Numpy

October 6, 2020

1 Scientific computing in Python with Numpy and Numba

1.1 Numpy: a brief introduction and overview of typical use cases

Website

Documentation

Illustrations shamelessly reproduced from "A Visual Intro to NumPy and Data Representation" (Jay Alammar).

1.1.1 Import and versions

[2]:
```
import numpy
import sys
print(f"Python version: {sys.version}")
print(f"Numpy version: {numpy.__version__}")
```

```
Python version: 3.8.5 (default, Jul 21 2020, 10:48:26)
[Clang 11.0.3 (clang-1103.0.32.62)]
Numpy version: 1.19.1
```

Typically, `numpy` is imported `np` to make it easier to call the package functions.

[3]:
```
import numpy as np
```

1.1.2 Arrays

The basic `numpy` structure is a *n-dimensional* array, in the sense of a dynamically allocated `C` or `Fortran` array.

Creation

[4]:
```
a = np.array([1, 2, 3, 4, 5])
print(a.shape)
print(a.dtype)
```

```
(5,)
int64
```

All `numpy` arrays have a shape and a datatype.

```
[5]: b = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
     b.dtype
```

```
[5]: dtype('float64')
```

```
[6]: c = np.array([1, 2, 3.0, 4.0, 5.0])
     print(c.dtype)
     print(type(c[0]))
```

```
float64
<class 'numpy.float64'>
```

```
[7]: c
```

```
[7]: array([1., 2., 3., 4., 5.])
```

Multi-dimensional arrays follow the same idea:

```
[8]: d = np.array([
         [1, 2, 3],
         [4, 5, 6]
     ])
     print(d.ndim)
     print(d.shape)
```

```
2
(2, 3)
```

```
[9]: e = np.array([
         [
             [1, 2, 3],
             [4, 5, 6],
         ],
         [
             [-1, -2, -3],
             [-4, -5, -6],
         ],
     ])
     print(e.ndim)
     print(e.shape)
```

```
3
(2, 2, 3)
```

Creation from a Python list should be reserved for simple, manual cases. Lots of methods exist to create typical structures.

```
[10]: f = np.zeros(20)  # or np.ones
      f
```

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

```
[11]: np.zeros((4, 4))
```

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

```
[12]: np.eye(5)
```

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

```
[13]: np.random.random(10)
```

```
[13]: array([0.65027084, 0.15339426, 0.96259359, 0.73749076, 0.74729732,
              0.88353929, 0.85353262, 0.97564368, 0.76841002, 0.98966212])
```

```
[14]: np.random.random((5, 2, 3))
```

```
[14]: array([[[0.33462069, 0.58966623, 0.66216628],
               [0.16473687, 0.45027944, 0.12955761]],

              [[0.6917416 , 0.71720602, 0.66615324],
               [0.26922009, 0.11339753, 0.44908625]],

              [[0.46675966, 0.59285679, 0.70027449],
               [0.4903823 , 0.01971502, 0.39025563]],

              [[0.57222753, 0.52576454, 0.50070404],
               [0.09045206, 0.00243409, 0.53892642]],

              [[0.82686394, 0.09741241, 0.60084521],
               [0.98814317, 0.04538618, 0.70397975]]])
```

```
[16]: # DON'T DO THAT
      a = np.ones(5)
      for i in range(0, 5):
          print(a[i])
```

```
1.0
1.0
1.0
```

```
1.0
1.0
```

Indexing Indexing is extremely powerful but can be quite complicated in some cases. However the "usual" cases are simple.

[19]:
```
e = np.array([
    [
        [1, 2, 3],
        [4, 5, 6],
    ],
    [
        [-1, -2, -3],
        [-4, -5, -6],
    ],
])
```

[20]:
```
e[0, 0, 0]
```

[20]:
```
1
```

[21]:
```
e[1, 1, 2]
```

[21]:
```
-6
```

[22]:
```
e[-1, -1, -1]
```

[22]:
```
-6
```

[25]:
```
e[0, :, :]  # Views and slices
```

[25]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[26]:
```
e[0, :, 1]
```

[26]:
```
array([2, 5])
```

[28]:
```
print(
    e[0, :, 0:1]
)
print(e[0, :, 0:1].shape)
```

```
[[1]
 [4]]
(2, 1)
```

[29]:
```
g = e[0, :, 1:]
g
```

[29]:
```
array([[2, 3],
       [5, 6]])
```

[36]:
```
g2 = np.array([[1, 2, 3, 4, 5, 6, 7], [10, 20, 30, 40, 50, 60, 70]])
g2[:, ::2]
```

[36]:
```
array([[ 1,  3,  5,  7],
       [10, 30, 50, 70]])
```

Transposition comes for free (no data copy)

[39]:
```
g.T   # copy free
```

[39]:
```
array([[2, 5],
       [3, 6]])
```

[48]:
```
# Array "physical" copy
np.copy(g2)
```

[48]:
```
array([[ 1,  2,  3,  4,  5,  6,  7],
       [10, 20, 30, 40, 50, 60, 70]])
```

Boolean indexing

[54]:
```
h = np.random.random(20)
print(h)
print(h > 0.5)

h[h > 0.5]
```

```
[0.77302896 0.11831892 0.94383515 0.02288236 0.1148379  0.27520311
 0.49742042 0.51689174 0.66428033 0.03767339 0.93999711 0.03194377
 0.24803129 0.12542603 0.50921674 0.46015099 0.04004487 0.4781803
 0.39352273 0.80824964]
[ True False  True False False False False  True  True False  True False
 False False  True False False False False  True]
```

[54]:
```
array([0.77302896, 0.94383515, 0.51689174, 0.66428033, 0.93999711,
       0.50921674, 0.80824964])
```

[57]:
```
g2.max()
```

[57]:
```
70
```

[58]:
```
np.max(g2)
```

```
[58]: 70
```

Example and performance

In ‘pure’ Python…

```
[64]: matrix = [
          [1.0, 0.0, 0.0, 0.0],
          [0.0, 1.0, 0.0, 0.0],
          [0.0, 0.0, 1.0, 0.0],
          [0.0, 0.0, 0.0, 1.0],
      ]
```

```
[66]: mean = np.zeros(4)
      mean
```

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

```
[67]: cov = 0.01 * np.eye(4)
      cov
```

```
[67]: array([[0.01, 0.  , 0.  , 0.  ],
             [0.  , 0.01, 0.  , 0.  ],
             [0.  , 0.  , 0.01, 0.  ],
             [0.  , 0.  , 0.  , 0.01]])
```

```
[71]: particles = np.random.multivariate_normal(mean, cov, int(1e6))
      print(particles.shape)
      particles
```

```
(1000000, 4)
```

```
[71]: array([[ 0.07602977,  0.02572549, -0.01628514, -0.04583431],
             [ 0.01230086,  0.07047807,  0.03786649,  0.22408412],
             [ 0.02669951, -0.16860823, -0.20860394, -0.09555031],
             …,
             [-0.07697248,  0.09769213, -0.01570948, -0.23148553],
             [ 0.01135801, -0.01295512, -0.0817861 , -0.04529333],
             [ 0.10392814, -0.09248552,  0.14674501, -0.04592692]])
```

```
[72]: particles = particles.tolist()
      type(particles)
```

```
[72]: list
```

```
[73]: %%timeit -n 1 -r 1
      output = []
      for n in range(0, len(particles)):
```

```
    tmp = [0.0, 0.0, 0.0, 0.0]
    for i in range(0, 4):
        for j in range(0, 4):
            tmp[i] += particles[n][i] * matrix[i][j]

    output.append(tmp)
print(len(output))
#output
```

```
1000000
4.91 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Using numpy

[74]:
```
matrix = np.eye(4)
matrix
```

[74]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

[75]:
```
particles = np.random.multivariate_normal(mean, cov, int(1e6))
print(particles.shape)
particles
```

```
(1000000, 4)
```

[75]:
```
array([[ 0.06656825, -0.04823821,  0.08759946, -0.17190058],
       [ 0.05711391, -0.00250155,  0.0752147 , -0.07481593],
       [-0.02898914,  0.11650021, -0.0032598 , -0.04036906],
       …,
       [ 0.07241193, -0.03315781, -0.02928451, -0.06910959],
       [-0.0799167 , -0.16015909, -0.19147755,  0.00706015],
       [ 0.08110189,  0.03850658, -0.01195113, -0.05527025]])
```

[77]:
```
particles.shape
```

[77]:
```
(1000000, 4)
```

[76]:
```
print(np.dot(matrix, particles.T).T.shape)
```

```
(1000000, 4)
```

[79]:
```
%%timeit -n 1 -r 1
np.dot(matrix, particles.T).T
```

```
6.82 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

1.2 Numba

Website

Documentation

1.2.1 Can we make it even faster ?

JIT: Just-In Time compilation

[81]:
```
import numpy as np
```

[82]:
```
from numba import njit
```

[83]:
```
@njit
def multiply(vector, matrix, destination):
    for n in range(0, len(particles)):
        for i in range(0, 4):
            for j in range(0, 4):
                destination[n, i] += vector[n, i] * matrix[i, j]
    return destination
```

[84]:
```
particles = np.random.multivariate_normal(mean, cov, int(1e6))
matrix = np.array([
    [1.0, 0.0, 0.0, 0.0],
    [0.0, 1.0, 0.0, 0.0],
    [0.0, 0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0, 1.0],
])
destination = np.zeros(particles.shape)
```

[88]:
```
%%timeit -n 1 -r 1
result = multiply(particles, matrix, destination)
result.shape
```

```
10.7 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

1.3 Particle tracking

1.3.1 Particle tracking in 10 lines:

- Track “particles”: vectors of coordinates (horizontal and vertical position and angle, momentum deviation, time lag)
- Represent a magnet (of length L) using a transfer map (in the linear case: a transfer matrix)
- Apply the transfer map (in the linear case: multiply the “vector” by the “matrix”)
- Repeat for many particles in the bunch (good sampling required, tracking many thousands to milion of particles)
- Repeat for all magnets in the ring
- Repeat for many turns

That’s a lot of matrix multiplication!

1.3.2 A quick example:

- Drift: field-free region, simple propagation
- Quadrupole: a bit more involved, see below

[91]:
```python
from numba import prange
from numpy import sqrt, cos, sin, cosh, sinh
```

[117]:
```python
@njit(parallel=True, fastmath=True)
def track_madx_drift(b1, b2, length):
    for i in prange(b1.shape[0]):
        px = b1[i, 1]
        py = b1[i, 3]
        pt = b1[i, 5]

        b2[i, 0] = b1[i, 0] + length * px
        b2[i, 1] = b1[i, 1]
        b2[i, 2] = b1[i, 2] + length * py
        b2[i, 3] = b1[i, 3]
        b2[i, 4] = b1[i, 4]
        b2[i, 5] = b1[i, 5]

    return b1, b2

@njit(parallel=True, fastmath=True)
def track_madx_quadrupole(b1, b2, length: float, k1: float, tilt: float):
    st: float
    ct: float

    if k1 == 0:
        return track_madx_drift(b1, b2, length)

    if tilt != 0.0:
        st = sin(tilt)
        ct = cos(tilt)

    for i in prange(b1.shape[0]):
        delta_plus_1 = b1[i, 4] + 1
        x = b1[i, 0]
        xp = b1[i, 1] / delta_plus_1  # This is the key point to remember
        y = b1[i, 2]
        yp = b1[i, 3] / delta_plus_1  # This is the key point to remember

        k1_ = k1 / delta_plus_1  # This is the key point to remember
        if k1_ > 0:
            kl = sqrt(k1_) * length
            sx = sin(kl) / sqrt(k1_)
            cx = cos(kl)
            sy = sinh(kl) / sqrt(k1_)
            cy = cosh(kl)
        else:
            kl = sqrt(-k1_) * length
            sx = sinh(kl) / sqrt(-k1_)
            cx = cosh(kl)
            sy = sin(kl) / sqrt(-k1_)
            cy = cos(kl)

        x_ = cx * x + sx * xp
        xp_ = (-k1_ * sx * x + cx * xp) * delta_plus_1
        y_ = cy * y + sy * yp
        yp_ = (k1_ * sy * y + cy * yp) * delta_plus_1


        b2[i, 0] = x_
        b2[i, 1] = xp_
        b2[i, 2] = y_
        b2[i, 3] = yp_
        b2[i, 4] = b1[i, 4]
        b2[i, 5] = b1[i, 5]

    return b1, b2
```

[110]:
```
b1 = np.copy(particles)
b2 = np.zeros(b1.shape)
```

[115]:
```
%%timeit -n 1 -r 1
track_madx_drift(b1, b2, 10.0)
```

```
3.81 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[116]:
```
%%timeit -n 1 -r 1
track_madx_quadrupole(b1, b2, 10.0, 200.0, 0.0)
```

```
8.88 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```