An Interactive and Comprehensive Working Environment for High-Energy Physics Software with Python and Jupyter Notebooks

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 072020 doi:10.1088/1742-6596/898/7/072020

An Interactive and Comprehensive Working Environment for High-Energy Physics Software with Python and Jupyter Notebooks

N Braun[1], T Hauth[1], C Pulvermacher[2], M Ritter[3]

[1] Karlsruhe Institute of Technology, Institut für Experimentelle Kernphysik, Wolfgang-Gaede-Str. 1, 76131 Karlsruhe, Germany
[2] High Energy Accelerator Research Organization (KEK), 1-1 Oho, Tsukuba 305-0801, Ibaraki, Japan
[3] Ludwig-Maximilians University Munich, Excellence Cluster Universe, Boltzmannstr. 2, 85748 Garching, Germany

E-mail: nils.braun@kit.edu, thomas.hauth@kit.edu, christian.pulvermacher@kek.jp

Abstract. Today's analyses for high-energy physics (HEP) experiments involve processing a large amount of data with highly specialized algorithms. The contemporary workflow from recorded data to final results is based on the execution of small scripts – often written in Python or ROOT macros which call complex compiled algorithms in the background – to perform fitting procedures and generate plots. During recent years interactive programming environments, such as Jupyter, became popular. Jupyter allows to develop Python-based applications, so-called notebooks, which bundle code, documentation and results, e.g. plots. Advantages over classical script-based approaches is the feature to recompute only parts of the analysis code, which allows for fast and iterative development, and a web-based user frontend, which can be hosted centrally and only requires a browser on the user side. In our novel approach, Python and Jupyter are tightly integrated into the Belle II Analysis Software Framework (basf2), currently being developed for the Belle II experiment in Japan. This allows to develop code in Jupyter notebooks for every aspect of the event simulation, reconstruction and analysis chain. These interactive notebooks can be hosted as a centralized web service via jupyterhub with docker and used by all scientists of the Belle II Collaboration. Because of its generality and encapsulation, the setup can easily be scaled to large installations.

1. Introduction

The Belle II experiment is currently being assembled at the asymmetric electron–positron collider SuperKEKB and is set to start taking data in 2017. It is the successor of Belle, which ran from 1998 to 2010, and used the same operating principle at the former KEKB collider to confirm the violation of CP symmetry in the B meson system. As a so-called B-factory, SuperKEKB operates on the $\Upsilon(4S)$ resonance, which allows it to produce and record data from a large number of B mesons. With the upgrade the instantaneous luminosity will be increased by a factor 40 to $8 \times 10^{35}\mathrm{cm}^{-2}\mathrm{s}^{-1}$. Ultimatively, Belle II is expected to record 50 times more collisions than Belle and to significantly increase the sensitivity of various searches and measurements [1]. Both the detector [2] and the software [3] used to process the data will be upgraded to take advantage of technological developments and to be able to handle the greatly increased data rate.

The Belle II analysis software framework (basf2) was developed based on the experience from Belle as well as other high-energy physics experiments. It consists of independent C++ modules joined together by a core library that takes care of configuration, exchange of data between modules, I/O, etc. In this framework, each module is responsible for a task of relatively limited scope and is compiled into a separate shared library. Configuration of basf2 is handled using steering files written in Python 3, where a typical specimen might create a sequence of modules (called "path") and set appropriate parameters for them. Python makes it easy to pre-define standard sets of modules for e.g. trigger simulation, which can be added using functions like `add_softwaretrigger_reconstruction()`. For physics analyses in particular, this is used to wrap common reconstruction tasks, including combination of particle candidates, vertex fits and application of cuts. As a result, analysts can write a high-level description of their reconstruction and selection procedure, which will be performed using common and well-tested code.

This configuration interface is provided using Boost.Python [4] and will load the requested module libraries on demand. Each module can define a set of parameters to modify its behaviour; generalized support for many different parameter types (including lists, tuples and dictionaries) is provided through C++ templates. Once event processing is started on a path of configured modules it will block and only return the control flow to Python after processing is complete.

In addition to the configuration of modules themselves, basf2 also provides a Python interface to the data exchanged between modules (i.e. their input and output data). As ROOT [5] is already used for the I/O functionality of the framework, PyROOT can be used to access the member functions of most classes without requiring further user action. Within the framework, this allows creating basf2 modules in Python; this feature is used frequently to create tests or prototypes.

In contrast to executing steering files via Python, Jupyter notebooks provide an enriched working environment: A browser-based frontend to an interactive Python session is used to execute commands and view results and visualizations, both of which can also be saved in the notebook. Users benefit from syntax highlighting and tab-completion as well as integration with data-science tools like ROOT, matplotlib or pandas.

2. Framework Integration of Jupyter Notebooks

The integration of Python into basf2 is already extensive, so using it with Jupyter notebooks is a natural and easy next step. However, using basf2 with Jupyter notebooks directly is not always very convenient: if a basf2 path is processed during a notebook execution and the process fails (because of wrong configuration, software issues etc.), the notebook kernel will immediately be killed and the logs of the process are lost. This does not work well with the quick turnaround times that are normally achieved when working with the notebook interface. Additionally, processing many paths during a notebook execution (also in parallel), which would suit to the narrative character of the notebooks, is inconvenient and implies a large code overhead. As the execution is given to the basf2 framework, all interactivity is lost during the process and there is not much improvement compared to just using basf2 from the command line (excepting the ability to store description, code and results in the same file).

The described drawbacks are solved by the Python package hep-ipython-tools [6][1], which was developed as an interface from Jupyter notebooks for software used in high-energy physics experiments. It decouples the calculation process from the Jupyter notebook kernel into an abstract `calculation` object, which can be controlled interactively from the notebook. This makes it possible to start multiple calculations at once and to monitor their progress while continuing to work in the notebook. Abstracting the basf2 calculation together with additional interactive widgets and convenience functions for an easier interplay between juypter and basf2

[1] The name was chosen when the Jupyter project was still called ipython [7].

not only improves the user experience but also accentuates the narrative and interactive character of the notebooks.

The hep-ipython-tools package was developed in the context of the Belle II experiment, but can easily be adopted by the software frameworks of other experiments.

3. Implementation Details

Because of the highly modular nature of basf2, more or less all information needed to process a basf2 calculation is encapsulated in the path[2]. The main entity of the hep-ipython-tools, the `CalculationProcess` – which is based on a `multiprocessing.Process` – therefore takes a path as an input, prepares (e.g. adds statistics and analysis modules only meaningful in the context of Jupyter notebooks) and runs it in a separate process. The way this calculation is done (and whether one passes a path or some other object) is an implementation detail and is decoupled from the generalized rest. The abstract `Calculation` object holds one or more[3] possible different instances of a `CalculationProcess` and has convenience functions to control the execution state of the `CalculationProcess` instances. The object is returned as a handle to the user when one of the convenience functions to process a basf2 path is called.

The `CalculationProcess` also holds a reference to the `Queue`, which can be used to communicate between the Jupyter notebook and the calculation process – keeping the interactive character of the notebooks. It is used to transport progress information as well as calculation results, e.g. statistics or output file names, which are generated during the calculation. It can be fed with key-value pairs by every Python basf2 module in the processed path as well as before and after the calculation.

Information from the calculation (e.g. stored in the communication queue) can either be directly accessed in raw form or, to underline the interactivity, can be visualized using widgets shipped with the hep-ipython-tools. A widget is a graphical element written in HTML and uses JavaScript (especially via the `ipywidgets` Python [9] and the jQuery JavaScript [10] libraries) for interactivity and animation using the information stored in the calculations or in the basf2 framework. They can be shown directly in Jupyter notebooks (as the frontend itself is also written in HTML) and fit seamlessly into the workflow. The HTML, CSS and JavaScript code for the widgets is generated on the fly using Python and the collected data.

Examples for these widgets include visualizations of the log output of the basf2 process, a progress bar to show its execution status, an interactive basf2 path visualization with all modules and their parameters, and tabular representations of the data entities exchanged between the modules during calculation or the statistics of their process runtime.

To better explain the general picture, two examples of widgets are shown here. The first one (see Figure 1), shows the data entities that are exchanged between modules through basf2 mechanisms. The entity names together with their number are collected by a Python basf2 module, which is added automatically to the path by the `CalculationProcess` object. The data is then communicated via the `Queue` to the Jupyter notebook where it is used to generate HTML code.

The second one (see Figure 2) can be used to show the log output of the processed calculations. The `CalculationProcess` configures basf2 to write all its output into a temporary file, which is created and deleted by the hep-ipython-tools, by including a specialized basf2 module into the processing path. After the calculation is finished (no matter if successfully or not), the file content is read, analyzed and corresponding HTML is generated.

[2] There are some additional parameters like the maximum number of events to process or the random seed, which are optional arguments to the `CalculationProcess`.

[3] If more than one `CalculationProcess` is present, they will be processed in parallel when the calculation is started. The process to create many different `CalculationProcess` resembles the grid search algorithm of the sklearn package [8].

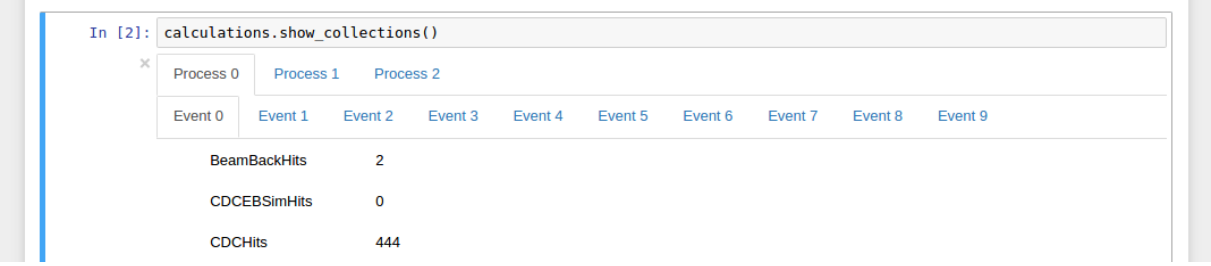

Figure 1. Screenshot of a part of a Jupyter notebook, showing a widget to visualize the collections that are exchanged during basf2 module calculation. The user can choose interactively between the three processes (which were calculated in parallel before) and different events.

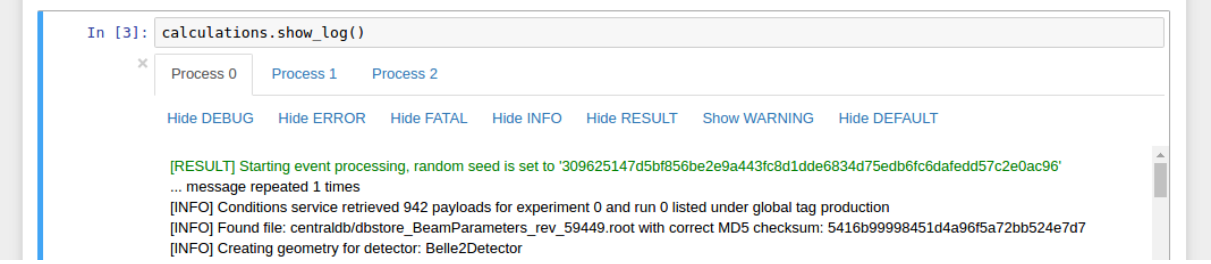

Figure 2. Screenshot of another part of a Jupyter notebook – this time showing the log widget, which visualizes the output to stdout of the three basf2 calculations. The user can choose, in which detail (e.g. error or debug) the log should be shown.

More information and examples can be seen in the Jupyter notebooks that are also part of this project [11] or in a screencast showing the core functionality [12].

4. Application: Full Physics Analysis with Jupyter Notebooks

The Belle II analysis tools are an integral part of basf2 and are a Python-based library to facilitate physics analysis in Belle II. A rich set of utility functions allows the scientist to perform standard analysis tasks like reconstruction of a possible decay or Monte-Carlo matching in a compact fashion:

```
reconstructDecay('J/psi -> mu+ mu-', '2 < M < 4', path=my_path)
...
matchMCTruth('J/psi', path=my_path)
```

The analysis tools therefore allow to express and process complex analysis steps in a compact form within a Jupyter notebook. Furthermore, the resulting n-tuples can then be read in and processed further in the same notebook.

Here, the scientist can take full advantage of the rich set of libraries available for scientific work in Python. The SciPy [13] library includes matplotlib for plotting directly into the notebooks, numpy [14] and pandas [15] for fast computations on large in-memory datasets and a multitude of fitting and statistics functions. The popular machine-learning toolkit SciKit-learn [16] is closely integrated with SciPy's data structures. The well-known ROOT data analysis framework was recently also extended with an integration to Jupyter notebooks [17]. ROOT can now output its plots directly into notebooks and plots are interactive and support zooming and panning.

All the above-mentioned tools can also be employed with the classical approach using scripts to perform the data analysis and storing the output plots in files. However, there are some

significant benefits in using notebooks which is increasingly attracting analysis users to switch to this workflow.

The analysis notebook does not only contain the source code, but also "tells the story of the analysis" by including the resulting plots, text cells with headings, formulas and explanation text to structure and describe each analysis step Figure 3. This significantly increases the readability and traceability of the analysis. The notebook can also be easily shared with co-workers or reviewers which can, due to the inlined plots, quickly arrive at an understanding of the analysis procedure.

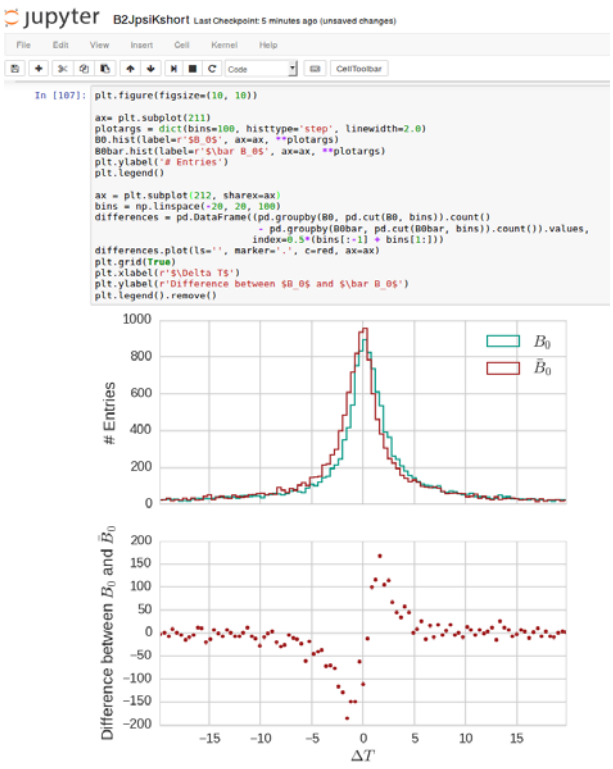

Figure 3. Screenshot of a Jupyter notebook showing an example analysis code and output of simulated events.

Probably the biggest advantages of Jupyter notebooks is the fast turn-around time and the direct feedback between data and visualization. The classical script-based approach often requires to rerun the full script to see the impact of minor modifications. In contrast, if the analysis notebook is properly structured into cells, changes in one cell can be executed and the output of this one cell checked immediately. This is especially valuable in earlier stages of the analysis, where an exploratory method aids in finding the best possible analysis strategy, for example, when the best quantity for a cut value needs to be determined.

This advantage is even more tangible if the input data is loaded in the first cell of the notebook, for example from ROOT n-tuples or pandas dataframes. In contrast to scripts, which load the input data every time, notebooks keep the data in memory, which significantly decreases the processing time in case of multiple executions.

Organizing the contents of notebooks as the complexity of an analysis grows can be challenging. We have found that the best way is to outsource mature analysis code into a Python package and import it into the notebook. One additional consideration when working with notebooks is the slightly more complex version control management. Because the entire content (including figures) of notebooks is stored, the differences between two revisions can be large, even when the changes to Python code are minor. One strategy which proved useful is to execute Jupyter's Clear function, which removes all text and plot output in a notebook, before committing the changed notebook to version control. This can also be handled by a commit hook.

5. Application: Outreach and Education

The possibility to create self-describing notebooks by interlacing code, documentation and plot cells in one notebook makes them ideal for education and outreach purposes. Introductory text can be followed by compact fragments of Python code to explain a specific functionality. The learners can be encouraged to execute the code fragment themselves and modify the code slightly to see the immediate impact in the form of changed text output or plots. Furthermore, practical

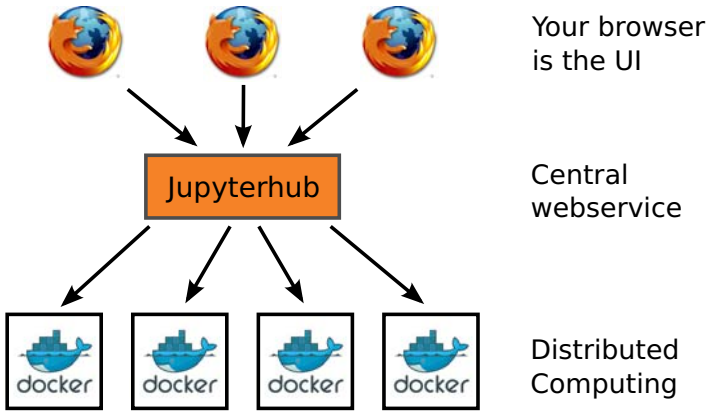

Figure 4. Multiple users load their notebooks in their browser from the centrally hosted jupyterhub service. In the backend, jupyterhub can instantiate multiple docker containers to isolate the user's code execution.

tasks can be placed inside the notebooks, which can be solved using the gained knowledge.

Within the Belle II collaboration, we have created multiple tutorial notebooks for new collaboration members which have been successfully used in workshops and to give a first orientation to students starting their bachelor or master thesis.

Due to the self-contained nature of notebooks, they are also a perfect medium to present scientific methods and results to the interested public, for example by making notebooks of a selected event reconstruction available on the experiment's home page, as demonstrated by the LIGO collaboration [18].

6. Application: Documentation

Similar to tutorials, notebooks can also be used to provide documentation for framework libraries and tools. While this option is currently not used at Belle II, we might consider updating the documentation as more people are using notebooks for their work.

Another area where notebooks are used successfully is bug reporting, where test cases can be attached to issue tracker tickets. Complex bugs are intricate and only occur with a certain event and often can only be properly identified by looking at one specific distribution. As our notebooks can contain event generation, simulation, reconstruction and result plotting in one compact file, they are ideal to isolate the issue, plot the problematic variable and forward this notebook to experts for further examination.

7. Hosting Jupyter Notebooks with Jupyterhub and Docker

The jupyterhub project provides a scalable way to host Jupyter notebooks of the experiment's users on a central service. For the users, this has the advantage that they do not have to take care of a local software installation, the notebooks can be accessed from anywhere and the analysis input data is located on a high-performance (potentially distributed) storage backend.

For the administration team, jupyterhub provides an elegant way to separate users and simplify software deployment: for each Jupyter user, a separate docker container instance is created to execute the notebook code in an isolated fashion Figure 4. This way, the software requirements of multiple different experiment groups can be met easily.

We successfully used the described setup to centrally host notebooks for tutorials and workshops for up to thirty users. Larger hosting options, like CERN's Service for Web based

ANalysis (SWAN) [19], exist and we expect that more research institutions will provide a notebook hosting option to their users.

8. Conclusion

The hep-ipython-tools library was developed in the context of the Belle II software framework and provides a common, experiment-independent basis to seamlessly integrate HEP frameworks with Jupyter notebooks. At the core of this library lies the `calculation` object, which encapsulates the framework execution and can exchange information between the running framework process and the notebook. A set of optional notebook widgets have been developed to give execution feedback to the user and to present the framework-specific results using Jupyter's unique visualization features. HTML and JavaScript output were used to generate interactive content for the user. Overall the code necessary to achieve this tight integration is compact but enables a wide range of possibilities with Jupyter notebooks.

Combined with Belle II's powerful, Python-based analysis tools, complete analyses can be developed using Jupyter notebook and our users especially welcome the fast turnaround times. Other applications include Jupyter notebooks for outreach and for internal tutorials and external education of the interested public.

More and more scientists in the HEP field are moving to Jupyter notebooks for their software development and data analysis needs. The popular ROOT data analysis framework recently was extended by a Jupyter notebook integration and hosting services for notebooks, like CERN's SWAN, start to emerge. We expect the amount of such services to increase in the future and, quite possibly, having a notebook hosting service will become a standard for every HEP experiment in a few years time.

References

[1] Urquijo P 2015 *Nucl. Part. Phys. Proc.* **263-264** 15–23
[2] Abe T *et al.* (Belle II) 2010 Belle II Technical Design Report Tech. rep. (*Preprint* `1011.0352`)
[3] Moll A 2011 *Journal of Physics: Conference Series (CHEP 2010)* **331** 032024
[4] Boost.Python website `http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/` (last accessed 27 January 2017)
[5] Brun R and Rademakers F 1997 *Nucl. Instrum. Meth.* **A389** 81–86
[6] Braun N 2017 `hep-ipython-tools` repository `https://github.com/hep-ipython-tools/hep-ipython-tools`
[7] Jupyter Project 2015 The Big Split `https://blog.jupyter.org/2015/04/15/the-big-split/`
[8] `GridSearchCV` reference `http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV` (last accessed 24 January 2017)
[9] `ipywidgets` library `https://github.com/ipython/ipywidgets` (last accessed 24 January 2017)
[10] `jQuery` website `https://jquery.com/` (last accessed 21 January 2017)
[11] Braun N 2017 `hep-ipython-tools` examples `https://github.com/hep-ipython-tools/example-notebooks`
[12] Braun N 2016 Screencast of the usage of `hep-ipython-tools` `http://www-ekp.physik.uni-karlsruhe.de/~nbraun/movie.mp4`
[13] `SciPy` website `https://www.scipy.org/` (last accessed 24 January 2017)
[14] `Numpy` website `http://www.numpy.org/` (last accessed 24 January 2017)
[15] `Pandas` website `http://pandas.pydata.org/` (last accessed 24 January 2017)
[16] `scikit-learn` website `http://scikit-learn.org` (last accessed 24 January 2017)
[17] ROOT Notebook integration website `https://root.cern.ch/notebooks/HowTos/HowTo_ROOT-Notebooks.html` (last accessed 24 January 2017)
[18] Signal Processing with GW150914 – LIGO Collaboration `https://losc.ligo.org/s/events/GW150914/GW150914_tutorial.html` (last accessed 27 January 2017)
[19] SWAN website `https://swan.web.cern.ch/` (last accessed 24 January 2017)