Computing Methods in Particle Physics

Karolos Potamianos, Elisabeth Schopf

February/March 2022

The lecture slides and content are heavily based on what originally has been put together by Matthew Bass and modified by Giacomo Artoni. Huge credit goes to them!

Lecture Overview

1. **Introduction - today**
   a. Commonly used computing tools
   b. Setting up the computing environment for this course
   c. Python, numpy, matplotlib examples
2. Bayesian Inference & Fitting
3. Classification problems & machine learning
4. Deep Learning

Each lecture will have some slides + practical, hands-on examples

→ Aims: Introduce a broad range of computing concepts and tools commonly used in particle physics with a focus on machine learning. We can't possibly cover everything in this lecture but hope to give an overview and starting point for more in depth studies.

Materials and contact

For any questions feel free to contact us: karolos.potamianos@cern.ch, elisabeth.schopf@cern.ch

All lecture material can be found here: https://gitlab.com/oxford-physics/cmpp

Further reading:

- "Effective Computation in Physics", A. Scopatz, K. D. Huff
- Frequentism and Bayesianism: A Python-driven Primer
- HSF analysis essentials for HEP data analysis
- TMVA user's guide
- scikit-learn user's guide

Particle Physics and Programming

In particle physics you are most likely working with large data sets and collaborating with others → efficient data handling, analysis and collaborative software development are key!

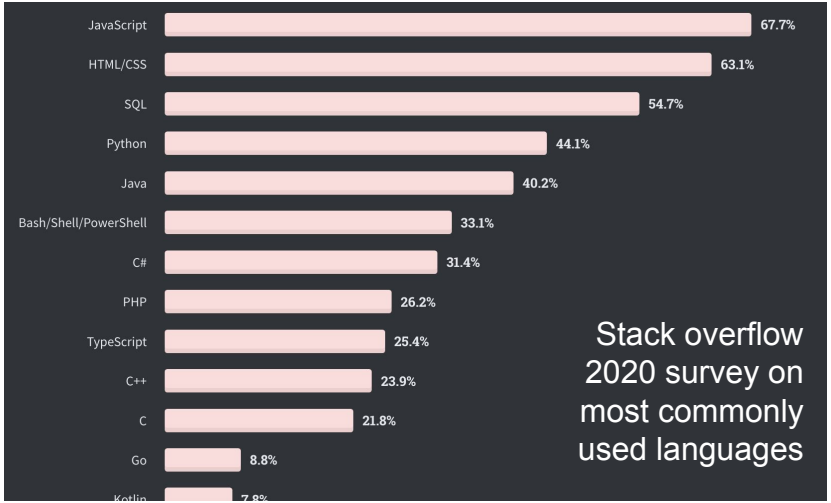
Tools you likely will use in one way or another:

- Programming languages
- Linting & debugging
- Shell environments & shell scripts
- Plotting tools
- Data management tools
- Version control
- Build systems

Programming Languages and Coding

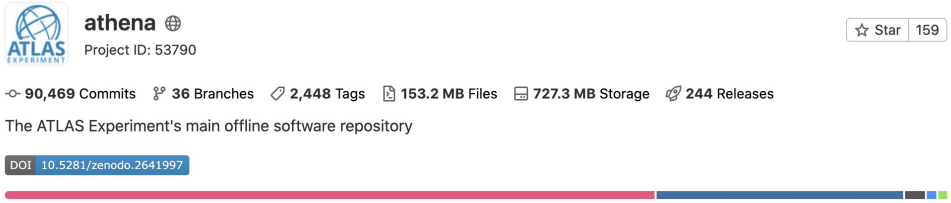
Most commonly used languages

There is a breadth of programming languages out there. In particle physics (analysis) C++ and python are most commonly used. In technical settings you might also encounter others (java, SQL, etc.)



+…

ATLAS reconstruction software



68.4% C++ 26.0% Python

For data analysis at the end of the chain, i.e. after reconstruction, pre-processing of the data, python is becoming increasingly popular

- Python requires no extra compilation step
- It interfaces to many external libraries easily, e.g. ROOT, sci-kit learn, etc.

You will likely share your code and its development with others and need to rerun pieces of code after many months/years of not using them, e.g. re-running plots for your thesis → Reminder of best practices (not only for python):

- **Give variables/functions sensible, self-explanatory names**
- **Take the time to write comments in the code**

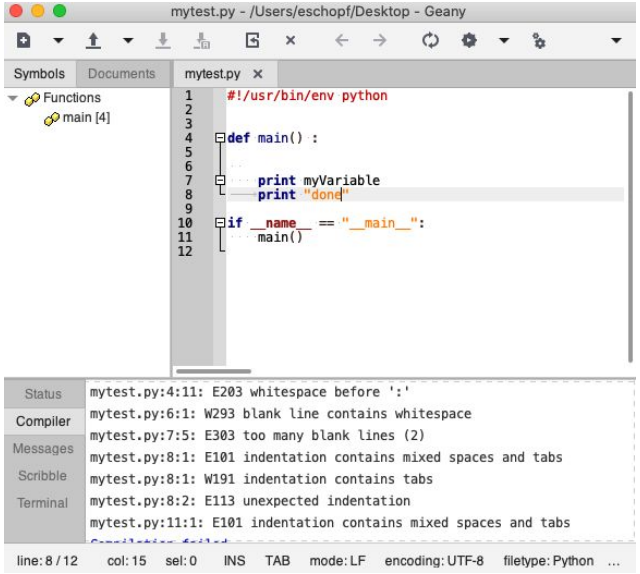
Tools for inspecting your code: lint & syntax check

A linter checks your code for **coding style issues** → helps to maintain good readability of the code and cross-editor compatibility

Most editors - geany, emacs, etc. - come with built in linters

Many editors also allow you to **check code syntax for errors**

- **Attention:** depending on compiler environment and language version the errors/suggestions you get locally in your editor might not be the same as in your final environment
- **Good practice to always test the code in the place where it is intended to run!**



Tools for inspecting your code: debuggers

UNIVERSITY OF OXFORD

Debuggers can be very powerful tools to **inspect the logic of your code and help finding unexpected errors or output**

A debugger allows to step through the code line by line, evaluate expressions at each point in the code, etc.

- Some editors come with built in debuggers
- Python also has its own debugger pdb that comes with (most) python installations
- For C++ use, e.g. gdb

Debugging example

```
(base) eschopf@ppmaclap01 Documents % python -m pdb ~/Desktop/mytest.py
> /Users/eschopf/Desktop/mytest.py(4)<module>()
-> def main() :
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(13)<module>()
-> if __name__ == "__main__":
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(14)<module>()
-> main()
(Pdb) step
--Call--
> /Users/eschopf/Desktop/mytest.py(4)main()
-> def main() :
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(6)main()
-> counter = 0
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(7)main()
-> if counter > 0 :
(Pdb) p counter
0
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(9)main()
-> print (myVariable)
```

Stepping through the code

Evaluate value of counter

mytest.py

```
1   #!/usr/bin/env python
2
3
4   def main() :
5
6       counter = 0
7       if counter > 0 :
8           print ("found event")
9       print (myVariable)
10
11      print ("done")
12
13  if __name__ == "__main__":
14      main()
15
```

Tools for inspecting your code: profiling

If performing millions of time, computing and memory extensive operations

- Processing millions of data events
- Applying complex corrections
- Handling multiple copies of objects
- …

it might be useful to understand **which parts of the code consume most resources** to potentially optimise in that regard

There are open source tools available, e.g. valgrind, to do so

Version control: keeping track of code developments

Version control tracks and integrates changes to your code. In its core it allows to:

- Keep track of changes made by providing some form of change log/history
- Integrate (“merge”) code changes into existing code
- Revert changes or go back to previous code versions
- Create version tags of the code in a specific point in time/development

For common code and code used to produce public results version control is **a must to ensure reproducibility and be able to recall the logic that went into developing** the analysis.

Keeping your personal code under version control is a good habit too!

Version control software

One of the most popular version control softwares is git:

- Github is a popular place to host software
- Gitlab is another one and extensively used for CERN based projects

Other version control software are svn, bazaar, etc.

Build systems

Complex (C++ based) analysis frameworks often require a **build system that manages the compilation and takes care of dependencies and environment setups**.

It also handles software version requirements, cross-platform issues, etc.

On a small scale often having a **Makefile** is sufficient for more complex software build systems like **CMake, CMT, MRB**, etc. are available

Usually your experiment's software tools come with their build system and you probably don't need to set it up yourself

Data Handling and Visualisation

Data Management

Experiments all have their specific data management strategy - mostly centered around ROOT.

In most cases data is provided in an already pre-processed and machine readable format and you will just perform specific higher-level tasks on them and visualise the output.

In cases where you need to process raw data yourself or produce simulations you can in most cases follow your experiment's recipes (and in an ideal case: tutorial)

- **Metadata is important for reproducibility** since it keeps track of what happened to the data, which software versions it was processed with, etc.

Data Management Tools

The general idea of all data management tools is to **store data in a table-like format while preserving relationships** between fields

Many database tools/languages available: SQL (popular in industry), Pandas (for python), etc.

ROOT (often used in particle physics):

- TTree class with branches to store data (ints, floats, etc., but also vectors or physics objects as implemented by ROOT, e.g. TLorentzVector)
- Storage of custom C++ objects (e.g. reconstructed “Electron” objects)
- Storage of analysis objects (histograms, canvases)

It is useful to know other ones than ROOT because for interfacing to external tools, e.g. machine learning tools, it is sometimes necessary to convert data back and forth

Shell Scripts

Easy way to **automate simple, repetitive tasks**, like:

- Executing a sequence of commands, e.g. setting up your environment
- Renaming/copying files following a specific pattern
- Submitting jobs to a batch system
- …

and allows to interact with Unix & experiment specific tools. Often faster than writing and executing a python script

Most common scripting language: **Bash** (also common zsh)

```bash
#!/bin/bash

ID=0

# make your life easier by not typing out long strings
if [ $ID = 0 ]
then
  IDSTRING="AnalysisZero"
elif [ $ID = 1 ]
then
  IDSTRING="AnalysisOne"
fi

# do something with files in specific cases
if [ $ID == 0 ]
then
  rm ${IDSTRING}_*categoryXY*.txt
fi

# do something with files
for file in ${IDSTRING}*txt
do
  cat $file >> allData.txt
done
```

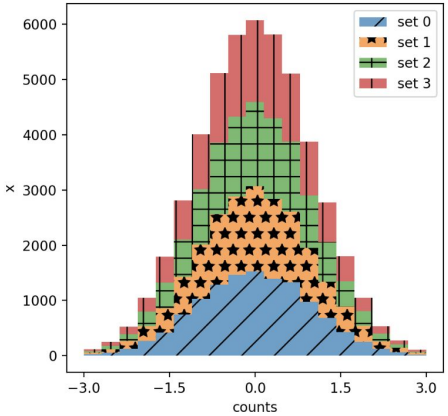
Plotting Tools

There are ample tools available to visualise data!

Which one is most convenient to use might also depend on the format your data is provided in and existing tools (however, it's always good to know what's out there)

- ROOT: very common in particle physics; pyROOT interfaces to python
- Matplotlib: visualisation library designed for python
- Gnuplot: command line driven visualisation tool
- …

All of them offer wide range of plotting types and options



Provides a wide array of functionalities tailored towards particle physics needs:

- Plotting (e.g. TH1 - 1D histograms, TGraph - graphs, etc.)
- Data management (e.g. TTree)
- Commonly used objects (e.g. TLorentzVector - four momentum vectors), mathematical functions (see e.g. TF1 formula class) and operations
- Fitting (RooFit)
- GUI (see TBrowser)
- Machine learning (see TMVA class)

It is possible to install ROOT on your computer; if you are running on a particle physics computing cluster (e.g. Oxford PPUnix or CERN lxplus) most likely it is already installed there

Summary

There are lots of open source computing tools and languages available to analyse, handle and visualise data; which one is best to use often depends on what you want to achieve and general practices in your experiment

How to get started on a project:

- Think about your data sources and analysis needs
- Setup version control or check-out and setup version controlled experiment specific framework
- Make sure to document (in-line comments, READMEs, docstring for python, doxygen, etc.)
- Start coding and make frequent commits

Setting up your Environment for this Course

Operating systems and computing environments

Any - Windows, Linux, MacOS - will work for this course and data analysis in general.

If specific software requires a specific operating system it is possible to virtually run it (virtual machines/emulators), e.g. VirtualBox a virtual machine host tool

If you don't want to pollute your computer with software or run a virtual machine you might also choose to not work on your computer directly but instead:

- ssh to Oxford's or your experiment's/lab's computing cluster
- Work on an online hosted computing environment, e.g. google colaboratory, CERN Swan, etc.

Suggested exercises for today's lecture

1. Set-up your computing environment (see next slides)
   a. It is always good to have a minimal working (python) set-up on your computer
   b. For this course we will mostly use jupyter workbooks, which will not run from the PPUnix or lxplus computing clusters
   c. For this course we have at least one example with ROOT, which requires connection to PPUnix
2. Get the matplotlib, numpy and pandas examples and check if they are working in your setup
3. Set-up a github/gitlab account (if you don't have one yet) and create a new project for this lecture to house your jupyter notebooks and changes
4. Get familiar and play around with these examples and tools :-)

Do not hesitate to ask in case something is unclear or not working and we will try to help!

Set-up for your computer: installing python

Check if python is already installed, e.g. type in your terminal `which python` (if yes, you can type `python -V` to get the python version number you are using)

If you need to install python, would like to update to a newer version or add python libraries that are not part of the default installation:

- "Raw" installation directly from python (Linux, Windows, Mac) and add packages by e.g. downloading from gitlab (not recommended)
- Using an installation environment that also can help downloading additional python libraries and takes care of dependencies, e.g.:
  - Anaconda*
  - Homebrew
  - pip* (needs existing python installation but comes by default with newer python versions)

*I can personally recommend

Adding additional packages/libraries to python

Useful things to add for this course (and also in general):

```
numpy scipy ipython ipython-notebook matplotlib pandas pytables
nose setuptools sphinx mpi4py seaborn pymc pymc3
```

They can be installed via:

- `pip install <pkg/lib>` (or `pip3` instead of `pip` for python version 3.*)
- `conda install <pkg/lib>`
- `brew install <pkg/lib>`

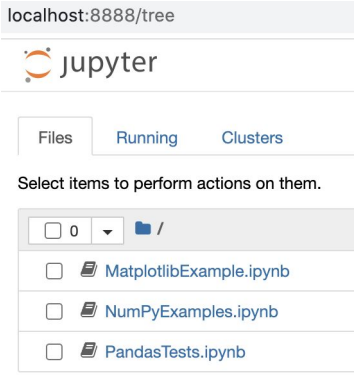
Not all packages are available in homebrew or anaconda default locations

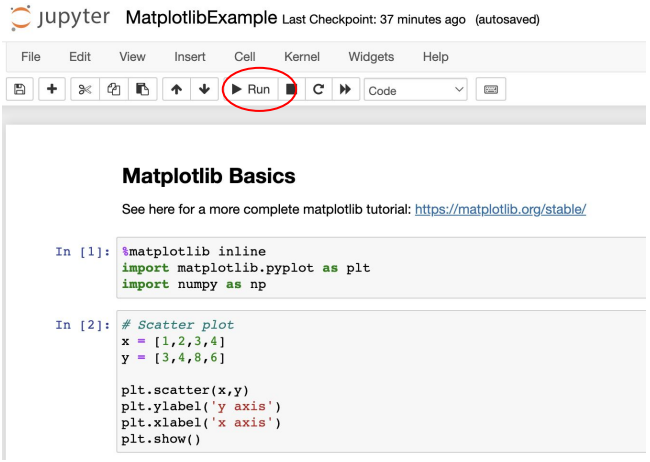
- So far I have made the best experience with pip/pip3
- Anaconda has the advantage that it allows to install libraries/packages from other sources (browse available sources for packages here)

jupyter notebooks

Jupyter allows to interactively work with python and provides immediate feedback, very useful for tweaking and fine-tuning!

Typing `jupyter notebook` in the terminal will open a browser session including all files and sub-directories of the current directory you are in





You can define blocks of code and run through them step by step and separately (attention with inter-block dependencies!)

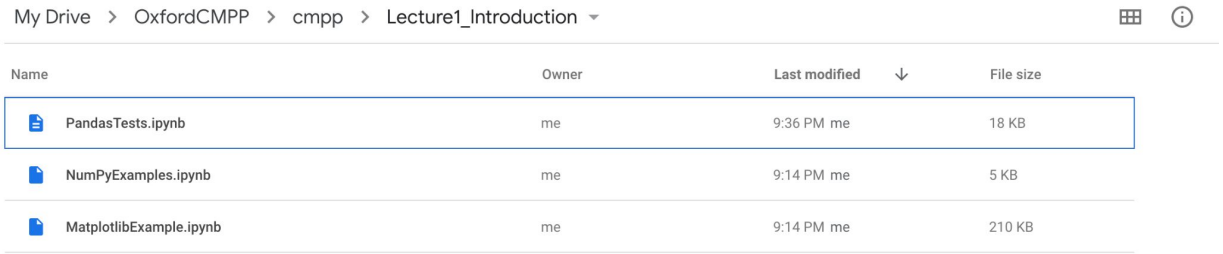
Once you are done hit “Logout”/“Quit” in the browser or close the program in the terminal where you started the session

P.S.: Copy-pasting the code into a *.py file will give you a python script that you can execute on your machine without jupyter

Google Colaboratory

Colab is a Python development environment that runs in the browser using Google Cloud (for free) ; it's also a Jupyter Notebook.

- To download the repo into your Google Drive, use [this Notebook]
- Once you ran it (you'll need to give it access to your Drive), just go to My Drive > OxfordCMPP > cmpp and click on any of the Notebooks

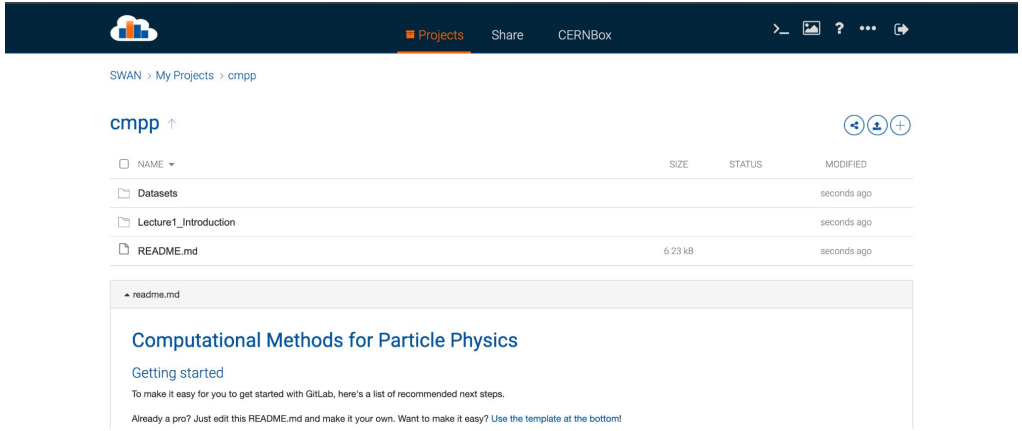


CERN Swan

CERN offers a **S**ervice for **W**eb based **AN**alysis (SWAN): https://swan.cern.ch

- Note that you need a CERN account to use it; it can be a lightweight account.

Once your session is started, and you've chosen the configuration (default is OK), you can use the "Download from git" button (see right) and point to https://gitlab.cern.ch/oxford-physics/cmpp.git



Docker

Docker is a system to manage and run containers. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

Very flexible and allows running from any machine with an internet connection and Docker installed; to get Docker, check out https://docs.docker.com/get-docker/

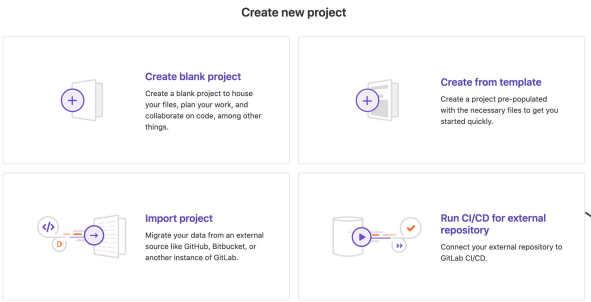
```
# Clone the repo
git clone https://gitlab.com/oxford-physics/cmpp.git
# Build the image (once or at every update), from the cmpp folder
docker build -t cmpp docker
# Run the container
docker run --rm -it -p 8888:8888/tcp -v $(pwd):/cmpp cmpp:latest root
--notebook
# Go to http://localhost:8888 and enter the code from the Docker container
```

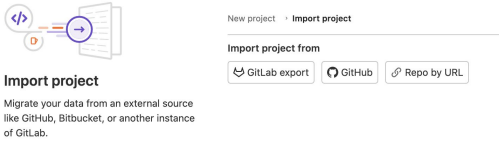
Github/Gitlab

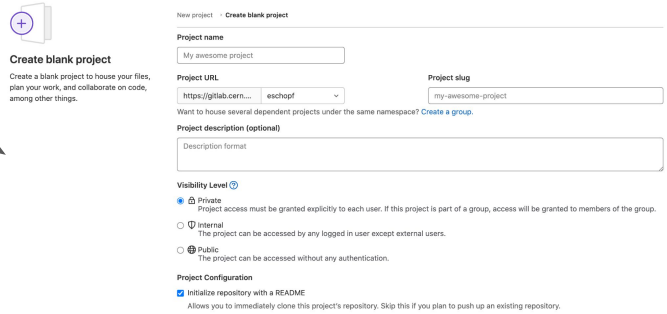
Github and gitlab offer free accounts for non-commercial use

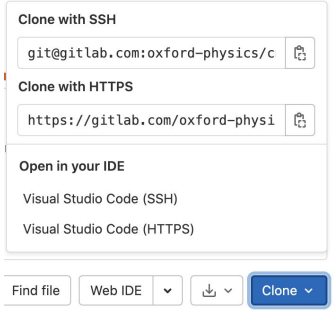
You can sign up e.g. via an existing google account or if you have a CERN account it should be connected to CERN's instance of gitlab

For more info, see e.g. this [git tutorial]









Gitlab example: creating a new project or import an existing one

Check out the project with `git clone <path>`

User guides for libraries used in today's exercises

Numpy

Matplotlib

Pandas