Run to the Source

The Effective Reproducibility of Robotics Code Repositories

By Enric Cervera

In recent years the robotics community has actively embraced the open paradigm, and research articles are commonly enriched with the inclusion of a source code repository of software. However, the reproducibility of such code is not straightforward, and it may become increasingly difficult with the evolution of software. There is a need for providing not only the source code but also an executable version with all of the necessary library dependencies. A solution based on software containers is presented in this article, with some unique advantages. First, the executable package is automatically generated from the last version of the source code; second, it is archived in the same cloud service that hosts the code repository; third, it integrates seamlessly with the development workflow of the research code; finally, it does not consume any local computing resources from the researcher. The executable code can then be downloaded and run by other users, with the only requirement being installing a specific software for running containers. This article presents the complete workflow, which is then applied to some illustrative examples of source code repositories of articles published at robotics conferences.

INTRODUCTION

Public source code repositories are becoming increasingly available in association with the research articles published in robotics conference proceedings and journals. During the last decade, the open paradigm has gained popularity among the robotics community with many successful stories of software integration and development of complex systems. Most of them are based on the Robotic Operating System (ROS) [1], an open framework that has enabled a significant breakthrough in sharing robotics software [2], [3].

According to the statistics collected in our review of the IEEE *Xplore* digital library [4], nearly one fifth of the articles published in the last edition of the IEEE International Conference on Robotics and Automation (ICRA) and the last volume of *IEEE Robotics and Automation Letters* (*RA-L*) include a code repository. Figure 1 depicts the percentage of articles published in *ICRA Proceedings* and *RA-L* since 2019 that include an open source code repository. In both plots the trend is positive, reaching almost 20% of the articles in 2022.

This trend is even more significant due to the fact that the total number of published articles (in *RA-L* + *ICRA Proceedings*) has increased from 1,622 to 2,020 articles in those years.

Not only are the source code repositories available; they are also actively used by other researchers. In Figure 2 we analyze the code repositories published in *ICRA Proceedings* and *RA-L* between 2019 and 2022 based on their number of *forks*. A fork is a copy of a repository by another user. Forks let researchers make changes to a project without affecting the original repository.

As one would expect, older repositories have more forks than recent ones. The percentage of repositories with more forks increases progressively for past editions of the conference, whereas the number of not-forked repositories decreases. These trends mean that the community members effectively reuse the code repositories for their own research.

Despite the fact that the code is freely available, reproducing the results of a research article is not straightforward [5]. A source code repository must be compiled and linked to the

Digital Object Identifier 10.1109/MRA.2023.3336470
Date of current version: 26 December 2023

required libraries prior to execution. It may be dependent on specific versions of such libraries that run on an outdated operating system (OS) version. Running software that is just a few years old may turn out to be problematic in a current system, unless some kind of virtualization is used.

This problem has been widely recognized in the robotics community [6], and cloud hosting of the software was proposed for reproducing the research results of articles published in *IEEE Robotics & Automation Magazine* [7]. This service is a big step forward, but it demands some effort from researchers for adapting their software. In addition, it might not be suited for interactive applications with a GUI.

To overcome these issues, we present a simple workflow based on continuous integration and continuous delivery (CI/CD) techniques [8] for producing an executable version of a code repository that can be downloaded and run in a straightforward way. Moreover, the executable package is automatically generated from the last version of the source code and archived together with the code repository. Last but not least, the proposed workflow runs silently in the cloud, and it does not require the installation of any software by the researcher or use any resources from the local computer.

The rest of the article is organized as follows. The reproducibility problem is briefly presented in the next section. The section "Building From the Source" describes the proposed method for automatically building a reproducible binary package of a source code repository, and the section "Repositories of Articles" presents some practical examples of repositories included in recently published articles. Finally, the section "Conclusions" gives some final remarks and possible extensions.

THE REPRODUCIBILITY PROBLEM

As pointed out previously, most published code cannot be reproduced in a straightforward way [5]. Lack of documentation and rapid evolution of software components prevent the reproducibility of published code.

During the review of code repositories investigated for writing this article, we found that there was a great variability in the quality of the documentation. In some repositories even a simple README file was missing. Many of them included some basic instructions but lacked the necessary details (such as OS version and required libraries) for building the software without hassle. Other repositories presented complete building instructions and the versions of the libraries, but the building process was time consuming. Sometimes the necessary libraries were not available as binary packages, and they had to be compiled, meaning that their dependencies had to have been previously installed.

Overall, this is a recursive procedure that can become frustrating if any of the dependencies is poorly documented or causes a compilation error due to a change in the application programming interface (API). Software containers can provide an optimal solution for delivering a software package along with all of its dependencies. We define optimality by the number of commands in a script that are needed for downloading, installing, and running the source code repository. In our examples shown in the section "Repositories of Articles," a single command is sufficient. By contrast, with the PyPI package manager, a script with eight commands is necessary in the simplest example for creating the virtual environment, installing the dependencies, and running the software.

Containers are runtime instances of *images*, which are packages of software that include all of the necessary elements to run in a given environment. A workflow based on containers was presented in [9], which allowed the definition of the software dependencies for compiling and running a code repository.

Container technology is not widely used in robotics yet, and building an image may depend on the availability of required packages in online repositories.

Information for building an image in an automatic way is included in very few repositories. This is typically a text file named `Dockerfile`, which is a document that contains all of the commands a user could call on the command line to assemble an image [10].

However, this solution cannot ensure the building of the software without errors in the future. For example, building the image included in an article published at ICRA 2019 [11] raises the error

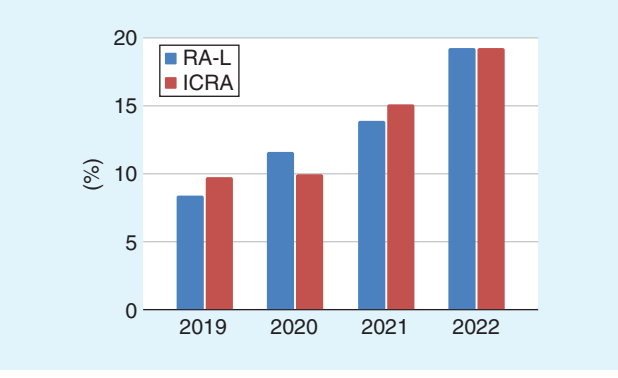

FIGURE 1. Percentage of articles including a source code repository published in *RA-L* and *ICRA Proceedings* since 2019 (articles published in the journal and presented at the conference are not counted twice; only the journal publication is taken into account).

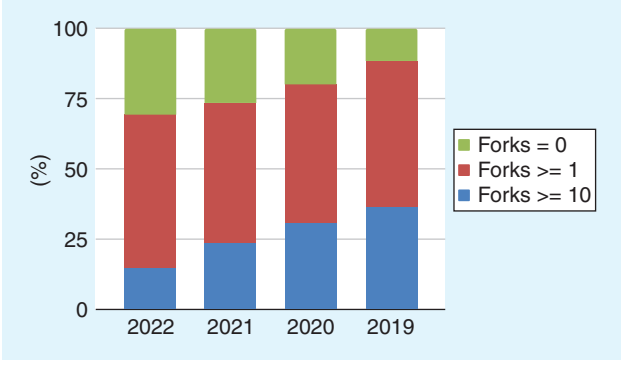

FIGURE 2. Percentage of repositories with respect to their accumulated number of forks since publication to present, for articles published between 2019 and 2022 in *ICRA Proceedings* and *RA-L*.

```
E: Unable to locate package libcudnn7
E: Version '2.7.8-1+cuda10.2'
   for 'libnccl2' was not found
E: Version '2.7.8-1+cuda10.2'
   for 'libnccl-dev' was not found
```

The image is based on Ubuntu 18.04, a long-term support distribution supported until April 2023, but the package that raises the error is not published in the Ubuntu repositories. Instead, it is published in the Nvidia repositories, where only versions 2.8.4 and newer are available [12].

The older a software becomes, the more likely it is to miss necessary library versions in public repositories. It is necessary to generate the executable package as soon as the source code is updated and archive it (together with its third-party dependencies) for public reuse to ensure reproducibility.

BUILDING FROM THE SOURCE

CI/CD services automatically compile and link a code repository and then build the software into a deliverable package. Their use in the robotics domain is gradually becoming more popular, yet they are still underutilized when compared to other fields of software development [8], [13].

In this article we propose a minimalistic CI/CD approach for building and archiving a complete executable environment (a software image) for a given source code repository, with the following features:

- It is automatically generated with each update of the code, thus keeping the executable in sync with the source.
- It is archived in the same cloud service as the code repository (namely GitHub), not requiring any additional account or service.
- It does not interfere with the workflow of the researchers in the development of their code and does not use any local computing resources.

Our approach consists of two steps: first, writing a document with the instructions for building a software image (the section "From Source to Run") and, second, defining a workflow for automatically executing the building instructions and archiving the generated software image when the source code repository is updated (the section "Building and Archiving the Package"). After the software image has been generated, it can be easily distributed; the section "Running the Code" presents three illustrative examples, ranging from basic text-only output to sophisticated GUIs with 3D simulation and visualization. Finally, the section "Comparison With Other Approaches" compares our approach with those of other alternative package managers.

Although CI/CD workflows are routinely used in software engineering and described in popular programmers' websites, such as Medium and tech blogs, the presented approach illustrates for the first time such a workflow with detailed instructions for building reproducible repositories of robotics software from those published in conferences and journals.

FROM SOURCE TO RUN

The steps that are typically presented in the documentation of a repository for building software are as follows:

1) choosing the operating system
2) installing the library dependencies
3) compiling the source code
4) defining a command for launching the software.

The level of detail and completeness of the information determine the success of the execution. Even a perfectly defined set of instructions may not be natively executable because the OS of choice is different from the OS running in the testing machine, requiring some type of virtualization. In addition to this, machines might have different architectures (arm versus x86).

The same steps are used for building a software image, with the only difference that some specific keywords have to be used, as shown in Algorithm 1.

We have forked the public repository https://github.com/ros2/demos and created this Dockerfile in the root folder of the repository.

The first line of the Dockerfile defines the base image—in this case, a distribution of ROS. Lines 2 and 3 create a workspace folder and copy the repository contents to it. The dependencies are installed in lines 4–8. The repository is compiled in lines 9–11. Finally, lines 12–14 are included for sourcing the user workspace, and the last lines (15 and 16) define the command to be executed by default.

BUILDING AND ARCHIVING THE PACKAGE

The Dockerfile defines the steps for building the software, but the set of orders for launching the building process is given in a different configuration file written in YAML. The

ALGORITHM 1: Example of Dockerfile for building a code repository with ROS.

```
 1 FROM ros:rolling
 2 RUN mkdir -p/ros2_ws/src
 3 COPY . /ros2_ws/src/.
 4 RUN . /opt/ros/$ROS_DISTRO/setup.sh \
 5  && apt-get update && rosdep install -y \
 6       --from-paths ros2_ws/src \
 7       --ignore-src \
 8  && rm -rf /var/lib/apt/lists/*
 9 RUN . /opt/ros/$ROS_DISTRO/setup.sh \
10  && cd /ros2_ws \
11  && colcon build
12 RUN sed --in-place --expression \
13   '$isource "/ros2_ws/install/setup.
     bash"' \
14   /ros_entrypoint.sh
15 CMD ["ros2", "launch", "demo_
       nodes_cpp", \
16       "talker_listener.launch.py"]
```

proposed workflow is presented in Algorithm 2. This file must be included in the folder `.github/workflows` of the repository.

Lines 2–4 define the event that triggers the building process; in this case, a push to the master branch of the repository. (Pushing is the operation of uploading the changes to the online repository.) Consequently, upon any update in the source code, a new image will be built and archived.

Lines 5–7 define the container registry where the package is going to be archived. A container registry is a server-side application that stores and lets developers distribute Docker images. Examples of container registries are Docker Hub, Azure Container Registry, and Google Container Registry.

Recently, code repository sites like GitHub and GitLab have launched their own integrated container registries [14], [15]. Their advantage is a complete integration with the code repositories and the available CI/CD pipelines that create and publish Docker images.

We propose the use of the GitHub container registry because the majority of repositories for articles published in *ICRA Proceedings* and *RA-L* are hosted in GitHub; thus, authors do not need to register in an additional service. Nonetheless, it can be replaced by other registries, e.g., Docker Hub [16].

Lines 8–35 define the steps of the workflow for building and archiving the Docker image into the registry. First, the repository is checked out (lines 15 and 16). Second, the system logs in to the container registry defined previously (lines 17–22). This can be easily customized to other registries (Docker, Azure, Google, Amazon Web Services, etc.) as explained in https://github.com/docker/login-action. Third, the system extracts the tags and labels of the Docker image that will be needed later (lines 23–28). Finally, the image is built and pushed to the registry (lines 29–35). The building context can be customized here, but it is typically the root folder of the code repository.

Those steps are independent from the content of the repository itself, so the same YAML file can be used for a wide range of applications.

To sum up, to make a code repository available as a Docker image, the researcher must write a specific Dockerfile with the instructions for building the code and also a generic YAML file with the workflow, and then upload both files to the code repository. Then, on every update in the code, a package will be automatically built and archived, readily available for other users to download and run.

The presented workflow generates images for a single architecture (x86). It could be extended in the future to multiple architectures (e.g., arm) since Docker supports multi-platform images. On the other hand, code optimization may be tricky since the optimized image might be downloaded by a user with a different processor version. In that case, the hardware requirements should be strictly specified in the documentation.

The building process may take some time, depending on the amount of code to compile and the additional packages that need to be downloaded. In our tests, building the example repositories took from some minutes to less than an hour, using the free resources for public repositories. According to the GitHub documentation, each job in a workflow can run for up to 6 h of execution time.

ALGORITHM 2: Definition in YAML of the workflow for building and archiving the Docker image of a source code repository (the complete version can be viewed at https://github.com/RobInLabUJI/ros2_demos/blob/rolling/.github/workflows/publish-image.yaml).

```
1  name: Create and publish a Docker image
2  on:
3    push:
4      branches: ['master']
5  env:
6    REGISTRY: ghcr.io
7    IMAGE_NAME: ${{ github.repository }}
8  jobs:
9    build-and-push-image:
10     runs-on: ubuntu-latest
11     permissions:
12       contents: read
13       packages: write
14     steps:
15       - name: Checkout repository
16         uses: actions/checkout@v3
17       - name: Log in to the Container
               registry
18         uses: docker/login-action@f054...
19         with:
20           registry: ${{ env.REGISTRY }}
21           username: ${{ github.actor }}
22           password: ${{ secrets.GITHUB_
                       TOKEN }}
23       - name: Extract metadata (tags,
               labels)
24         id: meta
25         uses: docker/metadata-action@9866...
26         with:
27           images: ${{ env.REGISTRY }}/
28                   ${{ env.IMAGE_NAME }}
29       - name: Build and push Docker
               image
30         uses: docker/build-push-
                 action@ad44...
31         with:
32           context: .
33           push: true
34           tags: ${{steps.meta.outputs.
                   tags}}
35           labels: ${{steps.meta.outputs.
                     labels}}
```

More details about creating or adapting workflows can be found at https://docs.github.com/en/actions/using-workflows.

RUNNING THE CODE

To download and run a Docker image, an application must be installed by the user on his/her computer. The de facto standard for such an application is Docker [17]. If the application uses a GUI or a GPU for computing, then the extensions named Nvidia Docker (https://github.com/NVIDIA/nvidia-docker) and OSRF/rocker (https://github.com/osrf/rocker) must be installed too.

Downloading an image and running a container requires a single order in the terminal:

```
docker run ghcr.io/<user>/<repository>:<branch>
```

where <user> is the name of the GitHub user account where the repository is hosted, <repository> is the name of the repository, and <branch> is the name of the branch. Packages are public; thus, there is no need to log into GitHub for downloading.

If the image has not been downloaded yet, the system connects to the GitHub registry and downloads it. Next, the container is launched, and the command defined in the Dockerfile is executed. If the image was downloaded previously, the local copy is used.

A different command can be executed simply by appending it to the order

```
docker run ghcr.io/<user>/<repository>:<branch> <command>
```

In that case, the default command is ignored, and the new command is executed instead. The image can also be downloaded without execution with the order

```
docker pull ghcr.io/<user>/<repository>:<branch>
```

Now we present the output of the execution. The first example is the repository with the ROS demonstrations presented in the section "From Source to Run," which is launched with

```
docker run ghcr.io/robinlabuji/ros2_demos:rolling
```

The default command launches two processes, a publisher and a subscriber, which publish and read a text topic, respectively, and send the output to the terminal, as shown in Figure 3. They are stopped by pressing Ctrl-C in the terminal.

A different command can be run in the same container, e.g., calling a ROS service:

```
docker run ghcr.io/robinlabuji/ros2_demos:rolling \
ros2 launch demo_nodes_cpp add_two_ints.launch.py
```

For the second example, we have forked the repository https://github.com/ros-controls/ros2_control_demos, which contains a demonstration of GUI using RViz.

For running such an application, we use OSRF/rocker, a tool with support for X11 and GPUs. It greatly simplifies the configuration of the container, which is executed with the order

```
rocker --x11 \
ghcr.io/robinlabuji/ros2_control_demos:master
```

The graphical output consists of two windows, RViz and a Joint State Publisher, as shown in Figure 4. The user can interact with the sliders and buttons in the publisher window, and the 3D model in RViz is updated accordingly. Similarly, the menus and visualization options in RViz can be freely changed.

For the last example, we have forked the repository https://github.com/ROBOTIS-GIT/turtlebot3_simulations, which contains Gazebo simulations of the TurtleBot 3 robot. It is launched with

```
ecervera@zotac:~$ docker run ghcr.io/robinlabuji/ros2_demos:rolling
[INFO] [launch]: All log files can be found below /root/.ros/log/2022-12-13-11-26-57-436671-e96bda96ea30-1
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [talker-1]: process started with pid [57]
[INFO] [listener-2]: process started with pid [59]
[talker-1] [INFO] [1670930818.578945693] [talker]: Publishing: 'Hello World: 1'
[listener-2] [INFO] [1670930818.579997484] [listener]: I heard: [Hello World: 1]
[talker-1] [INFO] [1670930819.578732808] [talker]: Publishing: 'Hello World: 2'
[listener-2] [INFO] [1670930819.579211867] [listener]: I heard: [Hello World: 2]
[talker-1] [INFO] [1670930820.578657717] [talker]: Publishing: 'Hello World: 3'
[listener-2] [INFO] [1670930820.578884660] [listener]: I heard: [Hello World: 3]
^C[WARNING] [launch]: user interrupted with ctrl-c (SIGINT)
[INFO] [listener-2]: sending signal 'SIGINT' to process[listener-2]
[INFO] [talker-1]: sending signal 'SIGINT' to process[talker-1]
[listener-2] [INFO] [1670930821.136147982] [rclcpp]: signal_handler(signum=2)
[talker-1] [INFO] [1670930821.136741477] [rclcpp]: signal_handler(signum=2)
[INFO] [listener-2]: process has finished cleanly [pid 59]
[INFO] [talker-1]: process has finished cleanly [pid 57]
ecervera@zotac:~$
```

FIGURE 3. Output of the ROS2 demonstration repository.

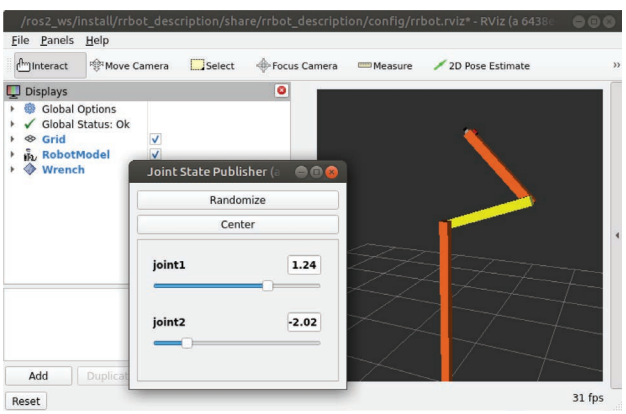

FIGURE 4. Graphical example of the ROS2 control package repository.

```
rocker --x11 \
--env TURTLEBOT3_MODEL=waffle_pi \
--name turtlebot3_sim \
ghcr.io/robinlabuji/turtlebot3_simulations:\
foxy-devel
```

To interact with the simulation, the user opens a second terminal, connects to the running container, and teleoperates the robot with the keyboard:

```
docker exec -it turtlebot3_sim bash
source /opt/ros/foxy/setup.bash
ros2 run turtlebot3_teleop teleop_key-
board
```

The graphical output is shown in Figure 5, with the robot in the Gazebo simulator scene.

The Dockerfiles of the second and third examples can be found in our forked copies of the original repositories:

- https://github.com/RobInLabUJI/ros2_control_demos
- https://github.com/RobInLabUJI/turtlebot3_simulations.

COMPARISON WITH OTHER APPROACHES

There exist other approaches for packaging and distributing software, e.g., PyPI [18] and Conda [19]. These are software package managers that can create, save, load, and switch between environments on the local computer. Software and its dependencies are stored in an environment, and the version of the Python language can be changed. Besides that, Conda is not limited to Python programs; it can package and distribute software for any language.

The main difference between those approaches and ours is that they execute the code on the OS of the host. In our setup, Docker containers are running on a software image that can use a different OS, much like a virtual machine.

Recently, a new approach has been presented for integrating Conda and ROS: RoboStack [20]. It allows the installation of different ROS versions simultaneously in the same machine, using Conda environments. However, RoboStack is not compatible with C++ or Python libraries, which can only be installed via apt. Also, the RoboStack project does not contain all of the available ROS packages. Most of the missing packages require further dependencies to be ported, are abandoned, or do not yet work with Python 3.

In our system, we can use either Python 2 or Python 3, and we use many previous ROS versions, e.g., Indigo based on Ubuntu 14.04, which is available in the official ROS images of Docker Hub. Moreover, our method can accommodate the aforementioned package managers: in the examples described in the sections "Python Application With Graphical Output" and "Deep Learning Application on a GPU," PyPI is used for installing software dependencies.

REPOSITORIES OF ARTICLES

We are now going to apply the presented workflow to the code repositories of three articles published at the ICRA. They have been randomly selected among the articles that included the necessary information for building the code without much trouble and clear instructions for running an example.

In addition, we aimed to use different environments in terms of the programming language, the dependency libraries, and the hardware requirements.

PYTHON APPLICATION WITH GRAPHICAL OUTPUT

The first repository belongs to an article [21] published at ICRA 2020 that uses Python for the implementation of an unscented Kalman filter. It includes different demonstrations of applications of the filter that display graphically the variables of the system and the estimation error.

ALGORITHM 3: Dockerfile for building the code repository of [21], an article published at ICRA 2020 that uses Python for the implementation of an unscented Kalman filter.

```
FROM ubuntu:20.04
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update \
  && apt-get install -y \
  python3-pip \
  python3-tk \
  && rm -rf /var/lib/apt/lists/*
RUN python3 -m pip install --upgrade pip
RUN mkdir /ukfm
COPY . /ukfm/.
RUN cd /ukfm/python \
 && python3 -m pip install -r
    requirements.txt
RUN cd /ukfm/python \
 && python3 -m pip install .
WORKDIR /ukfm/python
ENV PYTHONPATH=/ukfm/python
CMD ["python3", "demo.py"]
```

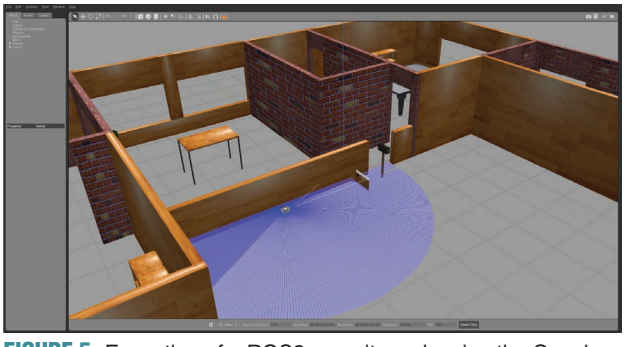

FIGURE 5. Execution of a ROS2 repository showing the Gazebo window with a mobile robot.

The Dockerfile for this repository is presented in Algorithm 3. The authors reported the testing of their code with Python 3.5 on an Ubuntu 16.04 machine, but we were not able to build an image with those versions because of some errors in the installation of the Python packages. Instead, we used Python 3.8 on an Ubuntu 20.04 image (line 1). A text file with the requirements was already present in the repository, which considerably eased the process (line 12). However, some additional requirements were necessary (lines 5 and 6), which were not mentioned in the documentation.

As for the YAML file, we used exactly the same document presented in the previous section without any modification. The repository can be tested with

```
rocker --x11 ghcr.io/icra-2020/ukfm:master
```

A more complete demonstration showing a set of figures can be executed with

```
rocker --x11 ghcr.io/icra-2020/ukfm:master \
python3 examples/pendulum.py
```

The graphical output is shown in Figure 6.

ALGORITHM 4: Commands for installing the source code repository and its dependencies with PyPI and running the demonstration code.

```
1  git clone https://github.com/ICRA-2020/ukfm.git
2  cd ukfm/python
3  python -m venv ukfm_env
4  source ukfm_env/bin/activate
5  pip install -r requirements.txt
6  pip install -e .
7  python demo.py
8  deactivate
```

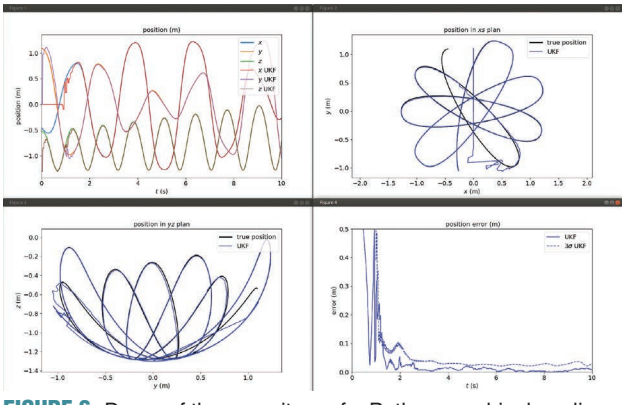

FIGURE 6. Demo of the repository of a Python graphical application showing 2D plots of the results for a pendulum simulation.

COMPARISON WITH PYPI

In the previous example, a single command is sufficient for downloading and executing the Docker image of the repository. If the user wants to reproduce the results using PyPI directly, the necessary steps are shown in Algorithm 4.

First, the source code repository must be cloned (line 1); the user creates a virtual Python environment in line 3, where the software is installed (lines 5 and 6); finally, the demo is executed in line 7, and upon termination the virtual environment is deactivated (line 8).

Using a Docker image is much more straightforward since the image contains all of the dependencies in binary format, and the demonstration script can be readily executed in the same command used for downloading the image.

ROS C++ PACKAGE WITH GAZEBO SIMULATION

The second repository is included in an article [22] published at ICRA 2021, consisting of a C++ implementation of mobile robot planning benchmarks.

We built the image with Ubuntu 18.04 and ROS Melodic as suggested by the authors (see Algorithm 5). The use of a ROS image as a starting point (line 1) reduces the number of additional installs. Nevertheless, we found that a few packages not mentioned in the documentation had to be installed (lines 4–6).

The C++ code is compiled in lines 13–16, and a script is modified for using the new workspace in lines 16–18. Finally,

ALGORITHM 5: Dockerfile for building the code repository of [22], an article published at ICRA 2021 that uses C++ for the implementation of mobile robot planning benchmarks.

```
1  FROM osrf/ros:melodic-desktop-full
2  RUN apt-get update \
3   && apt-get install -y \
4      ros-melodic-navigation \
5      ros-melodic-teb-local-planner \
6      libceres-dev \
7    && rm -rf/var/lib/apt/lists/*
8  RUN mkdir -p/catkin_ws/src \
9   && cd /catkin_ws/src \
10  && git clone \
11  https://github.com/NKU-MobFly-Robotics/p3dx.git
12 COPY . /catkin_ws/src/.
13 RUN . /opt/ros/$ROS_DISTRO/setup.sh \
14  && cd /catkin_ws \
15  && catkin_make
16 RUN sed --in-place --expression \
17     '$isource "/catkin_ws/devel/setup.bash"' \
18     /ros_entrypoint.sh
19 CMD ["roslaunch", "move_base_benchmark", \
20     "move_base_benchmark.launch"]
```

the demonstration command is defined in lines 19 and 20. Again, the YAML file is the same with a single modification in the name of the branch (`melodic-devel` instead of `master`).

This application uses hardware acceleration for the graphical output; thus, a Nvidia card is required, and the code is executed with the command

```
rocker --x11 --nvidia \
ghcr.io/icra-2021/local-planning-benchmark:\
melodic-devel
```

The output is shown in Figure 7. The user defines the goal by clicking in the RViz window, a plan is computed, and the simulated robot starts moving toward the goal in Gazebo. The visualization is updated in real time.

In this example, all of the ROS nodes run in the same container, but it is straightforward to connect these containerized applications to other ROS nodes running on the host or in other containers. The user needs only to run Docker with the `--net=host` option, which makes the processes inside the container look like they were running on the host itself, from the perspective of the network.

DEEP LEARNING APPLICATION ON A GPU

For the third example, we forked the repository of an article [23] published at ICRA 2019, using deep learning for road-object segmentation from a lidar point cloud. The Dockerfile is shown in Algorithm 6.

This repository has more software and hardware requirements, namely TensorFlow, Compute Unified Device Architecture (CUDA), and a GPU. (TensorFlow is a software library for machine learning and artificial intelligence with a focus on training and inference of deep neural networks. CUDA is a parallel computing platform and API that allows software to use GPUs for general-purpose processing.) It is worth noting that the user does not need to install either TensorFlow or CUDA in the local computer; only the GPU driver is necessary.

The authors recommend Ubuntu 16.04, CUDA 8, and TensorFlow 1.4, but we could not find a proper image in the repositories. Instead, we have chosen Ubuntu 18.04, CUDA 10, and TensorFlow 1.13 for installing the repository and running a simple demonstration.

This repository requires a Nvidia GPU card, and the result is not displayed but saved in a folder shared between the container and the local host. Therefore, we use two commands: the `mkdir` command, which creates the folder, and the `rocker` command with the option `--volume` for executing the container and saving the results to the shared folder:

```
mkdir /tmp/samples_out && \
rocker --volume /tmp/samples_out:\
/SqueezeSegV2/data/samples_out:rw \
--nvidia \
ghcr.io/icra-2019/squeezesegv2:master
```

The program uses an already-trained network for segmenting cars and cyclists in a road, and the saved output images are shown in Figure 8.

ALGORITHM 6: Dockerfile for building the code repository of [23], an article published at ICRA 2019 that uses TensorFlow with CUDA on a GPU for road-object segmentation from a lidar point cloud.

```
1  FROM nvidia/cuda:10.0-cudnn7-devel-
       ubuntu18.04
2  ARG DEBIAN_FRONTEND=noninteractive
3  RUN apt-get update \
4   && apt-get install -y \
5      python-pip \
6    && rm -rf/var/lib/apt/lists/*
7  RUN python -m pip install --upgrade pip
8  RUN  mkdir/SqueezeSegV2
9  COPY . /SqueezeSegV2/.
10 RUN cd /SqueezeSegV2 \
11   && python -m pip install -r
       requirements.txt
12 WORKDIR /SqueezeSegV2
13 CMD ["python", "src/demo.py"]
```

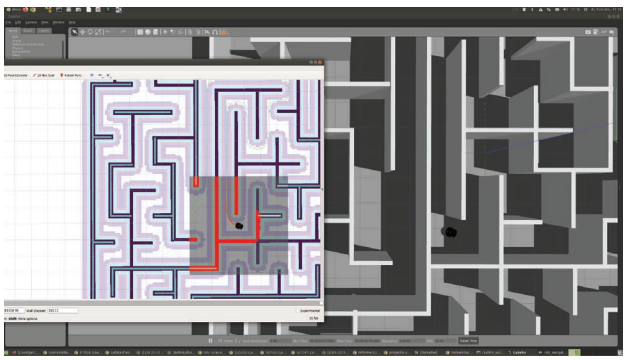

FIGURE 7. Demo of the repository of [22] showing a 3D simulation in Gazebo and the visualization in RViz.

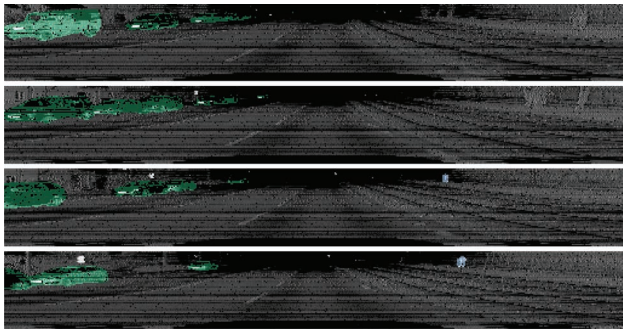

FIGURE 8. Demo of the repository of [23] showing examples of the output label map overlapped with the projected lidar signal. Green masks indicate clusters corresponding to cars, and blue masks indicate cyclists.

TOWARD PRACTICAL REPRODUCIBILITY

In the previous examples, we have shown how to execute Docker images that are already built and stored in the GitHub registry. This method would have solved the problem presented in the section "Building From the Source" if the authors of that repository had built the image while the necessary libraries were available in the repositories.

In that case, the binary of the libraries would have been integrated in the Docker image and conveniently stored in the registry. Any user could later download and execute the image in a container without needing to build it from the source.

The presented examples give a broad overview of applications that can be used as a template or reference for other projects. The repositories, Dockerfiles, and workflow actions are publicly available at

- https://github.com/icra-2020/ukfm
- https://github.com/icra-2021/local-planning-benchmark
- https://github.com/icra-2019/squeezesegv2.

Those repositories are forks from the original ones published with the articles. Instructions for running the Docker images have been included, so they can be easily adapted for similar applications. Typically the only change will be the replacement or addition of the necessary library dependencies, which is straightforward for binary packages. For packages only available as source code, the compilation and installation instructions should be rewritten as Docker commands.

CONCLUSIONS

Robotic researchers are becoming increasingly aware of the importance of releasing their source code to the community, and many code repositories are available together with the articles published in conference proceedings and journals.

GitHub repositories are the most popular option for publishing the source code among the robotics community. We have presented a simple workflow that can be added to any code repository for the automatic generation of a software image. The resulting image is archived in the GitHub registry and can be downloaded and executed in containers by other researchers.

As software development is a continuous process, there is a need for a binary executable version of the code that includes all of its library dependencies. An interested researcher should be able to reproduce the code by simply downloading and executing this binary version, without any compilation.

In addition, researchers should be offered a way to build the packaged version of their software without any change or additional requirement in their workflows.

The solution presented in this article suits well any source code hosted in GitHub, the choice of the vast majority of researchers in the robotics community. But it can also be adapted to other cloud services, such as GitLab or Bitbucket.

The main contribution of this article is to present a workflow that not only generates a software image of a code repository and its dependencies but also allows the resulting image to be archived as a binary package in a site (the registry). Any user can later download and execute the code with a single command. The only previous requirement is the installation of the container software (Docker) and the graphical drivers in case a GPU is required.

The proposed workflow only requires of the developer two steps: first, writing a Dockerfile for building a software image and, second, using the YAML file presented in this article (or a similar one for a different container registry) for the workflow action that will automatically build and archive the image.

The workflow does not interfere with the development of the code, and it runs silently in the cloud, thus not requiring any local computing resources. A user needs to install the Docker application for running containers and the necessary extensions for using GUIs and GPUs.

The software dependencies are installed in the resulting software image and archived for future use. Any software package manager can be used (apt, PyPI, Conda), which ensures the generality of the method.

Our approach has some limitations, though. First and foremost, it cannot solve the lack of documentation of the original code repository; and second, from a practical point of view, the Docker image can be very large and take a long time to download from the registry. Access to the public registries may be limited in the future, and public organizations should be encouraged to create their own registries for ensuring that the software remains publicly available.

ACKNOWLEDGMENT

This article describes research done at the Robotic Intelligence Laboratory. Support for this laboratory is provided in part by Generalitat Valenciana (PROMETEO/2020/034) and by Universitat Jaume I (UJI-B2021-27).

DISCLAIMER

The views and opinions expressed in this article are those of the author and are not affiliated with any of the projects mentioned.

AUTHOR

Enric Cervera, Robotic Intelligence Laboratory, Jaume I University, 12071 Castelló, Spain. E-mail: ecervera@uji.es.

REFERENCES

[1] M. Quigley et al., "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, Kobe, Japan, 2009, vol. 3, p. 5.

[2] S. Cousins, B. Gerkey, K. Conley, and W. Garage, "Sharing software with ROS [ROS topics]," *IEEE Robot. Autom. Mag.*, vol. 17, no. 2, pp. 12–14, Jun. 2010, doi: 10.1109/MRA.2010.936956.

[3] S. Cousins, "Exponential growth of ROS [ROS Topics]," *IEEE Robot. Autom. Mag.*, vol. 18, no. 1, pp. 19–20, Mar. 2011, doi: 10.1109/MRA.2010.940147.

[4] A. Durniak, "Welcome to IEEE Xplore," *IEEE Power Eng. Rev.*, vol. 20, no. 11, p. 12, Nov. 2000, doi: 10.1109/39.883281.

[5] 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, Jan. 2019, doi: 10.1109/LRA.2018.2878604.

[6] F. Bonsignorio and A. P. del Pobil, "Toward replicable and measurable robotics research [From the Guest Editors]," *IEEE Robot. Autom. Mag.*, vol. 22, no. 3, pp. 32–35, Sep. 2015, doi: 10.1109/MRA.2015.2452073.

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

[8] V. Rashitov and M. Ivanou, "Continuous integration and continuous delivery in the process of developing robotic systems," in *Proc. Int. Conf. Objects, Compon., Models Patterns*, Cham, Switzerland: Springer-Verlag, 2019, pp. 342–348.

[9] E. Cervera and A. P. Del Pobil, "ROSLab: Sharing ROS code interactively with Docker and JupyterLab," *IEEE Robot. Autom. Mag.*, vol. 26, no. 3, pp. 64–69, Sep. 2019, doi: 10.1109/MRA.2019.2916286.

[10] R. White and H. Christensen, *ROS and Docker*. Cham, Switzerland: Springer International Publishing, 2017, pp. 285–307.

[11] S. Pillai, R. Ambruş, and A. Gaidon, "SuperDepth: Self-supervised, super-resolved monocular depth estimation," in *Proc. Int. Conf. Robot. Automat. (ICRA)*, 2019, pp. 9250–9256, doi: 10.1109/ICRA.2019.8793621.

[12] Packages for Linux and Unix. Accessed: Dec. 15, 2023. [Online]. Available: https://pkgs.org/search/?q=libnccl2

[13] S. Teixeira, R. Arrais, and G. Veiga, "Cloud simulation for continuous integration and deployment in robotics," in *Proc. IEEE 19th Int. Conf. Ind. Inform. (INDIN)*, 2021, pp. 1–8, doi: 10.1109/INDIN45523.2021.9557476.

[14] "Working with the Container registry." GitHub. Accessed: Dec. 15, 2023. [Online]. Available: https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry

[15] "GitLab container registry." GitHub. Accessed: Dec. 15, 2023. [Online]. Available: https://docs.gitlab.com/ee/user/packages/container_registry/

[16] J. Cook, "Docker hub," in *Docker for Data Science*. Berkeley, CA, USA: Apress, 2017, pp. 103–118.

[17] C. Anderson, "Docker [Software Engineering]," *IEEE Softw.*, vol. 32, no. 3, pp. 102–c3, May/Jun. 2015, doi: 10.1109/MS.2015.62.

[18] "The Python package index." PyPI. Accessed: Dec. 15, 2023. [Online]. Available: https://packaging.python.org/en/latest/tutorials/installing-packages/

[19] "Conda documentation." Conda. Accessed: Dec. 15, 2023. [Online]. Available: https://docs.conda.io/projects/conda/en/latest/user-guide/index.html

[20] T. Fischer, W. Vollprecht, S. Traversaro, S. Yen, C. Herrero, and M. Milford, "A RoboStack tutorial: Using the robot operating system alongside the Conda and Jupyter data science ecosystems," *IEEE Robot. Autom. Mag.*, vol. 29, no. 2, pp. 65–74, Jun. 2022, doi: 10.1109/MRA.2021.3128367.

[21] M. Brossard, A. Barrau, and S. Bonnabel, "A code for unscented Kalman filtering on manifolds (UKF-M)," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, 2020, pp. 5701–5708, doi: 10.1109/ICRA40945.2020.9197489.

[22] J. Wen et al., "MRPB 1.0: A unified benchmark for the evaluation of mobile robot local planning approaches," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, 2021, pp. 8238–8244, doi: 10.1109/ICRA48506.2021.9561901.

[23] B. Wu, X. Zhou, S. Zhao, X. Yue, and K. Keutzer, "SqueezeSegV2: Improved model structure and unsupervised domain adaptation for road-object segmentation from a LiDAR point cloud," in *Proc. Int. Conf. Robot. Automat. (ICRA)*, 2019, pp. 4376–4382, doi: 10.1109/ICRA.2019.8793495.