If CMake does not know your compiler, the contents will be empty.
(Source: https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/Useful-Variables)

Snippet 4

The first statement adds an executable target called calibration. It is build from the source file /src/calibration.cpp.
(Source: https://cmake.org/cmake/help/latest/command/add_executable.html)

The second statement specifies which files / libraries should be used to link a given target and its dependents.
(Source: https://cmake.org/cmake/help/latest/command/target_link_libraries.html)

Part 2: SO(3) and SE(3) Lie groups

According to the hint we should use the Taylor expansion and put the odd and even items together. We know the following:

1. The identity $\hat{w}^3 = -(w^T w) \cdot \hat{w}$ is used to split the $\hat{w}$ terms into even and odd.
2. Using the fact $\theta^2 \equiv w^T w$ and the identity from above we can split the $(\hat{w})$ terms into even and odd the following way: $\hat{w}^{2n+2} = (-1)^n \theta^{2n} \hat{w}^2$ and $\hat{w}^{2n+1} = (-1)^n \theta^{2n} \hat{w}$.

$$
\begin{aligned}
\sum_{n=0}^{\infty} \frac{1}{(n+1)!} (\hat{w})^n &= I + \frac{\hat{w}}{2!} + \frac{\hat{w}^2}{3!} + \frac{\hat{w}^3}{4!} + \dots \\
&= I + \sum_{n=0}^{\infty} \Big(\underbrace{\frac{\hat{w}^{(2n+1)}}{(2n+2)!}}_{\frac{\hat{w}}{2!} + \frac{\hat{w}^3}{4!} + \dots} + \underbrace{\frac{\hat{w}^{(2n+2)}}{(2n+3)!}}_{\frac{\hat{w}^2}{3!} + \frac{\hat{w}^4}{5!} + \dots} \\
&= I + \sum_{n=0}^{\infty} \Big(\underbrace{\frac{(-1)^n \theta^{2n}}{(2n+2)!}}_{\frac{1}{2!} + \frac{\theta^2}{4!} + \dots} \cdot \hat{w} + \underbrace{\frac{(-1)^n \theta^{2n}}{(2n+3)!}}_{\frac{1}{3!} + \frac{\theta^2}{5!} + \dots} \cdot \hat{w}^2 \\
&= I + (\frac{1}{2!} + \frac{\theta^2}{4!} + \dots) \cdot \hat{w} + (\frac{1}{3!} + \frac{\theta^2}{5!} + \dots) \cdot \hat{w}^2 \\
&= I + (\frac{1 - \cos\theta}{\theta^2}) \cdot \hat{w} + (\frac{\theta - \sin\theta}{\theta^3}) \cdot \hat{w}^2 \\
&= J
\end{aligned}
$$

Second part of this exercise can be found in include/visnav/ex1.h.

1 Part 3: What is SLAM?

Most of these questions are already answered very nicely and compactly in the paper, so the relevant information has been extracted. In addition I added the source from where I extracted it.

1. Why would a SLAM system need a map?

   The need to use a map of the environment is twofold:

   (a) First, the map is often required to support other tasks; for instance, a map can inform path planning or provide an intuitive visualization for a human operator.

   (b) Second, the map allows limiting the error committed in estimating the state of the robot. In the absence of a map, dead-reckoning would quickly drift over time; on the other hand, using a map, e.g., a set of distinguishable landmarks, the robot can "reset" its localization error by re-visiting known areas (so-called loop closure)

   (Source: Information extracted from the Chapter: Introduction of the paper)

2. How can we apply SLAM technology into real-wold applications?

   The architecture of a SLAM system includes two main components: the front-end and the back-end. The front-end abstracts sensor data into models that are amenable for estimation, while the back-end performs inference on the abstracted data produced by the front-end.

   (Source: Information extracted from the Beginning of the chapter: Anatomy of a modern slam system)

3. Describe the history of SLAM:

   The problem of simultaneous localization and mapping has made great strides in the last 30 years. Back then, it all started with the presentation of a seminal paper by Lu and Milios, followed by the work of Gutmann and Konolige. Since then, several approaches have improved efficiency and robustness. Along this path, some important questions have been answered, mainly by the development of new sensors and new computational tools.

   Modern SLAM systems mainly use maximum-a-posteriori estimation in the back-end, performed on a preprocessed version of the sensor data. The front-end itself extracts relevant features from the sensor data and associates the measurement with a particular landmark.

   Since SLAM systems can be vulnerable in many ways, the failure can be algorithmic or hardware related. Much research has been done on these issues. These robustness issues are usually addressed in the front-end, which is tasked with making correct data associations. However, a lot of work still needs to be done here, e.g. current SLAM solvers still have problems with the presence of outliers.

   Another issue where much work and progress still needs to be made is scalability. In practice, computation time and memory requirements are limited by the resources of the robot. Therefore, it is important to design SLAM methods whose computational and memory complexity remains limited. Successive linearization methods based on direct linear solvers may imply increasing memory consumption. Progress has been made in this area by working on, for example, sparsification methods, out-of-coure and multi-robot methods. However, there are still open problems such as map representation and learning, forgetting and remembering in long-term mapping.

   Much work has also been done in the topic of how to represent geometry. For the 2D case, this has been addressed by the IEEE RAS Map Data Representation Working Group. For the 3D case, on the other hand, there are a variety of approaches, such as landmark-based sparse representations, low-level raw dense representations, or boundary and spatial-partitioning dense representations. Of course, there are many more and much research is being done in this area, however, these metric representations are still largely unexplored.

   In recent years, there has also been a great deal of progress in establishing performance guarantees for SLAM algorithms. Despite the progress in these areas, questions remain about generality, guarantees, and verification.

   The approaches described so far describe SLAM as an estimation problem that is performed passively. Controlling the motion of a robot with the goal of minimizing the uncertainty of its map representation and localization is referred to as active SLAM. In order to use active SLAM in real applications, problems such as fast and accurate predictions of future states or performance guarantees must be solved.

   As can be seen, much research has been done in the past and there is still much to be done in the future.

   (Source: Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age, https://arxiv.org/pdf/1606.05830.pdf)