DYNAC: EXTENSIONS, UPDATES, AND UPGRADES

Stephen Molloy[*], Eugene Tanke, European Spallation Source, Lund, Sweden

Abstract

DYNAC [1] is a multi-particle beamline simulation code suitable for modelling of the motion of protons, heavy ions, or electrons, moving through linear accelerators and beam transport lines. In this paper, we document extensions written in Python. It will be shown how these Python extensions add a considerable amount of flexibility to DYNAC, while maintaining the calculation speeds available from the core Fortran source. Real-world use-cases are discussed. In addition, some improvements that have been made to the DYNAC source are reported.

INTRODUCTION

DYNAC is a robust multi-particle simulation code for single-pass beamlines, with a long history of development, and users in many accelerator laboratories.

DYNAC operates by reading in a text file containing a list of commands. The Fortran-based executable reads each of the lines in turn, and performs the necessary actions. For example, the command,

```
QUADRUPO
   3.5 -1.34 0.75
```

will act on the phase-space of the simulated particles in a 0.75 cm aperture radius quadrupole with an effective length of 3.5 cm, and a pole-tip field strength of -1.34 kG.

Other commands exist for many other element types, as well as to output phase-space plots, emittance plots, and many other actions. The DYNAC manual [2] contains a full description of all possible commands.

The sequential style of data input to DYNAC – that is, that the DYNAC executable reads and acts on each line of input in turn – leads to the possibility of manipulating the flow of data into DYNAC programmatically.

The most recent update to DYNAC included an additional input flag that can alter the way in which input is provided to the executable. Instead of an input file that will be opened and read by the executable, the data may be streamed in via an operating system ‘pipe’. This simple change opens up an alternative range of possibilities by allowing a different style of usage of the DYNAC executable.

DYNAC IN PYTHON

Python is a language that is very widely used in the scientific community, and so is a very good choice for an extension of DYNAC. A Python wrapper – ‘Pynac’ – has been written that allows the use of DYNAC from a Python environment.

Pynac is distributed via the Python Package Index using `pip install Pynac`.

[*] stephen.molloy@esss.se

An Ephemeral DYNAC Server

The basic method of operation of Pynac is via a Python class that encapsulates the main sources of information needed for DYNAC. Then, when DYNAC is required to compute a result, a DYNAC instance is started as a Python subprocess. Python can then pipe the various commands needed by DYNAC into this subprocess, which will then perform the standard DYNAC operations.

In practice, this is accomplished by the following Python code,

```
self.dynacProc = subp.Popen(
            ['dynacv6_0','--pipe'],
            stdin=subp.PIPE,
            stdout=subp.PIPE,
            stderr=subp.PIPE
        )
```

This calls the ‘`dynacv6_0`’ executable with the ‘`--pipe`’ flag to indicate that the input will come from stdin rather than an input file. If successful, the result will be an object, `self.dynacProc`, that behaves very much like a short-lived DYNAC server. That is, a process that is instantiated as needed, and only lives as long as it is needed. Data can be written to the stdin attribute of this process, which will consume it just as if it came from an input file. This may be thought of as a type of ephemeral server process.

Note that these low-level operations are hidden from the user. While the open nature of Python means that a user can delve down to this level of detail if they want, the Pynac class is designed to be used at a much higher level.

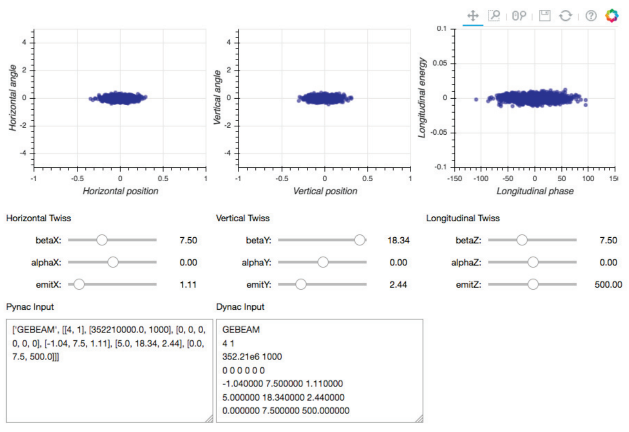
EXAMPLES

Jupyter notebooks [3] provide a powerful interactive environment for using Pynac, so the following examples will be shown as Jupyter notebooks.

Beam Construction

A basic need for almost all modelling codes is to provide a way for the user to specify the beam parameters at the input of the transport line. DYNAC provides several ways to do this, one of which, `GEBEAM`, allows the user to specify the Twiss parameters for a beam that is Gaussian in all six degrees of freedom.

Pynac provides a utility function to construct a GUI in the Jupyter notebook that allows the user to vary the Twiss parameters, and immediately see the phase-space plots of the resulting beam. In addition, the relevant DYNAC & Pynac commands are also provided so that these can be copied into the appropriate input file for use in a simulation. This is shown in figure 1.



Horizontal Twiss
betaX: 7.50
alphaX: 0.00
emitX: 1.11

Vertical Twiss
betaY: 18.34
alphaY: 0.00
emitY: 2.44

Longitudinal Twiss
betaZ: 7.50
alphaZ: 0.00
emitZ: 500.00

Pynac Input

```
['GEBEAM', [[4, 1], [352210000.0, 1000], [0, 0, 0, 0, 0, 0], [-1.04, 7.5, 1.11], [5.0, 18.34, 2.44], [0.0, 7.5, 500.0]]]
```

Dynac Input

```
GEBEAM
4 1
352.21e6 1000
0 0 0 0 0 0
-1.040000 7.500000 1.110000
5.000000 18.340000 2.440000
0.000000 7.500000 500.000000
```

Figure 1: Beam construction GUI implemented with Pynac.

Each time the user updates a Twiss parameter, a DYNAC process is called with the simplest possible input – the relevant `GEBEAM` command, and a command to output the particle phase-space. This output is then consumed by Pynac to generate the Jupyter notebook plots.

Single Run with Plotting

A standard use case of DYNAC is to run a particular input file, and then immediately view the output plots created by any plotting commands included in that file. This is easily accomplished in Pynac using the usual `Pynac` class to perform the simulation, and the `PynPlt` class to generate the plots expected by experienced DYNAC users.

An example of such a plot is shown in figure 2.

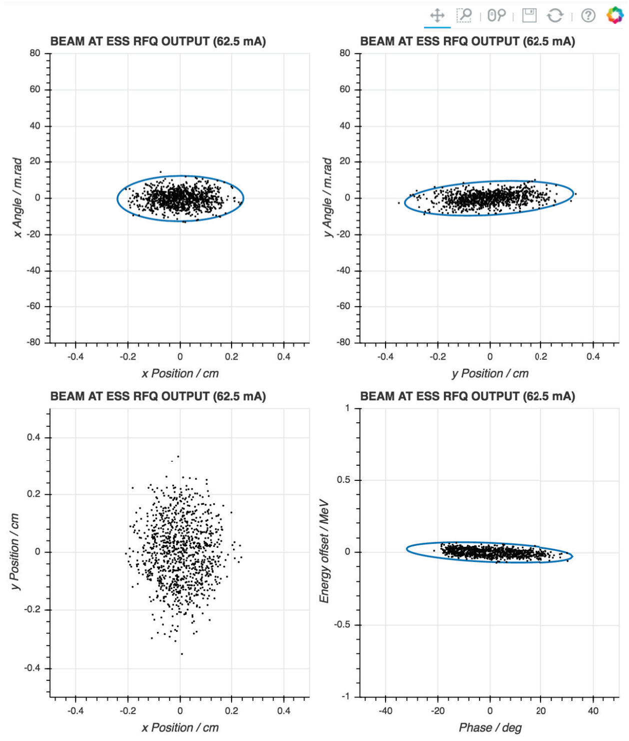


Figure 2: An example of a typical Pynac plot.

Parallel Operation

Since Pynac is pure Python (excepting the Fortran source in DYNAC), it can make use of the the multithreading and multiprocessing capabilities in the Python Standard Library.

As a basic (but still useful) example, consider the study of the effect of quadrupole gradient errors on the growth of the emittance throughout a linac. This is typically performed by generating a large number of linacs, each with a different set of errors, and modelling the dynamics of the beam as it moves through the lattice. Each of these linac simulations is completely uncoupled from the others – that is, the results of a particular simulation are in no way influenced by those of another. Therefore, this problem can be split between multiple processing cores, enabling a substantial speed-up in the simulation time.

Pynac provides a utility function, `multiProcessPynac`, to take care of the details of multiprocessing. As input, this function takes a list of the input files that DYNAC will need for the simulation (e.g., the command file, field definitions, etc.), and a function to perform for each of the simulations (e.g., to apply the random errors, and track the beam). In addition it needs to be provided with the number of iterations to run, and the number of processing threads to make use of. These last two inputs default to 100 and 8 respectively.

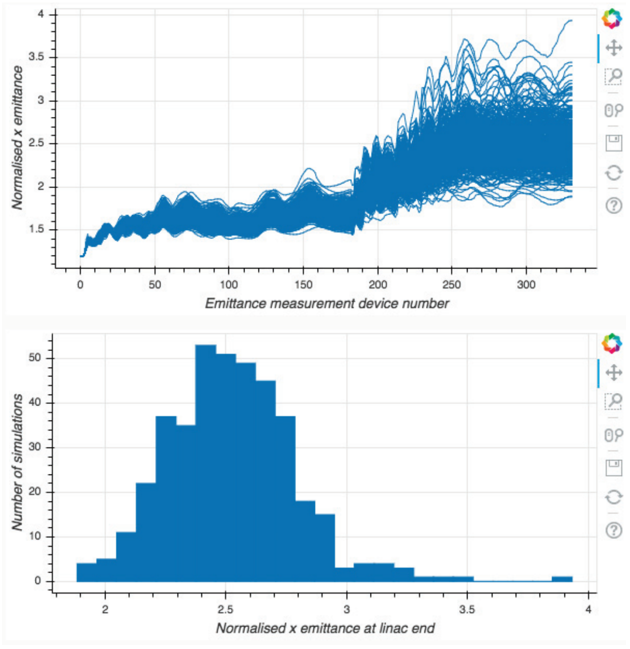


Figure 3: Error studies of an ESS linac lattice with random errors in the quadrupole field.

To demonstrate the operation of the multiprocessing capabilities of Pynac, this function was used in a simulation of the performance of the ESS linac when subjected to quadrupole field errors. Particle tracking was done with 1000 macroparticles for 400 error cases, and the results are shown in the figure 3. Since this calculation was done as a demonstration of the operation of the multiprocessing capabilities of Pynac, the official lattice was not used. This means that no interpretations of the physics of the ESS design can be made from these results.

CODE STRUCTURE

The Pynac source code is licensed with the GPL, and can be browsed in the GitHub repository [4]. Documentation is hosted online [5], with automated links to GitHub to keep the code and documentation aligned. In addition, the testing is automated in a way that allows users to ensure that their version of Python is compatible [6].

In order to avoid excessive nesting of imports, the source consists only of four files (excluding documentation, examples, the test suite, etc.).

Core.py

This file contains the primary simulation functionality of Pynac. As can be seen from the foregoing examples, the two key items in this file are the `Pynac` class and the `multiProcessPynac` function, however the `Builder` class is also useful for interactively exploring some of the physics of a charged particle lattice[1].

The `Pynac` class is instantiated in one of two ways:

1. Providing the name of a DYNAC input file to the constructor, for example, `Pynac(filename='dynacfile.in')`. This will parse the file into Pynac's internal format.
2. By supplying a lattice in Pynac's internal format (perhaps from another simulation instance) to a `from_lattice` class method.

Plotting.py

The primary purpose of this file (containing only one class – `PynPlt`) is to duplicate the action of the `plotit` command provided by DYNAC. That is, to produce all the plots asked for in the DYNAC input file.

This functionality will be used mostly when performing quick simulations. The intent of Pynac is to allow a more flexible approach to data analysis, and therefore it is more likely that the user will make use of traditional Python plotting tools, e.g., Matplotlib, Bokeh, Plotly, etc.

DataClass.py & Elements.py

These files contain the definitions of some types used by Pynac. They should not be thought of as internally defined types, since it is expected that the user will want to make use of many of these in order to manipulate their lattice. For example, the classes defined in `Elements.py` are the standard accelerator elements that would be expected to appear in an accelerator tracking code[2].

[1] Note that, as of the time of publication of this paper, this class is still in development.

[2] These elements have been authored 'as needed', and so currently lags considerably behind the library of elements available in DYNAC.

DYNAC IMPROVEMENTS

In addition to the aforementioned new `--pipe` option in DYNAC R17, which allows to read DYNAC cards from standard input, a new type code, `MHB` (Multi-Harmonic Buncher) has been added. With `MHB` the motion of particles crossing a multi-harmonic buncher can be simulated in a thick lens approximation. It was already possible to do so using the thin lens approximation. The base frequency and as many as 4 harmonics can be simulated, whereby the motion of the particles is calculated with a numerical method akin to the one used with the `CAVNUM` type code.

Currently under preparation is the possibility of running multiple instances of DYNAC at the same time, in view of error studies. A framework for this has been prepared and successfully tested on multi-core Windows and Linux machines. The framework also works on Macs, but currently not in multi-core mode. This framework will be made available alongside the next DYNAC release and is straightforward in implementation.

CONCLUSION AND OUTLOOK

In conclusion, Pynac is a new development allowing the use of DYNAC from within a Python environment. There are still a significant number of capabilities to be developed – in particular a full suite of Pynac representations of DYNAC lattice elements – and work is progressing on this. The intention is to allow the direction of code development to be strongly influenced by user requests rather than by an artificial timeline.

Several improvements in DYNAC were also presented, as well as an announcement of the upcoming multi-core framework currently under preparation.

REFERENCES

[1] `https://dynac.web.cern.ch/dynac/dynac.html`

[2] *DYNAC V6R17 User Guide*, P. Lapostolle, S. Valero, E. Tanke, `http://dynac.web.cern.ch/dynac/beta/dynacb.html`

[3] `http://jupyter.org/`

[4] `https://github.com/se-esss-litterbox/Pynac`

[5] `http://pynac.readthedocs.io/en/latest`

[6] `https://travis-ci.org/se-esss-litterbox/Pynac`