Software environment management

Alexander Held[1]

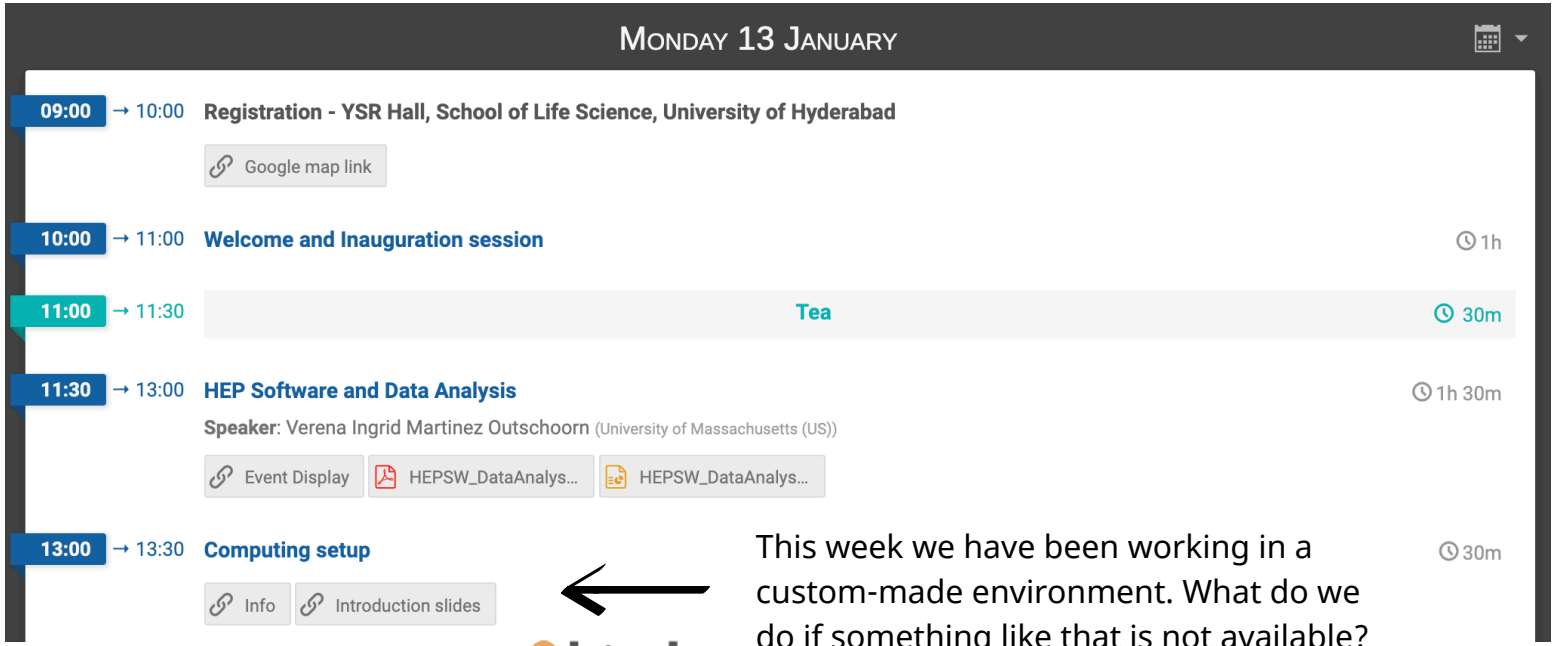
[1] University of Wisconsin-Madison

HSF India Hyderabad
https://indico.cern.ch/event/1394564/
Jan 16, 2025

This work was supported by the U.S. National Science Foundation (NSF) under Cooperative Agreements OAC-1836650 and PHY-2323298.

iris hep

Why are we only talking about this (only) now?



This week we have been working in a custom-made environment. What do we do if something like that is not available?

- Some slides are based on a **similar talk** from a previous workshop by **Gordon Watts** (link) — check those out as well!

A typical situation

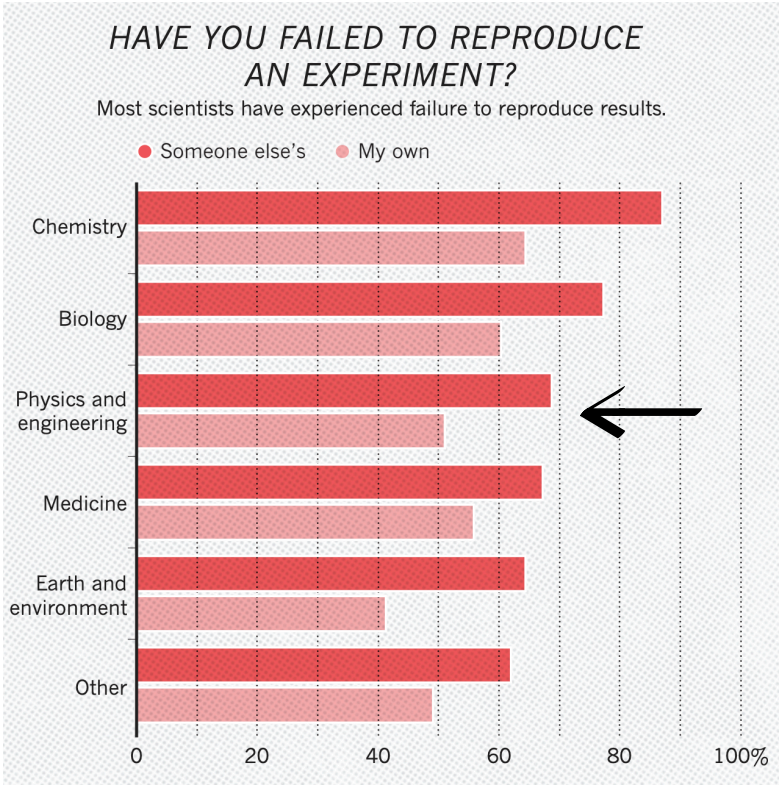
- You are a **new collaborator** in a group of people performing a **physics analysis**

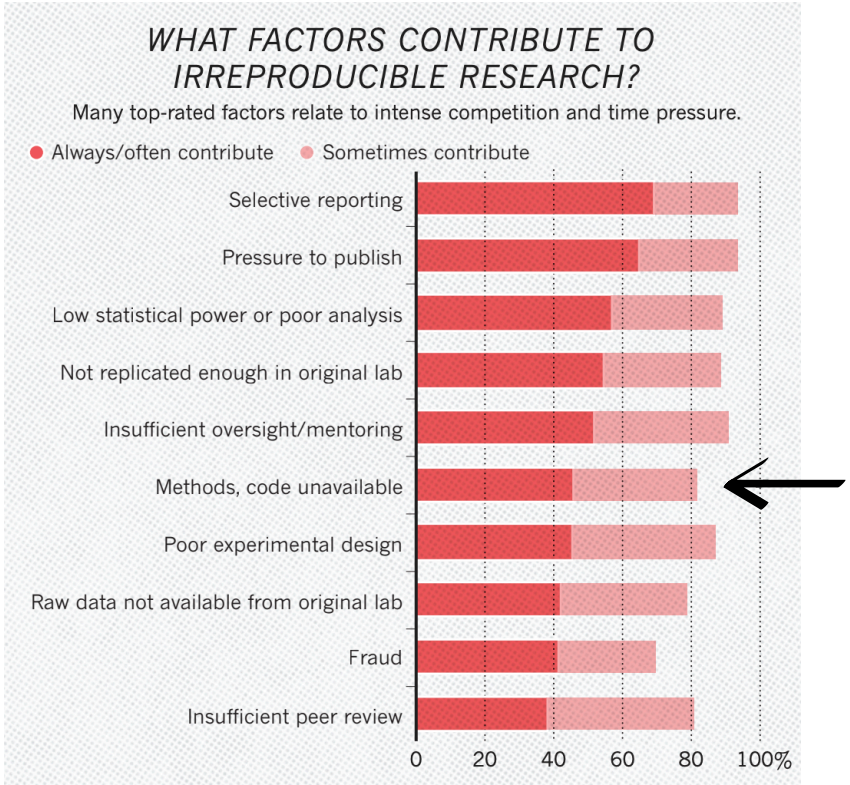
- To get started, you might need to find **answers to a few questions**:
  - Are there dedicated computing resources for me to use?
  - How can I access the data I want to analyze?
  - Is there specific software I need to use?
  - How do I create a software environment to perform my work in? ← *will look at some patterns for this today*

Best practices: reproducibility & reuse

Reproducibility in sciences

- From **"1,500 scientists lift the lid on reproducibility"** [Nature volume 533, pages452–454 (2016)]





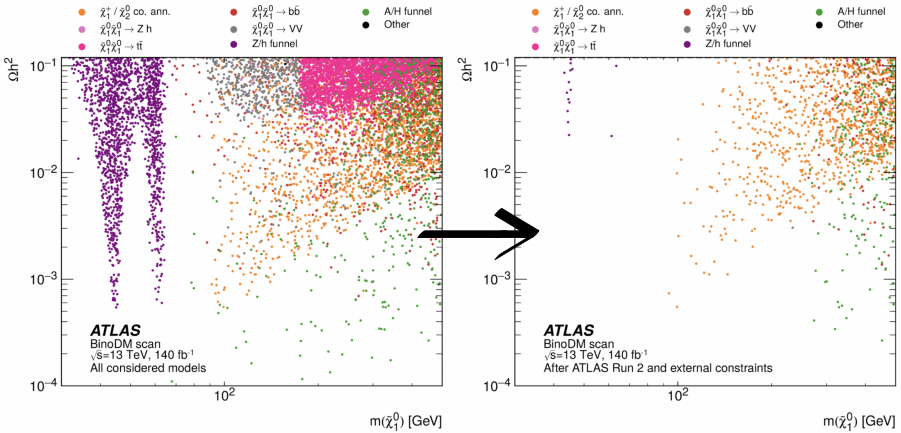
Reproducible research

- **Classification according to "The Turing Way"** [https://book.the-turing-way.org/]

		Data	
		Same	Different
Analysis	Same	Reproducible	Replicable
	Different	Robust	Generalisable

Reproducibility and reuse

- With **analyses preserved**, we can **reuse** them
  - further extend physics impact!



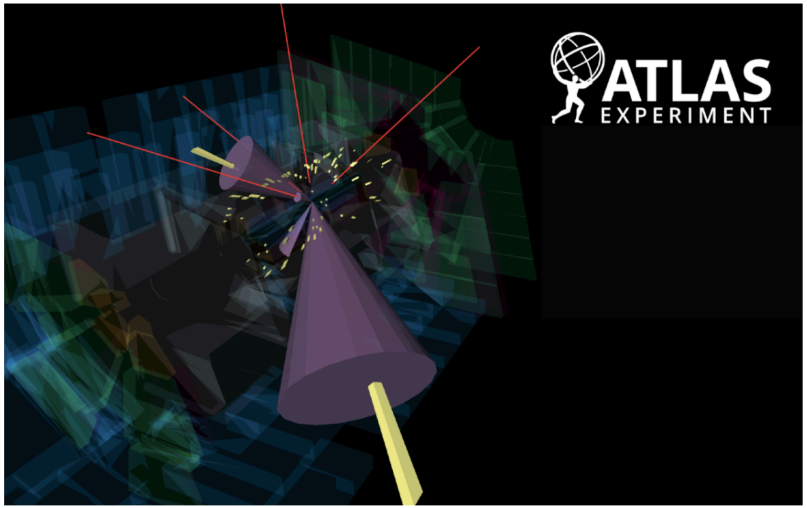
unexcluded supersymmetry models before (left) and after (right) the 2023 ATLAS pMSSM analysis

EP Newsletter of the EP department

NEWSLETTER NEWS ARCHIVE SEMINARS & COLLOQUIA NEW ARRIVALS

Extending ATLAS Physics Reach with Analysis Reuse Technology

Matthew Feickert (University of Wisconsin Madison) 10th Mar 2024



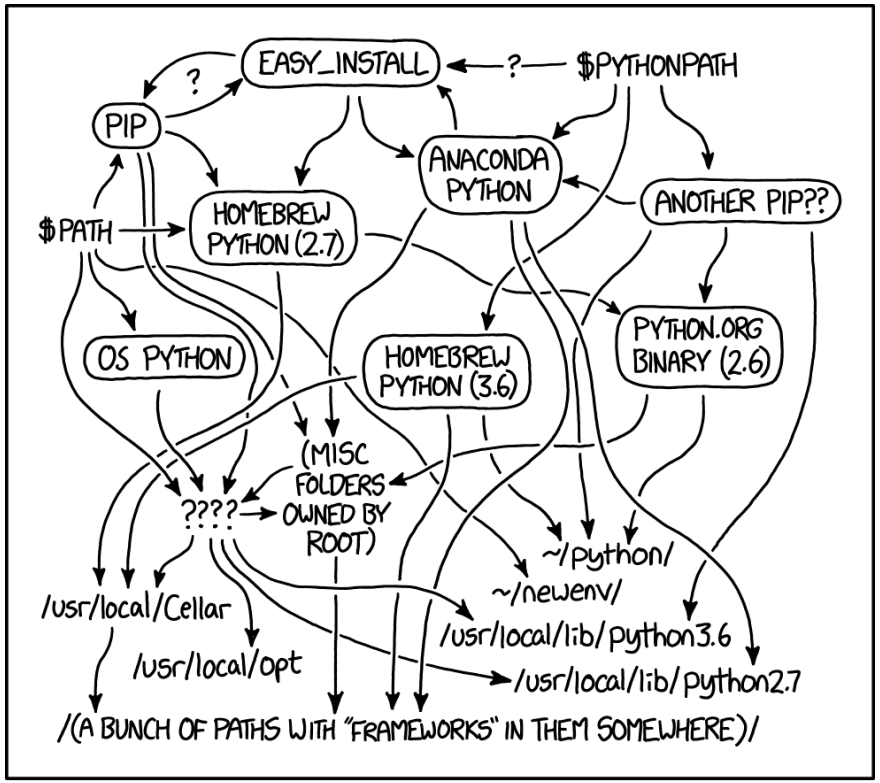
https://ep-news.web.cern.ch/content/extending-atlas-physics-reach-analysis-reuse-technology

Requirements for reusable data analysis

- Preservation of **input data**
  - see e.g. https://opendata.cern.ch/
- Preservation of **analysis code**
  - see e.g. https://zenodo.org/
- Preservation of **environment** to run software in
  - typically containers, see e.g. https://hub.docker.com/
- **Instructions for how to run** everything
  - ideally automated and ready-to-run, see workflow languages
- **Focus on environments** today, but all of these points are worth thinking about more deeply

Python environments

Python environment management



https://xkcd.com/1987/

MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Hands-on: dependencies and problems

- So far we have been using a **pre-configured software environment** designed to contain everything we need

- It is common to need functionality from a **libraries (= “dependency”)** which may not (yet) be available
  - Python has an extensive Standard Library but we already used a lot of additional libraries this week
  - we will now see how to handle dependencies in practice

- **Exercise**
  - if you do not yet have a BinderHub instance running, start one
    - https://binderhub.ssl-hep.org/v2/gh/research-software-collaborations/courses-hsf-india-january2025/HEAD
  - navigate to the `ManagingSoftware` directory and open the `networkx_visualization.ipynb` notebook
  - try running the notebook — what happens?

pip

- **pip** is the **official Python package manager** https://pip.pypa.io/en/stable/ (**pip** = **p**ip **i**nstalls **p**ackages)
  - install a package: `pip install package-name`
    - also from e.g. GitHub: `pip install git+https://github.com/org/name.git@branch_name`
  - update a package: `pip install --upgrade package-name`

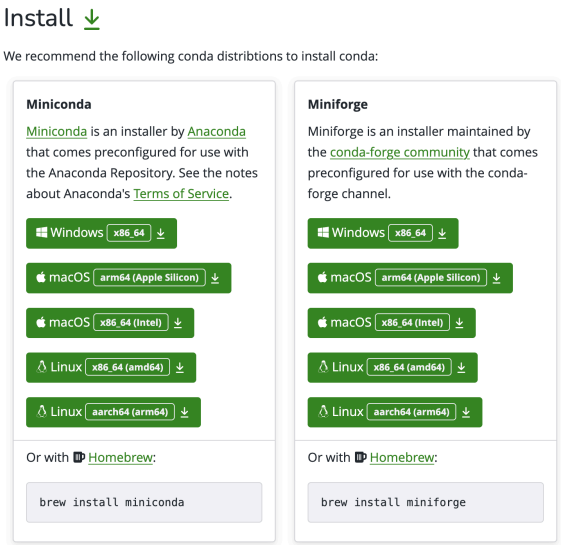
- You can also package up your own code and distribute it on the **Python Package Index (PyPI)**
  - see a tutorial at https://packaging.python.org/en/latest/tutorials/packaging-projects/
  - distributing packages like this is a great way to share code with others
  - putting your package on e.g. GitHub is already one way of distribution!

- Creating a package has gotten **much easier in past years**
  - at a minimum you need a `pyproject.toml` file as shown on the right
  - in practice you should add more metadata to this, see the tutorial

Conda

- **Conda** is a **package manager**, https://docs.conda.io/projects/conda/
  - Conda is general purpose and supports software beyond just Python
    - language-agnostic, cross-platform

- **Distinction with pip**
  - pip installs *Python* packages in *any* environment
  - Conda installs *any* package in *Conda* environments



- We generally want to use packages from the **`conda-forge`** (https://conda-forge.org/) channel
  - this channel is free to use (see also https://www.anaconda.com/blog/is-conda-free)
  - the Anaconda company also provides a `default` channel and generates revenue from that
  - use `Miniforge` (not `Miniconda`) and and you will be using the `conda-forge` channel by default

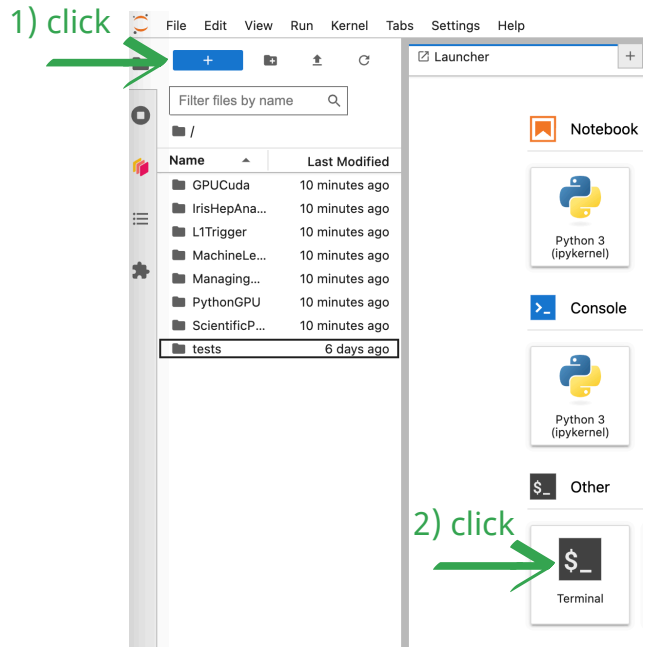
Hands-on with conda (1/2)

- Create and use a **new conda environment**
  - `conda create -n my-env python=3.12`
  - `conda activate my-env`
  - check what is installed in the environment
    - `conda list`
    - `pip list`

- Now try to **run the script**
  - `python networkx_visualization.py`

- Exercise: **make this work!**
  - use the `conda install` command to install missing dependencies

using the terminal here



Hands-on with conda (2/2)

- If your script works, create a **file listing everything in the environment** that you could share with others
  - `conda export -f environment.yml`

- Now **generate a new environment** from this file and ensure you can still run `networkx_visualization.py`
  - `conda env create -f environment.yml -n my-env-from-file`
  - `conda activate my-env-from-file`

- You could now keep this file around and give it to your collaborators to **share your software environment**
  - e.g. put it next to your code on GitHub

Comparing environments

- The software environment we have been using throughout this week is defined similarly with a **`environment.yml`** file
  - find it on GitHub: davidlange6/courses-hsf-india-january2025/blob/main/binder/environment.yml

- Which **differences** do you notice to your **environment file**?

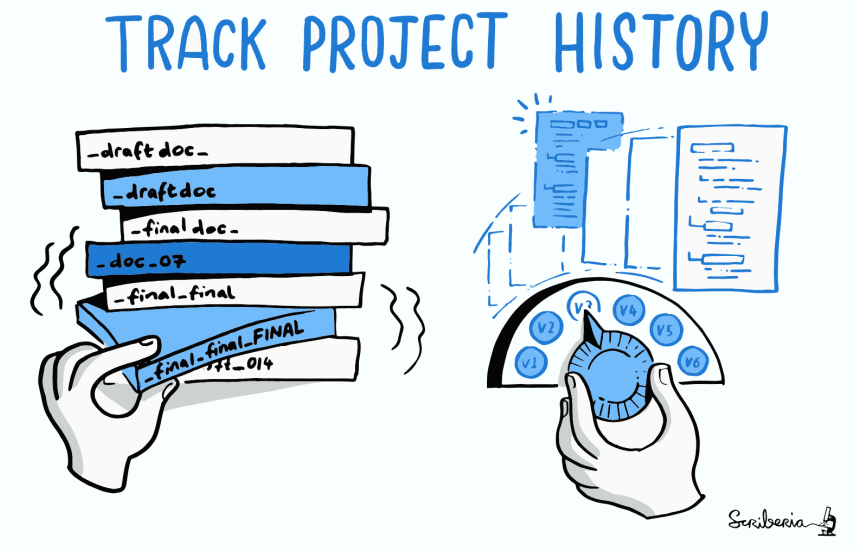
Code | Blame | 85 lines (82 loc) · 1.84 KB | Code 55% fast

```yaml
name: hsf-india
channels:
  - defaults
  - nvidia/label/cuda-12.4.1
dependencies:
  - python=3.10
  - libcurand
  - libcurand-dev
  - cuda-minimal-build=12.4
  - cupy
  - graphviz
  - nomkl #remove the large mkl lib stuff
  - pip
#  - emacs - broken after install, so just remove it
  - zstandard
  - zstd
  - clang-tools
  - fsspec
  - lz4
  - python-xxhash
  - pyarrow
```

Software versioning

- **Version control systems** can be used to track changes made to files
  - versions matter: new software features, bug fixes, interface changes …

- **Standard approach**
  - Git (https://git-scm.com/) to track software projects
  - GitHub, GitLab, … to develop & collaborate
  - deploy tagged versions of project to users



Versioning schemas

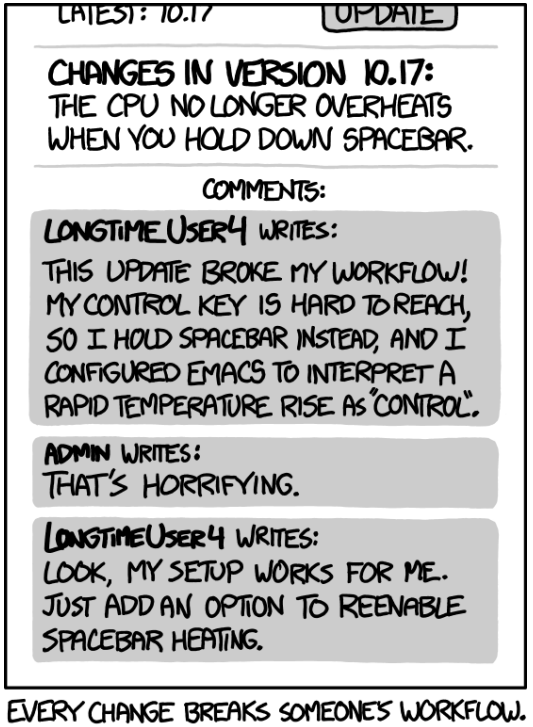
- Semantic versioning is a popular software versioning schema described at https://semver.org/

- Given a version number **MAJOR.MINOR.PATCH**, increment the:
  - MAJOR version when you make incompatible API changes
  - MINOR version when you add functionality in a backward compatible manner
  - PATCH version when you make backward compatible bug fixes

- There is **not always a unique correct way** to increment

- **Not every project** with versions that look like v1.2.3 **uses SemVer**

- **Other schemas exist**, e.g. CalVer (https://calver.org/)
  - e.g. coffea v2025.1.0



https://xkcd.com/1172/

SemVer examples (1/2)

- Given a version number **MAJOR.MINOR.PATCH**, increment the:
  - MAJOR version when you make incompatible API changes
  - MINOR version when you add functionality in a backward compatible manner
  - PATCH version when you make backward compatible bug fixes

- **Which version should we use for the code on the right?**

```python
# version 1.0.0
def my_function(number):
    print(f"{number}")
```

```python
def print_number(number):
    print(f"{number}")
```

SemVer examples (2/2)

- Given a version number **MAJOR.MINOR.PATCH**, increment the:
  - MAJOR version when you make incompatible API changes
  - MINOR version when you add functionality in a backward compatible manner
  - PATCH version when you make backward compatible bug fixes

- **How should we increment in both of these examples?**

```python
def print_number(number):
    print(f"{number}")
```

```python
def print_number(number, multiply_by=5):
    print(f"{number*multiply_by*1.1}")
```

```python
def print_number(number, multiply_by=5):
    print(f"{number*multiply_by*1.1}")
```

```python
def print_number(number, multiply_by=5):
    print(f"{number*multiply_by}")
```

Version conflicts and isolating environments

- You might have **conflicting version requirements** of your dependencies
  - example: one library which requires `uproot<5` and another which requires `uproot>=5`
  - this is solved by creating dedicated, isolated environments for your projects
  - we already created new isolated environments with conda

- **Virtual environments** are a **lightweight** way to manage isolated Python environments
  - you will need a Python interpreter already available to start with
  - create a new virtual environment with `python -m venv .venv`
    - the environment will be located in the folder `.venv`
  - activate the environment
    - `source venv/bin/activate` on Linux / macOS
    - venv\Scripts\activate on Windows
  - you can now install packages with pip and they are available within just that environment

Library vs application

Library

reusable code, functionality to use in your projects

example: dependencies of the cabinetry library

```
dependencies = [
    "pyhf[minuit]~=0.7.0",
    "boost_histogram>=1.0.0",
    "hist>=2.5.0",
    "tabulate>=0.8.1",
    "matplotlib>=3.5.0",
    "numpy",
    "pyyaml",
    "iminuit",
    "jsonschema",
    "click",
    "scipy",
    "packaging",
]
```

loose version specification: avoid version conflicts

Application

specific code for a particular purpose

example: dependencies for a specific data analysis pipeline

```
1     aiofile==3.8.8
2     aiohappyeyeballs==2.4.0
3     aiohttp==3.10.5
4     aiohttp-retry==2.8.3
5     aiosignal==1.3.1
6     annotated-types==0.7.0
7     anyio==4.4.0
8     argon2-cffi==23.1.0
[...]
200   zipp==3.20.1
```

precise version specification: ensure reproducibility

Other tools in this space

- **pipx** allows you to run Python applications in isolated environments
  - great e.g. for command-line tools like `uproot-browser`
  - documentation: https://pipx.pypa.io/

- **uv** is a drop-in **replacement for pip** with a range of additional features
  - written in Rust and extremely fast
  - documentation: https://docs.astral.sh/uv/

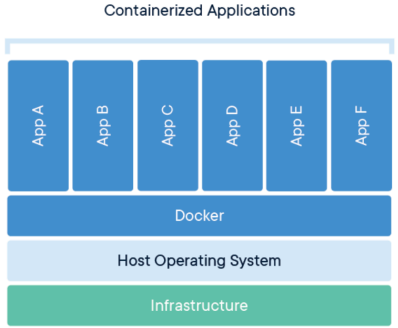
- **pixi** is a **package manager** similar to conda
  - slightly different philosophy: focused on projects instead of environments
  - similarly to uv: written in Rust and fast!
  - documentation: https://pixi.sh/
    - transitioning from conda: https://pixi.sh/dev/switching_from/conda/

Containers and images

Containers and images

- If we want truly **reproducible workflows**, the concepts we have covered so far are insufficient
  - we might have a huge stack of software dependencies, far reaching beyond just Python libraries
  - "it works on my machine" but not on another can still happen frequently and be difficult to debug / fix

- Containers provide **snapshots of full project environments to deploy them as production environments**
  - this can include libraries, environment & system settings, specific files you might need
  - you can share this environment with others: can significantly lower barrier to entry
  - the environment is isolated and can be reset in case something breaks

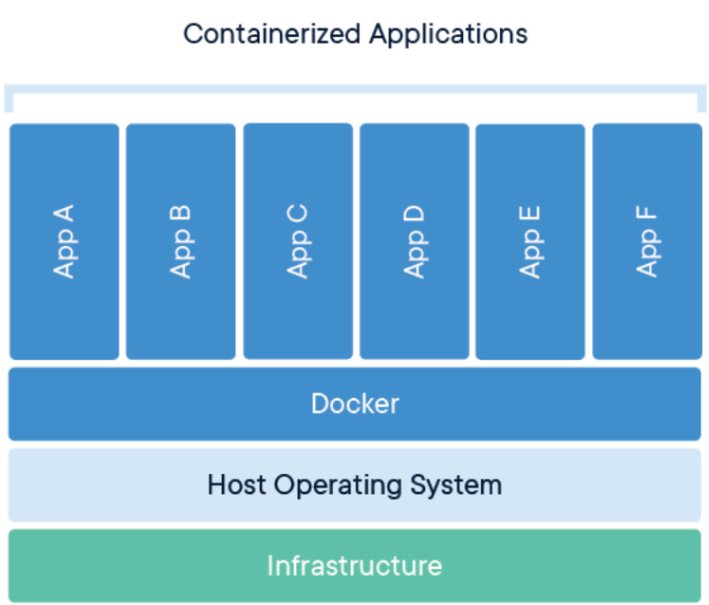


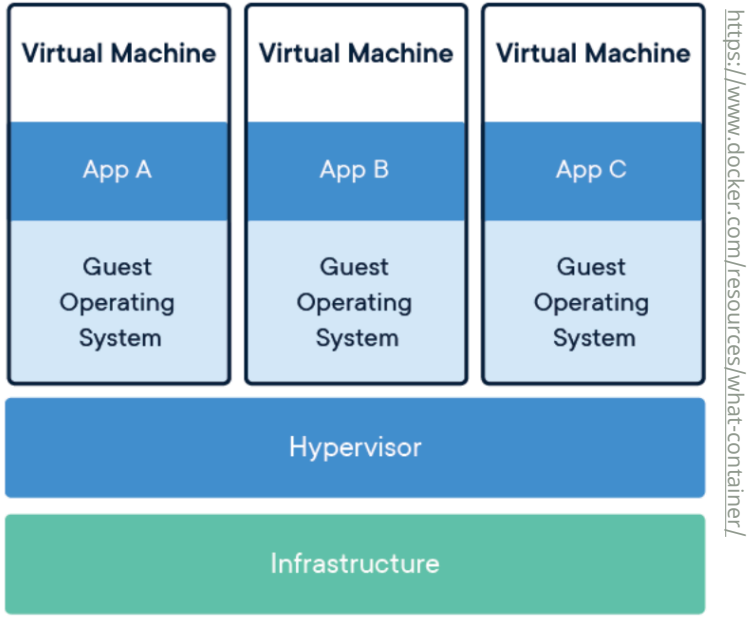
- **Images** package up everything needed to run a container (which adds state)

- Focus today on **Docker**, a very popular containerization tool
  - many others exist, see in particular Podman as a drop-in replacement and Apptainer (formerly Singularity)

Containers and virtual machines

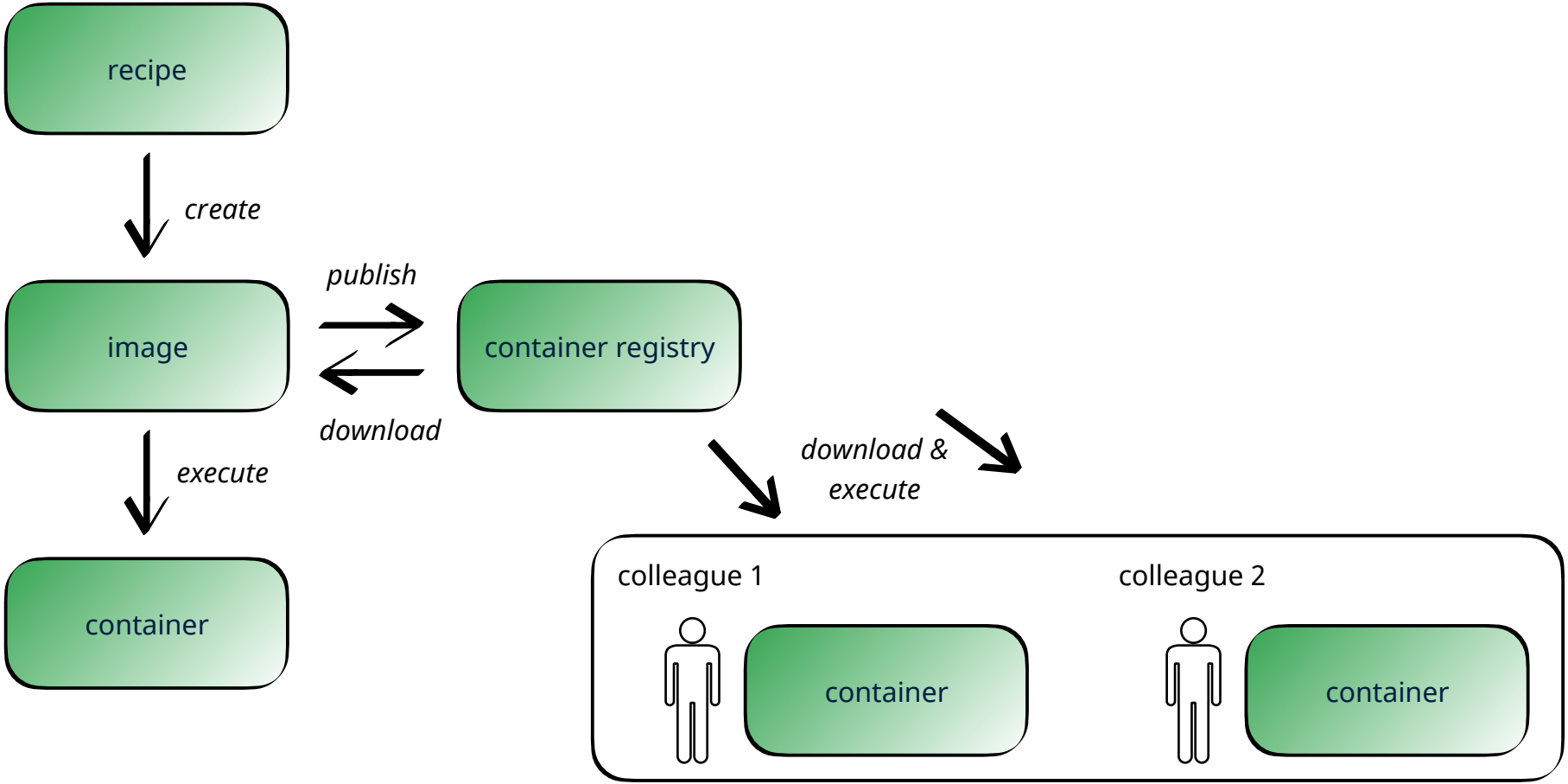
- **Containers** share the system kernel of the **host operating system**
  - generally better performance and more lightweight than virtual machines



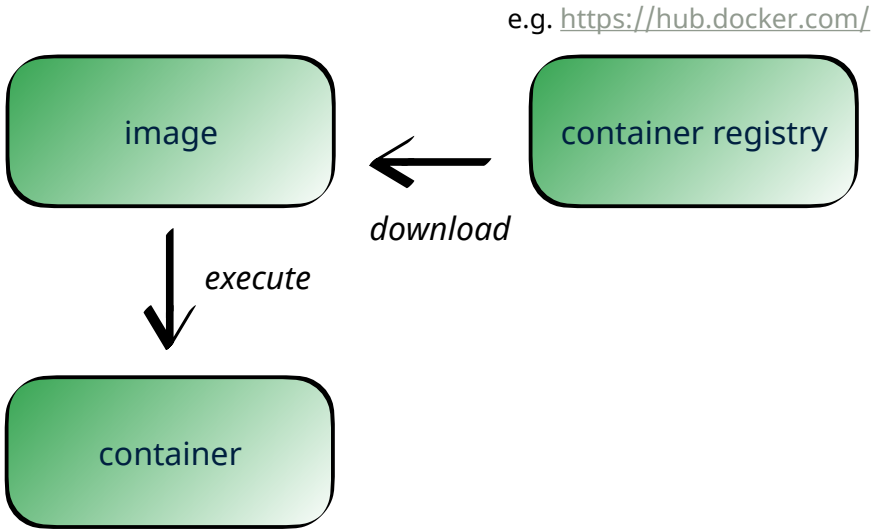


https://www.docker.com/resources/what-container/

General workflow



Using a container



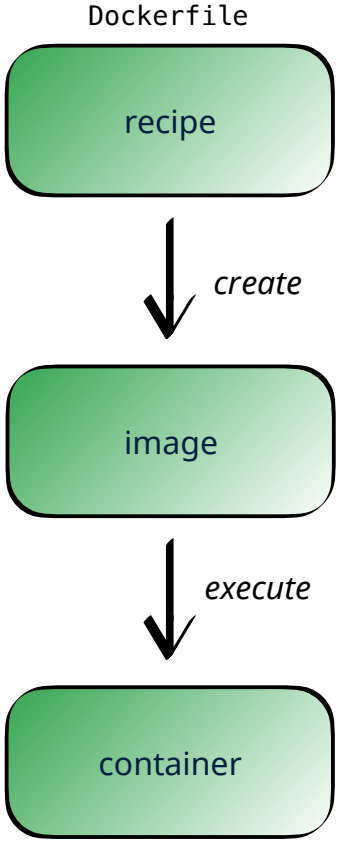
- **`docker image ls`** shows images you might already have
- download more with **`docker pull`**
  - example: `docker pull python:3.12-slim`

version, default "latest"

- Run container: **`docker run -it python:3.12-slim`**
  - `-it` for interactive session with terminal access
  - we land in a Python interpreter session
  - add e.g. `/bin/bash` at the end for a bash session
  - `exit` to leave the container

- **`docker container ls`** shows containers (optionally **`-a`**)
  - from here can pause / stop / remove containers
  - can also `attach` to running containers

Creating our own container



- A **`Dockerfile`** is a recipe for creating a new container:

build upon existing image → `FROM python:3.12-slim`
interactive session starts here → `WORKDIR /work`
copy file into image → `COPY requirements.txt .`
install dependency → `RUN python -m pip install -r requirements.txt`
start up bash by default → `ENTRYPOINT ["/bin/bash", "-l", "-c"]` `CMD ["/bin/bash"]`

- Create image: **`docker build -t my-image .`** (can be published with **`docker push`**)
  - it will now show up in your list with name “my-image” and default tag “latest”

- Test that a container using the new image can use the uproot library
  - mount `create_file_uproot.py`, launch with `-v host_path:container_path`

From BinderHub to your laptop & summary

Managing software environments on your laptop

- We started with exercises that all run on the **provided BinderHub resources**
  - this greatly simplifies this lecture: same starting point for everyone!

- If we have time left at this point (or as an optional exercise for afterwards): **repeat the exercises on your own laptop!**
  - we focused on popular tools, so lots of documentation and help available on the internet
  - worth having a look at the various tools the exist and find the best for your use case

Summary

- A range of approaches exists to **create reproducible and isolated environments**
  - Python-only? → virtual environments
  - other packages as well? → conda environments
  - package up full stack of software? → containers

- The **right approach** depends on your specific **use case and requirements**
  - a large range of widely used tools exists, often with great documentation (see links throughout these slides)

- **More recommended resources**
  - HSF training: Intro to Docker and Podman https://hsf-training.github.io/hsf-training-docker/
  - The Turing Way https://book.the-turing-way.org/
  - Python Packaging Authority: Python packaging user guide https://packaging.python.org/
  - Scientific Python topical guides: https://learn.scientific-python.org/development/guides/

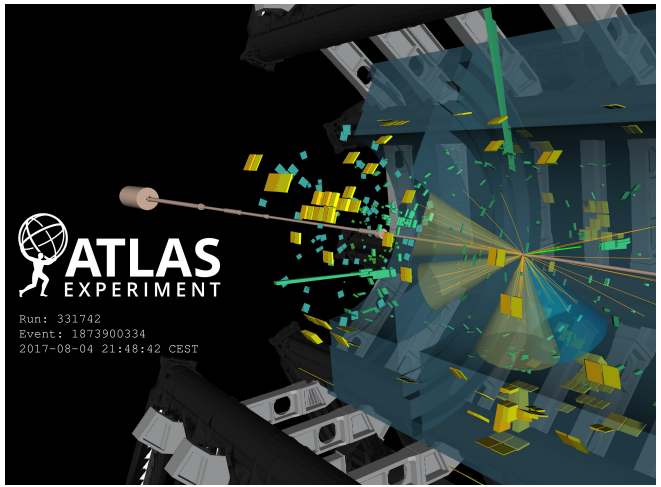
Backup

The big picture: collision to publication

1) collide protons

https://natronics.github.io/science-hack-day-2014/lhc-map/

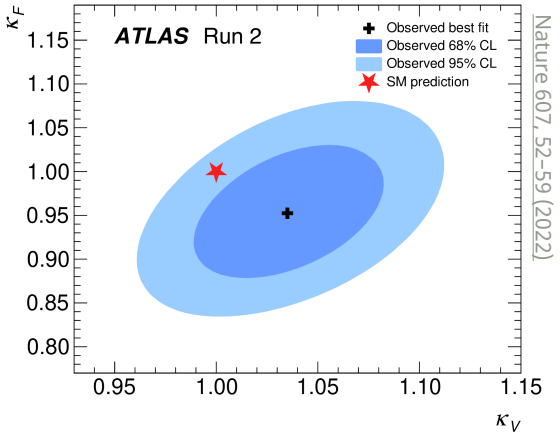
2) observe remnants



Phys. Lett. B 784 (2018) 173

O(100 M) files with
O(100 B) events
(data + simulation)

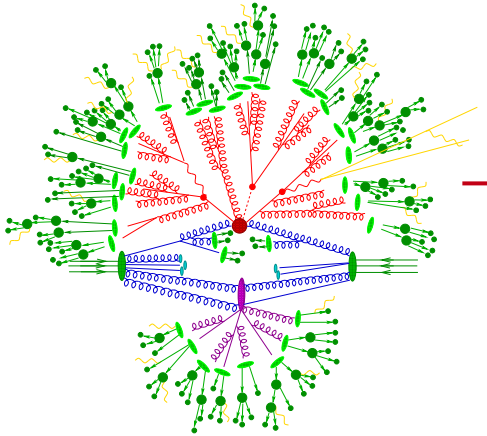
3) infer nature



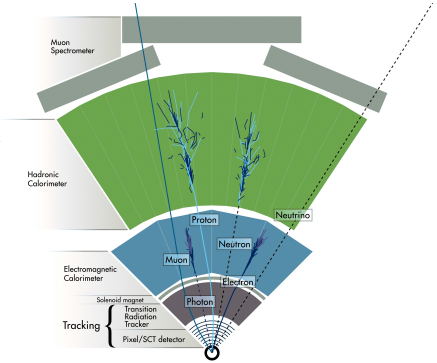
O(1000) sources of uncertainty

End-user physics analysis

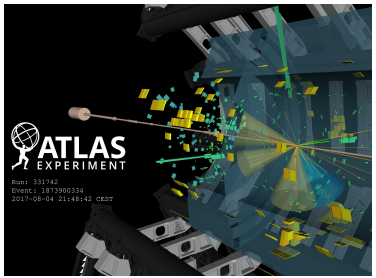
event generation → **detector interaction** → **object reconstruction** → **“analysis”**



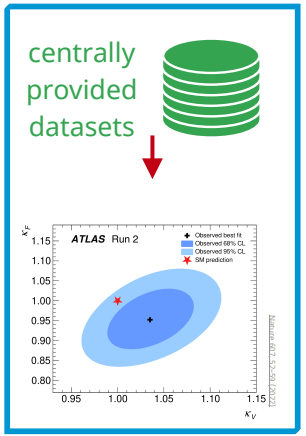
JHEP 0902 (2009) 007



CERN-EX-1301009



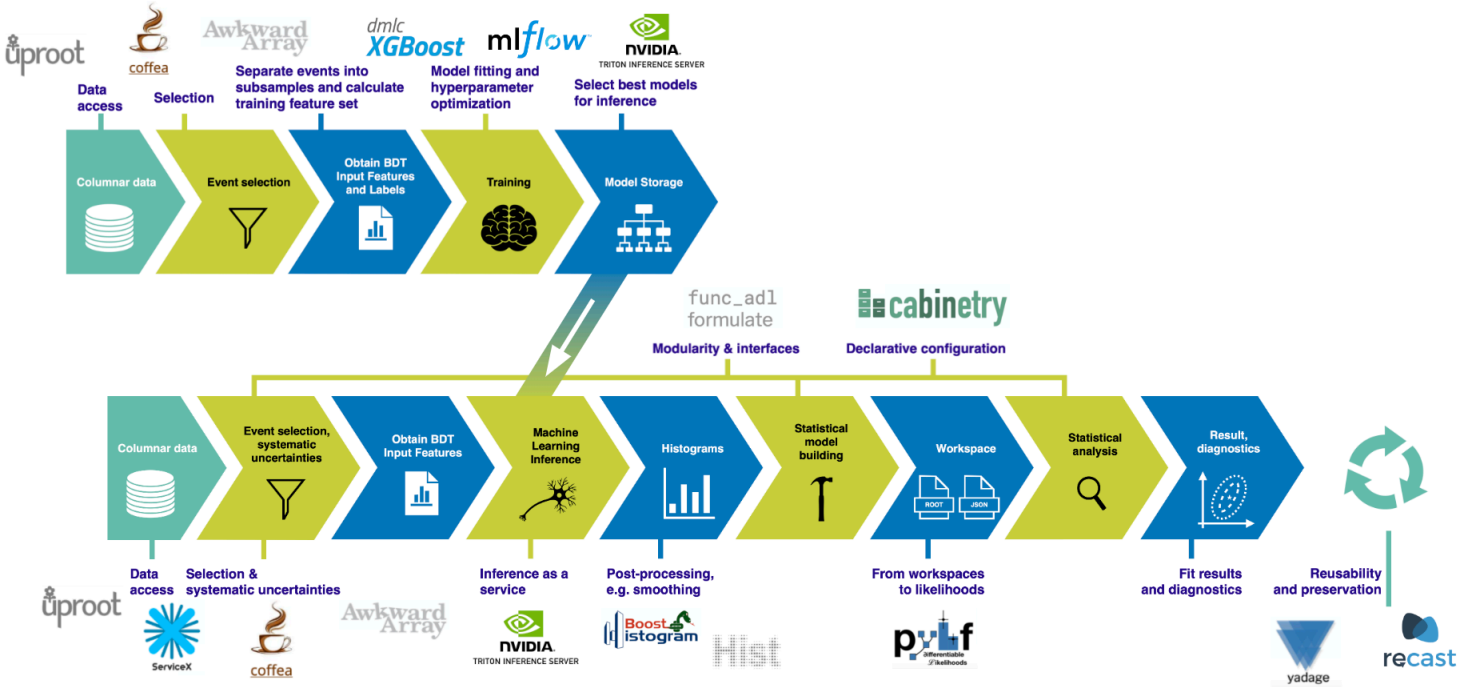
Phys. Lett. B 784 (2018) 173



- “Analysis” in practice: the whole pipeline **turning centrally provided datasets into results for a paper**
  - iterative process, optimize, debug, validate: low latency means faster time-to-insight

Preparing for the HL-LHC: the AGC project

- **Analysis Grand Challenge (AGC) project** (https://agc.readthedocs.io/) defines physics analysis task for HL-LHC R&D
  - multiple implementations available: reference with coffea, RDataFrame, Julia, columnflow



example pipeline in reference AGC implementation