ISBN: 978-3-95450-253-0 ISSN: 2673-5571 doi:10.18429/JACoW-HB2023-THBP32

Xobjects AND Xdeps: LOW-LEVEL LIBRARIES EMPOWERING BEAM DYNAMICS SIMULATIONS

S. Łopaciuk*, R. De Maria, G. Iadarola, CERN, Geneva, Switzerland

Abstract

Xobjects and Xdeps are Python libraries included in the Xsuite beam dynamics simulation software package. These libraries are crucial to achieving two of the main goals of Xsuite: speed and ease of use. Xobjects allows users to run simulations on various hardware in a platform-agnostic way: with little user intervention single- and multi-threading is supported as well as GPU computations using both CUDA and OpenCL. Xdeps provides support for deferred expressions in Xsuite. Relations among simulation parameters and functions driving properties of lattice elements can be defined or indeed imported from other tools such as MAD-X and then easily updated before or during the simulation.

INTRODUCTION

Xsuite [1, 2] is a relatively recent collection of tools for conducting beam dynamics simulations in particle accelerators. The main focus in the development of Xsuite has been to create a versatile tool allowing the users to easily, and with high performance, conduct particle tracking simulations. Xsuite's ease of use comes from the fact that it is a framework that can be interfaced with in Python 3, and, optionally, C. The former of the two languages is regarded as particularly user-friendly and already well-known in the community, while the latter is used for writing parts of Xsuite where performance is critical. Xsuite aims to be performant by relying on hardware-accelerated computation contexts (GPU and CPU multi-threading), which is complemented by the ease of set-up of a simulation: e. g. an existing lattice model defined in MAD-X can often be directly imported into Xsuite.

A particle accelerator lattice model consists of a series of beam elements (collectively referred to as a 'line'), each of which abstracts a real-life machine element. The parameters of each of the elements (e. g. the strength of a magnet) can depend on various factors, and can be driven by hyper-parameters of the simulation (as an example of a hyper-parameter consider the crossing angle of two beams in a collider); these hyper-parameters are often referred to as 'knobs'. The relationships between all of the parameters are preserved in Xsuite thanks to the Xdeps package. With Xdeps the user can inspect the simulation parameters, as well as efficiently change them before or during a simulation. Simulation runs are themselves enabled by the Xobjects package which is responsible for memory management, tracking code generation and its execution regardless of the chosen computation context.

In this paper we explore these two packages to give a more in-depth overview of their design and capabilities.

* szymon.lopaciuk@cern.ch

XOBJECTS

Beam elements can be naturally divided into two categories: *non-collective*, where the computation of the new coordinates of the tracked particles depend solely on its previous coordinates and the element itself, and *collective*, where the computed coordinates are linked to the coordinates of other particles in the bunch. Tracking a bunch through a non-collective element can be easily accelerated through the application of parallelisation: for every particle the same computation is performed independently.

Parallelisation can be achieved through CPU or GPU-based multi-threading; however, the development of portable parallelisable code is non-trivial, as there are currently different, sometimes competing, technologies, while hardware vendors tend to support some, but not all of them. In addition to the conventional serial CPU execution context, Xobjects aids in writing procedures that simultaneously support CPU-based multi-threading through OpenMP [3], as well as GPU platforms compatible with CUDA [4] or OpenCL [5].

Universal API

In essence, Xobjects provides a simple object-oriented programming interface which allows for defining binary objects in Python, and for defining their methods in an extended C syntax. Xobjects comes with a set of built-in types:

- the standard collection of integer types (signed and unsigned; 8, 16, 32, and 64 bit),
- floating-point types (32 and 64 bit real; 64 and 128 bit complex),
- multi-dimensional array types (fixed and dynamic shape; support for arbitrary strides),
- structure types, union types, references, strings.

For any compound type (i. e. structure or array) Xobjects can generate a C API that can be used to interact with the 'Xobject' from a C 'method'. Consider the following example of a structure representing an array of vectors (`xo.Float64[:]` is an dynamic-length array of double-precision real numbers):

```
import xobjects as xo
class Vectors(xo.Struct):
    x = xo.Float64[:]
    y = xo.Float64[:]
```

Xobjects can be used to automatically generate methods for getting the size of the arrays comprising our object, as well as getters and setters for their individual elements:

Content from this work may be used under the terms of the CC-BY-4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-253-0 ISSN: 2673-5571 doi:10.18429/JACoW-HB2023-THBP32

Content from this work may be used under the terms of the CC-BY-4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

```
int64_t Vectors_len_x(Vectors obj);
double Vectors_get_x(
    const Vectors obj,
    int64_t i0);
double Vectors_set_x(
    const Vectors obj,
    int64_t i0,
    double value);
```

With these functions, we can now, for example, write a parallel procedure to calculate all the individual vector lengths and store them in an Xobjects array (the type `ArrayNFloat64` used in the example below is the C equivalent of the Xobjects type `xo.Float64[:]`):

```
/*gpufun*/ void get_lengths(
    VectorsData v,
    ArrayNFloat64 result
) {
    int64_t n = Vectors_len_x(p);
    int64_t i = 0;
    //vectorize_over i n
    double x, y, l;
    x = Vectors_get_x(v, i);
    y = Vectors_get_y(v, i);
    l = sqrt(x*x + y*y);
    ArrayNFloat64_set(result, i, l);
    //end_vectorize
}
```

The implementation of `get_lengths` need only to be written once: due to the special annotation `//vectorize_over`, Xobjects can generate code that is compatible with serial execution, OpenMP, CUDA, and OpenCL as needed. If a multi-threaded context (either CPU or GPU) is chosen any code appearing between `//vectorize_over i n` and `//end_vectorize` will run on its own i-th among n threads; otherwise, Xobjects will simply insert a for loop. In this case, the compilation would be performed at runtime, however Xsuite also makes it possible to prebuild and store common tracking kernels.

In-Memory Serialisation

Not only is the API used to interact with Xobjects context-agnosticly, the same is true of the memory layout of the different types: an `xo.Struct` with a set of fields will be the same, no matter if it is on the CUDA context or CPU. In particular, the powerful array type of Xobjects is fully compatible with NumPy [6] arrays, and on CPU contexts it is in fact a wrapper around them. The custom striding that can be defined on arrays can be particularly helpful if interfacing with other Fortran-based code is desired.

On the GPU contexts Xobjects relies on the libraries CuPy [7] and PyOpenCL [8], which are Pythonic wrappers around CUDA and OpenCL, respectively. Each of these libraries provides an array interface analogous to the one provided by NumPy, which means that interacting with arrays on different context in Python is also largely transparent.

Besides being able to rely on out-of-the box solutions for the array interface, the more obvious benefit of the binary compatibility of Xobjects on different contexts is their portability. Whenever a certain computation is faster on a certain context (e. g. a non-collective tracking on GPU), it is trivial to move the data buffer containing the Xobjects to the other context; when a further computation benefits from a standard CPU context (e. g. due to collective effects, or a non-parallelisable algorithm) the data can be moved again.

Application in Xsuite

The above-described parts all come together in the tracking package of Xsuite: Xtrack. Beam elements are implemented as Xobjects structures, and the same is true for the particle ensemble object. Each element comes with a C tracking procedure (or Python, if performance is not key). Whenever possible the tracking is performed using a compiled kernel on the target context, and otherwise the context is changed to the CPU where a serial tracking function is used. Due to the common Xobjects API it is easy to implement one's own beam element: all that is needed to extend the built-in behaviour of Xtrack is a set of fields representing the parameters of the element and a tracking function implementation.

XDEPS

Users of the particle accelerator design program MAD-X [9] are certainly accustomed to the concept of deferred expressions. In MAD-X there are two types of assignments: immediate and deferred. When defining a value using the first kind of assignment the expression on the right-hand side of the equality sign is immediately evaluated and assigned to a parameter or variable. A value defined with the second kind of assignment is re-evaluated whenever any of the variables appearing in the expression change. As outlined in the introduction, this mechanism is useful when defining low-level lattice properties using higher-level parameters that are subject to change. Xdeps is the package implementing this functionality in Xsuite.

Design

While an internal part of MAD-X, the way deferred expressions are implemented in the Xdeps package is completely agnostic to the environment in which they are used. Xdeps can be employed to update properties of any Python object, which makes it both versatile and easier to maintain.

There are three key concepts in the design of Xdeps:

Manager A manager registers external Python objects with Xdeps and orchestrates the actions performed by Xdeps.

Task A task describes an action that modifies the values of a set of references (targets) depending on the values stored in another set of references (dependencies) and potentially its internal state.

Expression An expression represents, and can be built by, a generic Python expression, which can contain values, containers, and other expressions. It can represent references to slots inside containers to describe targets and dependencies of tasks, can trigger updates when values are set into it, and can define a task when an expression of references is assigned to another reference.

ISBN: 978-3-95450-253-0 ISSN: 2673-5571 doi:10.18429/JACoW-HB2023-THBP32

An Xdeps manager internally holds the graph of the relationships between references and tasks, which allows for fast updates of values, as well as introspection and code generation.

An Example

To visualise the preceding points, we set up the following example, also shown in Fig. 1: we have a namespace containing a pair of Cartesian coordinates, x and y, and we will use Xdeps to add a pair of planar coordinates, r and θ, that will be kept up to date whenever x or y is changed.

```
import xdeps as xd, math
from types import SimpleNamespace

pt = SimpleNamespace(x=1, y=1)

# Set up manager and reference
mgr = xd.Manager()
pt_ = mgr.ref(pt, 'pt')
math_ = mgr.ref(math, 'math')

# Define expression-based tasks
pt_.r = math_.sqrt(pt_.x**2 + pt_.y**2)
pt_.th = math_.atan2(pt_.y, pt_.x)

# Change x
pt_.x = 0

# Will now return 0, 1.0 and pi/2:
print(pt.x, pt.r, pt.th)
```

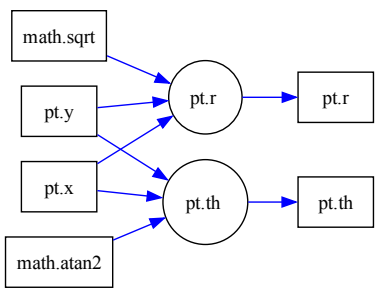

Figure 1: The graph of expressions (boxes) which are dependencies and targets of tasks (circles). The plot can be generated by Xdeps with `mgr.plot_deps()`.

In the above example we first set up two reference objects to serve as proxies to accessing the underlying Python objects. The first one, `pt_`, is for our data contained in the namespace `pt`. From now on, any change we wish to make to `pt` should be done through `pt_`: this is to ensure that Xdeps is aware of changes, and can execute the right actions. Since we also need mathematical functions, we create a reference to the `math` module (otherwise, writing `sqrt(pt_.x)` throws an error, as `pt_.x` is a reference object, not a number). A user can also implement their own functions as well: currently built into Xdeps (in the `functions` module) is a linear interpolator function (it can be given a table of (x, y) values) that can serve as a template for implementing more.

Importing from MAD-X

While the example in the previous section is a helpful demonstration of the basic features of Xdeps, the main interest comes from the ability of importing the expressions from existing models defined in MAD-X: this is accomplished in the `madxutils` module of Xdeps. In said module we have the formal grammar of MAD-X deferred expressions, which is used to generate a parser of these expressions using Lark [10]. The parser is then used to read in the deferred expression, building the graph of tasks and references in a manner similar to the one described above.

The Optimiser

Xdeps also hosts the optimiser module used by Xsuite for its optics matching functionality. At its core it uses the same Jacobian optimisation algorithm as the one tried and tested in MAD-X [11], but its user interface makes it easier to exercise fine-tuned control over the matching process.

To begin, the target conditions of the match need to be specified, together with their tolerances and weights, as well as the parameters which should be varied, together with their limits, weights, and the desired initial step size. The vary defaults can also be specified on the line in advance, to avoid duplication. By default, the match is performed against the results of a twiss, however any action returning some observables to optimise on can be specified.

Unless otherwise specified, the match will be performed and its results applied to the line, as long as the solution can be found within the specified tolerances. To help introspection, both the tolerance check, and the update of the parameters in the line can be overridden. The summary of the match can be printed (in a human-readable table also implemented in Xdeps) to see exactly how far off the results are from the targets, and similarly, a log of the match can be viewed to see all the steps of the iteration up to that point. The match state can be rolled-back to a previous point in history, and certain targets or parameters to vary can be switched on or off; a single step of the match can also be performed if needed.

CONCLUSION

Xobjects and Xdeps are two vital packages that are part of the Xsuite beam dynamics simulation toolkit.

Xobjects provides an abstract interface to interacting with beam elements and their tracking code, regardless of the execution context on which the simulation is run, be it single- or multi-threaded, CPU or GPU.

Xdeps is the deferred expressions engine behind Xsuite, which enables parametrisation of accelerator lattices designed in and imported from MAD-X. It also hosts the powerful matching optimiser used in Xsuite, which allows for a high degree of fine-tuning and introspection to the matching procedure.

Content from this work may be used under the terms of the CC-BY-4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-253-0 ISSN: 2673-5571 doi:10.18429/JACoW-HB2023-THBP32

REFERENCES

[1] Xsuite Documentation, CERN, Geneva, Switzerland, 2023. https://xsuite.readthedocs.io/en/latest/

[2] G. Iadarola *et al.*, "Xsuite: An integrated beam physics simulation framework," presented at HB'23, Geneva, Switzerland, Oct. 2023, paper TUA2I1, these proceedings.

[3] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998. doi:10.1109/99.660313

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *Proc. ACM SIGGRAPH'08*, Los Angeles, CA, USA, Aug. 2008, p. 16. doi:10.1145/1401132.1401152

[5] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, 2010. doi:10.1109/MCSE.2010.69

[6] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020. doi:10.1038/s41586-020-2649-2

[7] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-Compatible library for NVIDIA GPU calculations," in *Proc. NIPS'17*, Long Beach, CA, USA, Dec. 2017. http://learningsys.org/nips17/assets/papers/paper_16.pdf

[8] A. Klöckner *et al.*, *PyOpenCL*, version v2022.1.3, Zenodo, 2022. doi:10.5281/ZENODO.6533956

[9] L. Deniau, H. Grote, G. Roy, and F. Schmidt, *The MAD-X Program (Methodical Accelerator Design). User's Reference Manual*, CERN, Geneva, Switzerland, v5.08.01, 2022. https://mad.web.cern.ch/mad/webguide/manual.html

[10] E. Shinan *et al.*, *Lark – a parsing toolkit for Python*, 2017. https://github.com/lark-parser/lark

[11] R. de Maria, F. Schmidt, and P. K. Skowronski, "Advances in Matching with MAD-X.," in *Proc. ICAP'06*, Chamonix, Switzerland, Oct. 2006, pp. 213–215. https://jacow.org/icap06/papers/WEPPP14.pdf

Content from this work may be used under the terms of the CC-BY-4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI