ROSLab: sharing ROS code interactively with Docker and JupyterLab

Enric Cervera[1], Angel P. del Pobil[1,2]

[1] Department of Computer Science and Engineering, Jaume-I University, 12071 Castelló de la Plana, Spain
[2] Department of Interaction Science, Sungkyunkwan University, Seoul 03063, South Korea

The success of the Robot Operating System (ROS) and the advance of the Open Source ideas have radically changed, for the better, the experience of sharing software among people in the robotics community. Yet the lack of a suitable workflow for continuous integration and test verification in robotics still represents a significant obstacle for developing software that can be run by independent users for testing or reusing purposes.

A typical situation is that a developer produces a new ROS package and shares it through a public source repository, which other users can download and execute. However, the execution environment matters and may be incompatible due to users running different ROS distributions, incompatible versions of package dependencies, or third party libraries.

Another obstacle is the lack of suitable documentation for running the software. Most of the shared code is barely documented as it has been developed primarily for internal use within a research group. Developers are experts in robotics but not necessarily in software engineering. Some configuration or execution steps may not be so well explained in the documentation because of the familiarity with the software.

Consequently, there is a need for a complete, unambiguous, runnable environment for testing the publicly available ROS code, which guarantees that it functions correctly no matter what the particular configuration of the host is. Such an environment should be compatible with the typical workflow of ROS development for avoiding any extra burden in the process.

Recently, the use of Docker has proven to be a useful framework for enabling better, repeatable and reproducible environmental setups [1]. Docker is an implementation of Linux containers, an operating-system-level virtualization method for running isolated systems on a control host. It is similar to a Virtual Machine, with less overheads.

While Docker is steadily gaining popularity, it is not yet known by many users in the robotics community. In this paper, we aim to lower the barriers for adopting Docker by introducing ROSLab, a framework combining Docker and JupyterLab for turning a software code repository into a reproducible, runnable software package. ROSLab automatically generates a Docker image from a simple specification from the developer's environment.

The motivation for using JupyterLab is the need to integrate the documentation with the software, since there is a tendency for experimental code not to be accompanied by much documentation. JupyterLab is based on notebooks, interactive documents containing executable code and narrative text, thus allowing both the developer and the user to share explicitly the commands for running the software. In previous research, we have observed that incomplete or ambiguous documentation is a significant obstacle in the reuse of robotics code [2].

Figure 1 illustrates the code sharing process, and how it is extended with ROSLab. The outcome after the development of a novel research experiment programmed with ROS is typically a public

code repository (e.g. github), which contains one or several ROS packages, and a documentation file (README) with a description of the code, and some instructions for building and running the code.

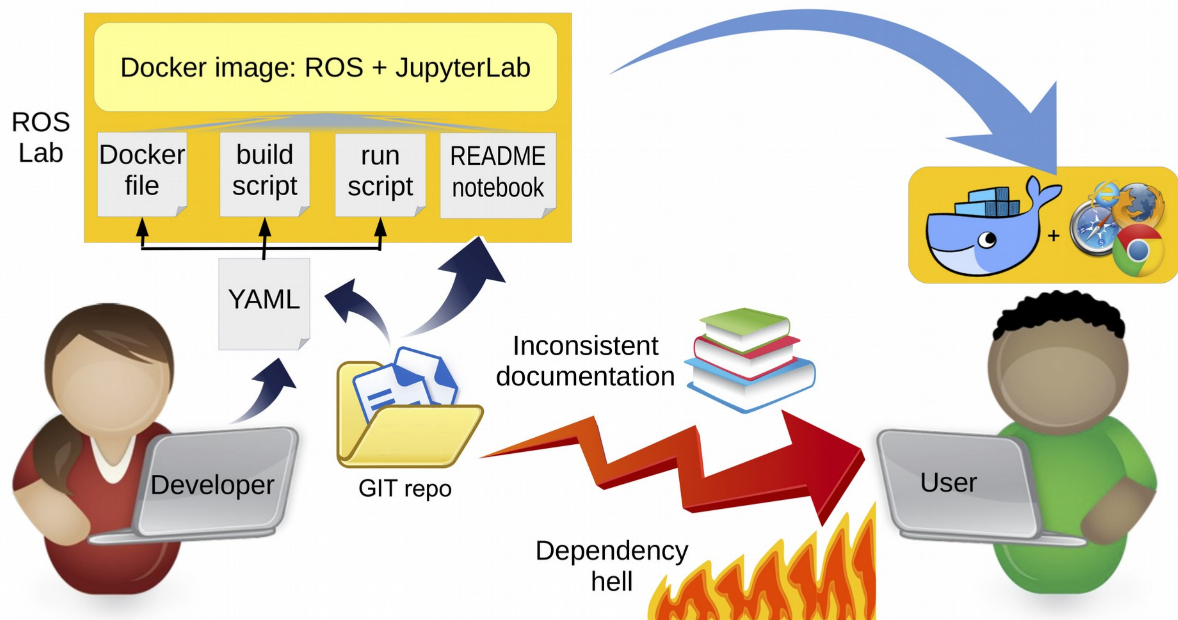

Figure 1: Using ROSLab for sharing a code repository circumvents the trouble of installing and running the software on different system configurations.

A new user interested in testing this new development faces two main difficulties: first, the ROS distribution in her host may be different than the developer's, be it newer (with deprecated packages) or older (lacking functionality), causing a situation where the package and library dependencies are liable to break down --the so-called "dependency hell"; second, the documentation may be incomplete or inconsistent: instructions about third-party libraries could be missing, or some configuration and execution steps have not been included.

What ROSLab does is to automatize the creation of Docker images: instead of writing a complete Dockerfile, the developer only needs to specify the necessary components for running the image, namely the ROS distribution, build method, and library dependencies. Those components are described in a YAML file, which is processed to generate the Dockerfile, a script for building the image, and another script for running it. Additionally, ROSLab processes the README file of the git repository for producing a JupyterLab notebook, where the command snippets are automatically separated into executable cells.

This process for creating Docker images is generic and can be extended to other software frameworks, but we have focused initially on ROS because of its widespread use in robotics. Nevertheless, the destination user only needs to install Docker and a web browser for running the software: no local ROS installation is necessary; yet if it exists, it will not interfere with the ROSLab Docker image.

This is a simple example of a YAML file written for the code repository of a paper published in ICRA 2018 [3], publicly available at https://github.com/ICRA-2018/nanomap_ros:

```
name: nanomap_ros
distro: kinetic
```

```
build: catkin_make

packages:
  - libeigen3-dev
  - ros-kinetic-cv-bridge
  - ros-kinetic-image-transport
  - liborocos-kdl-dev
  - ros-kinetic-tf2-sensor-msgs
```

For processing the YAML file, Docker must be installed in the host, and the ROSLab processing step is executed with the command

```
docker run --rm -v <REPOSITORY_FOLDER>:/project:rw roslab/create
```

where REPOSITORY_FOLDER is the full path of the local folder containing the git repository, where the YAML file is stored. The command produces these files in the same directory:

- `Dockerfile`: the full description of the Docker image, based on the requested ROS distribution, with the instructions for installing all the third-party dependencies, and building the repository code.
- `docker_build.sh`: a script file which invokes Docker for building the image.
- `docker_run.sh`: a script file which invokes Docker for running the image, and launching the JupyterLab server.

Next, the build script is run in a terminal, and the actual Docker image is built. Finally, the run script is run and the image is executed, with a JupyterLab server launched in the local host, to which the user can connect by opening this URL in the browser:

```
http://localhost:8888/lab/tree/README.ipynb
```

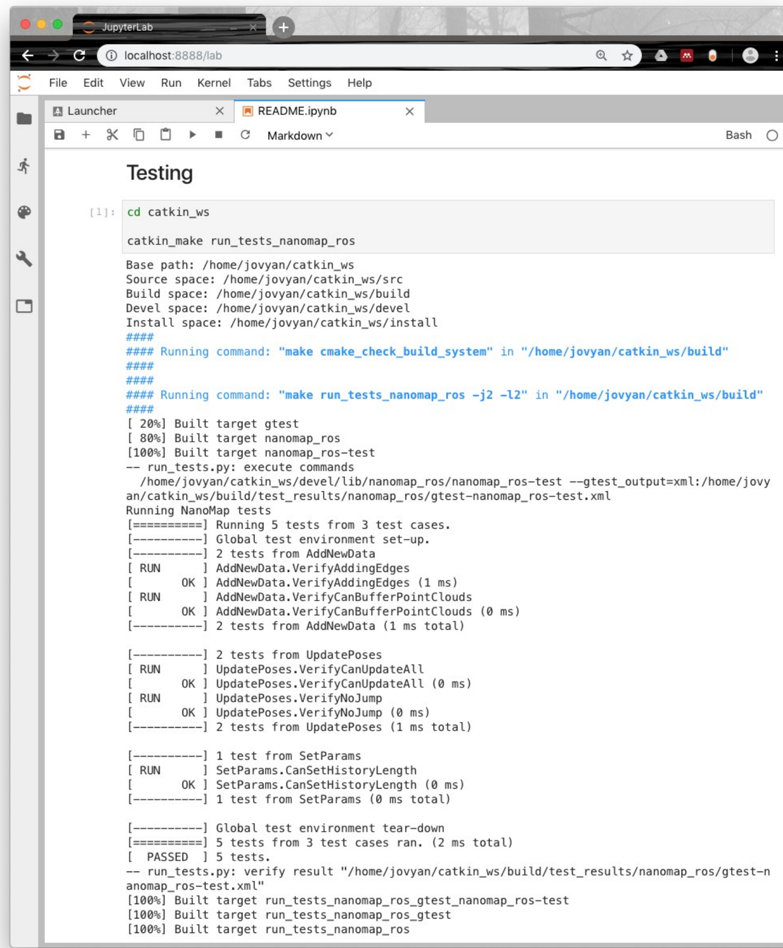

Figure 2: Execution of testing command in the README notebook of the nanomap_ros repository.

Figure 2 depicts the browser window after executing a testing command in the README notebook of the processed repository. It is worth noting that the “cd” command has to be added in order to change to the correct working directory, or the command produces an execution error.

A more complex example from another ICRA paper [4] includes the use of a GPU-accelerated runtime environment for 3D applications, the installation of packages from source, and mounting a host folder for accessing dataset files. It is available at https://github.com/ICRA-2018/VINS-Mono:

```
name: vins-mono
distro: kinetic
build: catkin_make
runtime: nvidia

volume:
  - host_path: /Data/EuRoC_MAV_Dataset
    container_path: /EuRoC_MAV_Dataset
    options: ro

packages:
  - ros-kinetic-cv-bridge
  - ros-kinetic-tf
  - ros-kinetic-message-filters
  - ros-kinetic-image-transport
```

```
source:
  - name: ceres-solver
   repo: https://github.com/ceres-solver/ceres-
solver.git
   depends:
       - libgoogle-glog-dev
       - libatlas-base-dev
       - libeigen3-dev
       - libsuitesparse-dev
   build: cmake
```

The information about the dependencies has been obtained from the README file in the repository, and the home page of the third-party library Ceres.

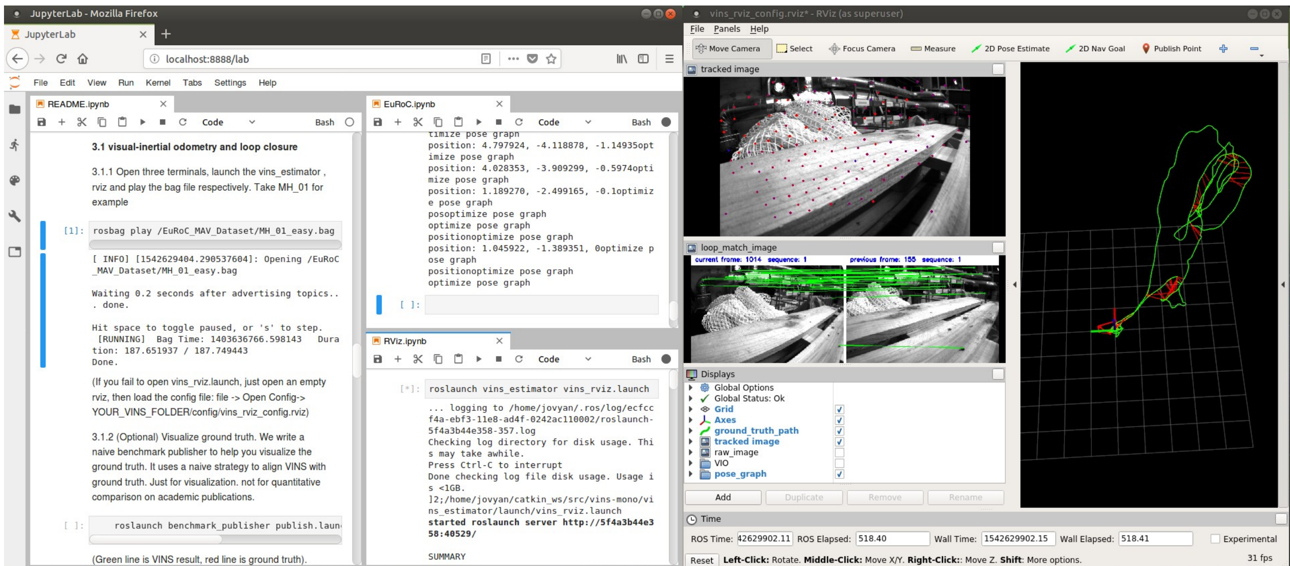

Figure 3: Running the VINS-Mono example.

The outcome of the execution is shown in Figure 3, which depicts the browser window (left) and the RViz visualization tool (right), which has been launched by one of the commands in the notebook.

In the last example, based also on an ICRA conference paper [5], the visualization tool is not Rviz but a custom application, demonstrating that ROSLab does not interfere with the package workflow. It is available at https://github.com/ICRA-2018/fast_change_detection and the YAML file is:

```
name: fast-change-detection
distro: kinetic
build: catkin_build
runtime: nvidia

packages:
  - libeigen3-dev
  - libboost-all-dev
  - qtbase5-dev
  - libglew-dev
  - libopencv-dev

source:
  - name: glow
    repo: https://github.com/jbehley/glow.git
    depends:
    build: catkin_build
```

and the output of the example is depicted in Figure 4: to the left, the notebook with the commands for running the example, and to the right, the 3D visualization of the execution.

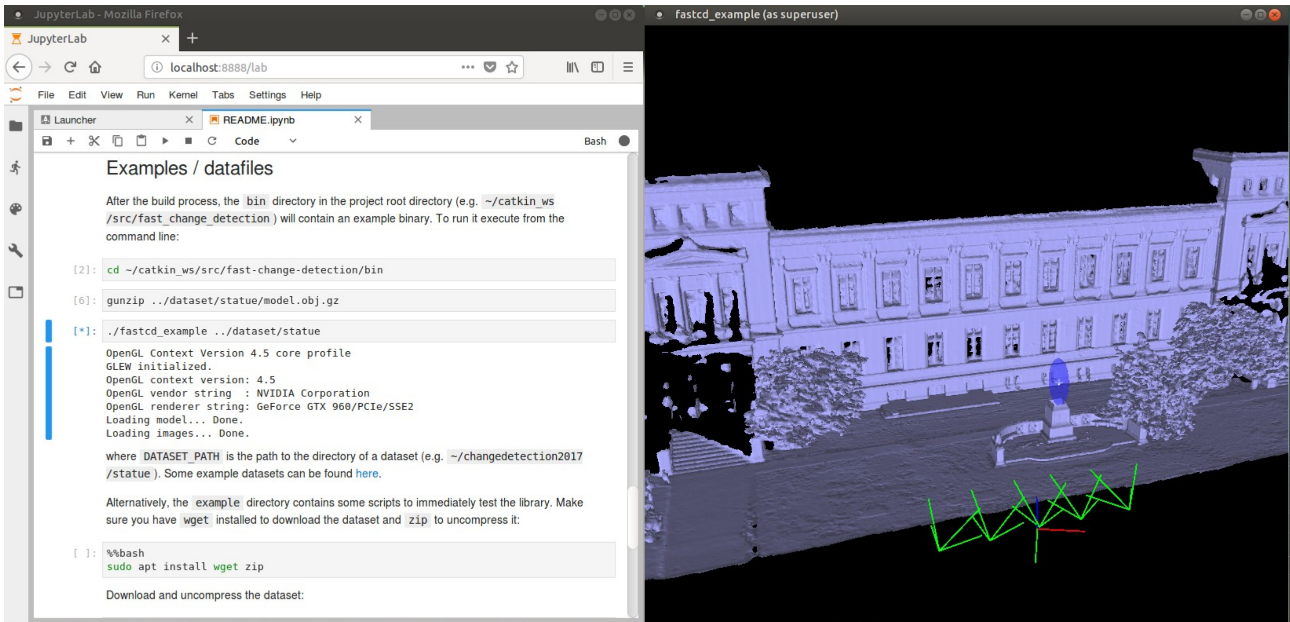

Figure 4: JupyterLab notebook and 3D visualization tool of the Fast Change Detection repository.

Based on the previous examples and other repositories of robotics software, we propose the following guidelines for writing the ROSLab YAML file in an existing git repository:

1. Define the ROS distribution (`kinetic`, `lunar`, or `melodic`).
2. Select the building method of the ROS package (`catkin_make`, `catkin_build`).
3. If 3D graphical output is needed, activate the `nvidia` runtime.
4. Add the software dependencies:
    1. Whenever possible, use binary packages.
    2. Otherwise, use source repositories: in this case, besides of the URL of the repository, you should specify its dependencies and building method.

The necessary disk space for running Docker containers can be large, specially in graphical environments. However, software layers can be shared among different Docker images, thus reducing the total amount of disk space used in the system.

Table 1 summarizes the size of the Docker images for the three presented examples, as well as the size of the ROSLab base images. The "Tag" column refers to the Docker tag that indicates some information about the version or variant of the Docker image ("latest" means the most recently built version for the examples; for ROSLab there are two versions: ROS kinetic with or without the graphical OpenGL runtime environment).

For each image, there is a "Shared" part, and a "Unique" part. The shared layers are stored on disk only once, thus saving quite some space. For the examples, the total size would be 5.131 + 4.471 + 2.517 = 12.119 GB, but thanks to the sharing feature of Docker, it is reduced to 3.742 + 1.474 + 1.39 + 0.729 + 1.042 = 8.377 GB. The saving would be even more significant if additional repositories were installed (based on the same ROSLab images).

REPOSITORY	TAG	SIZE	SHARED SIZE	UNIQUE SIZE
vins-mono	latest	5.131 GB	3.742 GB	1.39 GB
fast-change-detection	latest	4.471 GB	3.742 GB	729.6 MB
nanomap_ros	latest	2.517 GB	1.474 GB	1.042 GB
roslab/roslab	kinetic-nvidia	3.742 GB	3.742 GB	0 B
roslab/roslab	kinetic	1.474 GB	1.474 GB	0 B

Table 1: Images space usage, output of the command `docker system df -v`

The size of ROSLab images is quite similar to that of the official ROS images, with the addition of the JupyterLab software, which is rather small in comparison to ROS: the *kinetic-ros-base-xenial* image takes already 1.191 GB, and the size of the OSRF/ROS *kinetic-desktop-full image* amounts for 3.367 GB.

When the ROSLab images are used for the first time, they must be downloaded from the Docker cloud[1] to the local computer. The images are compressed, so their size is 511 MB and 1GB for the *kinetic* and *kinetic-nvidia* versions respectively, approximately one third of the uncompressed size. The download time will obviously depend on the user’s connection to Internet: for a typical 100 Mbps line, it takes approximately 1’30’’ to download and uncompress the ROSLab *kinetic* image and 3’50’’ for the *kinetic-nvidia* image.

The rest of the time needed for building a repository is the same as if it were built natively: downloading the extra packages and compiling the source code runs on Docker at practically the same speed as in a native system.

The main benefit of using JupyterLab is the strong linkage between documentation and execution: the code documented in the README notebook is actually executed in the same notebook. The alternative would be to open one or more terminals in the Docker image, then copy and paste the code examples from the README file to the terminal, and execute the code in there. This process is prone to errors, and can produce inconsistencies if the documentation is not updated with the changes derived from testing, or the developer forgets to document a step in the terminal.

The JupyterLab README notebook always contains the last version of the executable commands, which can be readily tested by simply restarting the notebook and running all the code cells: if there are no errors, it can be guaranteed that the code will run similarly for any user of the same Docker image.

In addition, using Docker allows the user to run the software no matter which operating system is installed in her computer. It also isolates the running environment from the local system, avoiding clashes with any pre-installed library or third-party software. The Docker image includes all the necessary dependencies, as defined by the developer.

The executable commands in the README file of a git repository should be written according to the Markdown specification,[2] i.e., either fenced by placing triple backticks ``` before and after the code block, or indented with four spaces.

1 https://hub.docker.com/r/roslab/roslab
2 https://github.github.com/gfm/

The computing environment beyond ROS does matter in reproducibility: e.g. some deep learning robotics algorithm are implemented in TensorFlow [6], a software that is under active development, and therefore it is very dependent on the software version. In the near future, ROSLab will be extended with additional base images for Tensorflow or other popular packages.

ROSLab is being developed at the Robotic Intelligence Lab of Jaume-I University and it is freely and publicly available for creating images for ROS Kinetic, Lunar, and Melodic at https://github.com/RobInLabUJI/ROSLab. While still experimental and under development, it is fully functional as demonstrated by the examples, obtained from research papers published at the IEEE ICRA 2018 conference.

We have also published some video tutorials to enable the interested reader to re-run from scratch the examples presented in this paper (https://tinyurl.com/ROSLabExamples) as well as one step-by-step example of how to set up a proper code repository for the application of ROSLab.

All the presented examples have been tested on a machine with a CPU equipped with 4 Intel Core i5-2500 at 3.3 GHz, 8 GB of RAM, a GPU NVIDIA Geforce GTX 960, running Ubuntu 14.04.5 LTS and Docker 18.03.1-ce with nvidia-docker 2.0.

Those examples are a tiny demonstration of the power of JupyterLab notebooks, since only bash commands are executed, and more ambitious examples could include Python or C++ code snippets.

Though the generated Docker images are executed locally in a host, they are compatible with online services such Binder (https://mybinder.org/), which allow the remote execution of JupyterLab servers. Obviously, RViz or other graphical commands could not be executed, but instead the ROS image would be running on the cloud, and the software could be tested by any user with a browser and Internet connection.

We also aim to make ROSLab compatible with other online software platforms like CodeOcean (https://codeocean.com/), which has been proposed as the recommended framework for implementing reproducible research papers in this magazine [7].

Acknowledgments

This paper describes research done at UJI Robotic Intelligence Laboratory. Support for this laboratory is provided in part by Ministerio de Economía y Competitividad (DPI2015-69041-R) and by Universitat Jaume I (UJI-B2018-74).

References

[1] R. White & H. Christensen. “ROS and Docker.” In *Robot Operating System (ROS). The Complete Reference (Volume 2)*, pp. 285-307. Springer, Cham, 2017.

[2] E. Cervera. “Try to Start it! The Challenge of Reusing Code in Robotics Research,” *IEEE Robotics and Automation Letters*, 4 (1): 49 – 56, 2019.

[3] P. R. Florence, J. Carter, J. Ware and R. Tedrake, “NanoMap: Fast, Uncertainty-Aware Proximity Queries with Lazy Search Over Local 3D Data,” *IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, pp. 7631-7638, 2018.

[4] T. Qin, P. Li and S. Shen, "Relocalization, Global Optimization and Map Merging for Monocular Visual-Inertial SLAM," *IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, pp. 1197-1204, 2018.

[5] E. Palazzolo and C. Stachniss, "Fast Image-Based Geometric Change Detection Given a 3D Model," *IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, pp. 6308-6315, 2018.

[6] M. Abadi,P. Barham, J. Chen, and X. Zheng. “Tensorflow: a system for large-scale machine learning,” *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation,*Vol. 16, pp. 265-283, 2016.

[7] F. Bonsignorio. "A new kind of article for reproducible research in intelligent robotics [from the field]." *IEEE Robotics & Automation Magazine* 24.3 (2017): 178-182.