SCIPY, NUMPY, AND MATPLOTLIB

Ali Farnudi

دانشگاه صنعتی شریف

The Abdus Salam International Centre for Theoretical Physics

- open-source add-on modules
- mathematical and numerical routines
- **precompiled**, **fast** functions

NumPy

*provides basic **routines** for **manipulating** **large arrays** and **matrices** of numeric data.*

SciPy

*extends the **functionality** of NumPy with a **substantial** **collection** of **useful algorithms.***

- cluster ➡ Vector Quantisation / Kmeans
- fftpack ➡ Discrete Fourier Transform algorithms
- integrate ➡ Integration routines
- interpolate ➡ Interpolation Tools
- io ➡ Data input and output
- lib ➡ Python wrappers to external libraries
- lib.lapack ➡ Wrappers to LAPACK library
- linalg ➡ Linear algebra routines
- misc ➡ Various utilities that don't have another home.

SciPy

- ndimage ➡ n-dimensional image package
- odr ➡ Orthogonal Distance Regression
- optimize ➡ Optimisation Tools
- signal ➡ Signal Processing Tools
- sparse ➡ Sparse Matrices
- sparse.linalg ➡ Sparse Linear Algebra
- sparse.linalg.dsolve ➡ Linear Solvers
- sparse.linalg.dsolve.umfpack ➡ Interface to the UMFPACK library: Conjugate Gradient Method (LOBPCG)

- special ➡ Airy Functions [*]
- lib.blas ➡ Wrappers to BLAS library [*]
- sparse.linalg.eigen ➡ Sparse Eigenvalue Solvers [*]
- stats ➡ Statistical Functions [*]
- spatial ➡ Spatial data structures and algorithms

NumPy

- Polynomial Mathematics
- Statistical computations
- Full suite of pseudo-random number generators and operations
- Discrete Fourier transforms,
- more complex linear algebra operations
- size / shape / type testing of arrays,
- splitting and joining arrays, histograms

- creating arrays of numbers spaced in various ways
- creating and evaluating functions on grid arrays
- treating arrays with special (NaN, Inf) values
- set operations
- creating various kinds of special matrices
- evaluating special mathematical functions (e.g. Bessel functions)
- To learn more, consult the NumPy documentation at
  http://docs.scipy.org/doc/

NumPy

- You can **import** the modules like most Python packages:

```python
import numpy

import numpy as np
```

- You can **import** the modules like most Python packages:

```
import numpy

import numpy as np
```

The **essential** and **basic** unit of NumPy, **the array**!

- Designed to be **accessed just like Python lists**
- **All elements** are of the **same type**
- Ideally suited for **storing** and **manipulating large** numbers of elements

- You can **import** the modules like most Python packages:

```
import numpy

import numpy as np
```

The **essential** and **basic** unit of NumPy, **the array**!

- Designed to be **accessed just like** Python lists
- **All elements** are of the same type
- Ideally suited for **storing** and **manipulating** large numbers of elements

```
In [27]: a=np.array([1,2,3,4,5,6,7,8],float)

In [28]: a
Out[28]: array([1., 2., 3., 4., 5., 6., 7., 8.])

In [29]: type(a)
Out[29]: numpy.ndarray

In [30]: a[:2]
Out[30]: array([1., 2.])

In [31]: a[:6:2]
Out[31]: array([1., 3., 5.])
```

- 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)
```

➤ **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**

a	$+$	b	$=$	$a+b$
[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**

$$a + b = a+b$$

a	b	$a+b$
[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]])
```

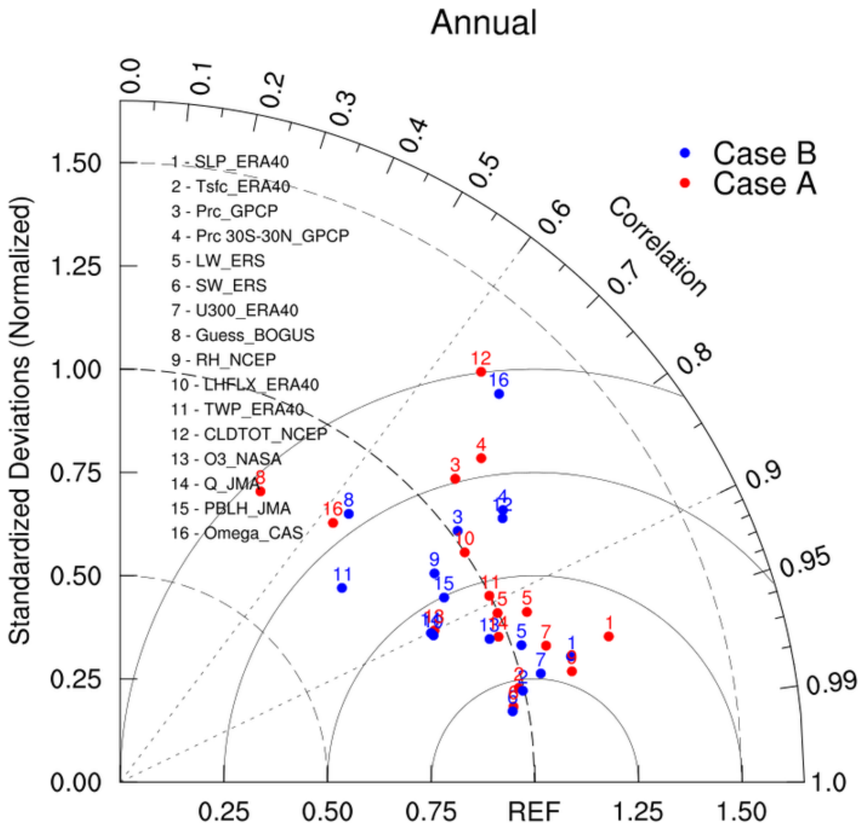
- many built-in routines for **linear algebra** are in the **linalg** submodule:

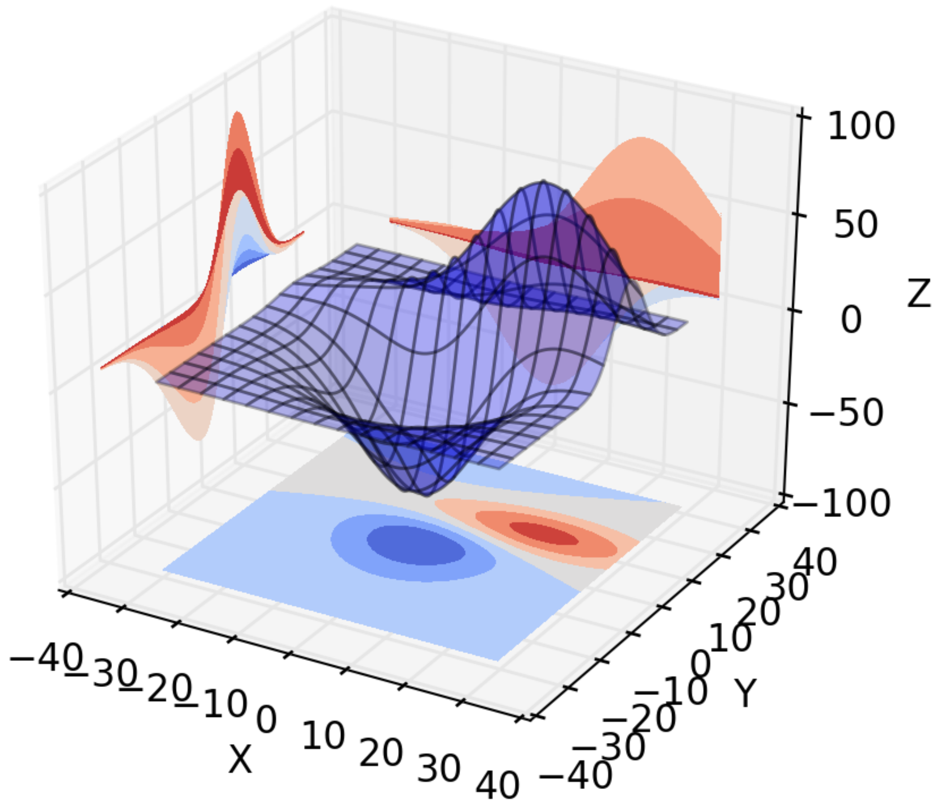
  - Singular Value Decomposition

```
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).





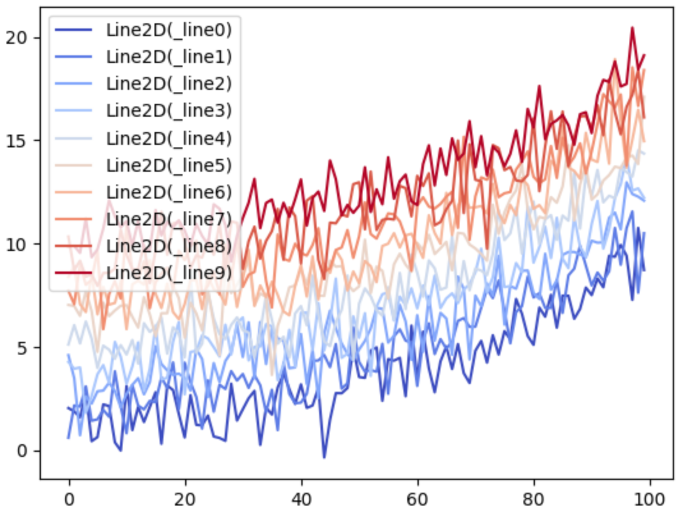
matplotlib

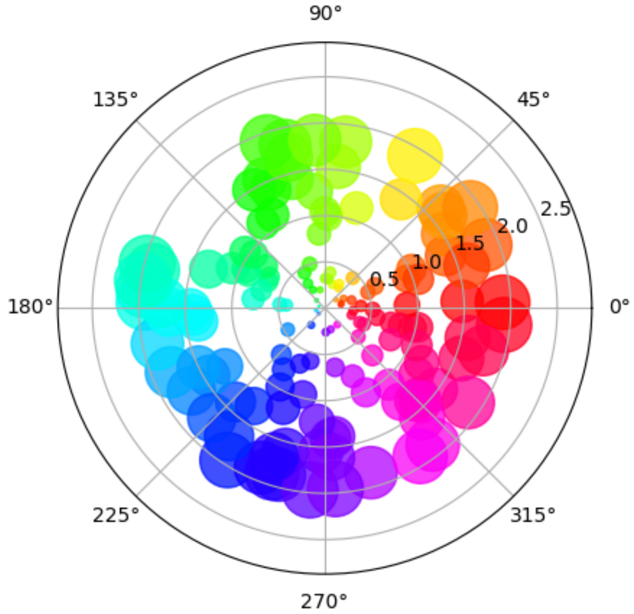
How can I make beautiful plots?

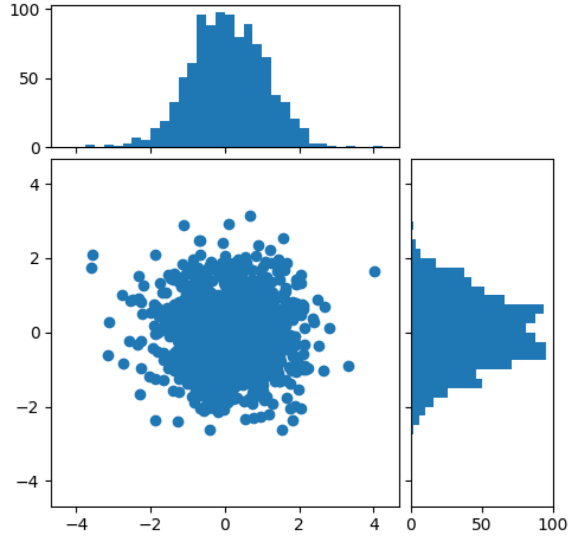
Take a look at the

Gallery!

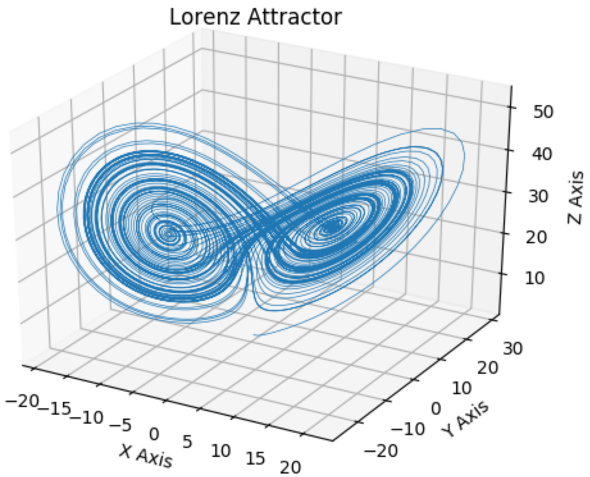
matplotlib

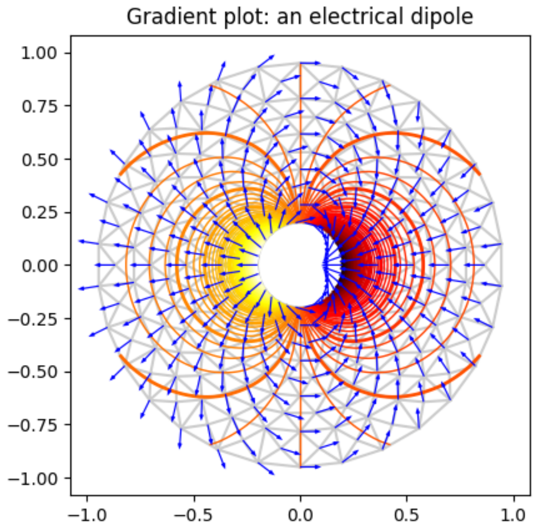


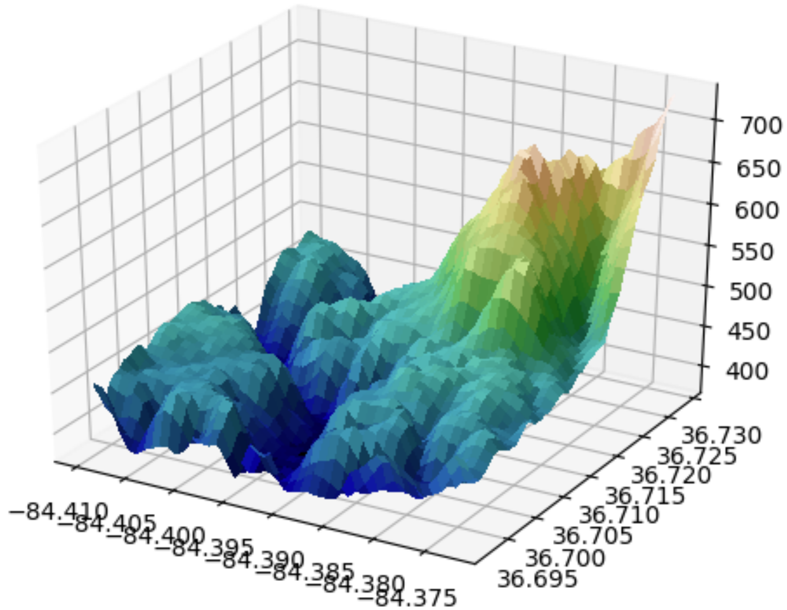


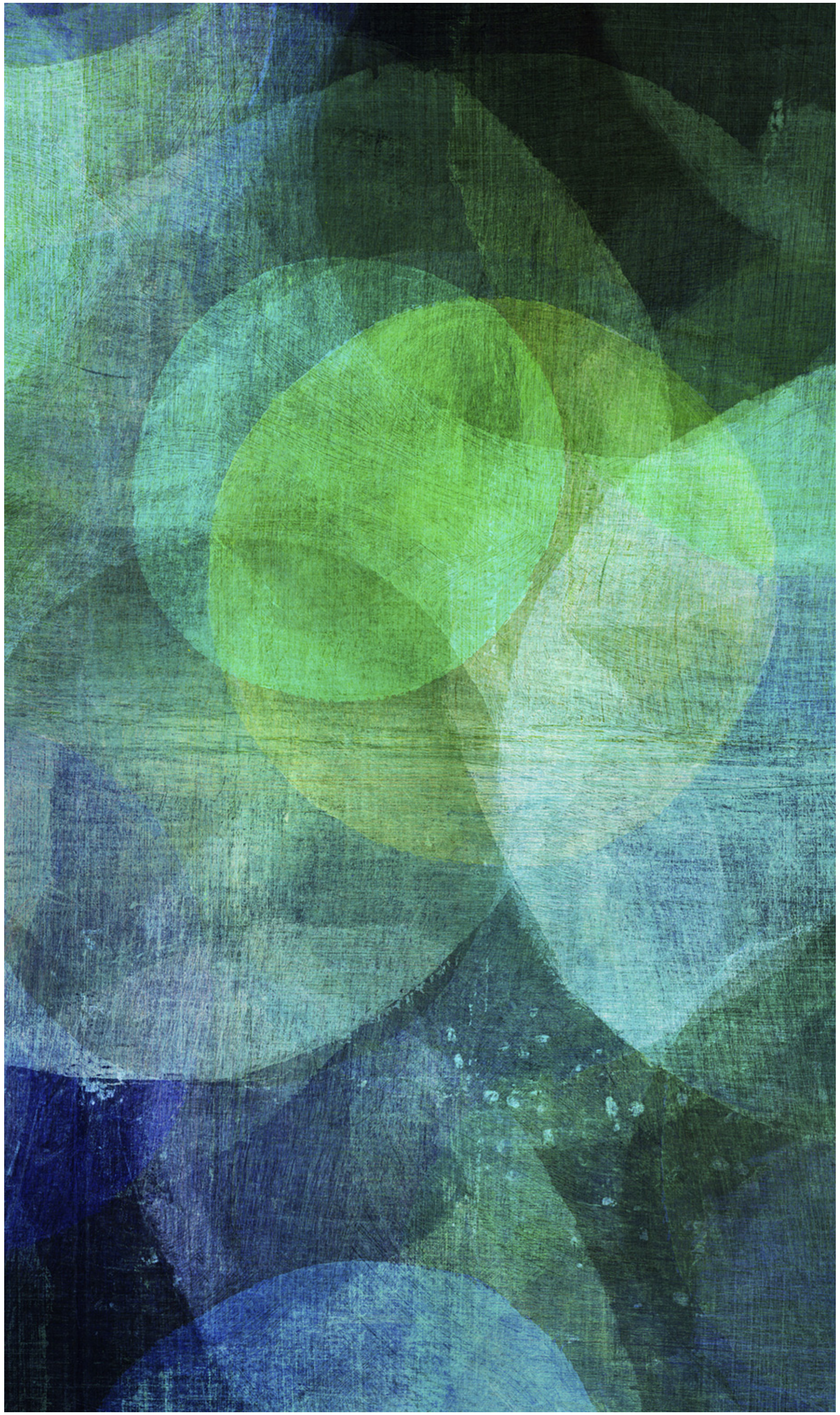


the Gallery!









EXERCISES

- particle animation
- large data memmap