TENLAB

Xiangrong, Xu
xx2367@columbia.edu

Xincheng, Xie
xx2365@columbia.edu

Songqing, Ye
sy3006@columbia.edu

Senhong, Liu
sl4839@columbia.edu

1. Introduction

TENLab is an imperative language that supports distributed matrix calculation and its basic syntax is a combination of Python, Matlab and C. In TENLab, everything needs to be wrapped into a tensor, and that's the underlying philosophy of our language, which is to say. In addition to primitive data types, TENLab also provides an advanced data type, i.e., void tensor, which supports user defines and implements some complex data structure. TENLab also provides an abstract wrapper for users to define its own distributed model for matrix computation. Ideally, TENLab also provides an underlying distributed model for matrix addition, subtraction and multiplication. Overall, the goal is to create a language that is as convenient as Python, but support a parallel matrix computation for acceleration.

2. Data Types

All the data in TENLab are in "tensor" type. Even a simple number is a zero-dimensional tensor. However, all the tensors have an overall data type which is determined by the data type of the smallest granularity in a tensor. For example, an integer-type tensor means every element in this tensor is integer. Generally, a tensor has the following structure:

```
; structure {type, ndim, dim, data}
{i32, i32, [n x i32], [...]}
```

For any operations, they must operate on tensors with the same data type; otherwise, it will throw an error.

Data types in TENLab can be classified as three primitive data types and one complex data type.

```
Type: int | float | char | var
```

2.1. Primitive Data Types

Primitive data types are fundamental data types in TENLab. Every type has its fixed size in memory and some related properties. For a tensor whose every element's primitive type is the same, then the tensor is also this primitive type; otherwise, if elements in a tensor have different primitive types, then this tensor belongs to the complex data type.

1. `int`

`int` means integer. It stores a 32-bit numeric value in memory.

2. `float`

`float` means floating-point number. It stores a 64-bit numeric value in memory. It follows IEEE Standard for Floating-Point Arithmetic (IEEE 754).

3. `char`

`char` means characters. It stores a 8-bit numeric value in memory. It is encoded by ASCII.

2.2. Complex Data Types

When data types in a tensor are various, then it is a `var` type tensor, which means elements in the tensor can contain all of three primitive data types. It is just like a nested version of different types of tensors. For example,

```
ts = var(['c', 1, 1.3]);
```

3. Lexical Convention

3.1. Comments

Comments serve as an annotation or a programmer-readable explanation in the source code. Any tokens in the comment will not be parsed.
TENLab supports two forms of comment:

1. Inline comment: adds `#` at the beginning of the comment, and the comment will stop at the end of the current line.

```
x = int(0) # this is a comment
```

2. Comment block: adds `'''` at both the beginning of the comment and the end of the comment block to make multiple lines of comments.

```
'''
This is
a multi-line
comment
'''
```

3.2. Identifiers

Identifier in TENLab is a case-sensitive ASCII sequence of one or more letters, digits and underscore `_`. The first character in an identifier must be a letter. Identifier cannot be a keyword.

```
# valid identifiers
```

```
helloTENLab
hello_TENlab
TENLab_ranks_1st_in_the_most_popular_programming_languages

# invalid identifiers
int
1st_TENLab
```

3.3. Operators

3.3.1. Arithmetic Operators

The following list shows basic arithmetic operations TENLab supports. Denote **A** and **B** as the first operand and the second operand in a binary operation. Constants are in the form of a 0-dimension tensor, and they should only appear on the right side of operators. Operators with '.' means element-wise operations.

Attention: A binary arithmetic operator should have operands of the same numeric types, that is to say any binary arithmetic operation between an int-type tensor and an float-type tensor is not allowed.

Addition(+): **A** and **B** should have the same dimension, or **B** is a 0-dim tensor.

Addition examples:

```
A = int([0, 1, 1, 0])
B = int([2, 3, 1, 5])
A - B  # A and B should have the same dimension

return
    int([-2, -2, 0, -5])

A + 1  # the second operand is a 0-dim tensor

return
    int([1, 2, 2, 11])

C = float([0.2, 0.3, 0.4, 0.6])
A + C  # not allowed!

A + 0.5    # not allowed!
```

Subtraction(-): **A** and **B** should have the same dimension, or **B** is a 0-dim tensor.

Subtraction examples:

```
A = int([0, 1, 1, 0])
B = int([2, 3, 1, 5])
A - B  # A and B should have the same dimension

return
    int([-2, -2, 0, -5])

A - 1  # the second operand is a 0-dim tensor

return
    int([-1, 0, 0, -1])
```

Matrix Multiplication(*): **A** should be an m by n tensor and **B** should be n by p tensor, or **B** is a 0-dim tensor.

Matrix multiplication examples:

```
A = int([[1, 3, 5], [2, 4, 7]])
B = int([[-5, 8, 11], [3, 9, 21], [4, 0, 8]])

A * B  # A should be an m by n tensor and B should be n by p tensor

return
    int([[24, 35, 114],[30, 52, 162]])

A * 2

return
    int([[2, 9, 25], [4, 16, 49]])
```

Element-wise multiplication(.*): **A** and **B** should have the same dimension

Element-wise multiplication examples:

```
A = int([1, 0, 3])
B = int([2, 3, 7])

A .* B  # A should be an m by n tensor and B should be n by p tensor

return
    int([2, 0, 21])
```

Division(/): **B** should be a 0-dim tensor. . The return value is an float-type tensor.

Division examples:

```
A = int([4, 5, 10])

A / 2  # B should be a 0-dim tensor

return
    float([2.0, 2.5, 5.0])
```

Power(^): **B** should be a 0-dim tensor.

Power examples:

```
A = int([2, 3, 5])

A ^ 2  # B should be a 0-dim tensor

return
    int([4, 9, 25])
```

Element-wise power(.^): **A** and **B** should have the same dimension.

Element-wise power examples:

```
A = int([2, 3, 5])
B = int([1, 0, 3])

A .^ B  # A and B should have the same dimension

return
    int([2, 1, 125])
```

Transpose(‘): Transpose is a unary operator.

Element-wise power examples:

```
A = int([[16, 2], [5, 11], [3, 8]])

A'  # A and B should have the same dimension

return
    int([[15, 5, 3],[2, 11, 8]])
```

Mod(%): **B** should be a 0-dim tensor.

Mod examples:

```
A = int([4, 6, 10])

A % 3  # B should be a 0-dim tensor

return
    int([1, 0, 1])
```

Floor Division(//): **B** should be a 0-dim tensor. The return value is an int-type tensor.

Remainder examples:

```
A = int([2, 3, 5])

A // 2  # B should be 0-dim tensor

return
    int([1, 1, 2])
```

3.3.2. Relational Operators

Relational operators are used to determining how two operands relate to each other. The relational operators always take two tensors as inputs. The return value of all the following relational operations is a logical tensor with elements set to logical 1 (**true**) where tensors **A** and **B** satisfy the operation; otherwise, the element is logical 0 (**false**).

Operator Name	**Description**
==	Determine equality
>=	Determine greater than or equal to
>	Determine greater than
<=	Determine less than or equal to
<	Determine less than
!=	Determine inequality

Examples are as follows:

```
X = int([1, 0, 0, 0, 1])
Y = int([0, 0, 1, 1, 0])
X >= Y
```

```
return
    int([1, 1, 0, 0, 1])
```

3.3.3. Logical Operators

TENLab also provides logical AND, OR and NOT operations. Logical operators should only appear between two expressions.

Operator Name	**Description**
&&	Logical OR operator
||	Logical AND operator
!	Logical NOT operator

Examples are as follows:

```
X = int([1 0 0 0 1])
Y = int([0 0 1 1 0])
!any(X)

return
    int([0])

any(X) || all(Y)

return
    int([1])
```

3.4. Keywords

Keywords are reserved words that are case-sensitive and cannot be used as identifiers. The following keywords are included in TENLab:

1. **Control**: if, elif, else, for, while, in, continue, break, return, read, print, exit, define
2. **Type**: int, float, string, void
3. **Parallel related:** parallel_define, overload, map, reduce, ___*___, ___+___, ___-___, using, end
4. **Tensor related**:

- cat: concatenate two tensors
- shape: return the shape of the tensor

Common tensor operations will be supported in the standard library.

3.5. Delimiters

A delimiter is a series of one or more characters used to define the boundary between separate, independent regions in plain text or other data streams.

1. **Whitespaces**: Any whitespace character such as a new line, a horizontal tab or a carriage return, used to separate tokens or statements from each other
2. Opening and closing **braces** '{' and '}', usually surrounding a block of statements to be treated as a unit, for example, the body of the method
3. Opening and closing **parentheses** '(' and ')', usually surrounding parts/pieces of an expression for the purposes of overriding the default precedence or simply for clarity, or surrounding the test part of an if/while statement
4. **Commas** ',', used to separate items in tensors.
5. **Colons** ':', used in a tensor declaration to index a specific range

3.6. Literals

Literals in TENLab can be divided into four categories: integer, float, character and tensor. Note that since everything is tensor, the first three are also a special zero-dimensional tensor.

3.6.1. Integer literals

An integer literal is a sequence of digits. This literal is only for decimal. For negative integers, add '-' in front of the digit sequence. '+' is not allowed to be prefixed. For example,

```
1341  -2123
```

3.6.2. Float literals

The first type of floating point literal is two sequences of digits with a dot between these two sequences. Either one of these two sequences could be empty, but not both. For negative integers, add '-' in front of the whole sequence (two sequences of digits plus a dot). '+' is not allowed to be prefixed.
The second type of floating point literal is two sequences with 'e' between these two sequences. Neither one of these two sequences could be empty. The sequence in the front is the first type of floating point literal or the integer literal. The sequence in the back must be the integer literal. For negative integers, add '-' in front of the whole sequence. The digit sequence behind the 'e' can also be prefixed with '-'.

The following are the examples:

```
3.2   .3124 123.  -1.2  -1.2e-10          1e-10
```

3.6.3. Character literals

A character literal is a single ASCII character enclosed by single quotes. For example,

```
'c'   '\0'  '\n'
```

3.6.4. Tensor literals

A tensor literal is grouped by brackets, with commas to separate different elements. Each element can be one of the four literals. For primitive data type tensors, dimensions and data type of each element must be matched. For example,

```
a = int([[1, 2], [3, 4], [5, 6]])
b = var([['a', 2, 4.], [4, '2']])
```

4. Declarations

4.1. Tensor Declarations

A tensor can be initialized in a couple of ways shown below.

4.1.1. From existing data

The users can wrap the existing data with brackets and a data type to declare a tensor:

```
A = int([0, 1, 2]) # A is now declared as a 1-dimensional tensor with 3
elements in it
```

To declare a 0-dim tensor, one can either declare with a given data type or not. For example,

```
# giving a data type
A = int(0)

# without giving any data type
A = 'a'
```

4.1.2. From shape and values with built-in function

The user can also create a tensor with the built-in functions, e.g., zeros(), ones(). These will be introduced in the following sections.

4.1.3. Create numerical ranges

One can also declare a tensor by giving the following expression:

```
Numerical-0: Numerical-1: Numerical-2
```

The Numerical-0 represents the beginning value of an interval, while Numerical-1 represents the end value of the interval. The interval includes the Numerical-0 but does not include the Numerical-1. And Numerical-2 represents the spacing between the values.

Here is an example of how to declare a tensor with such a statement.

```
A = 0:3:1
A

return
    [0, 1, 2]
```

4.2. Function Declarations

Function declaration in TENLab is first given a keyword, def, then followed by an identifier as the function name. Arguments are contained in parentheses after the function name. The body of a function is delimited by curly brackets. The function could either contain a return statement or not, but it could only return one tensor. Overall, the function declaration should be in the following form:

```
def foo(x, y) {
    statements
    return z # this is not required
}
```

5. Control Flow and Expressions

5.1. Control Flow

5.1.1. IF-ELSE Statements

In TENLab, we support the following three forms of conditional statements:

```
# if-only statement
if (expression) {
    statements
}
```

```
# if-else statement
if (expression) {
    statements
} else {
    statements
}

# if-elif statement
if (expression) {
    statements
} elif (expression) {
    statements
}
```

The expression will be evaluated from top to down. When one of them is evaluated as non-zero, then the corresponding series of statements will be executed. The expression must be delimited in parentheses. And the statements must be delimited in curly braces.

5.1.2. For Statements

For statements in TENLab is iterated over a list, which could be created and initialized in a tricky way as in the previous section. The for statement should be formed as:

```
for identifier in list {
      statements
}
```

Similar to the conditional statements, the statements inside the loop need to be delimited in curly braces.

5.1.3. While Statements

While statements in TENLab is pretty much the same as the conditional statements except for in a loop form:

```
while (expression) {
    statements
}
```

5.1.4. Expression

5.1.4.1. Precedence and Associativity Rules

The associativity rules of all the operators are shown in the following table, in order of precedence from top to bottom.

Operators	Associativity
=	right
||	left
&&	left
== !=	left
>= > <= <	left
+ -	left
* .* / % //	left
^ .^	right
‘	left
!	left

5.1.4.2. Assignment Expression

5.1.4.2.1. Tensor assignment

After declaration, the value in the tensor can still be modified, the user can access the tensor by index and modify it by giving a value. For example,

```
A = int([0, 1, 2])
A[0] = 1
A

return
   int([1, 1, 2])
```

5.1.4.2.2. Function Assignment

Although TENLab doesn’t support returning multiple tensors, one can pack multiple tensors in a tensor and return. When calling the function, you can unpack and assign multiple return values to different variables. For example,

```
x, y = foo(x, y)
```

6. User-Defined Parallel Environments

In TENLab, matrix operations can be paralleled in a map-reduce way. Users can define their own set of parallel functions to be used, called a parallel environment. Currently, we only allow parallel computation for +(Addition), -(Subtraction) and *(Matrix Multiplication).
To define a parallel environment, you do:

```
parallel_define <environment_name> {
    overload <operator_name> (X, Y) {
        map <name_of_function_one> {
            statements
            return <some_variable>
        }
        map <name_of_function_two> {
            statements
            return <some_variable>
        }
        reduce {
            return <some_variable>
        }
    }
}
```

Here map stands for the map functions, and reduce is for the reduce function. Note that there are two differences between map/reduce function definition and usual function definition:1. map/reduce functions must have return value. 2. Reduce function takes no arguments, and no names. In the reduce function, the name of map functions represents the return value of map functions. A concrete example is you may overload addition like this:

```
parallel_define EnvironmentTest {
    overload __+__ (x, y) {
        map f1 {
            z = x[:shape(x)[0]//2] + y[:shape(y)[0]//2]
            return z
        }
        map f2 {
            z = x[shape(x)[0]//2:] + y[shape(y)[0]//2:]
            return z
        }
        reduce {
            return cat(f1, f2)
        }
    }
}
```

To implement the function you defined, you can simply add using <environment name> to the beginning of the block and end <environment name> to end of the block:

```
using <environment_name>

statements

end <environment_name>
```

By doing so, the operations in the block, if specified by the environment, will use its implementation in environment definition. It is also possible to omit the end command. In that case, this parallel environment will be in effect until the end of this file.

Also, there is a standard library with parallel implementation provided called std_parallel. One can easily enjoy the power of multithreading by adding one single line in the front of your code and see everything becomes magically faster.

7. Built-in Functions

Matrix Control	
any(x)	If all elements of x are zero, return 0; otherwise, return 1
all(x)	If any element of x is zero, return 0; otherwise, return 1
sum(x)	Returns the sum of all elements in x
ones(x)	Return a tensor filled ones of the given shape x
zeros(x)	Return a tensor filled zeros of the given shape x
len(x)	Return the first dimension of shape(x)
Arithmetic	
int_of(x)	Convert x to an int tensor
float_of(x)	Convert x to a float tensor
floor(x)	Round every element in x towards negative infinity
ceil(x)	Round every element in x towards positive infinity

round(x)	Round every element in x towards closest decimal
abs(x)	Take abstract value of every element in x
log(x)	Compute log value of every element in x
Matrix computation	
inverse(x)	Compute the inverse of matrix x
solve(A,b)	Solve Ax=b, return x
svd(x)	Compute the svd of x , return U, S, V
eig(x)	Return a vector of eigenvalues of x.
eigv(x)	Return a tensor of eigenvectors of x.

8. Sample Codes

```
# Design a parallel environment with Strassen algorithm, which computes
matrix multiplication in O(N^2.8)
parallel_define Strassen {
    overload __*__ (x, y) {
        x11 = x[:shape(x)[0]//2,:shape(x)[1]//2]
        x12 = x[:shape(x)[0]//2,shape(x)[1]//2:]
        x21 = x[shape(x)[0]//2:,:shape(x)[1]//2]
        x22 = x[shape(x)[0]//2:,shape(x)[1]//2:]
        y11 = y[:shape(y)[0]//2,:shape(y)[1]//2]
        y12 = y[:shape(y)[0]//2,shape(y)[1]//2:]
        y21 = y[shape(y)[0]//2:,:shape(y)[1]//2]
        y22 = y[shape(y)[0]//2:,shape(y)[1]//2:]
        map f1 {
            return (x11+x22)*(y11+y22)
        }
        map f2 {
            return (x21+x22)*y11
        }
        map f3 {
            return x11*(y12-y22)
        }
        map f4 {
            return x22*(y21-y11)
        }
        map f5 {
            return (x11+x12)*y22
        }
        map f6 {
            return (x21-x11)*(y11+y12)
        }
        map f7 {
            return (x12-x22)*(y21+y22)
        }
        reduce {
            c11 = f1+f4-f5+f7
            c12 = f3+f5
            c21 = f2+f4
            c22 = f1-f2+f3+f6
            return cat(cat(c11,c12,0),cat(c21,c22,0),1)
        }
    }
}


using Strassen

# Markov Process
P = float([[3/4, 1/4], [1/4, 3/4]]) # transition matrix
s = float([[0.2, 0.8]]) # initial state
# judge difference of two matrices' elements is less than 1e-5
def diff(prev, curr)
{
    if (shape(prev) != shape(crr))
    {
        exit(-1)
    }
    delta = prev - cur
    flag = 1
    shape_t = shape(delta)
    for (i in 0:shape_t[0]:1)
    {
        for (d in delta[i,0:shape_t[1]:1])
        {
            if (abs(d) > 1e-5)
            {
                flag = 0
            }
        }
    }
    return flag
}
# multiply state and transition matrix
def mulPs(s, P)
{
    return s, s * P
}
# check state after four transitions
print(s * (P^4))
# Iterates until stable
s_prev = zeros(shape(s))
while (diff(s_prev, s) == 0)
{
    s_prev, s = mulPs(s, P)
}
```