



HAL
open science

Using containers for data analysis

G. Henning

► **To cite this version:**

| G. Henning. Using containers for data analysis. 2020. hal-02553519

HAL Id: hal-02553519

<https://hal.science/hal-02553519>

Preprint submitted on 9 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Using containers for data analysis

Author: Greg Henning¹ (Université de Strasbourg, CNRS, IPHC UMR 7178)

Created: April 24th, 2020

Description: this paper describes how one can use container (in particular with docker) to perform data analysis, distribute analysis code,...

A full version of this document, with examples is available online at <https://gitlab.in2p3.fr/gregoire.henning/docker-for-research/>

Motivation

Development environment for data analysis is usually based on a unix system, mostly linux. Several linux distributions are specifically tailored for scientific purposes: [Scientific Linux](#), [Fedora Scientific CentOs](#) and [Cern Linux, BSD](#), ...

However, for practical purpose, the users may prefer more general distributions like Ubuntu, or even different OS such as MacOS and Windows.

The issue of having a unified environment for data analysis inside a collaboration between teams, labs, ... arises naturally from this situation.

Even for a simple few-people team, computing environment can be diverse between the experimental setup, user laptop and desktop computers, and grid infrastructure to run large scale analysis.

In the following paper, I'll discuss mostly the situation of a user running linux software on a windows computer, as it qualifies as the *worst* combination.

The two objective aimed here are:

- running linux softwares on a non-linux system
- having a defined, reproducible and distribuable environment for development and execution of an analysis code.

¹ ghenning@iphc.cnrs.fr

Possible solutions

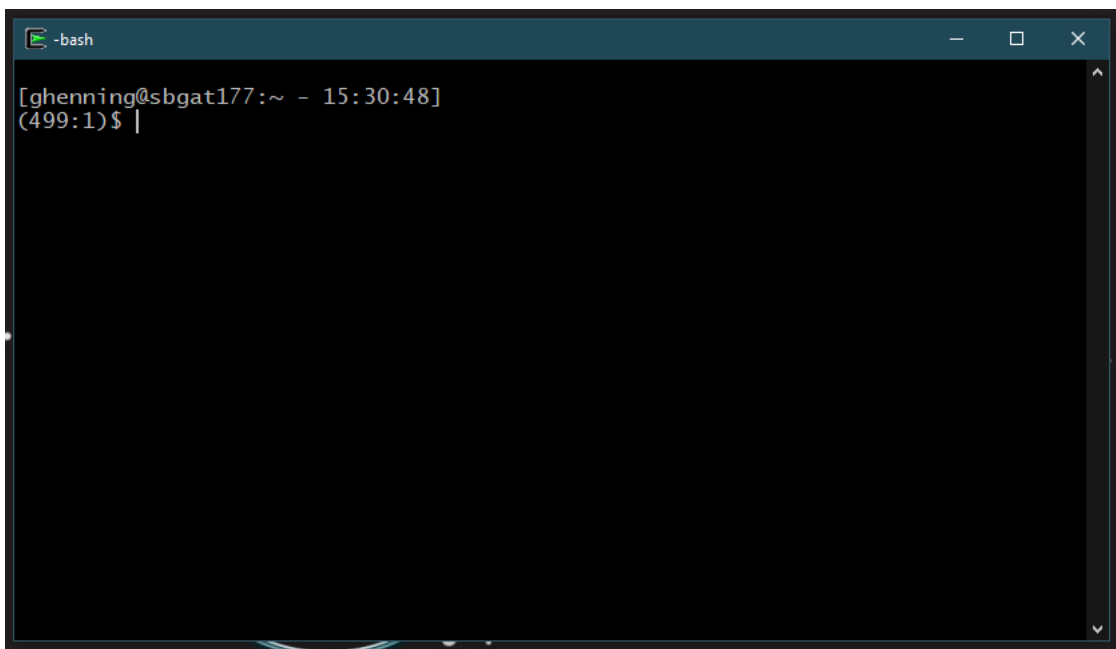
Connecting to a linux machine

The easiest way to run a linux environment on a windows machine is to install and run a remote desktop or secured connection client (e.g. ssh) on the client computer and connect to the linux host.

This method has the interest of simplicity, but requires a connection to the host, and does not provide a specific environment for development and execution, unless the host is specifically setup to that end.

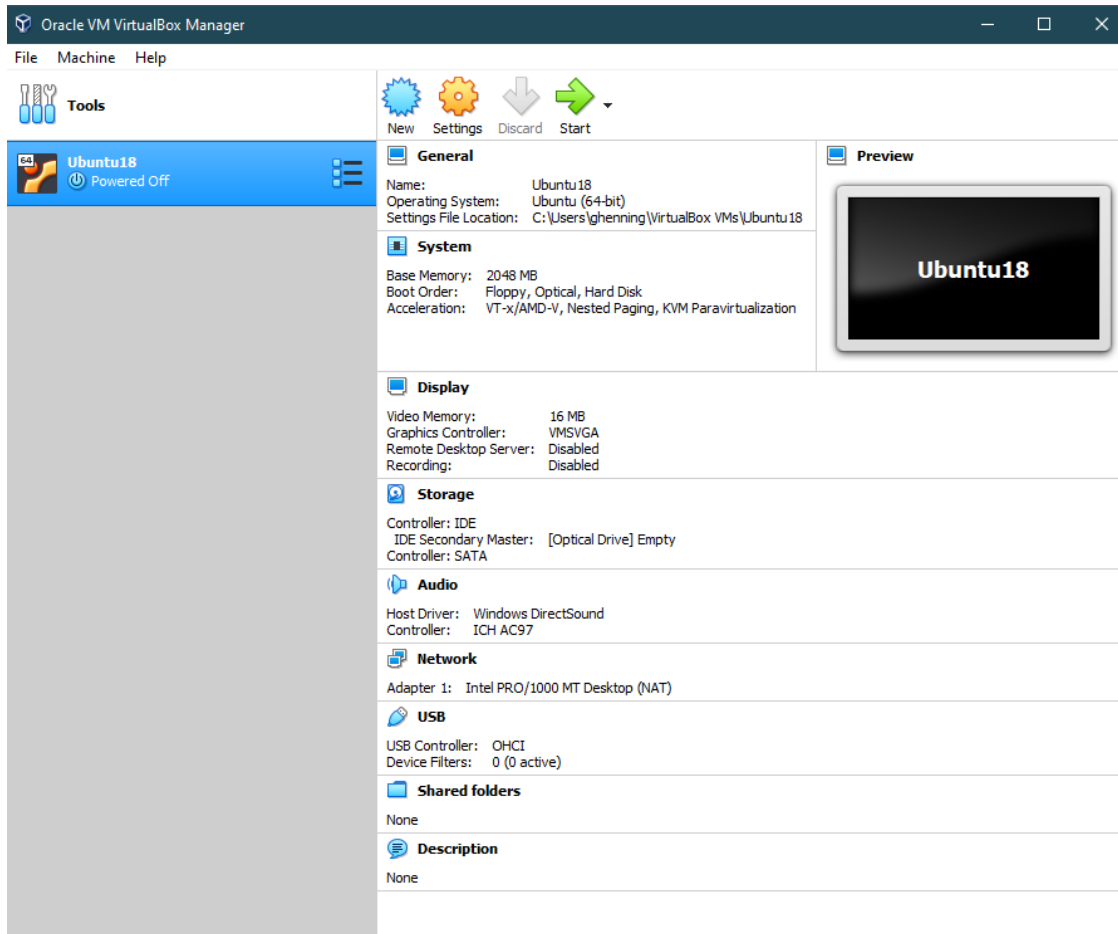
Cygwin

The `cygwin` environment for windows is a suite of *native* windows softwares and libraries that runs a POSIX compatible system. With `Cygwin`, it's possible to compile and run your software *as-if* it were on a linux system. However it is not a linux emulator and not all the applications written for linux will compile and run with `cygwin`.



Virtual machine

A virtual machine (such as [VMware](#) or [Virtual Box](#)) emulates the full computer system on top of the running operating system. This solution allows the user to define the available memory, CPU, ... of the virtual machine. Any system can then be installed as on a real computer.



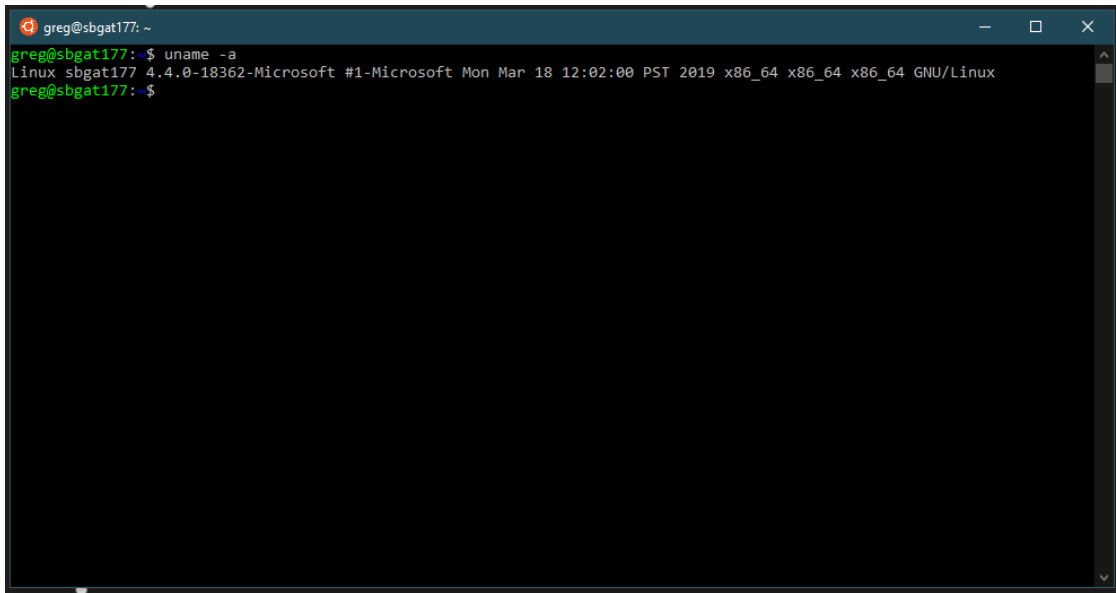
This solution offers the possibility of a very precisely defined and reproducible environment as the virtual machine and installed OS can be chosen. However it is not very efficient for several reasons:

- the whole computer has to be emulated, which costs a lot of memory and CPU on the host
- File exchange can be tricky between the host and the virtual machine
- The virtual machine system takes up to several tens of Giga-Bytes of space on the disk
- It is not possible (unless putting in place specific strategies for that) to restart the virtual machine to its initial state after each run, neither it is simple to run several instances of the same virtual machine in parallel.

Windows Subsystem for Linux

The [Windows Subsystem for Linux](#) (WSL) is available on Windows 10 and offers a Linux kernel for the Windows OS (natively running for WSL2, emulated in its first version).

This allows the Windows user to run linux distributions with ease and minimal overhead compared to a virtual machine. Some out-of-the-box distributions are available in the windows store, such as Ubuntu, SUSE, debian, Kali, ...

A terminal window titled 'greg@sbgat177: ~' with a dark background. The prompt 'greg@sbgat177:~\$' is followed by the command 'uname -a'. The output is 'Linux sbgat177 4.4.0-18362-Microsoft #1-Microsoft Mon Mar 18 12:02:00 PST 2019 x86_64 x86_64 x86_64 GNU/Linux'. The prompt 'greg@sbgat177:~\$' is shown again at the end of the output.

```
greg@sbgat177:~$ uname -a
Linux sbgat177 4.4.0-18362-Microsoft #1-Microsoft Mon Mar 18 12:02:00 PST 2019 x86_64 x86_64 x86_64 GNU/Linux
greg@sbgat177:~$
```

However, some disadvantages of virtual machines exist also: the installed distribution behave like the installed system on a virtual machine: only one instance can be runned and it can't be reset to an intial state.

Docker and containers

Many of these drawbacks are solved by the use of containers.

What is a container ?

A container is an environnement that runs a specific set of codes, with a given set of libraries and parameters.

The basis for a container is an *image* that contains all the code, its dependencies and settings needed. However it does not contain the underlying running system as it is the case for a virtual machine.

When the inage is run by an engine, the container runs on top of the operating system, but isolated from it (hence the name *container*). This means that the container will have its own environment and run only within it, while drawing on the computer capabilities of the host.

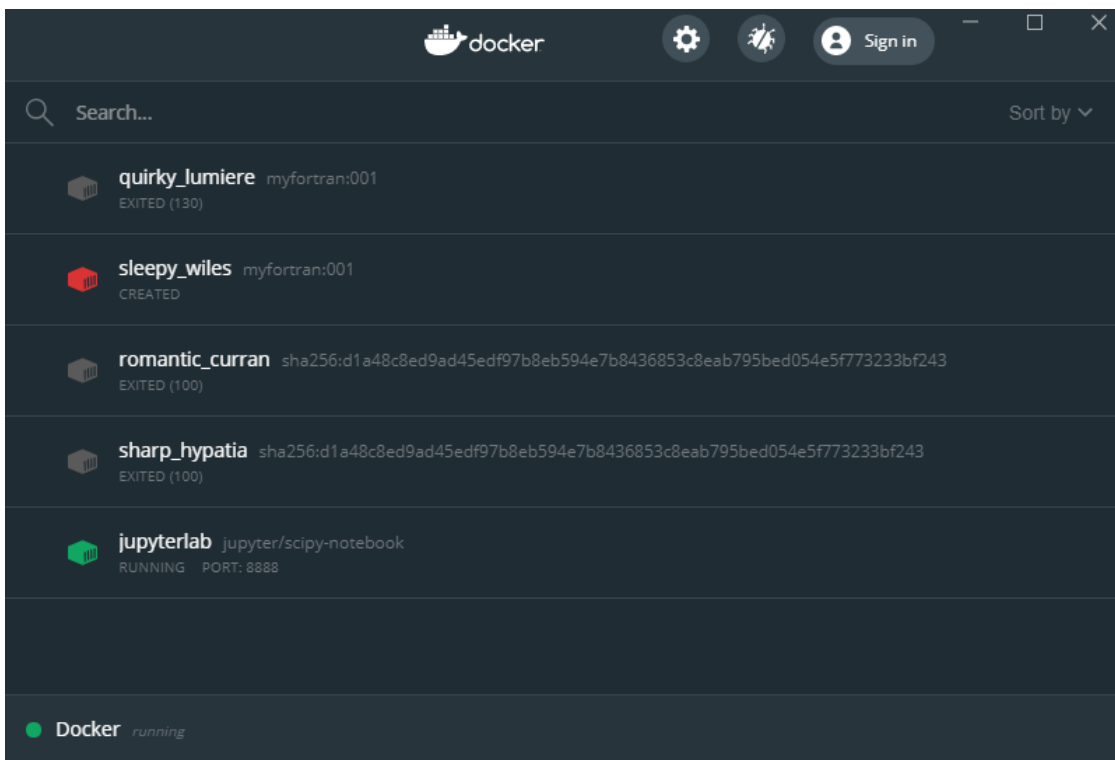
The image is the starting point and can be run multiple times in parrallel. Every new instance of the container will be separated from the others and from previous execution. Because of that, containers are widely used in web application backends.

As the image contains only the code of interest and not the OS ones (which are taken care of by the engine), the image file is smaller than a full virtual machine disk. Still, it can contains parameters, libraries, ... and therefore all the needed files to reproduce the system on any

instance of the engine. Thanks to that, it allows a great deal of uniformity to execute the code even on a diverse set of underlying infrastructures.

Docker and other container engines

Docker is a widely used container engine (or runner). It does all the necessary tasks to run the image with the appropriate containerization. One task of a container engine is to allow network or file exchange with the host, so that the running application within the container can be used with external control.



Other container engines exist, such as Singularity or [LXD](#), but we will discuss only docker here. Most of the examples given below can be transposed in some way to another container runner.

Rather than giving a detailed documentation of Docker, I will present a few use cases.

Use cases

Now that containers have been introduced, let us discuss some use-cases in order to illustrate the interest of using such tool.

L^AT_EX

Installing L^AT_EX on windows can be tedious (not outrageously so, but still more complex than `apt install latex` on Ubuntu). But one can easily use a docker image ready for latex and simply run it with docker to convert `.tex` files to pdf on demand.

We actually don't need to *get* inside the container, we just pass a command to docker and it is executed within the container, then the container simply exits.

```
dir src
docker run --rm -v %cd%\src:/media/usb -w="/media/usb" texlive/texlive-full latex -interaction=batchmode document.tex
docker run --rm -v %cd%\src:/media/usb -w="/media/usb" texlive/texlive-full latex -interaction=batchmode document.tex
docker run --rm -v %cd%\src:/media/usb -w="/media/usb" texlive/texlive-full dvipdfm document.dvi
dir src
```

(replace `%cd%` with ``pwd`` when using a bash-like shell instead of windows' batch).

The command is divided in several parts:

- `docker run` calls docker and tells it to run a container.
- `--rm` tells docker to clear (remove) the container after running (otherwise, its state is saved and one can go back to it later).
- `-v %cd%\src:/media/usb` mounts the host directory `%cd%` (i.e. the current working directory) onto the container's `/media/usb` point
- `-w="/media/usb"` tells docker that the working directory on the container will be `/media/usb`
- `texlive/texlive-full` is the name of the image to fetch from the [image repository](#) and run in the container
- `latex -interaction=batchmode document.tex` and `dvipdfm document.dvi` are the command to run in the container.

The first time the image is used to run a container, docker will fetch it from the docker hub. The image is fetched by parts, because the images are built by adding layers on top of each other. We will see in the next example how we can create our own image by adding some libraries to an existing one.

Note that the command `docker run ... latex document.tex` is called twice because that's the needed number for latex to properly build section indexing.

The latex compilation files appear directly and instantly in the directory, since there's no virtualization layers between the container and the host: latex runs directly on the files.

Test this in the [latex](#) directory of the examples

Fortran

It can happen that you have to use some tools written in Fortran. As for other languages, installing and running a fortran compiler on Windows can be tricky. The easiest way to

compile such software is therefore to start a container dedicated to fortran compilation. Once the code is compiled it can be run from inside the container.

In this example, we will use the ruler program provided by the IAEA as part of the *ENSDF Analysis and Utility Programs*.

We will also create our own fortran-dedicated image to be run in a container. To that end, we start by writing a Dockerfile:

```
FROM alpine:3.11.5

RUN apk add gfortran wget

RUN adduser --disabled-password --gecos '' fortran
USER fortran
WORKDIR /home/fortran/

CMD ["/bin/sh"]
```

We then create the docker image with

```
docker build -t myfortran:001 .
```

The image is built (i.e. the commands are executed and the resulting image is kept to be launch in a container on demand) and saved under the name and tag `myfortran:001`.

We can now launch the image in a interactive container (i.e. we will be able to type commands in the shell).

```
> docker run --rm -ti -v %cd%:/home/fortran/work myfortran:001
```

And now, we can download, compile and run the fortran software:

```
~ $ cd work
~/work $ wget --user-agent="Mozilla/4.0 (Windows; MSIE 7.0; Windows NT 5.1; S
V1; .NET CLR 2.0.50727)" --no-check-certifi
cate --no-cache --no-cookies https://www-nds.iaea.org/public/ensdf_pgm/analys
is/ruler/ruler-src-2019-01-24.zip
--2020-03-17 06:01:41-- https://www-nds.iaea.org/public/ensdf_pgm/analysis/r
uler/ruler-src-2019-01-24.zip
Resolving www-nds.iaea.org... 104.20.23.134, 104.20.22.134, 2606:4700:10::681
4:1686, ...
Connecting to www-nds.iaea.org|104.20.23.134|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 239955 (234K) [application/zip]
Saving to: 'ruler-src-2019-01-24.zip'

ruler-src-2019-01-24.zip  100%[=====
=====>] 234.33K  --.-KB/s    in 0.06s

2020-03-17 06:01:44 (3.81 MB/s) - 'ruler-src-2019-01-24.zip' saved [239955/23
```


9955]

```
~/work $ unzip ruler-src-2019-01-24.zip
Archive:  ruler-src-2019-01-24.zip
  creating: ruler-src/
  inflating: ruler-src/howto.txt
  inflating: ruler-src/Makefile
  inflating: ruler-src/nsdflib95.f
  inflating: ruler-src/ruler.ens
  inflating: ruler-src/ruler.f
  inflating: ruler-src/ruler1.rpt-0
  inflating: ruler-src/ruler1.tti
  inflating: ruler-src/ruler1.tto-0
  inflating: ruler-src/ruler2.out-0
  inflating: ruler-src/ruler2.prob-0
  inflating: ruler-src/ruler2.rpt-0
  inflating: ruler-src/ruler2.tti
  inflating: ruler-src/ruler2.tto-0
  inflating: ruler-src/ruler3.ens
  inflating: ruler-src/ruler3.out-0
  inflating: ruler-src/ruler3.prob-0
  inflating: ruler-src/ruler3.rpt-0
  inflating: ruler-src/ruler3.tti
  inflating: ruler-src/ruler3.tto-0
~/work $ cd ruler-src/
~/work/ruler-src $ gfortran ruler.f nsdflib95.f -o ruler
~/work/ruler-src $ ./ruler <ruler1.tti
```

```
RULER Version 4.1a [19-Nov-2017]
Modified by F.G. Kondev (ANL) on 12/12/11
```

```
      INPUT DATA FILE (DEF: ruler.inp):          OUTPUT REPORT FILE (DEF: ruler.rpt):
      Mode of Operation
      (R-Compare to RULs,B-Calculate BELW,BMLW)?    Assumed DCC theory
(Bricc-1.4%, Hsicc-3%, Other-?) - CURRENT DATA SET: 228TH    ADOPTED LEVELS,
GAMMAS
CURRENT DATA SET: 228TH    228AC B- DECAY
CURRENT DATA SET: 228TH    228PA EC DECAY
CURRENT DATA SET: 228TH    232U A DECAY
CURRENT DATA SET: 228TH    226RA(A,2NG)
CURRENT DATA SET: 228TH    230TH(P,T)
                        NO GAMMAS EXPECTED
CURRENT DATA SET: 228TH    230TH(A,A 2NG)
```

```
Program completed successfully
~/work/ruler-src $ diff ruler1.rpt ruler1.rpt-0
--- ruler1.rpt
+++ ruler1.rpt-0
@@ -1,5 +1,5 @@
```

```
-RULER Version 4.1a [19-Nov-2017] RUN ON 17-Mar-2020  
+RULER Version 4.1a [19-Nov-2017] RUN ON 24-Jan-2019
```

```
Comparison to RUL s Mode  
INPUT FILE: ruler.ens  
~/work/ruler-src $ echo "Done !"  
Done !  
~/work/ruler-src $ exit
```

The commands are listed in the `commands.sh` file of the example directory.

Dedicated softwares and libraries

Science and data analysis often comes with many dedicated softwares and libraries. Keeping up with the versions, ... on a single machine can be tedious. It's even harder - as mentioned in the introduction - in a large collaboration where many team members have to use the same environment.

That's when docker (or containers) come in play and make everything simpler. Members of the team just have to use the same image for their work and the homogeneity of software and libraries environment is assured.

Cern's Root

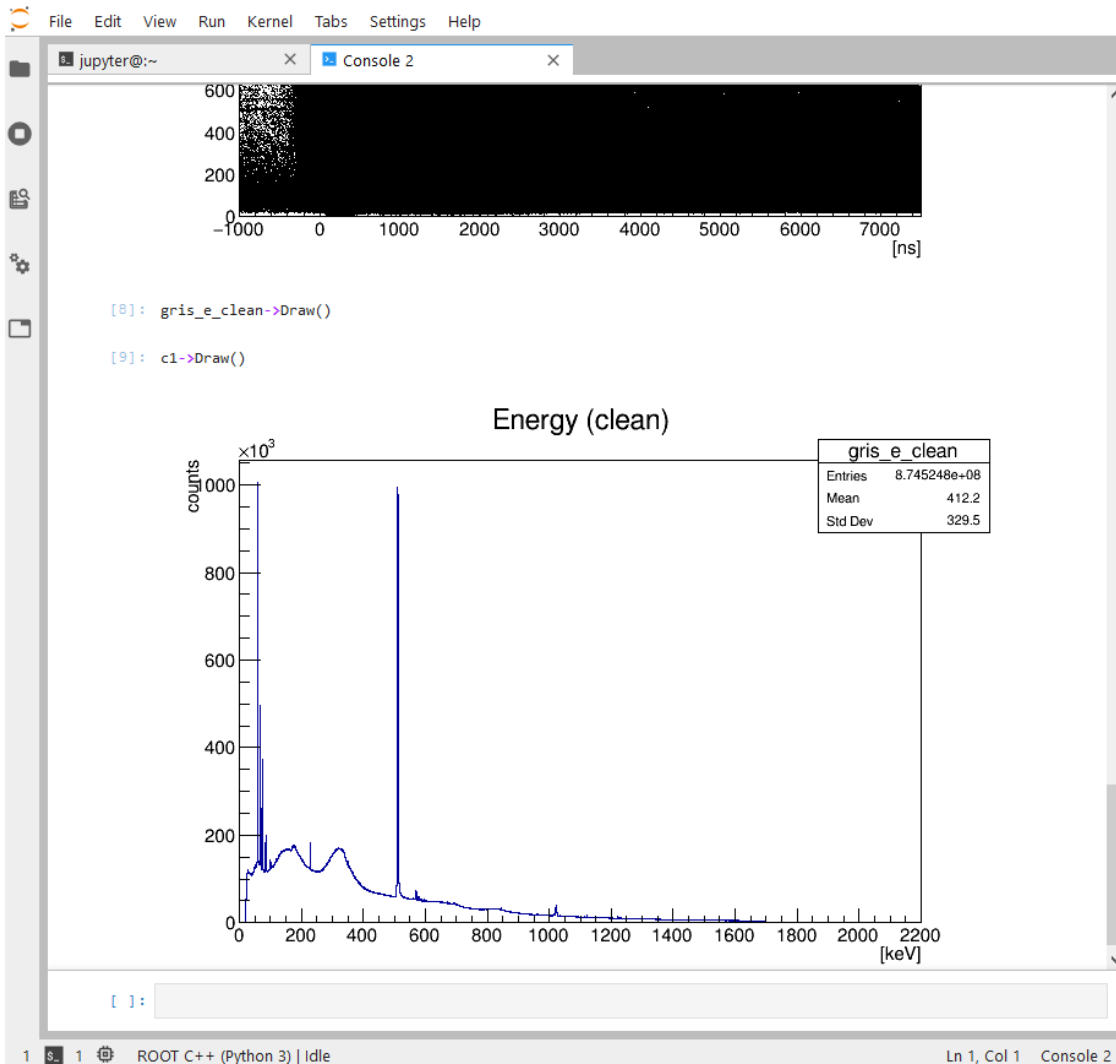
The largely used [Root](#) library has several versions with sometimes no backward compatibility between major ones.

Docker can help for both cases: either to keep up to date with the latest version without having to reinstall or upgrade your own computer, or keeping one static version.

Unfortunately, there has not been any support for official root image for a while: <https://hub.docker.com/u/rootproject>

A few basic images exist and can be used, but their interest stays limited as a working image. However, they can be used as basis for your own images.

If you want an image that can work for you out of the box, you can try unofficial ones. The image [akalinow/root-fedora31](#) works well and offer jupyterlab support, solving the issue of display with a container (a container is intended to work as command line only or over http connection).



In general, it is better if you or your own collaboration designs its own image with the specific tools needed, by building on top of existing ones.

Geant4

Like Root, there is **no** official image provided. An unofficial image for training purpose exist at <https://hub.docker.com/r/dplatten/geant4-educational> with the issue of visualisation solved thru the use of an RDP server running in the container to which you connect from outside. It's an elegant solution, but makes the image quite large.

In the following, we will write our own docker file for a simple geant4 image.

Writing the Dockerfile

The Dockerfile will contain the following lines:

- `FROM ubuntu:18.04` indicates that we are using ubuntu version 18.04 as basis for our image.

- ```

RUN apt-get update &&\
 apt-get install -y libxerces-c-dev qt4-dev-tools freeglut3-dev
libmotif-dev tk-dev cmake libxpm-dev libxmu-dev libxi-dev wget tar &&\
 apt-get clean

```

this updates the package list for the distribution, install the needed libraries to build Geant4 and clean up. It is important to put all the commands into one RUN line because each call to RUN creates a new *layer* in the image and potentially increase the size of the final image. If we were to clean in a separate RUN line, the previous layer would still contain the files cached by apt and the size of the final image will not be reduced by the subsequent `RUN apt-get clean` call.
- ```

RUN mkdir /geant4.setup && cd /geant4.setup && \
  wget http://cern.ch/geant4-data/releases/geant4.10.06.p01.tar.gz && \
  tar -xsf geant4.10.06.p01.tar.gz && \
  mkdir geant4-build && mkdir /geant4-install && \
  cd geant4-build && cmake -DCMAKE_INSTALL_PREFIX=/geant4-install -
DGEANT4_INSTALL_DATA=ON /geant4.setup/geant4.10.06.p01 && \
  make install && \
  cd / && rm -rf /geant4.setups

```

this series of command (again, put together to reduce the layer size) download the geant4 installation package, unpack it, configure it, compile it, install it and clean up the setup files.
- ```

RUN useradd -m geant
USER geant
WORKDIR /home/geant/

```

Creates a user geant and setup the image to start running with the working directory /home/geant and the user geant
- ```

RUN echo "cd /geant4-install/bin; source geant4.sh" >>
/home/geant/.bashrc &&\
echo "clear;" >>/home/geant4.bashrc

```

Sets up the bash environment for the user geant so that it loads the PATHS needed for geant4 when starting a new bash session.
- ```

CMD ["/bin/bash"]

```

Sets /bin/bash as the *entrypoint* of the image, i.e. the command that is run when starting a container.

The actual Dockerfile can be found [here](#). It contains a few additional commands to make the compilation easier.

The image building then is done by calling, from the directory where the dockerfile is,

```
docker built -t geant4:001 .
```

The different calls of RUN will then be executed. In particular, the compilation of the geant4 library will be done. It will take some time.

Once the image is built, you can verify it by calling

```
$ docker images
REPOSITORY TAG IMAGE ID CREATED
SIZE
alpine 3.11.5 a187dde48cd2 3 days ago
5.6MB
geant4 001 196bbfe60bfa 9 days ago
2.87GB
ubuntu 18.04 775349758637 4 months ago
64.2MB
faster 001 a9083b220ea6 3 months ago
389MB
```

We see that the size of the image is about 2.87 GB, most of it comes from the reaction data bases used by Geant4.

### Using the image

Now, we will start up an container using our new image and run an example of Geant4.

Start docker with

```
docker run --rm -ti geant4:001
```

We can now type the following [commands](#) in the bash terminal :

```
mkdir example
wget http://cern.ch/geant4-data/releases/geant4.10.06.p01.tar.gz
tar -xzf geant4.10.06.p01.tar.gz
cp -vr geant4.10.06.p01/examples/basic/B2 ./example/
rm -rf geant4.10.06.p01.tar.gz
cd example
mkdir B2-build
cd B2-build
cmake ../B2
make
cd B2a
ls -lsh
./exampleB2a run1.mac
```

This will download, prepare, compile and run an example script for Geant4. If it runs correctly, that means our distribution is ready to be used to run our simulations.

Obviously, you may want to add to this Dockerfile, for example to add support for the Root library...

### Using a software distributed via linux package

Containers can also be a simple solution to use a software distributed via only *one* channel (such as debian repository) on an *a priori* incompatible system.

For example, the faster acquisition system comes with some analysis tools that are distributed uniquely as debian packages. That prevents anybody running Windows, MacOS or a non-debian based flavor of Linux to use them... unless using containers.

It is very easy to create an [image](#) on which the package is installed and can be runned from:

```
FROM ubuntu:18.04
```

```
RUN apt-get update &&\
 apt-get install -y wget software-properties-common &&\
 apt-get clean
```

```
RUN wget -O - http://faster.in2p3.fr/distribution/fasterv2/fasterv2_repo_pgp_
pub_key.asc | apt-key add - && \
 apt-add-repository 'deb http://faster.in2p3.fr/distribution/fasterv2/ubun
tu/ bionic non-free' && \
 apt-key list && \
 apt-get update && \
 apt-get install -y fasterac &&\
 apt-get clean
```

```
RUN adduser --disabled-password --gecos '' faster
```

```
USER faster
```

```
WORKDIR /home/faster/
```

```
CMD ["/bin/bash"]
```

The image is built using `docker build -t faster:001 .` and then used as following for [example](#).

Start the container with

```
docker run -ti --rm -v %cd%:/home/faster/work faster:001
```

and type the following commands:

```
pwd
cd work
ls -lhs
faster_disfast -I 073_5650keV_0001.fast
exit
```

This allows you to use the faster tools on your local files as if you were on an Ubuntu 18.04 system.

### Publishing your own application by distributing binaries without the source code

Finally, using images can be useful wif you want to distribute your software (with all its dependencies) but **not** the source code. **Evidently, it is strongly recommended to distribute data and analysis source code openly as part of an Open Science strategy.** However, there are various good reasons why fully open distribution of your code is not

possible (because you use restricted distribution dependencies, part of the code is not open, ...) in that case, distributing the final binary is a good alternative.

Building an image, one can compile a code in a controlled environment and remove the sources from the final image, effectively distributing a guarantee-to-run binary without being concerned with users not having the necessary libraries, ... (since they are all in the image).

In this example, we will compile a C++ code written to read faster files and extract data in ascii formatted files and distribute it as an image. The code we use is not restricted in access, although not yet openly distributed because it has to go thru additional testing before release to the public.

The Dockerfile looks like this:

```
FROM gcc:9.3.0

COPY faster2h-c /setup
RUN ls -lhs && cd setup && ls -lhs &&\
 make binaries && ls -lhs &&\
 cp -v bin/faster2h.`hostname` /usr/bin/faster2h &&\
 cd / && rm -rf /setup
RUN adduser --disabled-password --gecos '' user
USER user
WORKDIR /home/user/

CMD ["/bin/bash"]
```

You see there that at the end of compilation, we `rm -rf /setup` to remove all source files.

The image can be used with `docker run --rm -ti -v %cd%/work:/home/user/work distrib:001` (from the [examples/distrib](#) directory), and launch as a test:

```
cd work
faster2h --conf=etc/the.config --map=etc/the.map --input=data/test_0001.fast
```

And it should work perfectly...

## Discussion

Now that we looked at a few example of how containers can be used to run your projects, let's consider the pros and cons.

Containers allow a controlled environment to be run in a consistent way one different platforms (Windows, MacOS, Linux, ...) with all the dependencies and needed libraries contained in the image. It's a great opportunity for open science, open source (even with the source code, having a ready-made environment where to run the code is nice). It is likely to be the basis of distributed grid job submission in the future.

On the other hand, one still have to install the proper container runner on the host machine, which may require special permission that normal users don't have. Using plenty of images and containers may accumulate *junk* (dead container, unused images, ...) on the host computer. Also, the average user may not know how to write a proper Dockerfile for his needs. Additionnaly, there are several container runners avaiable (docker, singularity, ...) and even if they have some cross compatibility, they are not used the same way.

## Perspectives

One thing to look forward in container runners close future is the distirbution of [Window's WSL2](#) (Windows subsystem for linux 2). As the first version relied on an emulated linux kernel, WSL2 will provide a native Linux kernel on the Windows host. This will speed up the execution, faciliate file access, ...

Large computing infracstructures such as [Open Science grid](#) and [EGI](#) are starting to integrate containers running in their job submission (more or less easily). It is certain than in the future, any job launch on a grid will include the name of an image to run the job on.

Finally, people developping code in [Github](#) or a [Gitlab](#) use docker images to run their [CI/CD](#) (*Continuous Integration/Continuous Deployment*) jobs.

## Conclusion

Containers offer the ability to define and distribute a consistent and uniform software environnement accross teams and machines in order to facilitate research work. The container images can be tailored to your own need and include all necessary software, libraries and dependencies needed to run the analysis work associated. Containers are ideal for deployment of code in multiple, parallel tasks that need to run the same environnement.