ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA177

TENSORICS - A JAVA LIBRARY FOR MANIPULATING MULTI-DIMENSIONAL DATA

K. Fuchsberger, A. Calia, J.-C. Garnier, A. Gorzawski, M. Hostettler, K. Krol
CERN, Geneva, Switzerland

Abstract

Accelerator control software often has to handle multi-dimensional data of physical quantities when aggregating readings from multiple devices (e.g. the reading of an orbit in the LHC). When storing such data as nested hashtables or lists, the ability to do structural operations or calculations along an arbitrary dimensions is hampered. Tensorics is a Java library that provides a solution to these problems. A Tensor is an n-dimensional data structure, and both structural (e.g. extraction) and mathematical operations are possible along any dimension. Any Java class or interface can serve as a dimension, with coordinates being instances of a dimension class. This contribution will elaborate on the design and the functionality of the Tensorics library and highlight existing use cases in operational LHC control software, e.g. the LHC luminosity server or the LHC chromaticity correction application.

INTRODUCTION

A common need in applications that manipulate numerical data is to organize them in data structures which allow easy transformations and calculations. This paper describes the *tensorics* [1] library for the Java programming language which provides several, complementary concepts to ease such tasks. Despite the libraries name is derived from "tensor", it contains several additional concepts which complement each other. The features are designed to work smoothly together, but each of them can of course also be used standalone. In the following sections, we give a short overview on the different concepts, together with some explanatory code examples.

TENSORS

The name “Tensorics” is derived from “Tensor”. Loosely speaking, a tensor in mathematics is a multidimensional data structure, whose dimensionality is given by the number of indices. A tensor of dimensionality N contains a value for each N-tuple of index values. Tensors in mathematics are usually denoted by noting their elements with a full set of indices. E.g. an element of a 3-dimensional tensor $\mathbf{A}$ would be denoted as a_{ijk}. Each index (i, j, k) can potentially have its own range (e.g. $1 <= i <= M_i$, $1 <= j <= M_j$, $1 <= k <= M_k$).

Another way to see this is that a tensor has a value for each point in an N-dimensional integer space. In the above notation a dimension is identified by the position of the respective index, and the coordinate in that dimension is given by the value of the index. These mathematical concepts are extremely useful, especially when it comes to operations on such tensors (as we see in later sections). Therefore, tensorics borrows many concepts from mathematics. At the same time it translates them into the programming language in a way that is aimed to form a powerful data structures which encourages readable code as much as possible and helps avoiding confusion and mistakes. For this reason, we use the word “Tensor” in an even sloppier manner.

The main particularity of a tensorics *tensor* is that a dimension is not identified by the position of the index, but by a java type (class). Instances of the respective type we denote as *coordinates*. A point within the N-dimensional coordinate space is then defined by a set of objects (instances of coordinate classes), of which each type must be exactly once. This key concept allows easier and less error-prone usage (because the order of the coordinates/indices is not relevant) and still leads to readable code.

A tensorics tensor has one type parameter, the type of the values it contains, usually denoted as `<V>`. Therefore, the tensor data structure can be used as container for any Java type. However, some operations on the tensors will be only possible for certain value types (e.g. mathematical operations).

An Example

Since tensorics concepts and syntax are best explained in a practical walk-through, we will use the following example throughout the subsequent sections:

Consider weather analysis: A data set consists of weather data from different cities and times. The class City and Time are defined and some constants are instantiated. Temperature values are stored in a tensor of doubles, for example:

Listing 1: Constants for examples.

```
City SF = City.ofName("San Francisco");
City LA = City.ofName("Los Angeles");

Time T1 = Time.of("2017-01-01 15:00");
Time T2 = Time.of("2017-01-02 15:00");

Tensor<Double> degrees;
/* creation omitted */
```

Accessing Values Assuming the above constants, we can then simply get temperature values from the tensor:

Listing 2: Accessing Tensor Values.

```
Double t = degrees.get(T1, SF);
```

As visible here, this looks very similar to getting values from a map, with the following important differences:

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA177

- The get method of a tensor accepts *N* arguments, one for each dimension.
- The get method of a tensor never returns `null`. It will throw an appropriate exception in case there is no value available in the tensor for the given set of coordinates.

In general, it shall be noted that all methods within the tensorics library are designed to fail fast. This is particularly important because tensorics, due to its flexible API, cannot rely on compile-time checks in many cases and thus some errors only appear at runtime.

The set of *N* coordinates is called a *position* in tensorics. Thus, the code from Listing 2 is equivalent to

Listing 3: Accessing Tensor Values through position.

```
Position position = Position.of(T1, SF);
Double t = degrees.get(position);
```

Main Entry Point The interfaces of tensorics objects is kept very slim and usually only provide the absolutely necessary methods. All the other operations on these objects is based on static methods operating on them. The main entry point for these methods (containing all the methods which are not specific to certain value types) is the class `Tensorics`. This class contains also, for example, a delegation method to the `Position.of()` method:

Listing 4: Alternate factory method for position.

```
Position position = Tensorics.at(T1, SF);
/* with static import: */
Position position = at(T1, SF);
```

Using a static import for this, allows concise code which will be particularly important when creating tensors.

Note: In all the following code examples, we assume that, whenever there is a plain method call, then it is a static method from the `Tensorics` class (or in other words that `Tensorics.*` is imported statically).

Creating Tensors All currently available implementations of tensors are immutable. The usual way to create them is through builders. For example, to create our temperature tensor and put 4 values into it, we would have to do something like:

Listing 5: Building a tensor.

```
Tensor<Double> degrees =
    builder(City.class, Time.class)
          .put(at(SF, T1), 12.5)
          .put(at(SF, T2), 14.2)
          .put(at(LA, T1), 17.5)
          .put(at(LA, T2), 19.2)
          .build();
```

Again, the syntax is very similar to building an immutable map. And indeed this is another way how a tensorics tensor can be seen: As a map from position to a value - and it can be transformed into one:

Listing 6: Tensor as map.

```
Map<Position, Double> degreesMap =
            mapFrom(degrees);
```

Scalar A tensor can have zero dimensions. This particular tensor we denote as *scalar* in tensorics. It has exactly one value at the position `Position.empty()`. A scalar can simply be created using the static factory method

Listing 7: Creating a scalar.

```
Scalar<Double> scalar = scalarOf(2.5);
```

Structural Operations

Up to now, we were simply using a tensor as a kind-of map with combined keys. However, the real power is unleashed only when it comes to transformations. For this it is useful to understand on additional concept:

Shape Just like a map has its set of keys, a tensorics tensor has a shape. It basically describes the structure of the tensor, without its values. Basically it contains the following information:

- The dimensions of the tensor (e.g. `Time.class` and `City.class` in the above example) and
- The available positions in the tensor.

The shape can be retrieved from the tensor and used for our example like the following:

Listing 8: Shape of a tensor.

```
Shape shape = degrees.shape();

Set<Class<?>> dims = shape.dimensionSet();
/* Contains Time.class and City.class */

int dim = shape.dimensionality();
/* Will be 2 */

Set<Position> poss = shape.positionSet();
/* contains the 4 positions */

int size = shape.size();
/* Will be 4 */
```

Extracting Subtensors One very common structural operation is extracting sub-tensors from a tensor:

Listing 9: Extracting Subtensors.

```
Tensor<Double> sfDegrees =
  from(degrees).extract(SF);
```

This will result in a 1-dimensional tensor, only containing coordinates of type `Time`. The complementary operation to this is called *merging* tensors.

Note: while in the `get` method, the number of coordinates always has to exactly match the dimensionality of the tensor

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA177

(otherwise the method will throw), the `extract` method takes any subset of the dimensions as argument; the `get` method returns the values of the tensor, while the `extract` method returns again a tensor. This implies that if coordinates for all dimensions are provided as arguments for the extract method, then a zero-dimensional tensor is returned. The returned tensor can be empty in case no elements exist at the extracted coordinates.

Mathematical Operations

One important motivation to use tensors is of course to have simple and intuitive ways to perform mathematical operations on them. While the structural operations - as described up to now - can be performed on tensors of any value types, it is clear that mathematical operations can be only done with tensor values of particular types.

Mathematical Structures Tensorics does not strictly restrict the types on which mathematical operations can be performed, but provides an extension mechanism through which - in principle - the mathematical capabilities can be added for any value type. In practice this makes only sense (and is only necessary) for a limited number of value types. The extension mechanism requires to provide (with a, b, c being tensor values):

- Two binary operations, addition (+) and multiplication (*) with the following properties:
  - both, + and * are associative: $a + (b + c) = (a + b) + c$; $a * (b * c) = (a * b) * c$.
  - both, + and * have an identity element (Called '0' for +, '1' for *): $a + 0 = a$; $a * 1 = a$.
  - both, + and * have an inverse element (Called '-a' for +, '1/a' for *): $a + (-a) = 0$; $a * 1/a = 1$.
  - both, + and * are commutative: $a + b = b + a$; $a * b = b * a$.
  - * is distributive over +: $a * (b + c) = a * b + a * c$.

  Mathematically speaking, the two operations form the algebraic structure of a *field* [2] over the tensor values `<V>`.
- Two additional binary operations: Power (a^b) and Root ($\sqrt[b]{a}$).
- A conversion function of the tensor values to and from doubles.

If these operations are provided to generic support classes of tensorics, then all the manipulations based in the following will be available by inheriting from these support classes. The biggest advantage of the approach used in tensorics for defining a field (and using external methods for calculations - not methods of the field elements) is that it (technically) does not impose any constraints on the value type and thus avoids e.g. wrapper objects as necessary in the field-implementations of other math libraries (e.g. Apache Commons Math [3]).

Out of the box, tensorics currently provides an implementation of these requirements for doubles. To simplify these very frequently required operations, it provides also a convenience class (`TensoricsDoubles`) with static delegation methods to the support classes. Such convenience will not be available out of the box for custom value types, but can be easily added in a similar way. Whenever there is trailing method call in the following examples, we will assume that it is a static method from the class `TensoricDoubles`.

Unary Operations Next to operations on tensors, the support classes also provide convenience operations for iterables. For example:

Listing 10: Unary Operations on Iterables and Tensors.

```
Iterable<Double> v = Arrays.asList(1.0, 2.0);
Iterable<Double> negv = negativeOf(v);
Double vsize = sizeOf(v);

Tensor<Double> t; /* creation omitted */
Tensor<Double> negt = negativeOf(t);
Double tsize = sizeOf(t);
```

Basic Statistics Some very simple statistical methods are provided out of the box. For iterables, the results are simply of type of the elements of the iterable:

Listing 11: Iterable statistics.

```
Iterable<Double> v = Arrays.asList(1.0, 2.0);
Double avg = averageOf(v);
Double sum = sumOf(v);
Double rms = rmsOf(v);
```

On the other hand, for tensors the application of statistical operations is usually done only in one dimension. This corresponds to a reduction of the tensor by one dimension. The provided fluent API reflects this (continuing our example from before):

Listing 12: Tensor statistics.

```
/* All these return Tensor<Double>: */
reduce(degrees).byAveragingOver(Time.class);
reduce(degrees).byRmsOver(Time.class);
reduce(degrees).bySummingOver(Time.class);
```

Binary Operations Calculating of operations between two tensors, finally makes the most use. These operations all start using the `TensoricDoubles.calculate(...)` method:

Listing 13: Tensor statistics.

```
/* degrees and offset are Tensor<Double> */
calculate(degrees).plus(offset);
calculate(degrees).minus(offset);
calculate(degrees).elementTimes(other);
calculate(degrees).elementDividedBy(other);
/* All these return Tensor<Double> */
```

Here both, the left and right operands are assumed to be tensors. However, bare values are also supported on both sides and will be implicitly be converted to scalars. The

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA177

four above-mentioned operations are the simplest ones, as they are based on element wise operations: Each element in the left tensor only requires the corresponding element in the right tensor to produce the corresponding element in the resulting tensor. However, this needs some other considerations: What happens if the two operands have different shapes? This problem can be treated in two stages, which are called *broadcasting* and *reshaping* in tensorics. They are explained in the following two sections. Tensorics has a very modular way to treat such cases: Different strategies can be used (and even implemented) by the user in special cases. If nothing is specified, a sensitive default will be used.

Reshaping This is the simpler of the two possible shape-inconsistencies: It means that both tensors in question have the same dimensions, but they have values for different positions (e.g. one has less entries than the other). The default behaviour for this case is, that the resulting tensor will have only values for the positions, which are contained in each of the tensor (The intersection of the position set).

Broadcasting The term *broadcasting* is borrowed from the python library *numpy* [4]. While the underlaying principle is very similar to the numpy one, there are several essential difference which comes from the fact that numpy uses multi-dimensional arrays with integer indices, while tensorics identifies its dimensions by classes: The default broadcasting strategy in tensorics broadcasts all dimensions which are *not* available in one tensor to the shape of the second tensor. In other words, a dimension which is not present in one, will be added to the other tensor and all coordinate values of the respective dimension will potentially be combined with all the positions of the other tensor. For example:

Listing 14: Tensor broadcasting.

```
Tensor<Double> temps = builder(Time.class)
  .put(at(T1), 10.5)
  .put(at(T2), 12.2).build();

Tensor<Double> offsets = builder(City.class)
  .put(at(SF), 2.0)
  .put(at(LA), 7.0).build();

Tensor<Double> result =
  calculate(temps).elementTimes(factors);
/* Will contain 4 positions:
   (SF, T1), (SF, T2), (LA, T1), (LA, T2)*/
```

The result will be exactly the same tensor as constructed in Listing 5. When performing binary operations, the two operands are first both broadcasted and then reshaped. This ensures that the dimensions are correct and then that all the relevant elements operate on their corresponding partners.

Inner Product This very particular multiplication of two tensors is basically the generalization of the matrix multiplication. The syntax is as simple as it can be:

Listing 15: Inner product syntax.

```
calculate(degrees).times(other);
```

To have this yield the expected results, co- and contra-variant dimensions have to be distinguished. In tensorics, this distinction is achieved by the following mechanism: By default, coordinates are assumed to be contravariant. Covariant coordinates are forced to inherit from the class `Covariant<C>`, where the generic parameter `<C>` is the type of the corresponding contravariant coordinate. Detailed information about this can be found in the tensorics source code documentation [5].

PHYSICAL QUANTITIES AND UNITS

Another very common problem in scientific applications is the proper treatment of units. At the current stage, tensorics currently uses internally an external library for this purpose (JScience [6]). However, as this library is not actively maintained anymore, it is foreseen to replace this implementation either by a different library or an internal implementation of physical quantities.

For this reason, tensorics already provides its own abstraction of units. A physical unit is represented by the class `Unit` and a value-unit pair is represented by the class `QuantifiedValue`. Factory methods for quantified values are available in the `Tensorics` class. Convenience overrides are provided which support both tensorics internal unit objects and JScience instances of units. Operations are available in the support classes for the corresponding value types, like for doubles e.g. in the class `TensoricDoubles`. With this, operations like the following are possible:

Listing 16: Quantities.

```
QuantifiedValue<Double> distance =
  Tensorics.quantityOf(10.0, SI.METER);

QuantifiedValue<Double> time =
  Tensorics.quantityOf(5.0, SI.SECOND);

QuantifiedValue<Double> speed =
  calculate(distance).dividedBy(time);
/* results in 2 m/s */
Double value = speed.value(); // 2.0
Unit unit = speed.unit(); // m/s
```

Also support methods to work with tensors of quantified values are provided, e.g.:

Listing 17: Tensors of Quantities.

```
Tensor<QuantifiedValue<Double>> measurement;
Tensor<QuantifiedValue<Double>> reference;
/* construction omitted */

Tensor<QuantifiedValue<Double>> difference =
  calculate(measurement).minus(reference);
```

Error and Validity Propagation

Especially when using tensors for measured values, it is important to understand the errors after a series of calcula-

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA177

tions. Further, it is can be that individual points in a tensor contain invalid data. It then makes no sense to do calculations with them. Tensorics provides dedicated mechanisms for this cases. The `QuantifiedValue`s contain two additional fields: a (boolean) validity flag and an optional value for an error (uncertainty). All the operations on quantified values (and on tensors of quantified values) take this fields into account. The exact behavior can again be configured by the use of explicit strategies. The defaults are:

- If an invalid value is used in a calculation, then the resulting value will be invalid.
- The values involved in the calculations will be treated as independent variables and the error is propagated to the resulting value accordingly [7].

Comparisons between quantities take into account their associated errors assuming Gaussian statistics. The confidence level is 95% unless specified otherwise. This allows to conveniently check if a quantity is significantly less, equal, or greater than another. For example, 90 ± 1m is significantly less than 100 ± 10m at a confidence level of 68% but not at 95%.

Listing 18: Comparison of Quantities.

```
QuantifiedValue<Double> q90pm1 =
  quantityOf(90.0, METER).withError(1.0);
QuantifiedValue<Double> q100pm10 =
  quantityOf(100.0, METER).withError(10.0);

/* false at 95% confidence (default): */
testIf(q90pm1).isLessThan(q100pm10);

/* true at 68% confidence: */
with(confidenceLevelOf(0.68))
    .testIf(q90pm1).isLessThan(q100pm10);
```

TENSORBACKED DOMAIN OBJECTS

While working with tensors gives all the flexibility of transformations and calculations, very often it is desirable to give more meaning to objects. Usually one would create dedicated domain objects in these cases. However, this would mean giving up all the convenient support methods. To combine the best of both approaches, tensorics provides a built-in mechanism for creating domain objects which wrap tensors inside and allow almost the same calculations and transformations as plain tensors. These objects are called `Tensorbacked`s and can be defined by the user as required. The simplest way to do so is to inherit from `AbstractTensorbacked`. An important property of tensorbacked objects is that each of them has a fixed set of dimensions, which are defined through the dedicated annotation `@Dimensions`. For example, if one would like to define some domain object that contains temperatures, one could do so by

Listing 19: Tensorbacked definition.

```
@Dimensions({Time.class, City.class})
public class TemperatureMap
    extends AbstractTensorbacked<Double> {
    /* empty (except a constructor) */
}
```

Instances of these classes can then be created using simply an existing tensor or a builder. Calculations can be performed like with bare tensors.

Listing 20: Tensorbacked Usage.

```
TemperatureMap measured = Tensorics
     .construct(TemperatureMap.class)
     .from(degrees);

TemperatureMap reference = Tensorics
    .builderFor(TemperatureMap.class)
    .put(at(SF, T1), 10.0)
    .build();

TemperatureMap diff = DoubleTensorics
    .calculate(measured).minus(reference);
```

When using a builder, the dimensions do not have to be given explicitly, as they are already defined through the annotation.

EXPRESSION LANGUAGE

All the examples in the previous sections described directly Java executable code. In addition to this, tensorics provides a Java internal domain specific language (DSL) to only describe calculation steps using the same operations as described before. This DSL does not directly execute the calculations, but instead creates an expression tree, which can be evaluated (resolved) in a separate step. Since these expressions can be resolved in different contexts, this can e.g. be used for subscription based online evaluation (e.g. processing data from devices) or processing logged data. This expression language is one of the cornerstones of a recently developed online analysis framework. More details can be found in the corresponding publication [8].

APPLICATIONS

The tensorics library became essential for several applications used to control CERN accelerators. For example, the LHC Luminosity Server [9] heavily uses tensorics for all its internal data storage and processing, in particular all the capabilities of tensors and quantified values. The same is true for the applications that control chromaticity [10] and coupling [11] of the LHC. The third intensive use of the package was described within [12] while treating on computations regarding reproducibility and stability of the beam orbit in the LHC.

The expression language was driven big steps further by the need of the recently developed injection diagnostics application for the LHC [13]. Together with the streaming pool library [14], it was generalized into a powerful analysis framework.

SUMMARY AND OUTLOOK

The tensorics library evolved during the past few years from an experimental prototype to a general purpose library

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA177

which is used in production within several applications at CERN.

The essential features, together with code examples have been introduced in the sections of this papers: The usage of Tensors, quantities and tensorbacked objects. For a more detailed explanations of the expressions DSL we refer to a dedicated paper [8].

The tensorics library is available as open source under Apache License 2.0. The paper describes the latest version at the time of writing which is considered as an alpha stage. While the basic concepts and core functionalities are well established, the API will still have to undergo some cleanup and symmetrization, before a stable version (1.0) will be released.

REFERENCES

[1] tensorics, https://github.com/tensorics

[2] Field (mathematics), Wikipedia, https://en.wikipedia.org/wiki/Field_(mathematics)

[3] Apache Commons, http://commons.apache.org/proper/commons-math

[4] numpy, https://github.com/numpy/numpy

[5] tensorics-core API, https://tensorics.github.io/projects/tensorics-core/javadoc/

[6] JScience, http://jscience.org

[7] Propagation of uncertainty, Wikipedia, https://en.wikipedia.org/wiki/Propagation_of_uncertainty

[8] K. Fuchsberger *et al.*, "A Framework for Online Analysis Based on Tensorics Expressions and Streaming Pool", presented at ICALEPCS'17, Barcelona, Spain, 2017, paper THPHA178, this conference.

[9] M. Hostettler *et al.*, "Online Luminosity Control and Steering at the LHC", presented at ICALEPCS'17, Barcelona, Spain, 2017, paper TUSH201, this conference.

[10] K. Fuchsberger, G. H. Hemelsoet, "LHC Online Chromaticity Measurement - Experience After One Year of Operation", in *Proc. IBIC'2016*, pp. 20-23, Barcelona, Spain, 2016, paper MOBL04.

[11] G.H. Hemelsoet *et al.*, "Online coupling measurement and correction throughout the LHC Cycle", presented at ICALEPCS'17, Barcelona, Spain, 2017, paper TUPHA119, this conference.

[12] A. Gorzawski, "Luminosity control and beam orbit stability with beta star leveling at LHC and HL-LHC", PhD thesis, Ecole Polytechnique Lausanne, Switzerland, 2016.

[13] K. Fuchsberger *et al.*, "Development of a new system for detailed LHC filling diagnostics and statistics", in *Proc. IPAC'17*, pp. 1905-1907, Copenhagen, Denmark, 2017, paper TUPIK088.

[14] A. Calia *et al.*, "Streaming Pool - Managing Long-Living Reactive Streams for Java", presented at ICALEPCS'17, Barcelona, Spain, 2017, paper THPHA176, this conference.