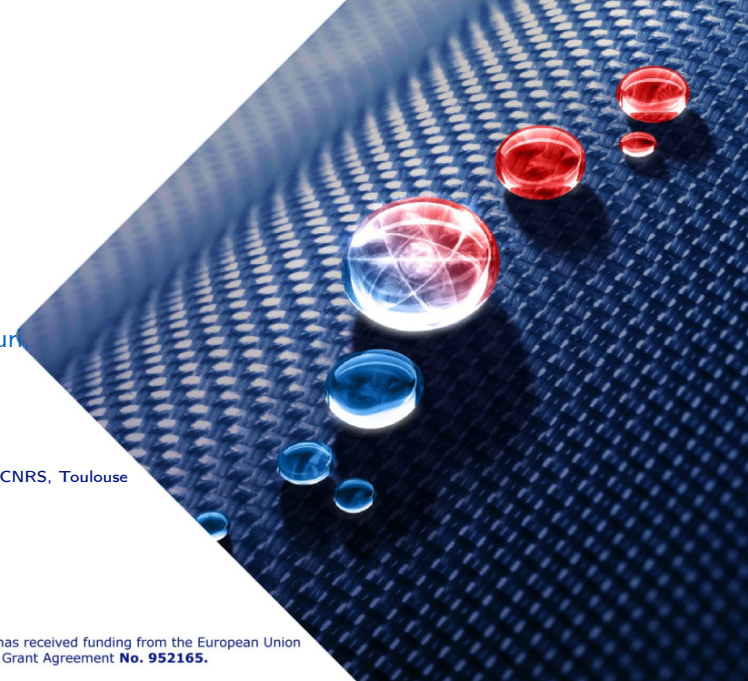
Targeting Real chemical accuracy at the EXascale

# QMCkl

Anthony Scemama, Vijay Gopal Chilkuri, William Jalby

1/03/2022

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.**

- Application Programming Interface (API) for main algorithms of QMC
- Pedagogical (WP1) *and* high-performance (WP3) implementations
- API should have enough abstraction to let HPC implementations handle internal data structures, memory allocations, numerical precision, GPUs ...
- API is expressed in C for maximum compatibility with codes
- Low level functions: linear algebra, small kernels (a.U.v)
- High-level functions: domain-specific
- In HPC implementation, everything is allowed as long as the HPC function returns the same value as the pedagogical one within the specified numerical precision.

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

Literate programming with *org-mode*:

- Comments are more important than code
- Can add graphics, LaTeXformulas, 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 report was auto-generated from the documentation

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

File Edit Options Buffers Tools Table Org Text Help

Save Undo

#+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 $R_l$. The radial function is a linear combination of \emph{primitive} functions that can be of type Slater ($p=1$) or Gaussian ($p=2$):

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

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** :noexport:
- **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:

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~                  |
```

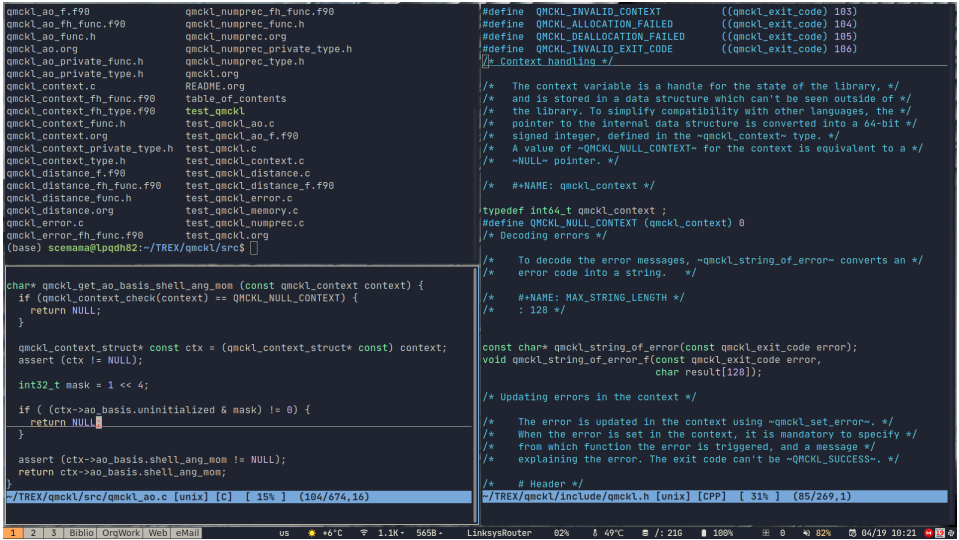
Requirements :

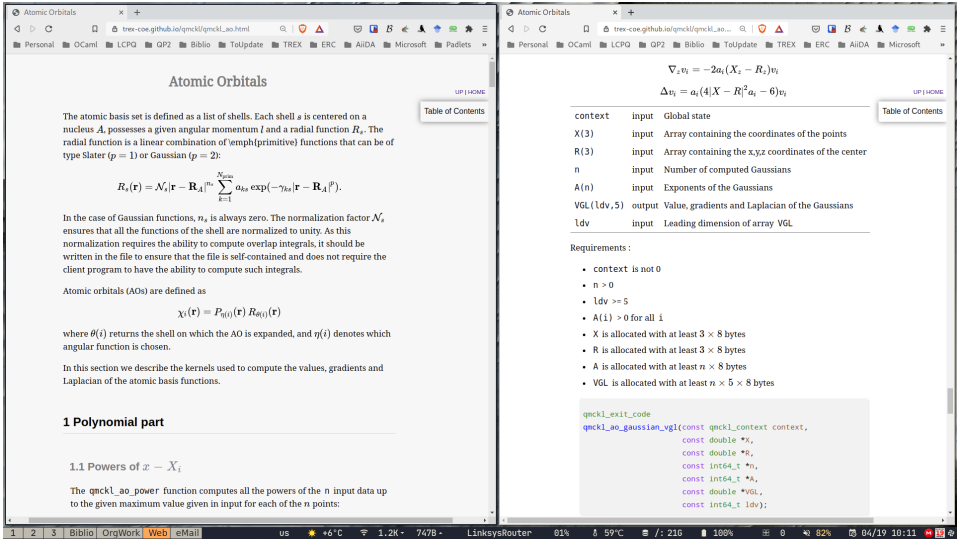
- ~context~ is not 0
- ~n~ > 0
- ~ldv~ >= 5
- ~A(i)~ > 0 for all ~i~
- ~X~ is allocated with at least $3 \times 8$ bytes
- ~R~ is allocated with at least $3 \times 8$ bytes
- ~A~ is allocated with at least $n \times 8$ bytes
- ~VGL~ is allocated with at least $n \times 5 \times 8$ bytes

```
#+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





Atomic Orbitals

trex-coe.github.io/qmckl/qmckl_ao.html

Personal OCaml LCPQ QP2 Biblio ToUpdate TREX ERC AiiDA Microsoft Padlets

# Atomic Orbitals

UP | HOME

Table of Contents

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 ($p = 2$):

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

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:

Atomic Orbitals

trex-coe.github.io/qmckl/qmckl_ao...

Personal OCaml LCPQ QP2 Biblio ToUpdate TREX ERC AiiDA Microsoft

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

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

UP | HOME

Table of Contents

| | | |
|---|---|---|
| `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` |

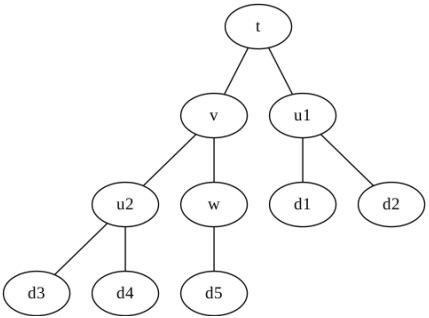
Requirements :

- `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);
```

1 2 3 Biblio OrgWork Web eMail us +6°C 1.2K- 747B- LinksysRouter 81% 59°C /: 21G 100% 0 82% 04/19 10:11

The quantities of interest are organized in a dependency graph:




- The user asks for $v$. $w$ and $u_2$ were computed.
- If the user asks for $t$, $v$ is re-used and only $u_1$ is computed.
- The sub-trees are independent, so they can be computed in parallel

- Each quantity is expressed as a function of other quantities:

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

  Here, $\chi$ depends on $P$ and $\mathcal{R}$.
- The context is a data structure containing the state of the library (equivalent to global variables).
- It contains all the computed quantities that may be re-used
- It is passed to all the functions
- The date is an integer which increments each time the electron coordinates change.
- The quantities are recomputed only if their date is older than the current date, like in a `Makefile`

```
qmckl_exit_code provide_chi(qmckl_context* ctx) {
  qmckl_exit_code rc;

  if (ctx->date > ctx->date_chi) {

    rc = provide_p(ctx);   // asserts that ctx->p is computed
    if (rc != QMCKL_SUCCESS) return rc;

    rc = provide_r(ctx);   // asserts that ctx->r is computed
    if (rc != QMCKL_SUCCESS) return rc;

    ctx->chi = compute_chi(ctx->p, ctx->r);
    ctx->chi_date = ctx->date;
  }
  return QMCKL_SUCCESS;
}
```

# What happens when a quantity is asked for

```
qmckl_exit_code qmckl_get_chi(qmckl_context ctx, double* chi, int64_t size_max) {
  // First, check context is valid, and that size_max is large enough. Then,

  qmckl_exit_code rc = provide_chi(ctx);
  if (rc != QMCKL_SUCCESS) return rc;

  memcpy(chi, ctx->chi, ctx->chi_size);
  return QMCKL_SUCCESS;
}
```

Note: the `memcpy` can be avoided using `qmckl_get_chi_inplace`, to be used with caution.

All QMCkl functions return an error code. A convenient way to handle errors is to write an error-checking function that displays the error in text format and exits the program.

```
subroutine qmckl_check_error(rc, message)
  use qmckl
  implicit none
  integer(qmckl_exit_code), intent(in) :: rc
  character(len=*)        , intent(in) :: message
  character(len=128)                   :: str_buffer

  if (rc /= QMCKL_SUCCESS) then
     print *, message
     call qmckl_string_of_error(rc, str_buffer)
     print *, str_buffer
     call exit(rc)
  end if
end subroutine qmckl_check_error
```

- The user gives input parameters to the library to initialize the context
- The arrays can be given one by one, but the easy way is to read a TREXIO file:

```
use qmckl
integer(qmckl_context)   :: qmckl_ctx
integer(qmckl_exit_code) :: rc
double precision         :: qmckl_ao_vgl(ao_num,5,elec_num, walk_num)

qmckl_ctx = qmckl_context_create()
rc = qmckl_trexio_read(qmckl_ctx, trexio_filename, len(trim(trexio_filename)))
call qmckl_check_error(rc, 'Read TREXIO')

rc = qmckl_set_electron_walk_num(qmckl_ctx, walk_num)
call qmckl_check_error(rc, 'Set walk_num'))

rc = qmckl_set_electron_coord(qmckl_ctx, 'N', elec_coord, size(elec_coord)) ! Increments date
call qmckl_check_error(rc, 'Set elec_coord'))

rc = qmckl_get_ao_basis_ao_vgl(qmckl_ctx, qmckl_ao_vgl, size(qmckl_ao_vgl)) ! Commputes AOs
call qmckl_check_error(rc, 'Get ao_vgl'))
```

# Example: Computing an atomic orbital on a grid

```
$ ao_grid <trexio_file> <AO_id> <point_num>
```

Start by fetching the command-line arguments:

```
  if (iargc() /= 3) then
     print *, 'Syntax: ao_grid <trexio_file> <AO_id> <point_num>'
     call exit(-1)
  end if
  call getarg(1, trexio_filename)
  call getarg(2, str_buffer)
  read(str_buffer, *) ao_id
  call getarg(3, str_buffer)
  read(str_buffer, *) point_num_x

  if (point_num_x < 0 .or. point_num_x > 300) then
     print *, 'Error: 0 < point_num < 300'
     call exit(-1)
  end if
```

Create the QMCkl context and initialize it with the wave function present in the TREXIO file:

```
qmckl_ctx = qmckl_context_create()
rc  = qmckl_trexio_read(qmckl_ctx, trexio_filename, 1_8*len(trim(trexio_filename)))
call qmckl_check_error(rc, 'Read TREXIO')
```

We need to check that `ao_id` is in the range, so we get the total number of AOs from QMCkl:

```
rc = qmckl_get_ao_basis_ao_num(qmckl_ctx, ao_num)
call qmckl_check_error(rc, 'Getting ao_num')

if (ao_id < 0 .or. ao_id > ao_num) then
   print *, 'Error: 0 < ao_id < ', ao_num
   call exit(-1)
end if
```

# Example: Computing an atomic orbital on a grid

Compute the limits of the box. For that, we first need to ask QMCkl the coordinates of nuclei.

```
rc = qmckl_get_nucleus_num(qmckl_ctx, nucl_num)
call qmckl_check_error(rc, 'Get nucleus num')

allocate( nucl_coord(3, nucl_num) )
rc = qmckl_get_nucleus_coord(qmckl_ctx, 'N', nucl_coord, 3_8*nucl_num)
call qmckl_check_error(rc, 'Get nucleus coord')
```

We now compute the coordinates of opposite points of the box:

```
rmin(1) = minval( nucl_coord(1,:) ) - 5.d0
rmin(2) = minval( nucl_coord(2,:) ) - 5.d0
rmin(3) = minval( nucl_coord(3,:) ) - 5.d0

rmax(1) = maxval( nucl_coord(1,:) ) + 5.d0
rmax(2) = maxval( nucl_coord(2,:) ) + 5.d0
rmax(3) = maxval( nucl_coord(3,:) ) + 5.d0

dr(1:3) = (rmax(1:3) - rmin(1:3)) / dble(point_num_x-1)
```

# Example: Computing an atomic orbital on a grid

We produce the list of point coordinates where the AO will be evaluated:

```
point_num = point_num_x**3
allocate( points(point_num, 3) )
ipoint=0
z = rmin(3)
do k=1,point_num_x
   y = rmin(2)
   do j=1,point_num_x
      x = rmin(1)
      do i=1,point_num_x
         ipoint = ipoint+1
         points(ipoint,1) = x
         points(ipoint,2) = y
         points(ipoint,3) = z
         x = x + dr(1)
      end do
      y = y + dr(2)
   end do
   z = z + dr(3)
end do
```

# Example: Computing an atomic orbital on a grid

We give the points to QMCkl:

```
rc = qmckl_set_point(qmckl_ctx, 'T', points, point_num)
call qmckl_check_error(rc, 'Setting points')
```

We allocate the space required to retrieve the values, gradients and Laplacian of all AOs, and ask the data to QMCkl:

```
allocate( ao_vgl(ao_num, 5, point_num) )
rc = qmckl_get_ao_basis_ao_vgl(qmckl_ctx, ao_vgl, ao_num*5_8*point_num)
call qmckl_check_error(rc, 'Setting points')
```

We finally print the value of the AO:

```
do ipoint=1, point_num
   print '(3(F16.10,X),E20.10)', points(ipoint, 1:3), ao_vgl(ao_id,1,ipoint)
end do
```

Loop-based: Splits loops in $m$ chunks and distributes chunks in different threads.

```
!$OMP PARALLEL DO
do i=1,N
...
end do
!$OMP END PARALLEL DO
```

- Advantages
  - Very low scheduling overhead
  - Control of memory locality
  - Easy to write and to think about
- Difficulties
  - Not optimal for inhomogeneous workloads
  - Limited to the scope of the loop

Task based Splits work to do into independent tasks, adds them to a queue and lets the threads perform these tasks

```
void provide_chi(qmckl_context* ctx) {
  if (ctx->date > ctx->date_chi) {
    #pragma omp task
    provide_p(ctx);
    #pragma omp task
    provide_r(ctx);
    #pragma omp taskwait
    ctx->chi = compute_chi(ctx->p, ctx->r);
    ctx->chi_date = ctx->date;
  }
}
```

- Advantages
  - Better load balancing
  - Tasks can be distributed on CPUs *and* GPUs
  - Dependencies between tasks can be expressed
- Difficulties
  - Scheduling tasks requires some CPU power $\Longrightarrow$ tasks need enough work to do

Usual storage: `double precision :: A(8,8)`

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 9 | 17 | 25 | 33 | 41 | 49 | 57 |
| 2 | 10 | 18 | 26 | 34 | 42 | 50 | 58 |
| 3 | 11 | 19 | 27 | 35 | 43 | 51 | 59 |
| 4 | 12 | 20 | 28 | 36 | 44 | 52 | 60 |
| 5 | 13 | 21 | 29 | 37 | 45 | 53 | 61 |
| 6 | 14 | 22 | 30 | 38 | 46 | 54 | 62 |
| 7 | 15 | 23 | 31 | 39 | 47 | 55 | 63 |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 |

Tiled storage: `double precision :: A(4,4,4,4)`

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 9 | 13 | 33 | 37 | 41 | 45 |
| 2 | 6 | 10 | 14 | 34 | 38 | 42 | 46 |
| 3 | 7 | 11 | 15 | 35 | 39 | 43 | 47 |
| 4 | 8 | 12 | 16 | 36 | 40 | 44 | 48 |
| 17 | 21 | 25 | 29 | 49 | 53 | 57 | 61 |
| 18 | 22 | 26 | 30 | 50 | 54 | 58 | 62 |
| 19 | 13 | 27 | 31 | 51 | 55 | 59 | 63 |
| 20 | 24 | 28 | 32 | 52 | 56 | 60 | 64 |

## Advantages

- Much Better locality in memory: sizes calibrated to cache sizes
- Performance of matrix multiplication is constant
- Fast transposition
- Blocks can be directly sent to GPUs: no problem with leading dimension
- Computation time of tasks is easy to estimate

## Disadvantages

- Code more difficult to read/write: more nested loops
- Mapping between math and code is not simple
- Random access is very inconvenient
- Not naturally adapted to external libraries and codes

- In QMC, we manipulate small matrices
- Compilers with -O3 usually generate efficient code for large data sets
- Optimized BLAS libraries work with 2D arrays, which is not necessarily the most efficient pattern
- Matrices in QMCkl will be internally tiled: needs efficient linear algebra for small (sub-)matrices
- We implement low-level functions using x86 or ARM assembly:
  - Performance is independent of the compiler
  - Implementation is specific to tiled arrays
- One generic kernel produces multiple assembly versions with a code generator

## Pedagogical only

- MOs
- Potential (ee, eN, NN)
- Inverse Slater matrix

## Optimized high-level

- AOs
- Jastrow
- Sherman Morrison
- Adjugate : $\mathrm{B} = \mathrm{adj(A)} = \det(\mathrm{A})\,\mathrm{A}^{-1}$

## Optimized low-level functions for tildes matrices

- Matrix multiplication: heavily used in Jastrow
- Rank-$k$ update: necessary for Sherman-Morrison
- Matrix-vector multiplication:
- $u^{\dagger}.A.v$: necessary for $\Psi = \sum_{ij} C_{ij} D_i^{\uparrow} D_j^{\downarrow}$

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

can be rewritten as

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

with

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

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

with

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

$$\sum_{i \neq j} r_{ij}^{k} \left[ R_{i\alpha}^{l} + R_{j\alpha}^{l} \right] R_{i\alpha}^{lm} R_{j\alpha}^{m}$$

$$= \frac{1}{2} \sum_{i} \sum_{j} r_{ij}^{k} \left( R_{i\alpha}^{lm} R_{j\alpha}^{m} + R_{i\alpha}^{m} R_{j\alpha}^{lm} \right)$$

$$= \frac{1}{2} \sum_{i} \sum_{j} \left[ R_{i\alpha}^{lm} r_{ij}^{k} R_{j\alpha}^{m} + R_{i\alpha}^{m} r_{ij}^{k} R_{j\alpha}^{lm} \right]$$

$$= \frac{1}{2} \left[ \sum_{ij} R_{i\alpha}^{lm} r_{ij}^{k} R_{j\alpha}^{m} + \sum_{ij} R_{i\alpha}^{m} r_{ij}^{k} R_{j\alpha}^{lm} \right]$$

$$= \frac{1}{2} \left[ \sum_{ij} R_{i\alpha}^{lm} r_{ij}^{k} R_{j\alpha}^{m} + \sum_{ji} R_{j\alpha}^{m} r_{ji}^{k} R_{i\alpha}^{lm} \right]$$

$$= \sum_{i} \sum_{j} R_{i\alpha}^{lm} r_{ij}^{k} R_{j\alpha}^{m} = \sum_{i} R_{i\alpha}^{lm} P_{i\alpha}^{km}$$

with $P_{i\alpha}^{km} = \sum_{j} r_{ij}^{k} R_{j\alpha}^{m}$

$P_{ik,\alpha m} = \sum_{j} r_{ik,ji} R_{j,\alpha m}$ $\quad O\left(N_{ex}^{2} N_{\alpha} \times n^{i}\right)$

$J = \sum_{i\alpha} R_{lm,i\alpha} P_{i\alpha,km}$ $\quad O\left(n^{3} \times N_{ex} \times N_{\alpha}\right)$

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

$$\chi_i(\mathrm{r}) = \mathcal{M}_i\, P_{\eta(i)}(\mathrm{r})\, R_{\theta(i)}(\mathrm{r})$$

## $R_{\theta(i)}(\mathrm{r})$ : Radial part

- For each nucleus, beyond a given e-N distance all exponentials are zero
- We call the exp function only if the argument is small enough: $\gamma_{ks} |\mathrm{r} - \mathrm{R}_A|^2 \leq -\log(10^{-12})$
- The same radial part is reused for multiple AOs ($p_x, p_y, p_z$)

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

$$\chi_i(\mathrm{r}) = \mathcal{M}_i\, P_{\eta(i)}(\mathrm{r})\, R_{\theta(i)}(\mathrm{r})$$

**$P_{\eta(i)}(\mathrm{r})$ : Polynomial part**

- For each nucleus, we know the max angular momentum $l_{\text{max}}$
- We compute all the powers of $x, y, z$ up to $l_{\text{max}}$ by successive multiplications, and their 1st and 2nd derivatives
- These values are re-used for all AOs

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

$$\chi_i(\mathrm{r}) = \mathcal{M}_i\, P_{\eta(i)}(\mathrm{r})\, R_{\theta(i)}(\mathrm{r})$$

## Combining $R$ and $P$

- Very low arithmetic intensity
- $P$ and $R$ are computed together for each atom ($P$) and each shell ($R$) to reduce memory operations

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

## Work in progress

- Sort arrays of exponents $\gamma_{ks}$ in increasing order per atom. Identify duplicates that can occur between shells. We know that after indice $k_{\max}$ all exponents are zero, so we can vectorize the computation of all exponentials centered on $A$.
- Express the coefficients in the *generally contracted* format (as in MOLCAS):

$$\gamma = \begin{bmatrix} 0.158 \\ 0.502 \\ 1.792 \\ 7.903 \\ 52.56 \end{bmatrix} \quad e_k = e^{\gamma_k |r - R_A|^2} \quad \mathrm{A} = \begin{bmatrix} 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \\ 0.852933 & 0.0 & 0.0 \\ 0.189684 & 0.0 & 0.0 \\ 0.025374 & 0.0 & 0.0 \end{bmatrix} \quad \mathrm{R} = \mathrm{A}^\dagger e$$

- Introduce low-level functions
- Introduce tiled arrays
- AOs into sparse format
- AOs in spherical coordinates
- Sparse AO $\rightarrow$ MO transformation
- Python interface for prototyping
- Multi-determinant wave functions

## Pseudopotentials

- Should we pre-compute $\langle Y_{lm}|\phi(r)\rangle$ of the determinantal component and store it in TREXIO?
- I have no experience in programming pseudos with quadratures: I don't know how to do it efficiently

## Periodic systems

- I have no experience with periodic systems. What changes between isolated system and periodic?

## Possible strategy

- Write a latex file with all formulas and detailed explanations
- It is the most difficult and time-consuming part of writing code in QMCkl. Writing code is faster than writing documentation.