Exercise Sheet 1

Anselm Coogan

<2019-10-22 Tue>

1 Exercise 1: What is SLAM

1.1 Why would a SLAM system need a map?

- can inform path planning
- provides intuitive visualization for humans

1.2 How can we apply SLAM technology into real-world applications?

- we need SLAM in applications where a prior map is not available but the robot needs to be aware of its surroundings and relative position
- SLAM is useful for military applications, agriculture, autonmous driving, cleaning/service robots
- to apply SLAM you need a set of sensors (LIDAR, depth, RGB, DVS, etc.) which are the processed by the system's fron-end. This front-end interprets the sensor information (fusion, segmentation, filtering, feature tracking, etc.). It is also responsible for loop closure. The processed sensory data is then sent to the back-end part of the system that usually performs MAP estimation resulting in a SLAM estimate.

1.3 Describe the history of SLAM

1.3.1 The classical age (1984-2004)

- introduction of probabilistic SLAM formulations: MAP, Extended Kalman Filter, Rao-Blackwellised Particle Filters
- decoupling of main challenges: efficiency and robust data association

1.3.2 Algorithmic analysis age (2004-2015)

- study of SLAM fundamentals: observability, convergence, and consistency
- development of main open source libraries for SLAM (Ceres, GTSAM, g2o, iSAM, SLAM++)
- using sparsity to create efficient SLAM solvers

1.3.3 Robust perception age (2015-present)

- SLAM "solved" for specific conditions (e.g. using LIDAR to map an indoor 2D non-dynamic environment with a slow-moving robot)
- actively researched/unsolved problems:
  - robust performance (in the face of outliers, changing maps, non-rigid world, degredation of sensors)
  - high-level understanding (semantic maps, dynamic vs static parts)
  - resource awareness (dealing with influx in CPU stress, taking advantage of idle processor)
  - task-driven perception (filter non-relevant perceptual information, produce adaptive map representations)

2 Exercise 2: git, cmake, gcc, merge-requests

- Prepends the path where cmake will look for modules with the **include** and **find_package** commands with the current dir (.) and the subdirectory **cmake_modules**
- CMAKE_CXX_STANDARD
  - sets the optional property for the build to use the appropriate compiler and linker flags for the C++11 standard to use the appropriate flags for C++ 11. If a newer compiler/linker is available it will be used (i.e. 14 or 17). If only older versions are available, it will fallback to those.
  - Disables the decay to older C++ standards. If C++11 or newer is not available, compilation will fail.

  - Forces the build to use standard C++11 dialect and not an extended variant of this (such as gnu++11). Having the property set to ON (default) may result in cross-platform build bugs

- CMAKE_CXX_FLAGS Sets global flags and specific flags for specific build targets (specified through e.g. `cmake -DCMAKE_TYPE=DEBUG ..`)
  - O0: compiles without optimization
  - O3: compiles with full optimization
  - g: compiles with GDB debugging enabled
  - EIGEN_INITIALIZE_MATRICES_BY_NAN: initializes matrices with NaN values, makes it easier to find uninitialized matrices when debugging
  - NDEBUG: disables assert statements in code
  - ftemplate-backtrace-limit=0: disables the limit of instantiation notes for a single error (default is 10)
  - Wall: enables all warnings during build time
  - EXTRA_WARNING_FLAGS: set of flags defined previously to account for different compilers (gcc and clang) e.g.: Wsign-compare (warning generated when signed and unsigned values are compared), Wno-exceptions (disables clang enabled exception warnings)
  - march: is set to native so that compiler generates instructions for the build system's CPU

3 Exercise 3: SO(3) and SE(3) Lie groups