How to use PyXLTensor

1 Initialization of the tensors

Tensors can be initialized in two different ways, by directly specifying all elements or by defining the tensor shape. The other parameters necessary to the initialization are the tensor name, its indices and the metrics related to the various indices, if they are different from the identity.
tensor_name (string): Name of the tensor
indices_string (string): String containing all the indices in order, if an index name is preceded by ^ is an upper index, if it is preceded by _ is a lower index.
tensor=None (list, tuple or numpy.array): If given, specifies the values of the tensor.
shape=None (list, tuple or numpy.array): If no tensor is given, specifies the dimensions of the tensor.
metrics=None (list, tuple or numpy.array): Lists the metric tensors associated with each index. If unspecified, the identity metric is assumed for each index.

```
import PyXLTensor as xlt

"""
the tensor t1 has two indices, alpha (up) and beta (down), the value of
     the tensor is known and all the metrics are the identity.
the tensor t2 has two indices, beta (up) and gamma (down), the value of
     the tensor is unknown and the metric associated with gamma is not
    the identity.
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^beta_gamma', shape=(2, 2), metrics=[[[1, 0],
    [0, 1]], [[-1, 0], [0, 1]]])
```

The tensor name is important to keep track of the unknown tensors. It's fundamental that unknown tensors do not share the same name.

2 Basic operations

The PyXLTensor library allows for tensor arithmetic using the standard symbols. For example additions and subtractions can be performed directly with + and -. These operations require that the indices of the tensors are consistent with each other. They can be in different orders as long as there are indices that share the same name, size, metric and are both up or both down.

```
import PyXLTensor as xlt

"""
Correct:
Even if the indices are not in the same order they have the same name,
    size, positioning an metrics.
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '_beta^alpha', shape=(2, 2))
t3 = -t2 + t1
```

```
import PyXLTensor as xlt

"""
Wrong, respectively:
(t1) Mismatch of the name of the indices.
(t2) Mismatch of the position of the indices.
(t3) Mismatch of the dimention of the indices.
(t4) Mismatch of the metrics of the indices.
"""
t = xlt.Tensor('t', '^alpha_beta', tensor=[[1, 2], [0, -1]])

t1 = xlt.Tensor('t2', '^gamma_beta', shape=(2, 2))
t_sum = t + t1

t2 = xlt.Tensor('t2', '^alpha^beta', shape=(2, 2))
t_sum = t + t2

t3 = xlt.Tensor('t2', '^alpha_beta', shape=(2, 3))
t_sum = t + t3

t4 = xlt.Tensor('t2', '^alpha_beta', shape=(2, 2), metrics=[[[1, 0],
    [0, 1]], [[-1, 0], [0, 1]]])
t_sum = t + t4
```

PyXLTensor also supports multiplication and division by a scalar using * and /, respectively, and tensor contractions via @. When contracting tensors one has to make sure that the contracted indices are one up and the other down and share the same properties.

```
import PyXLTensor as xlt

t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^beta_gamma', shape=(2, 2), metrics=[[[1, 0],
    [0, 1]], [[-1, 0], [0, 1]]])
t3 = 2 * t1 @ t2
```

The library includes methods for summing and contracting lists of tensors, respectively `Tensor.sum_all` and `Tensor.contract_all`. All tensor rules of these operations must be respected like if those operations were performed one by one. If a string is provided after the list of tensors, the string will be the tensor name of the new tensor.

```python
import PyXLTensor as xlt

"""
Sum of tensors
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^alpha_beta', tensor=[[3, -1], [1, 1]])
t3 = xlt.Tensor('t3', '^alpha_beta', tensor=[[2, 1], [-4, 3]])
t4 = xlt.Tensor.sum_all([t1, t2, t3], 't4')

"""
Contraction of tensors
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^beta_gamma^i', tensor=[[[3, -1], [1, 1]], [[0,
    -2], [3, 2]]])
t3 = xlt.Tensor('t3', '^gamma_alpha', tensor=[[2, 1], [-4, 3]])
t4 = xlt.Tensor.contract_all([t1, t2, t3], 't4')
```

3 Block tensors

PyXLTensor supports the combination of tensor lists into a single block tensor by specifying the indices to be grouped. In order to specify which indices will be combined and how, we will give a list, `grouping_of_indices`. The elements of this list will be another list, with two elements, the first one is the name for the new index in the block tensor, the other is the list of the indices name that will be grouped together. All the other indices will be untouched and must coincide. Consider the following example:

$$T^{I}{}_{J}{}^{K}{}_{L} = \begin{pmatrix} a^{i_1}{}_{j_1}{}^{K}{}_{L} & b^{i_1}{}_{j_2}{}^{K}{}_{L} & c^{i_1}{}_{j_3}{}^{K}{}_{L} \\ d^{i_2}{}_{j_1}{}^{K}{}_{L} & e^{i_2}{}_{j_2}{}^{K}{}_{L} & f^{i_2}{}_{j_3}{}^{K}{}_{L} \end{pmatrix} \tag{1}$$

with $I = (i_1, i_2)$ and $J = (j_1, j_2, j_3)$. As always indices with the same names must have the same proprieties (size, up or down, metric) but the order of them does not matter. We can implement this tensor in the following way

```python
import PyXLTensor as xlt

"""
Block tensor
This is an 8x8x8x8 tensor, even if the indices are not in the same
    order the sizes are consistent among the tensors
"""
a = xlt.Tensor('a', '^i1^K_j1_L', shape=(2, 8, 1, 8))
b = xlt.Tensor('b', '^i1_L^K_j2', shape=(2, 8, 8, 3))
c = xlt.Tensor('c', '^K_L_j3^i1', shape=(8, 8, 4, 2))
d = xlt.Tensor('d', '^i2_j1^K_L', shape=(6, 1, 8, 8))
e = xlt.Tensor('e', '^K^i2_L_j2', shape=(8, 6, 8, 3))
f = xlt.Tensor('f', '^K^i2_L_j3', shape=(8, 6, 8, 4))
grouping_of_indices = [['I', ['i1', 'i2']],
                       ['J', ['j1', 'j2', 'j3']]]
T = xlt.Tensor.block_tensor([a, b, c, d, e, f], grouping_of_indices)
```

The order of the tensor in the list is not important and he name of the grouped index can be arbitrary and it doesn't have to be of the type $[name1,\ name2, \ldots,\ name\#n]$.

4 Symmetrization, anti-symmetrization, duality, δ and ϵ

Given a tensor it is possible to perform the (anti-)symmetrization of some indices. Those indices must be all upper indices or all lower indices, have the same size and the same metric. The normalization is such that the (anti-)symmetrization of (anti-)symmetric indices will result in the same tensor.
If a tensor is entered manually the code will not check if there are (anti-)symmetric indices. In general imposing the symmetrization on those tensors can help the code run faster when dealing with big tensors.

```python
import PyXLTensor as xlt

"""
(Anti-)Symmetrization
"""
t = xlt.Tensor('t', '^alpha^beta', tensor=[[1, 2], [0, -1]])
t_sym = t.symmetrize('alpha', 'beta')
t_asym = t.anti_symmetrize('alpha', 'beta')
```

Another operation that can be done is the duality operation. Given a set of indices the duality operation is equivalent to the contraction with a Levi-Civita tensor with the given indices and the division by the factorial of the number of contracted indices. The indices of the epsilon tensor are all lower (upper) if the contracted indices are all up (down). The convention used for the sign of the Levi-Civita tensor is $\epsilon^{1,\ldots,n} = \epsilon_{1,\ldots,n} = +1$.

```python
import PyXLTensor as xlt

"""
Duality
"""
t = xlt.Tensor('t', '^c', tensor=[1, -1, 0])
star_t = t.dual('a', 'b', 'c')
```

The Kronecker delta δ and Levi-Civita ϵ tensors are in-build functions and can be defined as expected, the delta takes as input the indices, given with the same form for the initialization of a normal tensor and the dimension of the two indices, while the epsilon only needs the indices.

```
import PyXLTensor as xlt

"""
Delta and epsilon tensors
"""
d8 = xlt.Tensor.delta('^i_j', 8)
epsilon = xlt.Tensor.epsilon('_a_b^c')
```

5 Managing the indices

Indices can be raised (lowered) using the method `.to_raise` (`.to_lower`) giving the indices to be raised (lowered). The operation of raising and lowering the indices is done through the metric. The metric is the tensor used to lower the indices, the inverse metric is calculated to raise the indices.
This method will return another tensor with the new indices, while the original tensor is left untouched.

```
import PyXLTensor as xlt

"""
Raising and lowering indices
"""
t = xlt.Tensor('t', '^i_j_k^l', shape=(2, 2, 2, 2), metrics=[[[-1, 0],
    [0, 1]], ] * 4)
t_all_up = t.to_raise('j', 'k')
t_all_down = t.to_lower('i', 'l')
```

There are two different ways to change the name of some indices.
The first one makes use of the method `Tensor.change_indices_name`, this method will have two lists as inputs, the first list will have the old names of the indices and the second one will have the corresponding new names.
The second one needs you to specify all the (old and) new names of the indices in order. It can be done by simply putting square brackets after the tensor and specifying all the new names in the correct order.
The internal order of the indices of the tensor can be seen by calling the attribute `.indices`. This ordering can be changed by simply using the method `Tensor.set_order_indices` and giving the new desired order. Notice that while all the other methods always give you a new tensor, independent from the original one, this method simply modifies the attributes of the object without returning anything.

```
import PyXLTensor as xlt

"""
Changing the indexing
"""
t = xlt.Tensor('t', '^a^j^c^l', shape=(2, 2, 2, 2))
print(t.indices)  # ['a', 'j', 'c', 'l']
t1 = t.change_indices_name(['j', 'l'], ['b', 'd'])
print(t1.indices)  # ['a', 'b', 'c', 'd']
```

```
t2 = t['a', 'b', 'c', 'd']
print(t2.indices)  # ['a', 'b', 'c', 'd']
t2.set_order_indices('d', 'c', 'b', 'a')
print(t2.indices)  # ['d', 'c', 'b', 'a']
```

$t1$ and $t2$ are the same tensor, even if we change the order of the indices of $t2$, so while after those operation $t1 + t2$ is still a valid operation $t + t1$ is not.

When relabeling the indices, if an upper index and a lower index happens to have the same name, a trace will be automatically performed.

6 Elements of the tensors

In order to keep track of the unknown variables in the tensors the class `Poly_Expression` is used, the elements of the tensor are instances of this class.
`Poly_Expression` objects support standard operations with the symbols +, -, * and / (notice however that the division can be used only to divide by a scalar). The expression of a `Poly_Expression` object is determined by its only attribute, `.sum_variables`.
`Poly_Expression.sum_variables` is a list that contains all the summed monomials of an expression, a empty list is associated with the value 0. The monomials are a list of two elements, a overall constant that multiplies the unknowns and the list of the product of the single variables. A variable is itself a list of two objects, the name of the tensor from which it is taken and a tuple with the indices values associated with that variable.
The `Poly_Expression` objects are handled entirely by the `Tensor` class so it is not necessary to create or modify instances of this class directly.

We can clarify better the structure of `Poly_Expression.sum_variables` with the following example:
If we define the two by two tensor x and we took the trace, the resulting element (of the 0-dimensional `Tensor`) is $x_{00} + x_{11}$, written as:

$$\Big[[1, [['x', (0, 0)]]], [1, [['x', (1, 1)]]]\Big].$$

The square of the previous expression is $x_{00}^2 + 2x_{00}x_{11} + x_{11}^2$, written as:

$$\Big[[1, [['x', (0, 0)], ['x', (0, 0)]]], [2, [['x', (0, 0)], ['x', (1, 1)]]], [1, [['x', (1, 1)], ['x', (1, 1)]]]\Big].$$

7 Reading the tensors

One way to get the elements of the tensor is to use the square brackets like it is done to change the indices name. When using the square bracket every numerical element will be taken as the value of the respective index (the index can range from 0 to the size of the index minus one). If all the entries given are numeric, the respective element of the

tensor will be returned, if the elements are both numeric and strings the corresponding subtensor will be returned.
Tensor can be shown by simply printing the object, the printing follows the same rules of a `numpy.array`.
The tensor can be obtained by calling the attribute `.tensor`, however this is a `numpy.array` of `Poly_Expression` and thus can be impractical to use. In the case of fully determined tensors the `numpy.array` composed by all the numerical entries of the tensor can be obtained by the method `.get_numeric_tensor`.

```
import PyXLTensor as xlt

"""
Getting the elements of the tensor
"""
t = xlt.Tensor('t', '^a^b', tensor=[[1, 2], [0, -1]])
print(t[0, 1])  # 2 (Poly_Expression)
print(t['a', 0])  # t^{a0} = (1, 0)^{a0} (Tensor)
print(t[1, 'c'])  # t^{1c} = (0, -1)^{1c} (Tensor)
print(t.get_numeric_tensor())  # [[1, 2], [0, -1]] (numpy.array)
```

8 Initializing a system of equations

PyXLTensor is intended for writing equations of the type $T = 0$, where T is a tensor. To generate a list of equations, first of all it is necessary a list containing all the unknown tensors. If the complete system of equation is generated by different tensor expression, the list of the unknown tensors must be the same for every set of equations (the order of the tensors mus be the same too). The method that returns this set of equations is `Tensor.get_equations(unknown_tensors)`.

The `Solver` class is used to read and process the systems equations. It can be initialized by simply giving the list of equations and the maximum degree of the polynomial that will be processed.

```
import PyXLTensor as xlt

"""
Write the equations and initialize the System.
"""
v = xlt.Tensor('v', '_a', shape=(3, ))
t = xlt.Tensor('t', '_a', shape=(3, ))
M = xlt.Tensor('M', '^a^b', tensor=[[0, 1, 0], [1, 0, 0], [0, 0, -1]])
u = xlt.Tensor('u', '_b', tensor=[0, 1, 1])
one = xlt.Tensor('', '', tensor=1)

Cond1 = M @ v['b'] + t.to_raise('a')
Cond2 = v @ M @ u
Cond3 = v @ t.to_raise('a') - one
Cond4 = v.to_raise('a') @ v - one

unknown_tensors = [v, t]
list_equations = []
list_equations += Cond1.get_equations(unknown_tensors)
list_equations += Cond2.get_equations(unknown_tensors)
list_equations += Cond3.get_equations(unknown_tensors)
list_equations += Cond4.get_equations(unknown_tensors)

system = xlt.System(list_equations, 4)
```

9 Obtaining and reading the solutions

In order to avoid floating-point errors when dealing with expressions that should be zero, there is the possibility to set the tolerance for the value 0. Each value whose magnitude is less than this threshold will be considered zero.
To obtain the solutions is sufficient to run `System.get_Solutions()`. This method can take as input the maximum number of step tried, this can be used as a safe measure if the systems take to long to be processed. By default, if no argument is given there will me no maximum number of steps.
This method will return two list, the first one will contain all the complete solutions while the second will contain all the partial solutions. The solutions can be turned back in tensor form by using the method `Tensor.get_tensors_from_Sol(unknown_tensors, Sol)`. This method takes as input the unknown tensors, with the same order used when generating the equations, and a solution from one of the two list. The method will return the unknown tensors where all the known entries are substituted with their respective values.

```
# continuation of the previous code

xlt.zero_tolerance = 1e-8  # default value 1e-12

Complete_Solutions, Undetermined_Solutions = system.get_Solutions()
C_Tensor_Solutions = [xlt.Tensor.get_tensors_from_Sol(unknown_tensors,
    Sol) for Sol in Complete_Solutions]
U_Tensor_Solutions = [xlt.Tensor.get_tensors_from_Sol(unknown_tensors,
    Sol) for Sol in Undetermined_Solutions]

print('Solutions to tensor equations')
for i, Tensor_Sol in enumerate(C_Tensor_Solutions + U_Tensor_Solutions)
    :
    print(f'Solution {i + 1}:\n')
    for tensor in Tensor_Sol:
        print(tensor)
```