Contributing

Thank you for your interest in getting involved with the KUKA Robot Learning Lab at KIT. On this page, you will find guidelines on how to help the lab.

Use the lab, report issues and help other users

You are already helping us if you use the lab! We are still in the early stages of the project and every feedback is valuable. You can ask questions in the forum or help other users there.

If you notice a bug, you can open an issue on Github on the SDK's issue tracker. Or if your issue is with a specific project, you can also open your ticket on the issue tracker of the respective project. Our projects are hosted under the KITRobotics group and prefaced with `rll_`.

Improving the documentation

If you notice any error or missing information in the documentation, feel free to create a pull request with the fixes, see Pull requests and commits.

The documentation is created using Sphinx and the source files for the documentation can be found in the SDK docs folder.

The docs README explains how the documentation can be built. The documentation is written using reStructuredText. Here is a great overview of the syntax. Sphinx has its own reference of the added extensions to the standard syntax.

Contributing source code

Members and students at IAR-IPR contribute directly to the IPR Gitlab repos. They also have an IPR account and can access the non-public repos. Their workflow is to fork the needed repos and to make pull requests. And they use the issue trackers in the IPR Gitlab.

External contributors are welcome to create pull requests for the Github repos. Please note that the main development work happens in the IPR Gitlab repos and we do not regularly sync the public repos on Github. Hence, it is possible that the Github repos are outdated from time to time.

Style guidelines

We follow the ROS C++ style guide which in turn follows the Google C++ style guide. The ROS Python style guide is used for Python code and it is based on the standard Python coding conventions.

Several linting tools are available to verify that your code does not violate any of the coding conventions and some of the tools are even able to fix your code automatically.

For C++, we are using ClangFormat and clang-tidy. For Python, flake8 and pylint are used. We also started to add type hints to our Python source code.

To install these tools, simply run:

```
sudo apt install clang-tidy-9 clang-9 clang-format-6.0 python-pip
pip install flake8 'pylint<2.0.0'
```

Configuration files for all these tools are provided in the SDK. You can symlink those to the `src` directory of your workspace to make sure that they are always found when you run the tools in a repo or in a package directory:

```
cd ~/rll_ws/src
ln -s  rll_sdk/.clang-format .
ln -s  rll_sdk/.clang-tidy .
ln -s  rll_sdk/.flake8 .
```

Pylint is not able to find configuration files in parent directories, so its configuration needs to be directly linked to the home directory:

```
ln -s  ~/rll_ws/src/rll_sdk/.pylintrc ~/
```

To automatically format your C++ code, run ClangFormat in the directory of the package you are currently working on:

```
find . -name '*.h' -or -name '*.hpp' -or -name '*.cpp' | xargs clang-format-6.0 -i -
style=file $1
```

The C++ code linter Clang-Tidy needs a package-specific command, e.g.:

```
run-clang-tidy-9.py -p ~/rll_ws/build/rll_move -header-filter=rll_move/*
```

For every catkin package you want to check, you will need to adapt the build path and the header filter. In this mode, Clang-Tidy will not touch your code and it will only print all sorts of warnings and errors that you should fix.

For some of its checks, Clang-Tidy is able to automatically fix your code, but beware: Clang-Tidy is not always right and it may actually break your code. You should definitely commit your code before you run Clang-Tidy with the `-fix` option like this:

```
run-clang-tidy-9.py -fix -p ~/rll_ws/build/rll_move -header-filter=rll_move/*
~/rll_ws/src/rll_sdk/rll_move
```

To check your Python code with flake8 and pylint, run these commands in the package directory:

```
python -m flake8
find * -iname "*.py" | xargs python -m pylint
```

Both tools will not change the code for you. You will need to apply the necessary fixes manually.

Continuous integration and tests

Every RLL repository has a Gitlab continuous integration pipeline with ROS Industrial CI configured and the pipeline is executed for every Git push to an IPR Gitlab repository. We do not yet have continuous integration configured for our Github repos.

The pipeline will try to build the code, run all available tests and the linting tools described in Style guidelines. catkin_lint additionally runs a set of checks on all `package.xml` and `CMakeLists.txt` files.

There is also a separate build with all warnings treated as errors to ensure that all of the compiler warnings have been addressed and that no warnings remain.

Tests are written using the ROS testing framework which uses Google Test for C++ and unittest for Python. The rll_move package has a README with some hints that be used as a reference for writing tests for rll_sdk packages. The rll_stack has its own test collection.

Regardless if you add new functionality or fix bugs, please always consider to extend the current set of tests for the respective package. If the RLL package you are currently working on does not yet have tests, be the first to add some!

Tests help us to quickly spot regressions and they help you to verify that your changes work properly and that you did not break anything.

Pull requests and commits

Before a pull request can be merged, continuous integration needs to pass without errors, see Continuous integration and tests. Additionally, please make sure that all commented code is removed. Complex code section should have comments with explanations. If Todos are left, add comments in the following format to the code to describe them: `TODO(your username): description`.

If you work on a larger issue that requires a big chunk of added or refactored code, a work in progress pull request is very welcome. Then you can get early feedback for your work and we get a better overview of how you intend to solve the issue. For a work in progress pull request, you create a regular pull request but you preface the title with `WIP:`. Then we know that your work is not yet finished and that you might still add changes. When you think that the pull request is ready to be merged, you simply remove the WIP status.

If possible, preface your Git commit title with the name of the component or package you are changing, e.g. `kinematics:`, `move iface:`, `tests:`. Besides a commit title, please also add a description to document your changes and explain your approach if it is not straightforward. You can omit the commit description if your changes are trivial.

The title and description of your pull request should be treated in the same way. The description should summarize the changes of the pull request and mention all open issues on the issue tracker which are addressed by the pull request. If you have a clean commit structure on your branch, then your branch will be merged directly. However, if your branch contains some intermediate commits with fixes, your branch will be squashed into a single commit before merging. In this case, it is especially important that your pull request description provides a complete summary and explanations, because this description will be used for the squash commit message.

Usually, you do not want to work on the master branch. Instead, you should checkout a feature branch with a descriptive name, e.g. the name of the feature or bug you are working on. You will use the same feature branch for your pull request later on.

Before you create your feature branch from the master branch, make sure that your master branch is up-to-date and contains the latest changes from the upstream repo.

It can happen that the upstream repo gets updated while you already work on your feature branch. You should try to get the latest changes into your feature branch if they either directly modify the code you are working on or if they could change the behavior of the testing scenarios that you use for your feature or bugfix. The cleanest way to get these changes into your branch is to do a rebase when pulling into your feature branch:

```
git pull -r upstream master
```

`upstream` is the name of the remote you chose for the upstream repo. To make sure that none of your work gets lost if something goes wrong, you should push your latest changes before you do the rebase. Afterwards, you will need to do a force push to update your remote with your rebased branch.

If you notice an error in your latest commit, try to amend the commit instead of adding a new commit with the fix. You can either add the fixes with `git add` and then running `git commit --amend`, or you can directly amend all new changes with `git commit -a --amend`. If you already pushed the latest commit, you will need to do a force push to overwrite the branch on your remote.

Generally, when in doubt, Git push to avoid loosing any of your work due to false Git operations or other circumstances!