TENSOR TOOLBOX: WRAPPING TO PYTHON USING SWIG

M.G. PETERSON * AND D.M. DUNLAVY †

Abstract. Tensor Toolbox was developed as a Matlab package to allow users to interact with tensors (ie. multidimensional or N-way arrays). A subset of this Matlab version was implemented in C++ with the focus on increased speed for computing. This paper discusses how a Python version was implemented using SWIG (a C++ wrapping tool) on the C++ subset of Tensor Toolbox. All three of the versions were compared against each other based on their computing time for the CPAPR (CANDECOMP/PARAFAC Alternating Poisson Regression) algorithm. The C++ version was by far the fastest, the Python version was the second, and the Matlab version third.

1. Introduction. Tensors are multidimensional arrays (i.e., N-way arrays) and are often used the fields of machine learning, chemometrics, signal processing, graph analysis, and more (see [5] for a survey for more details). Several different implementations of tensor data structures, models, and model fitting algorithms have been developed for data analysis in these fields: e.g., Matlab Tensor Toolbox (TTB) [2], N-Way Toolbox [1], among others. To date, much of the algorithmic work for tensors applied to data analysis has been implemented in Matlab. We present here an implementation of the operations and algorithms for tensor analysis in Python, following the main class structure of TTB and leveraging a C++ version of the data structures and algorithms available in TTB. We discuss some of the challenges in developing a Python version of a limited subset of functionality of TTB, which we call PyTTB. Further, we demonstrate some of the benefits of using PyTTB over TTB in terms of computational speed.

The CANDECOMP/PARAFAC (CP) tensor model can be used to approximate the relationships amongst data represented in a tensor. The CP model is often referred to as a high-order analog of principal components analysis (PCA) without orthogonality constraints on the factors. Like PCA, the CP model is a sum of vector outer products and can be used to identify a reduced-dimension approximation of data relationships. Algorithms for fitting CP models to data include CPALS (CP Alternating Least Squares) for continuous data and CPAPR (CP Alternating Poisson Regression) for count data. These two models have been implemented in the C++ version of TTB and available in PyTTB. CPAPR will be the method used for comparisons of the different implementations of TTB presented in Section 4.

The Matlab Tensor Toolbox is a software package used for constructing tensor objects and models, computing tensor factorizations, and performing data analysis of multiway data using those structures and models. One of the main strengths of TTB is its efficient computations and algorithms for large, sparse data. TTB was originally written in Matlab, and later a subset of the functions were implemented in C++ with the goal of improving speeds for larger tensors. Challenges is using these two implementations include the cost of a Matlab license for TTB and the lack of an interactive environment for the more efficient C++ implementation. PyTTB is an effort to address these two challenges.

PyTTB provides both an interactive environment for "on-the-fly" object modifications via the Python interactive shell and computational efficiency via binding to the C++ implementation mentioned above. Python is free to use and is a very common language used for interactive purposes. Also, including C++ code in Python

*University of New Mexico for Computer Science, mpeterson@unm.edu
†Sandia National Laboratories, dmdunla@sandia.gov

via extensions can be accomplished without requiring changes to the original C++ code using a software tool called SWIG.

2. Extending C++ Code Using SWIG. SWIG (Simplified Wrapper and Interface Generator)[1] is a software tool that wraps C++ classes and functions into other languages such as Perl, Java, Python, TCL, and more [3, 4]. Python wrapping using SWIG is the main focus in the sections below.

The SWIG files created by the developer are referred to as interface files. These files are where the developer can make use of the wrapping features that SWIG offers. The most important features available in SWIG and used in PyTTB are *extension blocks*, *python code blocks*, and *typemaps*.

2.1. Extension Blocks. Extension blocks allow the developer to add methods to a class without having to modify the original C++ code. This is useful when you have a method that only applies to the specific language you are trying to wrap to and not in the general use cases of the C++ code, such as mapping a custom object's `size()` method to `__len__` in Python.

Extensions are written in C++ and their scope are functions/methods defined in the original C++ code. This means that methods defined SWIG interface files are outside the scope of methods defined in extensions; even if two given methods are inside the same extension block. Despite the scoping constraint extensions prove useful, especially with adding target language built-in functions, such as `__str__` and `__len__` in Python.

2.2. Python Code Block. SWIG interface files are written in C++, but with the
`pythoncode` tag the user can write Python code which will be directly inserted into the generated `.py` package. This can be used for adding additional functions to the package or can be written within an extension block to add more methods to a class.

Unlike extension blocks, the scope of pythoncode blocks is the entire Python package. This allows the developer to use methods and functions previously defined elsewhere in the SWIG interface files, including those in extension blocks.

2.3. Typemaps. Typemaps are arguably the most convenient feature of SWIG. They are used for converting an object of one type into that of another type, either on input or output. This provides a simple, global way to convert an object from the target language into a C++ object, or vice versa.

For every typemap(in) (ie. a typemap from target language to C++), there needs to be a special typemap defined called a typecheck; this function returns a boolean value. A typecheck is used to determine whether an object can use a given typemap; it returns true if the input object can be converted or false if it cannot be converted.

An example of the impact of typemaps in SWIG on Python wrapping of C++ code is presented in Figure 2.1, where a package is imported and an array object is printed. A typecheck for the typemap used in `MyPackage.printArray()` would check if the input object is a sequence, only contained three items, and that the items contained inside the sequence were all integers. The typecheck would return false if any of those attributes were not true for the input object, otherwise it would run the conversion typemap.

There is a precedence value associated with each typecheck. This value is responsible for creating the order in which a typecheck gets called. The developer can

[1] `http://www.swig.org/`

EXAMPLE_1: (without a typemap)

```
>>> import MyPackage
>>> v = MyPackage.ThreeArray(1,2,3)   # Creates a ThreeArray.
>>> MyPackage.printArray(v)
Printing Array: [1,2,3]
```

EXAMPLE_2: (using a typemap)

```
>>> import MyPackage
>>> MyPackage.printArray([1,2,3])     # Using a Python list.
Printing Array: [1,2,3]
```

Fig. 2.1: MyPackage is a package that was created using SWIG; it contains a class object called ThreeArray which is a container object that holds three integers. MyPackage also has a function called printArray that takes in a ThreeArray as a parameter. EXAMPLE_1 shows how printArray would be used without a typemap. EXAMPLE_2 shows how it could work if a typemap from a Python list to ThreeArray was previously defined.

modify the precedence value of the typecheck that he/she has constructed.

3. Python Tensor Toolbox. PyTTB is the Python version of the C++ implementation of Tensor Toolbox and is intended to provide the same capabilities as the Matlab version, TTB. SWIG was used to map all the C++ functions and classes into Python (see Section 2 above). This section provides specific examples of how PyTTB utilizes features provided by SWIG.

3.1. Class Wrapping. Within the PyTTB package there are several C++ classes that were wrapped; `Array`, `IndxArray`, `FacMatrix`, `FacMatArray`, `Tensor`, `Sptensor`, and
`Ktensor`. Some of the classes are used primarily as an underlying data structure for the tensor classes. Each class has their own SWIG interface file containing the mapping rules from C++ for the given class.

3.1.1. Operator and Method Extensions. All of the classes implemented in PyTTB include an extension block. Within these extension blocks are arithmetic operators and special object methods for that particular class; such as `__add__`, `__sub__`, `__mul__`, `__div__`, `__pow__`, `__eq__`, `__setitem__`, `__getitem__`, `__iter__`, `__str__`, and `__len__`.

Below is an example of calling the special object methods `__add__` (using '+') and `__str__` (using `print`) for `Array`.

```
>>> import tensor_toolbox as ttb
>>> a = ttb.Array([2,1,3])
>>> b = ttb.Array([1,1,1])
>>> print a + b              #Elementwise addition
[3 2 4]
```

3.1.2. Error Handling. Within each class there is an exception block which contains a `try/catch` block. This is used for converting C++ errors into Python errors. One thing to note is that if an error is not explicitly thrown within a C++ method it will not be caught by the exception block.

This what an exception would look like:

```
>>> a = ttb.Array([2,3,1])
>>> b = ttb.Array([2,2])
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/.../python/tensor_toolbox.py", line 332,
  in __add__
    def __add__(self, *args):
    return _tensor_toolbox.Array___add__(self, *args)
StandardError: Sizes of Arrays do not match
```

3.1.3. Merge Types. These were functions added to the PyTTB package that are not included in the C++ version. This will merge two objects of the same class into one. For example, with an Array you could think of a merge as a concatenation function. With an Array like a FacMatrix we can merge as rows or columns; in other words it adds the rows from one FacMatrix to another, or adds the columns. An example can be seen in the listing below.

```
>>> import tensor_toolbox as ttb
>>> f1 = ttb.FacMatrix(2,2)
>>> f1.setValue(1)            # 2x2 matrix with all 1's
>>> f2 = ttb.FacMatrix(1,2)
>>> f2.setValue(2)            # 1x2 matrix with all 2's
>>> ttb.mergeAsRows(f1,f2)
_____________________________________
matrix
_____________________________________
Size = [ 3 2 ]
X(0,0) = 1
X(1,0) = 1
X(2,0) = 2
X(0,1) = 1
X(1,1) = 1
X(2,1) = 2
_____________________________________
```

3.1.4. CPALS and CPAPR. Although the CPALS and CPAPR algorithms for fitting CP models are provided in the C++ implementation, they are also implemented in PyTTB as well. In Python, they are implemented as an illustration of the use of several C++ classes and methods in a complete algorithm. These implementations are the ones used in experiments discussed in Section 4.

3.2. Style Choices. Several style choices were made during the process of developing the PyTTB package. There are other ways to achieve similar results, but these seem to be better in the long run, motivated by the goal to make it simple for developers to add new classes to PyTTB in the future.

3.2.1. EQUALITY TOLERANCE. For a few classes there is a method, `isEqual(object,tolerance)`, defined in the original C++ package. This method compares two objects of the same class and determines whether they are equal to each other within some tolerance level. This method seems like a logical choice to map to the `__eq__` operator in Python where the tolerance level would be zero, but there is a small problem. Even if the two objects are exactly the same, the method will return false because the tolerance level is a strict inequality; i.e. since the the comparision result is always greater than or equal to zero, it cannot be strictly less

than zero. So a global variable is defined, `EQUALITY_TOLERANCE`, that is set to `1.0e-8` by default and is used instead of zero for the `__eq__` methods to compensate for the strict inequality issue.

3.2.2. Python Code Blocks. The scope of extension blocks are very limited, but the scope of Python code blocks includes the entire package. The underlying data structures in the tensor classes are either an `Array` or a `FacMatrix`. So when defining a mathematical operator for the tensor class, such as `__add__`, a Python code block was used. This gave the develeoper the ability to use the underlying data structure's `__add__` method when defining the tensor's method for `__add__`.

This style was chosen to reduce the amount of error handling that had to be implemented as well as to provide a simple way to modify the code for multiple classes at once.

3.2.3. Typemaps. The goal of typemaps were to make the C++ code feel more Pythonic. Typemaps, such as Python list to `Array` and `IndxArray`, were implemented to allow users to easily change parameters on the fly while in the Python interface.

3.3. IPython Notebooks. IPython Notebooks[2] are a useful way to design a tutorial for Python code [6]. Notebooks are set up into cells which are classified as Markdown, Python Code, and various Header Fonts. This allows the designer to use a mixture of Python code and HTML when designing a tutorial.

The Python tutorials were designed to look exactly like the TTB tutorials provided in Matlab. The goal of this was to make an easy transition for Matlab users by making the environment as similar as possible.

When the design of the notebook is finished, there are options to convert the notebook to HTML or Python. The HTML version can be open/viewed from a browser and is useful for displaying the tutorial. The Python version will comment out everything except for Python Code cells, which is useful when a user wants to interact with the code directly.

4. Experiments. In this section, we present the results of comparing Matlab, C++, and Python implementations of the CPAPR (CP Alternating Poisson Regression) algoritm . Specifically, we focus on the speed of the implementations (from the user point of view), as we have verified that the 3 implementations generate identical iterations and solutions. The data used for testing were random sparse tensors containing approximately 0.1% nonzeros. The rank of the tensors, 10, was the same for all the tests. The dimensions of the tensors were all of size three, where the first two dimensions remained constant but the third varied. The complete set test characteristics can be found in Fig 4.1.

The second data set varied in all three dimensions. For each of the tests the dimensions were scaled by a factor of 2. More information about the data set is described in Fig 4.2.

Timing information were collected using the `time` Linux system call. The time recorded included the time to run the CPAPR algorithm as well as the time it took to read-in the sparse tensor data and the k-tensor initial guess. Comparisons are made using a paired sample t-test of timing information from the results of running CPAPR on ten different initial guesses for each test size.

The experiments were performed on system with 8 Intel i7-3940XM CPUs (3.00 GHz) with 32GB RAM running Ubuntu 12.04 (64-bit).

[2] `http://ipython.org/notebook.html`

Test Number	Dimensions	Rank	Nonzeroes	Coefficients
1	[20,30,15]	10	836	650
2	[20,30,50]	10	2,836	1,000
3	[20,30,100]	10	5,684	1,500
4	[20,30,500]	10	28,371	5,500
5	[20,30,1000]	10	56,569	10,500
6	[20,30,5000]	10	282,885	50,500

Fig. 4.1: A brief description of the attributes for each the test data files used to check performance on each of the languages. In this first data set only the third dimension was changed.

Test Number	Dimensions	Rank	Nonzeroes	Coefficients
1	[20,30,15]	10	836	650
2	[40,60,30]	10	6,808	1,300
3	[80,120,60]	10	54,476	2,600
4	[160,240,120]	10	434,638	5,200

Fig. 4.2: A brief description of the attributes for each the test data files used to check performance on each of the languages. In this second data set all dimensions were scaled by a factor of 2.

4.1. Python vs Matlab. Figs. 4.3–4.6 present the results of running CPAPR for the test cases described above. These results demonstrate that the Python implementation is much faster than the Matlab version. Also, this conclusion is confirmed by the results of the paired sample t-test results presented in Fig. 4.7 and Fig. 4.8. When Python and Matlab are compared it can be seen that the t-statistic is consistently negative and decreasing. This implies that Matlab computation time is increasing faster than that of the Python implementation as the problem size increases.

Test	1	2	3	4	5	6
C++	0.005	0.005	0.005	0.005	0.004	0.015
Python	3.042	15.098	32.823	179.425	354.016	969.710
Matlab	16.162	56.926	92.586	314.409	577.373	2,750.293

Fig. 4.3: Average timing results (in seconds) of 10 runs of CPAPR on each of the test cases described in Fig. 4.1.

4.2. Python vs C++. The C++ implementation was clearly the fastest of the three versions. In Fig. 4.7 one can see that the t-statistic for Python vs C++ is positive and increasing, which implies that the Python version's computation time is increasing at a faster rate than the C++ version. Another item to note is that the t-statistics for Python vs C++ is much larger than that of Matlab vs C++. This is an indication that the Python version's computation time is growing faster than the Matlab's version in respect to the C++ version.

Test	1	2	3	4
C++	0.004	0.004	0.005	0.005
Python	3.043	41.536	204.732	1,235.935
Matlab	16.318	132.133	609.617	4,729.493

Fig. 4.4: Average timing results (in seconds) of 10 runs of CPAPR on each of the test cases described in Fig. 4.2.

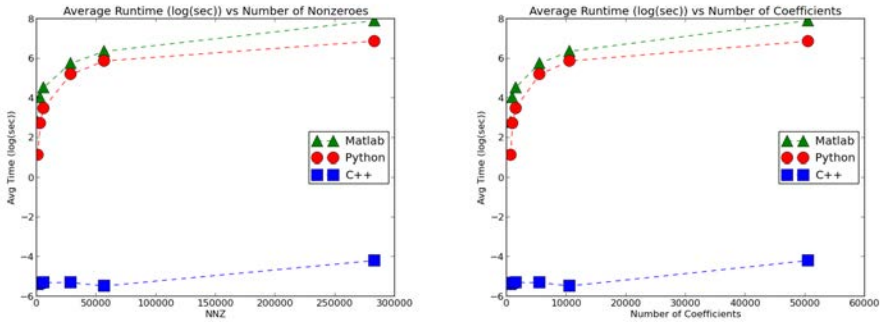

Fig. 4.5: Average timing results (in seconds on a log scale) of 10 runs of CPAPR on each of the test cases described in Fig. 4.1. The x-axes include the number of nozeros, NNZ (left) and number of coefficients in the CPAPR models (right).

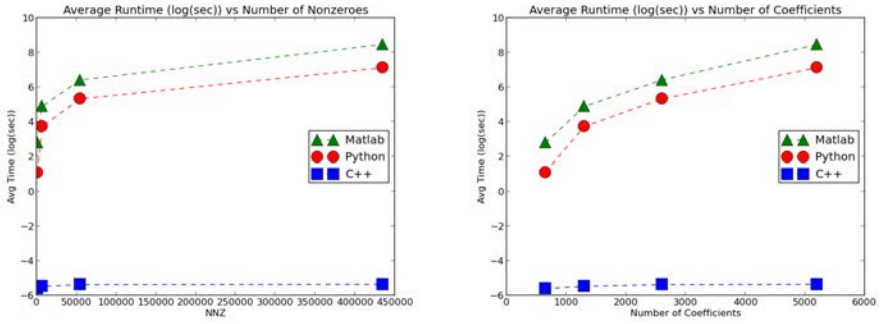

Fig. 4.6: Average timing results (in seconds on a log scale) of 10 runs of CPAPR on each of the test cases described in Fig. 4.2. The x-axes include the number of nozeros, NNZ (left) and number of coefficients in the CPAPR models (right).

	Python vs Matlab		Python vs C++		Matlab vs C++	
Test	t-stat	p-value	t-stat	p-value	t-stat	p-value
1	-4.93	1.08e-04	5.56	2.81e-05	6.20	7.44e-06
2	-5.35	4.41e-05	7.46	6.57e-07	7.53	5.71e-07
3	-7.12	1.24e-06	11.68	7.77e-10	11.70	7.55e-10
4	-9.25	2.92e-08	47.48	2.29e-20	22.31	1.45e-14
5	-12.44	2.81e-10	80.04	1.98e-24	36.04	3.11e-18
6	-28.63	1.83e-16	206.78	7.72e-32	44.70	6.71e-20

Fig. 4.7: Above is a short summary of statsistcal values of the experiments described in Fig. 4.1. A negative t-value indicates the first faster than the second, and a positive t-value implies the first is slower than the second. The p-value is the probability that one would be correct if they reject the hypothesis in regards to the t-value

	Python vs Matlab		Python vs C++		Matlab vs C++	
Test	t-stat	p-value	t-stat	p-value	t-stat	p-value
1	-5.12	7.18e-05	5.78	1.74e-05	6.42	4.73e-06
2	-17.45	9.92e-13	27.82	3.02e-16	26.58	6.74e-16
3	-166.05	3.97e-30	239.58	5.42e-33	266.94	7.74e-34
4	-296.58	1.16e-34	716.74	1.47e-41	405.88	4.11e-37

Fig. 4.8: Above is a short summary of statsistcal values of the experiments described in Fig. 4.2. For information about the meaning of the values please refer to Fig. 4.7.

5. Conclusion. The Python version was not as fast as the C++ version, but it was faster than the Matlab version. The goal of PyTTB was to offer an alternative to the Matlab version, while leveraging the efficiency of the C++ implementation when possible. PyTTB currently offers a subset of the Matlab TTB functionality, and this subset can produce the same results in less time.

The style choices chosen in PyTTB help consolidate shared functionality into reusable code. The use of `pythoncode` tags decreased the amount of error checking needed. It provided a coordinated approach to error checking of previously defined functions containing a diverse set of error checking measures. But the choice that impacted the code the most was typemaps.

Typemaps allowed an easy way to use make PyTTB feel more "Pythonic." The methods and functions defined in the C++ version were able to take in a larger variety of arguments by defining a hand full of mappings. Without the typemaps, a majority of the methods would have required overloaded implementations to handle Python objects (e.g., `lists`).

5.1. Future Work. As mentioned in Section 3.1.2, there is room for improvement in regards to error handling. There may be a way to use typemaps to handle errors, avoiding explicitly throwing errors. Another option would be to include error `try/catch` blocks in every class extension block; the typemaps would convert C++ errors into python errors, even if the C++ errors were not explicitly thrown.

Lastly, there is the the fact that C++ version is much faster than the Python version. There are some areas in the Python where optimization may be available, primarily where Python handles mathematical operations. In a few of the mathematical methods, a copy of the object is used instead of using the original. Generating a copy makes sense for generic cases, such as `z = a + b`. But if `a += b` is done instead, a copy does not need to be generated and the `a` object can be modified directly. Another option would be to run CProfile to help identify areas in the code that could lead to timing improvements.

REFERENCES

[1] C. A. Andersson and R. Bro, *The N-way toolbox for MATLAB*, Chemometrics & Intelligent Laboratory Systems, 52 (2000), pp. 1–4.
[2] B. W. Bader, T. G. Kolda, et al., *Matlab tensor toolbox version 2.5.* Available online, January 2012.
[3] D. M. Beazley, *Automated scientific software scripting with SWIG*, Future Gener. Comput. Syst., 19 (2003), pp. 599–609.
[4] T. Cottom, *Using SWIG to bind C++ to python*, Computing in Science Engineering, 5 (2003), pp. 88–97.
[5] T. Kolda and B. Bader, *Tensor decompositions and applications*, SIAM Review, 51 (2009), pp. 455–500.
[6] F. Pérez and B. E. Granger, *IPython: a system for interactive scientific computing*, Computing in Science and Engineering, 9 (2007), pp. 21–29.