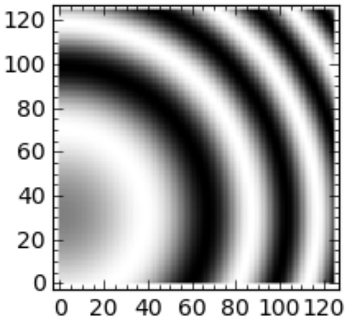
numpy & Laplace



Dodawanie wektorowe w Numpy

Mamy trzy tablice o rozmiarze *N=5*.

Chcemy je dodać,
czyli dla każdego *i<N*:

$$d_i = a_i + b_i + c_i$$

`a[:]`	11	23	32	23	12
`b[:]`	1	2	2	3	1
`c[:]`	10	20	30	20	10
`d[:]`	0	0	0	0	0

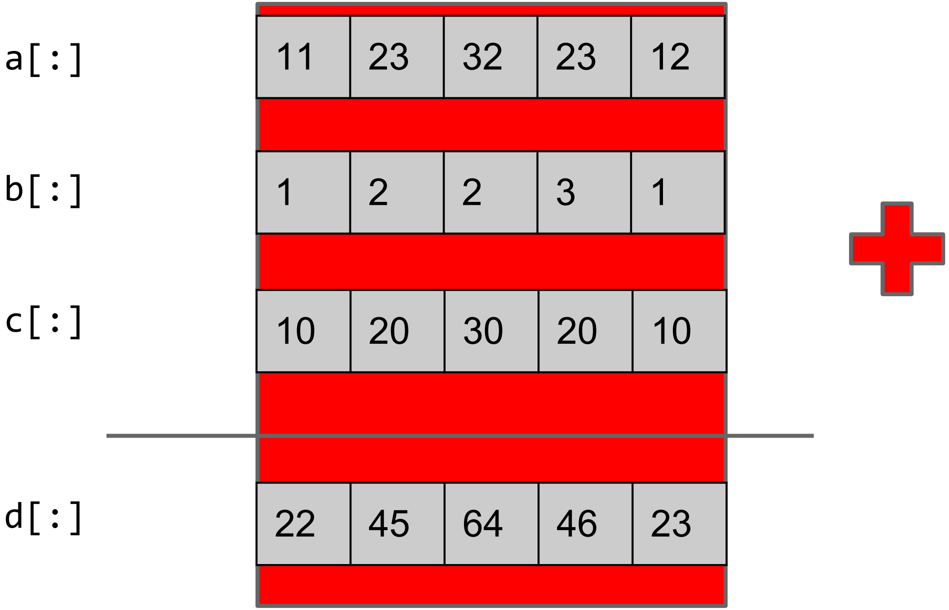
Dodawanie wektorowe w Numpy

Mamy trzy tabele o rozmarze *N=5*.

chcemy je wysumować: tzn. dla *i<N*:

$$d_i = a_i + b_i + c_i$$

```
import numpy as np
a = np.array([11,23,32,23,12])
b = np.array([1,2,2,3,1])
c = np.array([ 10,20,30,20,10])
d = np.zeros_like(a)
d = a+b+c
print d
```



[Permalink]

Numpy: array slicing

Mamy:

`a[from:to:step]`

`a[:],a`	11	23	32	23	12
`a[-1:1]`	11	23	32	23	12
`a[:-2]`	11	23	32	23	12
`a[0:3:2]`	11	23	32	23	12
`a[::2]`	11	23	32	23	12
`a[0::-1]`	12	23	32	23	11

to tyłu!

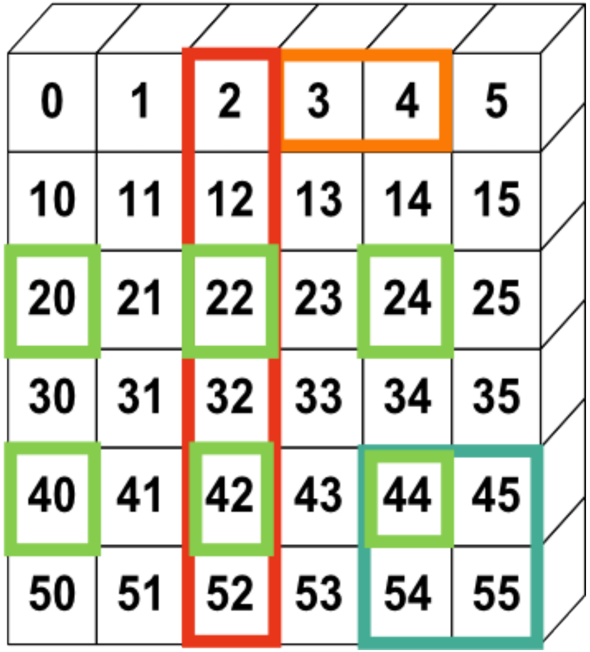
Numpy 2d array slicing

```
>>> a[0,3:5]
array([3,4])
```

```
>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
```

```
>>> a[:,2]
array([2,12,22,32,42,52])
```

```
>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



http://scipy-lectures.github.io/intro/numpy/numpy.html

Numpy: dodawanie w technice “slice”.

Mamy trzy tabele rozmiaru N=5.

Chcemy wykonać następującą operacje:

$$i>0 \land i<(N-1)$$

$$d_i = a_i + b_{i-1} + c_{i+1}$$

a[:]	11	23	32	23	12
b[:]	1	2	2	3	1
c[:]	10	20	30	20	10
d[:]	0	0	0	0	0

Numpy: dodawanie w technice “slice”.

Mamy trzy tabele rozmiaru N=5.

Chcemy wykonać następującą operacje:

$$i > 0 \wedge i < (N-1)$$

$$d_i = a_i + b_{i-1} + c_{i+1}$$

1. We slice all arrays:

`a[1:-1]`	11	23	32	23	12
`b[:-2]`	1	2	2	3	1
`c[2:]`	10	20	30	20	10
`d[1:-1]`	0	0	0	0	0

Numpy: dodawanie w technice “slice”.

Mamy trzy tabele rozmiaru N=5.

Chcemy wykonać następującą operacje:

$$i>0 \wedge i<(N-1)$$

$$d_i = a_i + b_{i-1} + c_{i+1}$$

2. All 4 slices are of **the same size**:

`a[1:-1]`

11	23	32	23	12

`b[:-2]`

1	2	2	3	1

`c[2:]`

10	20	30	20	10

`d[1:-1]`

0	0	0	0	0

Numpy: dodawanie w technice “slice”.

Mamy trzy tabele rozmiaru N=5.

Chcemy wykonać następującą operacje:

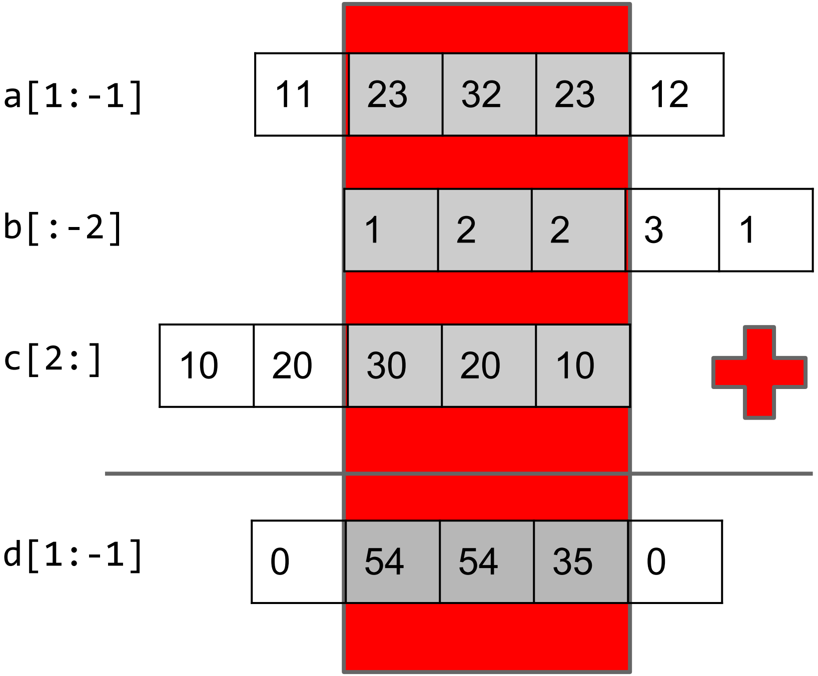
$$i>0 \wedge i<(N-1)$$

$$d_i = a_i + b_{i-1} + c_{i+1}$$

```
import numpy as np
a = np.array([11,23,32,23,12])
b = np.array([1,2,2,3,1])
c = np.array([ 10,20,30,20,10])
d = np.zeros_like(a)
d[1:-1] = a[1:-1]+b[:-2]+c[2:]
print d
```

[Permalink]

3. We can perform a vector addition.



Numpy i pętla for .

$$d_i = a_i + b_{i-1} + c_{i+1}$$

To samo można oczywiście wykonać za pomocą pętli for, ale będzie to nieefektywne w języku Python.

```
import numpy as np
a = np.array([11,23,32,23,12])
b = np.array([1,2,2,3,1])
c = np.array([ 10,20,30,20,10])
d = np.zeros_like(a)

for i in range(1,a.shape[0]-1):
    d[i] = a[i] + b[i-1] + c[i+1]

print "Elementwise loop:     ",d

d[1:-1] = a[1:-1]+b[:-2]+c[2:]

print "Numpy vector addition:",d
```

Permalink

```
Elementwise loop:      [ 0 54 54 35  0]
Numpy vector addition: [ 0 54 54 35  0]
```

Numpy i dyskretny Laplasjan w 1d

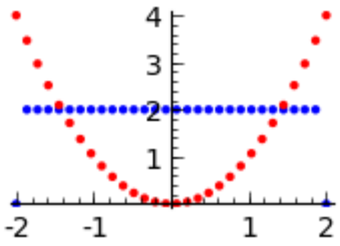
Możemy wykorzystać te techniki aby policzyć Laplasjan:

1. Próbkuje funkcję jednorodnie na odcinku
2. Stosujemy schemat różnic skończonych:

$$(\Delta^2 f)(x_i) \simeq \frac{1}{h^2}\left(f(x_{i-1}) + f(x_{i+1}) - 2f(x_i)\right)$$

3. Rysujemy wynik na wykresie dla sprawdzenia

```
import numpy as np
x = np.linspace(-2,2,30)
f = x**2
Lf = np.zeros_like(f)
h = x[1]-x[0]

Lf[1:-1] =1/h**2*( f[:-2]+f[2:]-2*f[1:-1] )

point(zip(x,Lf),figsize=4)+point(zip(x,f),color='red')
```



Permalink

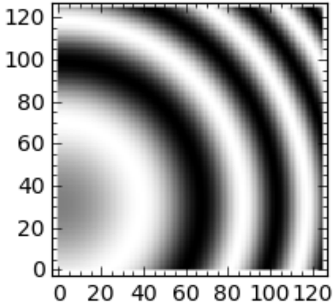
Numpy i dyskretny Laplasjan w 2d

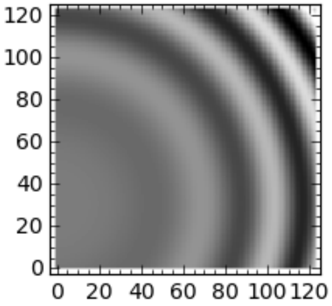
Możemy wykorzystać te techniki aby policzyć Laplasjan w 2d:

1. Próbkuje funkcję jednorodnie na prostokątnym obszarze.
2. Stosujemy schemat różnic skończonych:

$$(\Delta^2 f)(x_{ij}) \simeq \frac{1}{h^2}\Big(f(x_{i-1,j}) + f(x_{i,j-1}) + f(x_{i+1,j}) + f(x_{i,j+1}) - 4f(x_{i,j}) \Big)$$

3. Rysujemy wynik na wykresie dla sprawdzenia



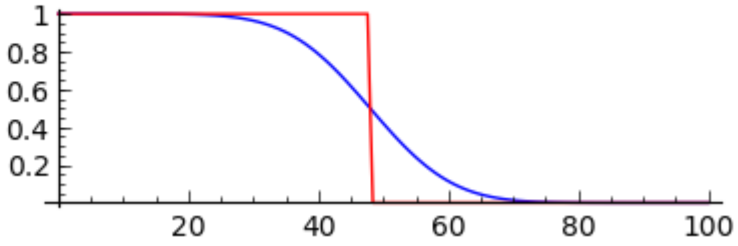


Permalink

```
import numpy as np
x = np.linspace(0,4,125)
y = np.linspace(-1,3,125)
h = x[1]-x[0] # !assume x,y spacing the same!
X,Y = np.meshgrid(x,y)
f = np.sin(X**2 + Y**2)
p1=matrix_plot(f,figsize=3,origin='lower')
Lf = 1/h**2 *\
 ( f[1:-1,:-2]+f[1:-1,2:]+\
   f[:-2,1:-1]+f[2:,1:-1]-4*f[1:-1,1:-1])
p2=matrix_plot(Lf,figsize=3,origin='lower')

Lf_a = -4*X^2*sin(X^2 + Y^2) - \
 4*Y^2*sin(X^2 + Y^2) + 4*cos(X^2 + Y^2)

p3=matrix_plot(Lf_a,figsize=3,origin='lower')

html.table([[r"$f(x,y)$",\
 r"$\Delta^2 f(x,y)$ numerical",\
 r"$\Delta^2 f(x,y)$ analytical"],[p1,p2,p3]])
```

Przykład: równanie dyfuzji

Możemy rozwiązać równanie dyfuzji stosując w czasie jawny schemat Eulera i dyskretyzację w przestrzeni drugiego rzędu.




```
import numpy as np

N = 125
D = 1.0
l =100.0
h = l/(N-1)
dt = 0.05
X = np.linspace(0,l,N)

u = np.zeros(N)
u[:60] = 1.0
p0 = line(zip(X,u),color='red')

for i in range(1000):
    u[1:-1] = u[1:-1] + dt*D/h^2*(u[2:]+u[:-2]-2*u[1:-1])
    u[0] = u[1]
    u[-1] = u[-2]

p = line(zip(X,u),figsize=(7,2))+p0
show(p)
```