SIGNAL PROCESSING SYSTEM DESIGN LAB

YASH ANAND
(23EE65R22)

1 Introduction

```
pip install numpy
pip install matplotlib
pip install scipy
```

These are the libraries which we have installed in the juypter using pip package manager.

1.1
Here we have imported Numpy as np.
The next line creates an array of 32 bit floating point numbers.
The itemize property shows the number of bytes per item which is 4 in our case as output.
OUTPUT-4

1.2
This command computes the sine of the input array of all ones, using Numpy's unary function, np.sin.
OUTPUT-array([0.84147096, 0.84147096, 0.84147096], dtype=float32)

1.3
Here this command computes dimensions and size of the array
OUTPUT-(2, 3)

1.4
Here we are using the numpy slicing rules where the colon ":" character selects all elements in the corresponding row or column.
OUTPUT-

```
array([1, 4])
array([2, 5])
array([1, 2, 3])
array([4, 5, 6])
```

1.5
Here we have used the Numpy slicing that can select sections of an array depending upon the slicing rule used.
OUTPUT-

```
array([[1, 2, 3], [4, 5, 6]])
array([[2, 3], [5, 6]])
array([[1, 3], [4, 6]])
```

1.6
Here we have used the pass-by-reference semantics of the Numpy,so it has created the views into the existing array without implicitly copying it.
Also np.ones creates a 3x3 NumPy array with all elements set to 1.
OUTPUT-

```
array([[ 1., 1., 1.], [ 1., 1., 1.], [ 1., 1., 1.]])
array([[ 1., 1., 1., 1.], [ 1., 1., 1., 1.], [ 1., 1., 1., 1.]])
array([[1., 1., 1., 1.], [1., 1., 1., 1.], [1., 1., 1., 1.]])
```

Here the last command will execute same as above, but it will not assign a value to it.

1.7
Here using this code we can see that changing the individual elements of x does not affect y because we have made

a copy in 1.7.As here we have changed the element in x at [0,0] position to 999 but this value is not changed in y
OUTPUT-
array([[999., 1., 1.], [1., 1., 1.], [1., 1., 1.]]) array([[1., 1., 1., 1.], [1., 1., 1., 1.], [1., 1., 1., 1.]])

1.8
Here x[:2,:2] means to slice the first 2 rows and the first 2 columns of the array x
The [0,0] notation is used to access an element in an array. Here we can see that Numpy views point at the same memory as their parents which is a consequence of the pass-by-reference semantics, so on changing an element in x updates the corresponding element in y.
OUTPUT-
array([[999., 1., 1.], [1., 1., 1.], [1., 1., 1.]])
array([[999., 1.], [1., 1.]])

1.9
From the commands we have used in this listing, we can see that the indexing can also create copies.
Here, y is a copy, not a view, because it was created using indexing whereas z was created using slicing. So here even though y and z have the same entries, only z is affected by changes to x.
OUTPUT-
array([0, 1, 2, 3, 4])
array([0, 1, 2])
array([0, 1, 2])
array([999, 1, 2, 3, 4])
array([0, 1, 2])
array([999, 1, 2])

1.10
The [].flags.owndata attribute tells us whether the array [] owns its own data or borrows it from another object.
Numpy arrays have a built-in flags.owndata property that can help keep track of views until you get the hang of them.
OUTPUT-
TRUE
TRUE
FALSE

1.11
The np.matrix() function creates a new matrix from a list of lists
The * operator is used to perform matrix multiplication
Also Numpy arrays support elementwise multiplication, not row-column multiplication.
OUTPUT-
matrix([[1], [4], [7]])

1.12
From the given command we can see that it is easy and fast to convert between Numpy arrays and matrices because doing so need not imply any memory copying.
Even here we did not have to bother about converting x because the left-to-right evaluation automatically handles that.
OUTPUT-
array([[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]])
matrix([[3.], [3.], [3.]])

1.13
Here using meshgrid(,) command we can creates two-dimensional grids.
OUTPUT-
array([[0, 1], [0, 1]])
array([[0, 0], [1, 1]])

1.14
From the given command we can add two arrays that have compatible shapes.
OUTPUT-

```
array([[0, 1], [1, 2]])
```

1.15

Here we can see that using Numpy broadcasting, we can skip creating compatible arrays using meshgrid and instead accomplish the same thing automatically by using the None singleton to inject an additional compatible dimension.

OUTPUT-

```
array([0, 1]
array([0, 1])
array([[0, 1], [1, 2]])
array([[0, 1], [1, 2]])
```

1.16

In this Listing, the array shapes are different, so the addition of x and y is not possible without Numpy broadcasting. Also here we can see that the broadcasting generates the same output as using the compatible array generated by meshgrid.

OUTPUT-

```
array([[0, 1], [0, 1], [0, 1]])
array([[0, 0], [1, 1], [2, 2]])
array([[0, 1], [1, 2], [2, 3]])
array([[0, 1], [1, 2], [2, 3]])
```

1.17

Numpy broadcasting works in multiple dimensions. We have started here with three one-dimensional arrays and create a three-dimensional output using broadcasting. The x+y[:None] part created a conforming two-dimensional array, and due to the left-to-right evaluation order, this two-dimensional intermediate product is broadcast against the z variable, whose two None dimensions create an output three-dimensional array.

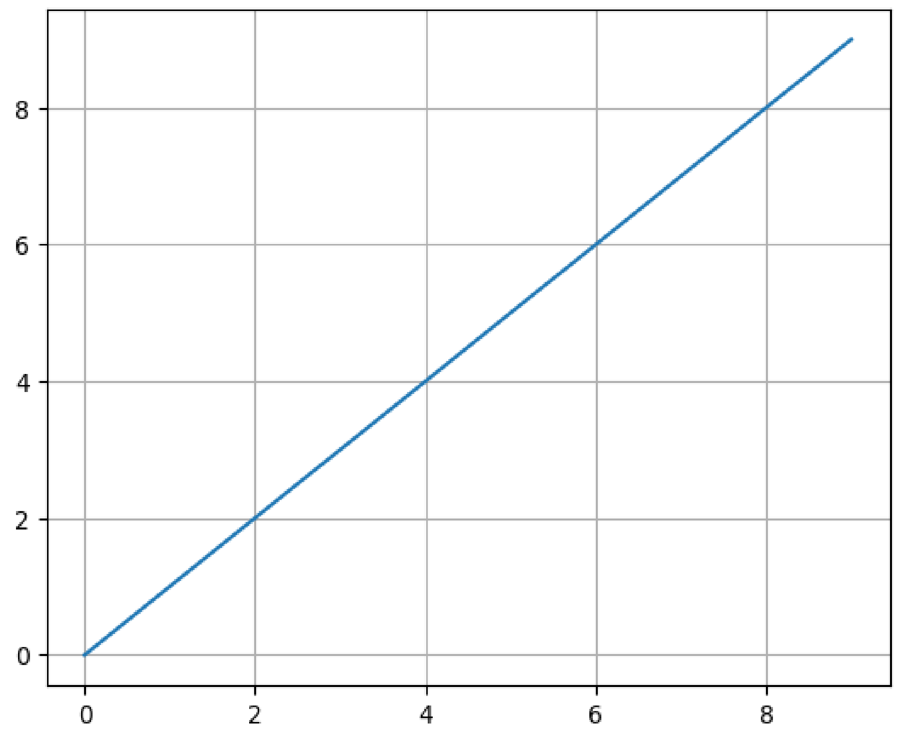
OUTPUT-

```
array([ [[0, 1], [1, 2], [2, 3]], [[1, 2], [2, 3], [3, 4]], [[2, 3], [3, 4], [4, 5]], [[3, 4], [4, 5], [5, 6]]])
```

1.18

Here we have imported theMatplotlib module using the respective naming convention.
The ploted a range of numbers and forces the plot to render.

OUTPUT-



0
2
4
6
8
0
2
4
6
8