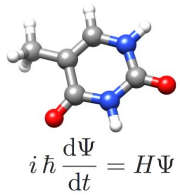
Designing a modern C++/Fortran Interface by Example

Maximilien A. Ambroise

July 3, 2020

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386



COSINE
COMPUTATIONAL SPECTROSCOPY IN NATURAL SCIENCES AND ENGINEERING

Background

Education

- B.Sc. in Nanoscience (University of Basel, Switzerland)
- M.Sc. in Chemistry (University of Aarhus, Denmark)
- Since 2018: PhD Candidate in Theoretical Chemistry (University of Heidelberg, Germany)

COSINE

- Computational Spectroscopy in Natural Sciences and Engineering
- A Marie Skłodowska-Curie Innovative Training Network, part of Horizon 2020
- Devising novel theoretical tools and computational codes rooted in Electronic Structure Theory for the investigation of organic photochemistry

Sparse Matrices

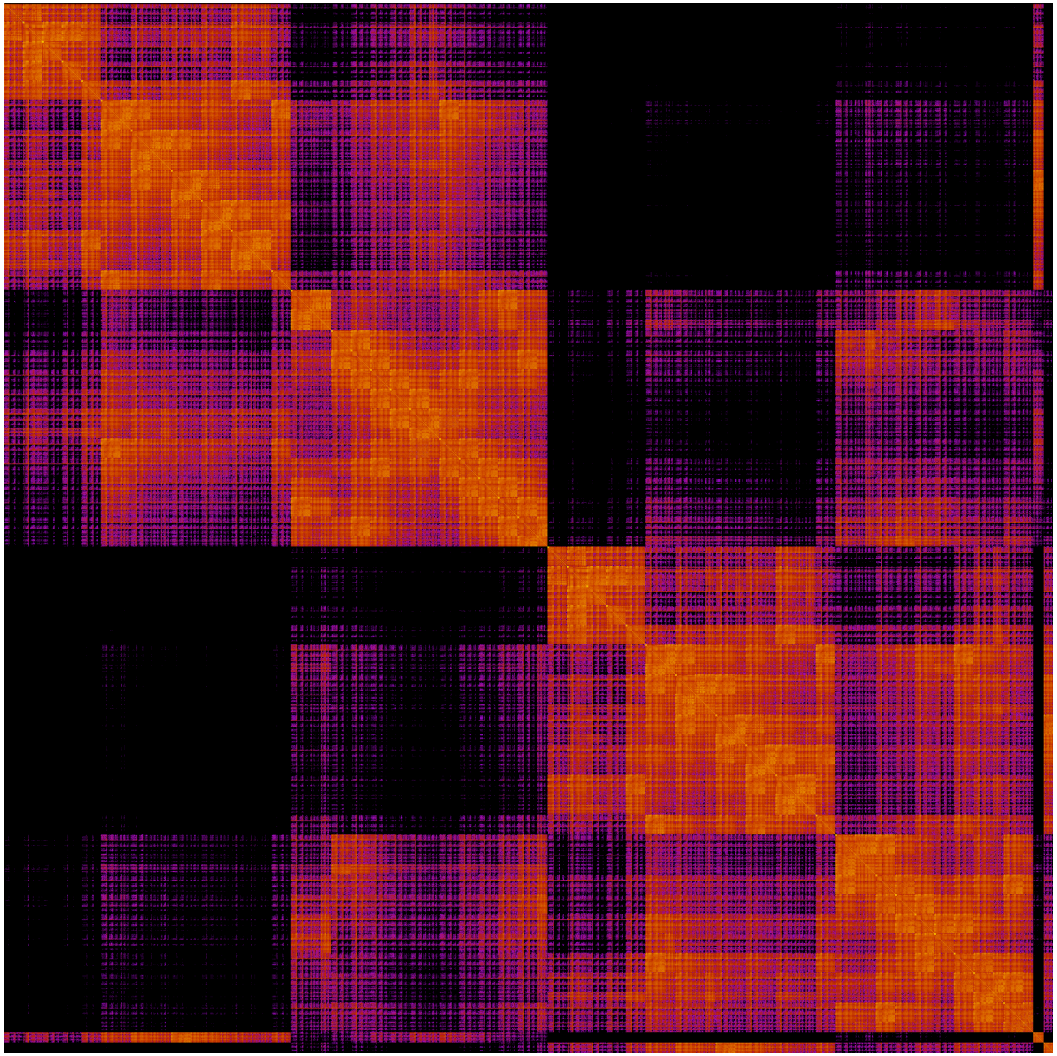
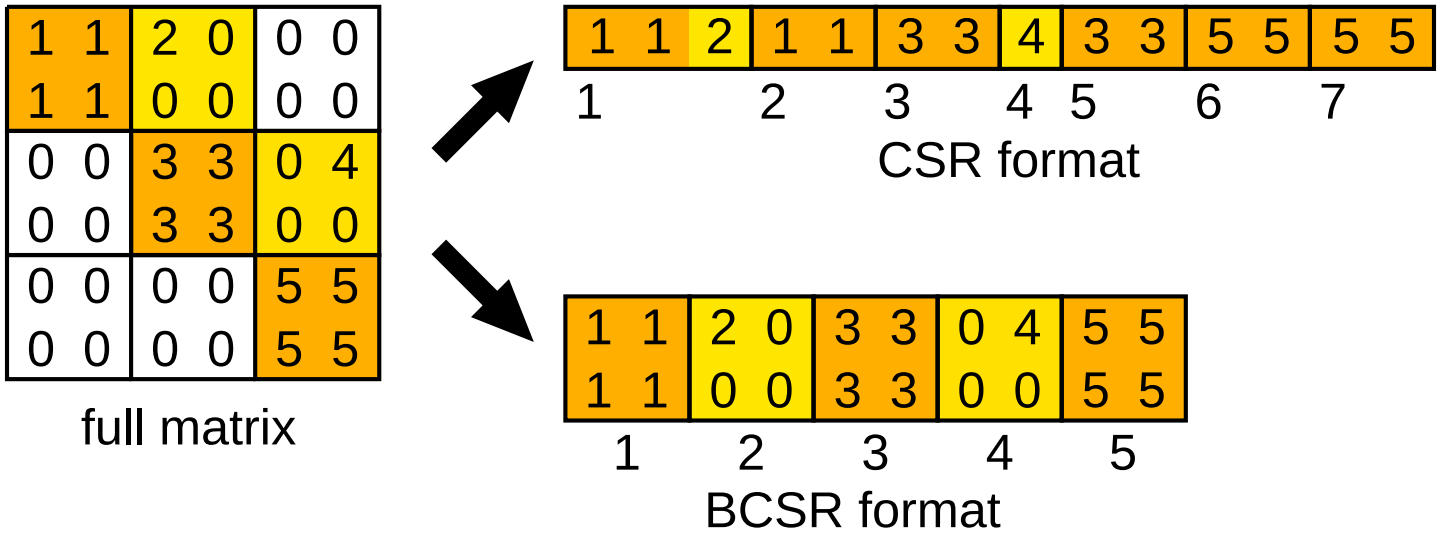
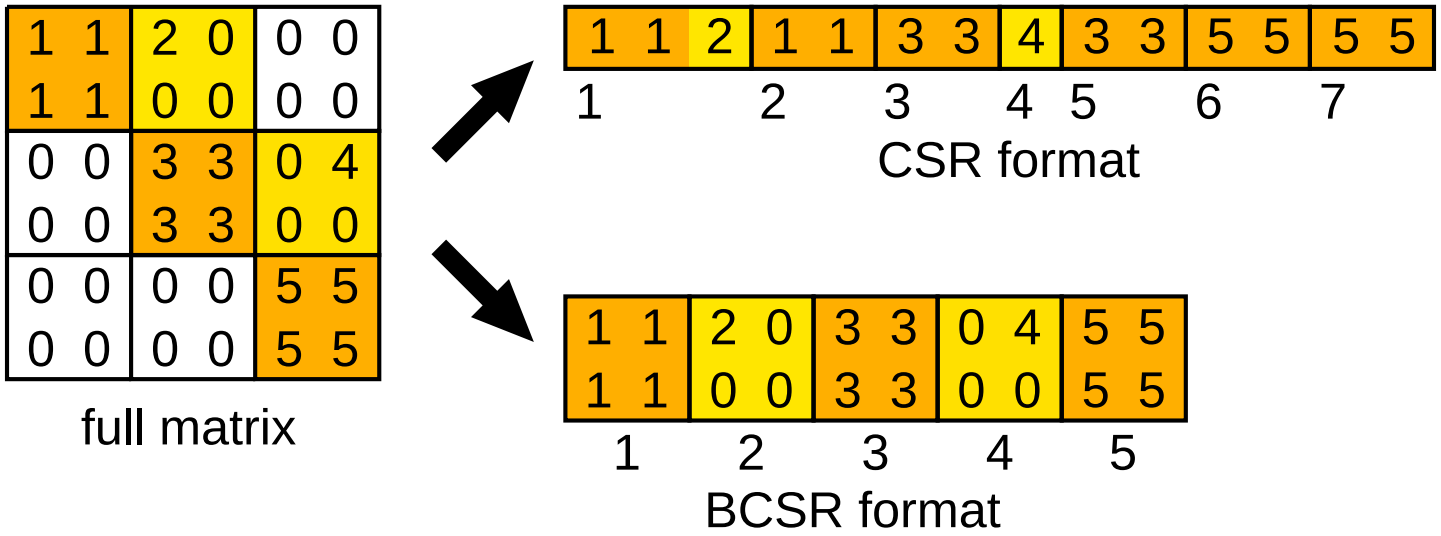


Figure: Fock matrix for the phosphonucleoside pair Adenine-Guanine (HF/pc-3, N_{bas}=3440)

Sparse Matrix Formats

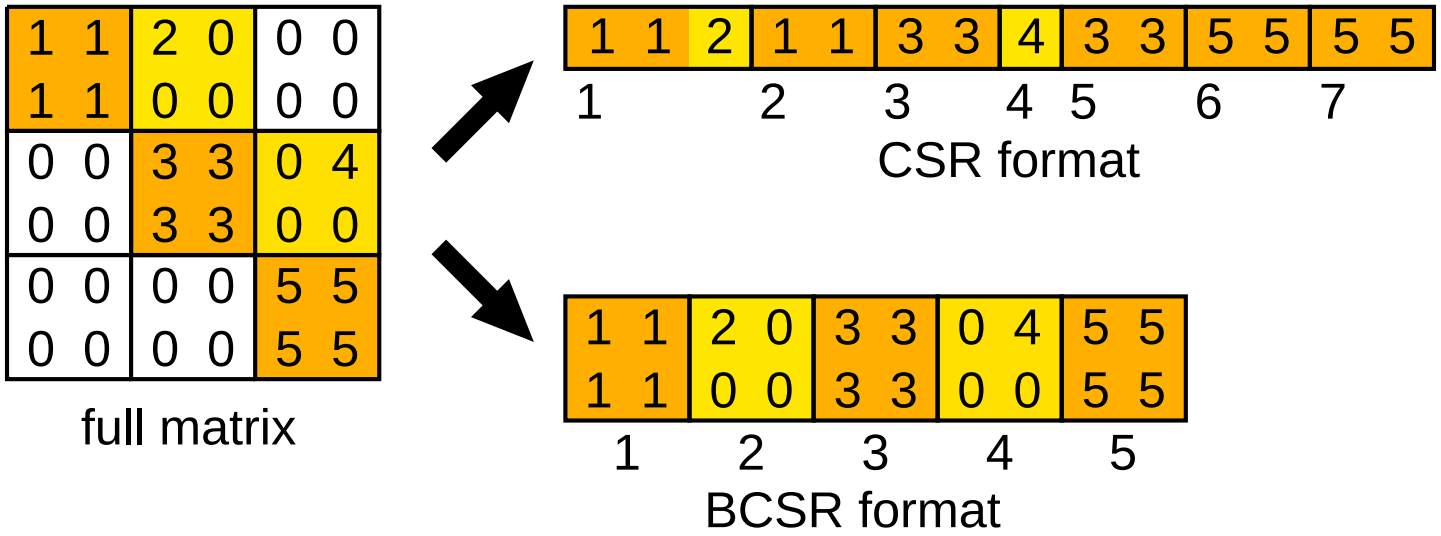


Sparse Matrix Formats



- C++ libraries: libtensor, TiledArray, (Cyclops Tensor Framework)

Sparse Matrix Formats



- C++ libraries: libtensor, TiledArray, (Cyclops Tensor Framework)
- Fortran libraries: DBCSR

DBCSR: Distributed Block Compressed Sparse Row

- Best suitable candidate for my needs: DBCSR by CP2K developer group
- MPI and OpenMP parallel, can exploit Nvidia and AMD GPUs via CUDA and HIP.
- Very small block sizes possible with libxsmm
- Written in Fortran 2008
- Open-source, available on github at `https://github.com/cp2k/dbcsr`

Fortran to C to C++

Example: sparse tensor contraction

$$C_{klm} = 2.0 * A_{ijk} * B_{ijlm} + 1.0 * C_{klm} \tag{1}$$

Fortran to C to C++

Example: sparse tensor contraction

$$C_{klm} = 2.0 * A_{ijk} * B_{ijlm} + 1.0 * C_{klm} \tag{1}$$

```
TYPE(dbcsr_t_type) :: A, B, C

... ! setup tensors using dbcsr_t_create
... ! fill them with data

CALL dbcsr_t_contract(alpha = 2.0, tensor_1 = A, &
          tensor_2 = B, beta = 1.0, tensor_3 = C, &
          contract_1 = [1,2], notcontract_1 = [3], &
          contract_2 = [1,2], notcontract_2 = [3,4], &
          map_1 = [1], map_2 = [2,3])
```

Fortran to C to C++

Example: sparse tensor contraction

$$C_{klm} = 2.0 * A_{ijk} * B_{ijlm} + 1.0 * C_{klm} \tag{1}$$

```
0  void *A, *B, *C;
1  ... // setup tensors using c_dbcsr_t_create
2
3  int *conA, *nconA, *conB, *nconB, *mapAC, *mapBC;
4  int conAsize, nconAsize, conBsize, nconBsize,
5      mapACsize, mapBCsize;
6
7  ... // setup indices
8  dbcsr_t_contract(2.0, A, B, 1.0, C, conA, conAsize, nconA,
9                   nconAsize, conB, conBsize, nconB, nconBsize,
10                  mapAC, mapACsize, mapBC, mapBCsize, NULL,
11                  NULL, NULL, NULL, NULL, NULL, NULL, NULL,
12                  NULL, NULL, NULL, NULL);
```

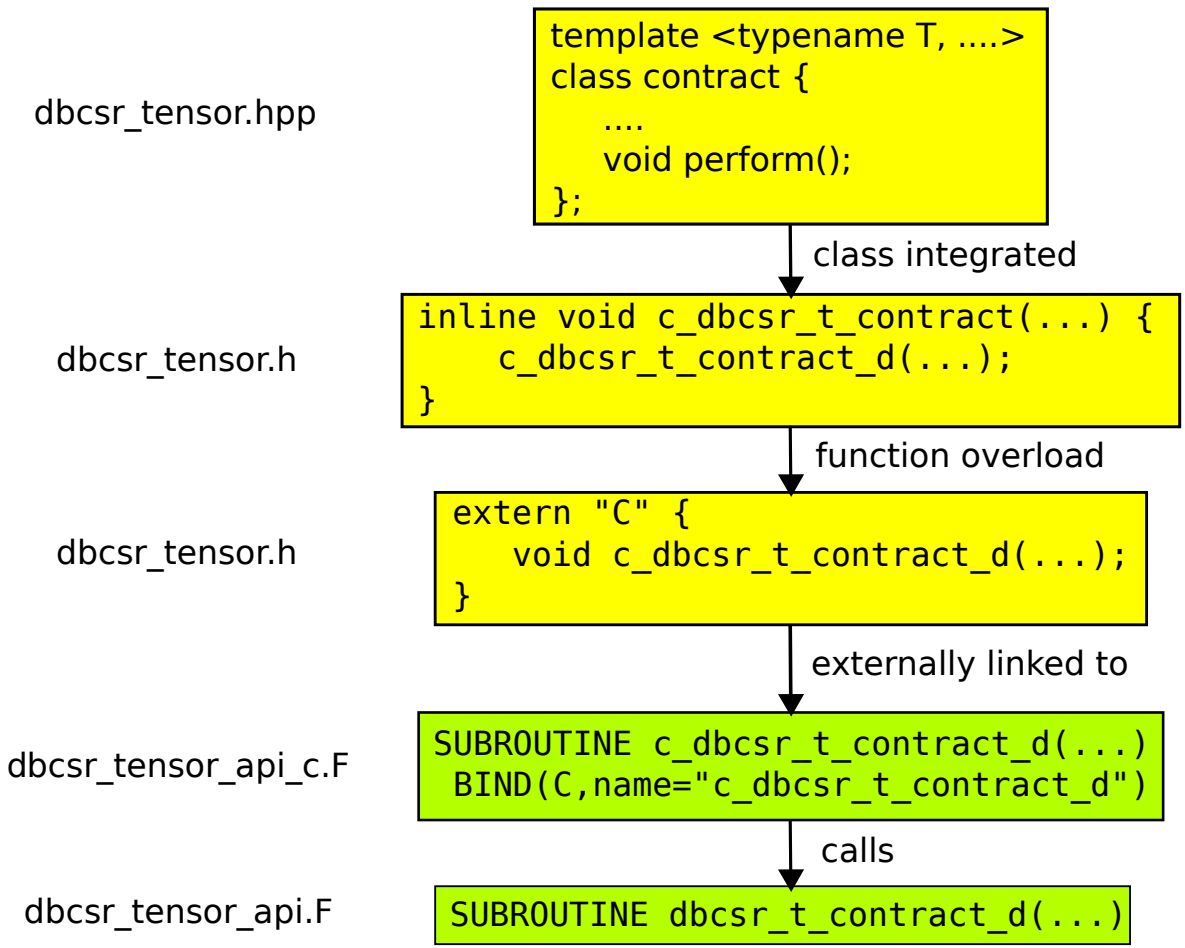
Fortran to C to C++

Example: sparse tensor contraction

$$C_{klm} = 2.0 * A_{ijk} * B_{ijlm} + 1.0 * C_{klm} \tag{1}$$

```
0 dbcsr::tensor<3,double> A, C;
1 dbcsr::tensor<4,double> B;
2
3 ... // setup tensors
4
5 dbcsr::contract(2.0, A, B, 1.0,C).con1({0,1}).ncon1({2})
6 .con2({0,1}).ncon2({2,3}).map1({0}).map2({1,2}).perform();
7
8 // alternatively:
9 dbcsr::contract(2.0,A,B,1.0,C).perform("ijk, ijlm -> klm");
```

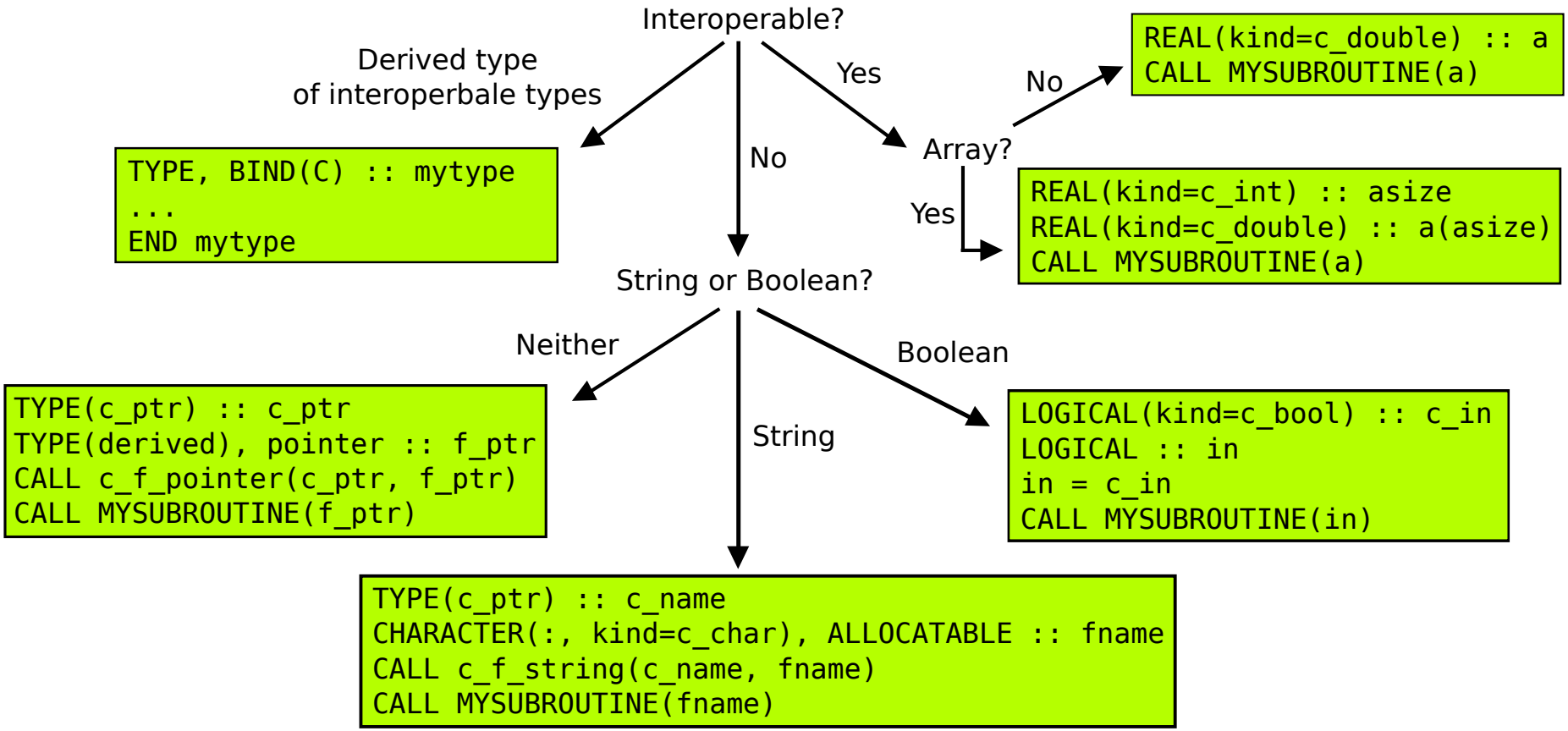
Calling Structure



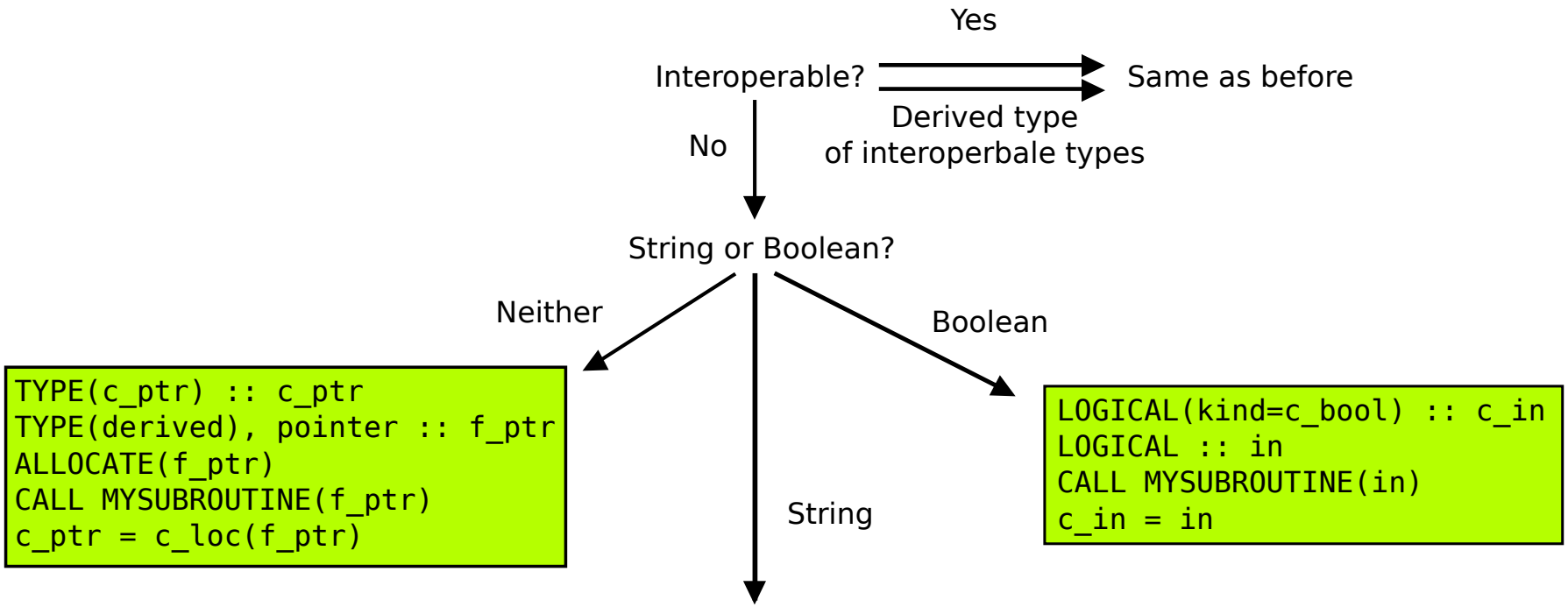
Interface Subroutine

```
0 USE, INTRINSIC :: ISO_C_BINDING
1
2 SUBROUTINE c_dbcsr_t_contract_d (...) &
3     BIND(C,name="c_dbcsr_t_contract_d")
4
5     CALL dbcsr_t_contract_d(...)
6
7 END SUBROUTINE
```

Interfacing to Subroutines: INTENT(IN)



Interfacing to Subroutines: INTENT(OUT)



Interoperable types

```
REAL(kind=c_double), INTENT(IN), VALUE :: c_alpha, c_beta
REAL(kind=c_double), INTENT(IN), OPTIONAL :: c_filter_eps
INTEGER(kind=c_int), INTENT(IN), OPTIONAL :: c_unit_nr
INTEGER(kind=c_long_long), INTENT(INOUT), &
    OPTIONAL :: c_flop

CALL dbcsr_t_contract(..., alpha=c_alpha, beta=c_beta, &
   filter_eps = c_filter_eps, unit_nr = c_unit_nr ...)
```

```
INTEGER(kind=c_int), INTENT(IN), VALUE :: c_con_1_size, ...
INTEGER(kind=c_int), INTENT(IN) :: c_con1(c_con_1_size), ...

INTEGER(kind=c_int), INTENT(IN), DIMENSION(2, con_1_size),&
        OPTIONAL :: c_bounds_1, ...

CALL dbcsr_t_contract_d(..., contract_1 = c_con_1 + 1, &
                        bounds_1 = c_bounds + 1, ...)
```

Interoperable types and arrays: remarks

- Arrays allocated in C cannot be deallocated in Fortran and vice-versa (compiler-dependant)

```
0 int* c_array = nullptr;
1 c_allocate_array_in_fortran(&c_array);
2
3 free(c_array); // compiler errors for Intel and Cray
```

Interoperable types and arrays: remarks

- Arrays allocated in C cannot be deallocated in Fortran and vice-versa (compiler-dependant)

```
0 int* c_array = nullptr;
1 c_allocate_array_in_fortran(&c_array);
2
3 free(c_array); // compiler errors for Intel and Cray
```

- Ideally, we should know the array size in C before passing it to Fortran, so we can allocate it

Interoperable types and arrays: remarks

- Arrays allocated in C cannot be deallocated in Fortran and vice-versa (compiler-dependant)

```
0 int* c_array = nullptr;
1 c_allocate_array_in_fortran(&c_array);
2
3 free(c_array); // compiler errors for Intel and Cray
```

- Ideally, we should know the array size in C before passing it to Fortran, so we can allocate it
- To allow cross-language allocation, we have to use CFI descriptors (not used here).

Non-interoperable types

- In our case, we use pointers:

```
TYPE(c_ptr), INTENT(INOUT) :: c_tensor_1, c_tensor_2, &
                              c_tensor_3
TYPE(dbcsr_t_type), POINTER :: tensor_1, tensor_2, tensor_3

CALL c_f_pointer(c_tensor_1,tensor_1)
CALL c_f_pointer(c_tensor_2,tensor_2)
CALL c_f_pointer(c_tensor_3,tensor_3)

CALL dbcsr_t_contract(..., tensor_1 = tensor_1, &
                           tensor_2 = tensor_2, &
                           tensor_3 = tensor_3 ...)
```

Non-interoperable types

- Optional non-interoperable types:

```
0  TYPE(c_ptr), INTENT(OUT), OPTIONAL :: c_pgrid_1, c_pgrid_2..
1  TYPE(dbcsr_t_pgrid_type), POINTER :: pgrid_1, pgrid_2, ...
2
3  IF (PRESENT(c_pgrid_1) .AND. &
4      .NOT. PRESENT(c_pgrid_2) .AND. ...) THEN
5
6    NULLIFY(pgrid_1)
7
8    CALL dbcsr_t_contract(..., pgrid_1 = pgrid_1, ...)
9    c_pgrid_1 = c_loc(pgrid_1)
10
11 ELSE IF ... ! differentiate other cases
12   ...
13 ELSE
14   CALL dbcsr_t_contract(...)
15 ENDIF
```

Final Subroutine

```
0 SUBROUTINE c_dbcsr_t_contract_d(c_alpha, c_tensor_1, &
1     c_tensor_2, c_beta, c_tensor_3, c_contract_1, contract_1_size
2     c_notcontract_1, notcontract_1_size, c_contract_2, &
3     contract_2_size, c_notcontract_2, notcontract_2_size, &
4     c_map_1, map_1_size, c_map_2, map_2_size, c_bounds_1, &
5     c_bounds_2, c_bounds_3, c_optimize_dist, c_pgrid_opt_1, &
6     c_pgrid_opt_2, c_pgrid_opt_3, c_filter_eps, c_flop, &
7     c_move_data, c_retain_sparsity, c_unit_nr, c_log_verbose) &
8     BIND(C, name="c_dbcsr_t_contract_d")
```

Templating Using FYPP

- To easily duplicate the subroutine for multiple types (float, double, complex ...), we can use a preprocessor
- FYPP: a Python powered preprocessor to extent Fortran with metaprogramming capabilities by Bálint Aradi (https://github.com/aradi/fypp)

```
#:for dsuffix, dbase, dctype in cf_dtype_float_list
SUBROUTINE c_dbcsr_t_contract_${dsuffix}$(...)
   BIND(C, name="c_dbcsr_t_contract_${dsuffix}$")

   ${dbase}$(kind=${dctype}$), INTENT(IN) ::   c_alpha
   ...
END SUBROUTINE
#:endfor
```

Templating Using FYPP

- generates:

```
0  SUBROUTINE c_dbcsr_t_contract_f(...) BIND(C,...)
1   REAL (kind=c_float), INTENT(IN) ::   c_alpha
2     ...
3  END SUBROUTINE
4
5  SUBROUTINE c_dbcsr_t_contract_z(...) BIND(C,...)
6   COMPLEX(kind=c_float_complex), INTENT(IN) :: c_alpha
7     ...
8  END SUBROUTINE
9
10 SUBROUTINE c_dbcsr_t_contract_d(...) BIND(C,...)
11  REAL (kind=c_double), INTENT(IN) ::   c_alpha
12    ...
13 END SUBROUTINE
14
15  ...
```

Extern C Function

- Similarly, we can use the preprocessor in the header file:

```
extern "C" {
#:for dsuffix, ctype in c_dtype_float_list
void c_dbcsr_t_contract_${dsuffix}$ (
       const ${ctype}$ c_alpha,
       void* c_tensor_1,
       void* c_tensor_2,
       const ${ctype}$ c_beta,
       void* c_tensor_3,
       ...);
#:endfor
}

#:for dsuffix, ctype in c_dtype_float_list
inline void c_dbcsr_t_contract (const ${ctype} c_alpha, ...)
         c_dbcsr_t_contract_${dsuffix}$(c_alpha, ...);
}
#:endfor
```

Wrapping it up in a C++ class

- Finally, we can use all the power of modern C++ templating to generate a more object-oriented structure:

```
// named parameter/constructor idiom
dbcsr::tensor<3> A = dbcsr::tensor<2>::create()
    .name("A").ngrid(grid3).map1({0}).map2({1,2})
    .blk_sizes(block_sizes);

dbcsr::tensor<3> B = dbcsr::tensor<2>::create_template(A)
    .name("B");
...

dbcsr::contract(2.0, A, B, 3.0, C)
    .move(true).log_verbose(true)
    .perform("ijk, ijl -> kl");
```

Where to find it...

- The C/Fortran interface is available at

  `https://github.com/cp2k/dbcsr`

- The C++ interface is available at

  `https://github.com/ambmax00/chem/`

  Work in progress: Computing excited state properties of molecules using sparse tensor algebra.

Contact me if you have any questions:

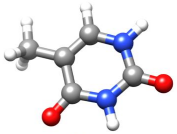
- `maximilien.ambroise@iwr.uni-heidelberg.de`

Acknowledgments

- Prof. Andreas Dreuw, and his research group at Heidelberg Univeristy
- Prof. Christian Ochsenfeld, and his research group at LMU
- The COSINE network and its nodes
- Alfio Lazarro and Patrick Seewald from the DBCSR development team

IWR Interdisciplinary Center for Scientific Computing

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386



$$i\hbar\frac{\mathrm{d}\Psi}{\mathrm{d}t} = H\Psi$$

COSINE

COMPUTATIONAL SPECTROSCOPY IN NATURAL SCIENCES AND ENGINEERING

European Commission