6. Matrices

6.1. Matrices

Creating matrices from the entries. Matrices can be represented in Python using 2-dimensional numpy array or list structure (list of lists). For example, the 3×4 matrix

$$A = \begin{bmatrix} 0 & 1 & -2.3 & 0.1 \\ 1.3 & 4 & -0.1 & 1 \\ 4.1 & -1 & 0 & 1.7 \end{bmatrix}$$

is constructed in Python as

In [ ]:
```
A = np.array([[0,1,-2.3,0.1], [1.3, 4, -0.1, 0], [4.1, -1.0, 0,
↪  1.7]])
A.shape
```

Out[ ]:
```
(3, 4)
```

In [ ]:
```
m,n = A.shape
print('m:', m)
print('n:', n)
```

```
m: 3
n: 4
```

Here we use numpy shape property to get the current shape of an array. As an example, we create a function to check if a matrix is tall.

In [ ]:
```
tall = lambda X: A.shape[0] > A.shape[1]
tall(A)
```

Out[ ]:
```
False
```

In the function definition, the number of rows and the number of columns are combined using the relational operator `>`, which gives a Boolean.

If matrix `A` is expressed as a list of lists, we can get the dimensions of the array using the `len()` function as follows.

```
In [ ]: A = [[0,1,-2.3,0.1], [1.3, 4, -0.1, 0], [4.1, -1.0, 0, 1.7]]
        print('m:', len(A))
        print('n:', len(A[0]))
```

```
m: 3
n: 4
```

Indexing entries. We get the ith row jth column entry of a matrix `A` using `A[i-1, j-1]` since Python starts indexing from 0.

```
In [ ]: A[0,2] #Get the third element in the first row
```

```
Out[ ]: -2.3
```

We can also assign a new value to a matrix entry using similar bracket notation.

```
In [ ]: A[0,2] = 7.5 #Set the third element in the first row of A to 7.5
        A
```

```
Out[ ]: array([[ 0. ,  1. ,  7.5,  0.1],
               [ 1.3,  4. , -0.1,  0. ],
               [ 4.1, -1. ,  0. ,  1.7]])
```

Equality of matrices. `A == B` determines whether the matrices `A` and `B` are equal. For numpy arrays `A` and `B`, the expression `A == B` creates a matrix whose entries are Boolean, depending on whether the corresponding entries of `A` and `B` are the same. The expression `sum(A == B)` gives the number of entries of `A` and `B` that are equal.

```
In [ ]: A = np.array([[0,1,-2.3,0.1], [1.3, 4, -0.1, 0], [4.1, -1.0, 0,
        ↪ 1.7]])
        B = A.copy()
        A == B
```

```
Out[ ]: array([[ True,  True,  True,  True],
               [ True,  True,  True,  True],
               [ True,  True,  True,  True]])
```

```
In [ ]: B[0,3] = 100  #re-assign one of the elements in B
        np.sum(A==B)
```

```
Out[ ]: 11
```

Alternatively, if `A` and `B` are list expressions (list of lists), the expression `A == B` gives a Boolean value that indicates whether the matrices are equivalent.

```
In [ ]: A = [[0,1,-2.3,0.1],[1.3, 4, -0.1, 0],[4.1, -1.0, 0, 1.7]]
        B = [[0,1,-2.3,100],[1.3, 4, -0.1, 0],[4.1, -1.0, 0, 1.7]]
        A == B
```

```
Out[ ]: False
```

Row and column vectors. In VMLS, n-vectors are the same as $n \times 1$ matrices. In Python, there is a subtle difference between a 1D array, a column vector and a row vector.

```
In [ ]: # a 3 vector
        a = np.array([-2.1, -3, 0])
        a.shape
```

```
Out[ ]: (3,)
```

```
In [ ]: # a 3-vector or a 3 by 1 matrix
        b = np.array([[-2.1],[-3],[0]])
        b.shape
```

```
Out[ ]: (3, 1)
```

```
In [ ]: # a 3-row vector or a 1 by 3 matrix
        c = np.array([[-2.1, -3, 0]])
        c.shape
```

```
Out[ ]: (1, 3)
```

You can see `a` has shape `(3,)`, meaning that it is a 1D array, while `b` has shape `(3, 1)`, meaning that it is a 3×1 matrix. This small subtlety generally won't affect you and `b` can be reshaped to look like `c` easily with the command `b.reshape(3)`.

Slicing and submatrices. Using colon notation you can extract a submatrix.

```
In [ ]: A = np.array([[-1, 0, 1, 0],[2, -3, 0, 1],[0, 4, -2, 1]])
        A[0:2, 2:4]
```

```
Out[ ]: array([[1, 0], [0, 1]])
```

This is different from the mathematical notation in VMLS due to the fact that Python starts index at 0. You can also assign a submatrix using slicing (index range) notation. A very useful shortcut is the index range : which refers to the whole index range for that index. This can be used to extract the rows and columns of a matrix.

```
In [ ]: # Third column of A
        A[:,2]
```

```
Out[ ]: array([ 1,  0, -2])
```

```
In [ ]: # Second row of A, returned as vector
        A[1,:]
```

```
Out[ ]: array([ 2, -3,  0,  1])
```

The numpy function `reshape()` gives a new $k \times l$ matrix. (We must have $mn = kl$, *i.e.*, the original and reshaped matrix must have the same number of entries. This is not standard mathematical notation, but they can be useful in Python.

```
In [ ]: A.reshape((6,2))
```

```
Out[ ]: array([[-1,  0],
               [ 1,  0],
               [ 2, -3],
               [ 0,  1],
               [ 0,  4],
               [-2,  1]])
```

```
In [ ]: A.reshape((3,2))
```

```
Out[ ]: ValueError: cannot reshape array of size 12 into shape (3,2)
```

Block matrices. Block matrices are constructed in Python very much as in the standard mathematical notation in VMLS. We can use the `np.block()` method to construct block matrices in Python.

```
In [ ]: B = np.array([0,2,3])           #1 by 3 matrix
        C = np.array([-1])              #1 by 1 matrix
        D = np.array([[2,2,1],[1,3,5]]) #2 by 3 matrix
        E = np.array([[4],[4]])          #2 by 1 matrix
        # Constrcut 3 by 4 block matrix
```

```
A = np.block([[B,C],[D,E]])
A
```

Out[ ]:
```
array([[ 0,  2,  3, -1],
       [ 2,  2,  1,  4],
       [ 1,  3,  5,  4]])
```

Column and row interpretation of a matrix. An $m \times n$ matrix A can be interpreted as a collection of n m-vectors (its columns) or a collection of m n-row-vectors (its rows). Column vectors can be converted into a matrix using the horizontal function `np.hstack()` and row vectors can be converted into a matrix using the vertical function `np.vstack()`.

In [ ]:
```
a = [1,2]
b = [4,5]
c = [7,8]
A = np.vstack([a,b,c])
B = np.hstack([a,b,c])
print('matrix A :',A)
print('dimensions of A :', A.shape)
print('matrix B :',B)
print('dimensions of B :', B.shape)
```

```
matrix A : [[1 2]
 [4 5]
 [7 8]]
dimensions of A : (3, 2)
matrix B : [1 2 4 5 7 8]
dimensions of B : (6,)
```

In [ ]:
```
a = [[1],[2]]
b = [[4],[5]]
c = [[7],[8]]
A = np.vstack([a,b,c])
B = np.hstack([a,b,c])
print('matrix A :',A)
print('dimensions of A :', A.shape)
print('matrix B :',B)
print('dimensions of B :', B.shape)
```

```
matrix A : [[1]
 [2]
 [4]
 [5]
 [7]
 [8]]
dimensions of A : (6, 1)
matrix B : [[1 4 7]
 [2 5 8]]
dimensions of B : (2, 3)
```

Another useful expression is `np.c_` and `np.r_`, which allow us to stack column (and row) vectors with other vectors (and matrices).

In [ ]:
```
np.c_[-np.identity(3), np.zeros(3)]
```

Out[ ]:
```
array([[-1., -0., -0.,  0.],
       [-0., -1., -0.,  0.],
       [-0., -0., -1.,  0.]])
```

In [ ]:
```
np.r_[np.identity(3), np.zeros((1,3))]
```

Out[ ]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])
```

6.2. Zero and identity matrices

Zero matrices. A zero matrix of size $m \times n$ is created using `np.zeros((m,n))`. Note the double brackets!

In [ ]:
```
np.zeros((2,2))
```

Out[ ]:
```
array([[0., 0.],
       [0., 0.]])
```

Identity matrices. Identity matrices in Python can be created in many ways, for example, by starting with a zero matrix and then setting the diagonal entries to one. You can also use the `np.identity(n)` function to create an identity matrix of dimension n.