ROSLab

Sharing ROS Code Interactively With Docker and JupyterLab

©ISTOCKPHOTO.COM/ANDREY SUSLOV

By Enric Cervera and Angel P. del Pobil

The success of the Robot Operating System (ROS) and the advance of open source ideas have radically changed and improved the experience of sharing software among members of the robotics community. Yet the lack of a suitable workflow for continuous integration and verification in robotics represents a significant obstacle to developing software that can be run by independent users for testing and reusing purposes.

A typical situation is that a developer produces a new ROS package and shares it through a public source repository from which users can download and execute the software. However, the execution environment may be incompatible due to users running different ROS distributions, incompatible versions of package dependencies, and third-party libraries. Another obstacle is the lack of suitable documentation for running the software. Most shared code

Digital Object Identifier 10.1109/MRA.2019.2916286
Date of publication: 15 July 2019

1070-9932/19©2019IEEE

is barely documented because it was 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 well explained in the documentation because of the developer's familiarity with the software.

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

Recently, Docker has proved to be a useful framework for enabling better, repeatable, and reproducible environmental setups [1]. Docker is an implementation of Linux containers; in other words, it is an OS-level virtualization method for running isolated systems on a control host. It is similar to a virtual machine but with fewer overheads. While Docker is steadily gaining popularity, it is not familiar to many users in the robotics community. In this article, we aim to lower the barriers to adopting Docker by introducing ROSLab, a framework combining Docker and JupyterLab that turns a code repository into a reproducible, runnable software package. ROSLab automatically generates a Docker image using a simple specification from the developer's environment.

The motivation for using JupyterLab is the need to integrate documentation with software, because there is a tendency for experimental code to be accompanied by little explanatory material. JupyterLab is based on notebooks, interactive documents containing executable code, and narrative text, thus allowing 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 to the reuse of robotics code [2].

Figure 1 illustrates the code-sharing process and how it is extended with ROSLab. After the development of a novel research experiment programmed with ROS, users typically have access to a public code repository (for instance, GitHub), which contains one or several ROS packages, a documentation file (README) with a description of the code, and some instructions for building and running the code. An inexperienced user who is interested in testing new developments faces two main difficulties. The first is that the ROS distribution in the user's host may be different than the developer's, whether 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. The second concern is that the documentation may be incomplete or inconsistent. Instructions about third-party libraries could be missing, or some configuration and execution steps might not have not been included.

There is a need for a complete, unambiguous runnable environment for testing publicly available ROS code.

What ROSLab does is automatize the creation of Docker images. Instead of writing a complete Dockerfile, the developer needs only to specify the necessary components for running the image, namely, the ROS distribution, build method, and library dependencies. Those components are

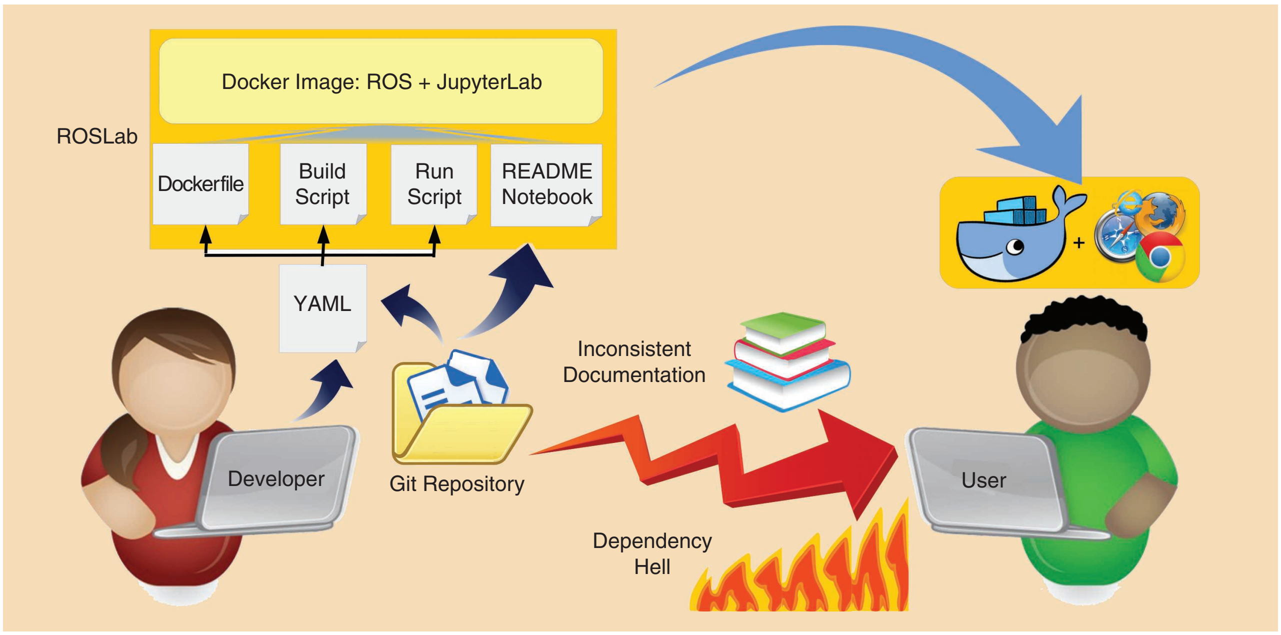


Figure 1. Using ROSLab for sharing a code repository circumvents the trouble of installing and running the software on different system configurations.

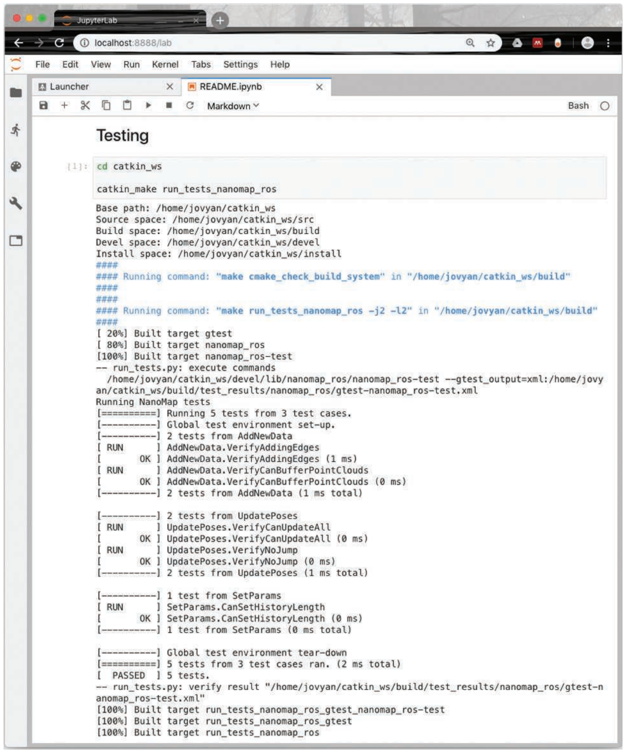
Incomplete or ambiguous documentation is a significant obstacle to the reuse of robotics code.

described in a Yet Another Markup Language (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 Git repository's README file 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 needs to install only Docker and a web browser to run the software. No local ROS installation is necessary; even if it exists, it will not interfere with the ROSLab Docker image.

The following is a simple example of a YAML file (publicly available at https://github.com/ICRA-2018/nanomap_ros) written for the code repository of a paper published in the 2018 IEEE International Conference on Robotics and Automation (ICRA) proceedings [3]:

```
name: nanomap_ros
distro: kinetic
build: catkin_make
```



```
[1]: cd catkin_ws

     catkin_make run_tests_nanomap_ros

Base path: /home/jovyan/catkin_ws
Source space: /home/jovyan/catkin_ws/src
Build space: /home/jovyan/catkin_ws/build
Devel space: /home/jovyan/catkin_ws/devel
Install space: /home/jovyan/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/jovyan/catkin_ws/build"
####
####
#### Running command: "make run_tests_nanomap_ros -j2 -l2" in "/home/jovyan/catkin_ws/build"
####
[ 20%] Built target gtest
[ 80%] Built target nanomap_ros
[100%] Built target nanomap_ros-test
-- run_tests.py: execute commands
  /home/jovyan/catkin_ws/devel/lib/nanomap_ros/nanomap_ros-test --gtest_output=xml:/home/jovy
an/catkin_ws/build/test_results/nanomap_ros/gtest-nanomap_ros-test.xml
Running NanoMap tests
[==========] Running 5 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 2 tests from AddNewData
[ RUN      ] AddNewData.VerifyAddingEdges
[       OK ] AddNewData.VerifyAddingEdges (1 ms)
[ RUN      ] AddNewData.VerifyCanBufferPointClouds
[       OK ] AddNewData.VerifyCanBufferPointClouds (0 ms)
[----------] 2 tests from AddNewData (1 ms total)

[----------] 2 tests from UpdatePoses
[ RUN      ] UpdatePoses.VerifyCanUpdateAll
[       OK ] UpdatePoses.VerifyCanUpdateAll (0 ms)
[ RUN      ] UpdatePoses.VerifyNoJump
[       OK ] UpdatePoses.VerifyNoJump (0 ms)
[----------] 2 tests from UpdatePoses (1 ms total)

[----------] 1 test from SetParams
[ RUN      ] SetParams.CanSetHistoryLength
[       OK ] SetParams.CanSetHistoryLength (0 ms)
[----------] 1 test from SetParams (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 3 test cases ran. (2 ms total)
[  PASSED  ] 5 tests.
-- run_tests.py: verify result "/home/jovyan/catkin_ws/build/test_results/nanomap_ros/gtest-n
anomap_ros-test.xml"
[100%] Built target run_tests_nanomap_ros_gtest_nanomap_ros-test
[100%] Built target run_tests_nanomap_ros_gtest
[100%] Built target run_tests_nanomap_ros
```

Figure 2. The execution of the testing command in the nanomap_ros repository's README notebook.

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

To process the YAML file, Docker must be installed in the host. The ROSLab processing step is executed with the command

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

with <DIR> being the full path of the local folder containing the Git repository where the YAML file is stored. The command produces the following 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 third-party dependencies and building the repository code
- *docker_build.sh*: a script file that invokes Docker for building the image
- *docker_run.sh*: a script file that 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 triggered, and the image is executed with a JupyterLab server launched in the local host, to which the user can connect by opening a uniform resource locator (URL) in a browser (http://localhost:8888/lab/tree/README.ipynb). 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 change directory command must be added to transition to the correct working directory; otherwise, the command produces an execution error.

A more complex example from another ICRA paper [4] includes the use of a graphics processing unit (GPU)-accelerated runtime environment for 3D applications, installation of packages from their source, and mounting a host folder for accessing data set 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. The outcome of the execution is shown in Figure 3, which depicts the browser window and the ROS visualization (RViz) tool that has been launched by one of the commands in the notebook.

In the final example, which is also based 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
```

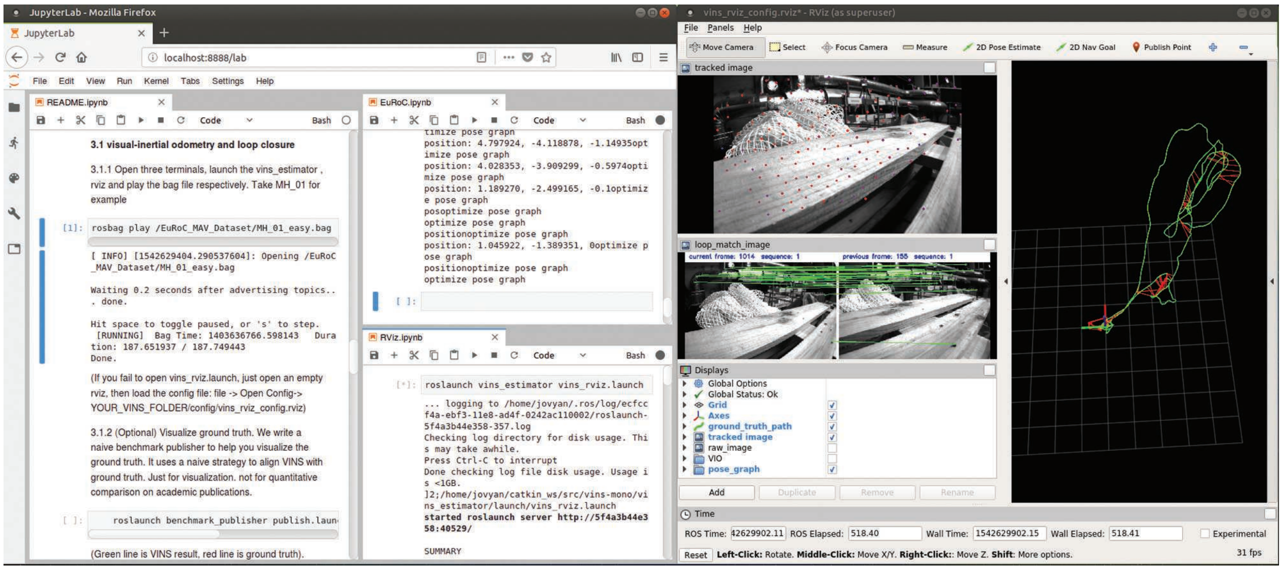
The output of the example is depicted in Figure 4.

Based on the examples presented in this article and in 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.
   a) Whenever possible, use binary packages.
   b) Otherwise, use source repositories. In this case, besides the URL of the repository, you should specify its dependencies and building method.

This process for creating Docker images is generic and can be extended to other software frameworks.

The necessary disk space for running Docker containers can be large, especially in graphical environments. However, software layers can be shared among different Docker



3.1 visual-inertial odometry and loop closure

3.1.1 Open three terminals, launch the vins_estimator , rviz and play the bag file respectively. Take MH_01 for example

```
[1]: rosbag play /EuRoC_MAV_Dataset/MH_01_easy.bag

[ INFO] [1542629404.290537604]: Opening /EuRoC
_MAV_Dataset/MH_01_easy.bag

Waiting 0.2 seconds after advertising topics..
. done.

Hit space to toggle paused, or 's' to step.
 [RUNNING]  Bag Time: 1403636766.598143   Dura
tion: 187.651937 / 187.749443
Done.
```

(If you fail to open vins_rviz.launch, just open an empty rviz, then load the config file: file -> Open Config-> YOUR_VINS_FOLDER/config/vins_rviz_config.rviz)

3.1.2 (Optional) Visualize ground truth. We write a naive benchmark publisher to help you visualize the ground truth. It uses a naive strategy to align VINS with ground truth. Just for visualization. not for quantitative comparison on academic publications.

```
[ ]:    roslaunch benchmark_publisher publish.laun
```

(Green line is VINS result, red line is ground truth).

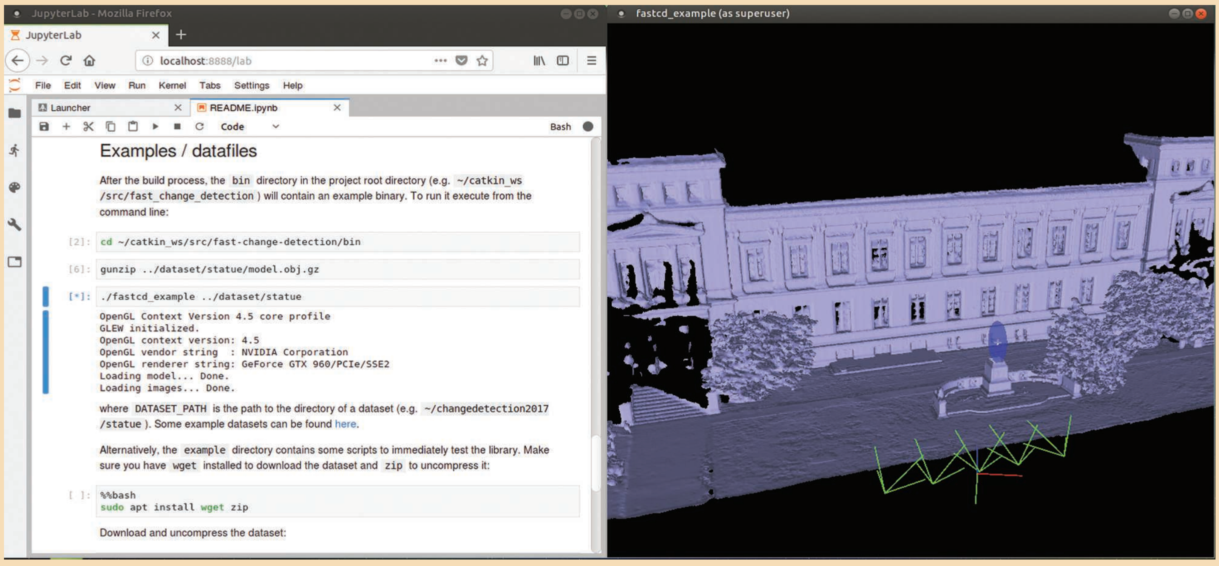
```
timize pose graph
position: 4.797924, -4.118878, -1.14935opt
imize pose graph
position: 4.028353, -3.909299, -0.5974opti
mize pose graph
position: 1.189270, -2.499165, -0.1optimiz
e pose graph
posoptimize pose graph
optimize pose graph
positionoptimize pose graph
position: 1.045922, -1.389351, 0optimize p
ose graph
positionoptimize pose graph
optimize pose graph

[ ]:
```

```
[*]: roslaunch vins_estimator vins_rviz.launch

... logging to /home/jovyan/.ros/log/ecfcc
f4a-ebf3-11e8-ad4f-0242ac110002/roslaunch-
5f4a3b44e358-357.log
Checking log directory for disk usage. Thi
s may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage i
s <1GB.
]2;/home/jovyan/catkin_ws/src/vins-mono/vi
ns_estimator/launch/vins_rviz.launch
started roslaunch server http://5f4a3b44e3
58:40529/

SUMMARY
```

ROS Time: 42629902.11 ROS Elapsed: 518.40 Wall Time: 1542629902.15 Wall Elapsed: 518.41

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click::** Move Z. **Shift**: More options.

Figure 3. Running the VINS-Mono example.


JupyterLab - Mozilla Firefox
JupyterLab
localhost:8888/lab
File Edit View Run Kernel Tabs Settings Help
Launcher
README.ipynb
Code
Bash
fastcd_example (as superuser)

Examples / datafiles

After the build process, the `bin` directory in the project root directory (e.g. `~/catkin_ws/src/fast_change_detection`) will contain an example binary. To run it execute from the command line:

```
[2]: cd ~/catkin_ws/src/fast-change-detection/bin
```

```
[6]: gunzip ../dataset/statue/model.obj.gz
```

```
[*]: ./fastcd_example ../dataset/statue
```

```
OpenGL Context Version 4.5 core profile
GLEW initialized.
OpenGL context version: 4.5
OpenGL vendor string  : NVIDIA Corporation
OpenGL renderer string: GeForce GTX 960/PCIe/SSE2
Loading model... Done.
Loading images... Done.
```

where `DATASET_PATH` is the path to the directory of a dataset (e.g. `~/changedetection2017/statue`). Some example datasets can be found here.

Alternatively, the `example` directory contains some scripts to immediately test the library. Make sure you have `wget` installed to download the dataset and `zip` to uncompress it:

```
[ ]: %%bash
     sudo apt install wget zip
```

Download and uncompress the dataset:

(a) (b)

Figure 4. The Fast Change Detection repository's JupyterLab notebook and 3D visualization tool. The (a) notebook with the commands for running the example and (b) the 3D visualization of the execution.

The main benefit of using JupyterLab is the strong link between documentation and execution.

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 Open Graphics Library runtime environment).

Table 1. Image space usage resulting from the command `docker system df -v`.

Repository	Tag	Size (GB)	Shared Size (GB)	Unique Size
VINS-Mono	Latest	5.131	3.742	1.39 GB
Fast Change Detection	Latest	4.471	3.742	729.6 MB
Nanomap_ros	Latest	2.517	1.474	1.042 GB
ROSLab/ ROSLab	Kinetic-nvidia	3.742	3.742	0 B
ROSLab/ ROSLab	Kinetic	1.474	1.474	0 B

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

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 already takes 1.191 GB, and the size of the Open Source Robotics Foundation/ROS kinetic-desktop-full image amounts to 3.367 GB.

When the ROSLab images are used for the first time, they must be downloaded from the Docker cloud (https://hub.docker.com/r/roslab/roslab) to the local computer. The images are compressed, so their size is 511 MB and 1 GB for the kinetic and kinetic-nvidia versions, respectively, which is approximately one-third of the uncompressed size. The download time will obviously depend on the user's Internet connection. For a typical 100-Mb/s line, it takes approximately 1 min 30 s to download and uncompress the ROSLab kinetic image and 3 min 50 s for the kinetic-nvidia image. The remainder of the time needed to build a repository is the same as if it were built natively. Downloading the extra packages and compiling the source code run on Docker at practically the same speed as in a native system.

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

The JupyterLab README notebook always contains the latest version of the executable commands, which can be readily tested by simply restarting the notebook and running all the code cells. The absence of errors is a guarantee that the code will run similarly for any user of the same Docker image. In addition, employing Docker enables the software to run no matter which OS is installed on the user's computer. It also isolates the running environment from the local system, avoiding clashes with any preinstalled 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 (https://github.github.com/gfm/), for example, fenced by triple backticks ``` before and after the code block or indented by four spaces. The computing environment beyond ROS does matter in reproducibility. For example, some deep-learning robotics algorithms are implemented in TensorFlow [6], which is under active development and, therefore, very dependent on the software version. In the near future, ROSLab will be extended with additional base images for TensorFlow and other popular packages.

ROSLab is being developed at the Robotic Intelligence Lab at Jaume I University, Castelló de la Plana, Spain, 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, it is fully functional, as demonstrated by the previous examples. We have published video tutorials to enable users to rerun our examples from scratch (https://tinyurl.com/ROSLabExamples) and one step-by-step example of how to set up a proper code repository for the application of ROSLab.

All the examples in this article were tested on a machine having a CPU equipped with four Intel Core i5-2500 processors at 3.3 GHz, 8 GB of random-access memory, and a GPU with an NVIDIA GeForce GTX 960 graphics card. The computer was running Ubuntu 14.04.5 LTS and Docker 18.03.1-ce with nvidia-docker 2.0. The examples provide a tiny demonstration of the power of JupyterLab notebooks, because only shell commands are executed: more ambitious examples could include Python or C++ code snippets.

Although the generated Docker images are executed locally in a host, they are compatible with online services, such as Binder (https://mybinder.org/), which allow the remote execution of JupyterLab servers. Obviously, RViz and other graphical commands could not be executed; instead, the ROS image would run in 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 articles for *IEEE Robotics and Automation Magazine* [7].

Acknowledgments

This article describes research conducted at the Jaume I University Robotic Intelligence Laboratory, Castelló de la Plana, Spain. Support for this laboratory is provided, in part, by Spain's Ministry of Economy and Finance under grant DPI2015-69041-R and Jaume I University under grant UJI-B2018-74.

References

[1] R. White and H. Christensen, "ROS and Docker," in *Robot Operating System (ROS): The Complete Reference (Volume 2),* A. Koubaa, Ed. New York: Springer, 2017, pp. 285–307.

[2] E. Cervera, "Try to start it! The challenge of reusing code in robotics research," *IEEE Robot. Autom. Lett.*, vol. 4, no. 1, pp. 49–56, 2019. doi: 10.1109/LRA.2018.2878604.

[3] P. R. Florence, J. Carter, J. Ware, and R. Tedrake, "NanoMap: Fast, uncertainty-aware proximity queries with lazy search over local 3D data," in *Proc. 2018 IEEE Int. Conf. Robotics and Automation (ICRA)*, Brisbane, Australia, pp. 7631–7638.

[4] T. Qin, P. Li, and S. Shen, "Relocalization, global optimization and map merging for monocular visual-inertial SLAM," in *Proc. 2018 IEEE Int. Conf. Robotics and Automation (ICRA)*, Brisbane, Australia, pp. 1197–1204.

[5] E. Palazzolo and C. Stachniss, "Fast image-based geometric change detection given a 3D model," in *Proc. 2018 IEEE Int. Conf. Robotics and Automation (ICRA)*, Brisbane, Australia, pp. 6308–6315.

[6] M. Abadi, P. Barham, J. Chen, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Systems Design and Implementation*, 2016, vol. 16, pp. 265–283.

[7] F. Bonsignorio, "A new kind of article for reproducible research in intelligent robotics," *IEEE Robot. Autom. Mag.*, vol. 24, no. 3, pp. 178–182, 2017. doi: 10.1109/MRA.2017.2722918.

Enric Cervera, Department of Computer Science and Engineering, Jaume I University, Castelló de la Plana, Spain. Email: ecervera@uji.es.

Angel P. del Pobil, Department of Computer Science and Engineering, Jaume I University, Castelló de la Plana, Spain, and Department of Interaction Science, Sungkyunkwan University, Seoul, South Korea. Email: pobil@uji.es.