TRIQS
a Toolbox for Research in Interacting Quantum Systems

Olivier Parcollet

Center for Computational Quantum Physics (CCQ)

Flatiron Institute, Simons Foundation

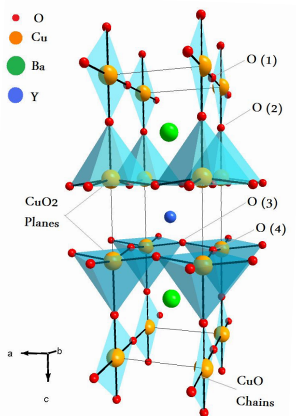
New York

Outline

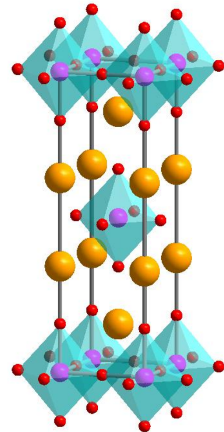
- Quick introduction to the Quantum Many-Body Problem
- The TRIQS project.
- A few technical topics
  - Hdf5
  - Python/C++ interface
  - Modern C++ and zero cost abstraction.

Many quantum particles in interaction

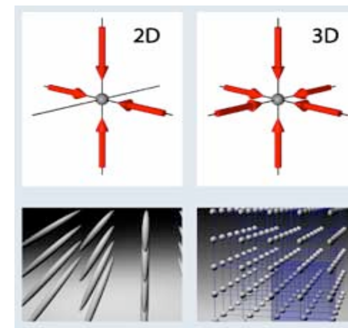
- Where ?
  - Electrons in a material.
  - Ultra-cold atoms in quantum optics.
- Why ?
  - Collective effects, low temperature.
  - New states of matter, e.g. high temperature superconductivity.



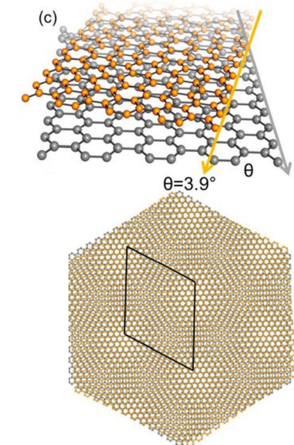
$YBa_2Cu_3O_{7-x}$



Sr_2RuO_4



Ultra-cold atoms



Twisted graphene (2018)

Quantum mechanics

- 1 electron in a crystal
  - Wavefunction $\psi(x)$

$$H = -\frac{\hbar^2}{2m}\nabla_x^2 + V_{\text{crystal}}(x)$$

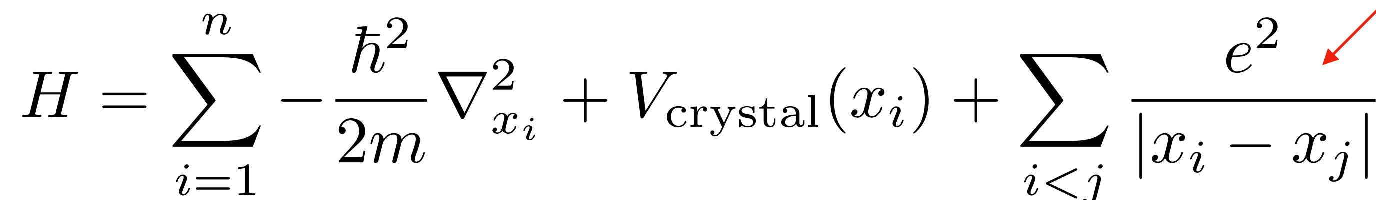
Schrödinger equation

$$H|\psi\rangle = E|\psi\rangle$$

- N electrons in a crystal (N = 10^{23})
  - Many-body wavefunction : N variables $x_1, \ldots, x_N$

$$\psi(x_1, \ldots, x_N)$$

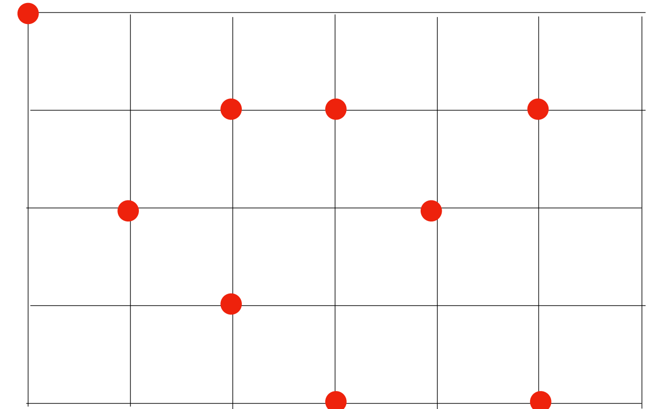
Coulomb interaction



Exponential complexity

- Electrons hopping on a lattice with N sites and interacting.
- Schrödinger equation : eigenvalue problem for the matrix H

$$H|\psi\rangle = E|\psi\rangle$$



- One site
  - 0 or 1 electron with spin up/down (Pauli principle, spin 1/2)
  - Hilbert space of dimension 4.
- The full lattice
  - Tensor product of each site Hilbert space
  - Dimension = 4^N
- H is a matrix with dimension exponential in N.

The quantum many body problem
is exponentially hard.

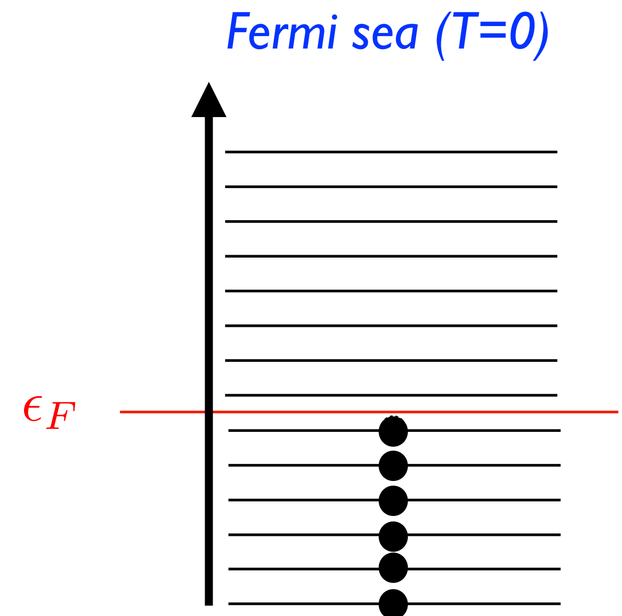
Really ?

Fermi gas

- Neglect the Coulomb interaction between electrons ?

$$H = \sum_{i=1}^{n} -\frac{\hbar^2}{2m}\nabla_{x_i}^2 + V_{\text{crystal}}(x_i) + \sum_{i<j} \frac{e^2}{|x_i - x_j|}$$

- Independent electrons + Pauli principle.
- Solve 1 electron problem.
- Many-body ground state = Fermi sea



- But interaction is not small !?
  kinetic energy = Coulomb interaction = 10eV, 10^6 K …

“Standard model” of solid state physics

- At low energy/temperature, approximately a Fermi gas. Quasi-electrons with e.g. effective mass $m^* > m_e$.
  Fermi liquid theory Landau 50’

- 1 electron in a effective potential (interactions “in average”)
  Density functional theory Kohn, 60’s

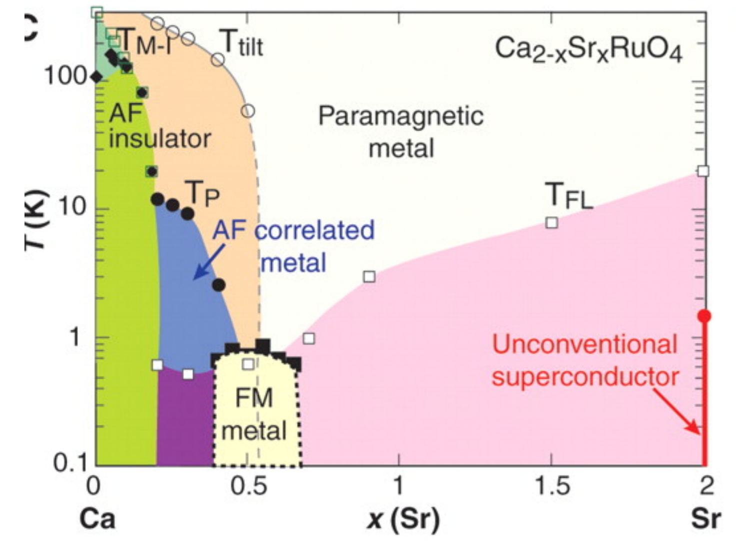
$$H = -\frac{\hbar^2}{2m}\nabla_x^2 + V_{effective}(x)$$

- Well established method. Many DFT codes available.

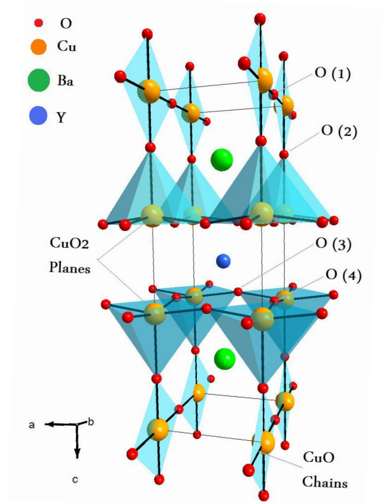
- Works very well in many cases but …

Strongly correlated systems

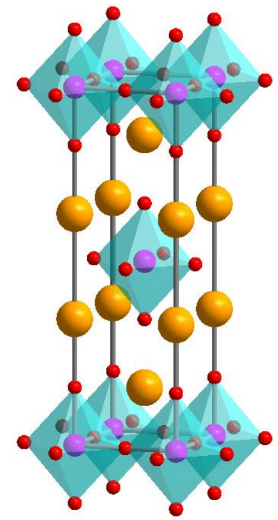
- … when this "standard model" breaks down !
- Studied at CCQ



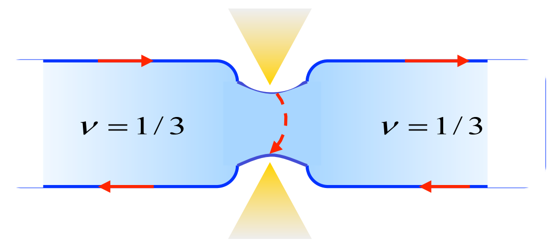
A lot of phases at low temperature !



High Tc cuprate superconductors



Sr_2RuO_4



Fractional Quantum Hall effect.

Mathematical framework ?

- Classical fluids
  Macroscopic physics described by some PDE, e.g. Navier-Stokes.
- Quantum case
  No Partial Differential (or Integral) Equations.
  Low energy, long distance physics collective effects described by a quantum field theory.

- Given a crystal structure or a simple model (electrons on a lattice) can we compute the physical properties ?
- Algorithmic complexity ?

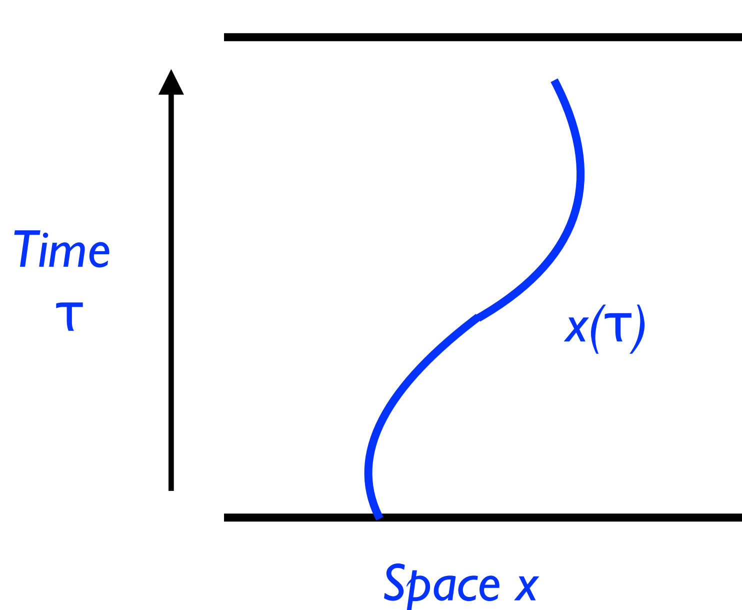
Study the many-body wave function

- Physical ground states are *not generic*, they have *structure*.
- Compact representation of ψ ?

- iTensor *(Cf Miles' talk)* Tensor representation.
- NetKet *(Cf Giuseppe's talk)*: Use a neural network

Path integral

- Another view of quantum mechanics *(Feynman)*
- Sum over trajectories/paths.
- E.g. one particle in quantum mechanics



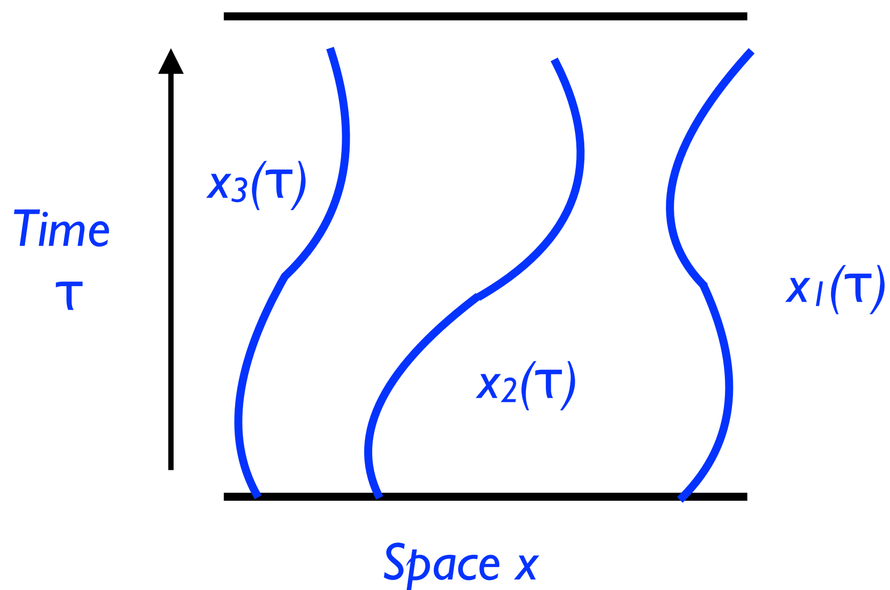
$$\int \mathcal{D}x(\tau) e^{-\int d\tau S(x(\tau))}$$

Action

Sum over all trajectories

Quantum many body path integral

- Multiple particles trajectories.



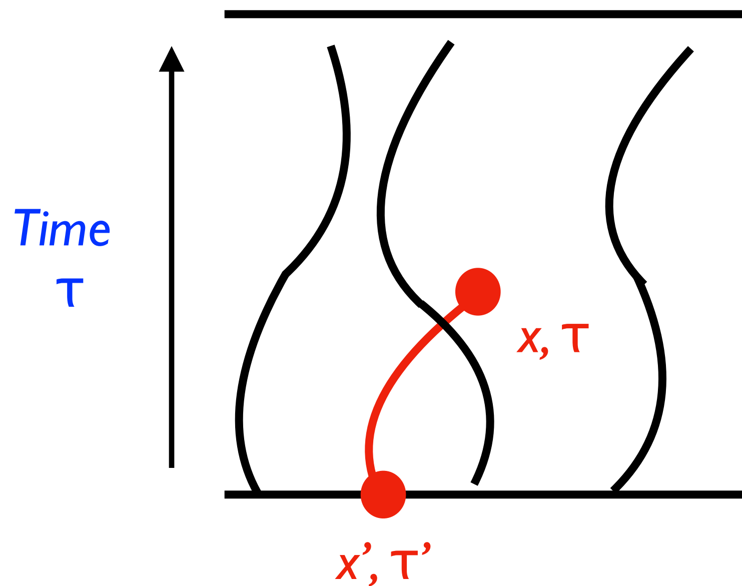
Action

$$\int \prod_i \mathcal{D}x_i(\tau) e^{-\int d\tau S(x_i(\tau))}$$

Sum over all trajectories

- Quantum Monte Carlo : Sample the trajectories stochastically *(Cf Hao Shi's talk)*

Green functions (correlation functions)



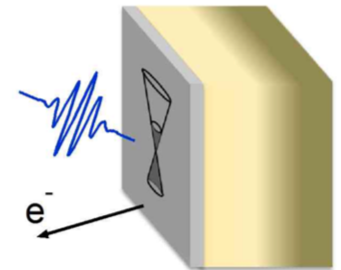
One body Green function

$$G(x - x', \tau - \tau') = \langle c(x,\tau) c^\dagger(x', \tau' = 0) \rangle$$

Two body Green function

$$G^{(2)} = \langle c(y,\tau_4) c(x,\tau_3) c^\dagger(x',\tau_2) c^\dagger(y',\tau_1) \rangle$$

- Projected "view" of the quantum many body fluid.
- Determines e.g. resistivity, photoemission, optics, …
- Strong coupling:
  infinite hierarchy of equations, no simple truncation



Photoemission (Photoelectric effect)

Quantum Embedding methods

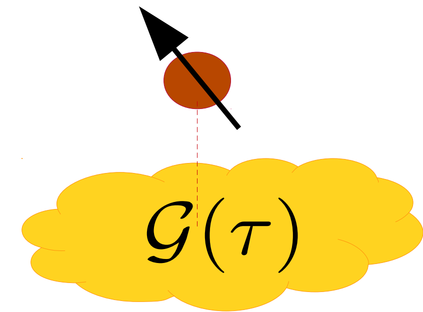
- A class of methods to compute the Green functions
- Principle: a few localised degrees of freedom in a bath of free electrons.

Weakly correlated systems

Strongly correlated systems

$$H = -\frac{\hbar^2}{2m}\nabla_x^2 + V_{effective}(x)$$

vs



1 electron in a potential

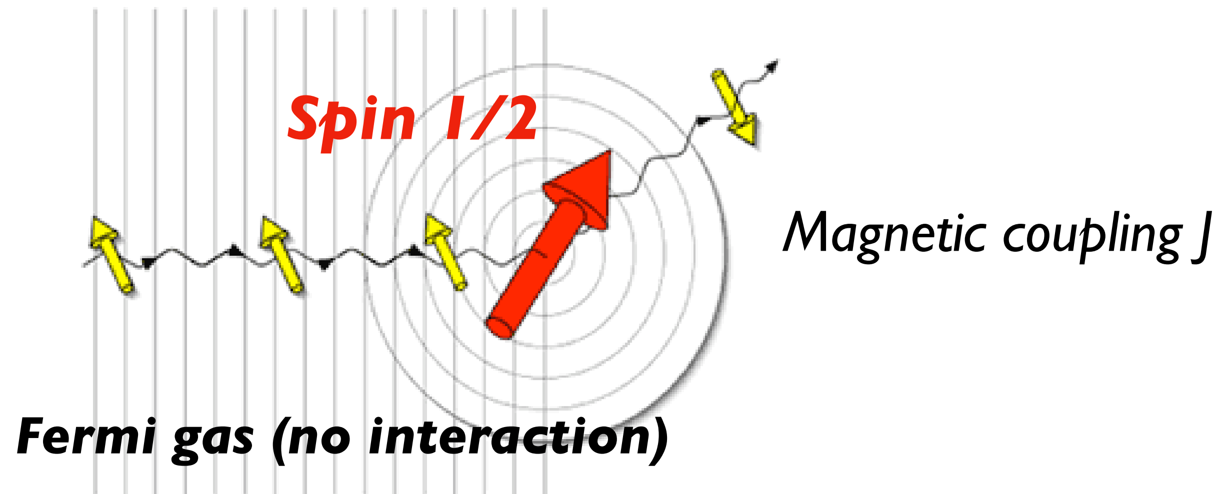
PDE

1 atom in a non-interacting bath
= impurity model

Building block of the approximation

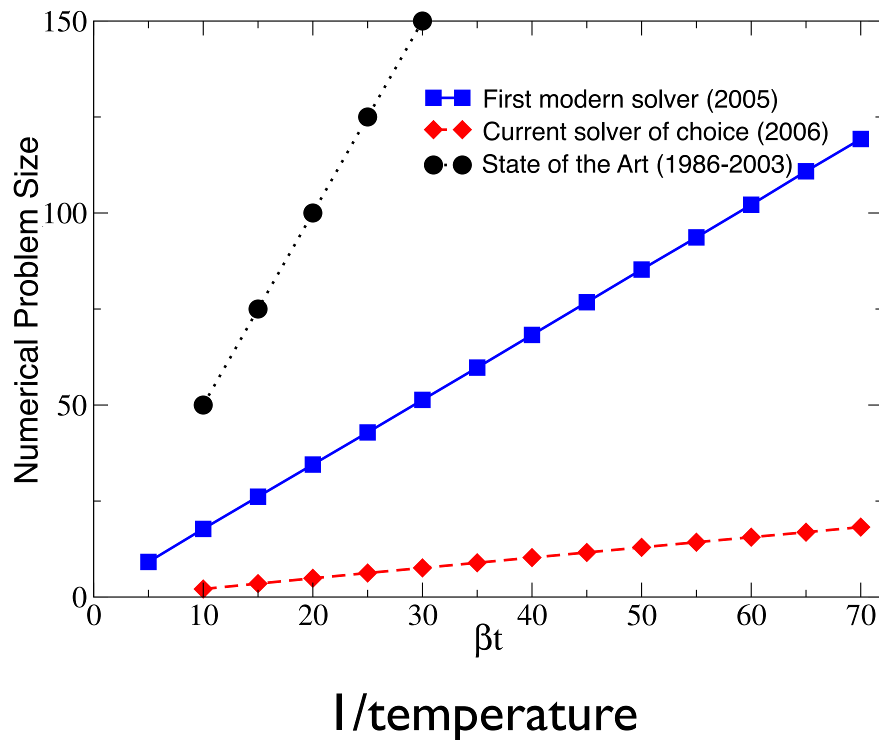
Dynamical Mean Field Theory
A. Georges, G. Kotliar, 1992

The building block is a still quantum many-body problem, but simpler.

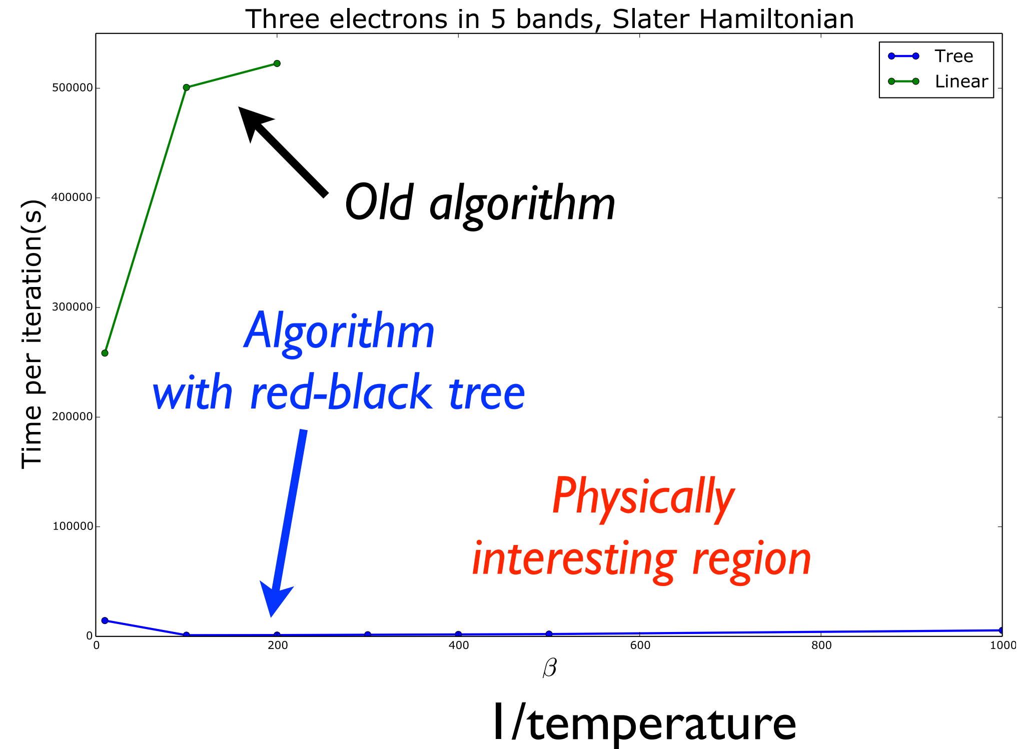


- At the frontier of solvability.
- Contains a lot of "many-body" effects.
- Strong coupling physics at low energy
  Screening/Confinement of the spin in the Fermi sea *(Kondo effect)*

Solving quantum impurity models
Progress in algorithms



1/temperature



1/temperature

- Computing time reduced by ~30^3
  25 years of Moore's law
- Beyond existing tools (linear algebra, fftw)

Summary so far

- Strongly correlated systems requires new approaches
- Quantum Embedding methods
  - Central object : Green functions
  - Building blocks : quantum impurities.
  - Solve them with HPC algorithms (e.g. Monte Carlo)
  - Assemble them in various possible ways,
    e.g. select the local degrees of freedom.

The TRIQS project

Python/C++ library

- Python for its flexibility
  - A language to assemble the building blocks, visualize.
  - Glue language with DFT electronic structure codes.
  - Team with very different skills/practices in computing

- C++ for performance and type system
  - E.g quantum impurity solver (e.g. Monte Carlo)
  - Interface is well defined. Wrapped in Python.

The TRIQS project : structure

- A main library and applications as Python modules

TRIQS based applications

Impurity solver 1

Impurity solver 2

Interface to electronic structure codes

TRIQS library

Green functions, general objets (arrays)

various solid state physics objects

Software engineering goals

- No reward in our field for young people to write/publish code.
  - Reduce development/maintenance time

- High quality code:
  - Clarity, simplicity : written to be read/understood.
  - Reusable components
  - High performance (human and machine).
- How ?
  - Libraries (std, triqs, …) make code smaller, easier to understand.
  - Code review, good practices. Coherence.

History and people

- First public version in 2012
- Contributors at CCQ, Polytechnique, Collège de France, Hamburg, Graz, ETH Zurich, Michigan University, ….

Michel Ferrero (Paris)

Markus Aichhorn (Graz)

Nils Wentzell (CCQ)

Hugo Strand (CCQ)

Manuel Zingl (CCQ)

Alice Moutenet (CCQ)

Projet management

- Open source GPL3 license.
- Version control git (Moved from svn in 2011).
- Github : distribution. Issue.
- Code Review on github.
- Continuous integration (Jenkins at Flatiron)
  Integrated with Github.
- Installation : cmake, Ubuntu packages, docker (Cf Nils' talk).

Test

- Test suite GoogleTest.
- Test coverage
  systematic for new code, improving for older parts.
- “Scientific” tests
  - Run a full calculation for a given problem
  - Quantum Monte Carlo can be delicate to test.

Documentation/Tutorials

- Documentation
    - RST (Python)
    - Build by Jenkins systems and pushed to github.io
- Contents:
    - Tutorial suite in iPython Notebooks.
      Used in Summer Schools
    - Work largely in progress. Scientists have little time
    - Writing good code or good documentation are different skills

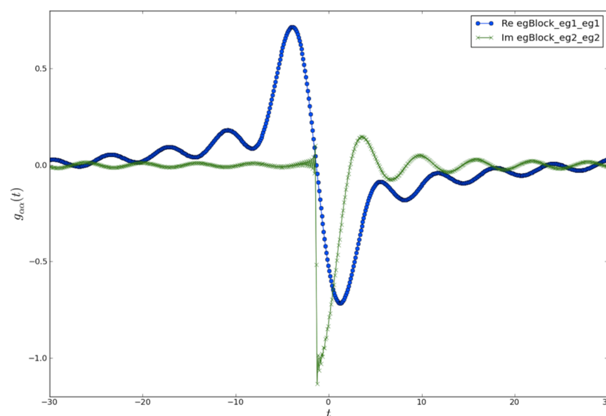
One central object of the library

Green function container

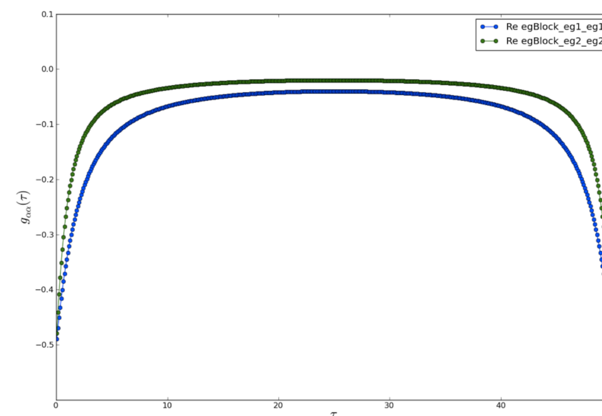
Green functions

- Functions : $G(\omega)$, $G_{ab}(x, \tau)$, $G_{ab}(k,\omega)$, $\Gamma_{abcd}(\omega,\nu,\nu')$, …
- Domains of definition (time, frequency, space)
- Mesh on the domain / representation (grid, Legendre, Chebychev)
- Target space: scalar, matrix, tensor valued.

Real time



Imaginary time



Green functions

- Multivariable Green functions
  - Cartesian product of domains/meshes, e.g.
  - $G_{ab}(x, \tau)$ is a function $D_x \times D_\tau \rightarrow Mn(R)$
  - Generic implementation (any product)
- Example

```
gf<cartesian_product<brillouin_zone, imfreq>, matrix_valued> g {...};


for (auto [k,w] : g.mesh())
   g[k,w] = 1/(w - 2*t *(cos(k[0]) + cos(k[1])));
```

$$G(k,\omega) = \frac{1}{\omega - 2t(\cos(k_0) + \cos(k_1))}$$

Green functions

- Many operations, e.g.
  - Algebra
  - Hdf5 I/O, MPI
  - Slice, partial evaluation
  - Transformation e.g. Fourier
- Difficulty
  - Make small objects that compose well.
  - Same problem as language designers, at smaller scale.
  - Learn library writing technique, e.g. notion of regular type in C++

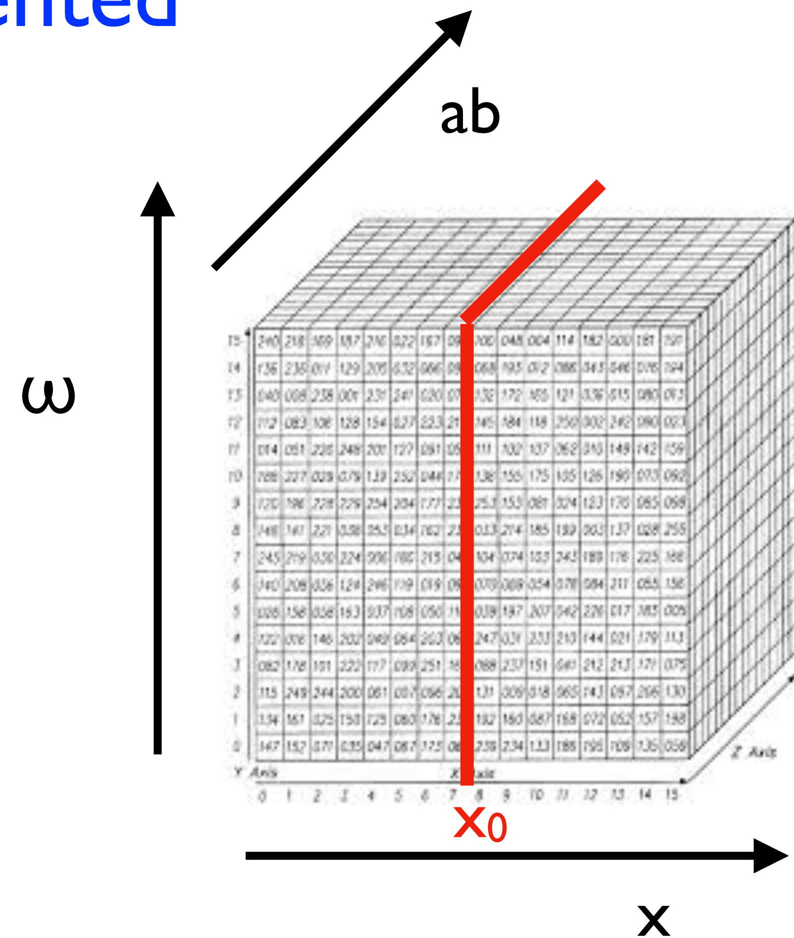
Partial evaluation

- Take a function $(x,\omega) \to G_{ab}(x,\omega)$
- Fix $x = x_0$, one gets a new function $\omega \to G_{ab}(x_0,\omega)$
- Usage :
  - We have a function *d* taking a function $\omega \to g_{ab}(\omega)$, e.g. to compute the density of fermions.
  - density vs x : $x \to d\ (\omega \to G_{ab}(x,\omega))$

```
auto g_xw = gf<cartesian_product<lattice, imfreq>, matrix_valued> {....};

density(g_xw[x0, _]);
```

Data oriented

- Green function is stored as a multidimensional array
- Partial view for fixed $x = x_0$
- Regular type gf
- View type gf_view
- Similar behaviour (generic code !), except copy
- Design questions : properties of views ? Do they own data e.g. ?



Some other components of the library

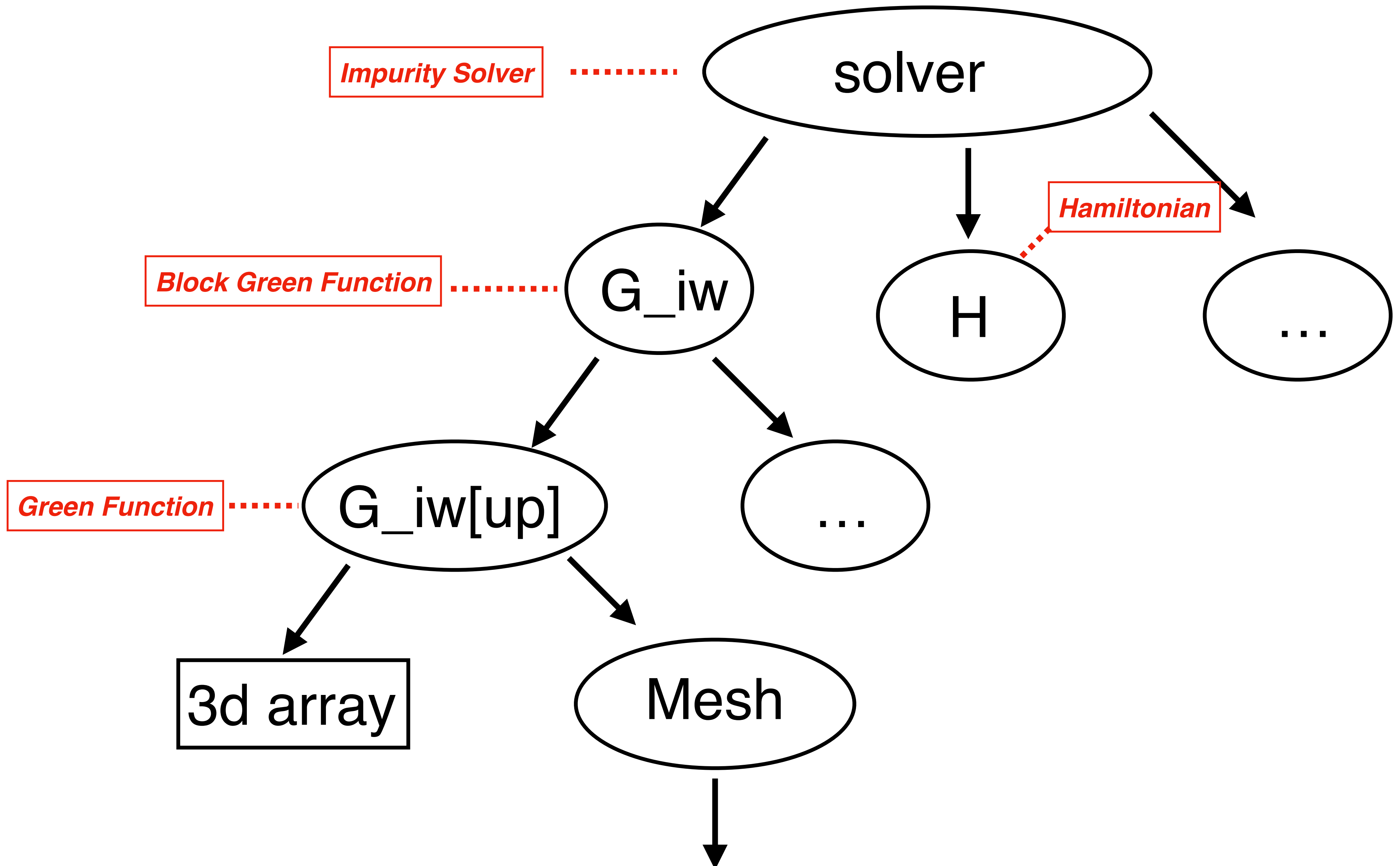
- Generic Monte Carlo
- Simple multidimensional arrays (C++)
- Many-body operators to write Hamiltonians (Python/C++)

```
H = U * c_dag(1,0) * c(1,0) * c_dag(2,0) * c(2,0)
```

- Solid state physics notions (Python/C++)
  Bravais Lattices, Brillouin zone, density of states.
- Interfaces for HDF5 (Python/C++)

HDF5

HDF5 : hierarchical tree structure, like directory



Retrieve complex object and use it

```
A = HDFArchive("dmft_bethe.h5",'r') # Open file in read mode
for it in range(21):
    if it%2: # Plot every second result
        oplot(A['G%i'%it].imag, '-o', name='G%i'%it)
```

oplot can plot many TRIQS objects via matplotlib

Retrieve Gi from the file, and use it at once

convergence

Im G1 0_0
Im G3 0_0
Im G5 0_0
Im G7 0_0
Im G9 0_0
Im G11 0_0
Im G13 0_0
Im G15 0_0
Im G17 0_0
Im G19 0_0

$G19(i\omega_n)$

ω_n

A little code sample

A taste of TRIQS in Python

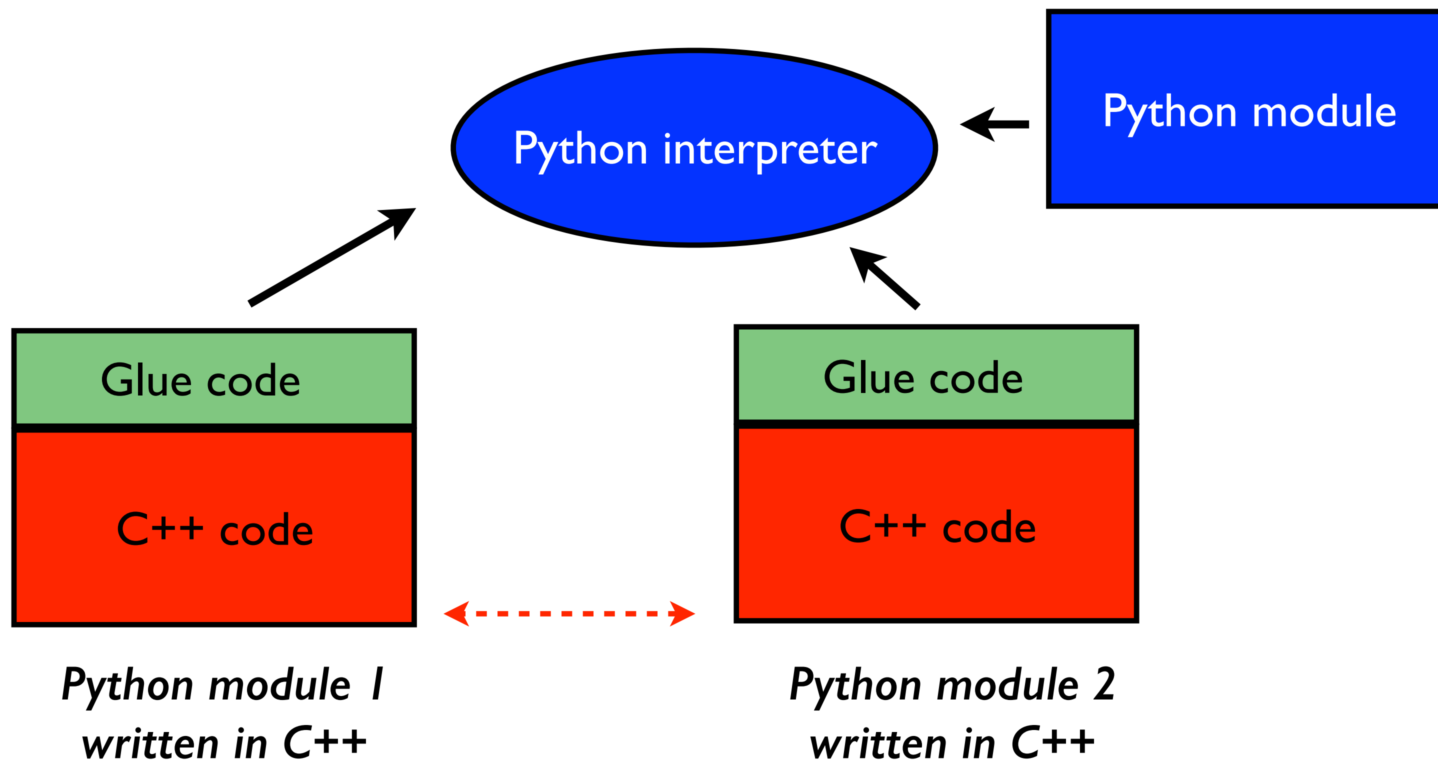
```
from pytriqs.gf import *
from ctint_tutorial import CtintSolver
from pytriqs.archive import HDFArchive


U = 2.5                # Hubbard interaction
mu = U/2.0             # Chemical potential
half_bandwidth=1.0 # Half bandwidth (energy unit)
beta = 40.0            # Inverse temperature
n_iw = 128             # Number of Matsubara frequencies
n_cycles = 10000       # Number of MC cycles
delta = 0.1            # delta parameter
n_iterations = 21      # Number of DMFT iterations

S = CtintSolver(beta, n_iw) # Initialize the solver

S.G_iw << SemiCircular(half_bandwidth) # Initialize the Green's function

for it in range(n_iterations): # DMFT loop
  for sigma, G0 in S.G0_iw:
    G0 << inverse(iOmega_n + mu - (half_bandwidth/2.0)**2 * S.G_iw[sigma] ) # Set G0

  S.solve(U, delta, n_cycles) # Solve the impurity problem

  G_sym = (S.G_iw['up']+S.G_iw['down'])/2 # Impose paramagnetic solution
  S.G_iw << G_sym

  with HDFArchive("dmft_bethe.h5",'a') as A:
     A['G%i'%it] = G_sym # Save G from every iteration to file
```

$$G_{0\sigma}^{-1}(i\omega_n) = i\omega_n + \mu - t^2 G_{c\sigma}(i\omega_n), \quad \text{for } \sigma = \uparrow, \downarrow$$

Python/C++ interface

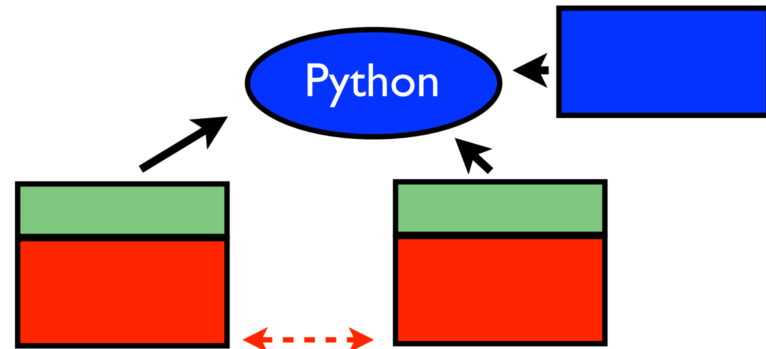
Python/C++

- C++ and Python are two quite different languages, e.g.:
  - Python : everything is a counted reference
  - C++: pointers, regular types (int, double, std::vector)
- Need some "glue" code between the C++ and Python



Python/C++

- Conversion
  - From an existing Python type to a C++ type and back
- Wrapping
  - Take a C++ class, function and make a new class, function in Python, e.g. a Green function class.
  - Wrapped type can be converted.



Requirements

- Must be automatic, specially for TRIQS applications.
  Parse the C++ code. No new code to write.

- Wrap C++ types
- Convertions
  - Library classes (e.g Green function, many-body operators)
  - array_view from/to numpy
  - Extensible : if T,U are convertible, vector<T>, tuple<U,T> too.

Tools

- Existing tools : only partial solutions to our problem
  Used swig, boost::python, cython over the years
- A little TRIQS tool : c++2py
  - Use LLVM/clang (libclang) to parse the C++ and build a representation of the C++ code.
  - Generates conversion/wrapping code accordingly.
  - Separated from TRIQS
  - Evolution : maybe reuse some newer projects for some parts (Google/CLIF, pybind11)

IPython cell magic

- Write C++ in a cell.
- Analyse the code, wrap it, compile the module and load it.

```
%reload_ext cpp2py.magic
```

```
%%cpp2py -C pytriqs
#include <triqs/gfs.hpp>
using namespace triqs::gfs;

void compute_g0(gf_view<cartesian_product<brillouin_zone, imfreq>, scalar_valued> g, double t) {
      for (auto [k,w] : g.mesh())
       g[k,w] = 1/(w - (-2*t) *(cos(k[0]) + cos(k[1])));
}
```

```
g0b = Gf(mesh=MeshProduct(kmesh, wmesh), target_shape=[])
compute_g0(g0b,t)
```

Modern C++

"Modern" C++

- C++ is evolving a lot.
- A new ISO standard every 3 years : C++11/14/17/20.

- C++ is becoming simpler for users, for library writers
  - High-level constructs like e.g. Python
  - Richer standard library
- Backward compatibility very rigorously enforced (not Python3 !)

- We use C++17 in current version.

Compilers

- TRIQS policy : use the current standard.
- Toolchains : GNU gcc, LLVM/Clang
- Implement new standards quickly (currently C++17, partly C++20).
- Quick installation of toolchains
  Docker, Singularity *(Cf. Nils Wentzell's talk yesterday)*

Subset of C++

- C++ is a multipurpose language, it support different "styles" imperative, generic, object oriented, functional …

- We use a subset of the language
  - Do not use some old features (too verbose, unsafe).
  - Modern C++ recommendations
    e.g use much less pointers, no new/delete.
  - No inheritance, no object orientation (virtual, co…)
  - Use generic programming. C++17 makes it much easier.

Tools associated with compilers

- llvm/clang toolchain.
- Clang-format : code formatting with team wide conventions.
- Bug prevention :
  - Clang-tidy, static analyser
  - Sanitizers : address, memory, thread (compiler options) Valgrind, e.g.

```
→ clang++ -fsanitize=address -g code.cpp
```

- C++ subset, automatic code rewrite : clang-tidy
- Reflection tools (libclang).

A taste of modern C++

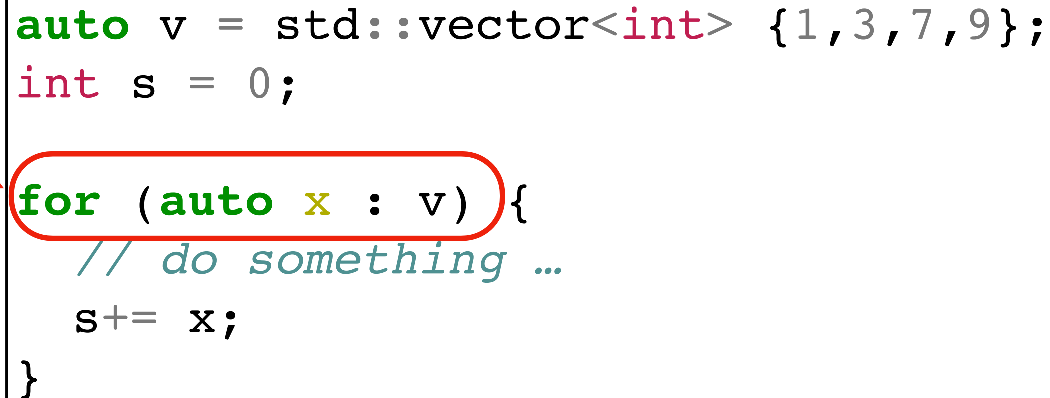
with a Pythonic angle

A simple loop

- A simple loop in Python …

```
v = [1,3,5,9]
s = 0
for x in v:
  s+=x
```

- … C++ equivalent. Main difference is types.



Simpler than what ?

- Modern C++

```
for (auto const & x : v) {
  // do something !
  s+= v;
}
```

- Intent is clearer :
  - *. Iterate on every elements in order*
  - *. v unchanged*
- As or more efficient.

- Old C++

```
for (std::vector<int>::const_iterator it=
v.begin(); it != v.end(); ++it) {
  // do something !
  s+= *it;
}
```

```
for (int i = 0; i < v.size(); ++i)
{
  // do something !
  s+= v[i];
}
```

Python style

- Python-like features can be implemented

Structured bindings in C++17

```
std::vector<int> vec2, vec1;

for (auto [n, x] : enumerate(vec1))
   // (0, x[0]), (1, x[1]), (2, x[2]), …

for (auto [x, y] : zip(vec1, vec2))
   // (x[0], y[0]), (x[1], y[1]), (x[2], y[2]), …
```

- Easy to use, less error prone.
- Implemented in TRIQS, not (yet) in standard library.
- Today, you still need a bit of expertise to write *enumerate* …

Soon

- … implementing *enumerate* will become very simple.
- Coroutines : an old idea, in progress for C++20.
- Generators like in Python. Same code, but with types.

```
def enumerate(X) :
    n=0
    for x in X:
        yield n, x
        n +=1
```

Python

```
template<typename T>
std::generator<std::pair<n, typename T::value_type>>

enumerate(T const & x) {
  int n=0;
  for (auto const & y : x) {
    co_yield std::pair{n,y};
    n++;
  }
}
```

Future C++ e.g. in clang -fcoroutines

Zero cost abstraction

A simple example

What is zero cost abstraction ?

- What is *simple* should be coded *simply*
- High level and yet fast.
- Very important for readability, long term maintenance, code review.

- Common wrong idea : compact, simple, readable code is slow.
- We want simplicity (abstraction), without any performance penalty (at zero cost).
- Generic programming is essential to achieve this.
  C++17, C++20 make it a lot easier.

Motivation

- A, B, C, Z: arrays of rank 5 e.g. We want to say

```
Z = A + B + C / 2;
```

- Naive object oriented way :
  - Each addition makes a new array
  - Slow : a lot of temporaries and loops !

$$Z = \underbrace{A + \underbrace{B + \underbrace{C/2}}}$$

Motivation

- A, B, C, Z: arrays of rank 5 e.g. We want to say

```
Z = A + B + C / 2;
```

- A basic answer is : write all the loops !

```
for (int i = 0; i < b1; ++i)
 for (int j = 0; j < b2; ++j)
  for (int k = 0; k < b4; ++k)
   for (int l = 0; l < b3; ++l)
    for (int m = 0; m < b5; ++m) {
     Z(i, j, k, l, m) =
         A(i, j, k, l, m) + B(i, j, k, l, m) + C(i, j, k, l, m) / 2;
    }
```

- Error prone, hard to read and code review.
- The compiler should do this for us !

Other example

- With our multidimensional array class

```
auto a = array<double, 3>(5, 2, 2);    // Declare a 5x2x2 array of double

sum(a * a);            // Sum all the square elements

max_element(abs(a)); // maximum of the absolute value of the array
```

- Rewriting it manually requires the code of *sum*

Let us consider a toy model.

The puzzle

- A and B : two matrices n x n, real valued.
  A function ***trace***
- We want to write

```
double r = trace (A + B);
```

- Instead of

```
double r = 0;
for (int i = 0; i < n; ++i)
      r += A(i, i) + B(i, i);
```

- A priori, zero cost abstraction seems impossible:
  - A + B computed first, before calling `trace`.
  - Scales as N^2 while hand-written code is N

The trace function

- Assume we have a *square_matrix* class
- Let us implement the trace

```
double trace (square_matrix const & m) {
 double r = 0;
 int d = dim(m); // size of the matrix d x d
 for (int i=0; i<d; ++i) r += m(i,i);
 return r;
}
```

- Only things I used here :
  - m(i,j) returns the value of the matrix m_{ij}
  - dim(m) returns the dimension

Generic programming

- A generic version of the function

```
template<typename M>
double trace (M const & m) {
 double r = 0;
 int d = dim(m); // size of the matrix d x d
 for (int i=0; i<d; ++i) r += m(i,i);
 return r;
}
```

- What can M be ?
  - m(i,j) returns the value of the matrix m_{ij}
  - dim(m) returns the dimension
- *trace* makes sense (i.e. compiles) only when these constraints on M are true

A few matrix classes

- A simple square matrix

```
class square_matrix {
 int n;
 std::vector<double> data;

 public:
 square_matrix(int n);

 double operator()(int i, int j) const { return data[i + n * j]; }
 friend int dim(square_matrix const& m) { return m.n; }
 // …
};
```

- A matrix whose form is known analytically.

```
struct hilbert_matrix {
 int n;
 double operator()(int i, int j) const { return 1.0 / (i + j + 1); }
 friend int dim(hilbert_matrix const& m) { return m.n; }
};
```

Back to our question

```
double r = trace (A + B);
```

- The sum of 2 matrices is a lazy object that :
  just keeps a reference to A, B
  evaluates the actual sum only on demand.

```
template <typename A, typename B> struct lazy_addition {
 A const & a;
 B const & b;
 double operator()(int i, int j) const { return a(i, j) + b(i, j); }
 friend int dim(lazy_add const& x) { return dim(x.a); }
};
```

```
template <typename A, typename B>
lazy_addition<A, B> operator+(A const& a, B const& b){
 return {a, b};
}
```

- The addition is too general, it takes any type ! Cf later…

What does the compiler do ?

```
double trace(lazy_addition const& m) {
 auto r = m(0, 0);
 int d = dim(m);
 for (int i = 1; i < d; ++i) r += m(i, i);
 return r;
}
```

```
template <Matrix A, Matrix B>
struct lazy_addition {
 A const& a;
 B const& b;

 double operator()(int i, int j)
const { return a(i, j) + b(i, j); }
```

- Replace and inline calls

```
double trace(lazy_addition const& m) {
 auto r = 0;
 int d = dim(m.a);
 for (int i = 0; i < d; ++i) r += m.a(i, i) + m.b(i,i);
 return r;
}
```

- The compiler rewrites the code for us
  - Exactly the hand written code
  - Scales like N, not N^2

Let us check

- Compare 3 code snippets (with Google Benchmarks)
  - With Trace (TRIQS library)

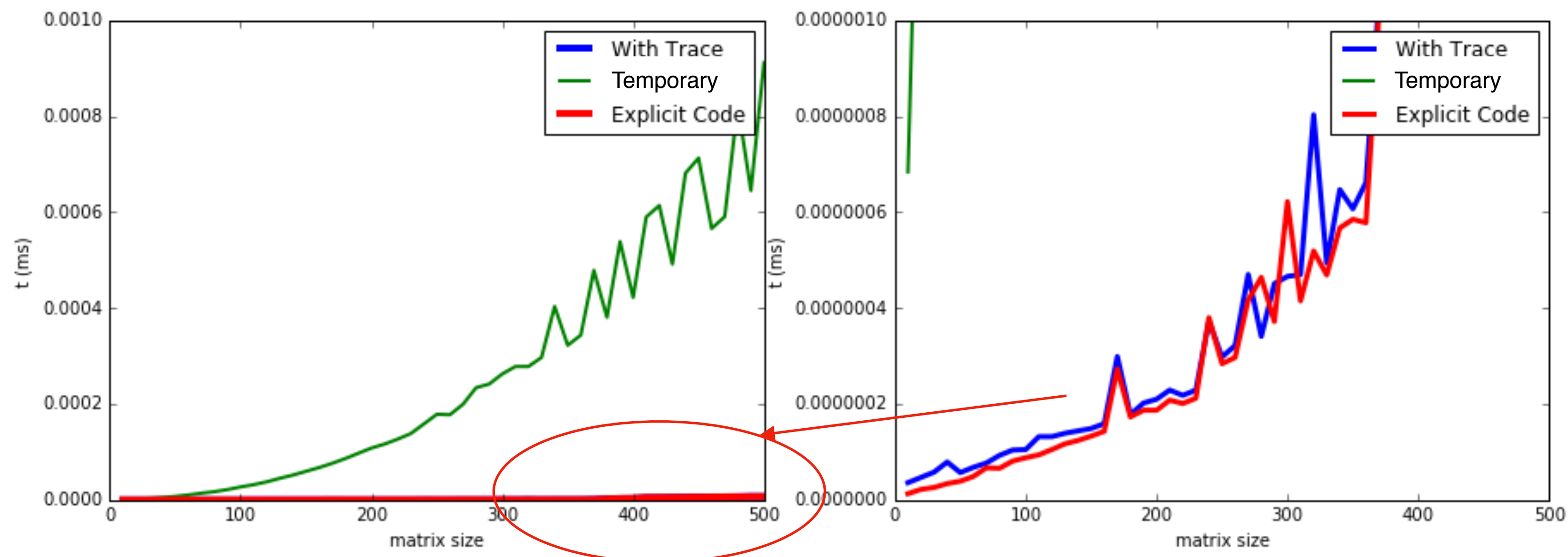
```
auto r = trace(A + B);
```

  - Explicit code (hand written)

```
for (int i = 0; i < N; ++i) r += A(i, i) + B(i, i);
```

  - Force temporary

```
auto r = trace(square_matrix{A + B});
```



The notion of concept

- A concept is a set of requirements on a type
- Example: Matrix

  The category of types that behave like a square matrix (of double)

  - m(i,j) returns the value of the matrix m_{ij}
  - dim(m) returns the dimension

- Optionally : in C++20, tell the compiler (already in gcc)

```
template <typename T> concept bool Matrix = requires(T m) {
  { m(0, 0)} -> double;
  { dim(m) } -> int;
 };
```

Finally

- The addition was too general, it took any type. Let's fix it.

```
template <Matrix A, Matrix B>
lazy_addition<A, B> operator+(A const& a, B const& b){
 return {a, b};
}
```

- Same thing for the trace

```
template<Matrix M>
double trace (M const & m) {
//…
}
```

- Compiler will issue clear error messages in other cases.
- No more long error message of template C++ code, including STL.

Analogy with mathematics

- Math
  - Notion of group.
  - General theorem that apply for every group
- Programming
  - Notion of concepts.
  - General algorithms that apply for every type which model the concept.
- Library design :
  - Find the most fruitful concepts for our field (e.g. solid state physics, quantum many-body problem)
  - A hierarchy of concepts, real type as leaf.
Similar to Julia type system

Continue analogy

- The category of Matrix types is closed under addition.

  Matrix + Matrix → Matrix

- square_matrix is not :
  square_matrix + square_matrix != square_matrix

Conclusion

- TRIQS as a library for quantum many-body problem
- Current developments
  - Scale up
  - Documentation.
  - More applications, more components in the library

Thank you for your attention