RooFit - Pythonic interaction with the RooWorkspace

About me:

Name: Yash Solanki
University: Indian Institute of Technology, Delhi
Major: Engineering Physics / **Minor:** Computer Science
Email: 252yash@gmail.com
Github: @yashnator
Linkedin: @solyash
Timezone: IST (UTC +5:30)
Language: English

Project details:

Organisation: CERN-HSF

Mentors:

- Jonas Rembser
- Lorenzo Moneta

Project duration: 350 Hours

Synopsis:

RooFit is a statistical analysis framework built in C++. As it is a part of *ROOT*, it has a Python interface called *PyROOT*. *PyROOT* helps create bindings between C++ and Python. The interaction between C++ and Python is done using *cppyy*, which uses the C++ interpreter *Cling* to generate bindings for C++ classes automatically.

RooFit uses RooWorkspace to manage its models, data and other information. RooWorkspace has a C++ like domain specific factory language in RooWorkspace::factory() in which commands are passed as C++-like strings like so-

```python
import ROOT

compact = False
w = ROOT.RooWorkspace("w")

if not compact:
    w.factory("Gaussian::sig1(x[-10,10],mean[5,0,10],0.5)")
    w.factory("Gaussian::sig2(x,mean,1)")
    w.factory("Chebychev::bkg(x,{a0[0.5,0.,1],a1[-0.2,0.,1.]})")
    w.factory("SUM::sig(sig1frac[0.8,0.,1.]*sig1,sig2)")
    w.factory("SUM::model(bkgfrac[0.5,0.,1.]*bkg,sig)")

else:
    w.factory(
        "SUM::model(bkgfrac[0.5,0.,1.]*Chebychev::bkg(x[-10,10],{a0[0.5,0.,1],a1[-0.2,0.,1.]}), "
        "SUM(sig1frac[0.8,0.,1.]*Gaussian(x,mean[5,0,10],0.5), Gaussian(x,mean,1)))"
    )
data = w["model"].generate({w["x"]}, 1000)

w.Import(data, Rename="data")

x = w["x"]
k = ROOT.RooKeysPdf("k", "k", x, data, ROOT.RooKeysPdf.NoMirror, 0.2)
w.Import(k, RenameAllNodes="workspace")

w.Print()
```

The main issue with this approach is that this is not Pythonic, as the language has a completely different syntax from Python. This poses the following problems to the users:

- The code could be more intuitive. It is equivalent to coding C++ as strings in Python.
- The syntax is not Pythonic. Class methods are not directly available in Python.
- It takes more work to debug and understand the code. We cannot use features such as syntax highlighting.
- Constants declared in the global scope are difficult to reuse. F-strings are required to pass them as arguments inside strings.
- Summation, convolutions, additions, etc., of two or more p.d.f.s, can't be made directly. (Currently, these are implemented as in rf512_wsfactory_oper.py)

Hence, it is essential to make interaction with RooWorkspace more Pythonic.

This project aims to make the interaction more Pythonic. This can be done in two steps:

- **Writing C++ plugin for RooWorkspace factory:**
  C++ headers use the arguments from Python calls to make the necessary objects/call the functions. The final objects/functions/classes are returned as pointers. These pointers are then iterable in Python with the help of cppyy.

Plugins for RooFactoryWSTool are created so that they can use RooJSONFactoryWSTool to make RooFit objects using the JSON String.

- **Writing Pythonizations:**
  Python libraries corresponding to the functions & classes are made that do the back-end work of calling C++ functions & classes. The returned objects are then available to the user as Python objects. This is done by the Implementation of RooWorkspace Pythonization at "_rooworkspace.py."

Further, to handle the functionality with summations, products, convolutions, etc., we have to implement separate plugins to map them to C++ headers. If the functions are simple enough, we can write the Python version of the operations too. This allows us to call operations on two or more p.d.f.s directly.

The corresponding documentation with appropriately chosen tutorials to demonstrate the usage of the new Pythonic interface is also required. Specific examples from HEP experiments can be used to write clear explanations. Unit tests are also written to test the functionality.

Benefits to the community:

- Pythonic implementation of RooWorkspace::factory() makes it significantly easier for users to write code in Python
- Python code becomes more intuitive, which makes it easier to debug and maintain the code.
- Constants can be defined easily and globally throughout the Python files.
- Special operations, such as the s*um* & *prod* of two p.d.f., can be done using simple operators like '+' & '*'.
- Improved documentation for new pythonizations to allow new users to learn RooFit easily, even without extensive C++ background.

Deliverables:

- Develop a Pythonic alternative to RooWorkspace::factory() in two steps:
  1.) Making Python libraries
  2.) Writing plugins for RooFactoryWSTool
- Pythonization of special operations of RooWorkspace such as *"SUM", "FCONV", "SIMUL", etc.*
- Make tutorials to demonstrate the usage of new functions in Python
- Write unit tests to check the functionality of Pythonic code
- Support for creating Histfactory p.d.f.s using the JSON I/O

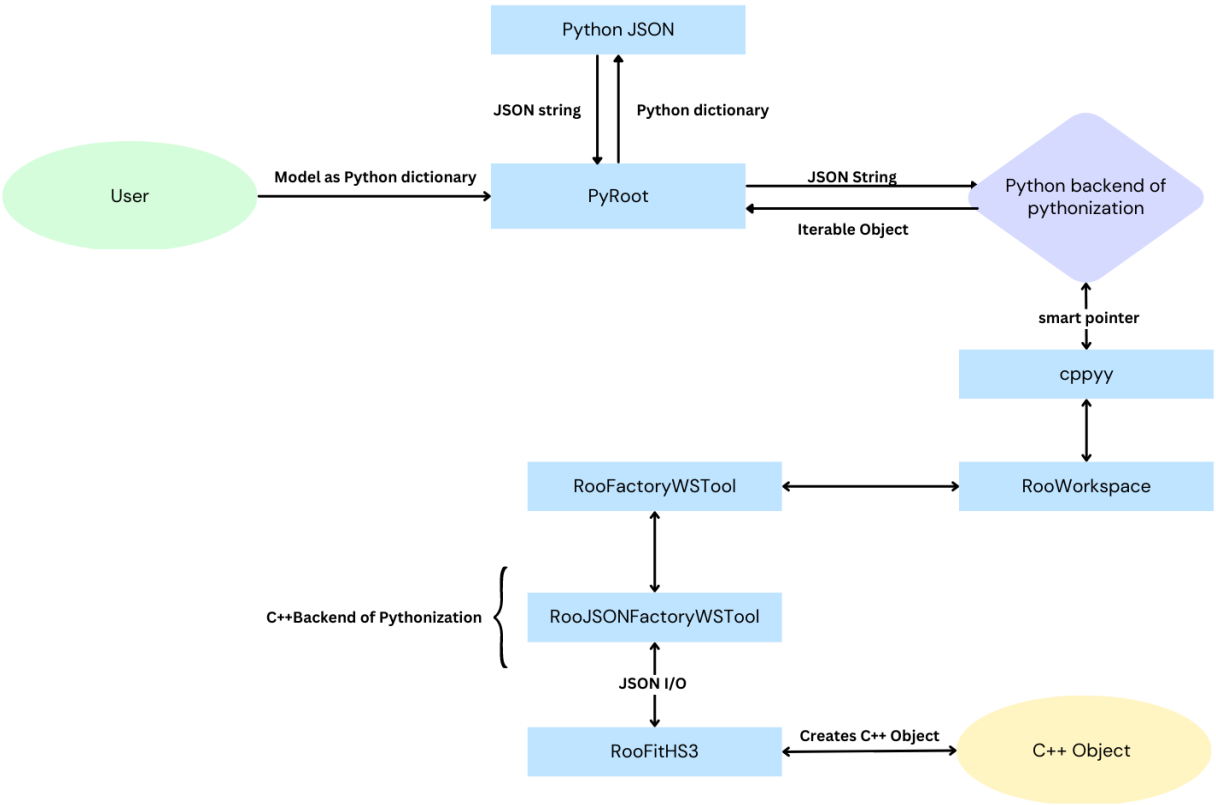
- Improve documentation for HistFactory using relevant examples from particle physics.

Overview & Homework:

To implement the JSON I/O, the roofiths3 library is used, which uses the *nlohmann* library at its backend. Python commands are parsed as JSON strings using the Python JSON library, which is then passed to the *gInterpreter,* the *cling* interpreter or ROOT.

On the C++ side, the parsed string from *gInterpreter* is then serialised as a *nlohmann::json* object. The JSON object is further used to build the input string for *gInterpreter*. This can either directly call the RooWorkspace factory language, or the factory language could be modified to make functions with different signatures having JSON objects as input.

The *unique_ptr* to the new object is returned to the C++ backend. This *smart pointer* is then parsed by Python with the help of *cppyy*. This creates an iterable Python object that has all the original class methods available. The following diagram gives an overview of the process:



To implement the support of operations over two or more p.d.f. Python operators must be overloaded for RooFit classes, as this cannot be directly mapped to JSON I/O. Similarly, for adding functionality for HistFactory, overloading Python methods might be required. Further, HistFactory may need additional bindings to be mapped to the correct header files. Hence, the case of HistFactory objects has to be handled separately.

After the Pythonization of RooWorkSpace, we will be able to use RooFit like in the following code snippet-

```
import ROOT

ws = ROOT.RooWorkspace("ws")

ws.factory("my_x[0, -10, 10]")                    # Declaration of variables
ws.factory("my_mu[0, -10, 10]")                   # inside WSFactory
ws.factory("my_sigma[2.0, 0.1, 10]")

const_fine_structure = 1/137                      # constants can be declared easily

model1 = dict(
    x="my_x",
    mu="my_mu",
    sigma="my_sigma",
    type="gaussian_dist",
)

model2 = dict(                                    # Model parameters are passed as
    x="my_x",                                     # a Python dictionary
    mu="my_mu",
    sigma="my_sigma",
    type="gaussian_dist",
)

model2["x"] = "new_x"

ws.factory(                                       # Create objects directly from
    name="my_gaussian",                              # from WSFactory
    x="my_x",
    mu="my_mu",
    sigma="my_sigma",
    type="gaussian_dist"
)

ws.factory(model1)                                # Creates instaces of model1
ws.factor(model2)                                 # Creates instaces of model2

ws["add_pdf"] = ws["gaussian_1"] + ws["gaussian_2"] # Addition of p.d.f.s
ws["FConv"] = ws["gaussian_1"] ** ws["gaussian_2"]  # Convolutions of p.d.f.s

ws.Print()
```

Timeline:

Duration	Task
April 4	• Deadline for submitting the project proposal
May 4	• Accepted Proposals are announced • The community bonding period starts
May 4 - May 29	• Familiarise with the ROOT and RooFit documentation & Community. • Understand the coding conventions and test system. • During this period, I will be active in the community and communicate with my mentor to finalise the modifications that need to be made and new “Pythonizations” that need to be created
– Official Coding Period starts –	
May 29 - June 14	**Writing C++ plugin for RooWorkspace factory:** • I will write plugins for RooFactoryWSTool. I will modify the headers and source code to use the RooJSONFactoryWSTool to support JSON I/O. So, I will direct the constructor calls towards the JSON factory. • Since basic Pythonization is completed by this time, I will write the unit tests for them.
June 15 - June 30	**Writing Pythonizations:** • I will make a Pythonization backend on the Python side using *cppyy* to parse objects to C++ headers. • I have to discuss with my mentor and community members the conventions of the Pythonic implementation • Previous Pythonizations can be used as a reference
July 1 - July 13	**Pythonization of special functions:** • Implement the use of special functions(sum, products, convolutions, etc.) directly on the p.d.f. objects in Python. • I will write a Python library to overload the Python operators to support RooFit p.d.f.s. This would typically involve a more significant part of the

	codebase. ● I will also write unit tests for new operators
July 14	Midterm evaluation deadline
July 15 - July 31	**Adding support for HistFactory p.d.f.s** ● HistFactory p.d.f.s store data as discrete values, so I need to handle the histograms separately. ● I will write separate bindings for histograms. This also includes support for histogram-specific functions such as modifying discrete values.
Aug 1 - Aug 14	**Documentation & tutorials:** ● Documentation for newly added Pythonization will be updated. ● I will use specific examples to make tutorials for the new Pythonic interface. This needs to showcase the new behaviour of Python classes—with particular emphasis on HistFactory tutorials.
Buffer period of 1 week for unpredictable delays or extra features	
August 21 - 28	Final Submissions

Other commitments during summers:

I will have my summer break during this project, and thus, I've no other commitments during this period. I'll be staying back at home for most of the part. I can spend around 40 hours on the project on average, though I will put in more time if required.

Background and Programming Skills:

I am a sophomore at IIT Delhi, India. I'm pursuing a major in Engineering Physics and a minor in Computer Science & Engineering. I am keenly interested in computer science, mathematical physics and high-energy physics. I like solving problems and learning new things.

Platform details & programming skills:

I work on MacOS Ventura. I use VSCode for development and bash for writing scripts. I am proficient in C, C++, Python & Java. I have experience with version control systems

like git and GitHub. If I'm stuck, I go to Google and always find a way to solve the problem!

I've been exploring my interest in machine learning & data science, and thus, I've become familiar with Python. My experience in C++ comes from developing OpenCV applications & programming contests.

Why me & Why this project?

As a physics & computer science student, I'm excited to develop programs that have applications in solving real-world physics problems. I've had prior experience with ROOT while making a machine learning model with my professor, who was analysing data from particle accelerators.

Python is widely used to analyse data nowadays. Thus, it is crucial to have a Pythonic implementation of essential frameworks such as RooFit. Passing C++-like commands in RooFit Framework is relatively time-consuming, especially for people unfamiliar with C++. Having faced the same difficulties while doing the project, I'm motivated to improve the RooFit library by making it more Pythonic.