Supplementary Manual 3 - Contributing to the COBRA Toolbox using git

▲ **CRITICAL STEP** This supplementary information is tailored to users who feel comfortable using the terminal (or shell). It is recommended for other users to use the MATLAB.devTools described in Steps 97-102. A Github account is required and *git* must be installed. You also must already have forked the *opencobra/cobratoolbox* repository by clicking on the fork button on the main *opencobra/cobratoolbox* repository page.

The repository of the COBRA Toolbox is version controlled with the open-source standard *git* on the public code development site https://github.com. Any incremental change to the code is wrapped in a commit, tagged with a specific tag (called SHA1), a commit message, and author information, such as the email address and the user name. Contributions to the COBRA Toolbox are consequently commits that are made on branches.

The development scheme adopted in the repository of the COBRA Toolbox has 2 branches: a *master* and a *develop* branch. The stable branch is the *master* branch, while it is the *develop* branch that includes all new features and to which new contributions are merged. Contributions are submitted for review and testing through pull requests, the *git* standard. The *develop* branch is regularly merged into the *master* branch once testing is concluded. The development scheme has been adopted for obvious reasons: the COBRA Toolbox is heavily used on a daily basis, while the development community is active. The key advantage of this setup is that developers can work on the next stable release, while users can enjoy a stable version. Developers and users are consequently working on the same code base without interfering. Understanding the concept of branches is key to submitting hassle-free pull requests and starting to contribute using *git*.

! CAUTION The following commands should only be run from the terminal (or the shell). An SSH key must be set in your Github account settings.

In order to get started, clone the forked repository:

```
$ git clone git@github.com:<username>/cobratoolbox.git fork-cobratoolbox
```

This will create a folder called *fork-cobratoolbox*. Make sure to replace *<username>* with your Github username. Any of the following commands are meant to be run from within the folder of the fork called *fork-cobratoolbox*.

```
$ cd fork-cobratoolbox
```

In order to complete the cloned repository with external code, it is recommended to clone all submodules:

```
$ git submodule update --init
```

Note that your fork is a copy of the *opencobra/cobratoolbox* repository and is not automatically updated. As such, you have to configure the address of the *opencobra/cobratoolbox* repository:

```
$ git remote add upstream git@github.com:opencobra/cobratoolbox.git
```

Now, there are two addresses (also called remotes) configured: *origin* and *upstream*. You can verify this by typing:

```
$ git remote -v
```

In order to update your fork, run the following commands:

```
$ git fetch upstream
```

First, update the *master* branch:

```
$ git checkout master # checkout the <master> branch locally
$ git merge upstream/master # merge the changes from the upstream repository
$ git push origin master # push the changes to the <master> branch of the fork
```

Then, update the *develop* branch

```
$ git checkout develop # checkout the <develop> branch
$ git merge upstream/develop # merge the changes from the upstream repository
$ git push origin develop # push the changes to the <develop> branch of the fork
```

? TROUBLESHOOTING Should the step fail to checkout the develop branch, you should create the develop branch first based on the *develop* branch of the upstream repository:

```
$ git checkout -b develop upstream/develop
```

Create a contribution and submit a pull request

Now, as the fork is up-to-date with the upstream repository, start a new contribution. A new contribution must be made on a new branch, that originates from the *develop* branch. Create the new branch:

```
$ git checkout -b <myBranch> develop
```

Now, you can make changes in the folder *fork-cobratoolbox*. Once you are done making changes, you can contribute the files. An important command that lists all changes is to retrieve the repository status:

```
$ git status
```

A list is displayed with new, modified, and deleted files. You can add the changes (even deletions) by adding the file:

```
$ git add <fileName>.<fileExtension>
```

▲ CRITICAL STEP Contrary to what is sometimes provided as a shortcut, it is not advised to add all files all at once using as this command will add **all** files, even hidden files and binaries.

```
$ git add . # bad practice
```

Then, commit the changes by setting a commit message <*yourMessage*>:

```
$ git commit -m "<myMessage>"
```

Finally, push your commit to Github:

```
$ git push origin <myBranch>
```

You should then see your commit online, and if ready, you can open a pull request. You can select your branch in the dropdown menu and list all commits by clicking on COMMITS.

Continue working on your branch after a while (rebase) If there have been major changes or if you want to continue working on a branch after a while, it is recommended to do a rebase. In simple terms, rebasing your branch shifts your commits to the top of the branch and includes all changes from the upstream repository. Before doing so, make sure that you do not have any uncommitted or local changes (*git status*).

```
$ git checkout develop
$ git fetch upstream
$ git merge upstream/develop
$ git submodule update
$ git checkout <myBranch>
$ git rebase develop
```

If you do not have any conflicts, you should see messages showing that your changes have been applied.

If however there are conflicts, it is advised to use a merge tool such as *kdiff3*. In order to install a merge tool or abort the rebase process, type:

```
$ git rebase --abort
```

In order to have the changes on <*myBranch*> reflected in the online repository, push the changes with force. Pushing with force is required as the history of the branch has been rewritten during rebase.

```
$ git push <myBranch> --force
```

Selectively use a commit on your branch (cherry-pick) Imagine having two branches called <*myBranch-1*> and <*myBranch-2*>. On branch <*myBranch-1*> is a commit with a SHA1 that you need on <*myBranch-2*>. You can cherry-pick the commit from <*myBranch-1*> to <*myBranch-2*> by typing:

```
$ git checkout myBranch-2
$ git cherry-pick SHA1
```

If there are no conflicts, the displayed message should contain the commit message and author information. In order to have the commit listed online, conclude the cherry-pick by pushing the commit to the remote repository:

```
$ git push myBranch-2
```

Displaying the history of a file Sometimes, the history of a file is not correctly displayed online. You can however display the history by typing:

```
$ git log --follow --pretty=short <fileName>.<fileExtension>
```

You can exit the screen by typing the letter *q*.

When the MATLAB.devTools are installed, you can also display the history of a file from within MATLAB:

```
>> history('fileName.fileExtension')
```