Session 1

Use test driven development (i.e. write the tests first) to design functions to give the square, cube and an arbitary power N for a number a.

In [1]:
```
import unittest

def square(a):
    return a**2
def cube(a):
    return a**3
def power(a,N):
    return a**N

#Class inherits from unittest
class test_square(unittest.TestCase):
    def test_square(self):
        assert square(2.) == 4.
    def test_cube(self):
        assert cube(2.) == 8.
    def test_power(self):
        assert power(2.,4) == 16.

#This should simply simply be
#unittest.main()
#to run the unittest but bug in this notebook needs arguments below
unittest.main(argv=["first-arg-is-ignored"], exit=False)
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.010s

OK
```

Out[1]:
```
<unittest.main.TestProgram at 0x4157c50>
```

Save these functions to number.py, put the tests inside an if statement: if **name** == "**main**". In a new script/session import with import number as num and try to call the functions.

In [3]:
```
def square(a):
    return a**2
def cube(a):
    return a**3
def power(a,N):
    return a**N



if __name__ == "__main__":

    import unittest

    class test_square(unittest.TestCase):
        def test_square(self):
            assert square(2.) == 4.
        def test_cube(self):
            assert cube(2.) == 8.
        def test_power(self):
            assert power(2.,4) == 16.

    #This should simply simply be
    #unittest.main()
    #to run the unittest but bug in this notebook needs arguments below
    unittest.main(argv=["first-arg-is-ignored"], exit=False)
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
```

We can import any python code in the same directory as your script by using import. For example, if you had a file called somepythoncode.py then you could open a new script (or python/ipython instance) and call `import somepythoncode`. All the functions/classes in the script somepythoncode are then available to use. BUT, all code not inside a function/class will run when you import. To avoid this, we can check if the automatically defined variable __name__ has the value `"__main__"`. If you run number.py using `python number.py` then **name** will be **main** and the tests will run. If we import numbers in a different script then **name** will not be "**main**" (here equal to `"number"`) and the code in the if statement will not run.

All we've done in this example is move the command `unittest.main()`, which runs our tests, inside an if statement. This then allows us to run tests on our module with `python number` and use the functions in our module (without running tests) with `import number`.

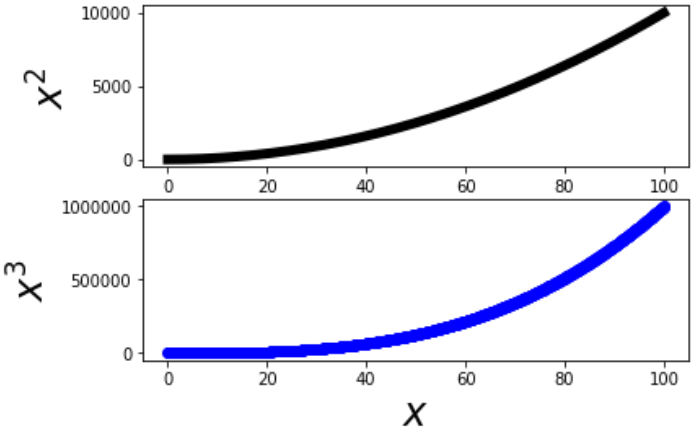
Save this file above as number.py and any instance of ipython in the same directory or a script will be able to import this using:

In [5]:
```
import number as num
print(num.square(2.), num.cube(3.))
print(num.square(4.) == num.power(4.,2))
```

```
(4.0, 27.0)
True
```

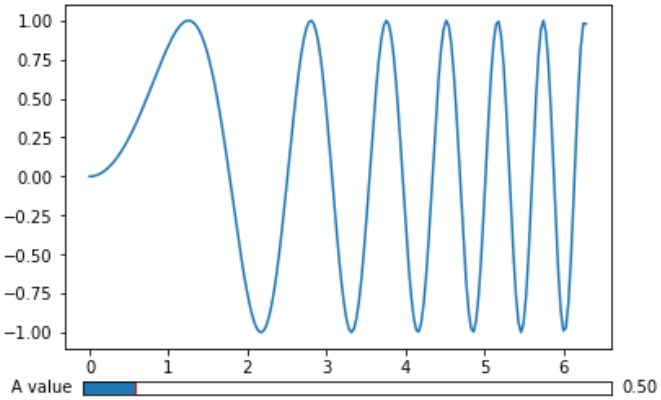
Create x=np.linspace(0., 100.,1000) and plot x2 and x3 using functions from the number module on seperate axes using plt.subplots(2,1). Label the x and y axis, change line colour, markers and width.

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt
import number as nb
x = np.linspace(0, 100., 1000)
x2 = nb.square(x)
x3 = nb.cube(x)
fig, ax = plt.subplots(2,1)
ax[0].plot(x, x2, lw=6., c='k')
ax[1].plot(x, x3, '-bo')
ax[1].set_xlabel("$x$", fontsize=24)
ax[0].set_ylabel("$x^2$", fontsize=24)
ax[1].set_ylabel("$x^3$", fontsize=24)
plt.show()
```



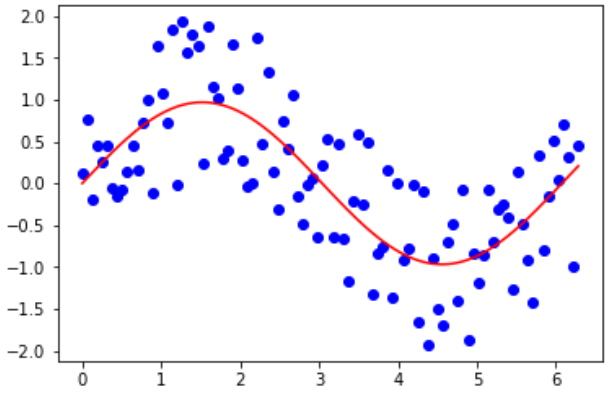
Run the slider example and adapt to plot sin(Ax^2) using function from number, num.square, with the value of A specified by the slider value.

In [15]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.widgets as mw
import number as num

#Setup initial plot of sine function
x = np.linspace(0, 2*np.pi, 200)
l, = plt.plot(x, np.sin(num.square(x)))
#Adjust figure to make room for slider
plt.subplots_adjust(bottom=0.15)
axslide = plt.axes([0.15, 0.05, 0.75, 0.03])
#Define function
def update(A):
    l.set_ydata(np.sin(A*num.square(x)))
    plt.draw()
#Bind update function to change in slider
s = mw.Slider(axslide, 'A value', 0., 5.)
s.on_changed(update)
plt.show()
```



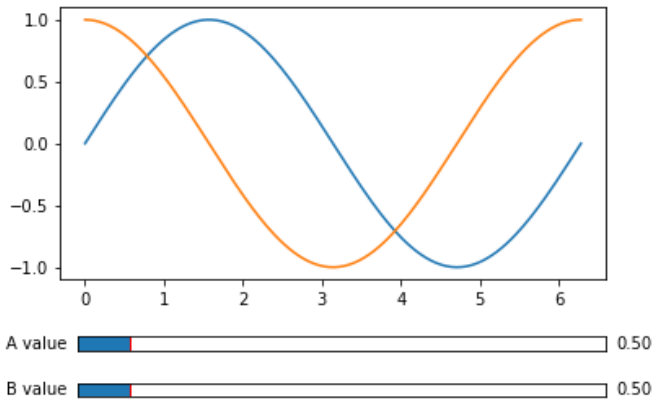
Fit an appropriate line to x = np.linspace(0, 2*np.pi, 100) y = y = np.sin(x) + (2.(np.random.random(100)-.5))

In [7]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

x = np.linspace(0, 2*np.pi, 100)
y = y = np.sin(x) + (2.*(np.random.random(100)-.5))
plt.plot(x, y, 'ob')

def sin(x, A, B):
    "Define line function"
    return A*np.sin(B*x)

params, cov = curve_fit(sin, x, y)
yf = sin(x, params[0], params[1])
plt.plot(x, yf, 'r-')

plt.show()
```



Develop a slider example with both sine and cosine on the plot updated by slider. Adapt this to add a new slider for a second coefficient B for cos(Bx).

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.widgets as mw
import number as num

#Setup initial plot of sine function
x = np.linspace(0, 2*np.pi, 200)
lsin, = plt.plot(x, np.sin(x))
lcos, = plt.plot(x, np.cos(x))

#Adjust figure to make room for slider
plt.subplots_adjust(bottom=0.3)
axslideA = plt.axes([0.15, 0.15, 0.75, 0.03])
axslideB = plt.axes([0.15, 0.05, 0.75, 0.03])

#Define function
def updateA(A):
    lsin.set_ydata(np.sin(A*x))
    plt.draw()

def updateB(B):
    lcos.set_ydata(np.cos(B*x))
    plt.draw()

#Bind update function to change in slider
sA = mw.Slider(axslideA, 'A value', 0., 5.)
sA.on_changed(updateA)
sB = mw.Slider(axslideB, 'B value', 0., 5.)
sB.on_changed(updateB)

plt.show()
```



Session 2

1) Create a Number class with a constructor to take in a variable a and store it as self.a, then add methods to square, cube and halve self.a.

In [22]:
```
import unittest
class Number():
    def __init__(self, a):
        self.a = a
    def square(self):
        return self.a**2
    def cube(self):
        return self.a**3
    def half(self):
        return self.a/2.
```

2) Create four different instances of Number using an integer, float, list and numpy array. Try calling square, cube and half, what do you notice about duck typing?

In [25]:
```
import numpy as np

import unittest
class Number():
    def __init__(self, a):
        self.a = a
    def square(self):
        return self.a**2
    def cube(self):
        return self.a**3
    def half(self):
        return self.a/2.

i = Number(4)
a = Number(4.)
l = Number([4.,6.])
n = Number(np.array([4.,6.]))

#We can loop over all instances and do the same thing
#because they are the same type and so they have the same
#interface. We get an error as we can't use a mathematical
#operation on a list.
for numinst in [i, a, n, l]:
    print(numinst.square(), numinst.cube(), numinst.half())
```

```
(16, 64, 2.0)
(16.0, 64.0, 2.0)
(array([ 16.,  36.]), array([  64.,  216.]), array([ 2.,  3.]))

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-c3f675f48f4a> in <module>()
     22 #operation on a list
     23 for numinst in [i, a, n, l]:
---> 24     print(numinst.square(), numinst.cube(), numinst.half())

<ipython-input-25-c3f675f48f4a> in square(self)
      6         self.a = a
      7     def square(self):
----> 8         return self.a**2
      9     def cube(self):
     10         return self.a**3

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

Duck typing lets us use the same class and methods for an integer, a float and a numpy array. This is the real strength of duck typing, we only have to write the code once. But it also lets us instantiate a class with a list and only causes an error when we try to use a method which couldn't work. This has the potential to create hard to debug errors.

3) In python, use pickle to save a list l=[1,2,3] then use a script to load and print

In [33]:
```
import pickle
l = [1, 2, 3]
pickle.dump(l, open("out.p",'w'))

#Load and print
m = pickle.load(open("out.p",'r'))
print(m)
```

```
[1, 2, 3]
```

4) Inherit str to create a new class MyString(str) and add a new method first_letter which returns the first letter. Create an instance of MyString and get the first letter

In [48]:
```
class MyString(str):
    def first_letter(self):
        return self[0]

s = MyString("Somestring")
print(s.first_letter())

#Note that string we inhereted from cannot do this
t = str("Somestring")
print(t.first_letter())
```

```
S

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-48-dca9925c5e85> in <module>()
      8 #Note that string we inhereted from cannot do this
      9 t = str("Somestring")
---> 10 print(t.first_letter())

AttributeError: 'str' object has no attribute 'first_letter'
```

5) Use subprocess to run the commad 'echo hello' ten times

In [47]:
```
import subprocess as sp
for i in range(10):
    sp.Popen("echo hello", shell=True)
```

hello

hello

hello

hello

hello

hello

hello

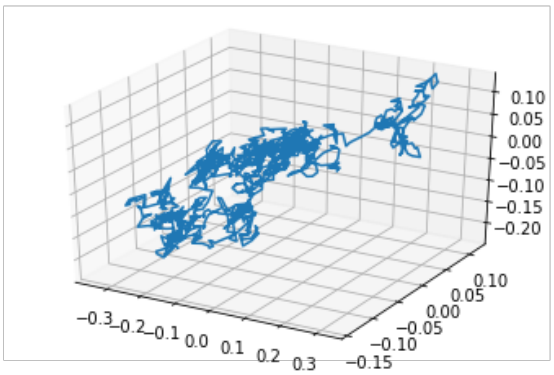
hello

hello

hello

6) Plot a 3D random walk using matplotlib with x=np.cumsum(np.random.randn(1000)*0.01) with similar for y and z.

In [43]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x=np.cumsum(np.random.randn(1000)*0.01)
y=np.cumsum(np.random.randn(1000)*0.01)
z=np.cumsum(np.random.randn(1000)*0.01)
ax.plot(x, y, z)
plt.show()
```



Data reader exercise

7) Refactor the three functions from the first hands-on: get_files, read_header and read_data into a single class postproc (use examples if you didn't finish this).

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import glob

#This is the base class, it makes sense to include most of the
#functionality here as both column and binary are very similar
class postproc():

    def read_header(headername):
        """
        Return a dictonary of
        header information
        from the header file
        """
        header = {}
        f = open(headername)
        for l in f.readlines():
            key, value = l.strip("\n").split(" ")
            try:
                header[key] = int(value)
            except ValueError:
                header[key] = float(value)

        return header

    def get_files(foldername, filename):
        """
        Return a list
        of all files in folder
        """
        files = glob.glob(foldername+filename+'*')
        files.sort()
        return files

    def read_data(filename):
        """
        Read the contents
        of each file
        """
        return np.genfromtxt(filename)
```

8) Write a constructor for the postproc class to take foldername, header and filename. Remove input arguments from get_files and get_header function and use self values in functions, include self.header=get_headers() and self.files=get_files().

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
import glob

#This is the base class, it makes sense to include most of the
#functionality here as both column and binary are very similar
class postproc():

    # Constructor for class, designed for particular folder,
    # header name and file type
    def __init__(self, foldername, headername, basename):

        self.foldername = foldername
        self.headername = headername
        self.basename = basename
        self.files = self.get_files()
        self.header = self.read_header()

    #Read header data
    def read_header(self):
        header = {}
        f = open(self.foldername + self.headername)
        for l in f.readlines():
            key, value = l.split()
            #Reading mixed int/float, try and ask forgiveness...
            try:
                header[key] = int(value)
            except ValueError:
                header[key] = float(value)
        return header


    #Read files in folder
    def get_files(self):
        files = glob.glob(self.foldername + self.basename + "*")
        files.sort()
        return files

    #Read file number
    def read_file(self, filename):
        "Could use other methods of reading here"
        return np.genfromtxt(filename)
```

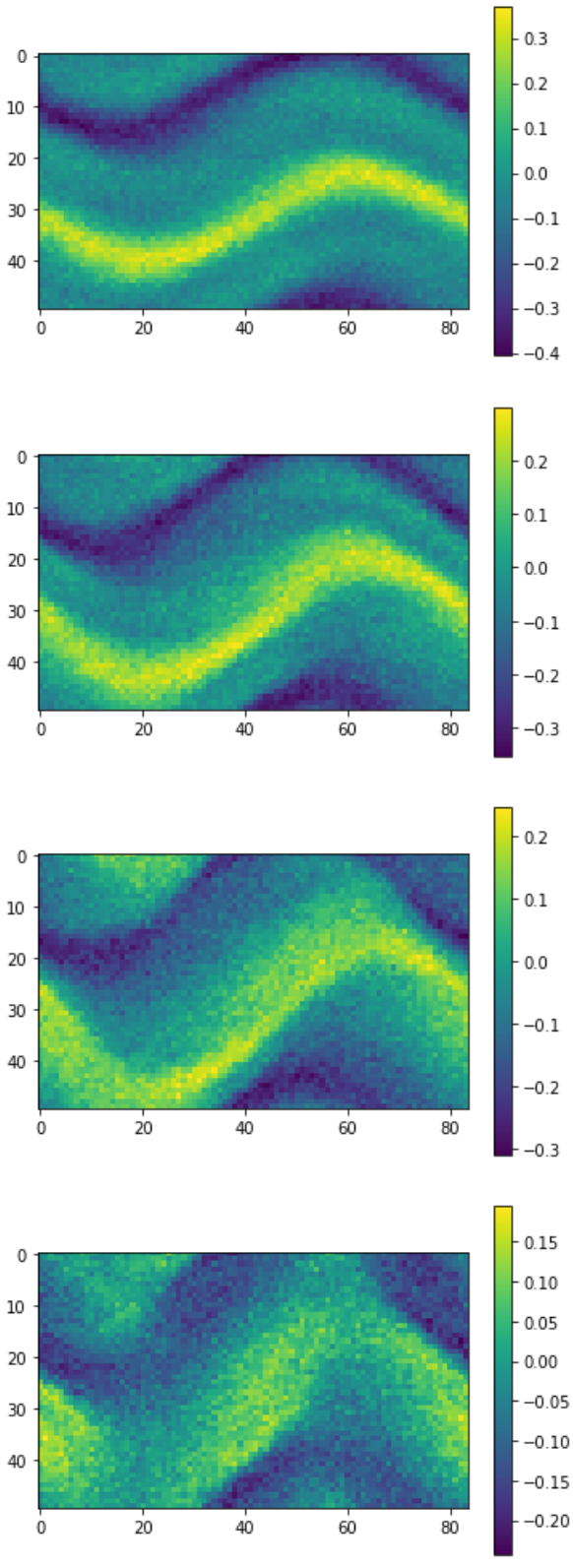
9) Make postproc a base class and using Python inherentance syntax, e.g. class postproc_binary(postproc):, create a range of different data readers.

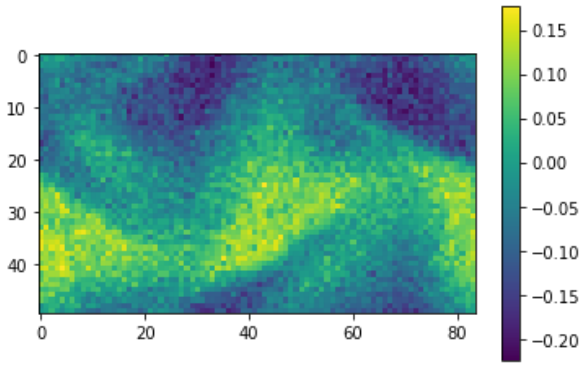
In [3]:
```
#Binary simply needs to redefine read_file
class postproc_binary(postproc):

    #Read file number
    def read_file(self, filename):
        return np.fromfile(open(filename,'rb'), dtype='d')

#Column simply needs to redefine read_file
class postproc_column(postproc):

    #Read file number
    def read_file(self, filename):
        return np.genfromtxt(filename)

#Text needs to redefine read_file and header
class postproc_text(postproc):

    #Read header data
    def read_header(self):
        f = open(self.foldername + self.headername)
        filestr = f.read()
        indx = filestr.find("FoamFile")
        header = {}
        for l in filestr[indx:].split("\n")[2:]:
            if l is "}":
                break
            key, value = l.strip(";").split()
            try:
                header[key] = int(value)
            except ValueError:
                header[key] = float(value)
        return header

    #Read file number
    def read_file(self, filename):
        f = open(filename)
        filestr = f.read()
        indx = filestr.find("internalField")
        return np.array(filestr[indx:].split("\n")[3:-3], dtype="d")
```

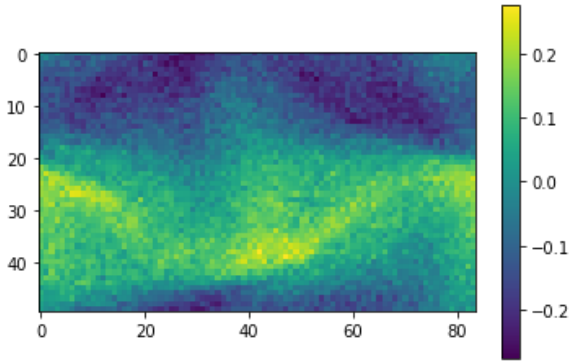
This can now be run as follows

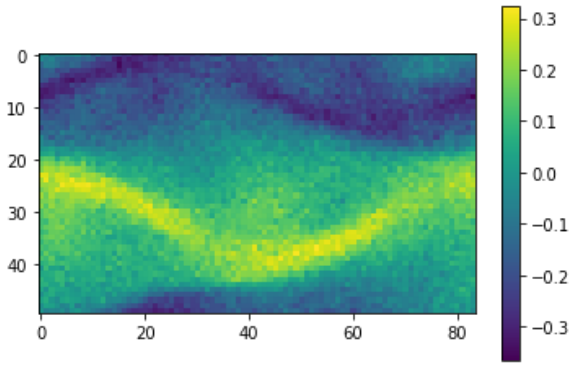
In [7]:
```
#Read Text, binary or column data based on input string
#====================
datasource = "binary"
#====================
if datasource is "text":
    pp = postproc_text('./'+datasource+'/', 'filename00000', 'filename')
elif datasource is "column":
    pp = postproc_column('./'+datasource+'/', 'header', 'filename')
elif datasource is "binary":
    pp = postproc_binary('./'+datasource+'/', 'header', 'filename')

#Interface is the same for all objects
print("Datasource is " + datasource)
for filename in pp.files:
    data = pp.read_file(filename)
    field = data.reshape(pp.header['Nx'],pp.header['Nz'])
    plt.imshow(field.T)
    plt.colorbar()
    plt.show()
```

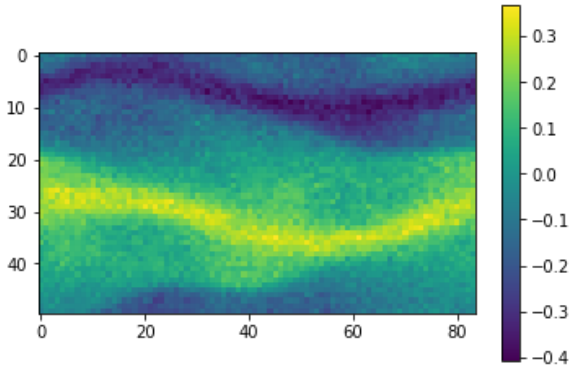
```
Datasource is binary
```

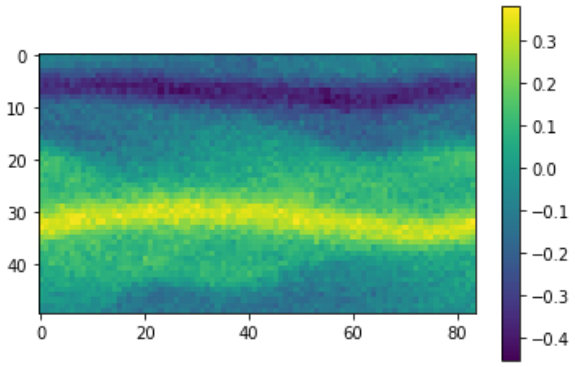


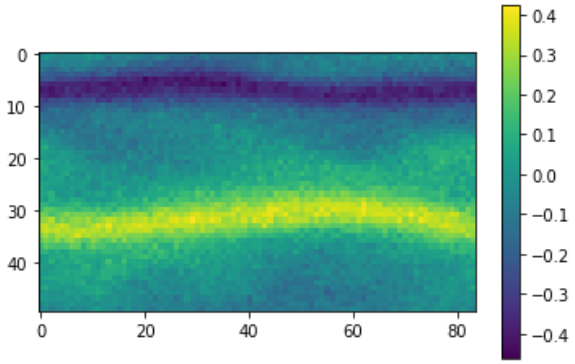

0
10
20
30
40
0
20
40
60
80
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20

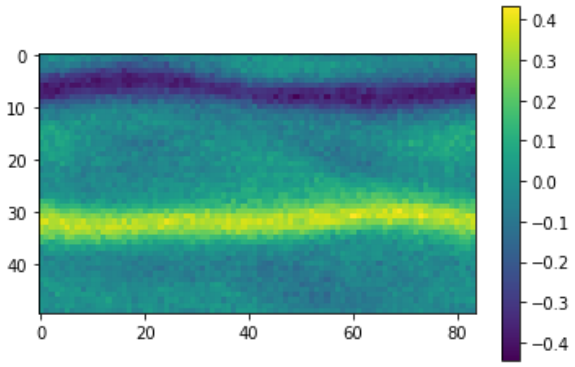

0
10
20
30
40
0
20
40
60
80
0.2
0.1
0.0
−0.1
−0.2

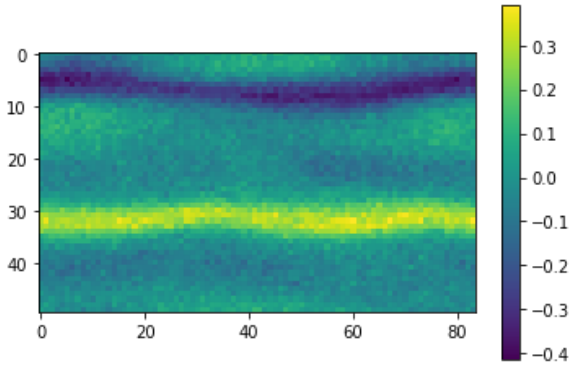

0
10
20
30
40
0
20
40
60
80
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3


0
10
20
30
40
0
20
40
60
80
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4


0
10
20
30
40
0
20
40
60
80
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4


0
10
20
30
40
0
20
40
60
80
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4


0
10
20
30
40
0
20
40
60
80
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4


0
10
20
30
40
0
20
40
60
80
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4

In [ ]: