ISBN: 978-3-95450-184-7 doi:10.18429/JACoW-IPAC2018-THPAK049

SIMULATION CODE DESIGN FOR THE INTERPRETED LANGUAGE USING THE COMPILED MODULE *

Kei Fukushima†, Takashi Yoshimoto, Zhengqi He‡, Michael Davidsaver, Tong Zhang, Guobao Shen§, Masanori Ikegami, FRIB, Michigan State University, MI 48824, USA
Ji Qiang, LBNL, Berkeley, CA 94720, USA

Abstract

We are planning to use two types of the accelerator simulation codes for FRIB (Facility for Rare Isotope Beams). One is the linear envelope tracking code “FLAME” for fast simulations. FLAME can calculate the FRIB-linac beam envelope within an order of ms. This is useful in systematic surveys, wide range optimizations and so forth. This code, written in C++, was designed with Python interface from the beginning. On the other hand, “Advanced-IMPACT” is the particle tracking code dedicated for precise and realistic calculations, which can simulate the particle losses, nonlinear and space-charge effects. This code is refactored from the Fortran code IMPACT-Z developed in LBNL. Both codes provide the compiled modules for Python to support flexible inputs and direct outputs management in memory. In other words, they can be directly connected to the modern scientific tools through the Python interface without delay in the data transport. In addition, these modules can accomplish the interactive simulation processes without losing computational efficiency. We report the knowledges applicable for other accelerator simulation codes among those obtained through these developments and designs.

INTRODUCTION

Numerical simulation for accelerator plays important role in the beam study, and it is used for various purpose like systematic survey, parameter optimization, design study and so forth. On the other hand, most of simulation codes provide a compiled executable file and its input/output data are put on the disk storage. Thus, in the case of user wants to define input parameters by using some formulations, user need to calculate the value outside of the code, and then write the value into the input file. If the input parameter is recursive to the previous simulation results like optimizations, user need to access both input and output files for every iteration. In this case, the disk access speed is depend on the data size but slower than dynamic memory access, and the user-made application becomes complicated. Interpreted and interactive language is able to communicate the input/output data on dynamic memory, but if whole code is written in this type of language, the computational speed is limited.

To solve these issues, we can provide both the compiled module which contains the core part of the simulation code and the interpreted interface. Hereby, user can keep computational efficiency and use flexible input.

CODE DESIGN

Nowadays, interpreted language “Python” is used widely in academic purpose due to its plentiful scientific packages [1]. We have developed two accelerator simulation codes for FRIB (Facility for Rare Isotope Beams) operation and commissioning, and both codes provide compiled Python package for users. In both cases, the software directory tree is designed like,

```
software_name/
├── source/
└── python/
    └── package_name/
```

`source` directory contains the physics part of the simulation code, and `python` directory contains the codes for the compiled Python module and the interface. The build is managed by using CMake, and the structure of the build directory is,

```
build/
├── bin/
├── lib/
└── python/
    └── package_name/
```

where `bin` directory contains the executable file, `lib` directory contains the core library including the physics part for both the executable file and compiled Python module, and `python` directory contains the module and the interface to install. Physics part is defined in the core library, and it is compiled by using general compiler. Off course the difference between the executable file and Python module is just interface, thus the simulation results become the same.

In the Python interface, user will use the simulation process repeatedly by changing parameters. Therefore, the key methods for an accelerator simulation interface are “beam initialize”, “beam tracking”, and “parameter configuration”. Needless to say, the initial parameters must be input in its interface or defined by the external files, and the beam tracking method must provide the simulation results.

The user documentation is also important. In these simulation codes, we use the Python Documentation Generator

* Work supported by the U.S. Department of Energy Office of Science under Cooperative Agreement DESC0000661.
† fukushim@frib.msu.edu
‡ Present address: RIEKN Brain Science Institute, Wako, Saitama 351-0106. Japan.
§ Present address: Argonne National Laboratory, Argonne, IL 60439, USA.

"Sphinx" [2]. Sphinx automatically collects "docstrings" in Python code and generate the documentation, thus we can keep the documentation up-to-date easily. Sphinx supports HTML, latex base PDF, and etc as the documentation format.

Envelope Tracking Code: FLAME

FLAME (Fast Linear Accelerator Model Engine) is developed by FRIB [3–5]. This code is dedicated for fast simulation and can calculate the FRIB-linac beam envelope within an order of ms. The source part is written in C++, and the compiled Python module is provided by C++.

Remarkable features:

- Envelope tracking with multiple charge states
- Support general lattice elements and asymmetric rf cavity by using Thin-Lens-Model
- Transfer matrix caching for iterative running
- Python interface (include ipython-notebook)

The main class of FLAME module is `Machine()`, and user can use it in Python interface like:

```
>>> from flame import Machine
>>> M = Machine(open('FRIB_LS1.lat', 'rb'))
>>> S = M.allocState({})
>>> obs = range(len(M))
>>> result = M.propagate(S, observe=obs)
```

where `FRIB_LS1.lat` is the input lattice file. The memory space for the beam state is allocated by `allocState()`, and the envelope tracking is execute by `propagate()` method. Here, user can define the observing point in the lattice by using `observe` argument. This is because, in the case of the envelope tracking simulation, the whole beam data size is much smaller than the particle tracking simulation. Even if user stores the whole beam data for every elements, the data size does not matter. Thus, the `propagate()` method returns whole beam data at the observing points defined by the `observe` argument, and use can pick up the data what user wants. `propagate()` method supports to input tracking section also:

```
>>> M.propagate(S, start=3, max=10)
```

where `start` is the starting point and `max` is the number of propagation steps.

For parameter configurations, `Machine()` has `find()` and `reconfigure()` methods.

```
>>> idx = M.find(name='solenoid_name')[0]
>>> M.reconfigure(idx, {'B': 4.5})
```

Here, `find()` returns index numbers of the target element as a list, and `reconfigure()` make change the element parameter by using its index number and dictionary style definition.

Users can therefore assemble their own application by combining these methods for their research.

Particle Tracking Code: Advanced-IMPACT

IMPACT-z (Integrated Map and Particle Accelerator Tracking Code) is developed by LBNL [6, 7]. We have refactored the IMPACT-z as FRIB-branch named Advanced-IMPACT with keeping its physics equivalence. The original IMPACT is written in Fortran, the compiled Python module is generated by using "F2PY" [8].

Remarkable features:

- 3D field data caching for iterative running
- Synchronous phase input for RF cavity
- RFQ element by using 8-term potential
- Python interface (include ipython-notebook)

As I mentioned above, the core part is compiled by using the general compiler, and the whole computational time is the same as the original one. The biggest benefit for the computational speed is the 3D field data caching. In the case of using many 3D field data for the simulation, the data loading cost can be the bottleneck of the simulation time. For the one-time running, this data loading cost is inevitable. But for the iterative running, the process can reuse the 3D field data ad cut the loading cost.

The main class of IMPACT module is `Sequence`, and user can use it in Python interface like:

```
>>> from impact import Sequence
>>> sq = Sequence('FRIB_LS1.in')
>>> sq.distribute()
>>> sq.run()
```

`Sequence()` instance can be constructed with the input file, and it support the same format as the original IMPACT input file. `distribute()` method allocates and generates the initial beam distribution by using the input file parameter or substituting `particles` argument to this method. `run()` method executes the particle tracking, and this method supports to input tracking section also:

```
>>> sq.run(start=3, end=10)
```

Once user executes `run()` method, the simulation result can be found in methods in `Sequence()`. As an example:

```
>>> sq.hxrms('mm')
array([3.3121, 4.32432, ..., 1.22332])
```

where `hxrms()` returns horizontal rms beam size history with the input physical unit. `Sequence()` class supports all history results provided in the original IMPACT. Unlike the envelope tracking code FLAME, basic output of IMPACT is history results of the statistic values, and the whole beam output is supported by using a flag element.

For parameter configurations, `Sequence()` class has `search()` and `conf()` methods.

```
>>> idx = sq.search(name='solenoid_name')
>>> sq.conf(idx, {'B': 4.5})
```

Here, `search()` returns index numbers of the target element as a list, and `conf()` make change the element parameter by using its index number and dictionary style definition. These methods are designed similarly to FLAME for user friendliness.

USE CASE

Transverse Matching

By using the Python interface, user can build beam tuning script easily. For example, in the case of tuning the two quadrupole strength to adjust the rms beam size by using Advanced-IMPACT:

```
>>> import scipy as np
>>> des_xrms = 2.5e-3
>>> des_yrms = 2.5e-3
>>>
>>> def cost(params):
>>>   sq.conf('quad1', {'B2':params[0]})
>>>   sq.conf('quad2', {'B2':params[1]})
>>>   sq.distribute()
>>>   sq.run()
>>>   dx = sq.hxrms('pm1') - des_xrms
>>>   dy = sq.hyrms('pm1') - des_yrms
>>>   return np.sum(np.absolute([dx, dy]))
```

Here, we defined the cost function by using the difference between the desired value and the simulation result. Then, by using the `minimize()` function in SciPy package [9]:

```
>>> from scipy.optimize import minimize
>>> minimize(cost, [5.0, -5.0])
```

where `[5.0, -5.0]` is the initial parameter for the quadrupoles.

In this example, the parameter is two dimensions, but most of the optimizers support N-dimensional parameter for more complex systems.

Virtual Accelerator

In FRIB, we are developing the virtual accelerator with switchable simulation codes between FLAME and Advanced-IMPACT as shown in Fig 1. The virtual accelerator reproduces "EPICS" layer of the real accelerator [10]. Thus the physicist can develop and benchmark the physics application by using the virtual accelerator, and then use it in the real accelerator directly. Python has a package for EPICS channel access also, so the virtual accelerator application can be closed in Python. That makes reduction of the time from application design to release.

CONCLUSION

The Python interface for the numerical simulation codes will benefit both the end user and the application developer. In this paper, we have shown the code design of both FLAME and Advanced-IMPACT including the folder structure for

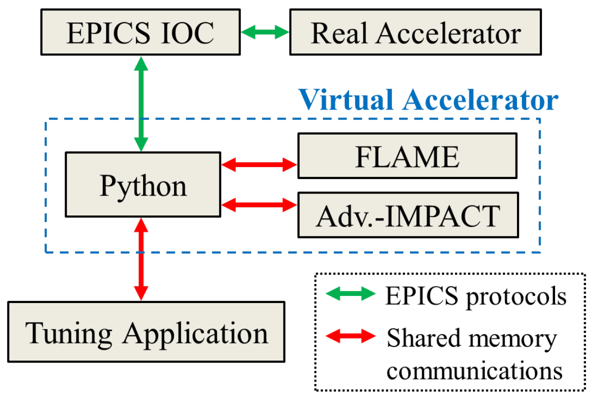

Figure 1: Schematic for virtual accelerator and application.

compiled Python module and the interface. The compiled module is the best solution for keeping both the data communication flexibility and the computational efficiency. It can be the next standard for the overall simulation codes.

REFERENCES

[1] Python, https://www.python.org/

[2] Sphinx, http://www.sphinx-doc.org/en/master/

[3] Z. He *et al*, in *in Proc. 28th Linear Accelerator Conf. (LINAC'16)*, East Lansing, MI, USA, Sep. 2016, pp. 100-103, doi:10.18429/JACoW-LINAC16-MOPRC015

[4] FLAME source code repository, https://github.com/frib-high-level-controls/FLAME

[5] FLAME API documentation, https://kryv.github.io/FLAMEdoc/

[6] J. Qiang, R. D. Ryne, S. Hbib, and V. Decyk, *J. Comput. Phys.*, vol. 163 p. 434, 2000.

[7] IMPACT, http://amac.lbl.gov/~jiqiang/IMPACT/

[8] F2PY, https://docs.scipy.org/doc/numpy/f2py/

[9] SciPy, https://www.scipy.org

[10] EPICS, https://epics.anl.gov/