PythonLinearAlgebra

branch GPL version | version 0.2.0 GPL | readme 0.2.0 GPL | last update 2/7/2020

This code implements part of the algorithm in Gilbert Strang's Introduction to Linear Algebra, and it is completely written in Python.

This branch is the main branch (licensed under GNU GPLv3), which will be used for further development. To see the minor branch (The Unlicense branch), please go to https://github.com/EPIC-WANG/PythonLinearAlgebra/tree/Unlicense.

CATALOG

INTRODUCTION

I'm a high school student who was self-studying Linear Algebra with the Professor Gillbert Strang's MIT OpenCourse. This is the code I used to practice linear algebra on.

This code includes most linear algebra operations as well as a basic interactive command line that allows users to perform computations.

Update Log

version 0.2.0:

- set positional and/or param-only arguments for serval methods.
- performance improved on `matrix.__abs__`, `matrix.__neg__`, `matrix.__eq__`.
- Add method `matrix.factorization_U`, producing U factorization.
- for `matrix.random_matrix` and `matrix.random_vector`, the param `type_: Union[type, tuple]` was replaced with params `complex_: bool =False` and `int_: bool = True`.
- `matrix.__repr__`: the printed matrix will display 0 or 0.0 instead of -0 or -0.0 .
- Random matrix generators could be called directly (without calling class matrix like `matrix.xxxxx`).
- A stronger, fast, shorter core was used for generating row-reduced form, computing sub-spaces, performing invert and matrix factorization.

Licence

Disclaimer of Warranty.

Limitation of Liability.

Tutorial

1. Install

To install and run this software, enter the folder "Python Linear Algebra" (https://github.com/EPIC-WANG/PythonLinearAlgebra/tree/GPL/Python Linear Algebra) and download the file in python script.

NOTE: To run the script, a python3 interpreter (3.8 and above) is required.

Also, this code could be imported by other python scripts by using:

```python
import alice
# or
from alice import *
```

or using interactive console:

```
this is a basic compute software of linear algebra developed by Wang Weizheng
For more information, type 'copyright()' or 'about()' in the console.


>>>
```

To exit the console (if you are using the console), type exit(). the console could accept python commands (a single line of code).

2. Basic operations

1. To create matrix or vector:

```
# create a matrix, use list[list]
X = matrix([[1,2,3],[2,3,4],[-1,-1,-1]])

# X =  [ 1    2    3]
#      [ 2    3    4]
#      [-1   -1   -1]

# create a vector, use list
Y = vector([1,2,3])

# Y = 1i_hat + 2j_hat + 3k_hat
#
# or it is same with
#
# Y = [1]
#     [2]
#     [3]
```

2. To perform basic operations with your matrices:

```
# multiply --
X*Y
# matrix:
#    [14]
#    [20]
#    [-6]

# add --
X+X
# matrix:
# [2, 4, 6]
# [4, 6, 8]
# [-2, -2, -2]

# transpose --
X.transpose()
# matrix:
# [1, 2, -1]
# [2, 3, -1]
# [3, 4, -1]

# invert --
X.invert()
# the matrix is singular. It is uninvertible.

# module (vector only) --
Y.module()
# 3.7416573867739413
```

3. To generate a random matrix:

```python
# generate a 4*4 matrix, from 0 to 10, integer value
X = matrix.random_matrix(5,6,2,-3,int)
# matrix:
# [-3, 2, 0, 0, 1, 1]
# [0, -3, -3, -2, 2, -2]
# [-1, 0, -3, 0, 0, 1]
# [-1, 2, 1, -2, 2, -2]
# [-2, 0, 0, -2, -1, 0]

# generate a default 4*4 matrix, from 0 to 10, by default, floating point value
X = print(matrix.random_matrix(type_ = float))
# matrix:
# [3.5043241003, 0.3713558007, 3.7324463516, 5.2982786823]
# [2.4268259983, 4.8068851973, 9.7347697987, 6.8648311404]
# [0.189252822, 7.126140657, 5.6800361683, 7.6905833487]
# [8.1493381191, 7.0644692704, 1.7996603637, 6.6048587747]

# generate a 3*4 matrix, from 0+0j to 10+10j, by default, integer real and imagine value
print(matrix.random_matrix(type_ = (complex, int), row = 3))
# matrix:
# [(3+2j), (10+3j), 1j, (6+0j)]
# [(9+5j), 7j, (8+5j), (3+6j)]
# [(1+4j), (8+4j), (7+8j), (10+9j)]
```

4. To perform advanced operations:

```python
Z = matrix([[1,2,3],[2,3,4],[-1,-1,-1],[3,4,5]])

Z.null_space()
# [vector:
# [1.0, -2.0, 1.0]
# ]

Z.column_space()
# [vector:
# [1, 2, -1, 3]
# , vector:
# [2, 3, -1, 4]
# ]

# get the row reduced form of matrix Z
Z.reduced_echelon_form()
# matrix:
# [1.0, 0.0, -1.0]
# [-0.0, 1.0, 2.0]
# [0.0, 0.0, 0.0]
# [0.0, 0.0, 0.0]

matrix.least_square([[1,2],[2,3],[4,4],[5,4]])
# least square of [[1, 2], [2, 3], [4, 4], [5, 4]] for y = Cx + D:
# C = 0.5000000000000006, D = 1.7499999999999998
```

Other methods could be found in the APIs

APIs (Public methods)

Note

1. these methods are recommended to use within basic computations. for more information, see the Note

2. all the methods, without notice, returns matrix (or vector) object.

The APIs under`<class 'alice.matrix'>`:

1. `matrix(my_matrix)`:

to create a matrix, use matrix(*your_matrix*)

- **param:** *my_matrix*: the matrix, use *list[list]* to input a matrix. Type: *Union[int, float, complex]* are supported as elements in matrix.

For example, To create:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

, type:

```
>>> X = matrix([[1,2,3],[2,3,4],[3,4,5]])
```

Also, you could create a matrix with different types of element:

$$T = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \qquad \theta = 2\pi$$

, type:

```
>>> theta = 2 * pi
>>> T = matrix([[cos(theta), -sin(theta)], [sin(theta), cos(theta)]])
```

Or create:

$$A = \begin{bmatrix} 2+3i & 4+2i \\ -1-2i & 3-4i \end{bmatrix}$$

, type (use j or J as complex part):

```
>>> A = matrix([[2+3j, 4+2j], [-1-2j, 3-4j]])
```

note, use double square braces to represent a matrix, even if your matrix has only one row.

For example:

$$X = \begin{bmatrix} 3 & 4 & 5 & 6 \end{bmatrix}$$

, type:

```
>>> X = matrix([[3, 4, 5, 6]])
```

the operations between matrix (+, -, *) are supported.

to compute:

$$\begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 4 & 3 \\ 2 & 2 & 4 \end{bmatrix}$$

, type:

```
>>> matrix([[1,2],[2,3],[3,4]]) * matrix([[5,4,3],[2,2,4]])
```

2. `matrix.get_value(self)`

Return the *matrix* object of the matrix variable.

- **return:** *list[list]* object, same as the param *my_matrix* in *matrix()*.

3. `matrix.get_row(self, row: int)`

Return the selected row of the matrix (index starts with 0).

- **return:** *list* object.

4. `get_column(self, column: int)`

Return the selected column of the matrix (index starts with 0).

- **return:** *list* object.

5. `get_size(self)`

Return the size of the matrix.

- **return:** *(int, int)* object.

6. `matrix.null_space(self)`:

Return the null space of a matrix, returns a list of vector.

To compute:

$$N\left(\begin{bmatrix}1 & 2 & 3\\ 2 & 3 & 4\\ 3 & 4 & 5\\ 1 & 1 & 1\end{bmatrix}\right) = span\left(\begin{bmatrix}1.0\\ -2.0\\ 1.0\end{bmatrix}\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}\right)$$

type:

```
>>> print(matrix([[1,2,3],[2,3,4],[3,4,5],[1,1,1]]).null_space())
[vector:
[1.0, -2.0, 1.0]
]
```

7. `matrix.left_null_space(self)`

returns the null space transpose (left null space) of the matrix.

8. `matrix.gauss_jordan_elimination(self)`

perform a gauss jordan elimination and return the gauss jordan elimination matrix.

to solve the equation set:

$$\begin{cases} x + 2y + 3z = 4 \\ 2x + 3y + 4z = 6 \quad\quad x, y, z \in \mathbb{R} \\ 3x + 3y + 3z = 4 \end{cases}$$

, transfer the equation set into matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 6 \\ 3 & 3 & 3 & 4 \end{bmatrix}$$

```
>>> X = matrix([[1,2,3,4],[2,3,4,6],[3,3,3,4]])
>>> print(X.gauss_jordan_elimination())
matrix:
[1.0, 0.0, -1.0, 0.0]
[-0.0, 1.0, 2.0, 2.0]
[0.0, 0.0, 0.0, -2.0]
```

9. `matrix.reduced_echelon_form(self)`

return the row reduced form of the matrix.

```
>>> X = matrix([[1,2,3,4],[2,3,4,6],[3,3,3,4],[3,4,5,6]])
>>> print(X.reduced_echelon_form())
matrix:
[1.0, 0.0, -1.0, 0.0]
[0.0, 1.0, 2.0, 0.0]
[-0.0, -0.0, -0.0, 1.0]
[0.0, 0.0, 0.0, 0.0]
```

10. `matrix.column_space(self, *, return_index: bool = False)`

return the column space (or the pivot column index) of the matrix.

- **param:** *return_index*: return the index of the pivot column, the default value is False.
- **return:** *matrix* object when *return_index* is False, otherwise *set* object.

to compute:

$$C\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 6 \\ 3 & 3 & 3 & 4 \\ 3 & 4 & 5 & 6 \end{bmatrix}\right) = span\left(\begin{bmatrix} 1 \\ 2 \\ 3 \\ 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 3 \\ 4 \end{bmatrix} \begin{bmatrix} 4 \\ 6 \\ 4 \\ 6 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}\right)$$

type:

```
>>> print(matrix([[1,2,3,4],[2,3,4,6],[3,3,3,4],[3,4,5,6]]).column_space())
[vector:
[1, 2, 3, 3]
, vector:
[2, 3, 3, 4]
, vector:
[4, 6, 4, 6]
]
```

11. `matrix.row_space(self, *, return_index: bool = False)`

return the row space (or the pivot row index) of the matrix.

- **param:** *return_index*: return the index of the pivot row, the default value is False.
- **return:** *matrix* object when *return_index* is False, otherwise *set* object.

12. `matrix.transpose(self)`

return the transpose matrix.

to compute:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 6 \\ 3 & 3 & 3 & 4 \\ 3 & 4 & 5 & 6 \end{bmatrix}^T$$

type:

```
>>> print(matrix([[1,2,3,4],[2,3,4,6],[3,3,3,4],[3,4,5,6]]).transpose())
matrix:
[1, 2, 3, 3]
[2, 3, 3, 4]
[3, 4, 3, 5]
[4, 6, 4, 6]
```

13. `matrix.invert(self)`

return the invert of the matrix, if matrix couldn't be inverted, a message: *the matrix is singular. It is un invertible.* will be printed.

to compute:

$$\begin{bmatrix} 0 & 2 & 1 & 3 \\ 2 & 2 & 6 & 4 \\ 3 & 4 & 3 & 4 \\ 3 & 4 & 7 & 6 \end{bmatrix}^{-1}$$

type:

```
>>> print(matrix([[0,2,1,3],[2,2,6,4],[3,4,3,4],[3,4,7,6]]).invert())
matrix:
[1.0, 5.0, 2.5, -5.5]
[-2.0, -7.5, -3.0, 8.0]
[-1.0, -3.0, -1.5, 3.5]
[2.0, 6.0, 2.5, -6.5]
```

14. `matrix.combine(self, target)`

Combine two matrices into one single matrix, the target matrix will appear at the right side of the given matrix.

For example, combine X and Y:

$$X = \begin{bmatrix} 1 & 1 & 3 \\ 2 & 2 & 4 \\ 3 & 4 & 3 \end{bmatrix}, Y = \begin{bmatrix} 4 & 3 & 3 \\ 3 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}$$

The result is:

$$\begin{bmatrix} 1 & 1 & 3 & 4 & 3 & 3 \\ 2 & 2 & 4 & 3 & 2 & 3 \\ 3 & 4 & 3 & 2 & 1 & 3 \end{bmatrix}$$

type:

```
>>> X = matrix([[1,1,3],[2,2,4],[3,4,3]]); Y = matrix([[4,3,3],[3,2,3],[2,1,3]])
>>> print(X.combine(Y))
matrix:
[1, 1, 3, 4, 3, 3]
[2, 2, 4, 3, 2, 3]
[3, 4, 3, 2, 1, 3]
```

15. `matrix.cast_to_complex(self, target)`

create a complex matrix and map the complex part to *target*.

- **param:** *target*: the matrix which will be mapped in to complex part

To map X and Y:

$$X = \begin{bmatrix} 1 & 1 & 3 \\ 2 & 2 & 4 \\ 3 & 4 & 3 \end{bmatrix}, Y = \begin{bmatrix} 4 & 3 & 3 \\ 3 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}$$

The result is:

$$\begin{bmatrix} 1+4i & 1+3i & 3+3i \\ 2+3i & 2+2i & 4+3i \\ 3+2i & 4+i & 3+3i \end{bmatrix}$$

type:

```
>>> X = matrix([[1,1,3],[2,2,4],[3,4,3]]); Y = matrix([[4,3,3],[3,2,3],[2,1,3]])
>>> print(X.cast_to_complex(Y))
matrix:
[(1+4j), (1+3j), (3+3j)]
[(2+3j), (2+2j), (4+3j)]
[(3+2j), (4+1j), (3+3j)]
```

16. `matrix.conjugate(self)`

Return the conjugate matrix of the given matrix.

17. `matrix.conjugate_transpose(self)`

Return the conjugate transpose matrix of the given matrix. it is same as `X.transpose().conjugate()`.

18. `matrix.round_to_square_matrix(self)`

Return a square matrix with $\mathbb{R}^{(max(m,n) \times max(m,n))}$, extra columns or rows will be filled with 0.

For example:

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$$

will be filled to:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 4 & 5 & 0 & 0 \end{bmatrix}$$

19. `matrix.get_projection_matrix(self)`

Return the projection matrix of a given matrix, This method uses to produce the projection matrix.

20. `matrix.project(self, target, *, to_orthogonal_space: bool = False)`

Return the projected matrix (even if you input a vector, see `vector()` methods)

- **param:** *target*: the target matrix on which you want to project.
- **param:** *to_orthogonal_space*: whether to project on the orthogonal space $(I - P)$ of the target matrix.

For example, to project matrix A on matrix X to get projected matrix p:

$$A = \begin{bmatrix} 1 & 1 & 2 & 5 \\ 2 & 2 & 3 & 3 \\ 3 & 4 & 4 & 8 \\ 4 & 5 & 5 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 2 & 3 & 1 & 4 \\ 3 & 4 & 1 & 2 \\ 4 & 5 & 1 & 7 \end{bmatrix}$$

$$p = \begin{bmatrix} 1.0 & 0.8444444444 & 2.0 & 4.4944444444 \\ 2.0 & 2.2444444444 & 3.0 & 3.7944444444 \\ 3.0 & 3.9777777778 & 4.0 & 7.9277777778 \\ 4.0 & 4.9333333333 & 5.0 & 0.7833333333 \end{bmatrix}$$

type:

```
>>> A = matrix([[1,1,2,5],[2,2,3,3],[3,4,4,8],[4,5,5,1]])
>>> X = matrix([[1,2,1,3],[2,3,1,4],[3,4,1,2],[4,5,1,7]])
>>> print(A.project(X))
matrix:
[1.0, 0.8444444444, 2.0, 4.4944444444]
[2.0, 2.2444444444, 3.0, 3.7944444444]
[3.0, 3.9777777778, 4.0, 7.9277777778]
[4.0, 4.9333333333, 5.0, 0.7833333333]
```

21. `matrix.least_square(data: list)`

compute the linear regression equation with the data list.

- **param:** *data*: input the data which represented by list[list].
- **return:** *None* (the data will be printed in the console.)

For example, to compute with the data set with $y = Cx + D$:

$X = (1, 2); (2, 3); (3, 3); (4, 6); (5, 7)$

type:

```
>>> matrix.least_square([[1,2],[2,3],[3,3],[4,6],[5,7]])

least square of [[1, 2], [2, 3], [3, 3], [4, 6], [5, 7]] for y = Cx + D:
 C = 1.2999999999999998, D = 0.3000000000000007
```

22. `matrix.to_vector(self, split_index: Union[range, list, tuple, set, None] = None)`

split the matrix with selected column and return a list of vector.

- **param:** *split_index*: the index of splitting the column in to the vector list. ***Default*** *None*, splitting all the columns in to the list.
- **return:** a list of vectors.

For example:

```
>>> # to split X:
>>> X = matrix([[1,2,3],[2,3,4]])
>>> # split the X with index [0,2]:
>>> print(X.to_vector([0,2]))
[vector:
[1, 2]
, vector:
[3, 4]
]
```

23. `matrix.to_list(self)`

- **return** a *list* object of matrix.

24. `matrix.to_str(self)`

- **return** a *str* object of the matrix's list.

25. `matrix.factorization_U(self, *, strict: bool = False)`

return the U factorization of the matrix.

- **param:** strict: to produce a strict upper-triangle matrix instead of upper-triangle matrix.

The APIs under`<class 'alice.vector'>`:

1. `vector()`

To create a vector, input `vector(_your_vector_)`. A vector only contains one column. You can input your vector as if inputting a single-column matrix. Or, you can use a pair of square brackets to represent a vector.

For example, to input X:

$$X = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

type:

```
>>> X = vector([[1],[2],[3],[4]])
# or using a better way:
>>> X = vector([1, 2, 3, 4])
```

All the *vector* element could perform the operations in class *matrix*.

1. `vector.dot(self, target)`

return the dot product of two vectors.

- **param:** *target*: the target vector.
- **return:** Result of the dot operation. a *int* or *float* object.

Note: the result is same as *your_matrix*.transpose() * *target*.

2. `vector.module(self)`

return the module (length) of the vector.

- **return:** a *int* or *float* object.

The matrices spanner APIs:

Note: all the matrices spanner methods are ***static***.

1. `matrix.identity_matrix(size)`

Generate and return a identity matrix with size *size* * *size*.

- **param:** *size*: the size of the identity matrix. type *int*.

2. `matrix.zero_matrix(size)`

Generate and return a zero matrix with size *size* * *size*.

- **param:** *size*: the size of the identity matrix. type *int*.

3. `matrix.random_matrix(row, col, /, max_, min_, *, int_, complex_, seed)`

Generate and return a random matrix with size *row* * *col*.

- **param:** *row*: the row count of the random matrix. type: *int*, default: *4*.
- **param:** *column*: the column count of the random matrix. type: *int*, default: *4*.
- **param:** *max_*: the max random value of the random matrix. type: *Union[float, int, complex]*, default: *10*.
- **param:** *min_*: the min random value of the random matrix. type: *Union[float, int, complex]*, default: *0*.
- **param:** *int_*: whether generate a integer matrix or not. type: *bool*, default: *True*.
- **param:** *complex_*: whether generate a complex matrix or not. type: *bool*, default: *False*.
- **param:** *seed*: the seed of the generator. Type: *Any*, default: *None*, (random choose a seed).

NOTE: If the matrix is a complex matrix, max \ min random value will be set based on the complex and the real part of param *max_* \ *min_* (if the type of *max_* \ *min_* is complex) or the complex and the real part are both param *max_* \ *min_* (if the type of *max_* is float or int).

To generate a random matrix:

```
# a default matrix
>>> print(matrix.random_matrix())
matrix:
[2, 2, 3, 1]
[1, 6, 10, 2]
```

```
[1, 1, 0, 1]
[3, 5, 3, 9]

# a floating point matrix with 6 rows and 7 columns, max: 10, min: -10
>>> print(matrix.random_matrix(6, 7, 10, -10, int_=False))
matrix:
[3.8615808488, 1.2674775011, 4.847243793, -1.0016590897, -9.4858038654, -0.8995657768, 9.309763
[-1.660605956, 8.8394380788, 9.2824632231, -8.4695465347, -6.041755394, 4.2838418496, 5.0610248
[9.8594147713, -7.4514601363, 8.2789013583, 4.504841112, -6.3012155999, -0.6093246257, 3.923953
[-5.4559652587, -0.3480076003, 3.8279018731, 3.4219349079, 6.7735594802, -5.6834095914, -3.3381
[0.5948846068, -9.3837370141, -4.5305839598, 0.5697977751, 5.9328243528, -5.7344232839, -1.3426
[-0.5584473399, -1.3399297884, -1.5035082544, 2.665211314, 4.4190972648, 4.3496340898, -7.72982

# a complex integer matrix with 5 rows and 5 columns, max real part: 2
# max complex part: 3, min real part: -1, min complex part: 1
>>> print(matrix.random_matrix(5, 5, 2+3j, -1+j, complex_=True, int_=False))
matrix:
[(1+3j), (2+2j), (1+2j), 3j, 2j]
[(1+2j), (1+1j), (2+3j), (2+2j), (2+3j)]
[(1+2j), 2j, (1+2j), 3j, (-1+2j)]
[(-1+3j), (1+2j), (2+1j), 1j, (-1+2j)]
[(-1+1j), (2+3j), (2+2j), (-1+3j), (-1+3j)]
```

4. `matrix.random_vector(length, max_, min_, type_, seed):`

generate a random vector with length *length*.

- **param:** *length*: the length of the random vector. type: *int*, default: *3*.
- **param:** *max_*: the max random value of the random vector. type: *Union[float, int, complex]*, default: *10*.
- **param:** *min_*: the min random value of the random vector. type: *Union[float, int, complex]*, default: *0*.
- **param:** *int_*: whether generate a integer vector or not. type: *bool*, default: *True*.
- **param:** *complex_*: whether generate a complex vector or not. type: *bool*, default: *False*.
- **param:** *seed*: the seed of the generator. Type: *Any*, default: *None*, (random choose a seed).

NOTE: The param: *max_* and *min_* is same as `matrix.random_matrix()`

Private methods

1. All private and static methods (starts with __) will overwrite the input matrix which indicates with the parameter *my_matrix*. If you want to remain the matrix unchanged, please use `copy.deepcopy(my_matrix)` or call the methods by `<class '__main__.matrix'>` object.
2. It is recommend to call static methods in `<class '__main__.matrix'>` if you decide to perform intense computing or to build methods inside `<class '__main__.matrix'>`, because most static methods in `<class`

`'__main__.matrix'>` returns `list[list]` object instead of matrix, which provides faster operation speed. But static methods are **NOT SUITABLE** for interactive programming and operation outside `<class '__main__.matrix'>`.