Using Docker for your analysis in KM3NeT

Stefan Geißelsöder
ECAP, FAU Erlangen-Nürnberg

March 7, 2017

1 Why should you use Docker for your analysis?

At some point there is a demand to repeat, change or externally check an analysis. Using Docker will simplify your work in these cases.

You can be sure that you will be able to reproduce the parts of your analysis that are covered by Docker even if the computer/cluster/cloud/whatever you used to obtain your result in the first place has been updated or when the specific machine is not available anymore. You can rely on getting the same results, regardless of changes in software versions. Reproducibility is a major benefit of using Docker.

Another advantage is that it also helps a student or colleague that is to reuse your work. He or she will be able to start where you left of and develop further without first having to worry about the details of how to compile, install or run your work.

2 Can you use Docker for your analysis?

For almost all analyses: yes.
The ideal use case is when your analysis can run on a single computer. Foreseeable complications can be dealt with:

- If you use many compute nodes e.g. to prefilter data in a customized way, you can run the evaluation on the prefiltered data with Docker. Alternatively, if the cluster/cloud you use for filtering also supports Docker, you can use a second Docker image for the filtering process.

- If you require dedicated hardware like GPUs, some additional work is required, but the general procedure stays the same. See e.g. [NVI17] for more details on this.

3 How can you use Docker?

We discuss a toy example in C++ on Linux here, but the steps translate to different languages and operating systems. This tutorial can be obtained at [Gei17a]. As a side remark, even though the example in Section 3.2 may be

sufficient to run your own analysis with docker, please also have a look at Section 3.3 for further relevant aspects.

3.1 Prerequisites

This tutorial assumes that you have your input files and your code available and that you have Docker installed[1] and working. On Ubuntu 16.04 you can type

```
sudo apt-get install docker.io
```

You can check if Docker is working correctly by typing "docker ps" into your favorite shell. If the output starts similar to "CONTAINER ID IMAGE COMMAND", you are good to go. If it fails, but "sudo docker ps" works, your user is not part of the Docker group. You can either prepend sudo to all docker commands or add your user to the Docker group (see [dig17]). The commands to do that would be

```
sudo groupadd docker
sudo gpasswd -a ${USER} docker
sudo service docker restart
newgrp docker  # or log out
```

3.2 A toy example

Everything required for our example is located in the folder toyExample. Within toyExample there is a folder named externalInput which contains the prefiltered input files, a folder named myAnalysis which contains the required source code, Makefiles, scripts, etc., a folder externalOutput where the output of the analysis will be stored, a script called runMyAnalysis.sh that will execute the analysis and a file called Dockerfile.

3.2.1 The Dockerfile

The Dockerfile encapsulates the own analysis software together with all software dependencies, aware or not, including even the basic operating system. It is used to create the new Docker image, that executes the analysis. In the example the Dockerfile reads:

```
FROM km3net/km3base:20160909
MAINTAINER S. Geisselsoeder <stefan.geisselsoeder@fau.de>

COPY myAnalysis /myAnalysis
COPY externalInput /input
COPY runMyAnalysis.sh /runMyAnalysis.sh

RUN make -C /myAnalysis
RUN chmod 755 runMyAnalysis.sh
```

[1] Information on how to install Docker can be found e.g. at [dig17] or [doc17] .

```
ENTRYPOINT ["/runMyAnalysis.sh"]
```

The Dockerfile starts with the keyword "FROM", stating from which base image this image will be derived. Typically these base images provide the operating system and some adaptations to it. In our case it's based on CentOS, specifically extended to provide software typically required for analyses in KM3NeT. "km3net" denotes the name of the collaboration on Dockerhub[col16] "km3base" is the name of the base image used for analyses in KM3NeT and "20160909" is the tag identifying which version of the base image is to be used. The common tag "latest" is not used intentionally, as this could change the basis of the analysis unnoticed. MAINTAINER is again a keyword, used to identify and contact the author.

To have the scripts, source code and input files for the analysis available in the Docker image, the keyword "COPY" is used. The first statement

```
COPY myAnalysis /myAnalysis
```

copies the local folder myAnalysis, which is located within the toyExample folder. It contains the source code of the toy example. The destination /myAnalysis specifies where the folder will be available when the Docker container is run. The same is done for the input files by

```
COPY externalInput /input
```

and for the script "runMyAnalysis.sh" that will be used to execute the example.

```
COPY runMyAnalysis.sh /runMyAnalysis.sh
```

This script is explained in Section 3.2.2.

"RUN" is a keyword in the Dockerfile that tells Docker to execute this statement once when creating the image. Hence the command

```
RUN make -C /myAnalysis
```

compiles the source code within the Docker image, producing the binary file. This is analogous to executing "make" within the folder toyExample/myAnalysis/, but in contrast to the command executed on the host computer, it uses the software versions provided by the base image (and potential changes made by previous commands). As this is a well defined state that cannot change over time, the same result will be obtained when the command is executed in the future. The command

```
RUN chmod 755 runMyAnalysis.sh
```

changes the access rights for the file, just as it would do on a host machine. To automatically run the analysis by default when the image is executed, the command

```
ENTRYPOINT ["/runMyAnalysis.sh"]
```

is used.

3.2.2 runMyAnalysis.sh

For simplicity and clarity we incorporate the workflow of the analysis into the docker image using a bash script. This is what "runMyAnalysis.sh" is for. In this example it reads:

```
#!/bin/bash
echo "Using inputs from /input"
ln -s /input /myAnalysis/input

echo "Executing analysis"
cd /myAnalysis
./bin/myAnalysisBinary

echo "Collecting all results to /output/"
mv outputfile.txt /output/
chmod -R 777 /output/
```

This file links the input to the path where the binary expects it to be (we could also copy it), enters the path where our software is located in the file system within the Docker image, executes the already compiled software and moves the resulting output to a desired location. The compilation happened in the Dockerfile at "RUN make -C /myAnalysis", but it could as well be done here. The difference is that the statement in the Dockerfile is executed once when creating the image, while a "make" in this script will be executed every time the image is started. Hence we prefer to have it in the Dockerfile. The change of access rights in the last line is one of the simplest solutions to deal with different users within Docker and on the host system.

Obviously this is a simple setup, but every analysis should be able to be executed by some sort of scripts.

3.2.3 Creating the Docker image

The information in Sections 3.2.1 and 3.2.2 is already sufficient to create a new Docker image. It can be done by executing

```
docker build -t my_analysis:v1 .
```

in the folder toyExample. "my_analysis" is the name we want the Docker image to have, "." is the relative path to the Dockerfile.

Docker will now automatically download all parts of the km3net base image (we specified that in the Dockerfile) which are not present already and then build the new image. Subsequent calls of this command will be faster, as no additional parts have to be downloaded.

If everything succeeded, the command "docker images" should now list at least the base image km3net/km3base with the version tag specified in the Dockerfile and the new image my_analysis with tag v1. A unique version tag is essential, as multiple iterations of an analysis could be produced in the future. To automatically ensure such a tag, the script "createDocker.sh" can be used. It executes the docker build command and adds the date as version tag.

3.2.4 Running the Docker container

After Section 3.2.3 a Docker image with the example analysis should be available. Actually running the example in a Docker container[2] is done using the command

```
docker run -v `pwd`/externalOutput:/output my_analysis:v1
```

While "docker run my_analysis:v1" would be enough to run the example, the output is stored in folder "/output/" as specified in Section 3.2.2. Without any further options this folder lives within the Docker container, but is not accessible from the outside. Therefore the option -v can be used to mount a folder from the host system to a folder in the container. With "-v 'pwd'/externalOutput:/output", all files that are written within the container to /output are actually stored to the folder ./externalOutput on the host system. We write 'pwd'/externalOutput instead of ./externalOutput because the paths may not be relative but have to be absolute when using the option -v.

When we run the Docker container, we see the output that would be printed when executing the script runMyAnalysis.sh directly, and afterwards the output is found in folder ./externalOutput. It should be noted that no changes made to the container by the execution of our analysis are stored to any image. Therefore all desired output has to end up in the designated output folder (or another mounted volume). Even though it is not mandatory for Docker, we propose to use the convention of /input and /output folders for Docker images. This convention simplifies the usage and exchange of images significantly.

We have now successfully ported our analysis to Docker, ensuring that it can easily be run on a different machine and will reliably yield the same results.

3.3 Installing and updating software

As the base image is derived from CentOS, one could update the contained software using yum. But **WE WANT TO AVOID CHANGING THE SOFTWARE OF THE BASE IMAGE!**

To achieve reproducibility, one should ideally rely on the exact software installed in the base image and only add your own code. If you require additional or updated third-party software, the best (and for you most convenient) way is to kindly ask the maintainer of the base image for an updated version. This doesn't cause unnecessarily effort, as others will likely require this change at some point in time, too.

3.3.1 Installing software yourself

Anyway, if you have a reason not to do this (the maintainer might be horribly busy, on vacation for six weeks, ...) you can also install additional third party software yourself, but ***only*** if the version number of the software is specified. This means for example that you may not use

[2]The object running the image (and taking care of various other things in the background) is called a container. One could think of it as being slightly similar to a binary file (image) and the process running it (container).

```
RUN yum -y install boost
```

but instead must use

```
RUN yum -y install boost-1.53.0
```

The flag "-y" is required as creating the Docker image isn't an interactive process. Hence the command would fail without it. If you don't know what version numbers are available for a certain software, you can use

```
docker run -it --entrypoint /bin/bash km3net/km3base:20160909
```

(Replace the tag with your base images)
This opens an interactive shell in a container running the base image. Then type "yum list boost" to see the available versions.

Since even specifying the version of software is not always an option, the next best solution would be to download a fixed state of third-party software and to add it to the image similar to what you would do with your own code, see Section 3.2.1. In this case you have to preserve the downloaded fixed state together with the Dockerfile and your own code, as it has just become part of your analysis software.

3.3.2 Smaller images after installing software

If you need to install software yourself, here are a few hints to produce smaller images:
- Install all required software in one command instead of multiple. Hence

```
RUN yum -y install boost-1.53.0 git-1.8.3.1
```

is to be preferred over

```
RUN yum -y install boost-1.53.0
RUN yum -y install git-1.8.3.1
```

The reason for this is that each of these commands creates an individual layer. Since the status of the image is stored for each layer, a single command requires less space than multiple commands.
- It is also a good idea to add "&& yum -y clean all" at the end of an installation, as unnecessary intermediate and cached files will be removed. The final command should therefore be

```
RUN yum -y install boost-1.53.0 git-1.8.3.1 && yum -y clean all
```

3.4 Dealing with large input files

Instead of copying the input files to the image as we did in Section 3.2.1, we could have also mounted a volume (using -v) as we did for the output files. The advantage of copying the input to the image is that it is stored together with the image, making it trivial to reproduce results in the end. For very large input files this becomes a disadvantage, as these would have to be stored, too.

The best possible approach in such a situation would be to try to extract the information relevant for this analysis, e.g. by removing unused raw data from the files. Alternatively, if the data ain't a custom selection but a standard set used by many analyses, one might think about including it into the base image. The last option should be not to include the input into the image at all, but to mount it. Even though the exact definition of "very large" (and especially "too large") depends on the actual situation and will certainly change over time, as a very coarse guide for orientation, input of up to 2 GB should be included in an image. Nevertheless, it is always a good idea to remove unnecessary parts from the input as long as the original data can be uniquely identified.

3.5 Removing deprecated containers and images

When beginning to experiment with Docker, it is likely that a multitude of containers and images will soon occupy a non-negligible amount of space on the machine. As we won't need our first baby-steps anymore, we can get rid of these images. Since Docker doesn't allow the removal of images that are still used in any container, we first have to get rid of deprecated containers.

```
docker ps -a
```

shows a list of all containers we have run. The relevant identifier for a container is its ID, e.g. "d6cc1ff47c72"

```
docker rm d6cc1ff47c72
```

removes this container. The file "cleanContainer.sh" in the main folder removes all found containers. Don't worry, removing containers does not hurt the images. Once all containers referring to an image are removed, the image itself can be removed.

```
docker images
```

shows the list of stored images. Note that the displayed size of the images also includes base images. Hence your own image is displayed *at least* as big as the base image, but, due to the layered structure of Docker images, in reality only requires approximately the difference of the two numbers. To remove an image one again uses its identifier, e.g. "49462b1e4dcb". The command would be

```
docker rmi 49462b1e4dcb
```

4 Using images with Dockerhub

Once an image has been created it can be uploaded to the KM3NeT Dockerhub account[3]. Beware however to only include input data that should become public in uploaded images. For not (yet) public data, either mount the input instead of including it to the image (see Section 3.2.1) or do not upload the image (yet), but store it on protected storage.

[3]This has to be done by the current administrator of the KM3NeT account.

The benefit is that sharing and executing the analysis becomes trivial. For instance to run the toy example it is sufficient to execute

```
docker run -v `pwd`/extOutput:/output km3net/my_analysis:20170306
```

This does not require you to build the image on your machine. The corresponding image has been created exactly as described in Section 3.2 and is downloaded like the base image has been in Section 3.2.3.

As we understand now how that is achieved, it is simple to use the same scheme for another analysis. We can execute the multiscale analysis [Gei16, Gei17b] with the command

```
docker run -v `pwd`/extOutput:/output km3net/multiscale:20170307
```

The project creating this image can also be found at [Gei17a].

5 Summary

Using Docker for your analysis is a moderate one-time effort, but it benefits yourself (easy to fulfill requests), your colleagues (easier to get involved in your project) and the collaboration (reliable reproducibility).

```
git clone https://github.com/sgeisselsoeder/dockerProjects.git temp
cd temp/toyExample
docker build -t my_analysis:v1 .
docker run -v `pwd`/externalOutput:/output my_analysis:v1
```

References

[col16] KM3NeT collaboration. https://hub.docker.com/u/km3net, 2016. 2017-03.

[dig17] digitalocean.com. How To Install and Use Docker on Ubuntu 16.04. https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04, 2017. 2017-03.

[doc17] docker.com. Install Docker. https://docs.docker.com/engine/installation, 2017. 2017-03.

[Gei16] Stefan Geißelsöder. Model-independent search for neutrino sources with the ANTARES neutrino telescope. *Ph.D. thesis*, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016. http://www.ecap.nat.uni-erlangen.de/publications/pub/2016_Geisselsoeder_Dissertation.pdf.

[Gei17a] Stefan Geißelsöder. Investigating Docker for ASTERICS. https://github.com/sgeisselsoeder/dockerProjects, 2017. 2017-03.

[Gei17b] Stefan Geißelsöder. Multiscale source search. https://github.com/sgeisselsoeder/multiscale, 2017. 2017-03.

[NVI17] NVIDIA. Build and run Docker containers leveraging NVIDIA GPUs. https://github.com/NVIDIA/nvidia-docker, 2017. 2017-03.