Python Performance for Plants and Profit

Dr. Olaf Flebbe
of ät oflebbe.de

Tübix 2018

About me

PhD in computational physics

Former projects: Minix68k (68k FP Emulation), Linux libm.so.5 (High Precision FP), perl and python for epoc, flightgear, msktutil…

Member Apache Software Foundation

Backend Software Architect at Bosch eBike

LUGT Mailinglist

Ein längeres Programm braucht bei meinem Kollegen 5 h und 10 Minuten, bei mir über 13 h.

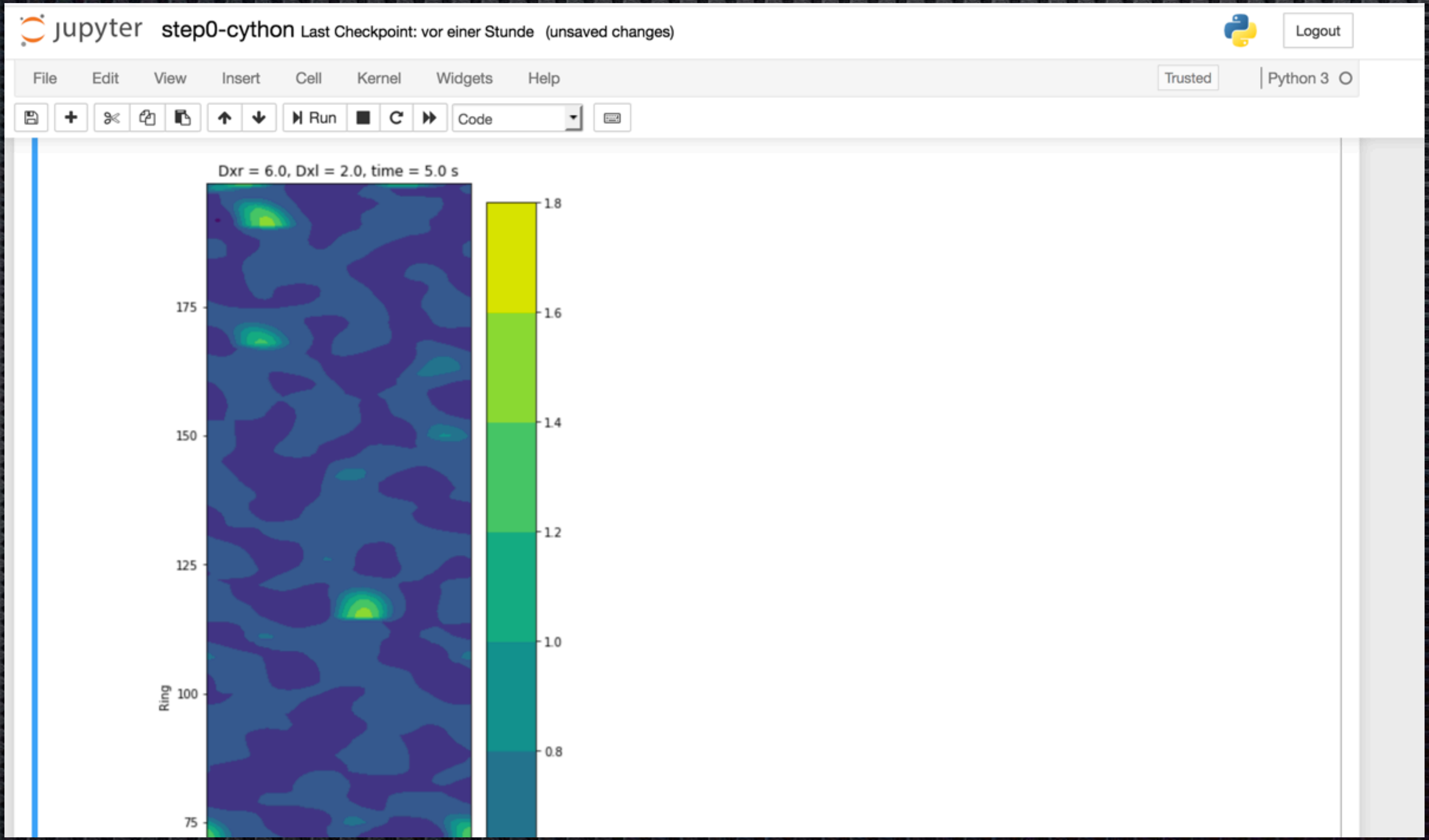
LUGT Mailinglist

Simulation Pflanzenbewegungen

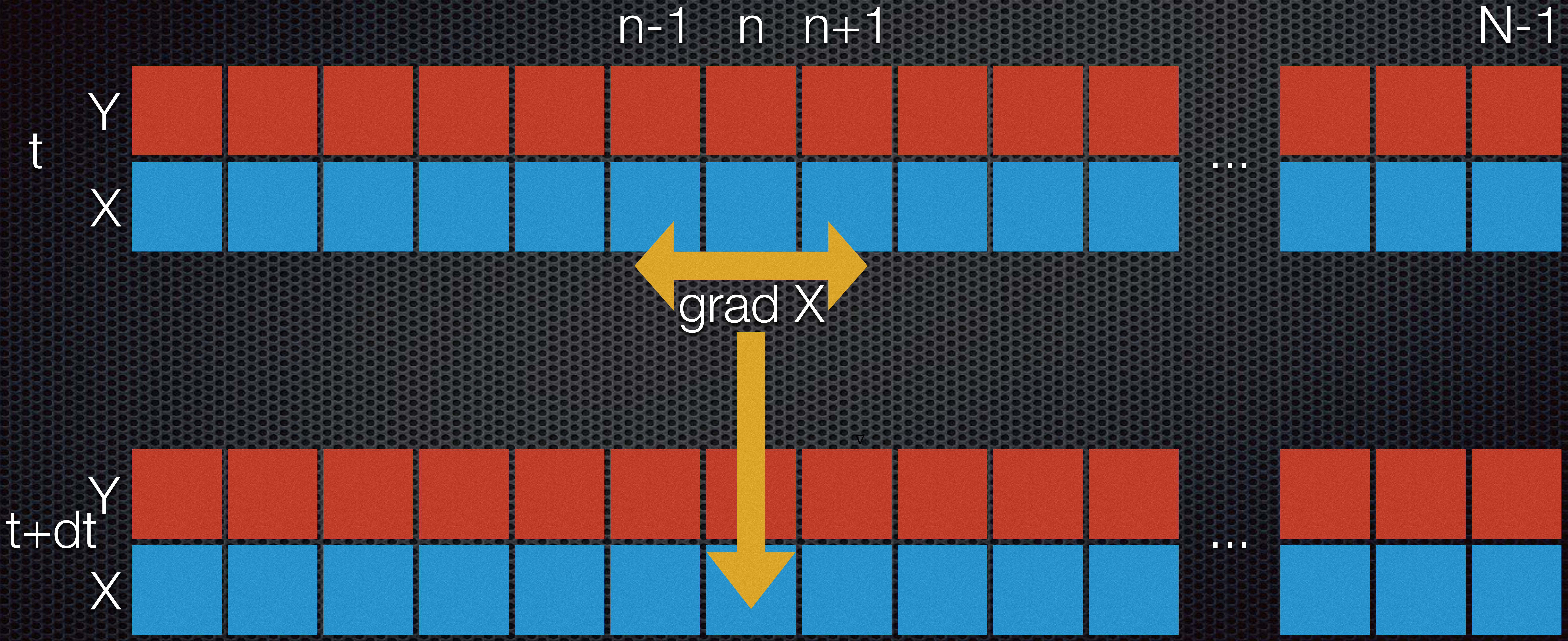
Inner Loop

```
for t in range(0,N):
   for ring in range(0,rings):
        for cell in range(0,ringsize):

            flux = ....

# Perform Runge-Kutta integration on current cell
            x = X[t][ring][cell]
            y = Y[t][ring][cell]
            k11 = h*dXdt(x,y,flux)
            k12 = h*dYdt(x,y)

            k21 = h*dXdt(x+0.5*k11, y+0.5*k12,flux)
            k22 = h*dYdt(x+0.5*k11, y+0.5*k12)

            k31 = h*dXdt(x+0.5*k21, y+0.5*k22,flux)
            k32 = h*dYdt(x+0.5*k21, y+0.5*k22)

            k41 = h*dXdt(x+k31, y+k32,flux)
            k42 = h*dYdt(x+k31, y+k32)

            X[t+1][ring][cell] = x + (k11 + 2*k21 + 2*k31 + k41)/6
            Y[t+1][ring][cell] = y + (k12 + 2*k22 + 2*k32 + k42)/6
```

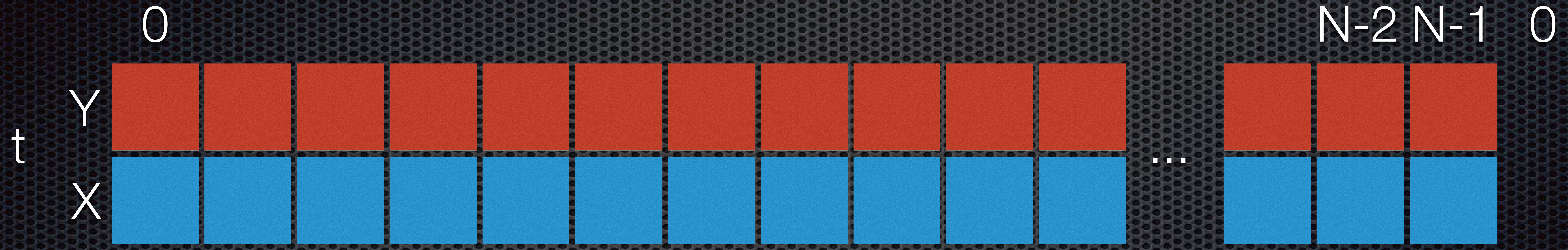
Running in jupyter



Diskretisierte DGL

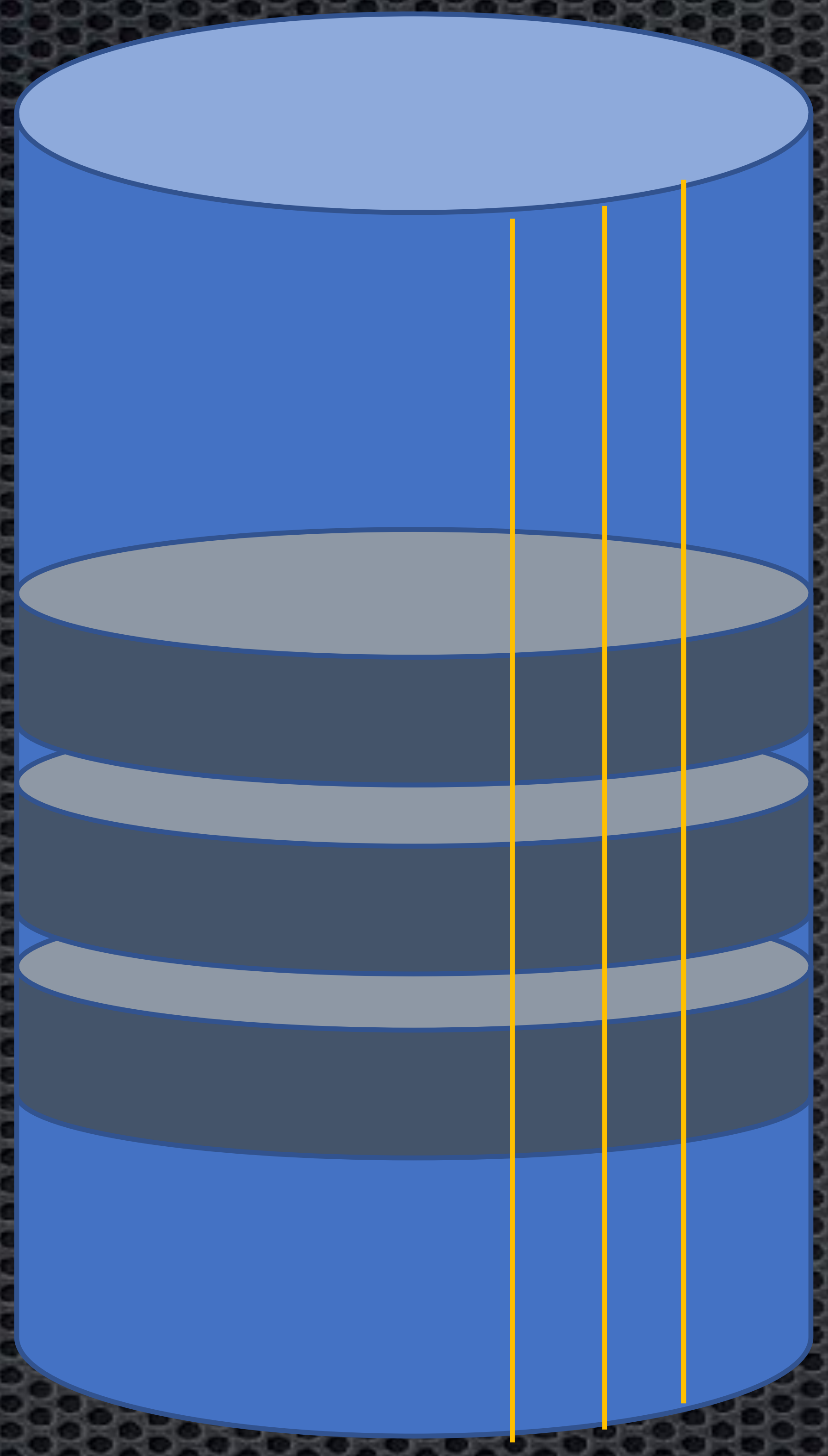


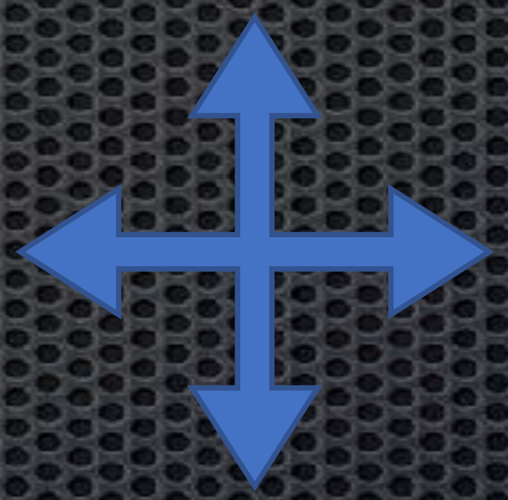
Randbedingung : Ring



Netztopologie

flux





Horizontaler Gradient

Vertikaler Gradient wenn Richtung stimmt

Numpy // python dynamische Typen

```
A = 1

A + 2 #  3

A = np.arange(3) #  [ 0, 1, 2]

B = A + 1                 #  [ 1, 2, 3] elementwise

A + B                      #  [ 1, 3, 5] elementwise

A * B                     #  [ 0, 2, 6] elementwise

A ** 2                   #  [ 0, 1, 4] elementwise
```

Numpy // python dynamic types

```
np.sin(A)            # elementwise sin

np.greater(A,B)  # elementwise true/false if A[i] > B[i]

A = np.arange(3)

#example functions work both with scalar and vector

def f(x) :

    return x**2


for i in range(0,3):                          # b = f(a)

    b[i] = f(a[i])
```

Grundsätzliche Schleife

```
for t in range(0,N):
  for ring in range(0,rings):
      for cell in range(0,ringsize):

         flux = ....

# Perform Runge-Kutta integration on current cell
         x = X[t][ring][cell]
         y = Y[t][ring][cell]
         k11 = h*dXdt(x,y,flux)
         k12 = h*dYdt(x,y)

         k21 = h*dXdt(x+0.5*k11, y+0.5*k12,flux)
         k22 = h*dYdt(x+0.5*k11, y+0.5*k12)

         k31 = h*dXdt(x+0.5*k21, y+0.5*k22,flux)
         k32 = h*dYdt(x+0.5*k21, y+0.5*k22)

         k41 = h*dXdt(x+k31, y+k32,flux)
         k42 = h*dYdt(x+k31, y+k32)

         X[t+1][ring][cell] = x + (k11 + 2*k21 + 2*k31 + k41)/6
         Y[t+1][ring][cell] = y + (k12 + 2*k22 + 2*k32 + k42)/6
```

Idee: Vectorisieren, viele voneinander
unabhängige Gleichartige Berechnungen

Profit

The tight internal loop is now done by numpy

Numpy uses optimized routines for vectors, matrices

BLAS (Basic Linear Algebra System)

These may be hand optimized to match processor

Vectorisieren

SIMD: Single Instruction, Multiple Data

Voraussetung: Keine Datenabhängigkeiten

Intel/AMD: AVX/AVX2, ... spezielle Register die mehrere FP aufnehmen können.

ARM: "Neon"

Dazu muss man den Code attributieren.

Boundary conditions

```
flux = 0.0

# horizontal diffusion
        if cell == 0:
            flux = Dxr*(X[index][ring][ringsize-1] - X[index][ring][cell]) \
              + Dxl*(X[index][ring][1] - X[index][ring][cell])
        elif cell==ringsize-1:
            flux = Dxr*(X[index][ring][cell-1] - X[index][ring][cell]) \
                 + Dxl*(X[index][ring][0] - X[index][ring][cell])
        else:
            flux = Dxr*(X[index][ring][cell-1] - X[index][ring][cell]) \
                 + Dxl*(X[index][ring][cell+1] - X[index][ring][cell])
```

Nachbarschaftsbeziehungen

```
# Boundary conditions will be handled automatically

xr = np.roll(X[index][ring], 1)
xl = np.roll(X[index][ring], -1)
flux = Dxr * (xr - X[index][ring]) + Dxl * (xl - X[index][ring])
```

boolean Indices

```
# vertical diffusion
            if X[index][ring-1][cell] > X[index][ring][cell]:
                flux += Dxd*(X[index][ring-1][cell] - X[index][ring][cell])
            if X[index][ring][cell] > X[index][ring+1][cell]:
                flux -= Dxd*(X[index][ring][cell] - X[index][ring+1][cell])
```

```
# Boolean indices !

d = np.greater(X[index][ring - 1], X[index][ring])

# d == True calc/do not calc corresponding value

# Array Access by Array of bools == Boolean index


flux[d] += Dxd * (X[index][ring - 1][d] - X[index][ring][d])
d = np.greater(X[index][ring], X[index][ring + 1])
flux[d] -= Dxd * (X[index][ring][d] - X[index][ring + 1][d])
```

Problem numpy

- Viel mehr geht nicht...

cython.org

Cython is an **optimising static compiler** for both the **Python** programming language and the extended Cython programming language (based on **Pyrex**). It makes writing C extensions for Python as easy as Python itself.

Cython gives you the combined power of Python and C to let you

- write Python code that calls back and forth from and to C or C++ code natively at any point.
- easily tune readable Python code into plain C performance by adding static type declarations.
- use combined source code level debugging to find bugs in your Python, Cython and C code.
- interact efficiently with large data sets, e.g. using multi-dimensional NumPy arrays.
- quickly build your applications within the large, mature and widely used CPython ecosystem.
- integrate natively with existing code and data from legacy, low-level or high-performance libraries and applications.

Cython: Loop

Idee: Den Rechenkern von Cython optimieren lassen

```
for t in range(0,N):
    for ring in range(0,rings):
        timestep(X, Y, cylinderConcPerCell, index, ring, t, tau, omega, Am)
```

Cython: timestep.pyx

```
cpdef timestep(X, Y, cylinderConcPerCell, index, ring,  t,  tau,  omega,  Am):

    for cell in range(0,ringsize):
        flux = ....

# Perform Runge-Kutta integration on current cell
        x = X[t][ring][cell]
        y = Y[t][ring][cell]
        k11 = h*dXdt(x,y,flux)
        k12 = h*dYdt(x,y)

        k21 = h*dXdt(x+0.5*k11, y+0.5*k12,flux)
        k22 = h*dYdt(x+0.5*k11, y+0.5*k12)

        k31 = h*dXdt(x+0.5*k21, y+0.5*k22,flux)
        k32 = h*dYdt(x+0.5*k21, y+0.5*k22)

        k41 = h*dXdt(x+k31, y+k32,flux)
        k42 = h*dYdt(x+k31, y+k32)

        X[t+1][ring][cell] = x + (k11 + 2*k21 + 2*k31 + k41)/6
        Y[t+1][ring][cell] = y + (k12 + 2*k22 + 2*k32 + k42)/6
```

Idee: Vectorisieren, viele voneinander
unabhängige Gleichartige Berechnung

Cython:
optimize timestep.pyx

```
@cython.boundscheck(False)
cpdef timestep(double[:,:,:] X, double[:,:,:] Y, double[:,:] cylinderConcPerCell, int index,
                    int ring, int t, int tau, double omega, double Am):
          cdef double flux
          cdef double x, y
          cdef double k11, k12, k21, k22, k31, k32, k41, k42
          cdef int cell
          flux = ....

# Perform Runge-Kutta integration on current cell
          x = X[t][ring][cell]
          y = Y[t][ring][cell]
          k11 = h*dXdt(x,y,flux)
          k12 = h*dYdt(x,y)

          k21 = h*dXdt(x+0.5*k11, y+0.5*k12,flux)
          k22 = h*dYdt(x+0.5*k11, y+0.5*k12)

          k31 = h*dXdt(x+0.5*k21, y+0.5*k22,flux)
          k32 = h*dYdt(x+0.5*k21, y+0.5*k22)

          k41 = h*dXdt(x+k31, y+k32,flux)
          k42 = h*dYdt(x+k31, y+k32)

          X[t+1][ring][cell] = x + (k11 + 2*k21 + 2*k31 + k41)/6
          Y[t+1][ring][cell] = y + (k12 + 2*k22 + 2*k32 + k42)/6
```

Idee: typed memoryviews
Schneller geht's nicht...

Leider funktioniert das nicht für
Scalar * Vector...

Deswegen ausgangsbasis ursprünglicher Code
Step0

Cython:
Auszug aus timestep.c

```
 /* "timestep.pyx":121
*            k42 = h*dYdt(x+k31, y+k32)
*
*            Xret = x + (k11 + 2*k21 + 2*k31 + k41)/6            # <<<<<<<<<<<<<<
*            Yret = y + (k12 + 2*k22 + 2*k32 + k42)/6
*            X[(index+1)%tau][ring][cell] = Xret
*/
  __pyx_v_Xret = (__pyx_v_x + ((((__pyx_v_k11 + (2.0 * __pyx_v_k21)) + (2.0 * __pyx_v_k31)) + __pyx_v_k41) / 6.0));


  /* "timestep.pyx":122
*
*            Xret = x + (k11 + 2*k21 + 2*k31 + k41)/6
*            Yret = y + (k12 + 2*k22 + 2*k32 + k42)/6            # <<<<<<<<<<<<<<
*            X[(index+1)%tau][ring][cell] = Xret
*            Y[(index+1)%tau][ring][cell] = Yret
*/
  __pyx_v_Yret = (__pyx_v_y + ((((__pyx_v_k12 + (2.0 * __pyx_v_k22)) + (2.0 * __pyx_v_k32)) + __pyx_v_k42) / 6.0));
```

Bewertung

- Aufwändig:
  - Typdefinitionen
  - Keine Operationen scalar mit vector (Broadcasting)
- Raum für weitere Optimierungen (Compiler)
- Weit verbreitet

numba.pydata.org

- JIT (Just in time) compiler für python, aufbauend auf LLVM
- Benötigt sehr viel Infrastrucktur
- (Anaconda bringt alles mit)
- Integriert mit numpy (!)

Logic from Numba

```
@njit(cache = True)
def ringdynamics( X, Y, cylinderConcPerCell, index, t, tau, h, omega, Am):

    for ring in range(rings):
        flux = ....

 # Perform Runge-Kutta integration on current cell

         x = X[index][ring]
        y = Y[index][ring]
        k11 = h * dXdt(x, y, flux)
        k12 = h * dYdt(x, y)
        k21 = h * dXdt(x + 0.5 * k11, y + 0.5 * k12, flux)
        k22 = h * dYdt(x + 0.5 * k11, y + 0.5 * k12)
        k31 = h * dXdt(x + 0.5 * k21, y + 0.5 * k22, flux)
        k32 = h * dYdt(x + 0.5 * k21, y + 0.5 * k22)
        k41 = h * dXdt(x + k31, y + k32, flux)
        k42 = h * dYdt(x + k31, y + k32)

        X[(index + 1) % tau][ring] = x + (k11 + 2 * k21 + 2 * k31 + k41) / 6
        Y[(index + 1) % tau][ring] = y + (k12 + 2 * k22 + 2 * k32 + k42) / 6
```

Idee: Vectorcode als Funktion damit dieser annotiert werden kann

@njit: Fehler, wenn Optimierung nicht durchgeführt werde

numba Bewertung:

- Sehr einfaches Handling
  - Geht oder Geht nicht.
- Kann nicht komplettes numpy
  - np.roll musste ersetzt werden durch "kompatible" Implementierung
  - Verfügbarkeit eingeschränkt

Was geht noch?

- Step 0 in "C" mit fixen Adresse

C

```
for (int ring = 0; ring < RINGS; ring++) {
		for (int cell = 0; cell < RINGSIZE; cell++) {
			double		flux = 0.;


			double flux = ....
			float		x = X[index][ring][cell];
			float		y = Y[index][ring][cell];
			double		k11 = h * dXdt(x, y, flux);
			double		k12 = h * dYdt(x, y);

			double		k21 = h * dXdt(x + 0.5 * k11, y + 0.5 * k12, flux);
			double		k22 = h * dYdt(x + 0.5 * k11, y + 0.5 * k12);

			double		k31 = h * dXdt(x + 0.5 * k21, y + 0.5 * k22, flux);
			double		k32 = h * dYdt(x + 0.5 * k21, y + 0.5 * k22);

			double		k41 = h * dXdt(x + k31, y + k32, flux);
			double		k42 = h * dYdt(x + k31, y + k32);
			X[(index + 1) % TAU][ring][cell] = x + (k11 + 2 * k21 + 2 * k31 + k41) / 6;
			Y[(index + 1) % TAU][ring][cell] = y + (k12 + 2 * k22 + 2 * k32 + k42) / 6;
```

Bewertung C

- Fixe Arrays sehr unflexibel
- Mehrdimensionale Dynamische Arrays sehr fehlerträchtig
- Binding über cython sehr einfach möglich
- Viel weiteres Optimierungspotential

Resultate

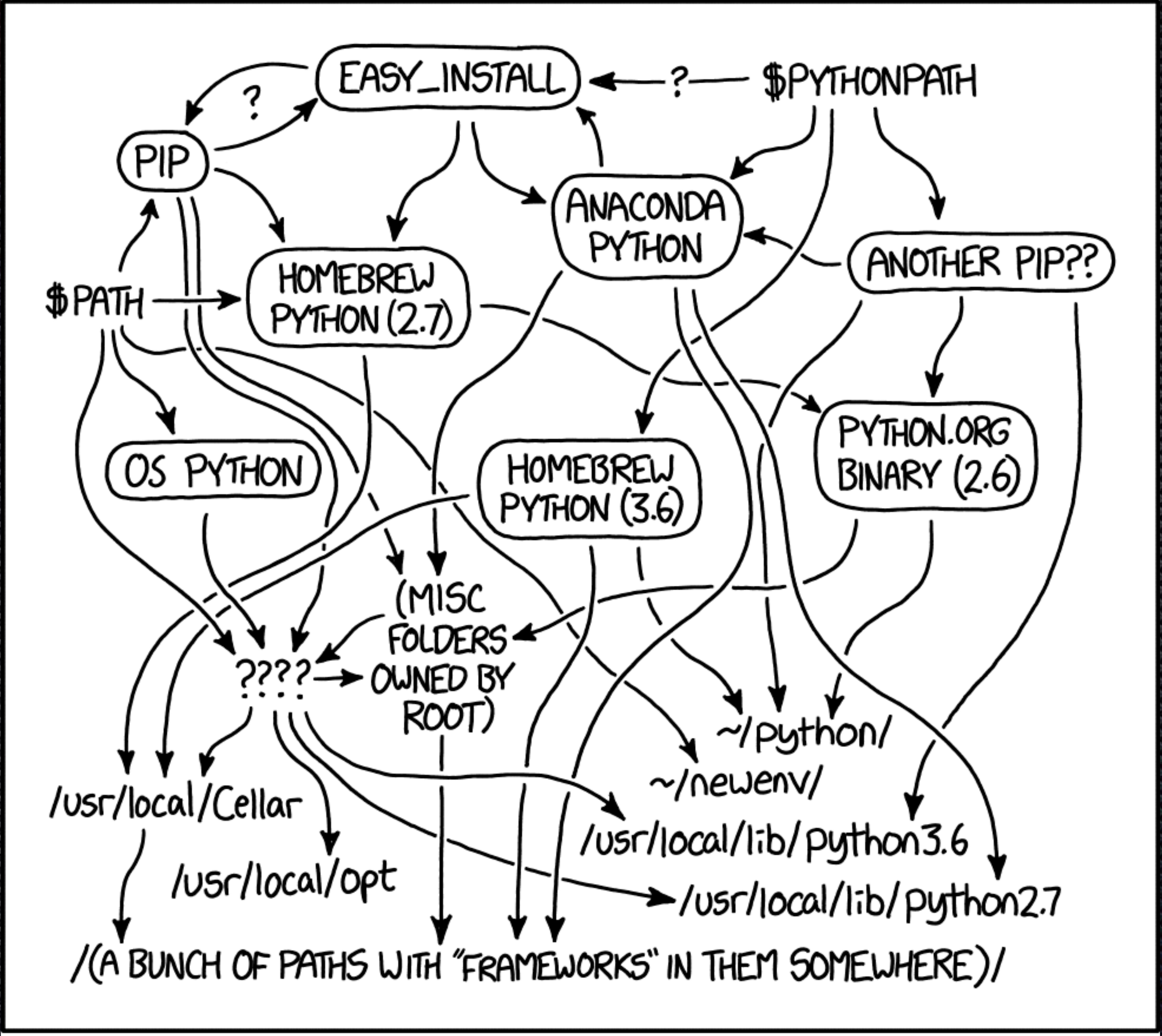
- step0: 125 sec
- step1/step2: 12sec
- step0 + optimites cython: 2sec
- step2 + optimiertes numba: 2sec
- step0 in C ohne matplotlib: 0.2sec

Ergebnis

- Python ist langsam
- Numpy ist schneller
- Cython ist aufwändig, sehr schnell
- Numba ist sehr einfach && sehr schnell, verfügbarkeit eingeschränkt
- Sowohl cython als auch numba integrieren gut in jupyter!
- C ist weiterhin am schnellsten.

Zum Mitnehmen

- jupyter + matplotlib ist geil zum explorativen Arbeiten.
- numba ist der Star, wenn es funktioniert.
- C ist und bleibt: Ressourcenschonend, schnell, aber fehlerträchtig und aufwändig.
- C++ ist schwierig den richtigen Stil zu finden
- Virtual environments verwenden!



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Danke!

github.com/oflebbe/pp4pp

of ät oflebbe dot de