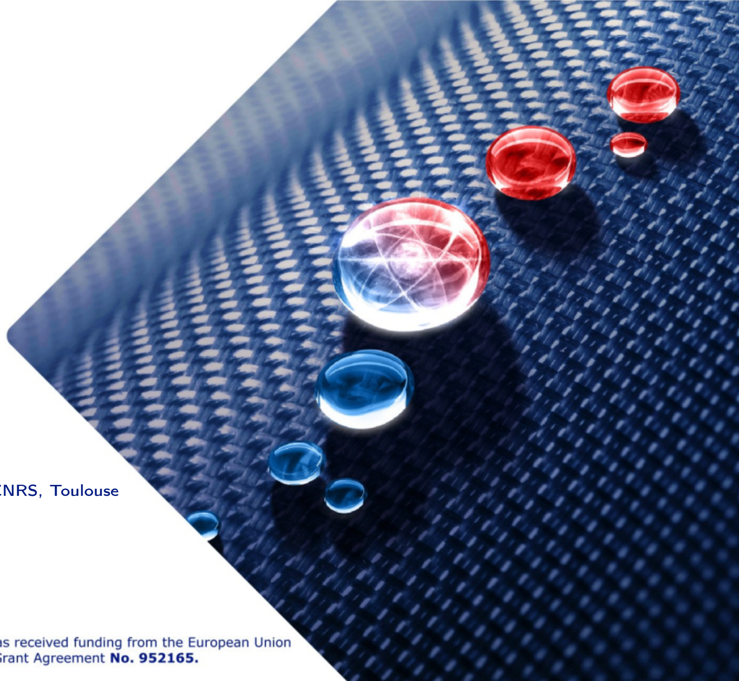
# Library development within TREX

Anthony Scemama

21/04/2021

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse (France)

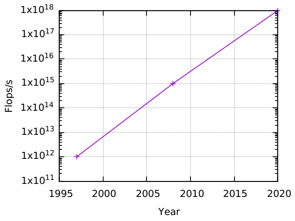


Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizoon 2020 research and innovation programme under Grant Agreement **No. 952165.**

## Presentation of TREX

## Worldwide technological competition

- 1997 : Teraflops/s[1]
- 2008 : Petaflops/s
- 2020? : Exaflops/s



- Expected increase of computational power is *exponential*
- Moore's Law is ending
- Technological breakthrough needed (quantum computing?)

[1]flops/s: floating point operations per second

## The supercomputing race

### Worldwide technological competition

- 1997 : Terascale : Distributed parallelism
- 2008 : Petascale : Multi-core chips or accelerators
- 2020? : Exascale : Hybrid architectures are inevitable

### Peak flops/s improved by $1000\times$. What about

- Memory capacity per core?
- Memory bandwidth? latency?
- I/O bandwidth? latency?
- Network bandwidth? latency?

## Exascale

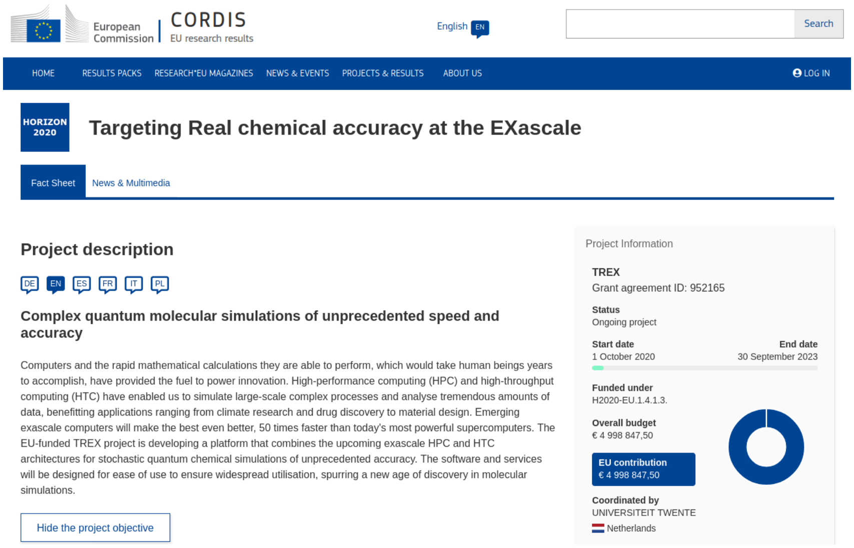
### Transition to exascale will be painful

- Network becomes slow *vs* computation
- Memory per core decreases
- Heterogeneous machines (accelerators)
- Need to find even more fine-grained parallelism

### Very few applications will scale

- Exascale machines will run high throughput computing (HTC) workloads

# The TREX CoE



European Commission | CORDIS EU research results

English EN | Search

HOME RESULTS PACKS RESEARCH*EU MAGAZINES NEWS & EVENTS PROJECTS & RESULTS ABOUT US LOG IN

HORIZON 2020

## Targeting Real chemical accuracy at the EXascale

Fact Sheet | News & Multimedia

### Project description

DE EN ES FR IT PL

**Complex quantum molecular simulations of unprecedented speed and accuracy**

Computers and the rapid mathematical calculations they are able to perform, which would take human beings years to accomplish, have provided the fuel to power innovation. High-performance computing (HPC) and high-throughput computing (HTC) have enabled us to simulate large-scale complex processes and analyse tremendous amounts of data, benefitting applications ranging from climate research and drug discovery to material design. Emerging exascale computers will make the best even better, 50 times faster than today's most powerful supercomputers. The EU-funded TREX project is developing a platform that combines the upcoming exascale HPC and HTC architectures for stochastic quantum chemical simulations of unprecedented accuracy. The software and services will be designed for ease of use to ensure widespread utilisation, spurring a new age of discovery in molecular simulations.

Hide the project objective

Project Information

**TREX**
Grant agreement ID: 952165

**Status**
Ongoing project

**Start date**
1 October 2020

**End date**
30 September 2023

**Funded under**
H2020-EU.1.4.1.3.

**Overall budget**
€ 4 998 847,50

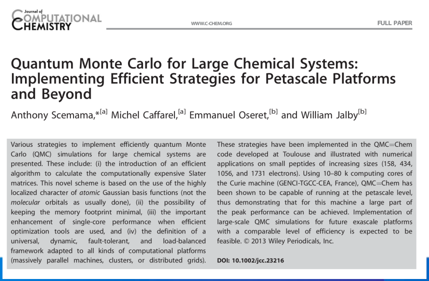
**EU contribution**
€ 4 998 847,50

**Coordinated by**
UNIVERSITEIT TWENTE
Netherlands

## Quantum Monte Carlo (QMC)

### QMC

- Extremely precise model
- Expensive in CPU
- Fully parallel
- Perfectly well adapted to HPC (in 2011, 0.96 PFlops/s)



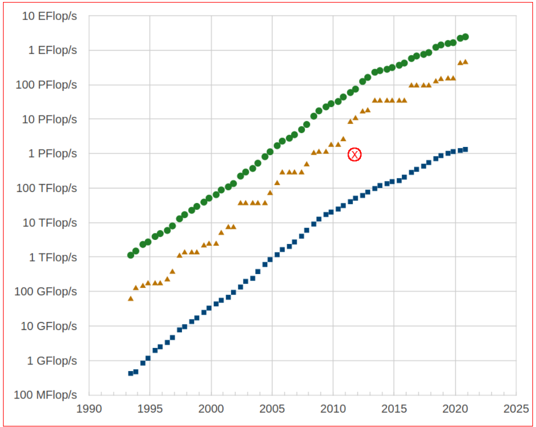
Journal of COMPUTATIONAL CHEMISTRY WWW.C-CHEM.ORG FULL PAPER

**Quantum Monte Carlo for Large Chemical Systems: Implementing Efficient Strategies for Petascale Platforms and Beyond**

Anthony Scemama,*[a] Michel Caffarel,[a] Emmanuel Oseret,[b] and William Jalby[b]

Various strategies to implement efficiently quantum Monte Carlo (QMC) simulations for large chemical systems are presented. These include: (i) the introduction of an efficient algorithm to calculate the computationally expensive Slater matrices. This novel scheme is based on the use of the highly localized character of *atomic* Gaussian basis functions (not the *molecular* orbitals as usually done), (ii) the possibility of keeping the memory footprint minimal, (iii) the important enhancement of single-core performance when efficient optimization tools are used, and (iv) the definition of a universal, dynamic, fault-tolerant, and load-balanced framework adapted to all kinds of computational platforms (massively parallel machines, clusters, or distributed grids). These strategies have been implemented in the QMC=Chem code developed at Toulouse and illustrated with numerical applications on small peptides of increasing sizes (158, 434, 1056, and 1731 electrons). Using 10–80 k computing cores of the Curie machine (GENCI-TGCC-CEA, France), QMC=Chem has been shown to be capable of running at the petascale level, thus demonstrating that for this machine a large part of the peak performance can be achieved. Implementation of large-scale QMC simulations for future exascale platforms with a comparable level of efficiency is expected to be feasible. © 2013 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23216



## QMC in a few words

Stochastic solution of the electronic Schrödinger equation (nuclei are fixed):

$$
\begin{aligned}
E &= \frac{\langle \Phi | \hat{H} | \Phi \rangle}{\langle \Phi | \Phi \rangle} = \frac{\langle \Phi | \hat{H} | \Psi \rangle}{\langle \Phi | \Psi \rangle} = \frac{\int \Phi(\mathrm{r}_1, \ldots, \mathrm{r}_N) \hat{H} \Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)\, \mathrm{d}\mathrm{r}_1 \ldots \mathrm{d}\mathrm{r}_N}{\int \Phi(\mathrm{r}_1, \ldots, \mathrm{r}_N) \Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)\, \mathrm{d}\mathrm{r}_1 \ldots \mathrm{d}\mathrm{r}_N} \\
&= \frac{\int \left[\Phi(\mathrm{r}_1, \ldots, \mathrm{r}_N) \Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)\right] \frac{\hat{H}\Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)}{\Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)}\, \mathrm{d}\mathrm{r}_1 \ldots \mathrm{d}\mathrm{r}_N}{\int \left[\Phi(\mathrm{r}_1, \ldots, \mathrm{r}_N) \Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)\right] \mathrm{d}\mathrm{r}_1 \ldots \mathrm{d}\mathrm{r}_N} \\
&\sim \frac{1}{M} \sum_M \frac{\hat{H}\Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)}{\Psi(\mathrm{r}_1, \ldots, \mathrm{r}_N)}, \text{ sampled with } (\Psi \times \Phi)
\end{aligned}
$$

| | |
|---|---|
| $\hat{H}$: | Hamiltonian |
| $E$: | Energy |
| $\mathrm{r}_1, \ldots, \mathrm{r}_N$: | Electron coordinates |
| $\Phi$: | Quasi-exact (fixed-node) wave function |
| $\Psi$: | Trial wave function |

## QMC in a few words

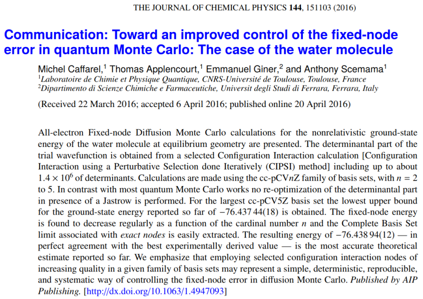
### In practice

- Walker: vector $(r_1, \ldots, r_N) \in \mathbb{R}^{3N}$ of electron coordinates
- Diffusion + drift with a birth/death process to sample the 3N-dimensional density $(\Psi \times \Phi)$
- At each step, $E_{\text{loc}}(r_1, \ldots, r_N) = \frac{\hat{H}\Psi(r_1,\ldots,r_N)}{\Psi(r_1,\ldots,r_N)}$ is computed
- The total energy is the the average of all the computed $E_{\text{loc}}$

### HPC

- Very low memory requirements (no integrals)
- Distribute walkers on different cores or compute nodes
- No blocking communication: near-ideal scaling
- Difficulty: parallelize within a QMC trajectory

## Fixed-node approximation


THE JOURNAL OF CHEMICAL PHYSICS **144**, 151103 (2016)

**Communication: Toward an improved control of the fixed-node error in quantum Monte Carlo: The case of the water molecule**

Michel Caffarel,[1] Thomas Applencourt,[1] Emmanuel Giner,[2] and Anthony Scemama[1]
[1]*Laboratoire de Chimie et Physique Quantique, CNRS-Université de Toulouse, Toulouse, France*
[2]*Dipartimento di Scienze Chimiche e Farmaceutiche, Universit degli Studi di Ferrara, Ferrara, Italy*

(Received 22 March 2016; accepted 6 April 2016; published online 20 April 2016)

All-electron Fixed-node Diffusion Monte Carlo calculations for the nonrelativistic ground-state energy of the water molecule at equilibrium geometry are presented. The determinantal part of the trial wavefunction is obtained from a selected Configuration Interaction calculation [Configuration Interaction using a Perturbative Selection done Iteratively (CIPSI) method] including up to about $1.4 \times 10^6$ of determinants. Calculations are made using the cc-pCV*n*Z family of basis sets, with $n = 2$ to 5. In contrast with most quantum Monte Carlo works no re-optimization of the determinantal part in presence of a Jastrow is performed. For the largest cc-pCV5Z basis set the lowest upper bound for the ground-state energy reported so far of −76.437 44(18) is obtained. The fixed-node energy is found to decrease regularly as a function of the cardinal number *n* and the Complete Basis Set limit associated with *exact nodes* is easily extracted. The resulting energy of −76.438 94(12) — in perfect agreement with the best experimentally derived value — is the most accurate theoretical estimate reported so far. We emphasize that employing selected configuration interaction nodes of increasing quality in a given family of basis sets may represent a simple, deterministic, reproducible, and systematic way of controlling the fixed-node error in diffusion Monte Carlo. *Published by AIP Publishing.* [http://dx.doi.org/10.1063/1.4947093]

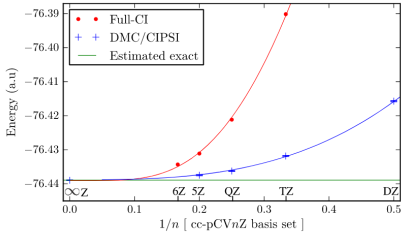
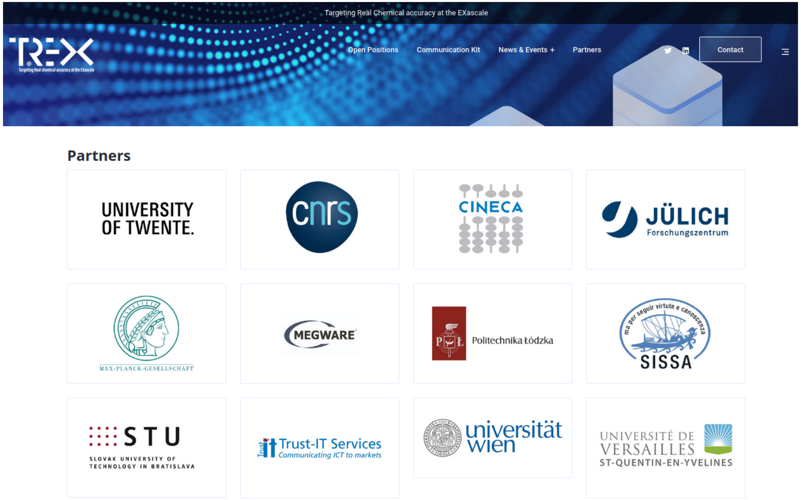


FIG. 1. CBS extrapolation of FCI and DMC/CIPSI energies. Error bars on DMC data are plotted but almost imperceptible.

- As $(\Psi \times \Phi)$ is a probability density, $(\Psi \times \Phi) \geq 0$ so $\Phi$ has the same sign as $\Psi$: fixed-node (FN) approximation.
- FN: only approximation of QMC. If the nodes of $\Psi$ coincide with the exact nodes, we obtain the exact energy
- Using increasingly large sCI determinant expansions (CIPSI), the fixed-node error can be controlled

## The TREX CoE



**Codes**

- CHAMP
- QMC=Chem
- TurboRVB
- NECI
- Quantum Package
- GammCor
- QML

## The TREX CoE

- TREX CoE: Targeting REal chemical accuracy at the eXascale
- Started in Oct. 2020
- Objective: Make codes ready for exascale systems
- Two regimes:
  - Single exascale run
  - Thousands of petascale simulations in high-throughput (HTC)
- How: Instead of re-writing codes, provide libraries
  - One library for high-performance (QMCkl)
  - One library for exchanging information between codes (input of QMC is $\Psi$)

## QMC kernel library (QMCkl)

## Programming for the exascale

- Progress in quantum chemistry may require codes with new ideas/algorithms
- New ideas/algorithms are implemented by physicists/chemists
- Different scientists have different programming language knowledge/preference
- Exascale machines will be horribly complex to program

**Question**

Is it reasonable to ask physicists/chemists to write codes for exascale machines?

## No: Proof

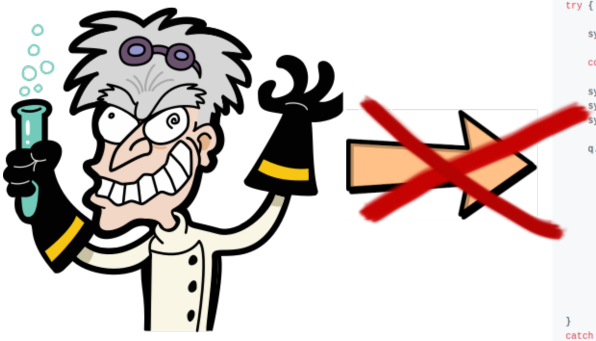
(from https://github.com/jeffhammond/dpcpp-tutorial)

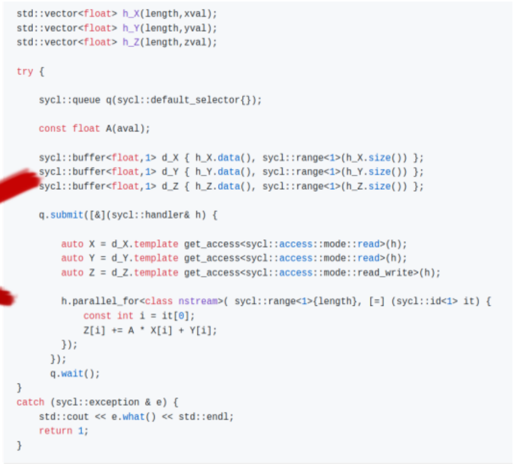
### Vector addition

```
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```

```
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

# No



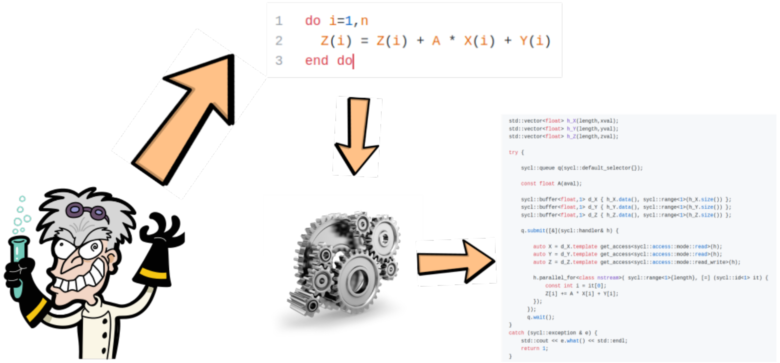


```
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

## The dream

A compiler[2] that can read an average researcher's code and transform it into highly efficient code on an exascale machine.



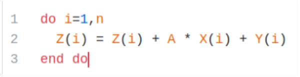
```fortran
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```

```cpp
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>(length), [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

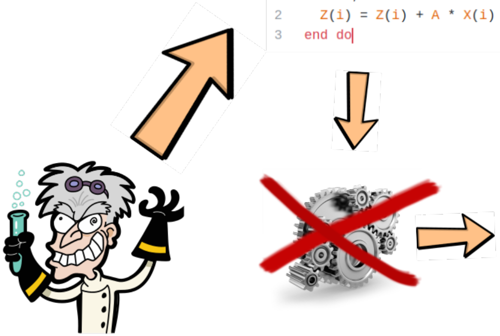
[2]Wikipedia: A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language)

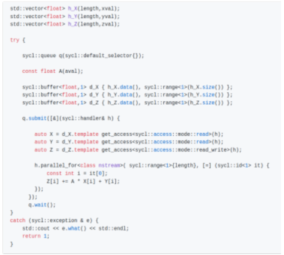
## Reality

*Artificial Intelligence* is not ready yet . . .



```fortran
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```



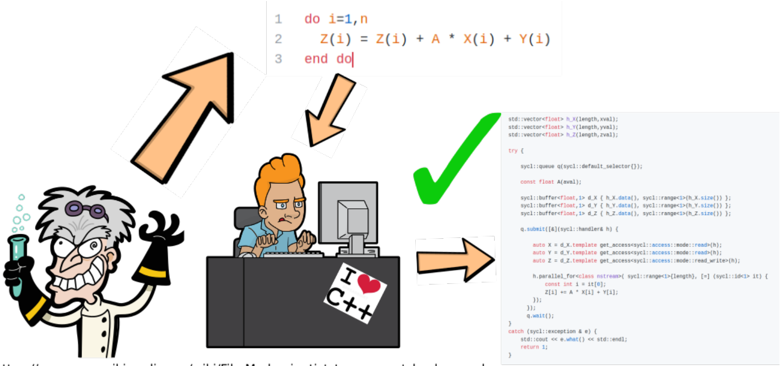


```cpp
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

# Reality

. . . so let's use *Natural Intelligence* and add a human layer between the machine and the researchers : a bio-compiler



```fortran
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```

```cpp
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

## Strategy

- Identify the common computational kernels of QMC
- Implement these kernels in a human-readable library (QMC experts)
- *Bio-compile* the human-readable library in a HPC-library (HPC experts)
- Scientists can link either library with their codes

## Benefits of this model

### For scientists

- We don't impose a programming language
- The code can stay easy to understand by the physicists/chemists
  Performance-related aspects are delegated to the library
- Codes will not die with a change in architecture
- Scientific code development does not break the performance
- Scientists don't lose control on their codes

### Separation of concerns

- Scientists will never have to manipulate low-level HPC code
- HPC experts will not be required to be experts in theoretical physics
- Better re-use of the optimization effort among the community

## QMCkl documentation library

- The API is C-compatible: QMCkl appears like a C library $\Longrightarrow$ can be used in all other languages
- System functions in C (memory allocation, thread safety, *etc*)
- Computational kernels in Fortran for readability
- A lot of documentation (remember: the HPC compiler is a human!)

## Literate programming

*Literate programming is a programming paradigm introduced by Donald Knuth in which a computer program is given an explanation of its logic in a natural language, such as English, interspersed with snippets of macros and traditional source code, from which compilable source code can be generated. (Wikipedia)*

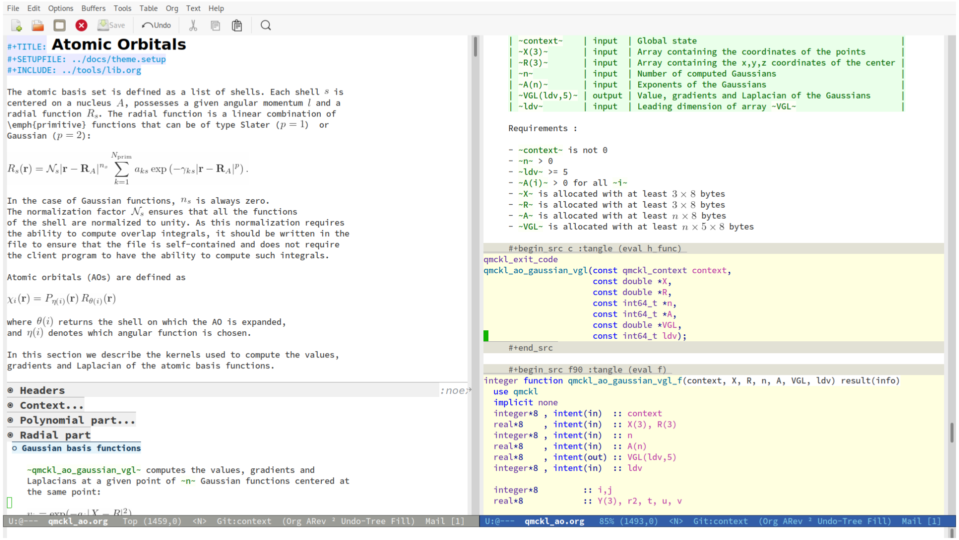
## Documentation library

Literate programming with *org-mode*:

- Here, comments are more important than code
- Can add graphics, $\LaTeX$formulas, tables, etc
- Documentation always synchronized with the code
- Some routines can be generated by embedded scripts
- Web site auto-generated when code is pushed
- Most of the the first EU report was auto-generated from the documentation

Instead of writing comments documenting code, we write code illustrating documentation.

# Literate programming with *org-mode*



```
#+TITLE: Atomic Orbitals
#+SETUPFILE: ../docs/theme.setup
#+INCLUDE: ../tools/lib.org
```

The atomic basis set is defined as a list of shells. Each shell $s$ is
centered on a nucleus $A$, possesses a given angular momentum $l$ and a
radial function $R_s$. The radial function is a linear combination of
\emph{primitive} functions that can be of type Slater ($p=1$) or
Gaussian ($p=2$):

$$R_s(\mathbf{r}) = \mathcal{N}_s |\mathbf{r}-\mathbf{R}_A|^{n_s} \sum_{k=1}^{N_\text{prim}} a_{ks} \exp\left(-\gamma_{ks} |\mathbf{r}-\mathbf{R}_A|^p\right).$$

In the case of Gaussian functions, $n_s$ is always zero.
The normalization factor $\mathcal{N}_s$ ensures that all the functions
of the shell are normalized to unity. As this normalization requires
the ability to compute overlap integrals, it should be written in the
file to ensure that the file is self-contained and does not require
the client program to have the ability to compute such integrals.

Atomic orbitals (AOs) are defined as

$$\chi_i(\mathbf{r}) = P_{\eta(i)}(\mathbf{r})\, R_{\theta(i)}(\mathbf{r})$$

where $\theta(i)$ returns the shell on which the AO is expanded,
and $\eta(i)$ denotes which angular function is chosen.

In this section we describe the kernels used to compute the values,
gradients and Laplacian of the atomic basis functions.

- ⊙ **Headers** :noe
- ⊙ **Context...**
- ⊙ **Polynomial part...**
- ⊙ **Radial part**
  - ○ **Gaussian basis functions**

~qmckl_ao_gaussian_vgl~ computes the values, gradients and
Laplacians at a given point of ~n~ Gaussian functions centered at
the same point:

$$v_i = \exp(-a_i |X-R|^2)$$

```
U:@---  qmckl_ao.org   Top (1459,0)  <N>  Git:context  (Org ARev ² Undo-Tree Fill)  Mail [1]
```

```
| ~context~      | input  | Global state                                       |
| ~X(3)~         | input  | Array containing the coordinates of the points     |
| ~R(3)~         | input  | Array containing the x,y,z coordinates of the center |
| ~n~            | input  | Number of computed Gaussians                       |
| ~A(n)~         | input  | Exponents of the Gaussians                         |
| ~VGL(ldv,5)~   | output | Value, gradients and Laplacian of the Gaussians    |
| ~ldv~          | input  | Leading dimension of array ~VGL~                   |
```

Requirements :

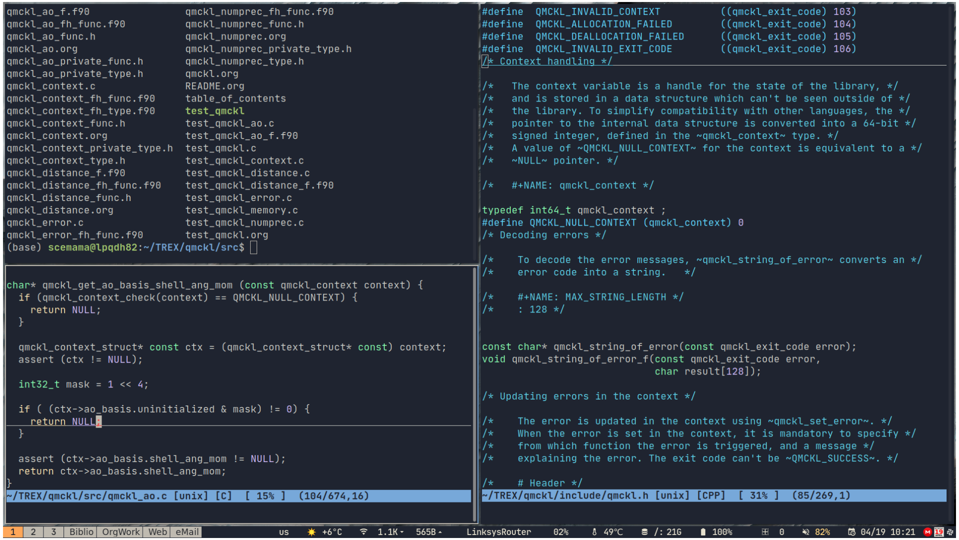
- ~context~ is not 0
- ~n~ > 0
- ~ldv~ >= 5
- ~A(i)~ > 0 for all ~i~
- ~X~ is allocated with at least $3 \times 8$ bytes
- ~R~ is allocated with at least $3 \times 8$ bytes
- ~A~ is allocated with at least $n \times 8$ bytes
- ~VGL~ is allocated with at least $n \times 5 \times 8$ bytes

```
    #+begin_src c :tangle (eval h_func)
qmckl_exit_code
qmckl_ao_gaussian_vgl(const qmckl_context context,
                      const double *X,
                      const double *R,
                      const int64_t *n,
                      const int64_t *A,
                      const double *VGL,
                      const int64_t ldv);
    #+end_src
```

```
    #+begin_src f90 :tangle (eval f)
integer function qmckl_ao_gaussian_vgl_f(context, X, R, n, A, VGL, ldv) result(info)
  use qmckl
  implicit none
  integer*8 , intent(in)  :: context
  real*8    , intent(in)  :: X(3), R(3)
  integer*8 , intent(in)  :: n
  real*8    , intent(in)  :: A(n)
  real*8    , intent(out) :: VGL(ldv,5)
  integer*8 , intent(in)  :: ldv

  integer*8        :: i,j
  real*8           :: Y(3), r2, t, u, v
```

```
U:@---  qmckl_ao.org   85% (1493,0)  <N>  Git:context  (Org ARev ² Undo-Tree Fill)  Mail [1]
```

# Generated code



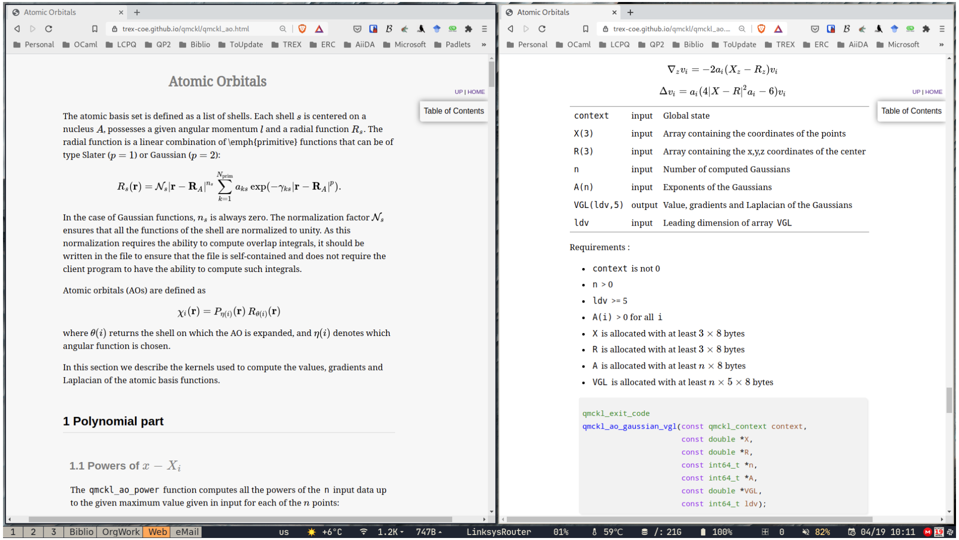
```
qmckl_ao_f.f90                qmckl_numprec_fh_func.f90
qmckl_ao_fh_func.f90          qmckl_numprec_func.h
qmckl_ao_func.h               qmckl_numprec.org
qmckl_ao.org                  qmckl_numprec_private_type.h
qmckl_ao_private_func.h       qmckl_numprec_type.h
qmckl_ao_private_type.h       qmckl.org
qmckl_context.c               README.org
qmckl_context_fh_func.f90     table_of_contents
qmckl_context_fh_type.f90     test_qmckl
qmckl_context_func.h          test_qmckl_ao.c
qmckl_context.org             test_qmckl_ao_f.f90
qmckl_context_private_type.h  test_qmckl.c
qmckl_context_type.h          test_qmckl_context.c
qmckl_distance_f.f90          test_qmckl_distance.c
qmckl_distance_fh_func.f90    test_qmckl_distance_f.f90
qmckl_distance_func.h         test_qmckl_error.c
qmckl_distance.org            test_qmckl_memory.c
qmckl_error.c                 test_qmckl_numprec.c
qmckl_error_fh_func.f90       test_qmckl.org
(base) scemama@lpqdh82:~/TREX/qmckl/src$
```

```c
char* qmckl_get_ao_basis_shell_ang_mom (const qmckl_context context) {
  if (qmckl_context_check(context) == QMCKL_NULL_CONTEXT) {
    return NULL;
  }

  qmckl_context_struct* const ctx = (qmckl_context_struct* const) context;
  assert (ctx != NULL);

  int32_t mask = 1 << 4;

  if ( (ctx->ao_basis.uninitialized & mask) != 0) {
    return NULL;
  }

  assert (ctx->ao_basis.shell_ang_mom != NULL);
  return ctx->ao_basis.shell_ang_mom;
}
~/TREX/qmckl/src/qmckl_ao.c [unix] [C]   [ 15% ]  (104/674,16)
```

```c
#define  QMCKL_INVALID_CONTEXT          ((qmckl_exit_code) 103)
#define  QMCKL_ALLOCATION_FAILED        ((qmckl_exit_code) 104)
#define  QMCKL_DEALLOCATION_FAILED      ((qmckl_exit_code) 105)
#define  QMCKL_INVALID_EXIT_CODE        ((qmckl_exit_code) 106)
/* Context handling */

/*   The context variable is a handle for the state of the library, */
/*   and is stored in a data structure which can't be seen outside of */
/*   the library. To simplify compatibility with other languages, the */
/*   pointer to the internal data structure is converted into a 64-bit */
/*   signed integer, defined in the ~qmckl_context~ type. */
/*   A value of ~QMCKL_NULL_CONTEXT~ for the context is equivalent to a */
/*   ~NULL~ pointer. */

/*   #+NAME: qmckl_context */

typedef int64_t qmckl_context ;
#define QMCKL_NULL_CONTEXT (qmckl_context) 0
/* Decoding errors */

/*    To decode the error messages, ~qmckl_string_of_error~ converts an */
/*    error code into a string.   */

/*    #+NAME: MAX_STRING_LENGTH */
/*    : 128 */


const char* qmckl_string_of_error(const qmckl_exit_code error);
void qmckl_string_of_error_f(const qmckl_exit_code error,
                             char result[128]);

/* Updating errors in the context */

/*    The error is updated in the context using ~qmckl_set_error~. */
/*    When the error is set in the context, it is mandatory to specify */
/*    from which function the error is triggered, and a message */
/*    explaining the error. The exit code can't be ~QMCKL_SUCCESS~. */

/*    # Header */
~/TREX/qmckl/include/qmckl.h [unix] [CPP]  [ 31% ]  (85/269,1)
```

```
 1  2  3  Biblio OrgWork Web eMail        us   +6°C   1.1K   565B   LinksysRouter  02%   49°C   /: 21G   100%   0   82%   04/19 10:21
```

# Generated web site


Atomic Orbitals
trex-coe.github.io/qmckl/qmckl_ao.html
Personal OCaml LCPQ QP2 Biblio ToUpdate TREX ERC AiiDA Microsoft Padlets
UP | HOME
Table of Contents
1 2 3 Biblio OrgWork Web eMail

## Atomic Orbitals

The atomic basis set is defined as a list of shells. Each shell $s$ is centered on a nucleus $A$, possesses a given angular momentum $l$ and a radial function $R_s$. The radial function is a linear combination of \emph{primitive} functions that can be of type Slater ($p = 1$) or Gaussian ($p = 2$):

$$R_s(\mathbf{r}) = \mathcal{N}_s |\mathbf{r} - \mathbf{R}_A|^{n_s} \sum_{k=1}^{N_{\text{prim}}} a_{ks} \exp(-\gamma_{ks} |\mathbf{r} - \mathbf{R}_A|^p).$$

In the case of Gaussian functions, $n_s$ is always zero. The normalization factor $\mathcal{N}_s$ ensures that all the functions of the shell are normalized to unity. As this normalization requires the ability to compute overlap integrals, it should be written in the file to ensure that the file is self-contained and does not require the client program to have the ability to compute such integrals.

Atomic orbitals (AOs) are defined as

$$\chi_i(\mathbf{r}) = P_{\eta(i)}(\mathbf{r})\, R_{\theta(i)}(\mathbf{r})$$

where $\theta(i)$ returns the shell on which the AO is expanded, and $\eta(i)$ denotes which angular function is chosen.

In this section we describe the kernels used to compute the values, gradients and Laplacian of the atomic basis functions.

### 1 Polynomial part

#### 1.1 Powers of $x - X_i$

The `qmckl_ao_power` function computes all the powers of the `n` input data up to the given maximum value given in input for each of the $n$ points:

$$\nabla_z v_i = -2a_i(X_z - R_z)v_i$$

$$\Delta v_i = a_i(4|X - R|^2 a_i - 6)v_i$$

| | | |
|---|---|---|
| `context` | input | Global state |
| `X(3)` | input | Array containing the coordinates of the points |
| `R(3)` | input | Array containing the x,y,z coordinates of the center |
| `n` | input | Number of computed Gaussians |
| `A(n)` | input | Exponents of the Gaussians |
| `VGL(ldv,5)` | output | Value, gradients and Laplacian of the Gaussians |
| `ldv` | input | Leading dimension of array `VGL` |

Requirements :

- `context` is not 0
- `n` > 0
- `ldv` >= 5
- `A(i)` > 0 for all `i`
- `X` is allocated with at least $3 \times 8$ bytes
- `R` is allocated with at least $3 \times 8$ bytes
- `A` is allocated with at least $n \times 8$ bytes
- `VGL` is allocated with at least $n \times 5 \times 8$ bytes

```
qmckl_exit_code
qmckl_ao_gaussian_vgl(const qmckl_context context,
                      const double *X,
                      const double *R,
                      const int64_t *n,
                      const int64_t *A,
                      const double *VGL,
                      const int64_t ldv);
```

## Identified kernels

At each QMC step, we need to evaluate $E_{\text{loc}}(\mathrm{r}_1, \dots, \mathrm{r}_N) = \frac{\hat{H}\Psi(\mathrm{r}_1,\dots,\mathrm{r}_N)}{\Psi(\mathrm{r}_1,\dots,\mathrm{r}_N)}$:

- $\Psi(\mathrm{r}_1, \dots, \mathrm{r}_N)$
- $\Delta_i \Psi(\mathrm{r}_1, \dots, \mathrm{r}_i, \dots, \mathrm{r}_N)$: kinetic energy
- $\vec{\nabla}_i \Psi(\mathrm{r}_1, \dots, \mathrm{r}_i, \dots, \mathrm{r}_N)$: drift in the stochastic process

### Main kernels

- AOs: $\chi(\mathrm{r}), \vec{\nabla}\chi(\mathrm{r}), \Delta\chi(\mathrm{r})$
- MOs: $\phi(\mathrm{r}), \vec{\nabla}\phi(\mathrm{r}), \Delta\phi(\mathrm{r})$
- Slater determinants (value, gradient, Laplacian)
- Pseudo-potential
- Jastrow correlation factor (eN, ee, eeN)

## Codesign strategy

1. Kernel extraction: QMC experts agree on the mathematical expression of the problem
2. A mini-application is written to find the best data layout with HPC experts from real-size examples
3. The kernel is written in the documentation library
4. HPC experts provide an HPC version of the kernel with the same API
5. The library is linked in the QMC codes of the CoE

## Our first application : 3-body Jastrow factor

$$J_{\text{een}}(\mathrm{r},\mathrm{R}) = \sum_{\alpha=1}^{N_{\text{nucl}}} \sum_{i=1}^{N_{\text{elec}}} \sum_{j=1}^{i-1} \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} c_{lkp\alpha} \left(r_{ij}\right)^k \left[\left(R_{i\alpha}\right)^l + \left(R_{j\alpha}\right)^l\right] \left(R_{i\,\alpha}\, R_{j\alpha}\right)^{(p-k-l)/2}$$

can be rewritten as

$$J_{\text{een}}(\mathrm{r},\mathrm{R}) = \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} \sum_{\alpha=1}^{N_{\text{nucl}}} c_{lkp\alpha} \sum_{i=1}^{N_{\text{elec}}} \bar{\mathtt{R}}_{i,\alpha,(p-k-l)/2}\, \bar{\mathtt{P}}_{i,\alpha,k,(p-k+l)/2} \quad (\downarrow \text{complexity})$$

with

$$\bar{\mathtt{P}}_{i,\alpha,k,l} = \sum_{j=1}^{N_{\text{elec}}} \bar{\mathtt{r}}_{i,j,k}\, \bar{\mathtt{R}}_{j,\alpha,l}. \quad \text{(GEMM)}$$

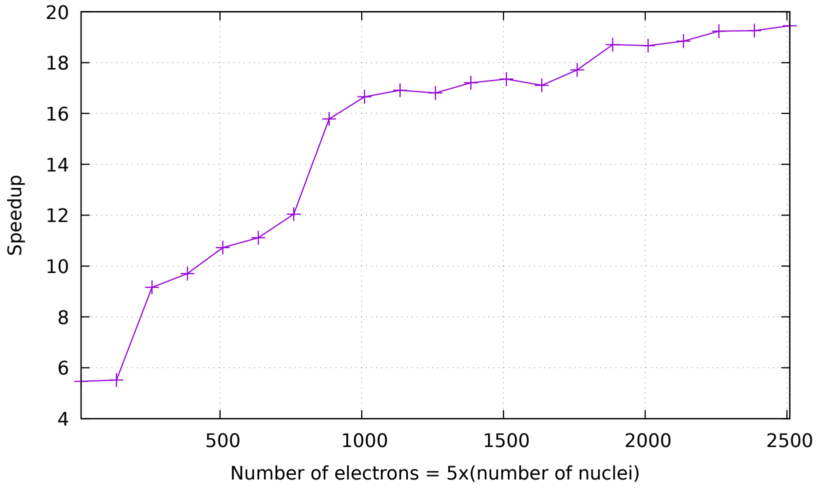
## Our first application : Gradient and Laplacian

$$\begin{aligned}\nabla_{im} J_{\text{een}}(\mathrm{r},\mathrm{R}) = & \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} \sum_{\alpha=1}^{N_{\text{nucl}}} c_{lkp\alpha} \sum_{i=1}^{N_{\text{elec}}} \bar{\mathtt{G}}_{i,m,\alpha,(p-k-l)/2}\, \bar{\mathtt{P}}_{i,\alpha,k,(p-k+l)/2} + \\ & \bar{\mathtt{G}}_{i,m,\alpha,(p-k+l)/2}\, \bar{\mathtt{P}}_{i,\alpha,k,(p-k-l)/2} + \bar{\mathtt{R}}_{i,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,m,\alpha,k,(p-k+l)/2} + \\ & \bar{\mathtt{R}}_{i,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,m,\alpha,k,(p-k-l)/2} + \delta_{m,4} \Big( \\ & \bar{\mathtt{G}}_{i,1,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,1,\alpha,k,(p-k-l)/2} + \bar{\mathtt{G}}_{i,2,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,2,\alpha,k,(p-k-l)/2} + \\ & \bar{\mathtt{G}}_{i,3,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,3,\alpha,k,(p-k-l)/2} + \bar{\mathtt{G}}_{i,1,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,1,\alpha,k,(p-k+l)/2} + \\ & \bar{\mathtt{G}}_{i,2,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,2,\alpha,k,(p-k+l)/2} + \bar{\mathtt{G}}_{i,3,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,3,\alpha,k,(p-k+l)/2} \Big)\end{aligned}$$

with

$$\bar{\mathtt{G}}_{i,m,\alpha,l} = \frac{\partial \left(R_{i\alpha}\right)^l}{\partial r_i}, \qquad \bar{\mathtt{g}}_{i,m,j,k} = \frac{\partial \left(r_{ij}\right)^k}{\partial r_i}, \qquad \text{and} \;\; \bar{\mathtt{Q}}_{i,m,\alpha,k,l} = \sum_{j=1}^{N_{\text{elec}}} \bar{\mathtt{g}}_{i,m,j,k}\, \bar{\mathtt{R}}_{j,\alpha,l}$$

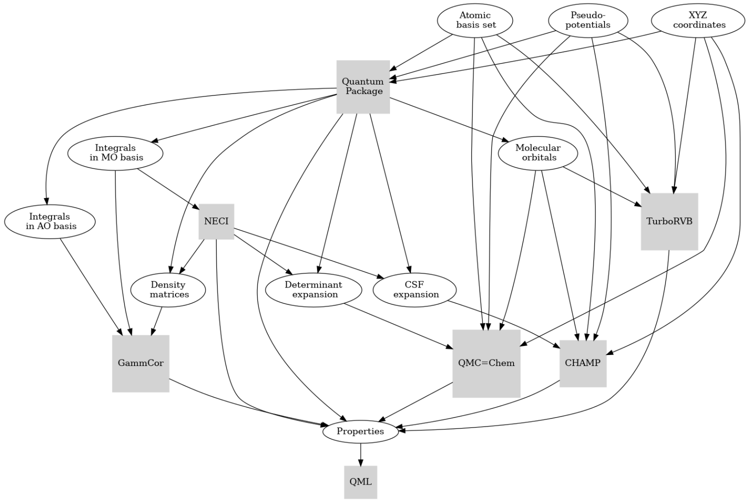
## Speed up



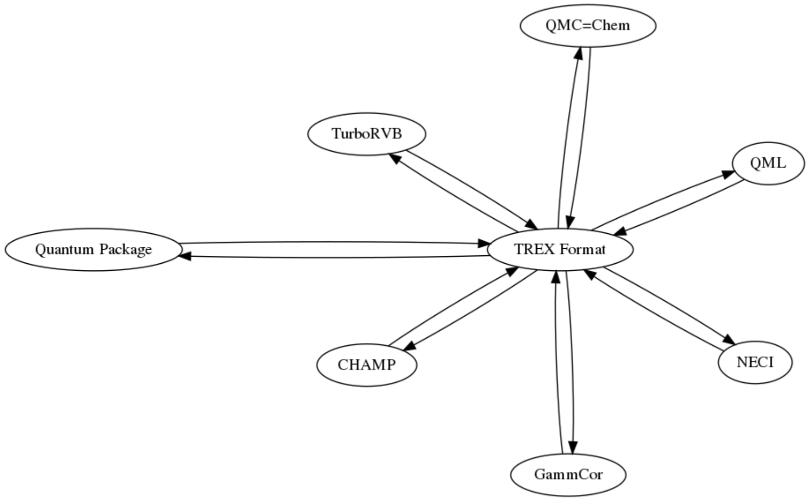
$\sim$ 80% of the AVX-512 peak is reached on a Skylake CPU.

## TREXIO: The TREX I/O library

# Current situation



## Goal

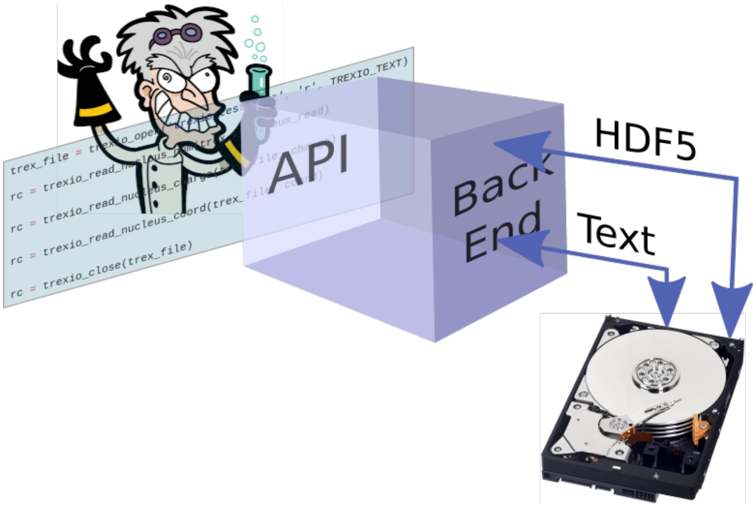

QMC=Chem
TurboRVB
QML
Quantum Package
TREX Format
CHAMP
NECI
GammCor

**Front-end**

- Definition of an API for to read/write wave functions
- C-compatible API: Easy bindings in other languages

**Back-end**

- HDF5: Efficient I/O
- Text: debugging, fallback when HDF5 can't be installed



```
trex_file = trexio_open('trexio_test[illegible]', 'r', TREXIO_TEXT)
rc = trexio_read_nucleus_num(trex_file, [illegible]num_read)
rc = trexio_read_nucleus_charge(trex_file, charge)
rc = trexio_read_nucleus_coord(trex_file, coord)
rc = trexio_close(trex_file)
rc =
```

## Front end

### Groups

| | | | | |
|---|---|---|---|---|
| Electron | AO | Basis | ECP | OneRDM |
| Nucleus | MO | Determinants | Jastrow | TwoRDM |

### Data

- Inside each group, multiple values.
- Strong conventions (atomic units, ordering of cartesian orbitals, *etc*)
- File is self-contained: no external knowledge is necessary to compute $\Psi(r_1, \ldots, r_n)$ (normalization factors, basis set parameters, *etc*)

## API

- Computable function names:
  `trexio_<read|write|has>_<group>_<data>[_32|_64]`
- return code for error handling
- Auto-generated from a JSON config file defining groups, data and types

```
"electron": {
    "up_num"                 : [ "int", [] ]
  , "dn_num"                 : [ "int", [] ]
},

"nucleus": {
    "num"                    : [ "int"  , [                        ] ]
  , "charge"                 : [ "float", [ "nucleus.num"          ] ]
  , "coord"                  : [ "float", [ "nucleus.num", "3"     ] ]
  , "label"                  : [ "char" , [ "nucleus.num", "32"    ] ]
  , "point_group"            : [ "char" , [ "32"                   ] ]
},
```

## Example

```fortran
subroutine read_xyz(trex_file, xyz_filename)
  use trexio
  implicit none
  integer*8, intent(in)         :: trex_file
  character*(128), intent(in)   :: xyz_filename
  integer*8                     :: nucl_num          ! Number of nuclei
  character*(256)               :: title             ! Title of the file
  character*(32), allocatable   :: nucl_label(:)     ! Atom labels
  real*8, allocatable           :: nucl_charge(:)    ! Nuclear charges
  real*8, allocatable           :: nucl_coord(:,:)   ! Nuclear coordinates
  integer*8                     :: i
  integer                       :: j
  integer                       :: info
  double precision, parameter   :: a0 = 0.52917721067d0

  open(unit=10,file=xyz_filename)

  read(10,*) nucl_num

  allocate(nucl_label(nucl_num),  &
           nucl_charge(nucl_num), &
           nucl_coord(3,nucl_num) )

  read(10,'(A)') title

  do i=1,nucl_num
     read(10,*) nucl_label(i), nucl_coord(1:3,i)

     info = trexio_element_number_of_symbol(trim(nucl_label(i)), j)
     call check_success(info, 'Unable to convert symbol to number')

     nucl_charge(i) = dble(j)
  end do

  close(10)

  ! Convert into atomic units
  nucl_coord = nucl_coord / a0

  info = trexio_write_nucleus_num(trex_file,nucl_num)
  call check_success(info, 'Unable to write number of nuclei')

  info = trexio_write_nucleus_coord(trex_file,nucl_coord)
  call check_success(info, 'Unable to write nuclear coordinates')

  info = trexio_write_nucleus_charge(trex_file,nucl_charge)
  call check_success(info, 'Unable to write nuclear charges')

  info = trexio_write_nucleus_label(trex_file,nucl_label)
  call check_success(info, 'Unable to write nuclear labels')

  beta_num  = int(sum(nucl_charge(:)))/2
  alpha_num = int(sum(nucl_charge(:))) - beta_num

  info = trexio_write_electron_up_num(trex_file,alpha_num)
  call check_success(info, 'Unable to write up electrons')

  info = trexio_write_electron_dn_num(trex_file,beta_num)
  call check_success(info, 'Unable to write dn electrons')

end subroutine read_xyz
```

## Links

- TREX web site : `https://trex-coe.eu`
- QMCkl documentation : `https://trex-coe.github.io/qmckl`
- QMCkl repository : `https://github.com/trex-coe/qmckl`
- TREXIO repository : `https://github.com/trex-coe/trexio`