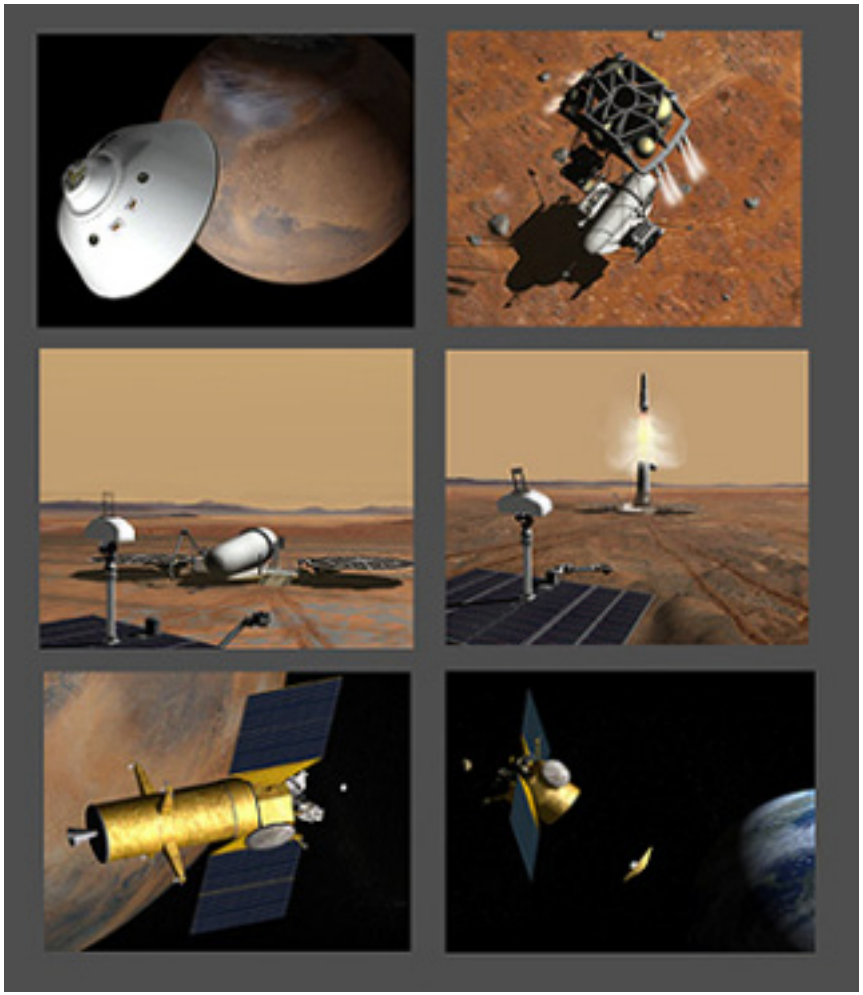
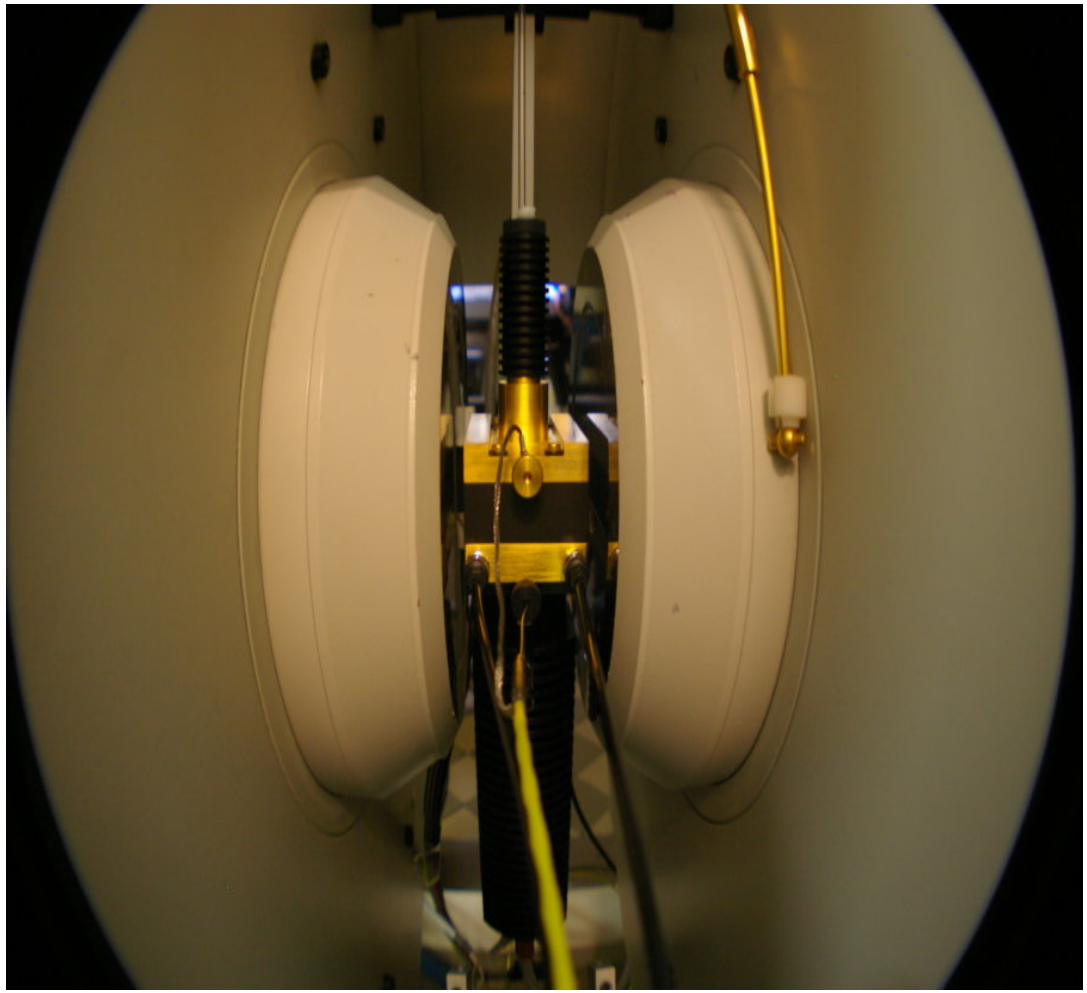
High-performance computing on gamer PCs

Yann Le Du, Mariem El Afrit, Laurent Binet, Didier Gourier
(LCMCP, Chimie ParisTech)

EUROPYTHON 2011
FLORENCE, JUNE 20-26

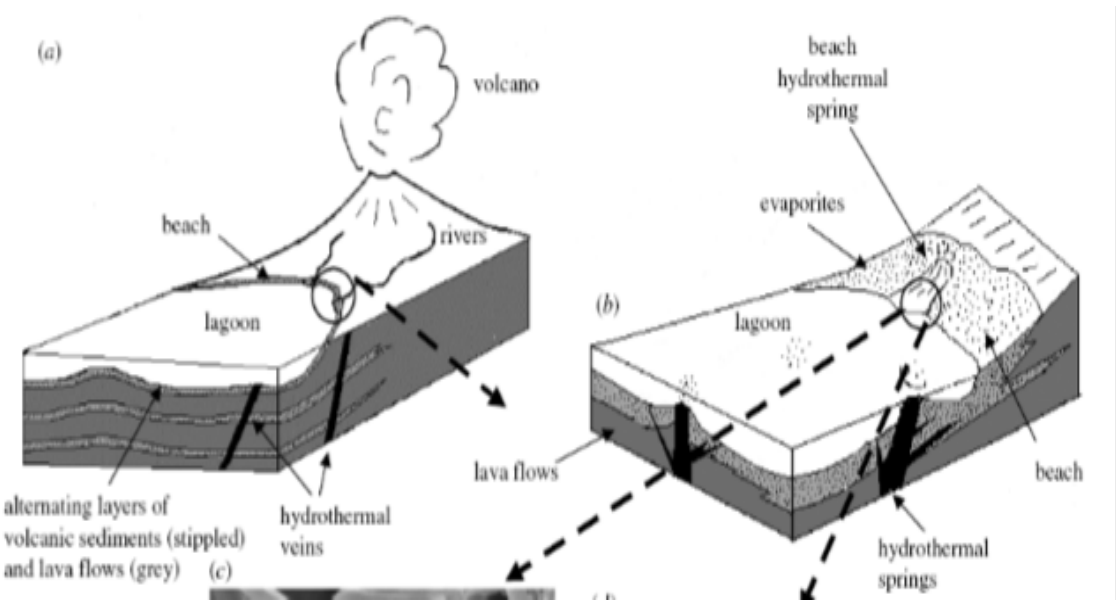
Framework: EPR imaging for exobiology

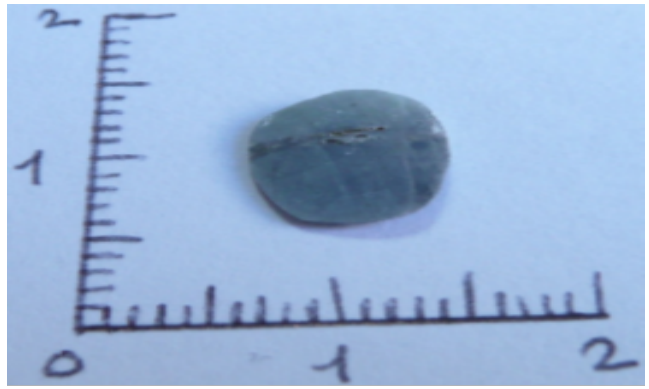


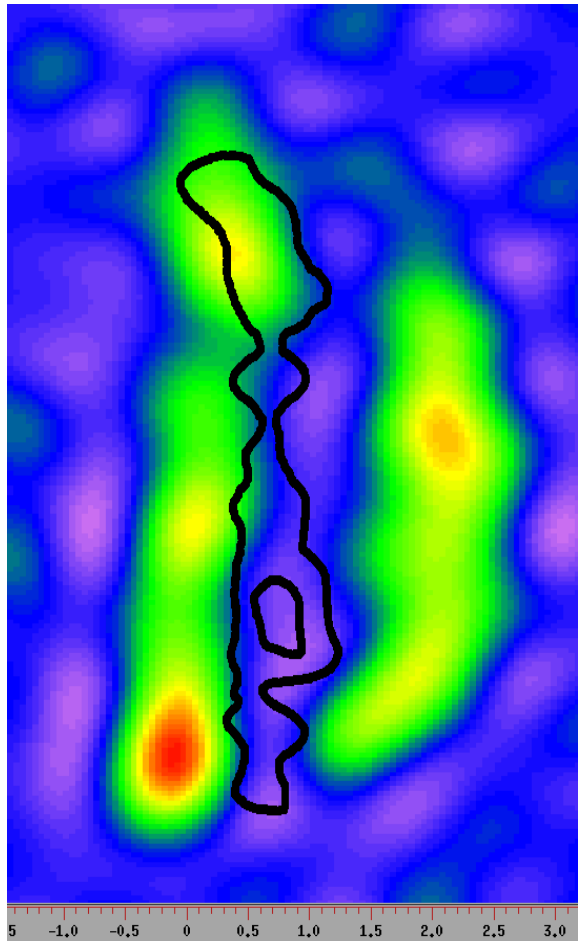


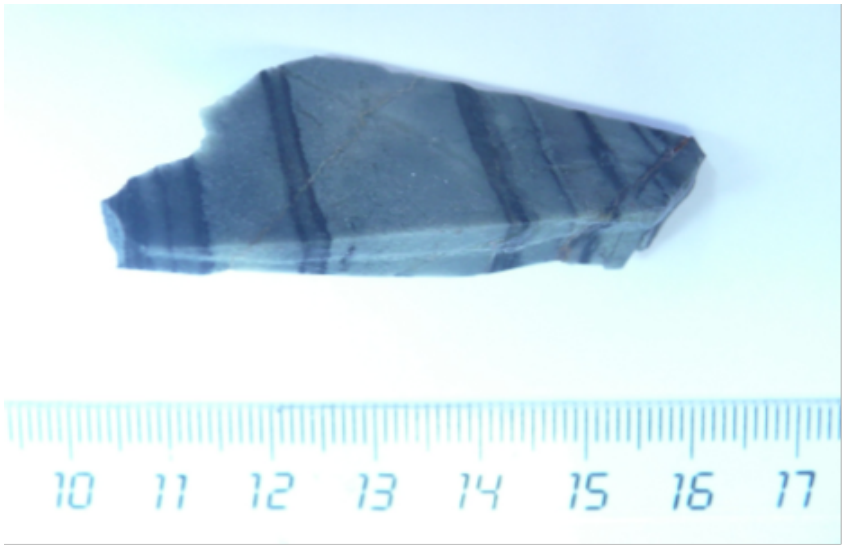
Goal: *in situ* EPR analysis with imaging to select samples on Mars

Context: EPR imaging for exobiology

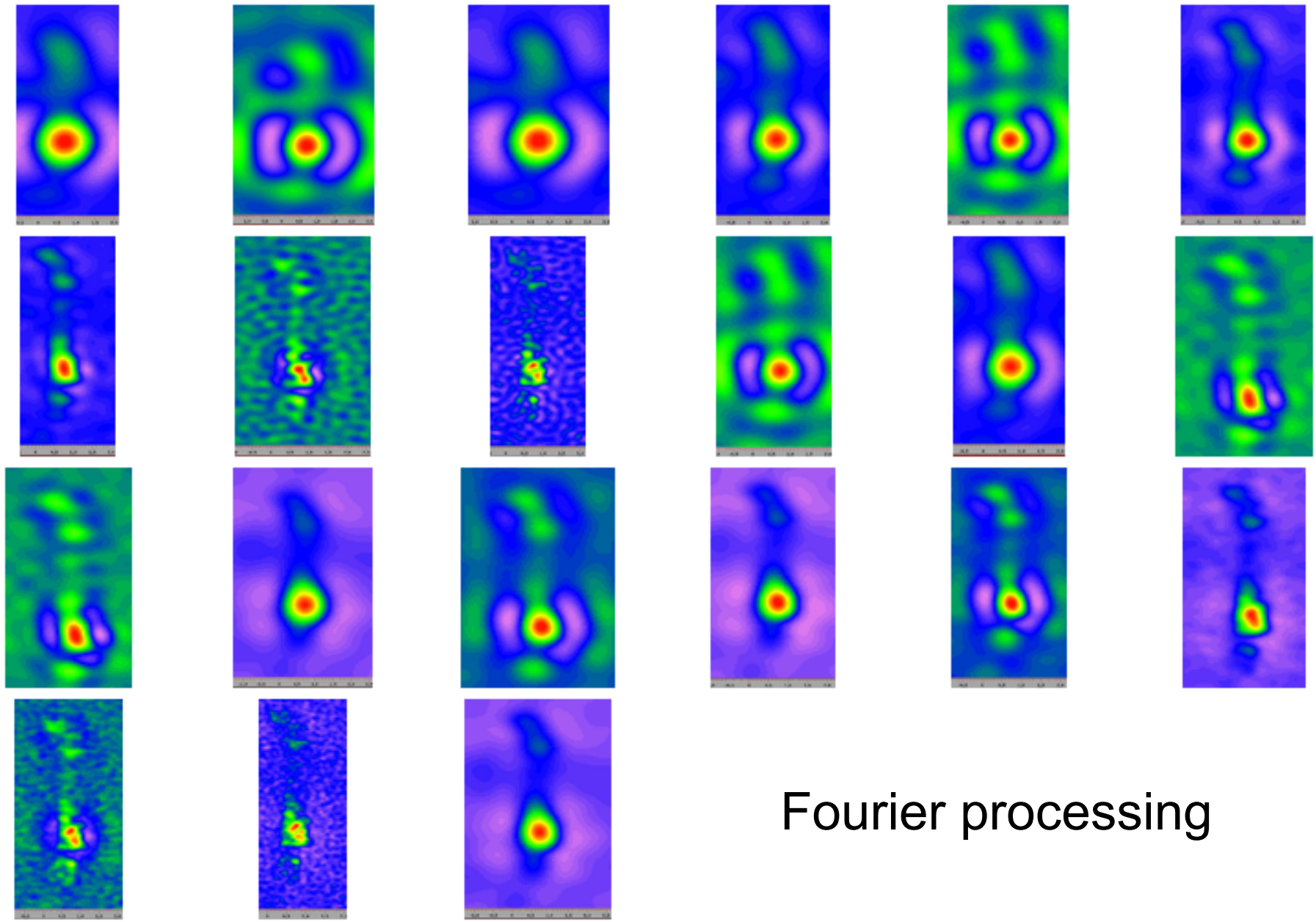








The manual processing



Fourier processing

The problem to solve

Inverse a Fredholm equation of type 1

r : knowm spectrum

c : known lineshape

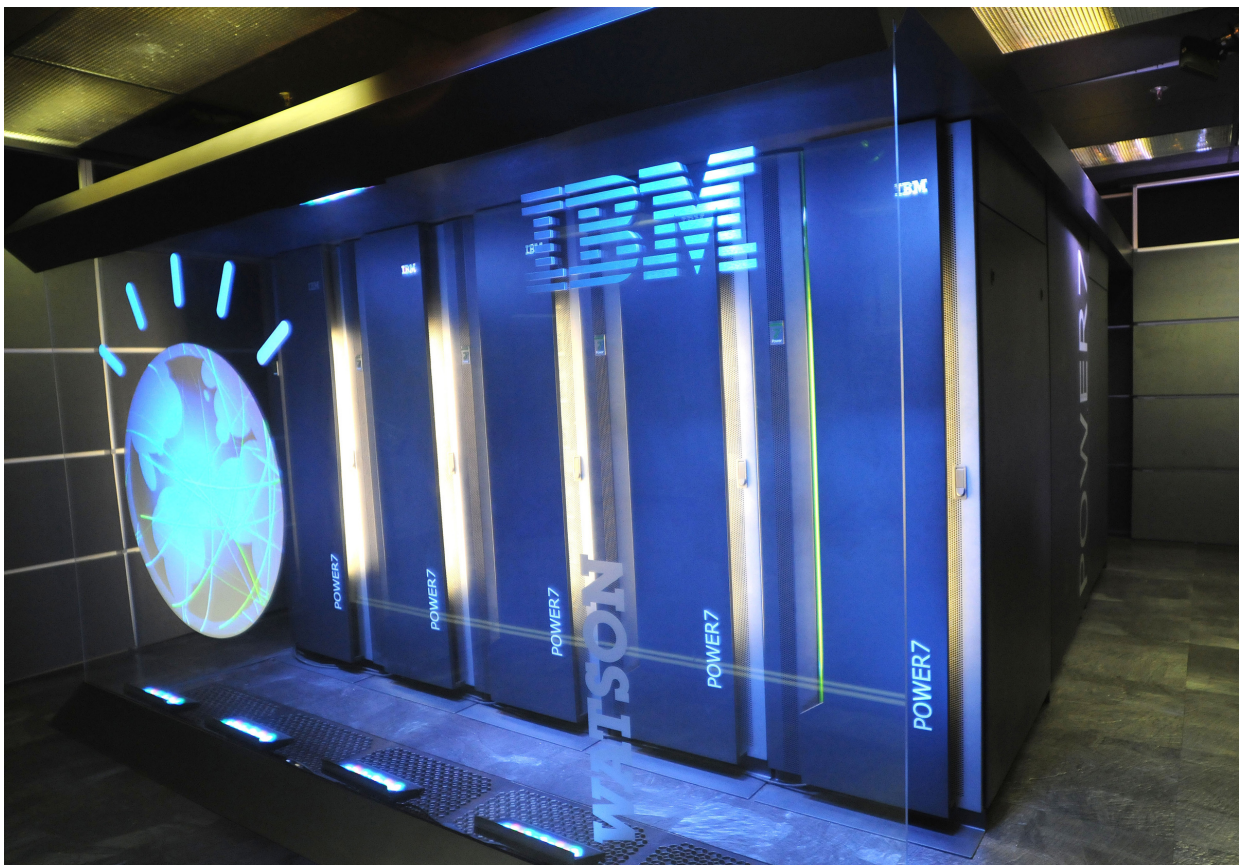
$$r(B) = \int_{\text{sample}} c(B + Gx)\, s(x)\, dx$$

s : unknown matter linear density

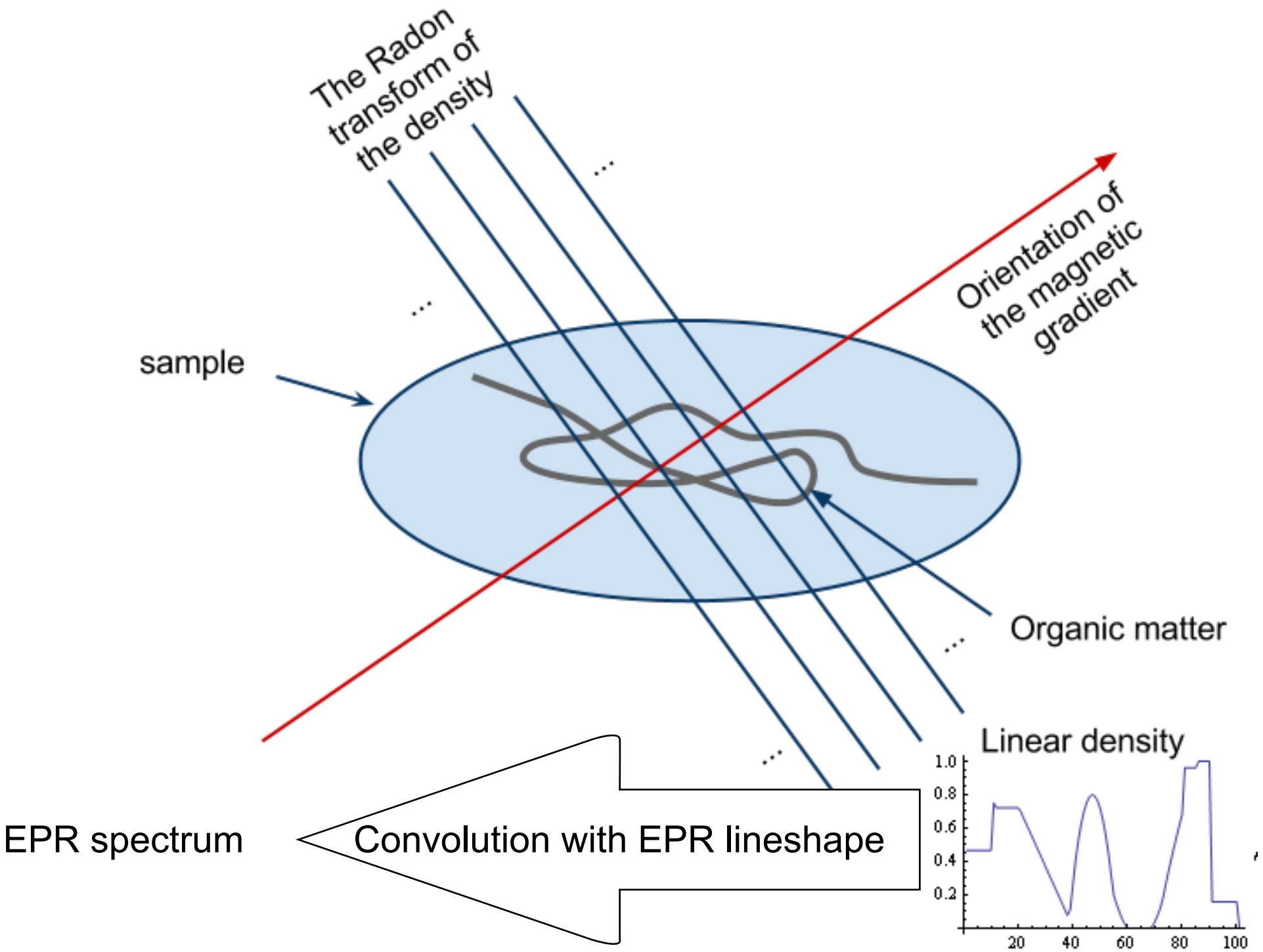
Machine learning method:

- teach *s* given *r*
- generate candidates

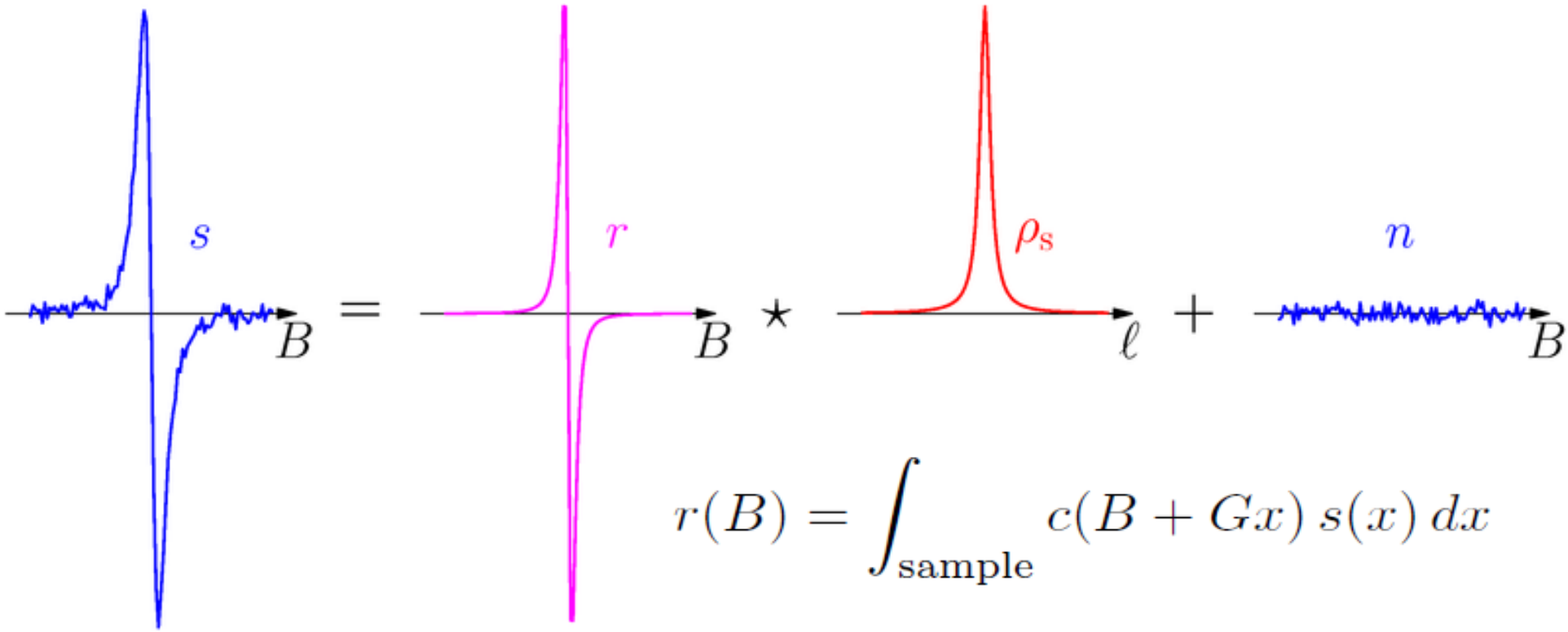
- combine candidates



The EPR spectrum and the linear density



Inverse the integral equation

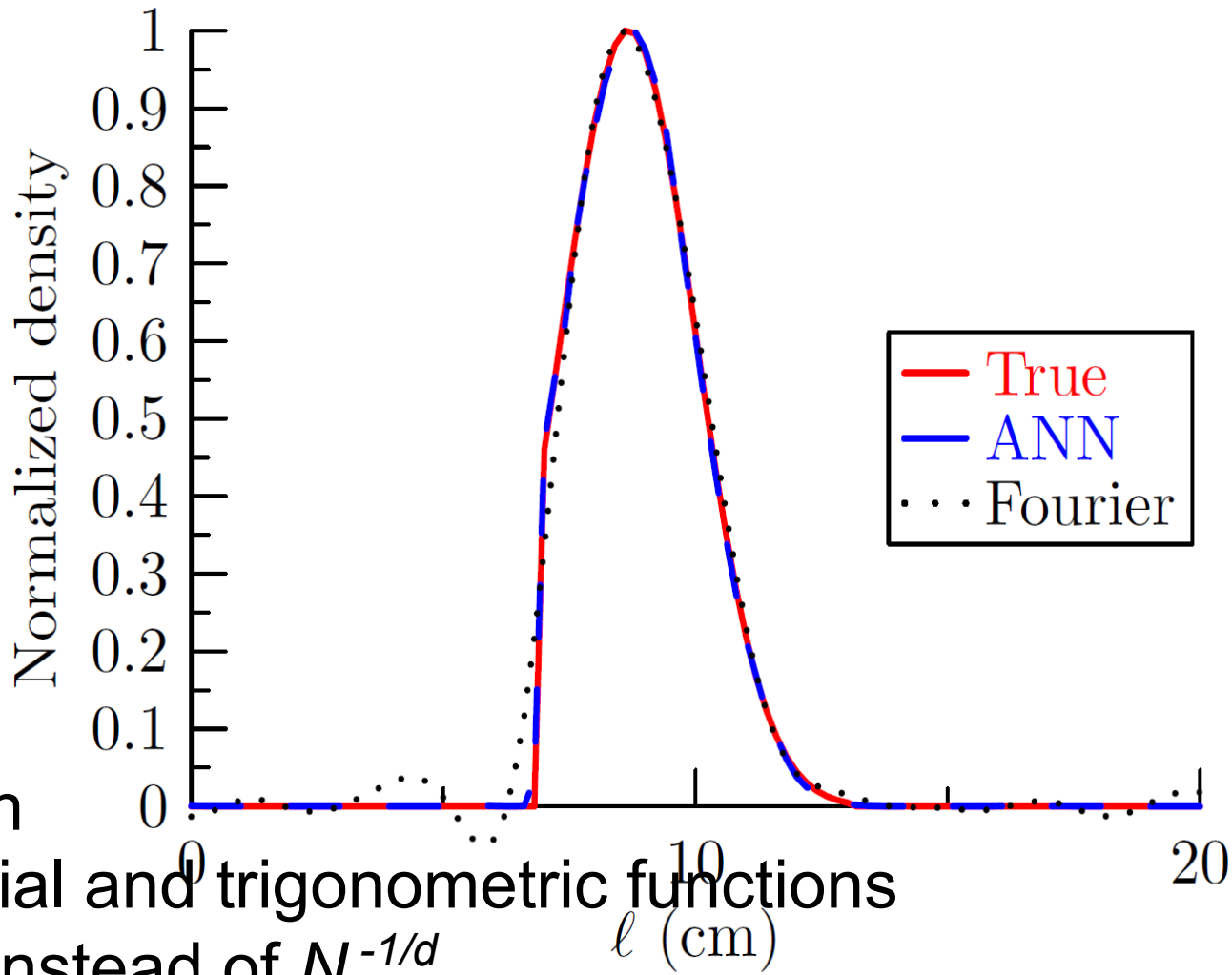


$$r(B) = \int_{\text{sample}} c(B + Gx)\, s(x)\, dx$$

- The Hadamard conditions
  - existence
  - unicity
  - stability
- Physical constraints :
  - positive density
  - special shapes to reconstruct

ANN : advantages

- Resistance to noise
  - comparison with optimal filters, Wiener deconvolution
- Parallelization
- Adapted to the context
- Very fast forward
- Universal approximation
  - better than polynomial and trigonometric functions
  - convergence in N^{-1} instead of $N^{-1/d}$

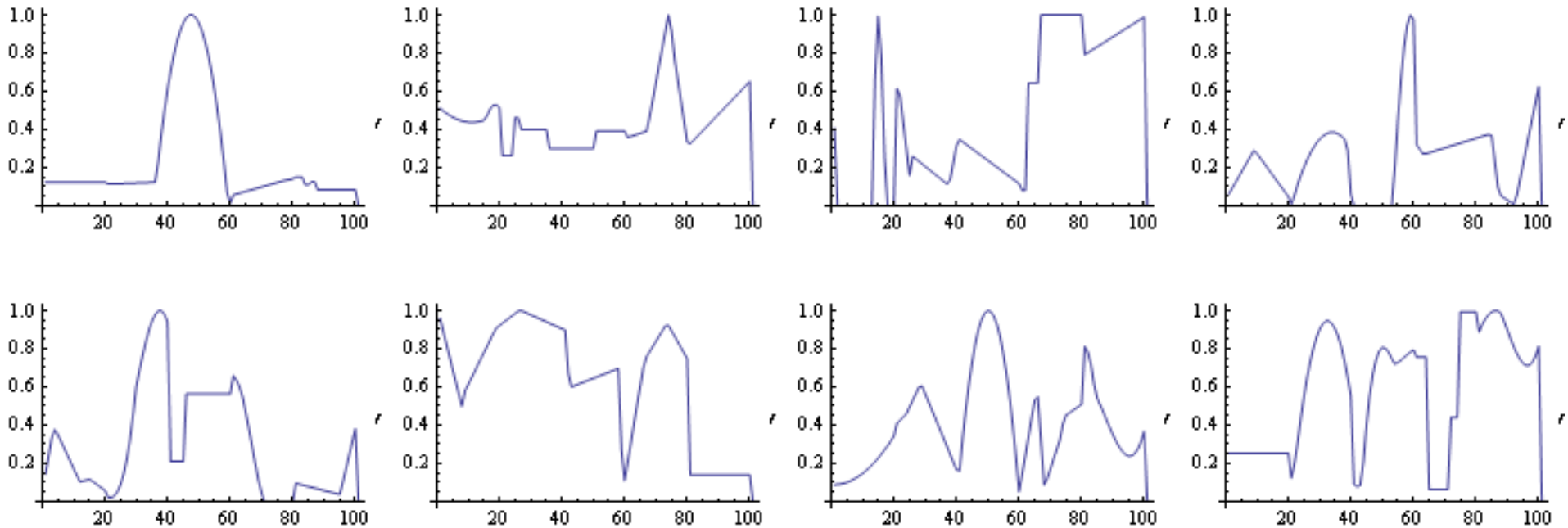


Modeling the machine and the density

Spectrum *r*

$$r(B) = \int_{\text{sample}} c(B + Gx)\, s(x)\, dx$$

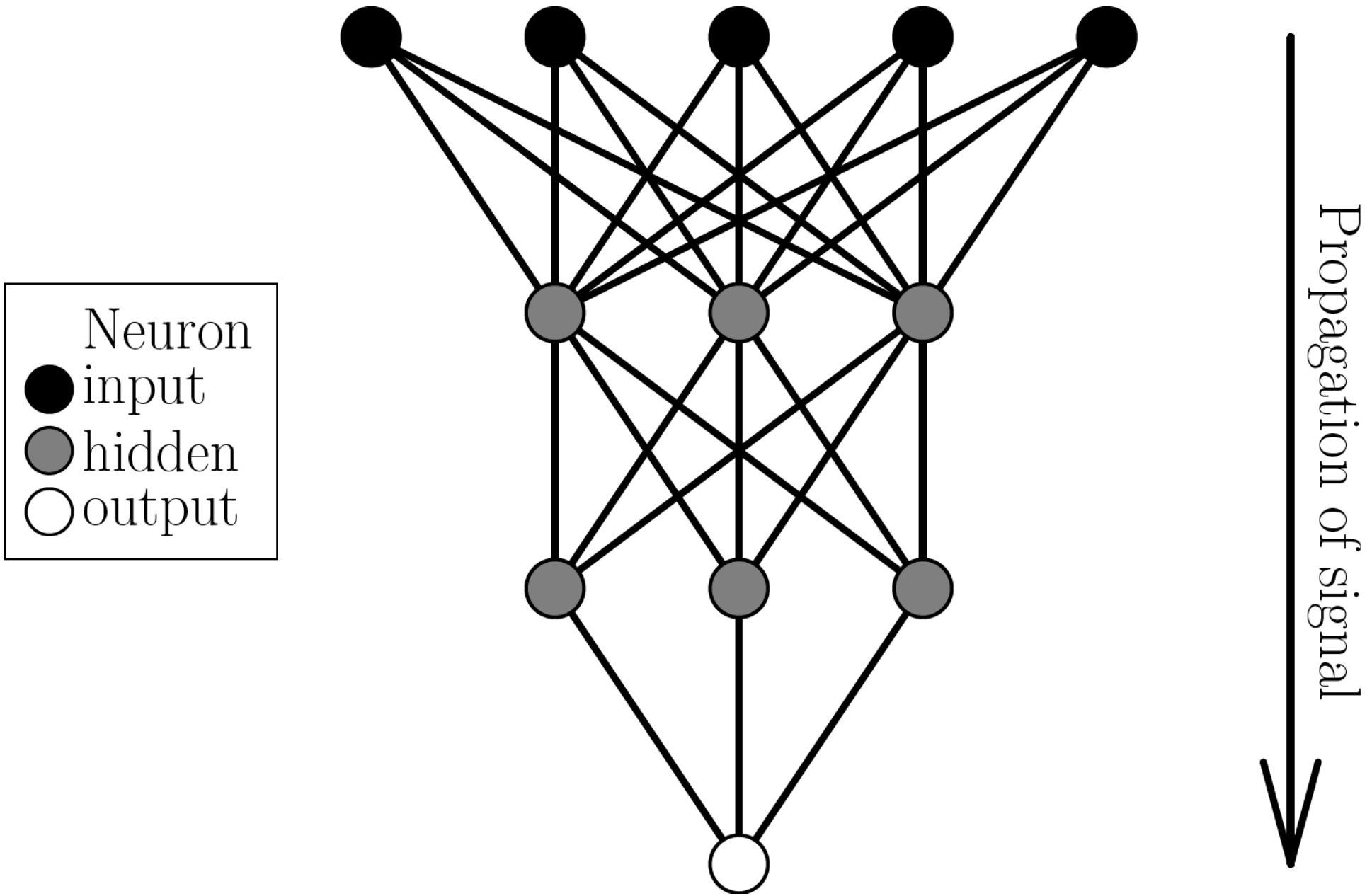
convolution kernel *c* = lineshape



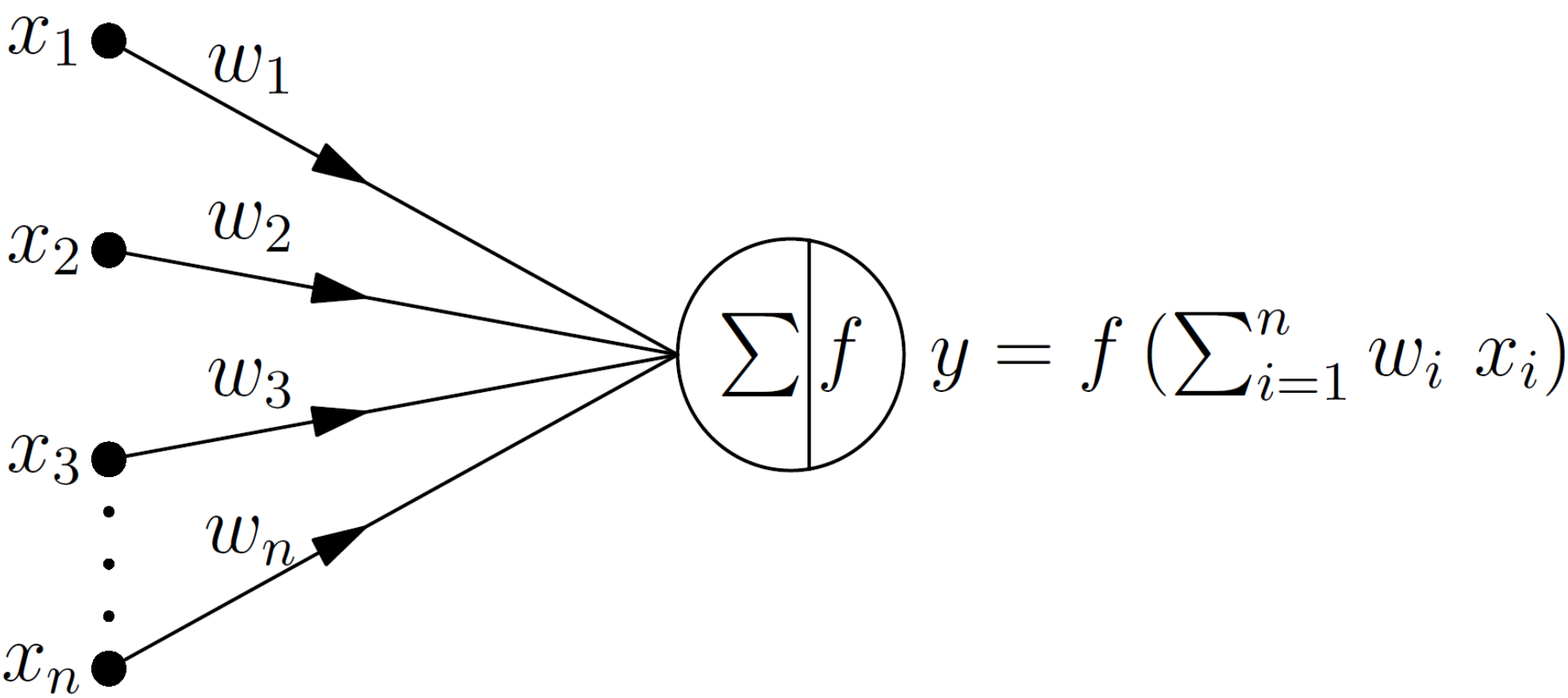
The density *s* : quasi-splines

PyLab (Scipy)

Neural networks and matrix algebra



Neural networks and matrix algebra



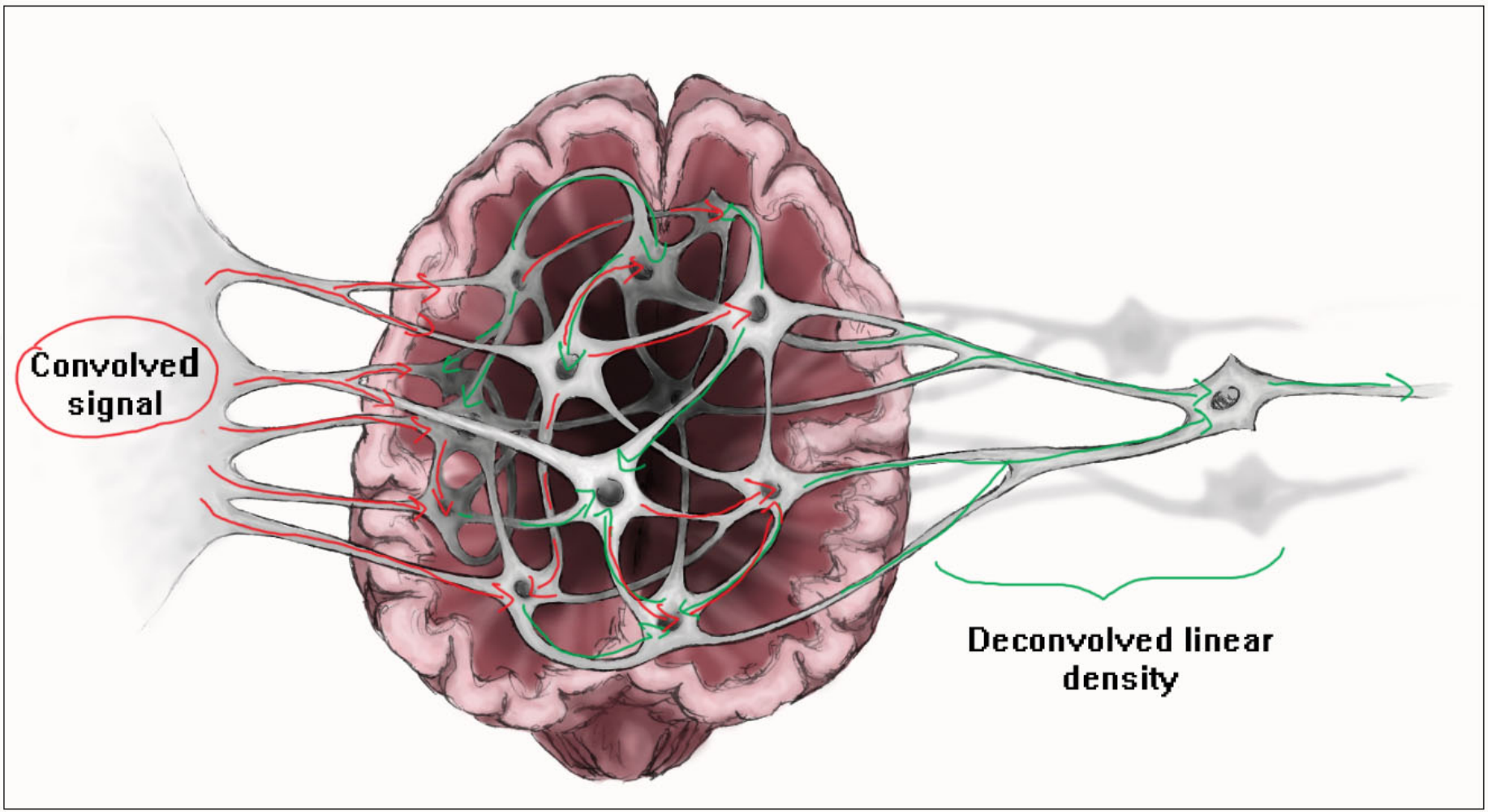
Neural network computations:

1. matrix of weights *M* and input data vector *v*
2. map activation function *f* on all of the product *Mv*

Deconvolution: Reservoir computing

Input / recurrent artificial neural network / output neural layer

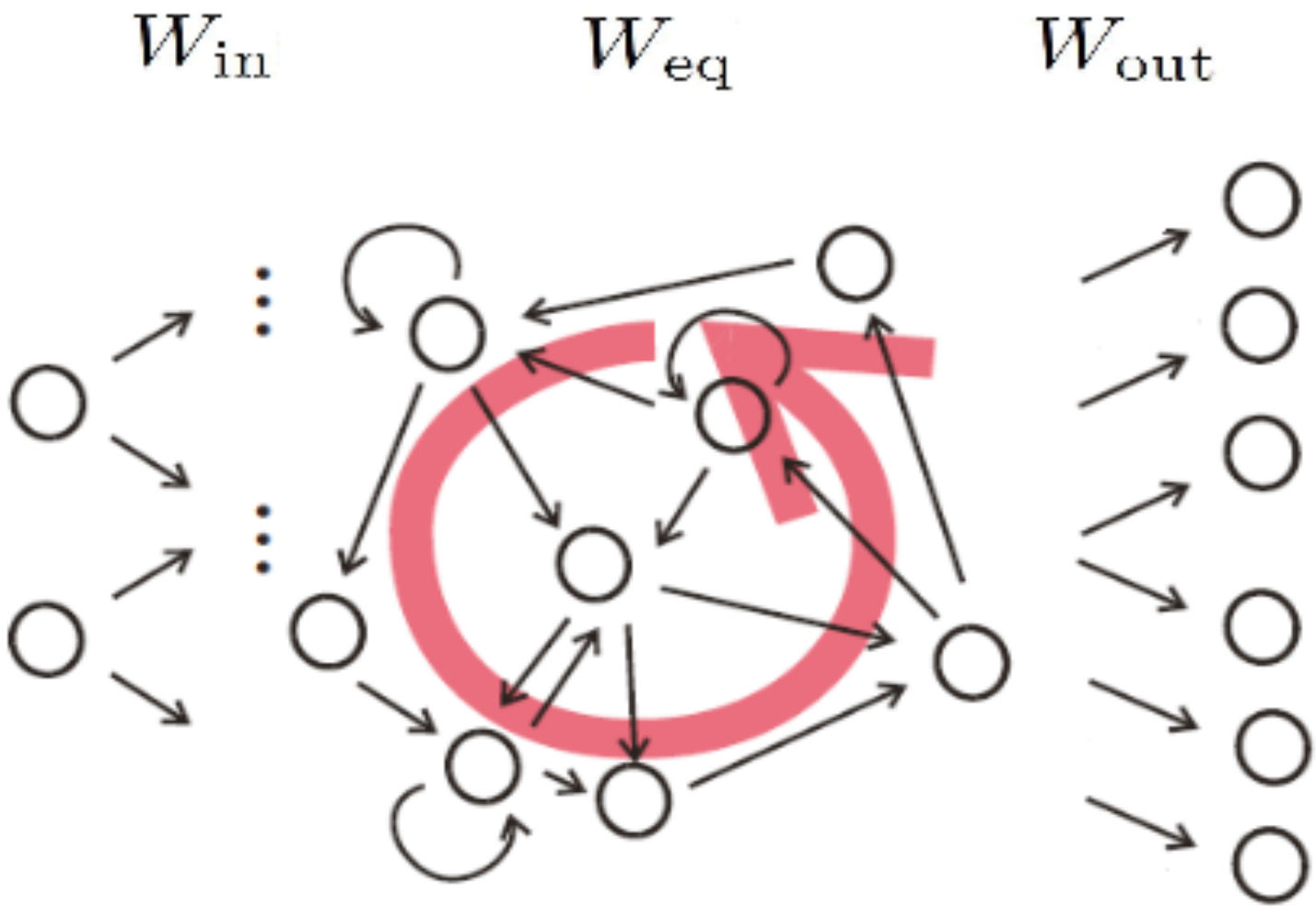
- *fly eye* modularity / one output for each density point.



Deconvolution model: Reservoir computing

Input / recurrent artificial neural network / output neural layer - modularity *fly eye* / one output for each density point.

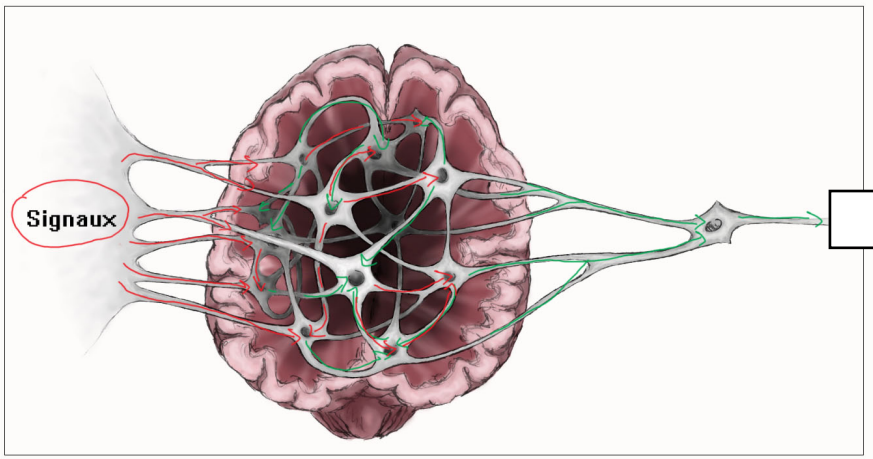
- sparse random directed graph
- `networkx` Python library



Matrix approach: necessary conditions to solve the problem

$$r(B) = \int_{\text{sample}} c(B + Gx)\, s(x)\, dx$$

$$r = Cs \quad \Rightarrow \quad s = C^{-1} r$$



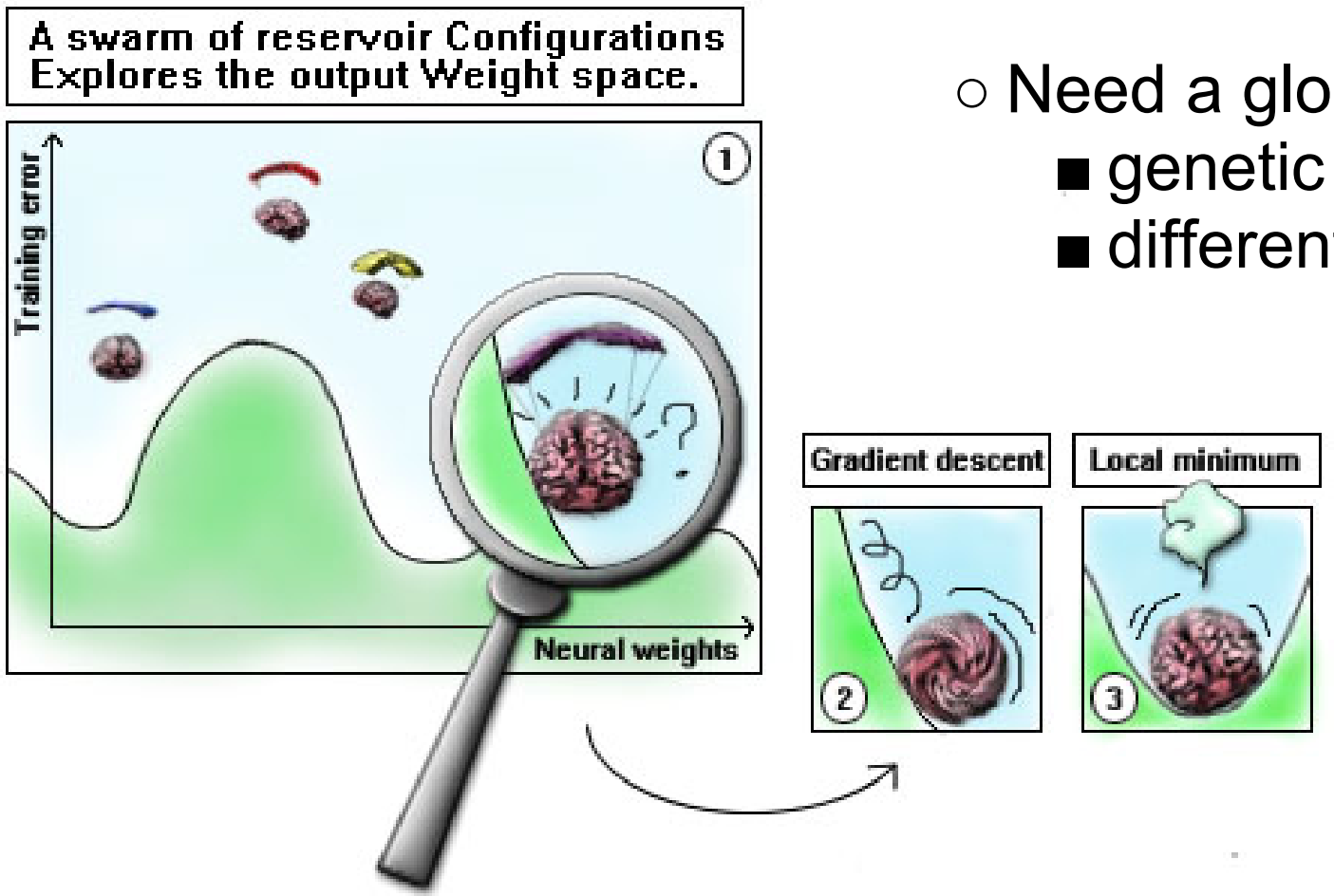
$$s = W_{\text{out}} f\left(W_{\text{eq}} W_{\text{in}} r_{\text{v}}\right)$$

linearization

$$W_{\text{out}} = C^{-1} \left(W_{\text{eq}} W_{\text{in}}\right)^{-1}$$

Finding the minimum

- Optimize the reservoir's weights
  - Many local minima
  - Need a global optimizer:
    - genetic algorithm
    - differential evolution

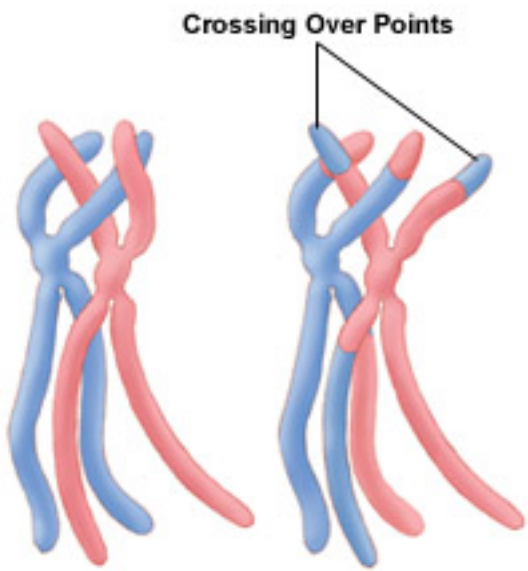


The three fundamental algorithms

- Modular r*eservoir computing*
  - *fly eye* architecture
  - deconvolution point by point

- *Particle swarm*
  - explore reservoir's initial parameter space

- Genetic crossover by differential evolution
  - move the swarm's particles (birds)



Parallelize these algorithms on **GPUs**

Network committee: choose one solution

- Every worker produces a reservoir candidate, with a *fly eye* structure.

- With cross validation : check how a fly-eye reservoir performs on examples it has never seen

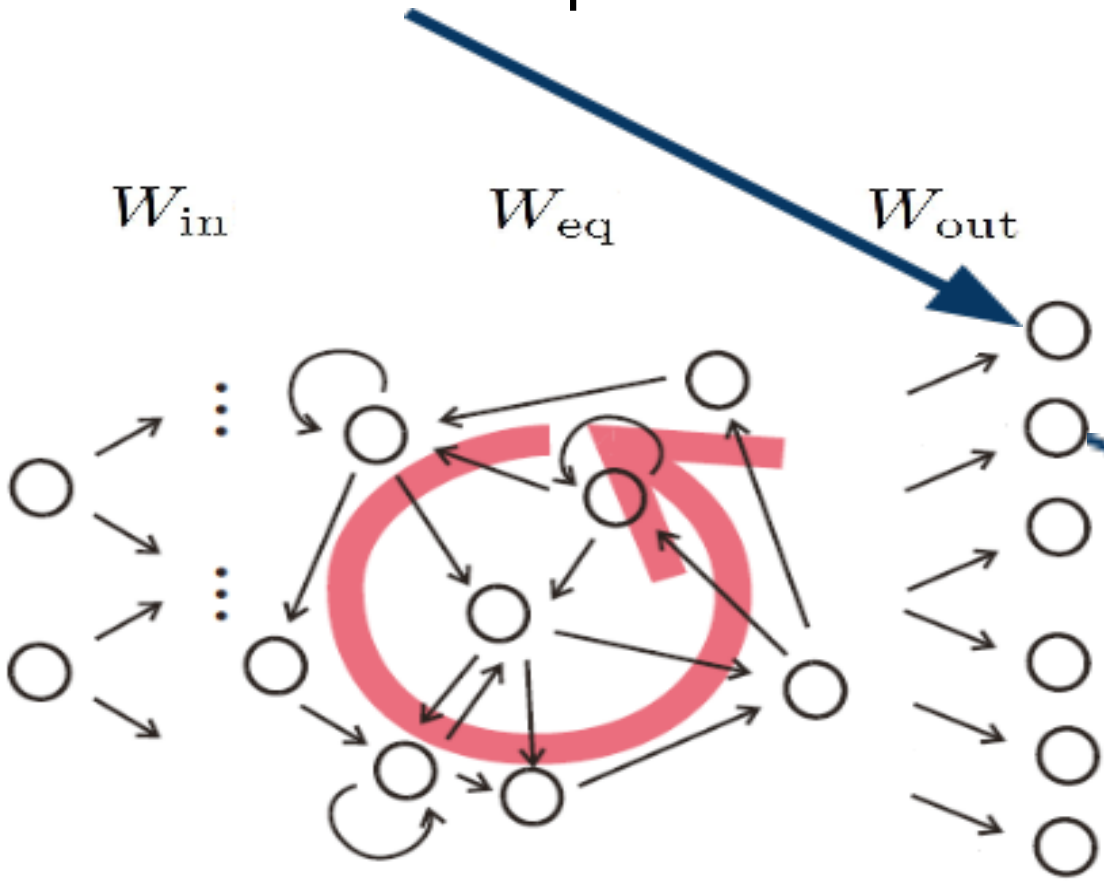
  - Fully parallelized : $S_{i,j} = W_{\text{out},\, i}\, W_{\text{eq}}\, W_{\text{in}}\, R_j$ (`PyCUDA`)

- Build $Q_{k,n}$ with k : reservoir number, *n :* ommatidium number, is the *possibilities* of success of reservoir

  - `argmin` parallel row reduction on $Q_{k,n}$

Fly eye reservoir final layer structure

- The fly eye is made of ommatidia
- Each density point reconstructed is an ommatidia output

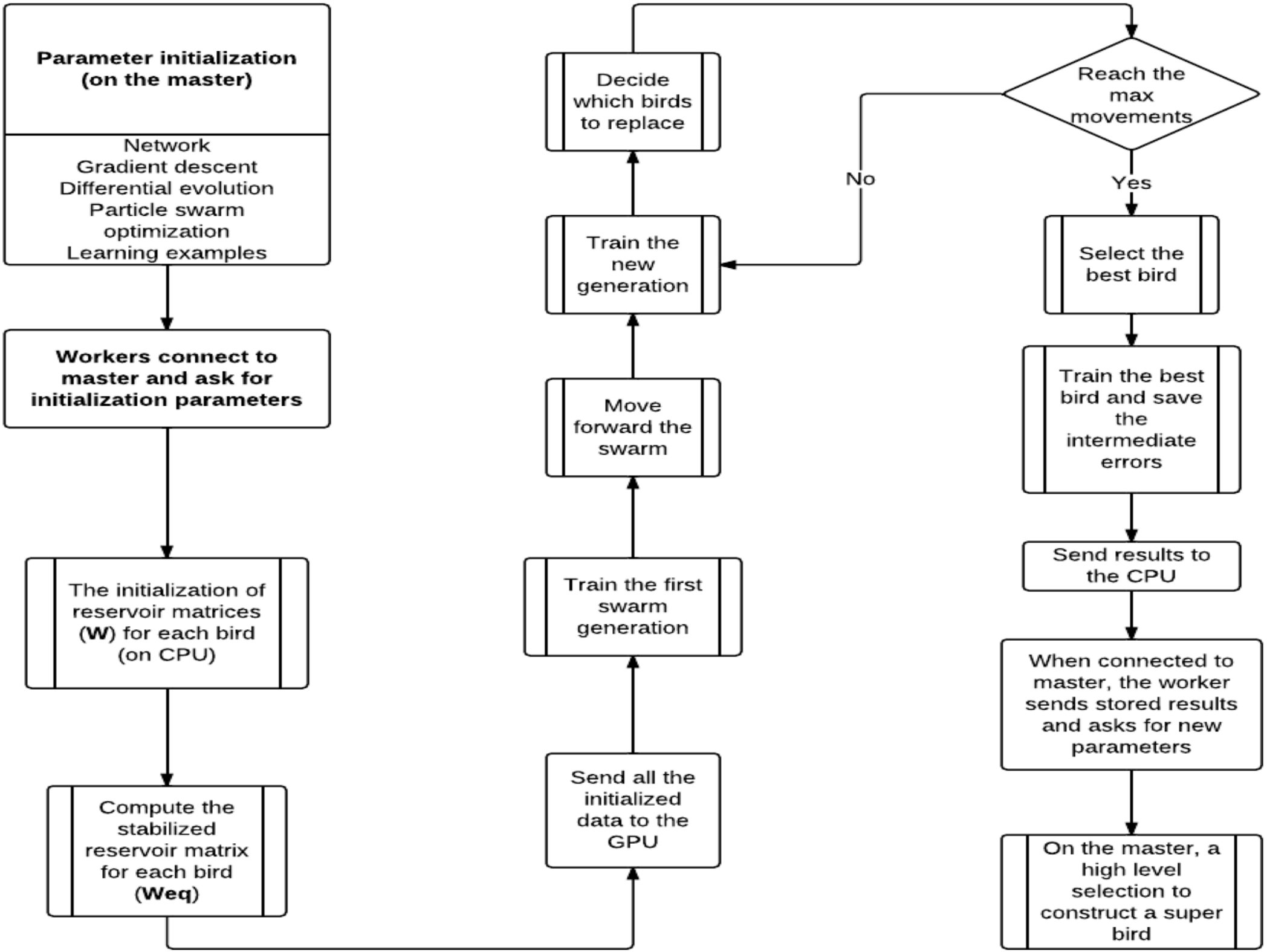


Output computation is parallelized :

$$S_i = W_{\text{out},\,i}\, W_{\text{eq}}\, W_{\text{in}}\, R$$

All S_i computed in parallel

PyCUDA


Parameter initialization (on the master)
Network
Gradient descent
Differential evolution
Particle swarm optimization
Learning examples
Workers connect to master and ask for initialization parameters
The initialization of reservoir matrices (W) for each bird (on CPU)
Compute the stabilized reservoir matrix for each bird (Weq)
Send all the initialized data to the GPU
Train the first swarm generation
Move forward the swarm
Train the new generation
Decide which birds to replace
Reach the max movements
No
Yes
Select the best bird
Train the best bird and save the intermediate errors
Send results to the CPU
When connected to master, the worker sends stored results and asks for new parameters
On the master, a high level selection to construct a super bird

Literate Programming

D. Knuth, 1984 :

"The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. ***He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding****, using a mixture of formal and informal methods that reinforce each other."*

Simple example: add vectors

```
__global__ void sumVectors(float *u, float *v, float *w)

{
  int i = threadIdx.x;
  w[i] = u[i] + v[i];
}
```

3 Defining the kernel that sums the vectors on the GPU

We'll begin with that part, because it's fun. So how do we proceed ? We should first be reminded that vectors are, in practice, lists of numbers, say

$$u = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \quad v = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix}$$

which give

$$u + v = \begin{pmatrix} u_0 + v_0 \\ u_1 + v_1 \\ u_2 + v_2 \end{pmatrix} \tag{1}$$

The reason for this comes from the fact that vectors are not simply lists of numbers, but describe an entity that lives in a vector space and that can thus be written as a linear combination of some basis vectors. We thus have

$$u = u_0\mathbf{e}_0 + u_1\mathbf{e}_1 + u_2\mathbf{e}_2$$
$$v = v_0\mathbf{e}_0 + v_1\mathbf{e}_1 + v_2\mathbf{e}_2$$

which of course leads directly to

$$u + v = (u_0 + v_0)\mathbf{e}_0 + (u_1 + v_1)\mathbf{e}_1 + (u_2 + v_2)\mathbf{e}_2$$

proving the component-wise addition of equation (1).

We should thus take three arguments, `u`, `v` and `w` and add the first two component-wise $w_i = u_i + v_i$, where i is an index that runs from 0 to the length of u minus 1. So because we are building a kernel, we use `C`, so we can map the last equation to `C` using:

2a ⟨*Sum the vector component of* `u[i]` *and* `v[i]` *to give* `w[i]` 2a⟩≡ (2d)

```
w[i] = u[i] + v[i];
```

Now, because each thread on the GPU computes the same kernel, yet has access to its own identity – that is its position in the *block* –, we can define the index i to be the thread `x` coordinate in the block,

2b ⟨*Define* `i` *as the thread* `x` *index* 2b⟩≡ (2d)

```
int i = threadIdx.x;
```

The parallelization is right there: all threads compute the same thing, i.e. implement the same function, but we can vary the argument they crunch thanks to an index that comes directly from each thread's position inside the computational *block*. Thus, if one needs to add vectors with, say, n components, it is straightforward to define a computational block of size `(n,1,1)`, so that each of the n threads works on a different vector component. We should thus make sure that the number of such coordinates corresponds to the length of `u` (which is the same as that of `v` and `u`):

2c ⟨*Define the block size of the threads running the computation as* `len(u)` 2c⟩≡

```
theBlockSize = (len(u),1,1)
```

The kernel is called with three arguments, the vectors u, v and w, which on the `C` side take the form of pointers. The function doesn't return anything, because it modifies w directly in memory. We'll call the kernel `sumVectors`:

2d ⟨*Define the sum kernel on* `u`, `v` *and* `w` 2d⟩≡ (1)

```
__global__ void sumVectors(float *u, float *v, float *w)
{
    ⟨Define i as the thread x index 2b⟩
    ⟨Sum the vector component of u[i] and v[i] to give w[i] 2a⟩
}
```

Lit. Prog. : maîtriser la complexité

E. Dijkstra, 1974

"*...**one hopes that tomorrow's programming languages will differ greatly from what we are used to now**: to a much greater extent than hitherto they should invite us to reflect in the structure of what we write down all abstractions needed **to cope conceptually with the complexity of what we are designing**.*

> *[...]In computer programming our basic building block has an associated time grain of less than a microsecond, but our program may take hours of computation time. I do not know of any other technology covering a ratio of 10^10 or more: the computer, by virtue of its fantastic speed, seems to be the first to provide us with an environment where highly hierarchical artefacts are both possible and necessary. This challenge, viz. the confrontation with the programming task, is so unique that this novel experience can teach us a lot about ourselves. **It should deepen our understanding of the processes of design and creation, it should give us better control over the task of organizing our thoughts.**"*

Reproducible computational research

D. Donoho, 2008 :

"*Scientific Computation is emerging as absolutely central to the scientific method. Unfortunately, it is error-prone and currently immature:* ***traditional scientific publication is incapable of finding and rooting out errors in scientific computation****; this must be recognized as a crisis.* ***Reproducible computational research, in which the full computational environment that produces a result is published along with the article****, is an important recent development, and a necessary response to this crisis.*"

The software

System

- Linux (Ubuntu server & desktop)
- Ext4 /
- **Btrfs** /data*

Development

- vim
- **Python**, PyCUDA, Scikits, CUV
- C, **Cython**
- git
- **Noweb** (literate programming)

Analysis

- Scipy, **iPython**
- **Sage**, Mathematica
- Mayavi2, Asymptote
- Google Docs

Sage and Cython

- Example : simple Monte-Carlo integrator

```
mcInt = lambda f,a,b,n:

(b-a)/n*sum([f(x) for x in [(b-a)*random()+ a for _ in range(n)]])
```

```
from random import random



def mcInt2(f,a,b,n):
        s = 0
        for _ in range(n):
            s += f((b - a) * random() + a)
        s *= (b - a) / float(n)
    return s
```

```
%cython
from random import random


cdef float f(float x):
        return x*x


def cython_mcInt2(float a,float b,int n):
    cdef float s = 0
    for j in range(n):
        s += f((b - a) * random() + a)
    s *= (b - a) / float(n)
    return s
```

The System

- Linux
  - rules HPC
  - CERN: 1GB/s of data, computing grid of 100,000 computers
  - so easy to play with and adapt to needs

- BTRFS
  - instant RAID0, excellent performance
  - no problem for static storage/read
  - git feeling

- Ext4
  - well tested, reliable for OS

Cython


```
mcInt = lambda f,a,b,n:(b-a)/n*sum([f(x) for x in [(b-a)*random()+a for _ in range(n)]])
```

```
time s=[mcInt(lambda x: x*x,0,1,1000) for _ in range(1000)]
```

```
Time: CPU 11.82 s, Wall: 11.84 s
```

```
from random import random

def mcInt2(f,a,b,n):
    for _ in range(1000):
        s = 0
        for _ in range(n):
           s += f((b - a) * random() + a)
        s *= (b - a) / float(n)
    return s
```

```
time mcInt2(lambda x: x*x,0,1,1000)
```

```
0.33307028410372003
Time: CPU 12.14 s, Wall: 12.16 s
```

```
%cython
from random import random

cdef float f(float x):
       return x*x

def cython_mcInt2(float a,float b,int n):
    cdef float s = 0
    for j in range(n):
        s += f((b - a) * random() + a)
    s *= (b - a) / float(n)
    return s
```

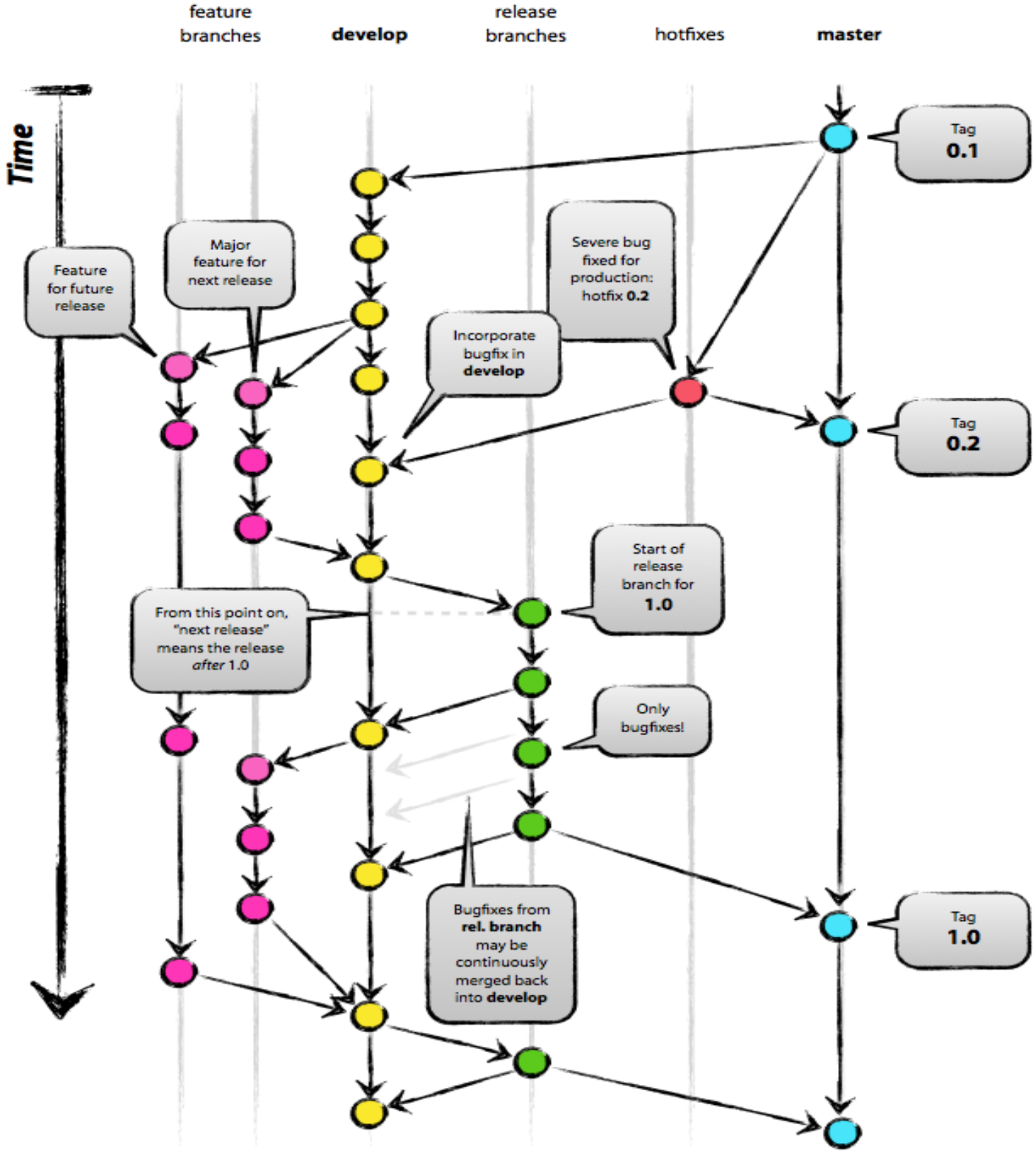
__work_sag...3_code_sage41_spyx.c __work_sag...ode_sage41_spyx.html

```
time sc=[cython_mcInt2(0,1,1000) for _ in range(1000)]
```

```
Time: CPU 0.12 s, Wall: 0.12 s
```

Git versioning

- Decentralized but centralized
- The main branches
  - master
  - develop
- Supporting branches
  - feature branches
  - release branches
  - hotfix branches



Source:
Vincent Driessen

The HPU4Science cluster

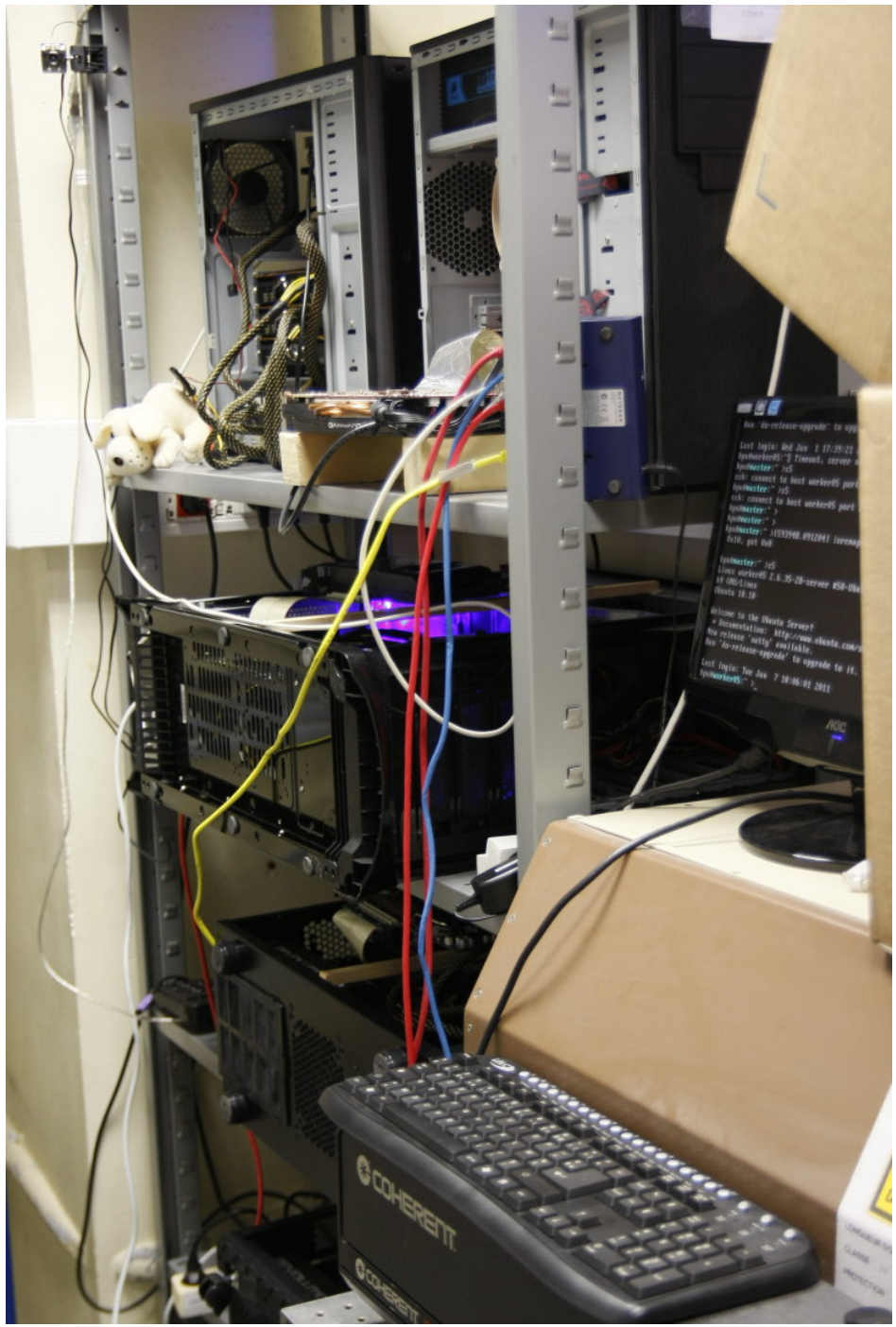
The master :

- centralize data, 20TB
- frame calculations
- combine all the candidates

6 workers :

- 3 to 7 GPUs
  - 3 to 7 HPUs
- all run the same algorithm
- save intermediate data

Linux OS (Ubuntu Server)

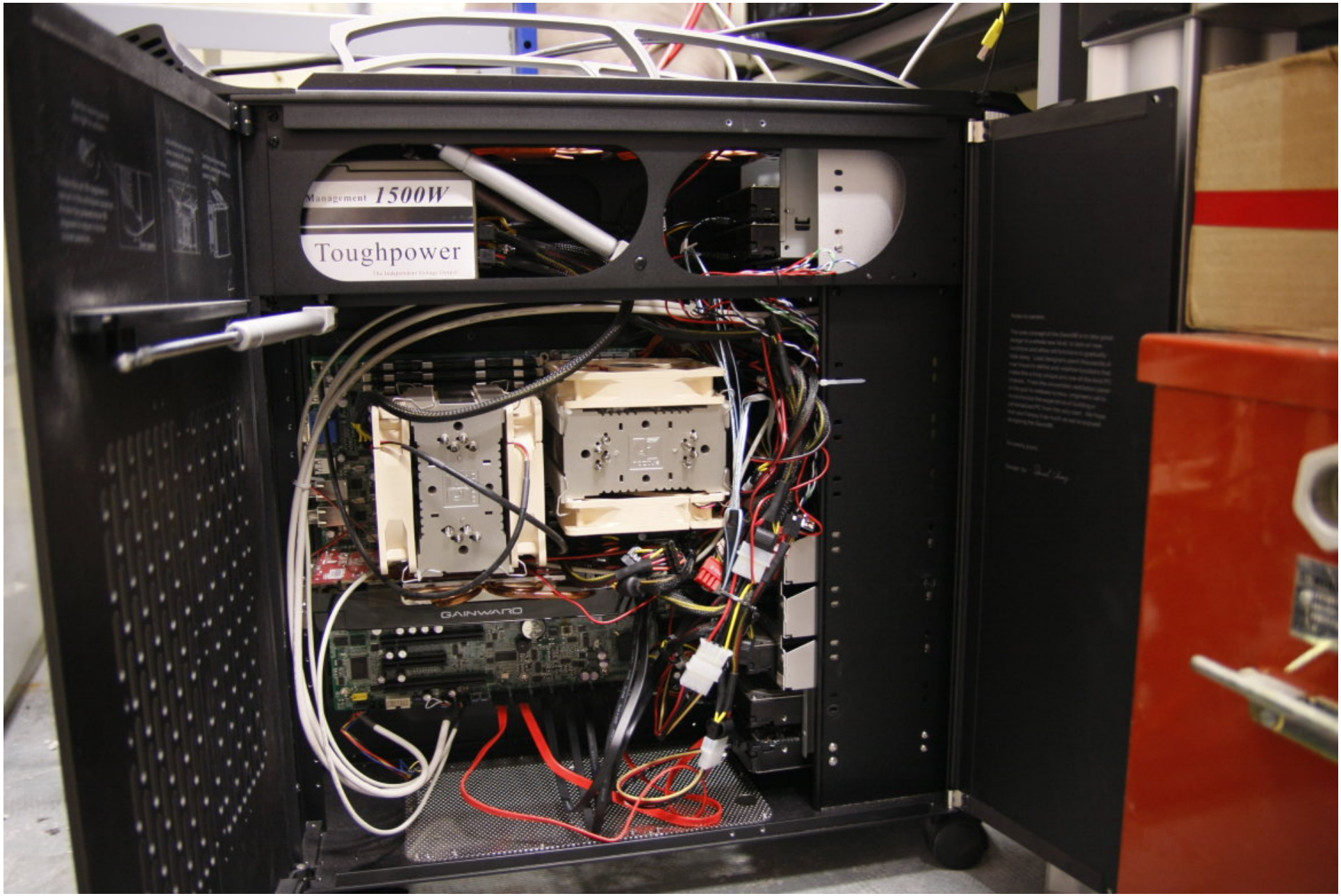


In the beginning...

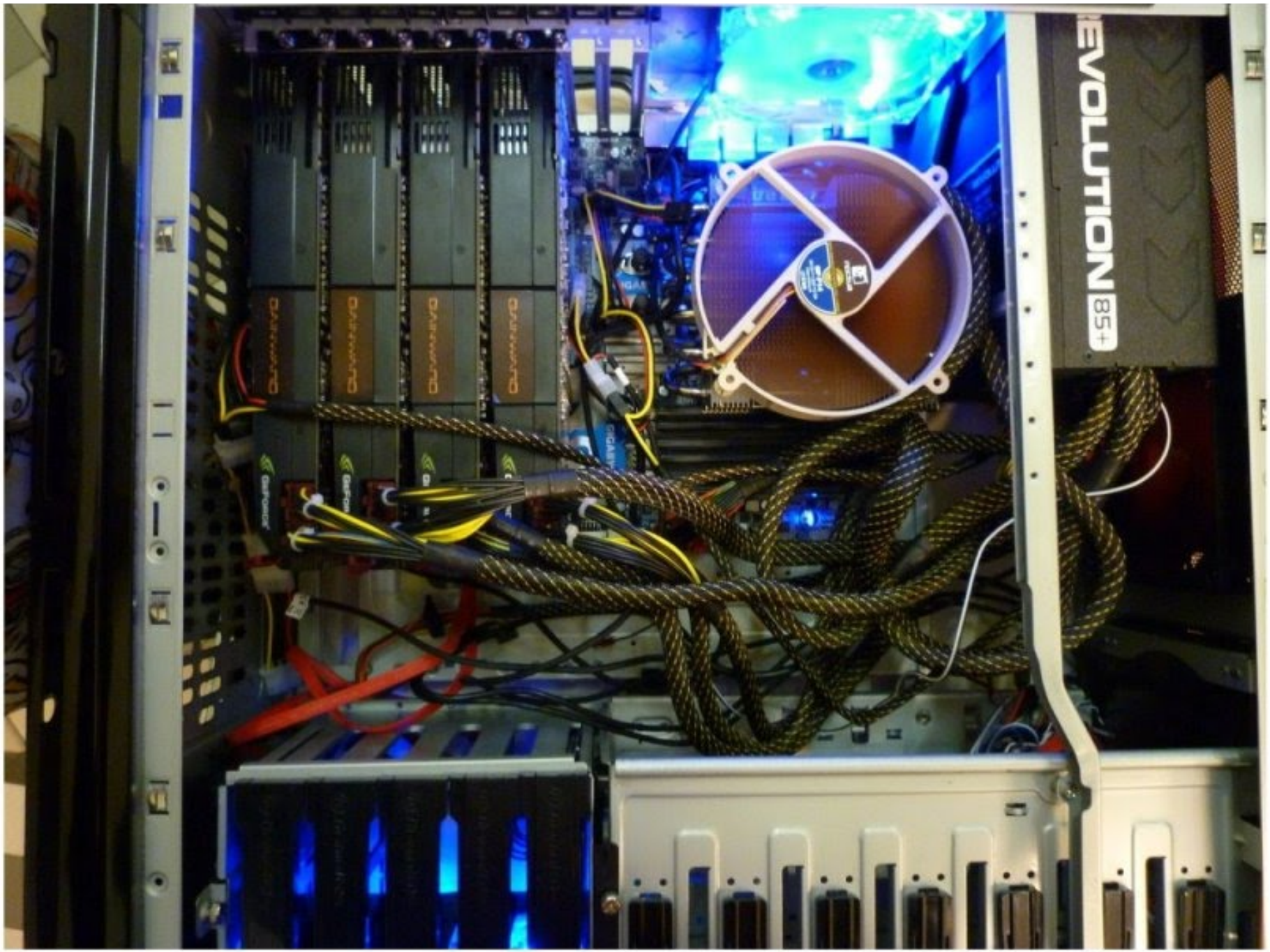
The GPU cards

Choice NVIDIA : GTX285, GTX295, GTX480, GTX580, GTX590

Master : data centralization

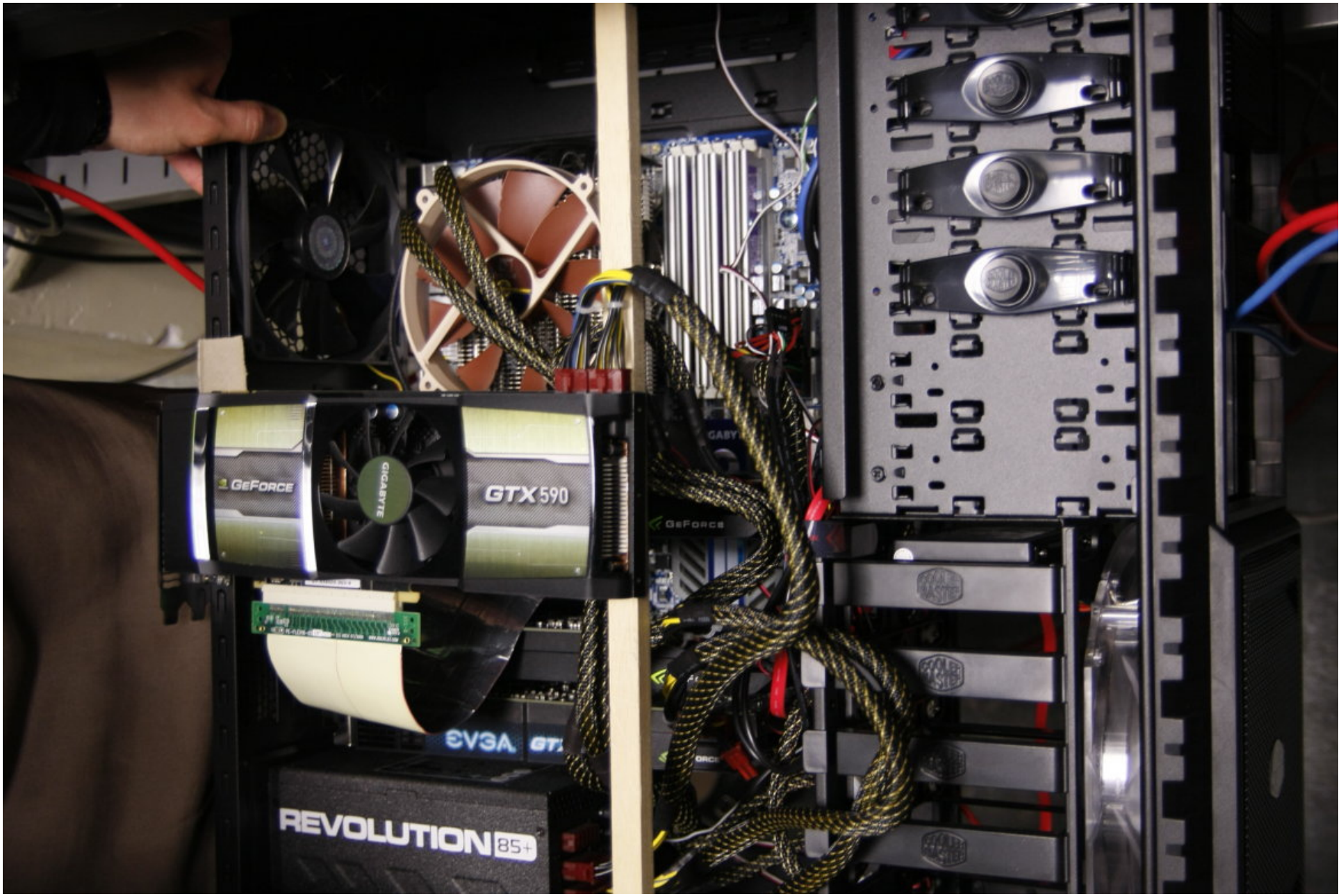


The worker03



The motherboard : Gigabyte GA-X58A-UD9

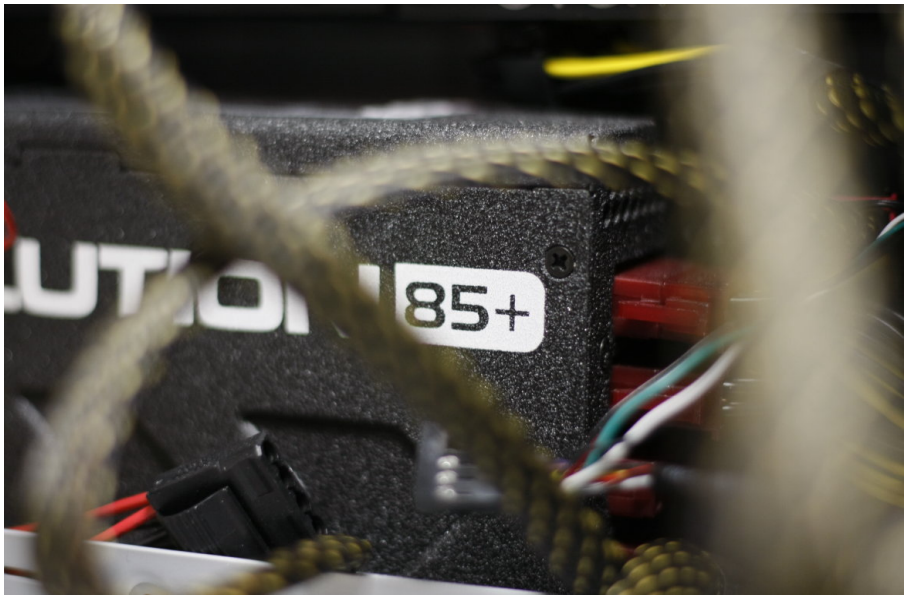
The worker05



Problems

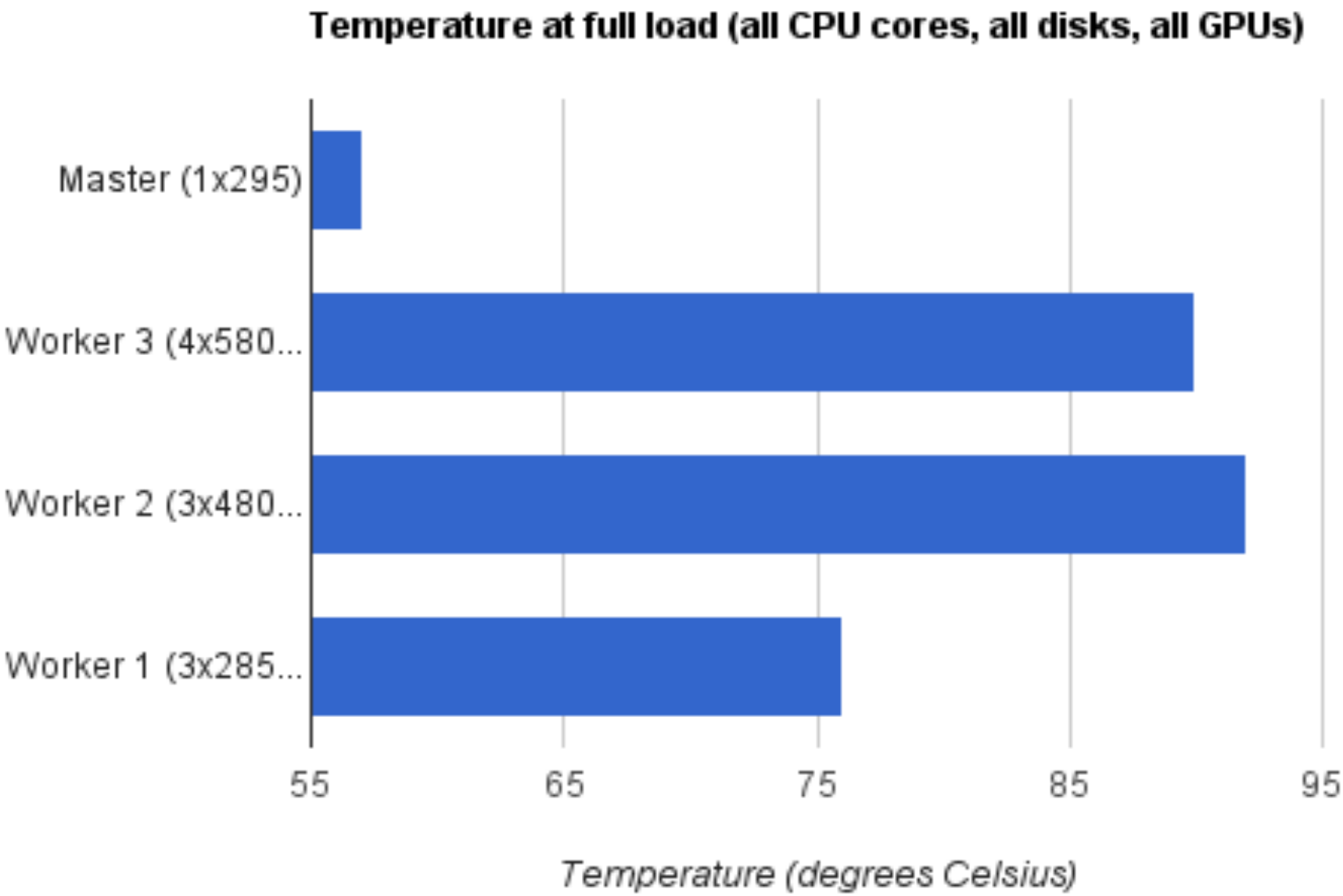
- temperature
  - risers
  - air circulation

- electric overconsumption
  - calculate the accurate consumption

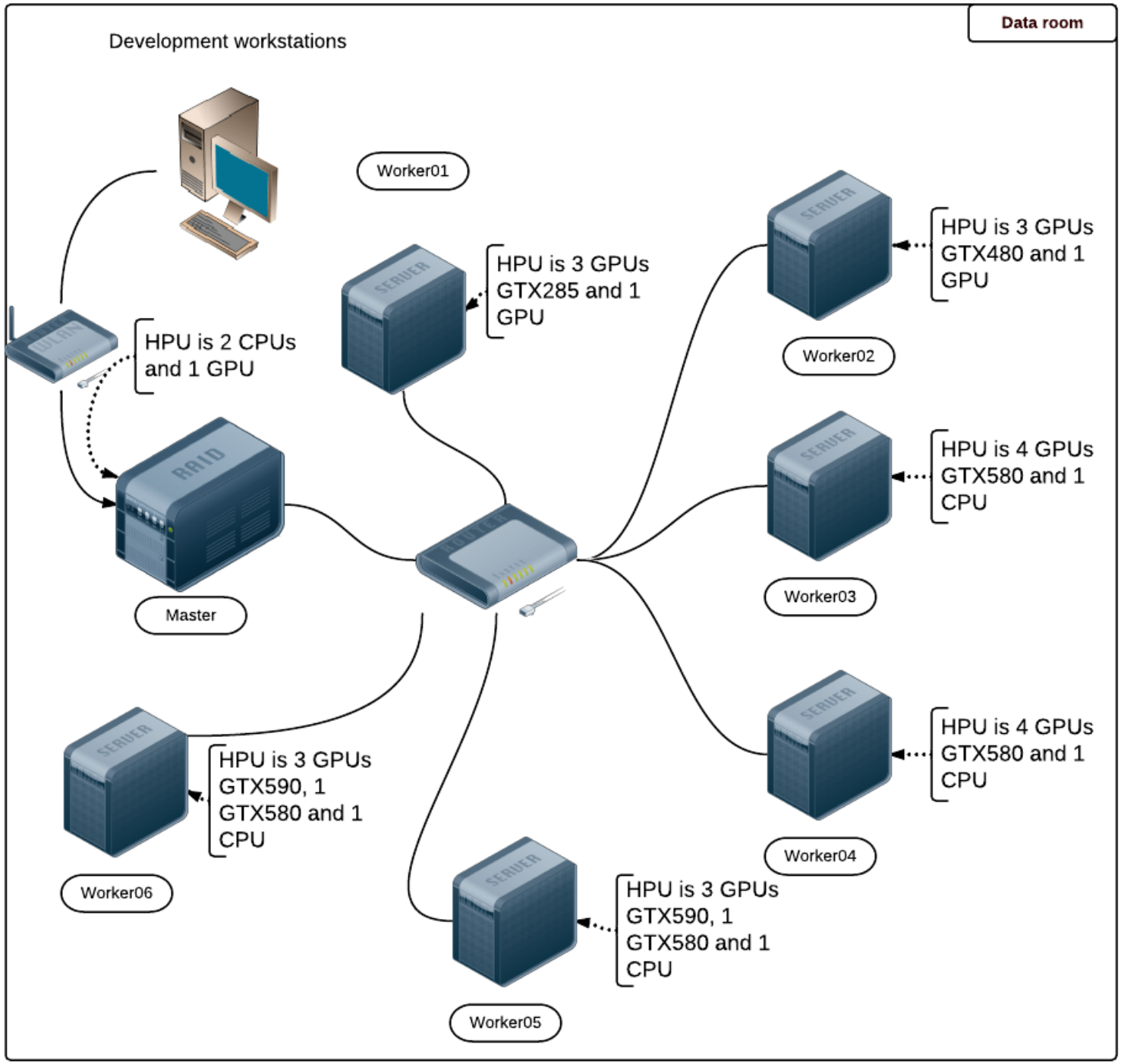


- defective components
  - extensive tests

Problem 1 : temperature



Distributed architecture

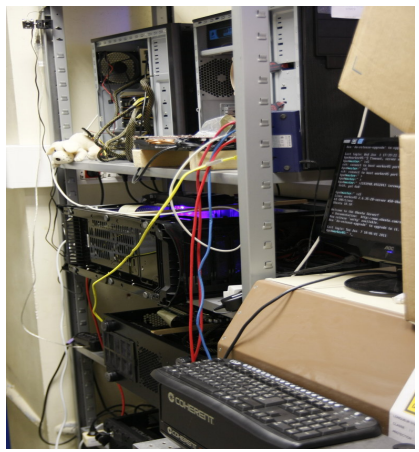


Software components for non-computational cluster activity

- Data storage/retrieval
  - Python `dictionaries` with `cPickle`
  - `Karrigell` server for analytical/visualization requests
- Communication worker/master
  - `multiprocessing` module
  - only the Client requests actions
  - update code
- Instant snapshots of activity

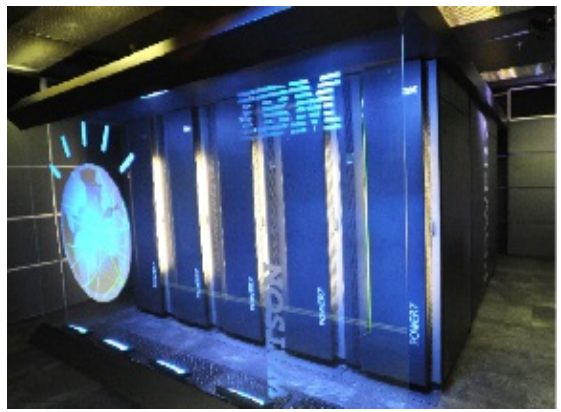
Compare HPU4Science/Watson

HPU4Science



vs

Watson IBM



cost: €30,000

power: 35 TFLOPS

power/€: 1,170 MFLOPS/€

storage: 20 TB

cost: approx. €22 million

power: 80 TFLOPS

power/€: 3.7 MFLOPS/€

storage: 20 TB

Short-term evolution (6 months)

- test OpenCL with PyOpenCL
- open source the code (CeCILL ?)
- move to GTX670 cards
- continue developing other applications
  - with machine learning
  - or for classical processing methods
- golden rule: stay close to "hot spring vents"

May Python help us find the Oracle !

The HPU4Science team

The Core

Yann Le Du, CNRS engineer, launched the HPU4Science project in 2009.

Mariem El Afrit, joined the project early in 2010 as an undergrad, through an internship financed by CNES and then by ANR ORIGINS/ENUSIM. She is now headed for a PhD which should begin in fall 2011 or beginning 2012.

The Asthenosphere

Laurent Binet, Chimie ParisTech associate professor, is an EPR and material science expert at LCMCP/Chimie ParisTech and a main contributor to the use of EPR in exobiology.

Didier Gourier, Chimie ParisTech professor, is an EPR expert and material science expert at LCMCP/Chimie ParisTech and he initiated the use of EPR of carbon in exobiology.

Hervé Vezin, CNRS research director, is continuous and pulsed wave EPR expert at the LASIR in Lille, and an old time close collaborator of the EPR group at the LCMCP. Hervé is the ANR ENUSIM/ORIGIN project leader, and is equipped with bleeding edge Bruker EPR spectrometers, including an imageing continuous wave and pulsed wave EPR device.

The HPU4Science team

The Crust

Yves Frapart, CNRS engineer, is the EPR imageing team leader at Paris 5, and his lab is equipped with multiple Bruker EPR spectrometers, including imageing ones. He has the first Bruker EPR imageing spectrometer that was installed in France, and at the time the most powerful one worldwide.

The Lithosphere

Jean-François Engrand, Paris 6 technician, is the man who knows all about the non computer hardware, and who always finds a solution to all the problems that lurk in those dark regions.

The Biosphere

Frédéric Mentink, a PhD student at LCMCP/Chimie ParisTech working on quantum information and Electron Paramagnetic Resonance, is an apt photographer and is working on giving us the best shots while pondering on the potentialities of compressed sensing.

Diane Robert-Magnenan, philosophy undergrad and artist, joined the project in 2009, and has contributed in many artistic ways, including the drawings for the third *Ars Technica* paper to be published at the end of May.

http://hpu4science.org

Yann Le Du, Mariem El Afrit *et al*.

ANR ENUSIM/ORIGIN 2009-2012, CNES RTS-Exobiologie

Chimie Paris

ParisTech

Communications

Scientific communications

- invited conference Europython 2011, june
- seminar Aristote, Polytechnique, june 2011
- seminar CNES/IDRIS, assimilation methods, june 2011
- conference EuroScipy 2011, august
- seminar JDEVLOG 2011, september

Communications *Information Technology*

- series of three articles in Ars Technica (avril/mai 2011) : *High Performance Computing on Gamers PC*