- Just like **lists**, **arrays** can be multidimensional (**Matrix**)

```
In [38]: a = np.array([[1, 2, 3], [4, 5, 6]], float)

In [39]: a
Out[39]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [40]: a[0,0]
Out[40]: 1.0

In [41]: a[0][1]
Out[41]: 2.0

In [42]: a.shape
Out[42]: (2, 3)
```

- Just like **lists**, **arrays** can be multidimensional (**Matrix**)

```
In [38]: a = np.array([[1, 2, 3], [4, 5, 6]], float)

In [39]: a
Out[39]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [40]: a[0,0]
Out[40]: 1.0

In [41]: a[0][1]
Out[41]: 2.0

In [42]: a.shape
Out[42]: (2, 3)
```

- Just like **lists**, **arrays** can be multidimensional (**Matrix**)

  - **Arrays** can be **reshaped**

```
In [56]: a = np.array(range(10), dtype=np.uint8)

In [57]: a
Out[57]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)

In [58]: a.reshape((5,2))
Out[58]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]], dtype=uint8)

In [59]: a
Out[59]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

- Just like **lists**, **arrays** can be multidimensional (**Matrix**)

- **Arrays** can be **reshaped**

```
In [56]: a = np.array(range(10), dtype=np.uint8)

In [57]: a
Out[57]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)

In [58]: a.reshape((5,2))
Out[58]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]], dtype=uint8)

In [59]: a
Out[59]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

➤ **Plain assignment** creates a **view**, **copies** need to be **explicit**

```
In [106]: a = np.array([1, -2, 3], dtype=np.int16)

In [107]: pointer=a

In [108]: view=a.view(np.uint16)

In [109]: copy=a.copy()

In [110]: a[0]=99

In [111]: a
Out[111]: array([99, -2,  3], dtype=int16)

In [112]: pointer
Out[112]: array([99, -2,  3], dtype=int16)

In [113]: view
Out[113]: array([   99, 65534,     3], dtype=uint16)

In [114]: copy
Out[114]: array([ 1, -2,  3], dtype=int16)
```

- You can **fill** an array with a **single value**

```
In [116]: a = np.array([1, 2, 3],float)

In [117]: a
Out[117]: array([1., 2., 3.])

In [118]: a.fill(99)

In [119]: a
Out[119]: array([99., 99., 99.])
```

- You can **fill** an array with a **single value**

```
In [116]: a = np.array([1, 2, 3],float)

In [117]: a
Out[117]: array([1., 2., 3.])

In [118]: a.fill(99)

In [119]: a
Out[119]: array([99., 99., 99.])
```

- Arrays can be **transposed easily**

```
In [121]: a = np.array(range(6),float).reshape((2, 3))

In [122]: a
Out[122]:
array([[0., 1., 2.],
       [3., 4., 5.]])

In [123]: a.transpose()
Out[123]:
array([[0., 3.],
       [1., 4.],
       [2., 5.]])
```

- You can **fill** an array with a **single value**
- Arrays can be **transposed** **easily**

```
In [121]: a = np.array(range(6),float).reshape((2, 3))

In [122]: a
Out[122]:
array([[0., 1., 2.],
       [3., 4., 5.]])

In [123]: a.transpose()
Out[123]:
array([[0., 3.],
       [1., 4.],
       [2., 5.]])
```

```
In [116]: a = np.array([1, 2, 3],float)

In [117]: a
Out[117]: array([1., 2., 3.])

In [118]: a.fill(99)

In [119]: a
Out[119]: array([99., 99., 99.])
```

- **Combining** arrays can be done through **concatenation**. **Careful**, the data is **copied**!

```
In [125]: a = np.array([1,2], float)

In [126]: b = np.array([3,4,5,6], float)

In [127]: c = np.array([7,8,9], float)

In [128]: np.concatenate((a, b, c))
Out[128]: array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

- You can **fill** an array with a **single value**
- Arrays can be **transposed easily**
- **Combining** arrays can be done through **concatenation**. **Careful**, the data is **copied**!

```
In [125]: a = np.array([1,2], float)

In [126]: b = np.array([3,4,5,6], float)

In [127]: c = np.array([7,8,9], float)

In [128]: np.concatenate((a, b, c))
Out[128]: array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

- **Multidimensional arrays** can be **concatenated** along a specific **axis**:

```
In [130]: a = np.array([[1, 2], [3, 4]], float)

In [131]: b = np.array([[5, 6], [7, 8]], float)

In [132]: np.concatenate((a,b),axis=0)
Out[132]:
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])

In [133]: np.concatenate((a,b),axis=1)
Out[133]:
array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])
```

➤ Some basic **array** definitions

```
In [142]: np.arange(5, dtype=float)
Out[142]: array([0., 1., 2., 3., 4.])

In [143]: np.linspace(30,40,5)
Out[143]: array([30. , 32.5, 35. , 37.5, 40. ])

In [144]: np.ones((2,3), dtype=float)
Out[144]:
array([[1., 1., 1.],
       [1., 1., 1.]])

In [145]: np.zeros(7, dtype=int)
Out[145]: array([0, 0, 0, 0, 0, 0, 0])

In [146]: a = np.array([[1, 2, 3], [4, 5, 6]], float)

In [147]: np.zeros_like(a)
Out[147]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

➤ Some basic array Algebra

```
In [149]: a = np.array([1,2,3], float)

In [150]: b = np.array([5,2,6], float)

In [151]: a+b
Out[151]: array([6., 4., 9.])

In [152]: a*b
Out[152]: array([ 5.,  4., 18.])

In [153]: np.dot(a,b)
Out[153]: 27.0

In [154]: b**a
Out[154]: array([  5.,    4., 216.])
```

- Watch out for automatic **shape extension** or **broadcasting**

```
In [156]: a = np.array([[1, 2], [3, 4], [5, 6]], float)

In [157]: b = np.array([-1, 3], float)

In [158]: a
Out[158]:
array([[1., 2.],
       [3., 4.],
       [5., 6.]])

In [159]: b
Out[159]: array([-1.,  3.])

In [160]: a+b
Out[160]:
array([[0., 5.],
       [2., 7.],
       [4., 9.]])
```

➤ Watch out for automatic **shape extension** or **broadcasting**

```
In [156]: a = np.array([[1, 2], [3, 4], [5, 6]], float)

In [157]: b = np.array([-1, 3], float)

In [158]: a
Out[158]:
array([[1., 2.],
       [3., 4.],
       [5., 6.]])

In [159]: b
Out[159]: array([-1.,  3.])

In [160]: a+b
Out[160]:
array([[0., 5.],
       [2., 7.],
       [4., 9.]])
```

- Watch out for automatic **shape extension** or **broadcasting**

  - You can control **shape extension** with **newaxis**

$$a \quad + \quad b \quad = \quad a+b$$

```
[1., 2.]      [-1.,  3.]     [0., 5.]
[3., 4.]      [-1.,  3.]     [2., 7.]
[5., 6.]      [-1.,  3.]     [4., 9.]
```

- Watch out for automatic **shape extension** or **broadcasting**
- You can control **shape extension** with **newaxis**

```
In [174]: a = np.zeros((2,2), float)

In [175]: b = np.array([-1., 3.], float)

In [176]: a + b
Out[176]:
array([[-1.,  3.],
       [-1.,  3.]])

In [177]: a + b[np.newaxis,:]
Out[177]:
array([[-1.,  3.],
       [-1.,  3.]])

In [178]: a + b[:,np.newaxis]
Out[178]:
array([[-1., -1.],
       [ 3.,  3.]])
```

- **NumPy** offers a large library of common **mathematical functions** that can be applied **elementwise** to arrays

a = np.array([2, 1, 9], float)

a.sum() -> 12.0

a.mean() -> 4.0

a.prod() -> 18.0

a.std() -> 3.55902608

a.var() -> 12.66666666

- **NumPy** offers a large library of common **mathematical functions** that can be applied **elementwise** to arrays

```
In [183]: a=np.linspace(0.3,0.6,4)

In [184]: a
Out[184]: array([0.3, 0.4, 0.5, 0.6])

In [185]: np.sin(a)
Out[185]: array([0.29552021, 0.38941834, 0.47942554,
0.56464247])
```

- Axis can be selected for **marginal statistic**:

```
In [187]: a = np.array([[0, 2], [3, -1], [3, 5]], float)

In [188]: a.mean(axis=0)
Out[188]: array([2., 2.])

In [189]: a.mean(axis=1)
Out[189]: array([1., 1., 4.])

In [190]: a.max(axis=0)
Out[190]: array([3., 5.])

In [191]: a>=2
Out[191]:
array([[False,  True],
       [ True, False],
       [ True,  True]])
```

- many built-in routines for **linear algebra** are in the **linalg** submodule:

```
In [193]: a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]],
float)

In [194]: a
Out[194]:
array([[4., 2., 0.],
       [9., 3., 7.],
       [1., 2., 1.]])

In [195]: np.linalg.det(a)
Out[195]: -48.00000000000003

In [196]: vals, vecs = np.linalg.eig(a)

In [197]: vals
Out[197]: array([ 8.85591316,  1.9391628 , -2.79507597])

In [198]: vecs
Out[198]:
array([[-0.3663565 , -0.54736745,  0.25928158],
       [-0.88949768,  0.5640176 , -0.88091903],
       [-0.27308752,  0.61828231,  0.39592263]])
```

- many built-in routines for **linear algebra** are in the **linalg** submodule:

```
In [193]: a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]],
float)

In [194]: a
Out[194]:
array([[4., 2., 0.],
       [9., 3., 7.],
       [1., 2., 1.]])

In [195]: np.linalg.det(a)
Out[195]: -48.00000000000003

In [196]: vals, vecs = np.linalg.eig(a)

In [197]: vals
Out[197]: array([ 8.85591316,  1.9391628 , -2.79507597])

In [198]: vecs
Out[198]:
array([[-0.3663565 , -0.54736745,  0.25928158],
       [-0.88949768,  0.5640176 , -0.88091903],
       [-0.27308752,  0.61828231,  0.39592263]])
```

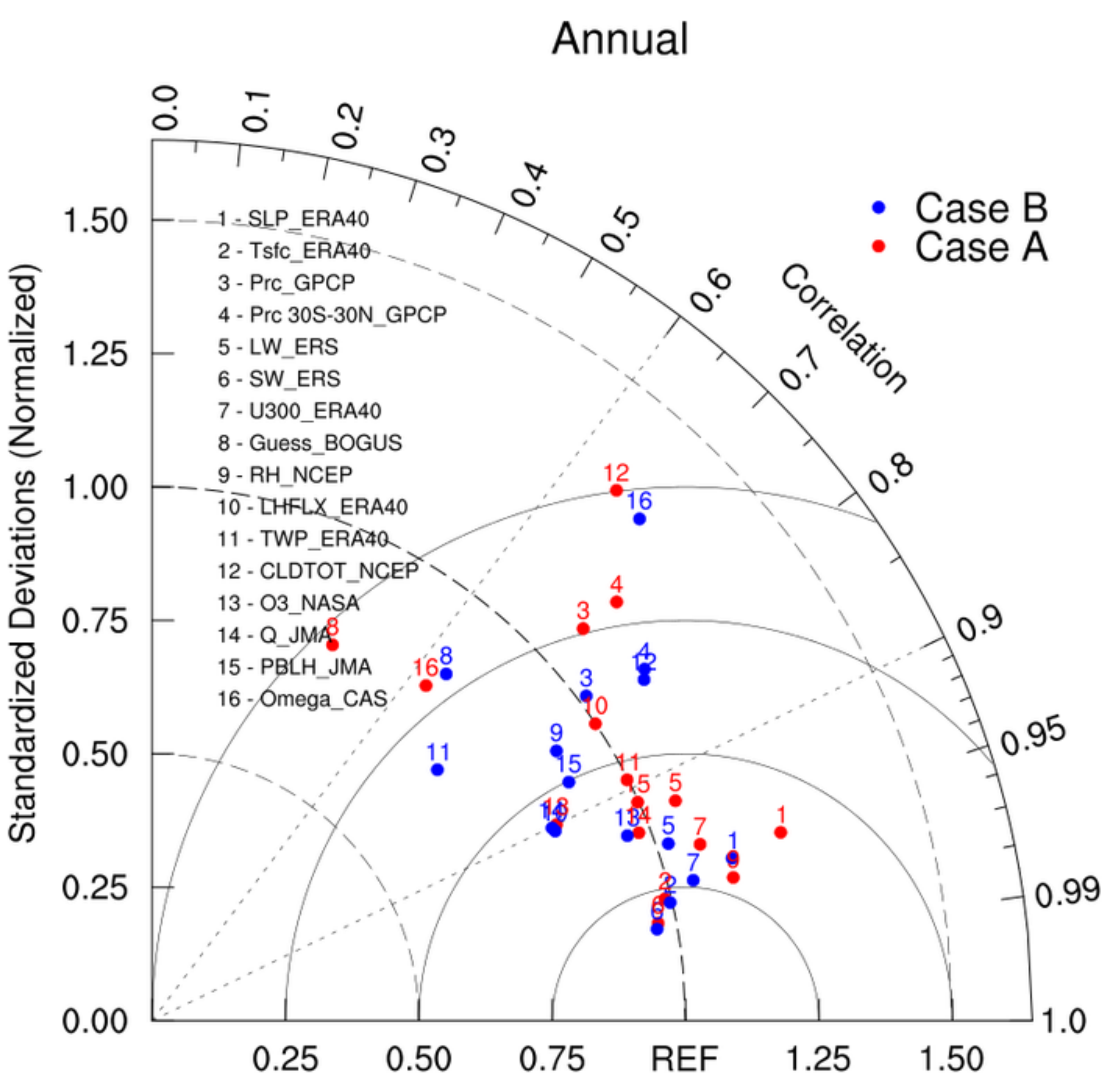
- many built-in routines for **linear algebra** are in the **linalg** submodule:

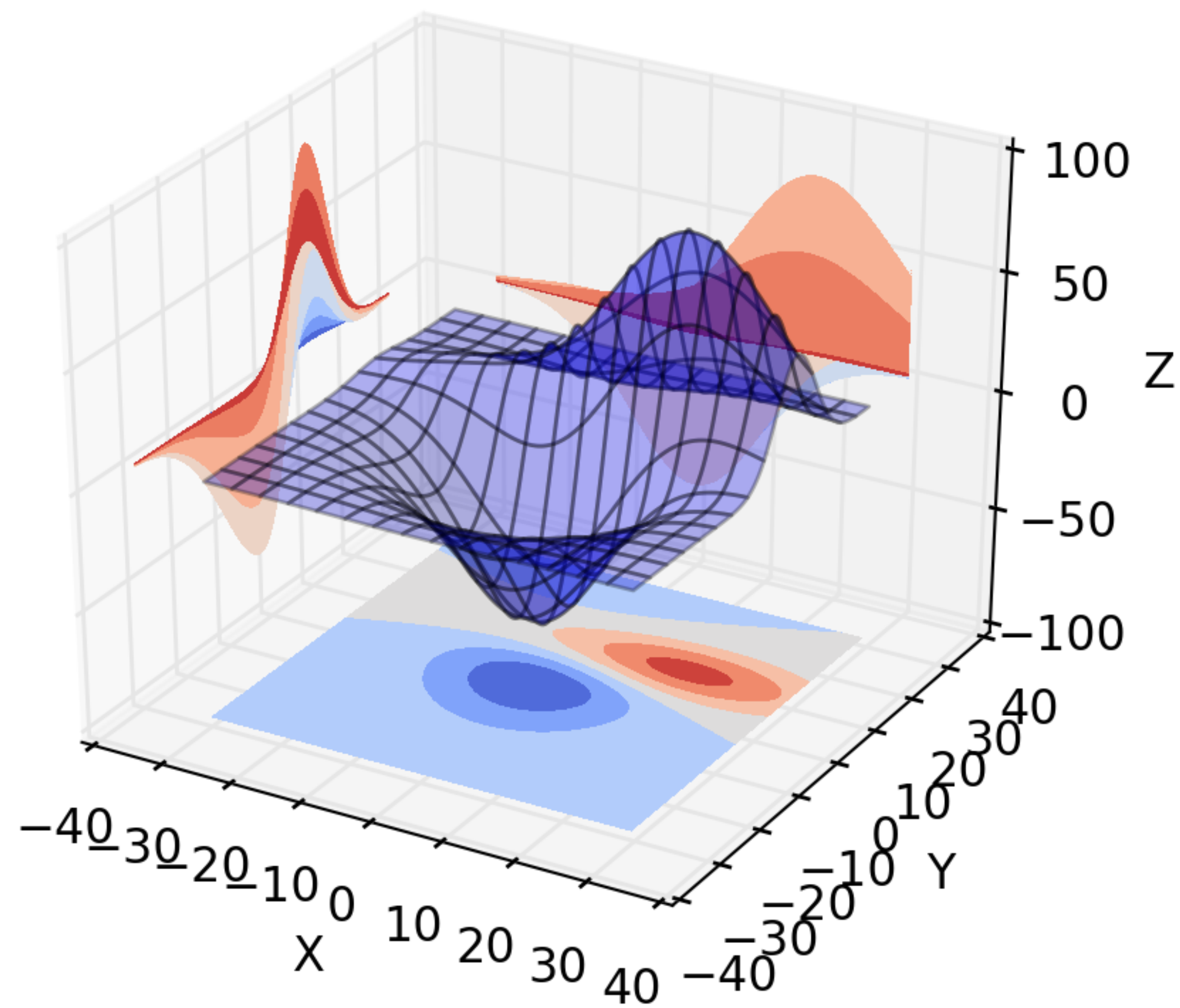
  - Singular Value Decomposition

```
In [200]: a = np.array([[1, 3, 4], [5, 2, 3]], float)

In [201]: U, s, Vh = np.linalg.svd(a)

In [202]: U
Out[202]:
array([[-0.6113829 , -0.79133492],
       [-0.79133492,  0.6113829 ]])

In [203]: s
Out[203]: array([7.46791327, 2.86884495])

In [204]: Vh
Out[204]:
array([[-0.61169129, -0.45753324, -0.64536587],
       [ 0.78971838, -0.40129005, -0.46401635],
       [-0.046676  , -0.79349205,  0.60678804]])
```

matplotlib

- **Powerful** library for **2D data plotting**, some **3D capability** Very well designed (common tasks easy, complex tasks possible).





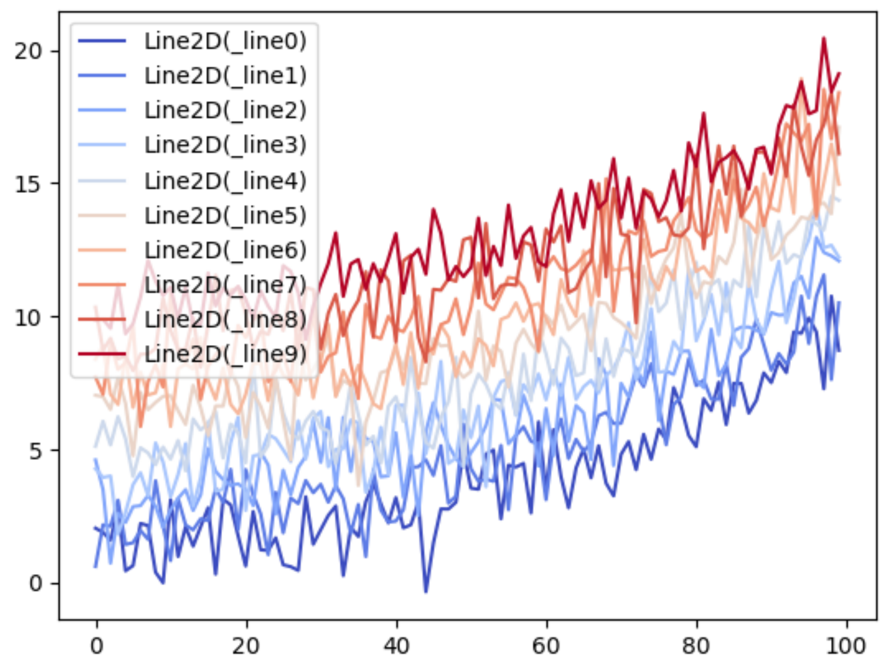
matplotlib

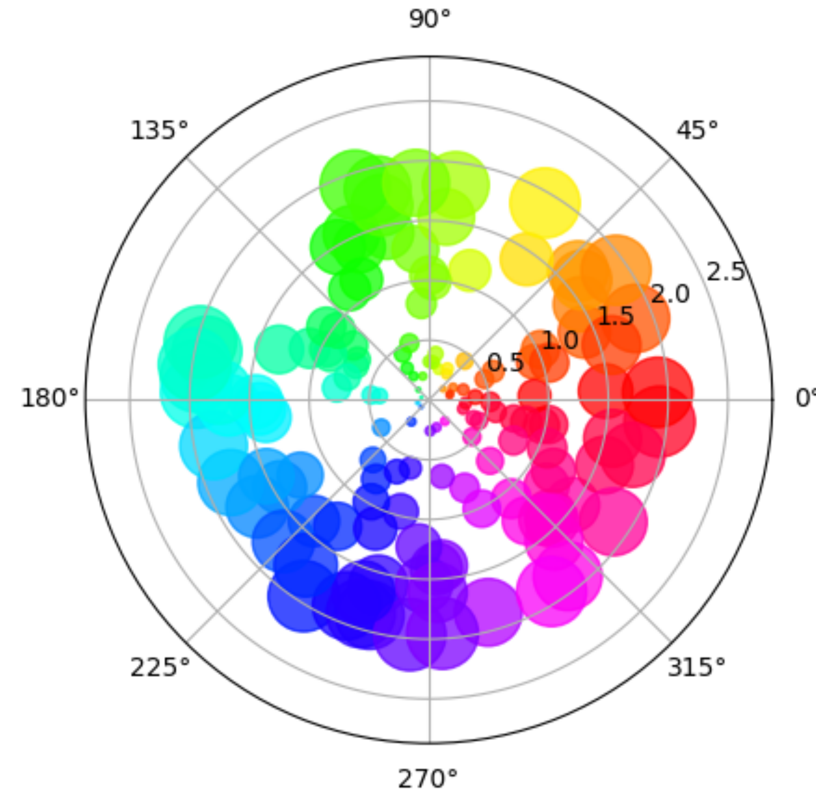
How can I make beautiful plots?

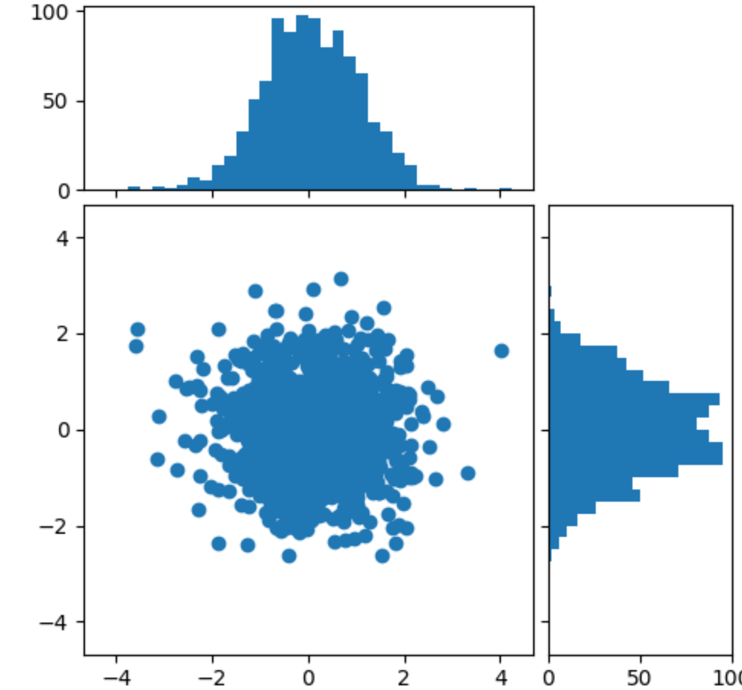
matplotlib

How can I make beautiful plots?

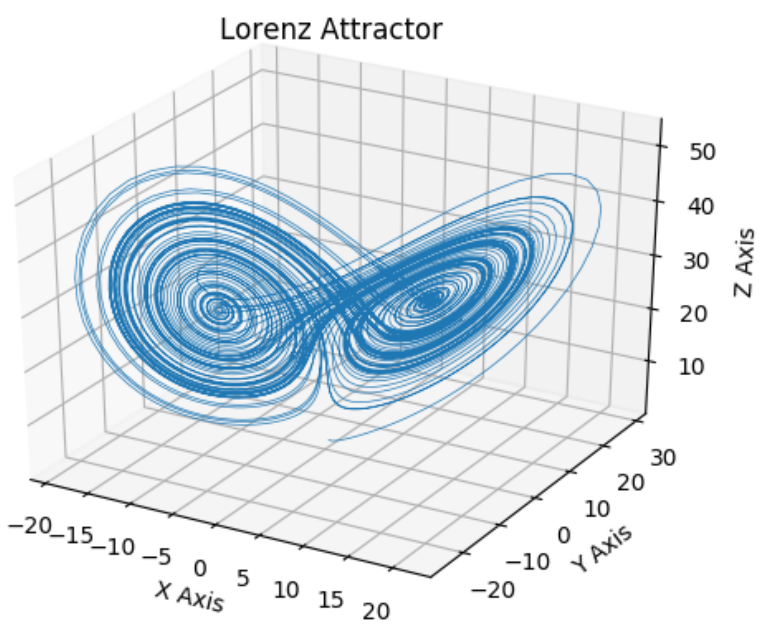
Take a look at the
Gallery!

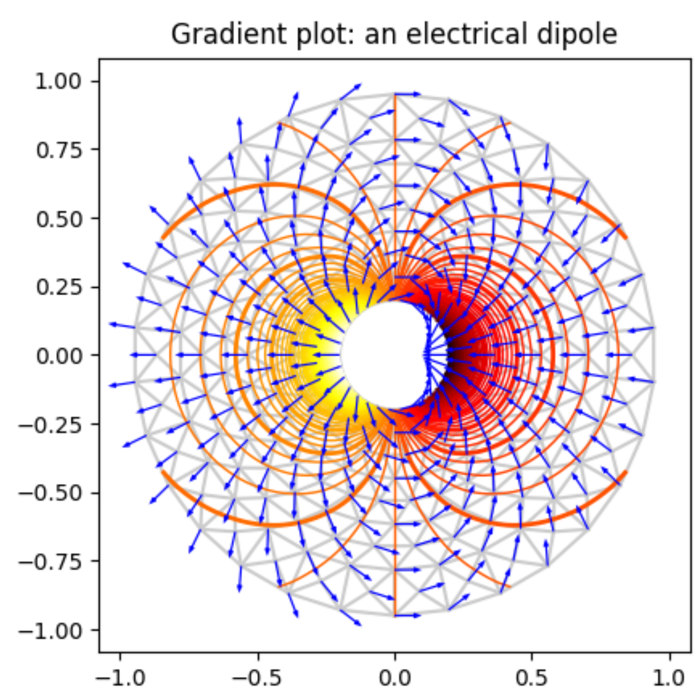

Line2D(_line0)
Line2D(_line1)
Line2D(_line2)
Line2D(_line3)
Line2D(_line4)
Line2D(_line5)
Line2D(_line6)
Line2D(_line7)
Line2D(_line8)
Line2D(_line9)
0
5
10
15
20
0
20
40
60
80
100

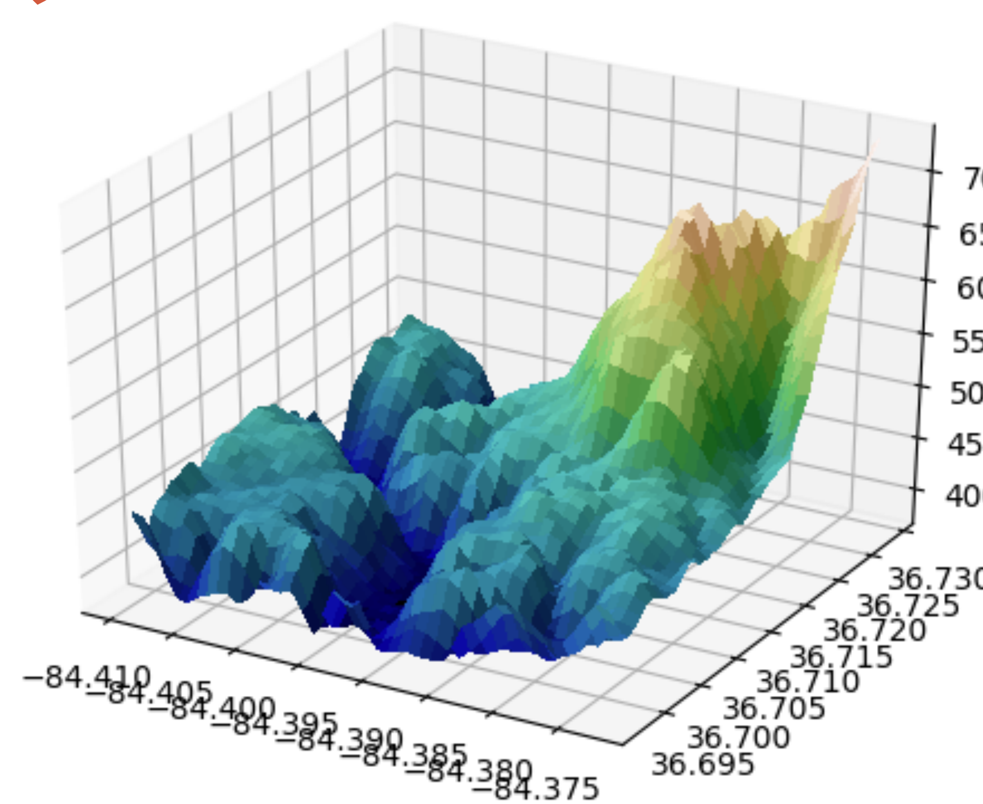

90°
135°
45°
180°
0°
225°
315°
270°
0.5
1.0
1.5
2.0
2.5


100
50
0
4
2
0
−2
−4
−4
−2
0
2
4
0
50
100

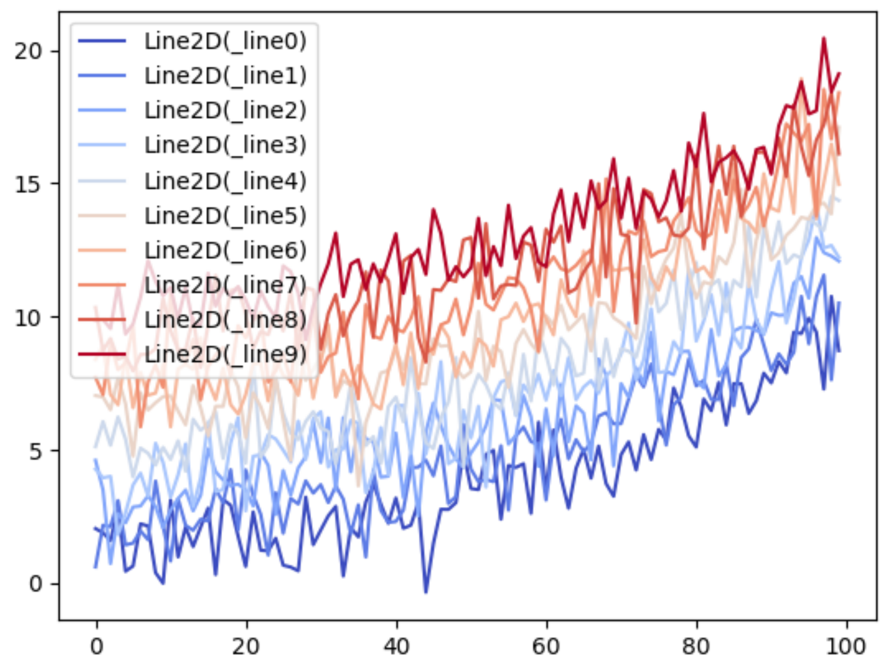
the Gallery!

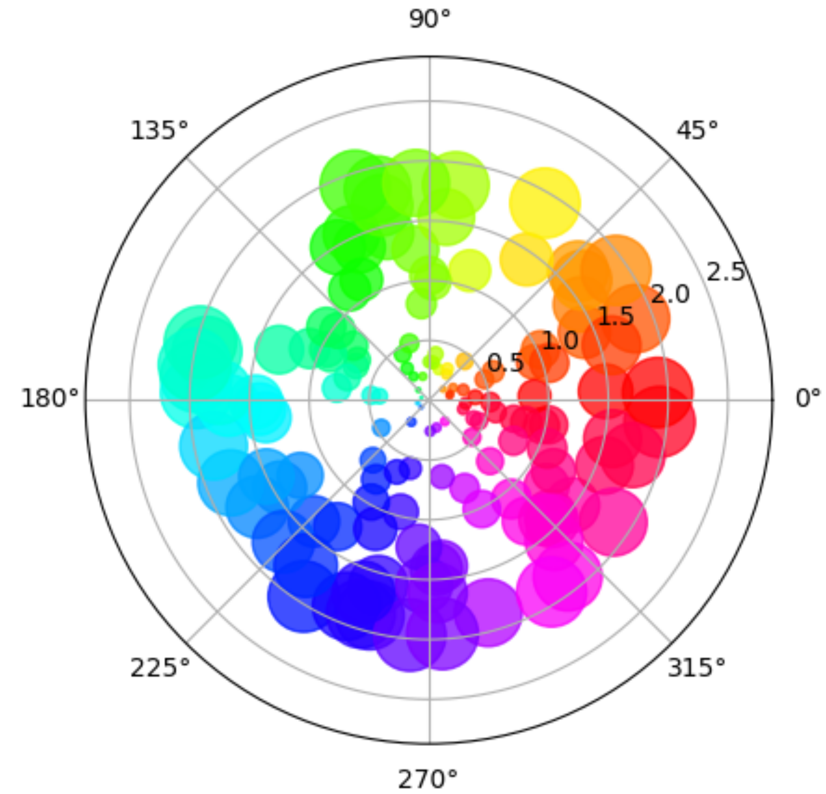

Lorenz Attractor
X Axis
Y Axis
Z Axis
10
20
30
40
50

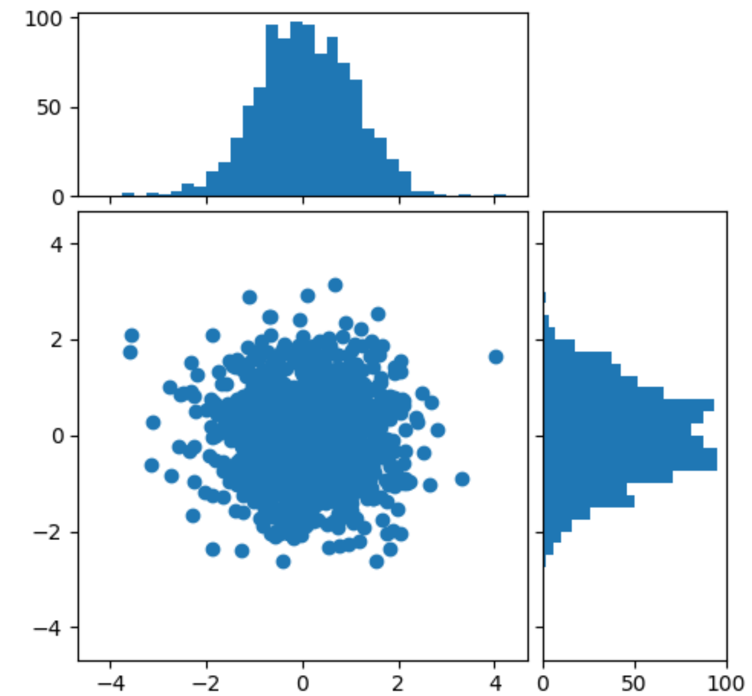

Gradient plot: an electrical dipole
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0


700
650
600
550
500
450
400
36.730
36.725
36.720
36.715
36.710
36.705
36.700
36.695
−84.410
−84.405
−84.400
−84.395
−84.390
−84.385
−84.380
−84.375

matplotlib







the Gallery!