High-performance computing using Python, GPUs & Modular approach to crystallographic software development

The European Synchrotron

Vincent Favre-Nicolin

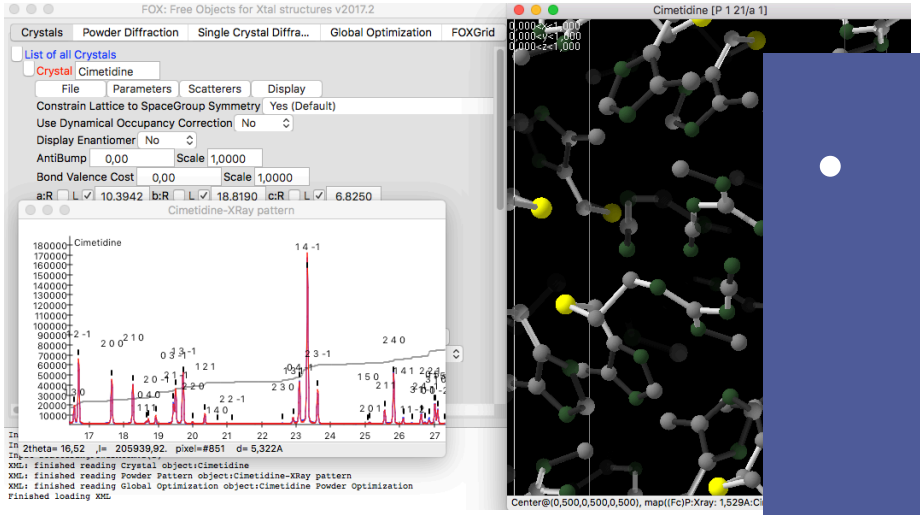
X-ray NanoProbe group, ESRF

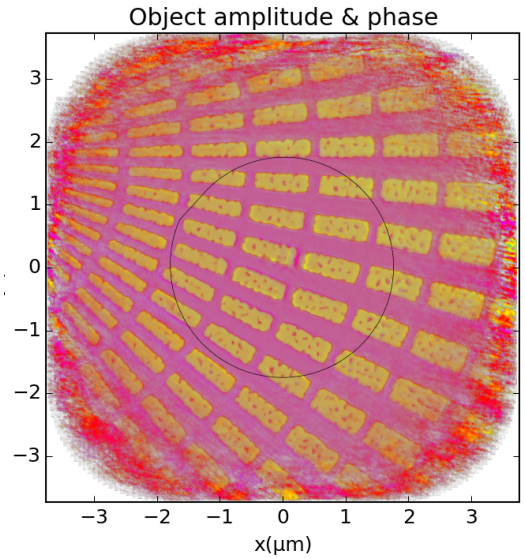
OUTLINE OF MY TALK

- Modular, object-oriented programming
- Efficient programming in Python
- GPU computing from Python
- A few words about Cloud computing

```
class CDI:
    """ Reconstruction class for two c
    """

    def __init__(self, iobs, support,
        """
        Constructor. All arrays are as

        Args:
            iobs: 2D/3D observed diffr
                  Assumed to be correc
                  Dimensions should be
            obj: initial object. If No
                 with uniform random 0
                 The data is assumed t
            support: initial support i
            pixel_size_object: pixel s
            lambdaz: wavelength * dist
            mask: mask for the diffrac
        """
        self.iobs = iobs.astype(np.float32)

        # TODO: estimate support from iobs
        self._support = support.astype(np.int8)

        if obj is None:
            self._obj = (np.random.uniform(0, 1, iobs.shape)
                         * np.exp(1j * np.random.uniform(0, 2 * np.pi, iobs.shape)) * self._support).astype(np.complex64)
        else:
            if copy_obj is True:
                self._obj = obj.copy().astype(np.complex64)
            else:
                self._obj = obj.astype(np.complex64)
```

MAIN COMPUTING PROJECTS





- *Ab initio* structure solution from powder diffraction:

  FOX: http://fox.vincefn.net

- Coherent diffraction Imaging using GPU

PyNX:

- https://software.pan-data.eu/software/102/pynx
- http://ftp.esrf.fr/pub/scisoft/PyNX/

PROGRAMMING STYLES: FLAT

As simple as it gets

```
import numpy as np

# Linear programming
l, k, h = np.mgrid[-10:11, -10:11, -10:11]
n_atoms = 1000
x = np.random.uniform(0, 1, size=n_atoms)
y = np.random.uniform(0, 1, size=n_atoms)
z = np.random.uniform(0, 1, size=n_atoms)

fhkl = np.empty(h.shape, dtype=np.complex)
for i in range(h.size):
    fhkl[i] = np.exp(2j * np.pi * (h.flat[i] * x + k.flat[i] * y + l.flat[i] * z)).sum()
```

PROGRAMMING STYLES: FUNCTIONS

```
...
 # Function
def calc_f_hkl(x, y, z, h, k, l):
    assert (x.shape == y.shape == z.shape)
    fhkl = np.empty(h.shape, dtype=np.complex)
    for i in range(h.size):
        fhkl[i] = np.exp(2j * np.pi * (h.flat[i] * x + k.flat[i] * y + l.flat[i] * z)).sum()
    return fhkl


fhkl = calc_f_hkl(x, y, z, h, k, l)
```

You can:

- **Re-use the function**
- **Optimize it**

PROGRAMMING STYLES: OBJECT-ORIENTED

```
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == k.shape)
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dtype=np.complex)
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi * (self.h.flat[i] * x + self.k.flat[i] * y + self.l.flat[i] * z)).sum()
```

You can:

- **Re-use the class**
- **Optimize it**
- **Use instance as storage for result**

```
rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()
```

OBJECT-ORIENTED: CLASSES

```
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == k.shape)
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dtype=np.complex)
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi * (self.h.flat[i] * x + self.k.flat[i] * y + self.l.flat[i] * z)).sum()


rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()
```

Class definition

Initialization function

Attribute (data member)

Protected attribute _

Method (member function)

class instance

WHY OBJECT-ORIENTED

```
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == k.s
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dty
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi *


rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()
```

- **Separate:**
  - Code (calculations)
  - Data (storage)
- **Abstract layer:**
  - No direct access to data
  - Methods must keep their signature (API)
  - Calculation/storage can be later optimized:
    - Limited or full-space hkl (FFT vs direct sum)
    - CPU or GPU ?
- **Better code organization**
- **Easy to document**
- **Easy to share**

WHY NOT OBJECT-ORIENTED

```
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == k.s
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dtype=np.complex)
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi * (self.h.flat[i] * x + self.k.flat[i] * y + self.l.flat[i] * z)).sum()


rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()
```

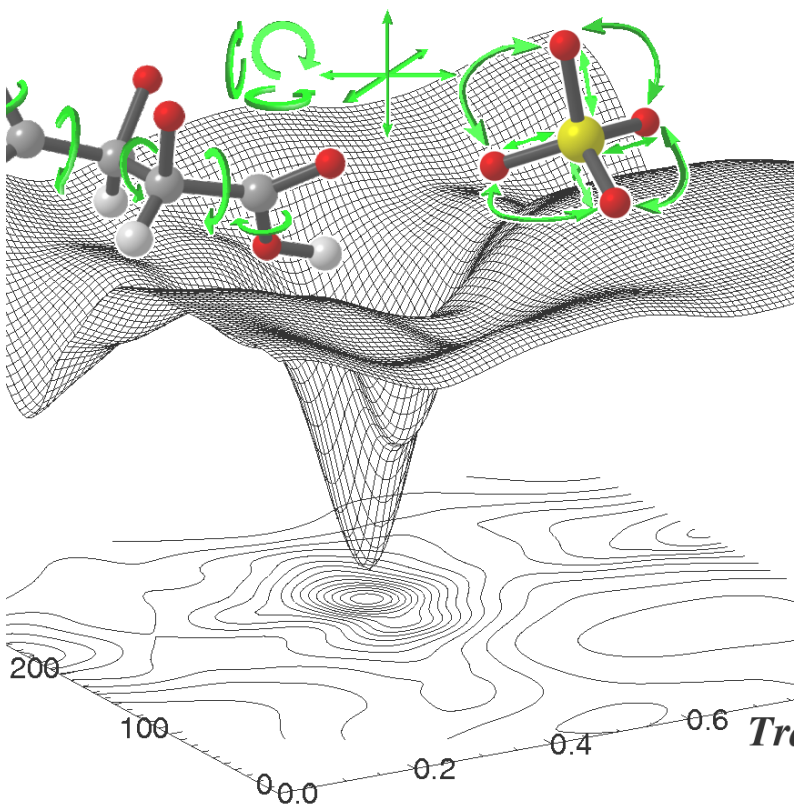
- It's complicated
- It's longer to write
- Simple code don't need objects
- Why document, it's trivial ?

PYTHON CLASSES: PROTECTED ATTRIBUTES

```
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == k.s
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dty
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi *


rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()    # YES
fhkl = _fhkl              # NO- may trigger the end of the world
```

- All attributes can be accessed in Python
- Members with a leading _ are a programmer's hint that the method or data may be changed in the future
- Use public access methods
- Use _ for data/methods which may change

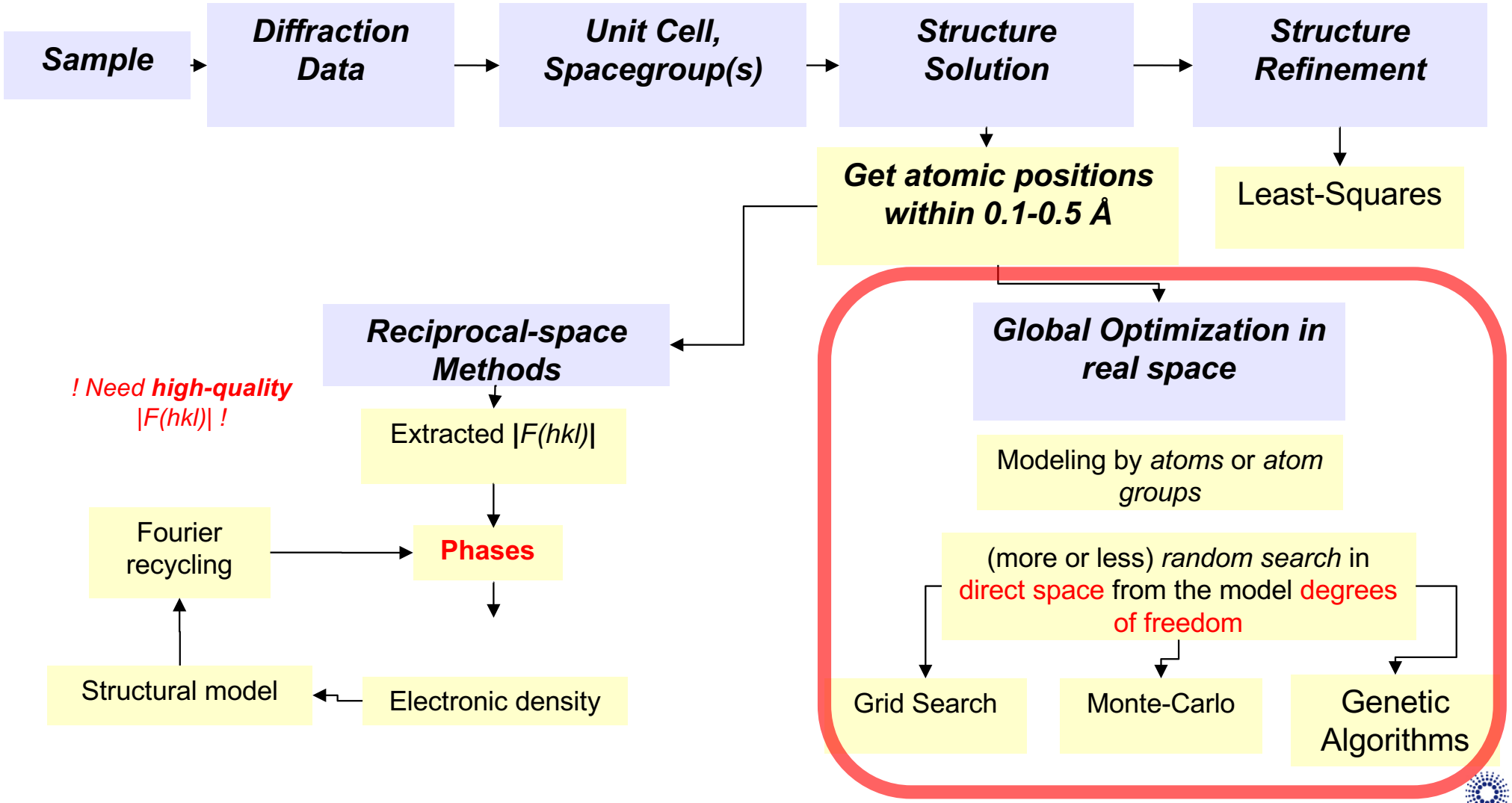
OO PROGRAMMING: BIG OR SMALL CLASSES



For any crystallographic computing, you can choose:

- Large classes with many methods
- A large number of small classes easily re-used

EXAMPLE PROBLEM: STRUCTURE SOLUTION



BIG CLASSES: REVERSE MONTE-CARLO

Parametrization
-> Degrees Of Freedom
(atomic positions, torsion angles, ..)

starting configuration → random change of parameters → evaluation of the new configuration: ***Cost (C)*** → is configuration better ? $C_n < C_{n-1}$

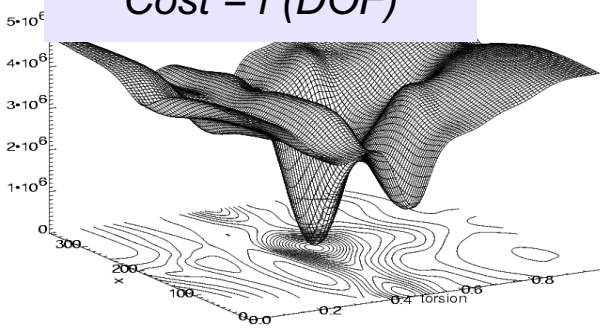
yes → keep configuration → random change of parameters

no → keep configuration with probability: $P = e^{\frac{-\Delta C}{T}}$ → random change of parameters

Temperature of the algorithm

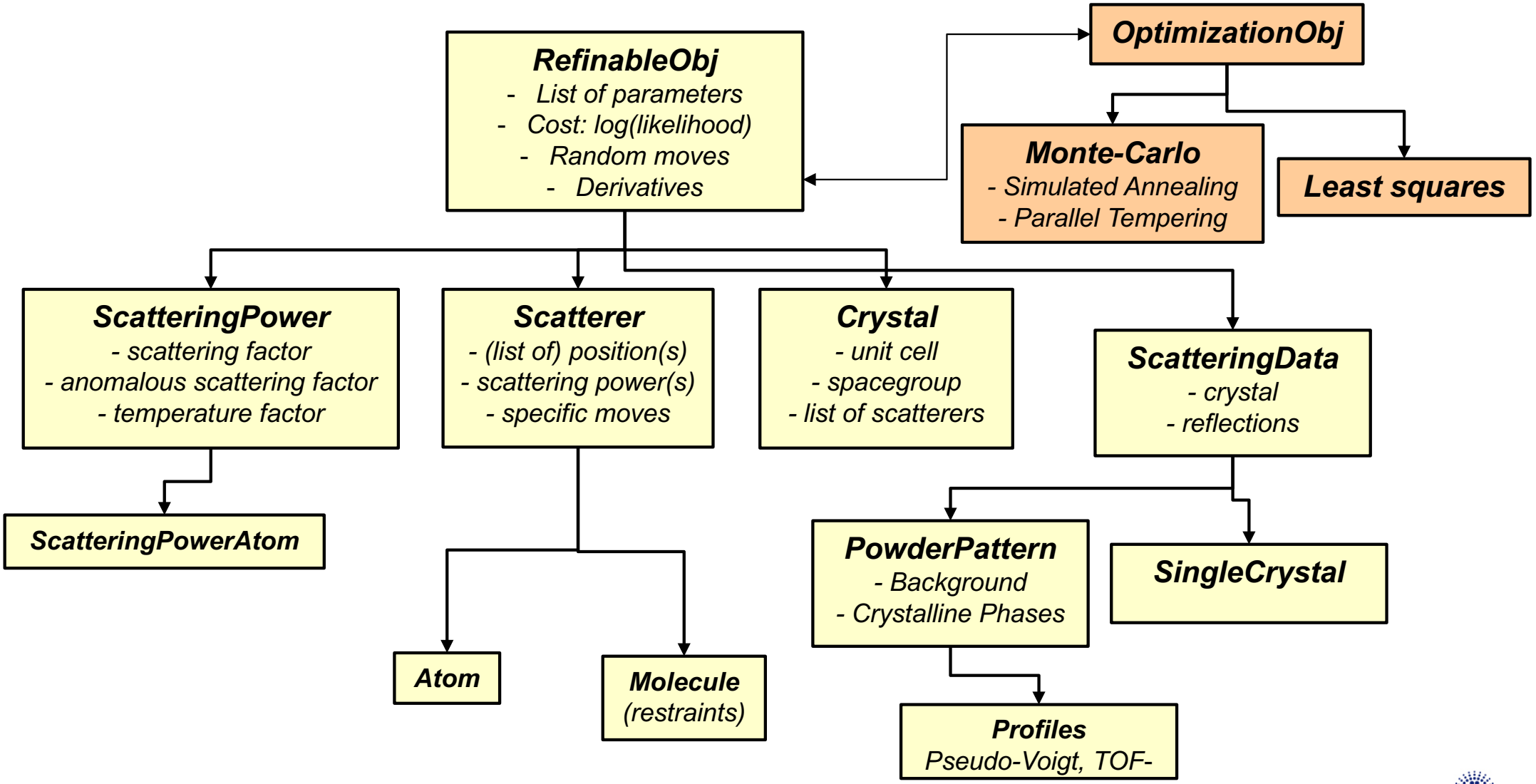
*Generate a distribution of configurations following **Boltzmann**'s law*

Hypersurface
Cost = f (DOF)



Need two classes:
- **Object to optimize**
- **Algorithm**

APPROACH 1: BIG CLASSES / OBJCRYST++



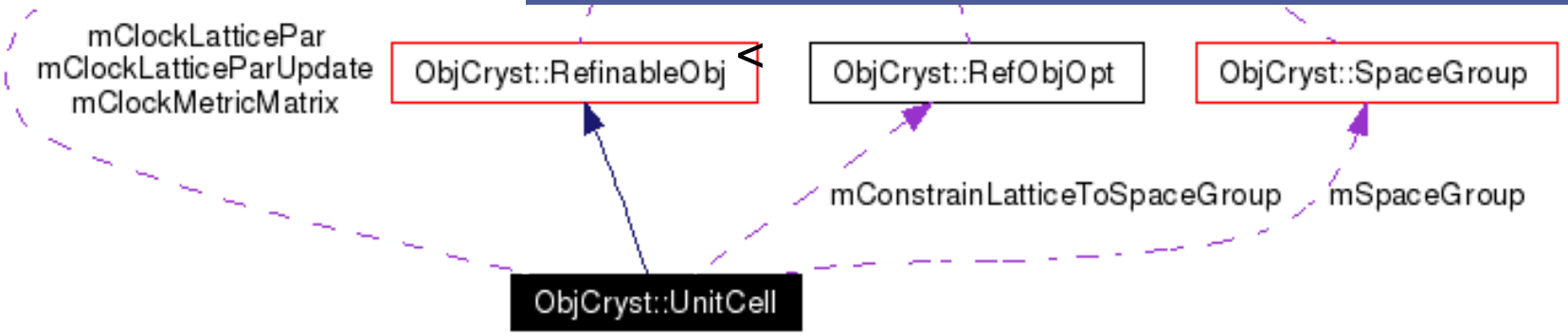
OBJCRYST++ API

Public Member Functions

	ReflectionProfile ()
	ReflectionProfile (const **Refl**
virtual	**~ReflectionProfile** ()
virtual **ReflectionProfile** *	**CreateCopy** () const=0
virtual CrystVector_REAL	**GetProfile** (const CrystVector
	Get the reflection profile.
virtual REAL	**GetFullProfileWidth** (const R
	Get the (approximate) full profile width

See API from:
http://fox.vincefn.net

- **Molecule**
- **UnitCell**
- **LeastSquares**



OBJCRYST++ API: AVOID RE-COMPUTATIONS

CrystVector_REAL	**mIhklCalcVariance** Variance on computed intensities for all reflecti
vector< **ReflProfile** >	**mvReflProfile** Reflection profiles for ALL reflections during the
std::map< **RefinablePar** *, vector< CrystVector_REAL > >	**mvReflProfile_FullDeriv** Derivatives of reflection profiles versus a list of
vector< pair< unsigned long, CrystVector_REAL > >	**mIntegratedProfileFactor** For each reflection, store the integrated value c
RefinableObjClock	**mClockIntegratedProfileFactor** Last time the integrated values of normalized p
map< const **ScatteringPower** *, CrystVector_REAL >	**mvScatteringFactor** Scattering factors for each **ScatteringPower**, as vectors v
map< const **ScatteringPower** *, CrystVector_REAL >	**mvRealGeomSF** Geometrical Structure factor for each **ScatteringPower**, a
map< const **ScatteringPower** *, CrystVector_REAL >	**mvImagGeomSF**
map< **RefinablePar** *, map < const **ScatteringPower** *, CrystVector_REAL > >	**mvRealGeomSF_FullDeriv**
map< **RefinablePar** *, map < const **ScatteringPower** *, CrystVector_REAL > >	**mvImagGeomSF_FullDeriv**

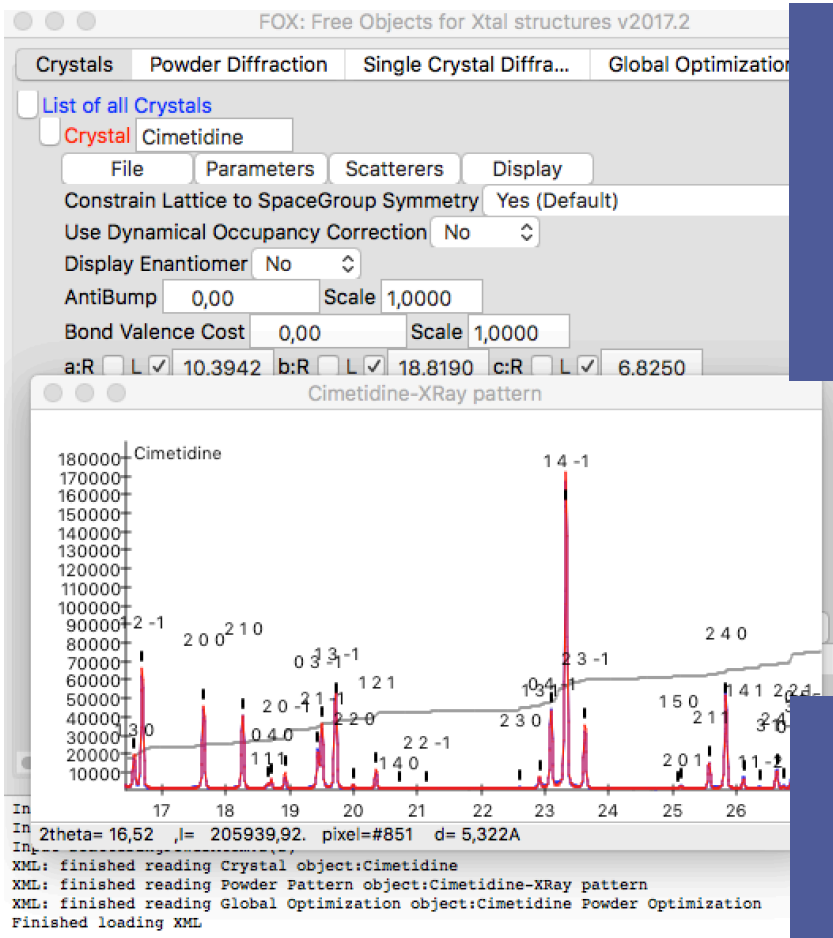
Powder pattern calculation:

- Did crystal structure change ?
  - No: re-use structure factors
  - Yes:
    - **Re-compute structure factors**
    - **Re-use table of atomic scattering factors**
- Did unit cell change ?
  - No: keep reflection positions
  - Yes: re-compute reflection positions
- Did reflection profiles change ?
  - No: re-use reflection profiles
  - Yes: re-compute reflection profiles

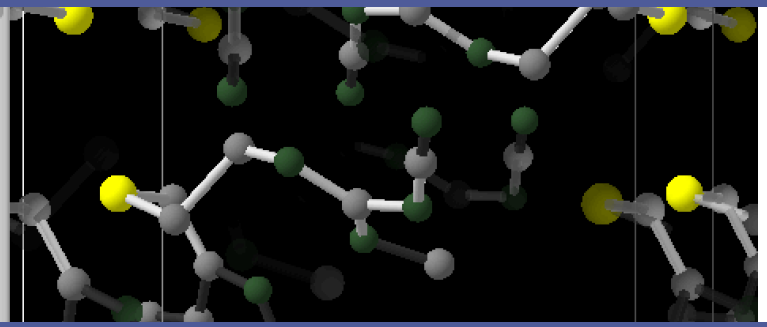
Note:

- *top objects know nothing about derived objects implementation – all is hidden behind OO API*
- *Optimization algorithms know nothing about crystallography !*

FOX/OBJCRYST++ SPEED



- 20 independent atoms, 100 reflections
- 10^4 to 5.10^4 configurations/s



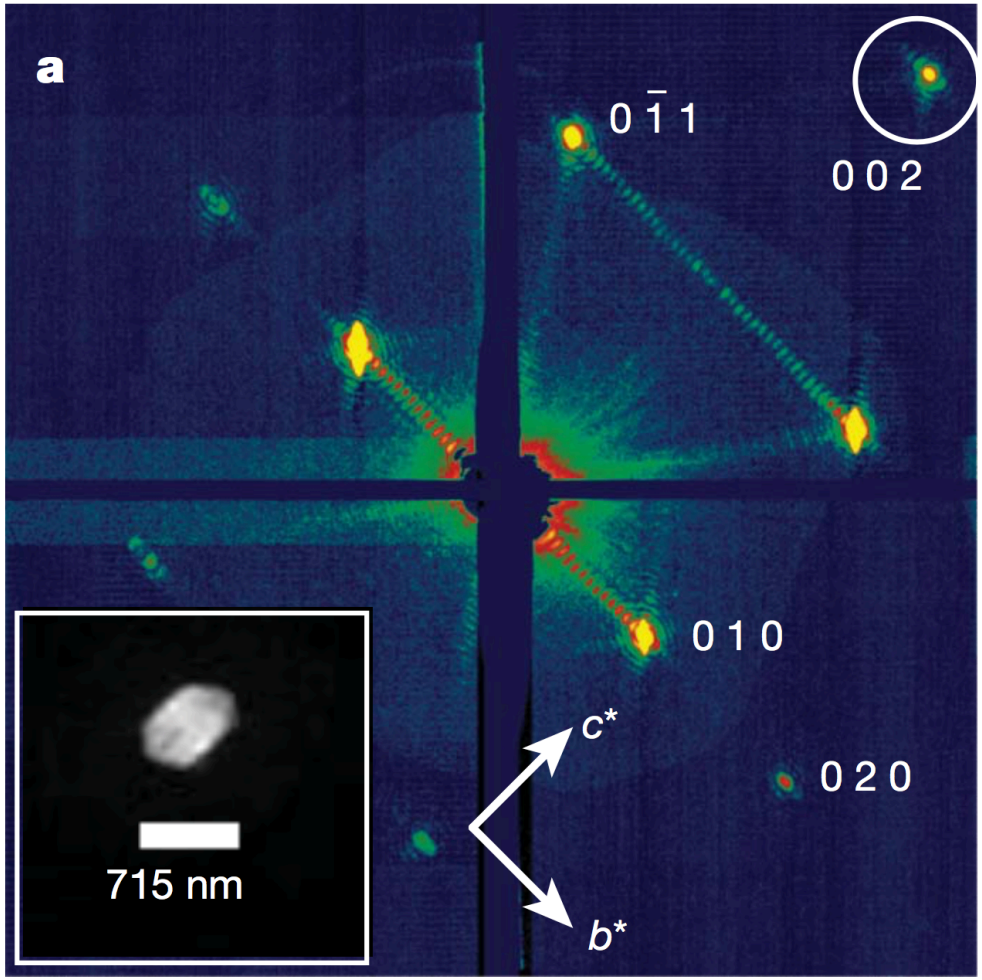
Drawback of the 'big object' approach: the library is optimized for algorithms needing large number of trials/s

APPROACH 2: SMALL CLASSES / TOOLKIT / PYNX

```
class ERProj(CLOperatorCDI):
    """
    Error reduction.
    """

    def __init__(self, positivity=False):
        super(ERProj, self).__init__()
        self.positivity = positivity

    def op(self, cdi):
        if self.positivity:
            self.processing_unit.cl_er_real(cdi._c
        else:
            self.processing_unit.cl_er(cdi._cl_obj, cdi._cl_support)
        return cdi


class ER(CLOperatorCDI):
    """
    Error reduction cycle
    """

    def __new__(cls, positivity=False, calc_llk=False):
        return ERProj(positivity=positivity) * FourierApplyAmplitude(calc_llk=calc_llk)
```

Idea:
decompose the computing problem in as many independent snippets as possible

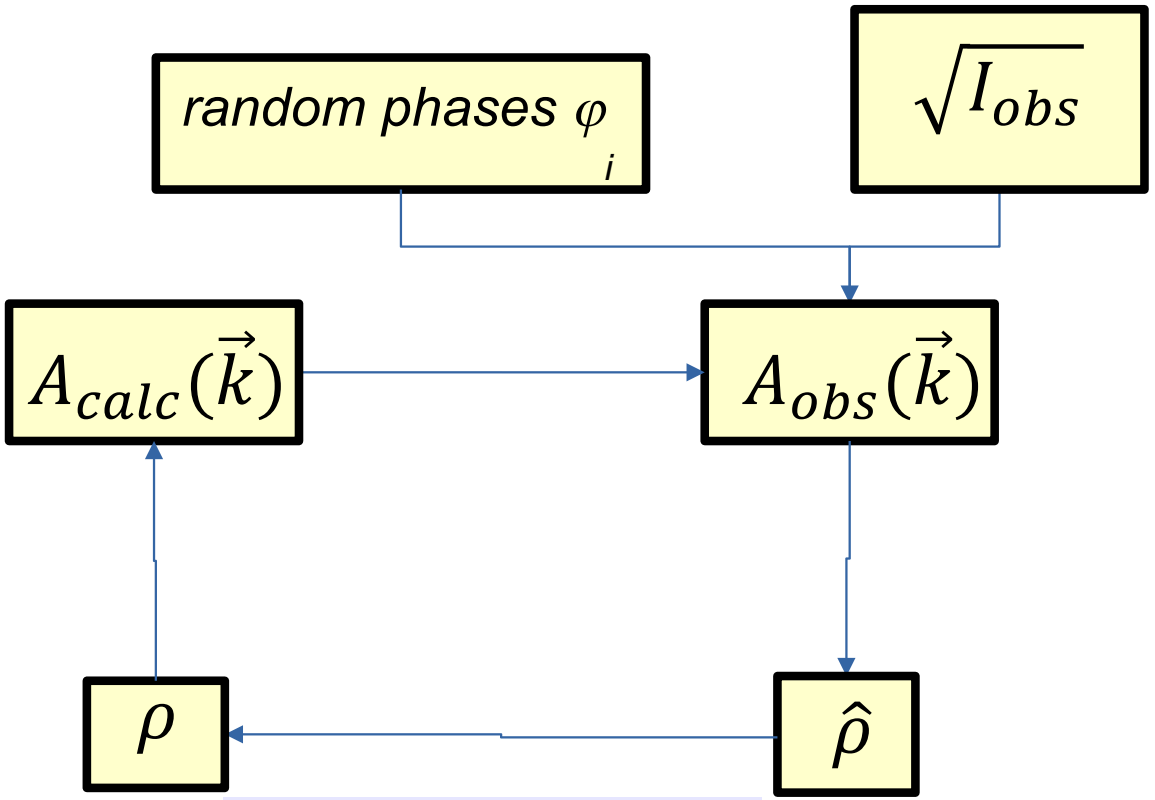
COHERENT DIFFRACTION IMAGING



- Illuminate a single crystal
- Scattering on detector is the Fourier Transform of the Crystal's shape
- The phase of the oscillations is lost !

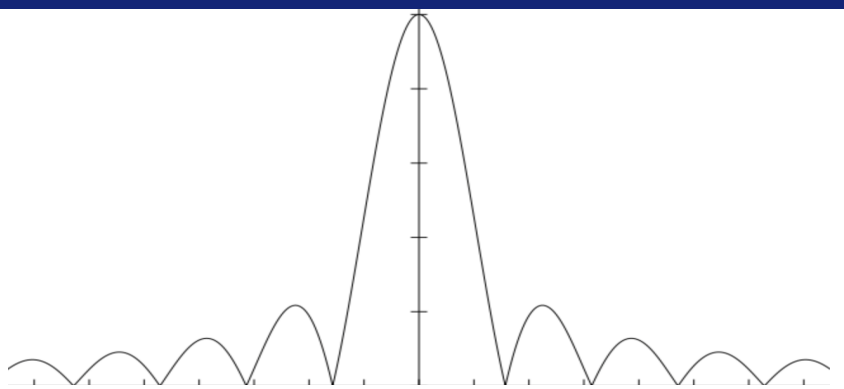
Nature **470**, 73 (2011)

COHERENT DIFFRACTION IMAGING ALGORITHM(S)



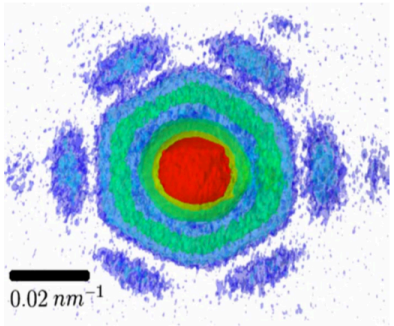
Density modification:

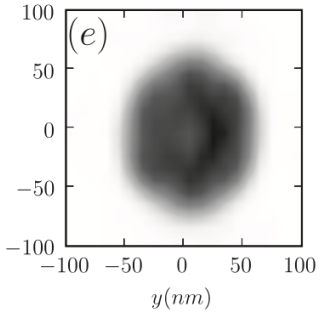
- Positivity (*)
- Finite support



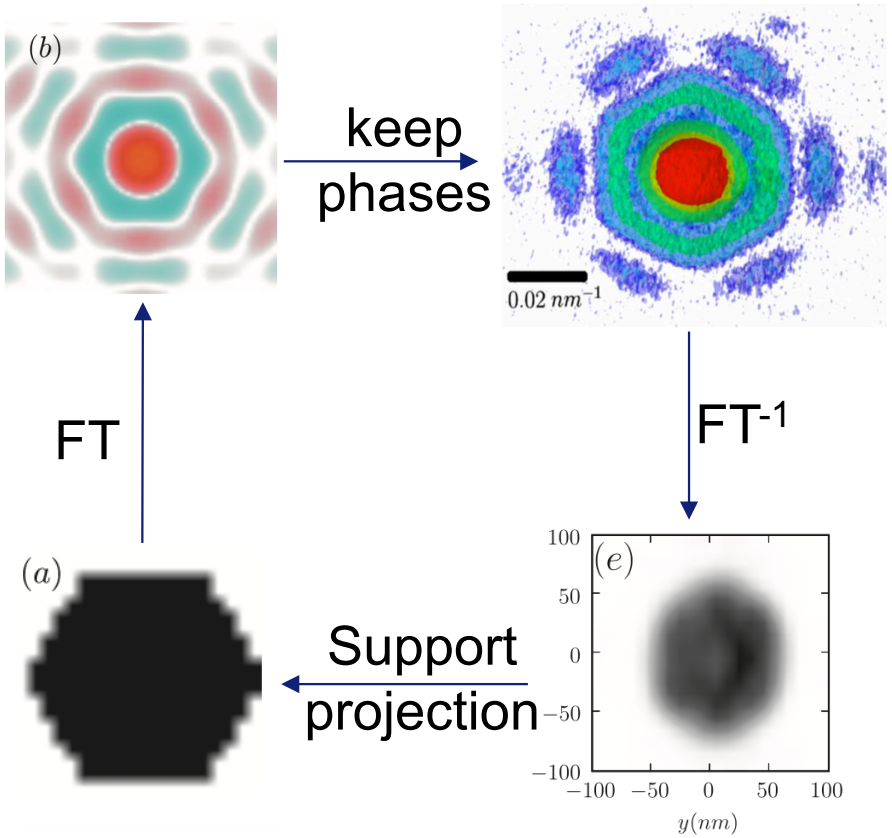
The algorithms can only converge if the problem is **over-determined**

→ need more than 1 point per fringe
“ oversampling ”





CDI: ERROR REDUCTION

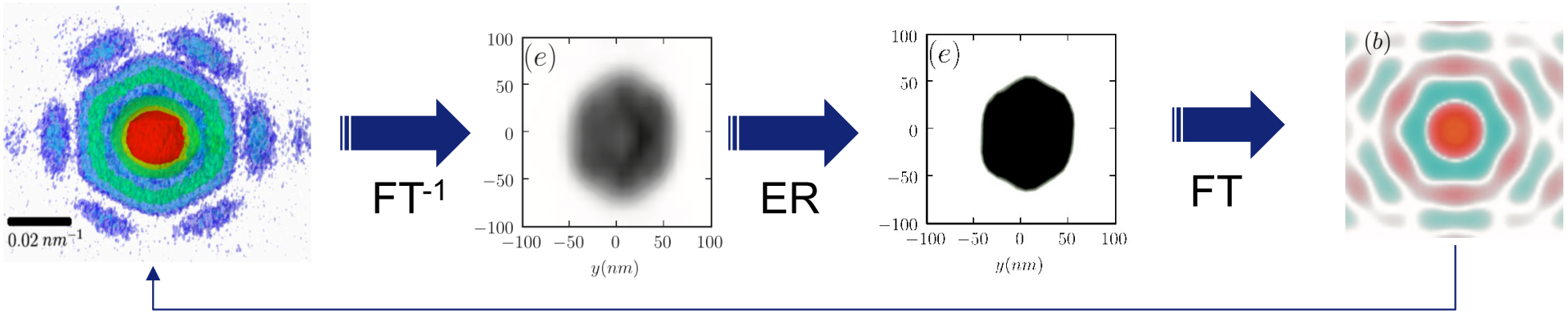


ER: support projection

=

set electronic density to zero outside a defined support

CDI: OPERATORS



Fourier Amplitude projection: keep only calculated phases

All operations on can be described as mathematical operators:

$$A_{calc}\ (i+1) = FA_{proj} * FT^{-1} * ER_{proj} * FT * A_{calc}\ (i)$$

CDI: OPERATORS

TABLE I. Summary of various algorithms.

Algorithm	Iteration $\rho^{(n+1)}=$
ER	$\boldsymbol{P}_s\boldsymbol{P}_m\rho^{(n)}$
SF	$\boldsymbol{R}_s\boldsymbol{P}_m\rho^{(n)}$
HIO	$\begin{cases}\boldsymbol{P}_m\rho^{(n)}(\boldsymbol{r}) & \boldsymbol{r}\in S\\ (\boldsymbol{I}-\beta\boldsymbol{P}_m)\rho^{(n)}(\boldsymbol{r}) & \boldsymbol{r}\notin S\end{cases}$
DM	$\{I+\beta\boldsymbol{P}_s[(1+\gamma_s)\boldsymbol{P}_m-\gamma_s\boldsymbol{I}]-\beta\boldsymbol{P}_m[(1+\gamma_m)\boldsymbol{P}_s-\gamma_m\boldsymbol{I}]\}\rho^{(n)}$
ASR	$\frac{1}{2}[\boldsymbol{R}_s\boldsymbol{R}_m+\boldsymbol{I}]\rho^{(n)}$
HPR	$\frac{1}{2}[\boldsymbol{R}_s(\boldsymbol{R}_m+(\beta-1)\boldsymbol{P}_m)+I+(1-\beta)\boldsymbol{P}_m]\rho^{(n)}$
RAAR	$\left[\frac{1}{2}\beta(\boldsymbol{R}_s\boldsymbol{R}_m+\boldsymbol{I})+(1-\beta)\boldsymbol{P}_m\right]\rho^{(n)}$

Marchesini, S. '*A unified evaluation of iterative projection algorithms for phase retrieval*'. Review of Scientific Instruments **78** (2007), 011301

P_m:

- Fourier transform the object
- Impose magnitude in Fourier space from observed intensity
- Back-Fourier Transform

P_s

- Replace density by zero outside of support

PYTHON OPERATOR OVERLOADING

```
class Operator:
    """
    Base class for an operator, applying e.g. to a wavefront object.
    """

    def __init__(self):
        self.ops = [self]

    def __mul__(self, w):
        """
        Applies the operator to an object.

        :param w: the object to which this operator will be applied.
                  If it is another Operator, the operation will be stored for later application.
        :return: the object, result of the operation. Usually the same object (modified) as w.
        """
        if isinstance(w, Operator):
            self.ops += w.ops
            return self
        self.apply_ops_mul(w)
        return w
```

Overload the __mul__ method:
A = Op * A
Is the same as A=Op.__mul__(A)

To apply multiple operators:
A = Op3 * Op2 * Op1 * A

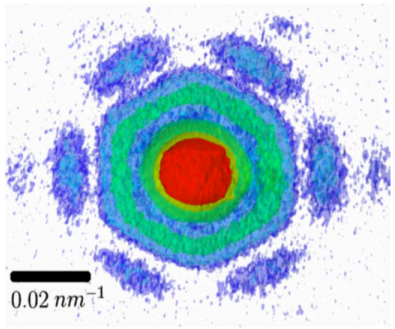
ERROR REDUCTION OPERATORS

```python
class FourierApplyAmplitude(CLOperatorCDI):
    """
    Fourier magnitude operator, performing a Fourier transform, the magnitude projection, and a backward FT.
    """
    def __new__(cls, calc_llk=False):
        return IFT() * ApplyAmplitude(calc_llk=calc_llk) * FT()


class ERProj(CLOperatorCDI):
    """Error reduction projection"""
    def op(self, cdi):
        self.processing_unit.cl_er(cdi._cl_obj, cdi._cl_support)
        return cdi

class ER(CLOperatorCDI):
    """  Error reduction cycle"""
    def __new__(cls, positivity=False, calc_llk=False):
        return ERProj(positivity=positivity) * FourierApplyAmplitude(calc_llk=calc_llk)
```

```opencl
/// Error reduction OpenCL code
void ER(const int i, __global float2 *d,
                __global char *support)
{
  if(support[i]==0) d[i] = (float2)(0,0) ;
}
```

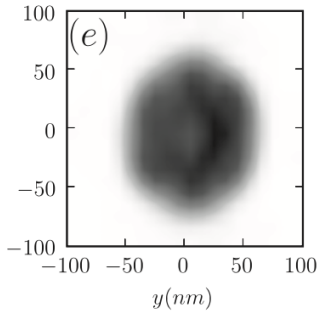
All operators are just a few lines of code

CHAINING OPERATORS



Also overload the __pow__ operator

Example full CDI reconstruction algorithm:




```
cdi = ER()**100 * (SupportUpdate() * ER()**50 * HIO()**200)**5 * cdi
```

Shrinks the support area

Hybrid Input-Output
Linear combination between current and previous crystal model (outside support)

BIG CLASSES VS TOOLKIT APPROACH

Public Member Functions

	UnitCell () Default Constructor.
	UnitCell (const REAL a, const REAL b, const REAL c, const string &SpaceGroupId) **UnitCell** Constructor (orthorombic) More...
	UnitCell (const REAL a, const REAL b, const REAL c, const REAL alpha, const REAL beta, const REAL gamma, const string &SpaceGroupId) **UnitCell** Constructor (triclinic) More...
	UnitCell (const **UnitCell** &oldCryst) **UnitCell** copy constructor.
	~UnitCell () Destructor.
virtual const string &	**GetClassName** () const Name for this class ("RefinableObj", "Crystal",...). More...
CrystVector_REAL	**GetLatticePar** () const Lattice parameters (a,b,c,alpha,beta,gamma) as a 6-element vector in Angstroems and radians. More...
REAL	**GetLatticePar** (const int whichPar) const Return one of the 6 Lattice parameters, 0<= whichPar <6 (a,b,c,alpha,beta,gamma), returned in Angstroems and radians. More...
const **RefinableObjClock** &	**GetClockLatticePar** () const last time the Lattice parameters were changed
const CrystMatrix_REAL &	**GetBMatrix** () const Get the 'B' matrix (**UnitCell::mBMatrix**)for the **UnitCell** (orthogonalization matrix for the given lattice, in the reciprocal space) More...
const CrystMatrix_REAL &	**GetOrthMatrix** () const Get the orthogonalization matrix (**UnitCell::mOrthMatrix**)for the **UnitCell** in real space. More...
const **RefinableObjClock** &	**GetClockMetricMatrix** () const last time the metric matrices were changed
CrystVector_REAL	**GetOrthonormalCoords** (const REAL x, const REAL y, const REAL z) const Get orthonormal cartesian coordinates for a set of (x,y,z) fractional coordinates. More...
void	**FractionalToOrthonormalCoords** (REAL &x, REAL &y, REAL &z) const Get orthonormal cartesian coordinates for a set of (x,y,z) fractional coordinates. More...
void	**OrthonormalToFractionalCoords** (REAL &x, REAL &y, REAL &z) const Get fractional cartesian coordinates for a set of (x,y,z) orthonormal coordinates. More...
void	**MillerToOrthonormalCoords** (REAL &x, REAL &y, REAL &z) const Get Miller H,K, L indices from orthonormal coordinates in reciprocal space. More...
void	**OrthonormalToMillerCoords** (REAL &x, REAL &y, REAL &z) const Get orthonormal coordinates given a set of H,K, L indices in reciprocal space. More...
virtual void	**Print** (ostream &os) const Prints some info about the **UnitCell**. More...
virtual void	**Print** () const
const **SpaceGroup** &	**GetSpaceGroup** () const Access to the **SpaceGroup** object.
SpaceGroup &	**GetSpaceGroup** () Access to the **SpaceGroup** object.
REAL	**GetVolume** () const Volume of Unit Cell (in Angstroems)

Big classes allow fine-tuning for specific applications

… allow more optimizations

… but are not very flexible

Small classes are much easier to re-use (cctbx)

… but can lead to a very large number of classes / function

```
class FourierApplyAmplitude(CLOperatorCDI):
    """
    Fourier magnitude operator, performing a Fourier transform, the
magnitude projection, and a backward FT.
    """
    def __new__(cls, calc_llk=False):
        return IFT() * ApplyAmplitude(calc_llk=calc_llk) * FT()
```

INLINE DOCUMENTATION

Python

```python
class Operator:
    """
    Base class for an operator, applying e.g. to a wavefront o
    """
    def __mul__(self, w):
        """
        Applies the operator to an object.
        """
        self.apply_ops_mul(w)
        return w
```

C++

```cpp
/** \brief Unit Cell class: Unit cell with
*/
class UnitCell
{
   public:
      /// Default Constructor
      UnitCell();
      /** \brief UnitCell Constructor (triclinic)
      *  \param a,b,c : unit cell dimension, in angstroems
      *  \param alpha,beta,gamma : unit cell angles, in radians.
      *  \param SpaceGroupId: space group symbol or number
      */
      UnitCell(const REAL a, const REAL b, const REAL c, const REAL alpha,
               const REAL beta, const REAL gamma,const string &SpaceGroupId);
```

- **Code is ALWAYS used much longer than you'd expect**
- **Even small code must be documented**
- **Use INLINE documenting**
- **Write/update documentation at the same time of the code**

DOCUMENTATION: DOXYGEN/C++

Public Member Functions

UnitCell ()
Default Constructor.

UnitCell (const REAL a, const REAL b, const RE
UnitCell Constructor (orthorombic) More...

UnitCell (const REAL a, const REAL b, const RE
UnitCell Constructor (triclinic) More...

Doxygen is a must-have tool for C++ documentation:

- **Automatic documentation**
- **Html, pdf**

Member Data Documentation

CrystMatrix_REAL ObjCryst::UnitCell::mBMatrix

B Matrix (Orthogonalization matrix for reciprocal space)

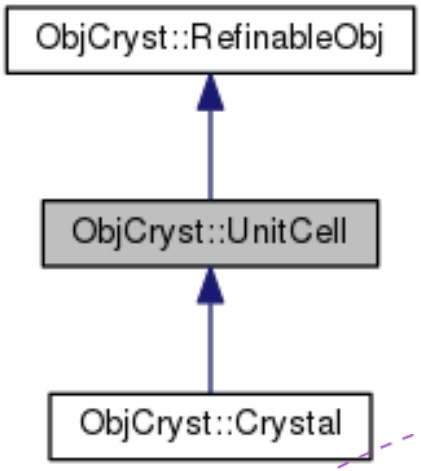
$$B = \begin{bmatrix} a^* & b^*\cos(\gamma^*) & c^*\cos(\beta^*) \\ 0 & b^*\sin(\gamma^*) & -c^*\sin(\beta^*)\cos(\alpha) \\ 0 & 0 & \frac{1}{c} \end{bmatrix}$$

$$\begin{bmatrix} k_x \\ k_y \\ k_z \end{bmatrix}_{orthonormal} = B \times \begin{bmatrix} h \\ k \\ l \end{bmatrix}_{integer}$$

.

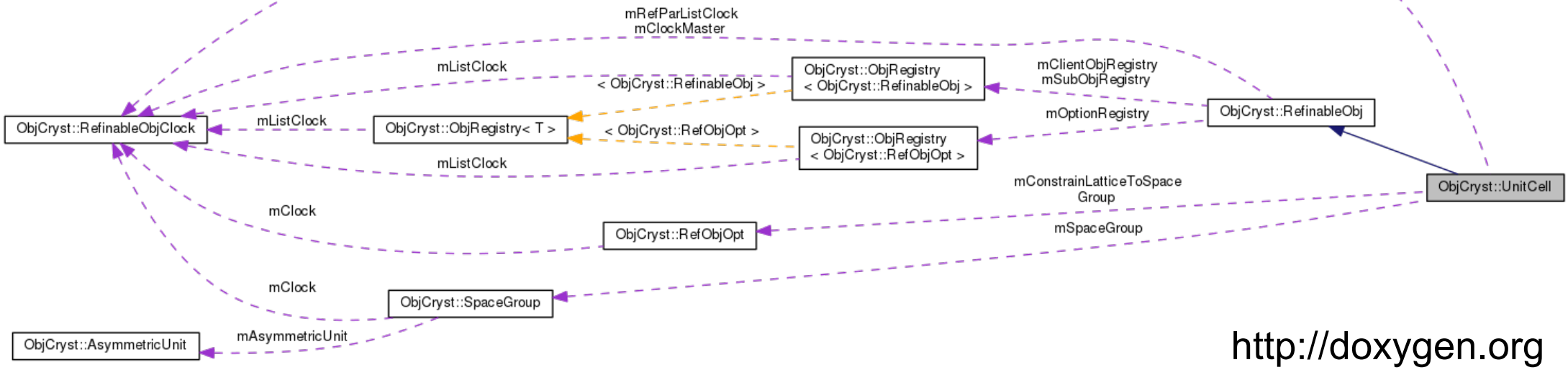
http://doxygen.org

DOCUMENTATION: DOXYGEN/C++



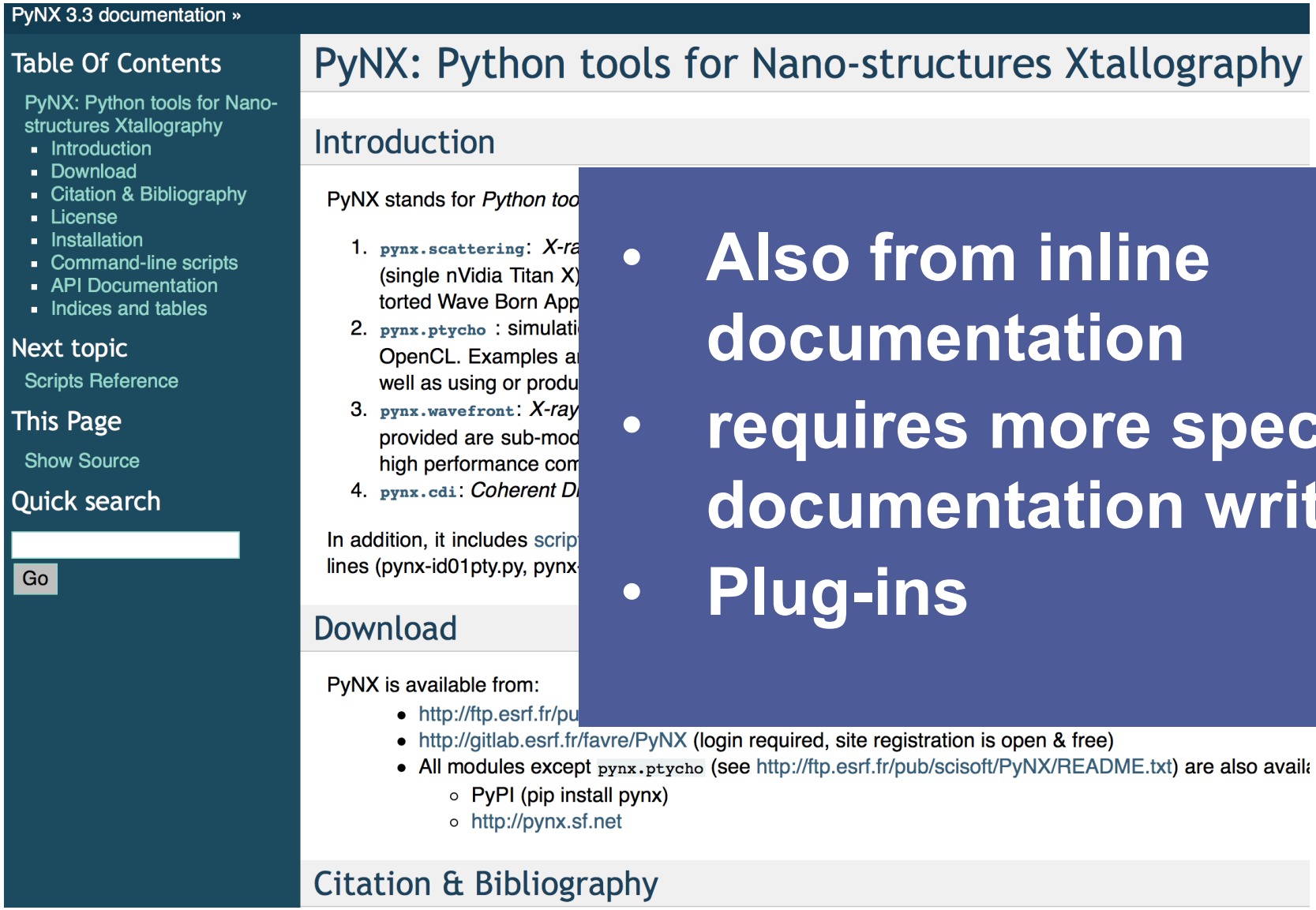
Doxygen is a must-have tool for C++ documentation:

- **Automatic documentation**
- **Html, pdf**
- **Dependencies, hierarchy, graphs**



http://doxygen.org

DOCUMENTATION: SPHINX/PYTHON



- **Also from inline documentation**
- **requires more specific documentation writing**
- **Plug-ins**

CONTROL VERSION SYSTEM: GIT

https://git-scm.com

- Full development history
- Branching, merging
- De-centralized (every copy is a full copy)
- Collaborative development
- Accountability
- Command-line or GUI
- Also useful for articles

```
1794        if("TextureEllipsoid"==tag.GetName())
1795        {
1796           mCorrTextureEllipsoid.XMLInput(is,tag);
1797           continue;
1798        }
1799        if("ReflectionProfilePseudoVoigt"==tag.GetName())
1800        {
1801           if(mpReflectionProfile==0)
1802           {
1803              mpReflectionProfile=new ReflectionProfilePseudoVoigt;
1804           }
1805           else
1806              if(mpReflectionProfile->GetClassName()!
                    ="ReflectionProfilePseudoVoigt")
1807              {
1808                 this->SetProfile(new ReflectionProfilePseudoVoigt);
1809              }
1810           mpReflectionProfile->XMLInput(is,tag);
1811           continue;
1812        }
1813        if("ReflectionProfileDoubleExponentialPseudoVoigt"==tag.GetName())
1814        {
1815           if(mpReflectionProfile==0)
1816           {
```

```
1689           }
1690           else
1691              if(mpReflectionProfile->GetClassName()!
                    ="ReflectionProfilePseudoVoigt")
1692              {
1693                 delete mpReflectionProfile;
1694                 mpReflectionProfile=new ReflectionProfilePseudoVoigt;
1695              }
1696           mpReflectionProfile->XMLInput(is,tag);
1697           continue;
1698        }
1699        if("ReflectionProfileDoubleExponentialPseudoVoigt"==tag.GetName())
1700        {
1701           if(mpReflectionProfile==0)
1702           {
1703              mpReflectionProfile
1704                 =new ReflectionProfileDoubleExponentialPseudoVoigt(this->
                       GetCrystal());
1705           }
```

UNIT TESTING

```
ValueError
/Users/favre/dev/pynx-private/pyn
     19 if __name__ == '__main__
     20     try:
---> 21         w = CDIRunnerID1
     22         w.process_scans(
     23     except CDIRunnerExce
```

Source of errors:

- New code bugs
- MacOS, Linux(es), Windows
- Python 2.7, 3.3…3.6
- Different hardware
- …

Use ‘unit tests’ which can be run automatically

```
import unittest
import numpy as np

class TestCrystalStructureFactor(unittest.TestCase):
  def test_centric(self):
    cryst = Crystal(a=4,b=5,c=6, spacegroup=‘P-1’)
    cryst.add_random_atoms(nb=10)
    l, k, h = np.mgrid[-10:11, -10:11, -10:11]
    self.assertTrue(np.allclose(cryst.get_structure_factor(h=h,k=k,l=l), 1e-6)
```

AUTOMATED TESTING

```
ValueError
/Users/favre/dev/pynx-private
     19 if __name__ == '__mai
     20     try:
---> 21         w = CDIRunner
     22         w.process_sca
     23     except CDIRunnerE
```

```
import unittest
import numpy as np

class TestCrystalStructureFacto
  def test_centric(self):
    cryst = Crystal(a=4,b=5,c=6, s
    cryst.add_random_atoms(nb=
    l, k, h = np.mgrid[-10:11, -10:1
    self.assertTrue(np.allclose(cryst.get_structure_factor(h=h,k=k,l=l), 1e-6)
```

Ideally:

- **Automatic testing every time new code is pushed to a server**
- **Always test against different platforms, libraries version…**
- **Lots of unit tests**
- **Continuous integration**

PART 2: EFFICIENT PYTHON

Fortran
C
C++
Java
Python

OpenCL, CUDA

Which languages are fast ?

- **Compiled vs interpreted vs just-in-time**
- **Development *vs* execution speed**
- **Command-line interpreter**

PYTHON SPEED

```
import timeit
import numpy as np
nb=100000

# With a python loop
nbiter=10
a=np.arange(nb) # loop
b=np.arange(nb)
c=np.arange(nb)
t1=timeit.default_timer()
for i in range(nbiter):
   for i in range(nb):
      c[i]=a[i]+b[i]

t2=timeit.default_timer()
mflops=(nb*nbiter)/(t2-t1)/1e6
print("Boucle : %6.3f Mflops"%(mflops))
```

Python *is* slow on *individual* floating point values

2.7 Mflop/s (on this laptop, 2.5GHz Intel i7)

AVOIDING LOOPS: NUMPY

```
import timeit
import numpy as np
nb=100000

# Using numpy operations
nbiter=1000
a=np.arange(nb) # loop
b=np.arange(nb)
c=np.arange(nb)
t1=timeit.default_timer()
for i in range(nbiter):
   c=a+b

t2=timeit.default_timer()
mflops=(nb*nbiter)/(t2-t1)/1e6
print("Speed : %6.3f Mflops"%(mflops))
```

- **Numpy is optimized, so use operations on vectors !**
- ***Never* use a loop on individual values !**

All lengthy operations should be delegated to python libraries

1.3 Gflop/s (on this laptop, 2.5GHz Intel i7)
Speedup x500

NUMPY VECTOR OPERATIONS

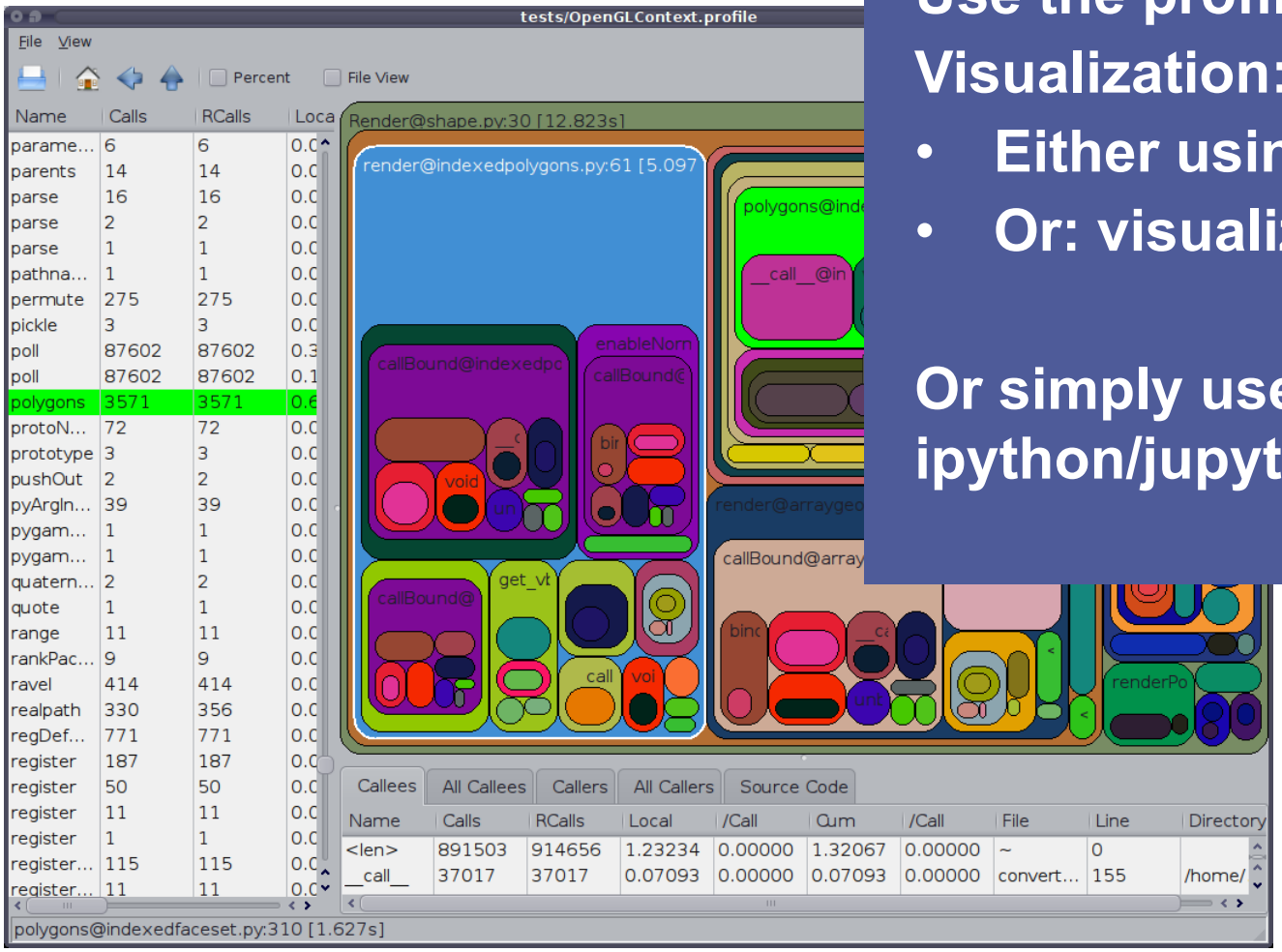
```
import numpy as np

nb = 1000
a = np.random.uniform(0, 1, size=(nb, nb))

# operation on a sub-array
a[10:-10, 20:-20] = b[10:-10, 20:-20] * c[10:-10, 20:-20]

# Count values > 0.7
(a > 7).sum()

# Extract values > 0.7 (result is flattened)
d = a[a > 0.7]

# double values > 0.7
a[a > 0.7] *= 2  # version 1
a += a * (a > 0.7)  # version 2
```

- **Use operations on vectors**
- **Also for:**
  - **sub-arrays**
  - **conditional access**

Only random walks can hardly be vectorized

PYTHON PROFILING

python -m cprofile -o log run.py



Use the profiler from python: cProfile

Visualization:

- **Either using the pstats module**
- **Or: visualize with runsnake**

Or simply use %timeit (within ipython/jupyter)

C++ IN PYTHON: CYTHON

```
def fib(n):
"""Print the Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print(b)
    a, b = b, a + b
```

`fib.pyx` file to be compiled

```
from distutils.core import setup
from Cython.Build import cythonize
setup( ext_modules=cythonize("fib.pyx"), )
```

`Setup.py`: compilation setup

```
$ python setup.py build_ext --inplace
```

Compilation

```
import fib
fib.fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Use code from python

PYTHON (JUPYTER) NOTEBOOKS

IP[y]: Notebook pynx-ptycho-cxi-siemensstar-id01 Last Checkpoint: Dec 09 15:20 (unsaved changes)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

In [1]:
```
# TODO: make pynx-cxipty.py lasses importable instead of just embedded in the script
from cxipty import PtychoRunnerScanCXI
from cxipty import params_generic as params
pylab.rcParams['figure.figsize'] = (12, 8)
print('Import OK')
```

```
Import OK
```

In [2]:
```
params['cxifile']='data/data.cxi'
params['probe'] ='focus,60e-6x200e-6,0.09'
params['gpu'] = 'K80'
params['algorithm'] = '20DM'
params['object'] = 'random,0.8,1,0,0.5'
params['verbose'] = 5
params['liveplot'] = True
```

In [ ]:
```
ws = PtychoRunnerScanCXI(params, 0)
```

In [ ]:
```
ws.load_data()  # Load 1025 frames from a maxipix detector using CXI/HDF5 data
```

In [ ]:
```
ws.prepare()
```

In [ ]:
```
ws.run()
```

In [ ]:
```
ws.run_algorithm('20AP')
```

In [ ]:
```
ws.run_algorithm('20ML')
```

In [ ]:
```
ws.run_algorithm('nbprobe=3,20AP')
```

In [ ]:
```
ws.run_algorithm('20ML')
```

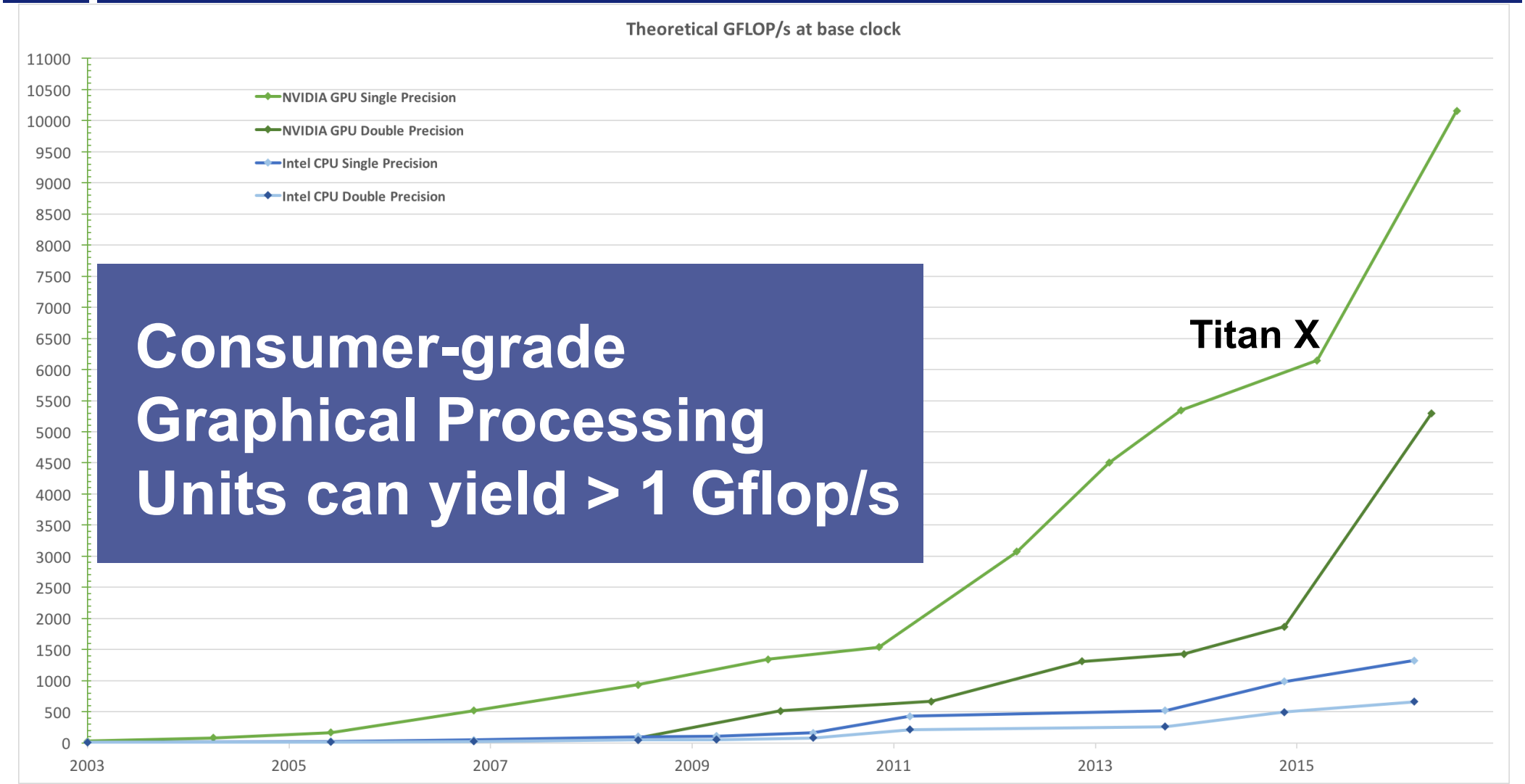
PYTHON #1 MISTAKES: COPY BY REFERENCE

```
In [1]: import numpy as np

In [2]: a=np.arange(8)

In [3]: b=a

In [4]: print(a)
[0 1 2 3 4 5 6 7]

In [5]: b[5]=90

In [6]: print(a)
[ 0  1  2  3  4 90  6  7]
```

- Python defaults to copy-by-reference
- Modifying a *shallow* copy of an object also changes the original object
- This saves memory !
- Memory is deleted after last referecing object is deleted (garbage collection)

- Whenever a real copy is needed:
  - Use the 'copy.deepcopy' function
  - For numpy array: a = b.copy()

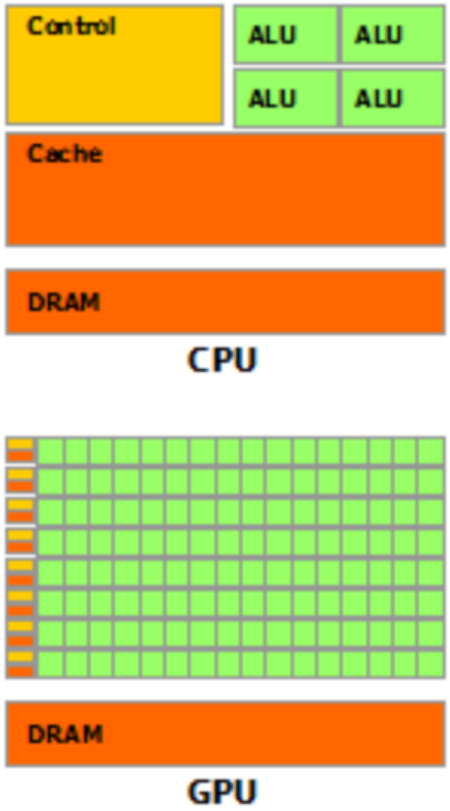
GPU: WHEN THE CPU IS NOT FAST ENOUGH



Consumer-grade Graphical Processing Units can yield > 1 Gflop/s

http://docs.nvidia.com/cuda/cuda-c-programming-guide/

GPU COMPUTING ARCHITECTURE



CPU use a few computing cores, with a general-purpose instruction set

GPU are optimized for > 10^4 parallel threads, with optimized instructions (fast exp, trigonometric functions,...)

Ex. GPU: 768 active threads/multiprocessor, with 30 multiprocessors

All GPU threads must execute the same code (on different data)

http://docs.nvidia.com/cuda/cuda-c-programming-guide/

OPENCL/GPU COMPUTING PRINCIPLE

Replace loops with functions (a kernel) executing at each point in a problem domain

E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

Traditional loops

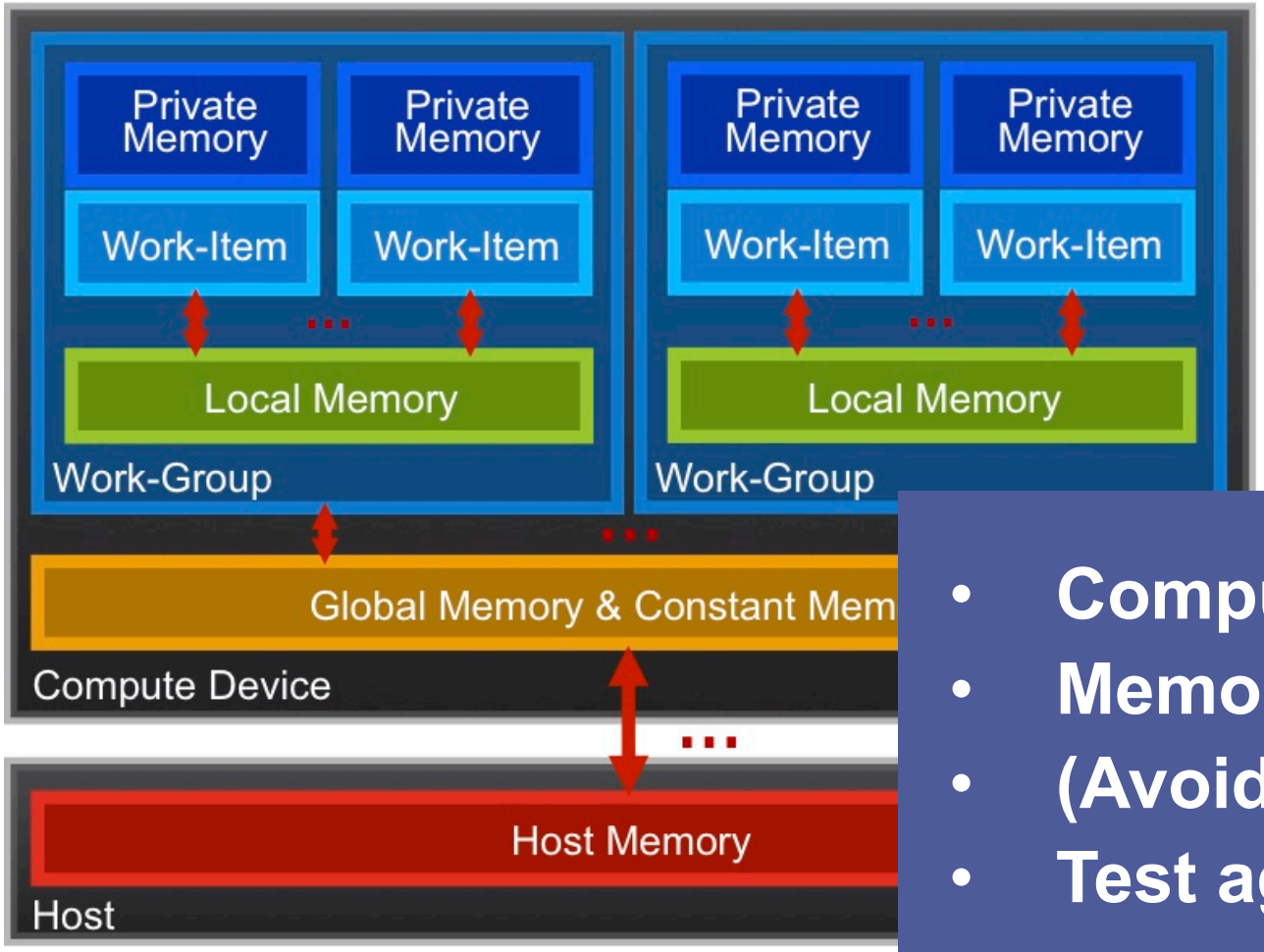
```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel,
// called work-items, execute
// in parallel
```

Slide from https://github.com/HandsOnOpenCL/Lecture-Slides

GPU EFFICIENCY & MEMORY



- Computing is cheap (fast)
- Memory access is expensive
- (Avoid tests)
- Test against different GPUs

https://github.com/HandsOnOpenCL/Lecture-Slides

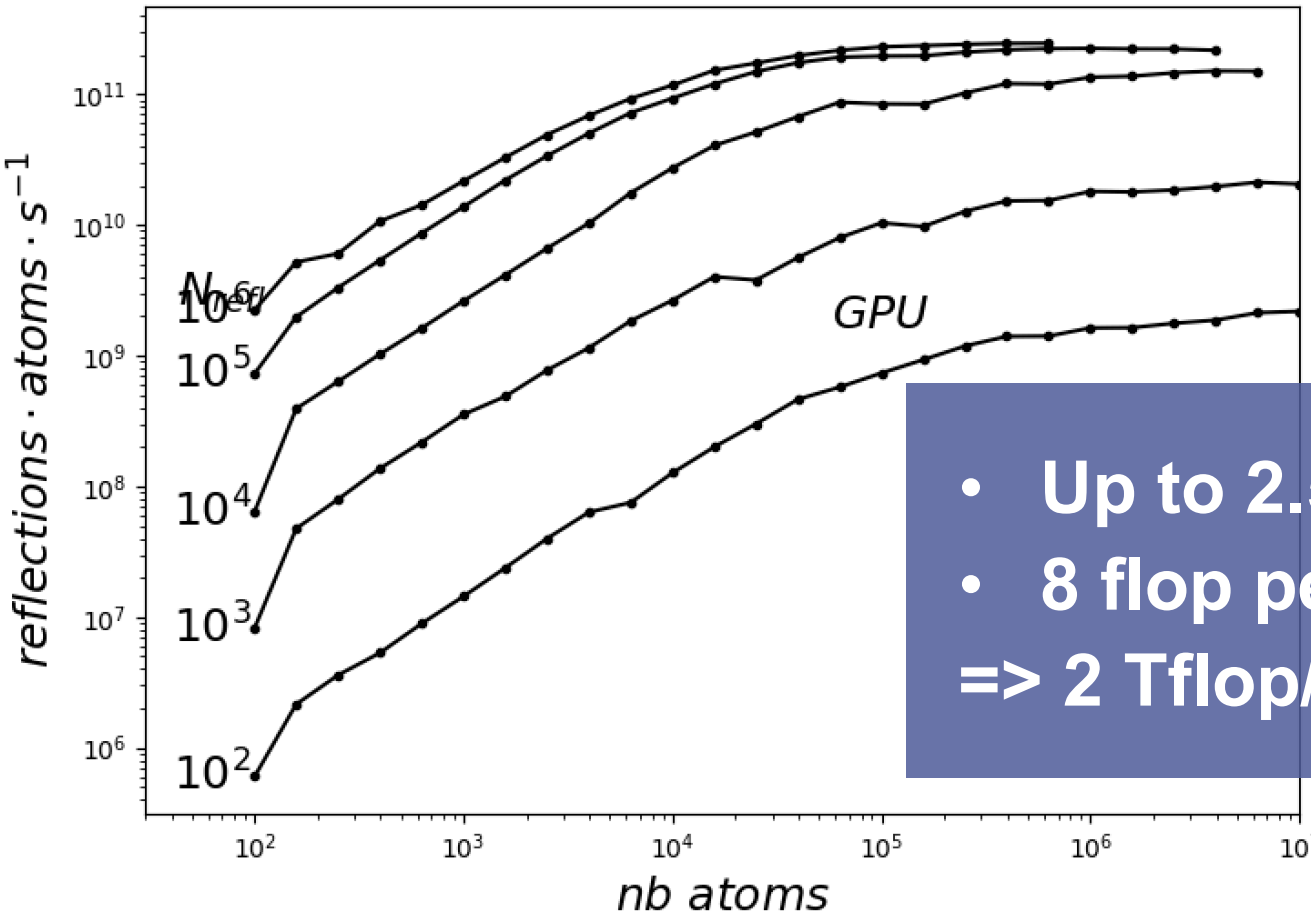
OPENCL: FAST STRUCTURE FACTOR

```
__kernel __attribute__((reqd_work_group_size(%(block_size)d, 1, 1)))
void Fhkl(__global float *fhkl_real,__global float *fhkl_imag,
          __global float *vx,__global float *vy,__global float *vz, const long natoms,
          __global float *vh,__global float *vk,__global float *vl)
{
  #define BLOCKSIZE %(block_size)d
  #define twopi 6.2831853071795862f
  // Block index
  int bx = get_group_id(0);
  int by = get_group_id(1);
  // Thread index
  int tx = get_local_id(0);

  const unsigned long ix=tx+(bx+by*get_num_groups(0))*BLOCKSIZE;
  const float h=twopi*vh[ix];
  const float k=twopi*vk[ix];
  const float l=twopi*vl[ix];
  float fr=0,fi=0;
  __local float x[BLOCKSIZE];
  __local float y[BLOCKSIZE];
  __local float z[BLOCKSIZE];
```

$$A(\vec{s}) = \sum f_i(s) e^{2\pi \vec{s}\cdot\vec{r}_i}$$

```
long at=0;
  for (;at<=(natoms-BLOCKSIZE);at+=BLOCKSIZE)
  {
    barrier(CLK_LOCAL_MEM_FENCE);
    x[tx]=vx[at+tx];
    y[tx]=vy[at+tx];
    z[tx]=vz[at+tx];
    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned int i=0;i<BLOCKSIZE;i++)
    {
      const float tmp=h*x[i] + k*y[i] + l*z[i];
      fi +=native_sin(tmp);
      fr +=native_cos(tmp);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  fhkl_real[ix]+=fr;
  fhkl_imag[ix]+=fi;
}
```

OPENCL: FAST STRUCTURE FACTOR



1 nVidia Titan X card (1300 EUR, 250W)

- **Up to 2.5x10^11 refl.atoms/s**
- **8 flop per atom-refl pair**

=> 2 Tflop/s

GPU FAST STRUCTURE FACTOR

F(hkl) computation

1) Electronic density within 1 unit cell on a fine grid

2) FFT

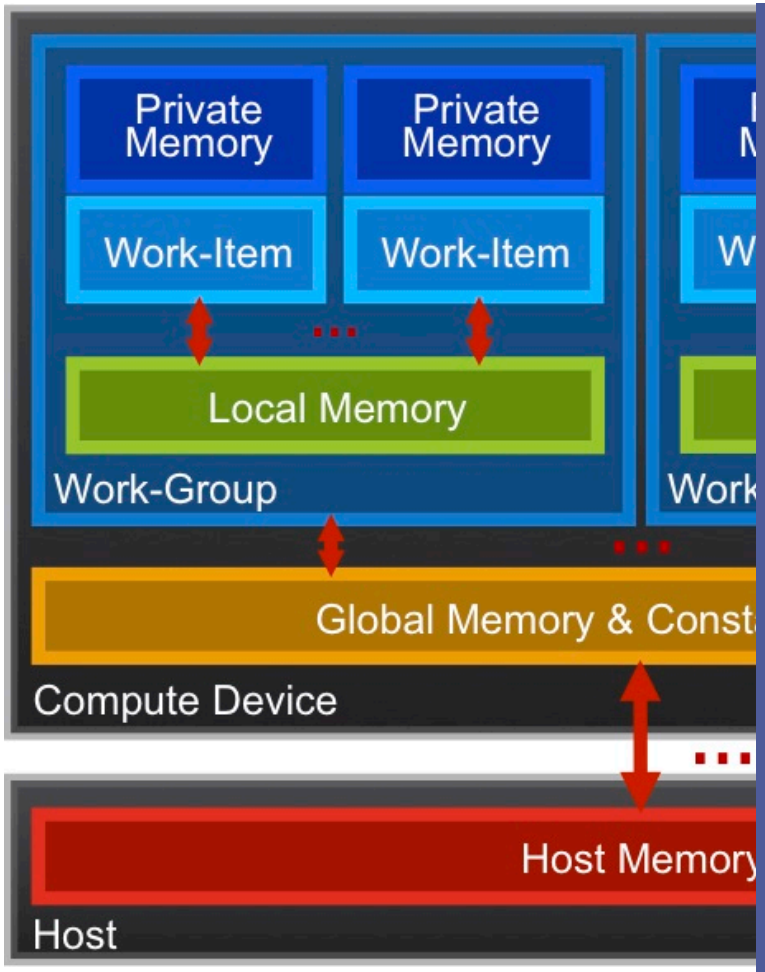
Direct calculation using atomic positions + scattering factors

$$A(\vec{s}) = \sum f_i(s) e^{2\pi \vec{s} \cdot \vec{r}_i}$$

- FFT will always be faster than direct calculations
- .. But only for a full-Fourier space calculation and many reflections

- Efficient for partial Fourier space calculations
- Nano-structures: many atoms, scattering around a single reflection

GPU COMPUTED CAN BE COMPLICATED



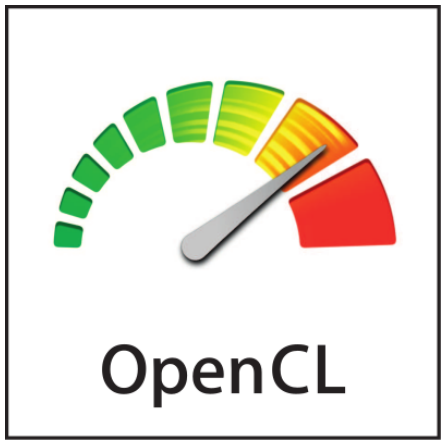
CUDA/OpenCL require:

- **Compiling GPU and C/C++**
- **Creating and transferring data between CPU (host) and GPU (device) memor*ies***
- **Initializing a computing *context* and *queue***
- **… it's hard maintenance**

https://github.com/HandsOnOpenCL/Lecture-Slides

PYTHON+OPENCL: EASY GPU COMPUTING

PyOpenCL




```
import numpy as np
import pyopencl as cl
a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)
res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)

prg = cl.Program(ctx, """
__kernel void sum(
    __global const float *a_g, __global const float *b_g, __global float *res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()

prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)
```

PYOPENCL: ELEMENT-WISE OPERATIONS

```
import numpy as np
import pyopencl as cl
from pyopencl.elementwise import ElementwiseKernel
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
n = 10
a_np = np.random.randn(n).astype(np.float32)
b_np = np.random.randn(n).astype(np.float32)

a_g = cl.array.to_device(queue, a_np)
b_g = cl.array.to_device(queue, b_np)

lin_comb = ElementwiseKernel(ctx,
    "float k1, float *a_g, float k2, float *b_g, float *res_g",
    "res_g[i] = k1 * a_g[i] + k2 * b_g[i]",
    "lin_comb")

res_g = cl.array.empty_like(a_g)
lin_comb(2, a_g, 3, b_g, res_g)

print((res_g - (2 * a_g + 3 * b_g)).get())  # Check result
```

Simple kernels when the same operation must be applied to all elements.

PYOPENCL: REDUCE KERNELS

Compute a single result from GPU arrays: sum, Chi^2,…

```
a = pyopencl.array.arange(queue, 400, dtype=numpy.float32)
b = pyopencl.array.arange(queue, 400, dtype=numpy.float32)

krnl = ReductionKernel(ctx, numpy.float32, neutral="0",
       reduce_expr="a+b", map_expr=" x[i]* x[i] - y[i] *y[i]",
       arguments="__global float *x, __global float *y")

chi2= krnl(a, b).get()
```

PYOPENCL PARALLEL ALGORITHMS

Elementwise, reduction kernels (and others: scan,..):

- Hide all the memory handling complexity
- Are simple enough that they are naturally optimised

And most important:

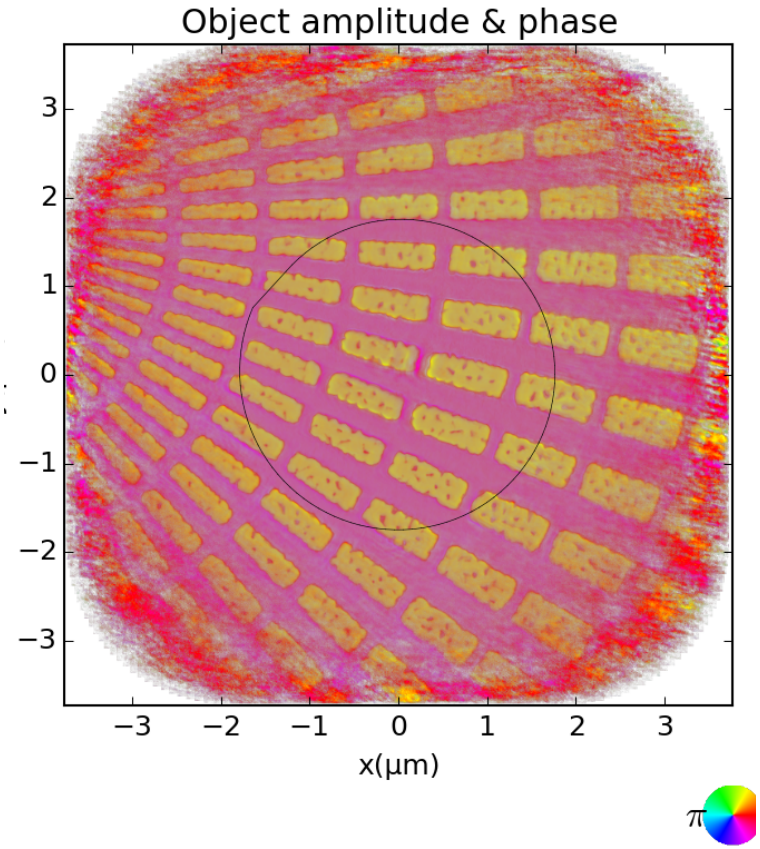
- A large number of crystallographic operations are simple vector operations: perfect candidate for GPU

PYOPENCL FFT

- **FFT supplied by clFFT (AMD) / gPyFFT**
- **Size with prime decomposition up to 13**
- **Performance up to a few 100 Gflop/s**

```
cl_psi = cla.zeros(cl_queue, (256, 256), np.complex64)
gpyfft_plan = gpyfft.FFT(cl_ctx, cl_queue, cl_psi, None)
for ev in gpyfft_plan.enqueue(forward=True): ev.wait()
for ev in gpyfft_plan.enqueue(forward=False): ev.wait()
```

PYOPENCL PERFORMANCE



In Ptychography

1025 frames of 400x400 pixels

0.24 s for one cycle with:

- **Forward & backward 2D FFT**
- **~2 elementwise kernels**

GPU COMPUTING EFFICIENCY

- Computing is cheap, memory access expensive
- *Chain* all calculations on the GPU (pipeline)
- GPU calculations are *asynchronous*: python execution continues before GPU commands are done. So the next GPU command can be prepared in the GPU queue
- 'Crystallography on-a-chip' approach

- Amdahl's rule: if you can optimize only fraction f of the execution time, the maximum speedup is 1/f
- Faster is *not* always better

CLOUD COMPUTING

Computing trends:
- Less desktop computers
- More laptops
- Tablets
- SmartPhones

In institutes:
- Clusters
- CPU and GPU
- Updates complicated (software, hardware)

Cloud computing is the future (and present)

- **On-demand availability:**
  - **Start machine in minutes**
  - **Start clusters**
- **Numerous configurations:**
  - **CPU, GPU**
  - **Memory**
  - **Storage latency**
- **Virtual machines:**
  - **Create one image per application**
  - **Stop distributing software / code !**
  - **Distribute images**

AMAZON EC2 (ELASTIC COMPUTING)

	vCPU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
Compute Optimized - Current Generation				
c4.large	8	3.75	EBS Only	$0.113 per Hour
c4.xlarge	16	7.5	EBS Only	$0.226 per Hour
c4.2xlarge	31	15	EBS Only	$0.453 per Hour
c4.4xlarge	62	30	EBS Only	$0.905 per Hour
c4.8xlarge	132	60	EBS Only	$1.811 per Hour
c3.large	7	3.75	2 x 16 SSD	$0.12 per Hour
c3.xlarge	14	7.5	2 x 40 SSD	$0.239 per Hour
c3.2xlarge	28	15	2 x 80 SSD	$0.478 per Hour
c3.4xlarge	55	30	2 x 160 SSD	$0.956 per Hour
c3.8xlarge	108	60	2 x 320 SSD	$1.912 per Hour
GPU Instances - Current Generation				
p2.xlarge	12	61	EBS Only	$0.972 per Hour
p2.8xlarge	94	488	EBS Only	$7.776 per Hour
p2.16xlarge	188	732	EBS Only	$15.552 per Hour
g2.2xlarge	26	15	60 SSD	$0.702 per Hour
g2.8xlarge	104	60	2 x 120 SSD	$2.808 per Hour
g3.4xlarge	47	122	EBS Only	$1.21 per Hour
g3.8xlarge	94	244	EBS Only	$2.42 per Hour
g3.16xlarge	188	488	EBS Only	$4.84 per Hour

CLOUD COMPUTING FOR DATA ANALYSIS

https://eoscpilot.eu

http://www.helix-nebula.eu

- Synchrotron & neutron facilities
- More accessible tools
- On-demand availability
- EU initiatives:
  - Helix Nebula Science cloud
  - European Open Science Cloud
  - http://pan-data.eu
- Open Data policies: more data will be made public

TAKE-AWAY: EFFICIENT PYTHON COMPUTING

- Choose wisely the approach when programming (big/small)
- Document from the start !
- Python is fast if well used
- Python relies on optimised libraries – use VECTOR operations
- Learn to work with cloud computing and virtual machines

Things I did not talk about:

- Supercomputing
- MPI

DON'T FORGET LICENSING

Popular Licenses

The following OSI-approved licenses are popular, widely used, or have strong communities:

- Apache License 2.0
- BSD 3-Clause "New" or "Revised" license
- BSD 2-Clause "Simplified" or "FreeBSD" license
- GNU General Public License (GPL)
- GNU Library or "Lesser" General Public License (LGPL)
- MIT license
- …

http://opensource.org

- **Code is ALWAYS used much longer than you'd expect**
- **Choose a LICENSE !**
- **Otherwise, it will become un-usable legally**
- **Preferably open-source !**

- **Note: 'public domain' does not exist everywhere**
- **Discussing with your legal department may be annoying, but it has do be done just once**

TUTORIALS

For a small group

Hands-on (with laptops, python, internet):

- SQL access to Crystallography Open Database
- Global optimization algorithms
- GPU computing basics + scattering calculations
- pyObjcryst: structure solution from powder pattern, using python (experimental)
- CDI and Ptychography using GPU with **PyNX**
- Basics of python/jupyter notebooks

Demo: ab initio structure solution from powder diffraction / FOX