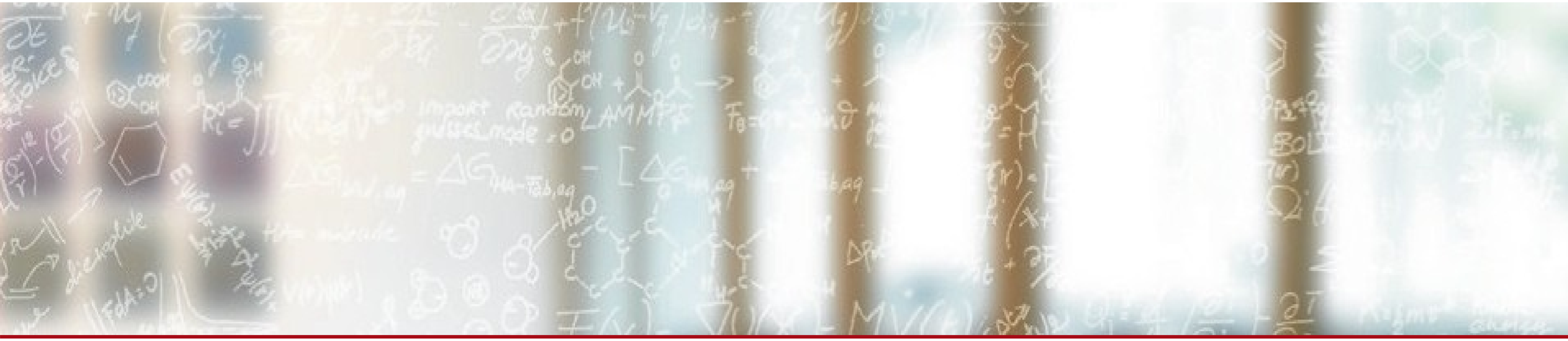


A C++-type-aware approach to bonded forces in classical Molecular Dynamics

Sebastian Keller
October 14th 2021

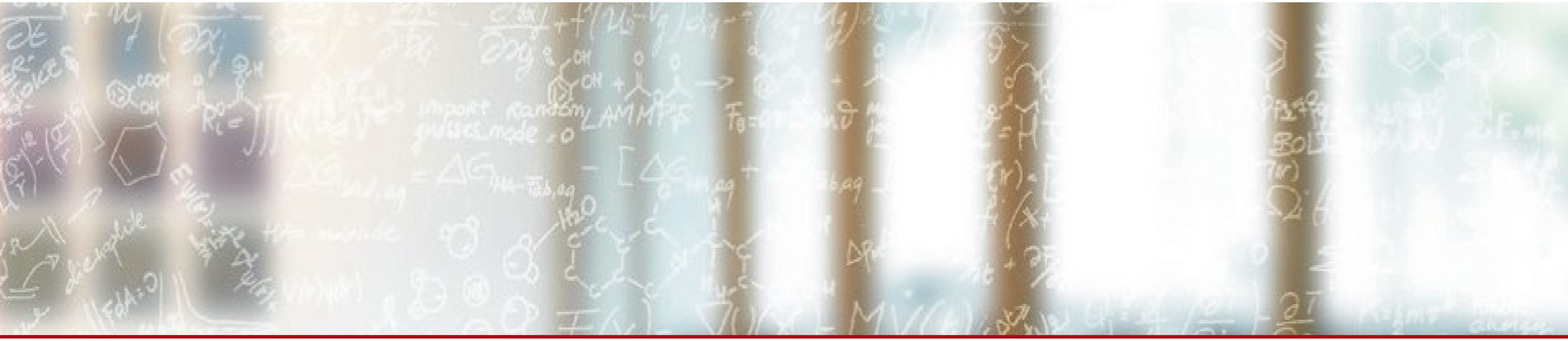
Background and motivation

Ongoing work at CSCS and KTH Stockholm: Modernization of bonded molecular forces in GROMACS.

- Current implementation in C
- Update to C++17
    - Strong typing and type lists
    - Tuples
    - SFINAE

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



A short introduction to bonded molecular forces in classical Molecular Dynamics

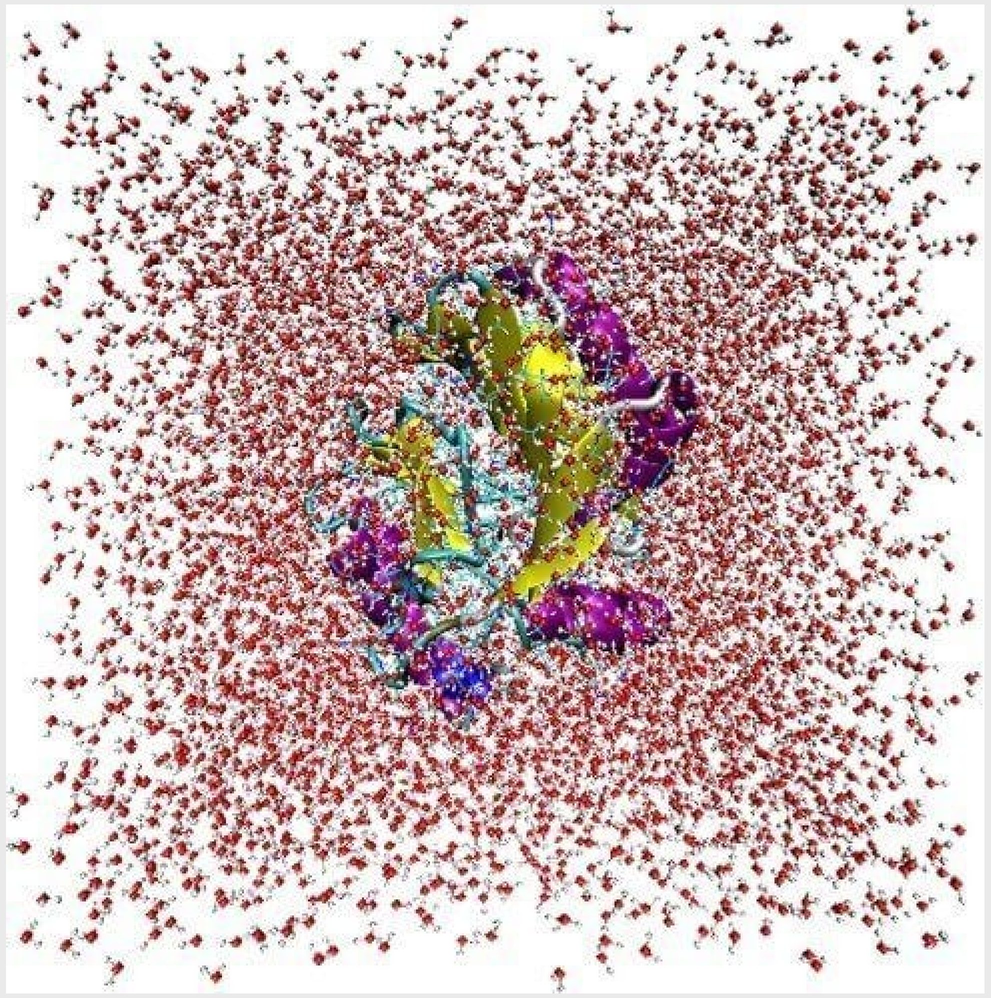
Classical Molecular Dynamics

Classical Molecular Dynamics (MD):

Approximation of atomic motion with Newtonian force laws

Applications:

- Proteins
- Enzymes
- Lipids
- Membranes
- Drug docking



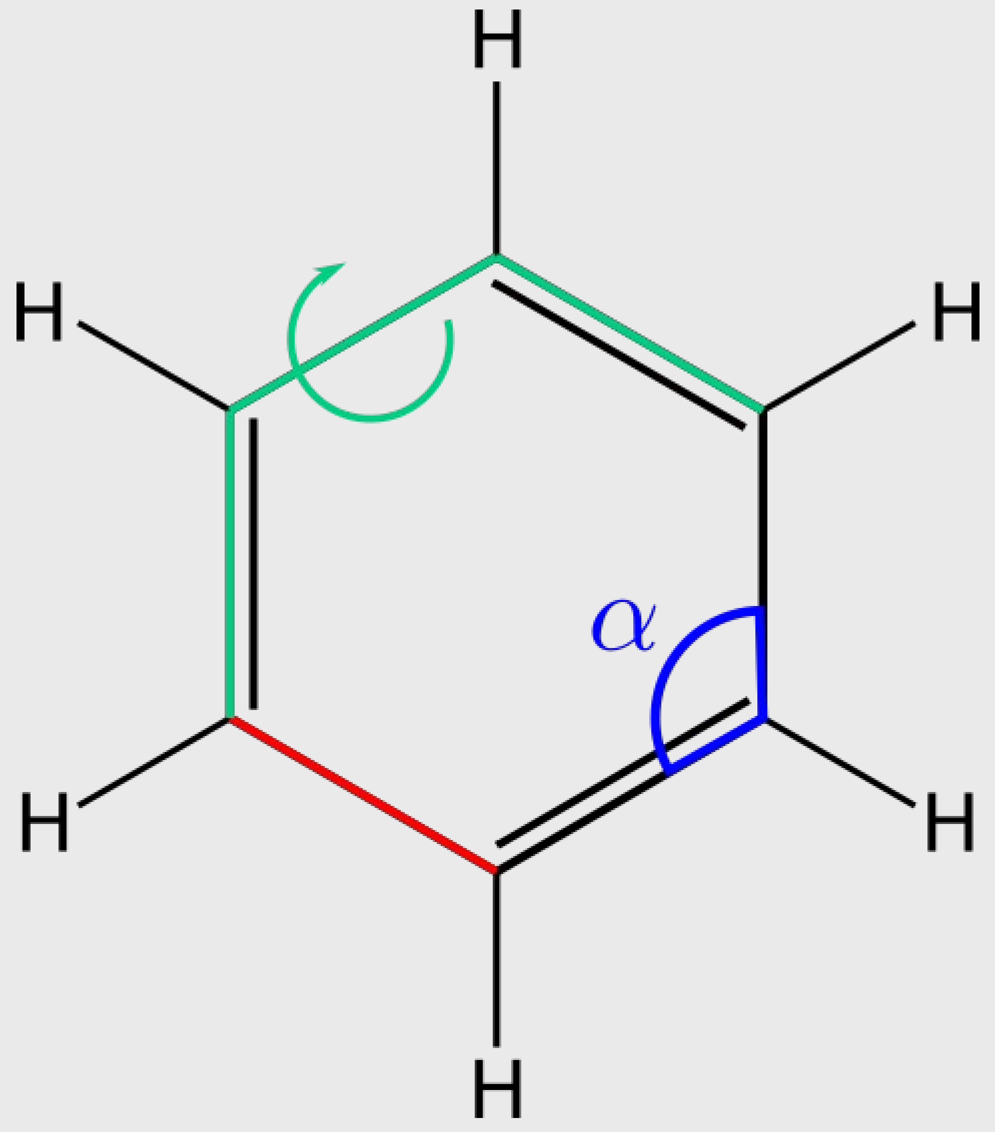
Bonded forces in molecules

Categories of forces due to bonded interactions:

- Bonds: 2 atoms 🟥
- Angles: 3 atoms 🟦
- Dihedrals: 4 atoms 🟩

Ongoing work: redesign of the bonded forces module in GROMACS

Moving from C to C++-17



Bonded force example: harmonic bonds

Harmonic bonds are like springs:

$$f(r) = -k\,(r - r_0)$$

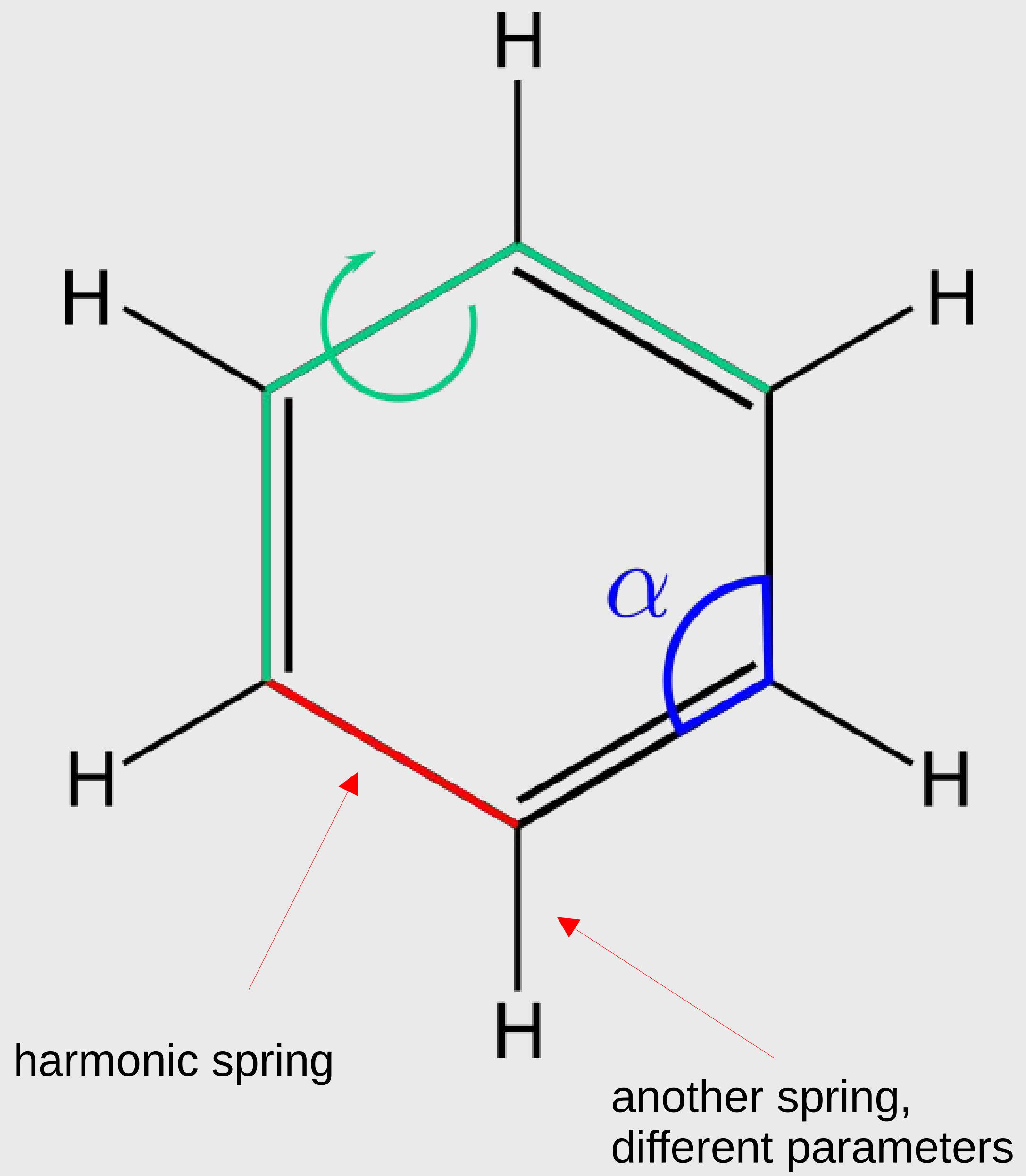
The bond has two parameters:

- spring constant k
- equilibrium distance r_0

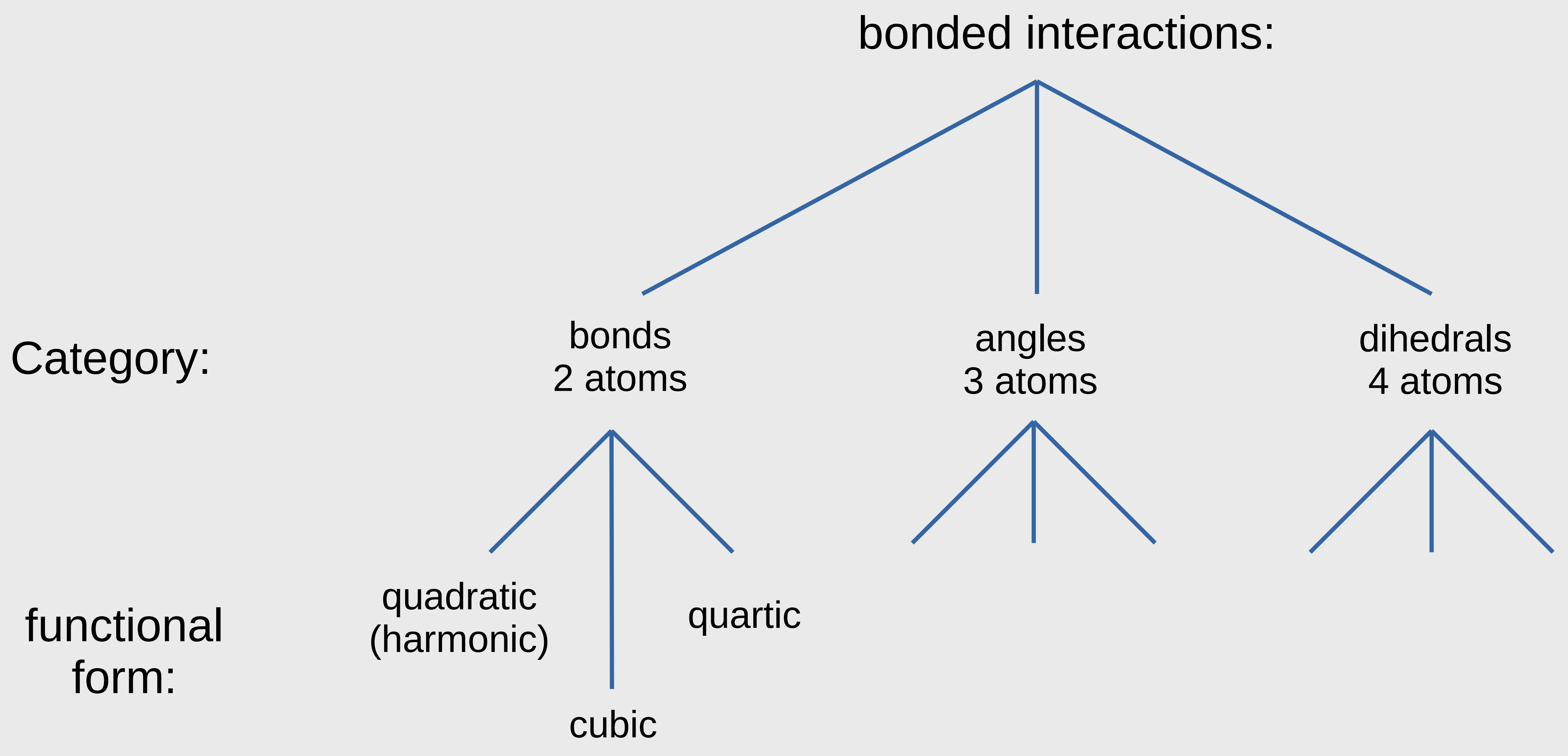
Different chemical bonds can be modeled with different parameters

But, different functional form (not harmonic) also possible

Different functional = different code
= different **interaction type**



Bonded forces classification



list of types: B_{harm}, B_{cubic}, B_{quart}, … , A_{harm}, A_{cubic}, … , D_{proper}, $D_{improper}$, $D_{RyckartBellman}$, ...

total ~30 types

Representation per interaction

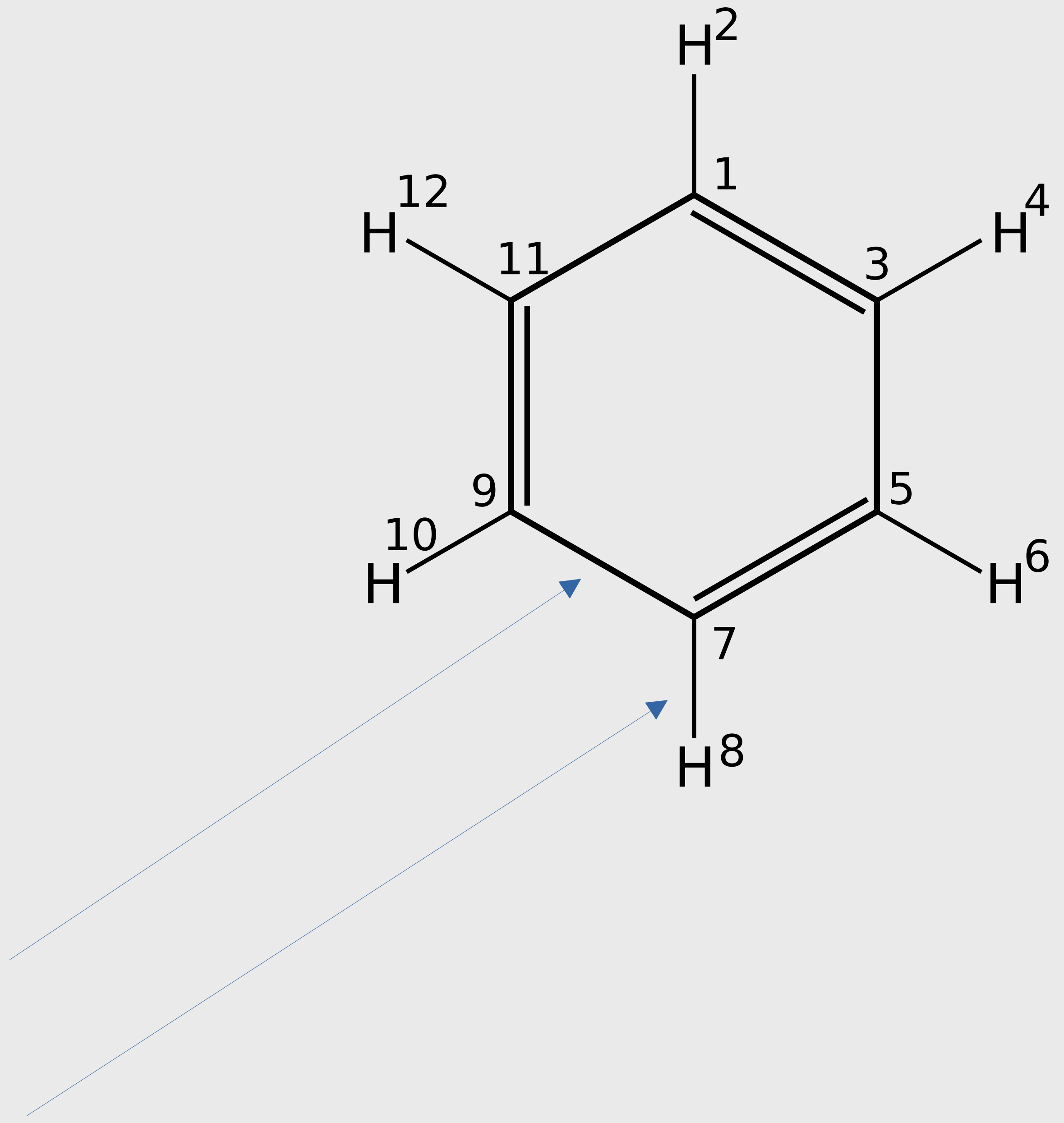
How are bonded interactions represented?

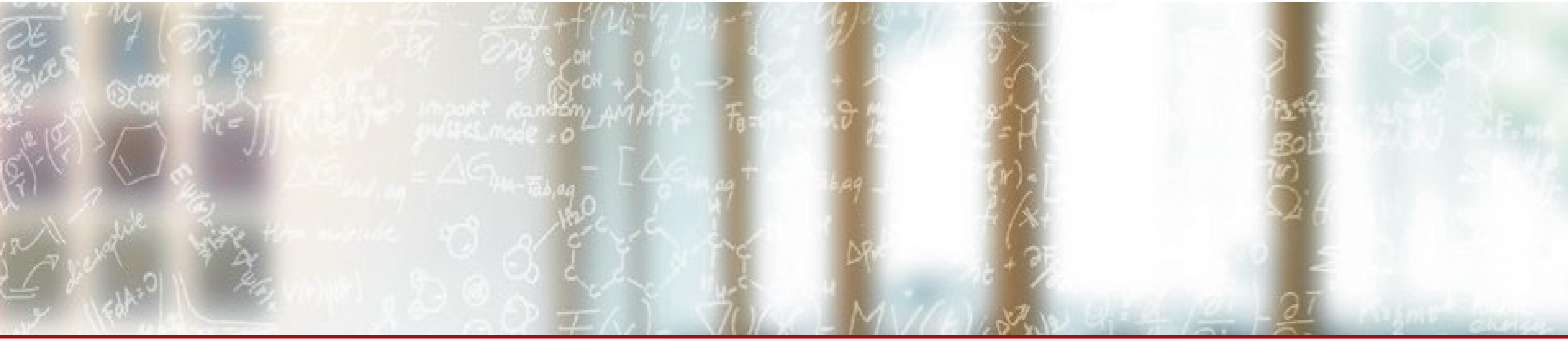
- atoms are numbered
- **parameter table**:

Index	k	r_0
0	$k_{C\text{-}C}$	$r_{C\text{-}C}$
1	$k_{C\text{-}H}$	$r_{C\text{-}H}$

- **index table**:

atom 1	atom 2	parameter
7	9	0
7	8	1





Part 1: Implementation in C

Interaction types in old-school C

GROMACS puts all bonded interaction types into a **union**.

An instance of `t_iparams` will take as much space as the biggest interaction type in the union.

```c
typedef union t_iparams
{
    struct
    {
        float r, k;
    } harmonic;

    struct
    {
        float klinA, aA, klinB, aB;
    } linangle;

    struct
    {
        float  theta, c[5];
    } qangle;

    /* ... */

};
```

Parameter and index tables in old-school C

- `InteractionDefinitions` contains bonded interactions of the entire system

```
struct InteractionDefinitions
{
    // parameter table
    std::vector<t_iparams> iparams;

    // index table per interaction type
    std::array<std::vector<int>, F_NRE> il;
};
```

Parameter table flattened over all interaction types.

Interaction indices

Number of interaction types is F_NRE

Looping over interaction types in old-school C

- `computeAllTypes` computes all forces due to bonded interactions

```
void computeAllTypes(const InteractionDefinitions& idef,
                     const float3*                 x,
                     float3*                       f)
{
    // loop over all interaction types
    for (int ftype = 0; ftype < F_NRE; ++type)
    {
        // retrieve function pointer from a table
        bondFunction* bonded = bondedInteractionFunctions[ftype];

        // compute all forces for ftype
        bonded(idef.iparams, idef.il[ftype], x, f);
    }
}
```

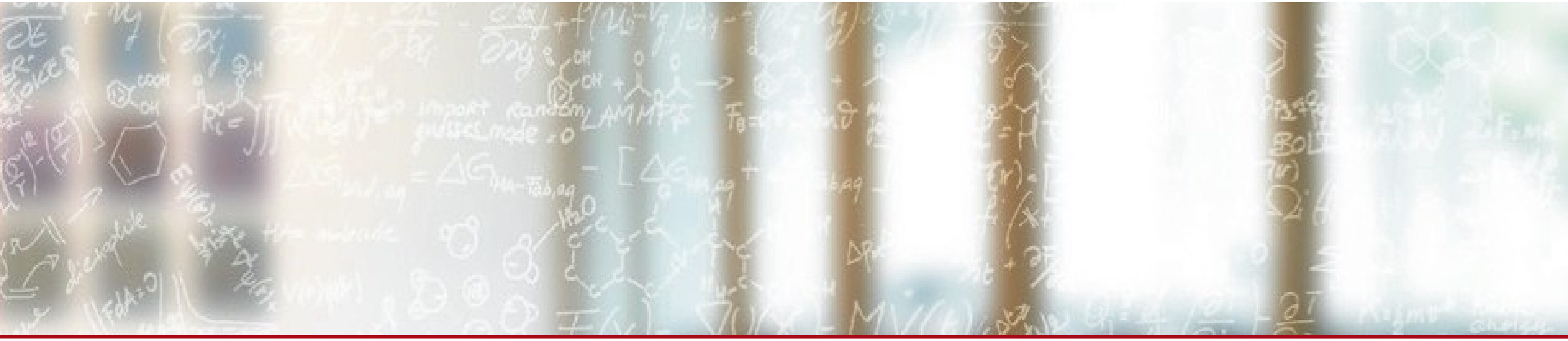
Looping over interactions in old-school C

```
// calculate forces due to harmonic bonds
void harmonicBonds(const t_iparams* params,
                   const int*       indices,
                   int              numIndices
                   const float3*    x,
                   float3*          f)
{
    // loop over all harmonic bonds
    for (int a = 0; a < numIndices; ++a)
    {
        // load indices
        int i          = indices[a];
        int j          = indices[a+1];
        int paramIndex = indices[a+2];

        // load parameters
        float r0 = params[paramIndex].harmonic.r;
        float k  = params[paramIndex].harmonic.k;

        // load coordinates

        // compute scalar force

        // store force vectors
    }
}
```

That's the only type-specific part. The rest is duplicated between types!

Moving to C++

- Distinct C++ types instead of C-union
- Avoid code duplication:
    - no separate function for looping over interactions of each type
    - one single interaction dispatch per category,
      shared index, parameter and coordinate loads, shared force spread and stores
- Extensibility: single place to add new interaction types



Part 2: Parameter and index tables with distinct C++ types

Distinct C++ types for interaction types

Instead of unions, we move to proper C++ types:

```
class HarmonicBond
{
public:
    HarmonicBond(float k, float r) : k_(k), r_(r) { }

    float forceConstant() const { return k_; }
    float equilConstant() const { return r_; }

private:

    // spring constant
    float k_;

    // equilibrium distance
    float r_;
};
```

C++ parameter and index list

```
template<size_t N>
using IndexArray = std::array<int, N>;

template<class Interaction>
using InteractionIndex = IndexArray<NCenter<Interaction>{} + 1>;


template<class InteractionType>
struct ParameterIndexTable
{
    using type = InteractionType;

    // parameter table
    std::vector<InteractionType> parameters;

    // index index table
    std::vector<InteractionIndex<InteractionType>> indices;
};
```

Number of indices depends on the category of `InteractionIndex` via `NCenter<>{}`!

CSCS ETH zürich

Adding an interaction category trait

We want

`NCenter<Interaction>{}`

to evaluate to the number of atoms involved in `Interaction`.

Let’s start by storing the category (number of involved atoms) of each interaction type.

```
template<class... Ts>
struct TypeList {};

using TwoCenterTypes   = TypeList<HarmonicBond, QuarticBond, ...>;
using ThreeCenterTypes = TypeList<HarmonicAngle, LinearAngle, ...>;
using FourCenterTypes  = TypeList<ProperDihedral, ...>;
```

Adding an interaction category trait

Implementation of `NCenter<Interaction>{}`

```
template<class Interaction, class = void>
struct NCenter { };


template<class Interaction>
struct NCenter<Interaction,
               std::enable_if_t<Contains<Interaction, TwoCenterTypes>{}>>
: std::integral_constant<std::size_t, 2>
{ };

template<class Interaction>
struct NCenter<Interaction,
               std::enable_if_t<Contains<Interaction, ThreeCenterTypes>{}>>
: std::integral_constant<std::size_t, 3>
{ };
```

Note: `Contains<A, B>{}` evaluates to true if A is contained in template parameters of B.

Adding an interaction category trait

How is `NCenter<HarmonicBond>{}` evaluated?

Base template: `NCenter<HarmonicBond>{}` is `NCenter<HarmonicBond, void>{}`

First specialization:

```
// evaluates to true
Contains<HarmonicBond, TwoCenterTypes>{}>

// evaluates to void
std::enable_if_t<Contains<HarmonicBond, TwoCenterTypes>{}>
```

So the first specialization *also* produces `NCenter<HarmonicBond, void>{}`.

The specialization is *more specialized*, so it is selected.
Therefore: `NCenter<HarmonicBond>{}` is derived from `std::integral_constant` with a value of 2.

Adaptive index tables

Thanks to `NCenter<Interaction>{}`, the following `ParameterIndexTables` have the correct number of indices per interaction.

```
// Index table with 3 indices per interaction
ParameterIndexTable<HarmonicBond>  harmonicBonds;

// Index table with 4 indices per interaction
ParameterIndexTable<HarmonicAngle> harmonicAngle;
```

The complete C++ type for all bonded interactions

Recall the C-style `InteractionDefinitions` that contained all bonded interactions.
What about C++?

```
// C++ type for bonded interactions of the entire system

using BondedInteractionData = std::tuple<ParameterIndexTable<HarmonicBond>,
                                         ParameterIndexTable<CubicBond>,
                                         ...
                                         ParameterIndexTable<HarmonicAngle>,
                                         ParameterIndexTable<LinearAngle>,
                                         ...
                                         ParameterIndexTable<ProperDihedral>,
                                         ...
                                    >
```

But: we already have the TypeLists for the interaction categories, we don't want to repeat ourselves! How can we generate the type above from those lists? (remember: there's ~30 types!

Automatic generation of BondedInteractionData

```
// TypeList of all supported interaction types
using InteractionTypes = Fuse<TwoCenterTypes, ThreeCenterTypes, FourCenterTypes>;

// First create a list of ParameterIndexTable for all interaction types,
// then put all lists into the std::tuple
using BondedInteractionData
    = Reduce<std::tuple, Map<ParameterIndexTable, InteractionTypes>>;
```

We needed some additional operations for TypeList to achieve that:

- `Fuse<Ls...>` : Combines multiple TypeLists into a single TypeList
- `Map<T, Ls>` : Instantiates T with each element in list Ls, one element at a time
  Example: `Map<T, TypeList<int, float>> == TypeList<T<int>, T<float>>`
- `Reduce<T, Ls>` : Instantiate T with all elements of Ls
  Example: `Reduce<T, TypeList<int, float> == T<int, float>`

TypeLists: further reference

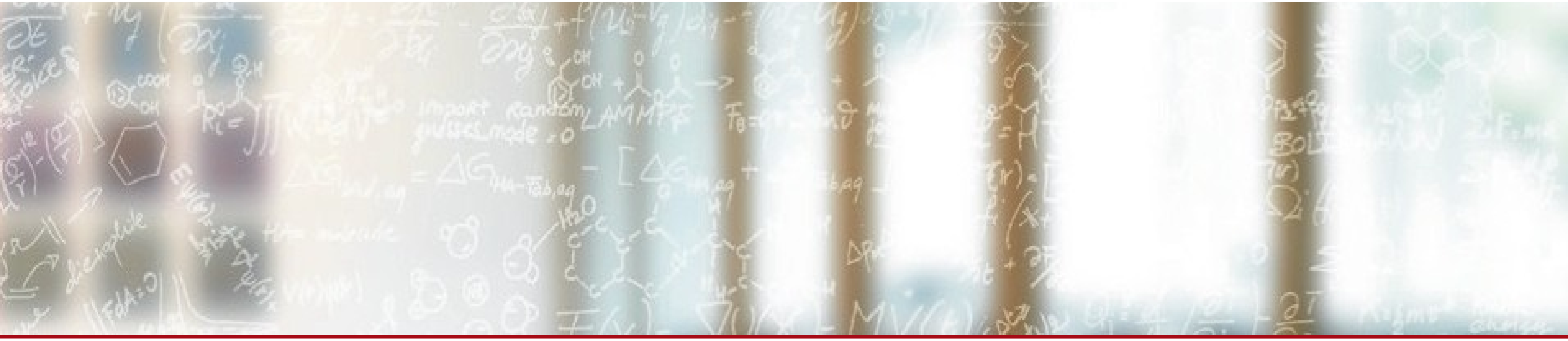
Due to time constraints, please refer to the following resources for exhaustive detail about the mentioned template meta functions revolving around TypeLists:

- These tests further document the problem statement by matching expected output to given input:

  https://gitlab.com/gromacs/gromacs/-/blob/master/api/nblib/util/tests/traits.cpp

- Complete implementation (also of `Contains`):

  https://gitlab.com/gromacs/gromacs/-/blob/master/api/nblib/util/traits.hpp



Part 3: Computing all interactions of one type in C++

Common dispatch for N-center interaction types

```
// overload for 2-center interaction types to compute forces
template<class TwoCenterType>
void dispatchInteraction(IndexArray<3>        index,
                         const TwoCenterType* bondParameters,
                         const float3*        x,
                         float3*              f)
{
    // reused part
    int i = std::get<0>(index);
    int j = std::get<1>(index);

    float3 xi = x[i];
    float3 xj = x[j];
    float3 dx = xi – xj;

    TwoCenterType bond = bondParameters[std::get<2>(index)];

    // type-specific part
    computeTwoCenter(bond, dx, &f[i], &f[j]);
}
```

Common dispatch for N-center interaction types

```
// overload for 3-center interaction types to compute forces
template<class ThreeCenterType>
void dispatchInteraction(IndexArray<4>         index,
                         const TwoCenterType* angleParameters,
                         const float3*        x,
                         float3*              f)
{
    // reused part
    int i = std::get<0>(index);
    int j = std::get<1>(index);
    int k = std::get<2>(index);

    float3 xi = x[i];
    float3 xj = x[j];
    float3 xk = x[k];
    float3 xik = xi – xk;
    float3 xjk = xj – xk;

    ThreeCenterType angle = angleParameters[std::get<3>(index)];
    float alpha = computeAngle(xik, xjk);

    // type specific part
    computeThreeCenter(angle, alpha, xik, xjk, &f[i], &f[j], &f[k]);
}
```

Looping over interactions

Thanks to the overloads of `dispatchInteraction`, we only need a single implementation to loop over all interactions that works for all types.

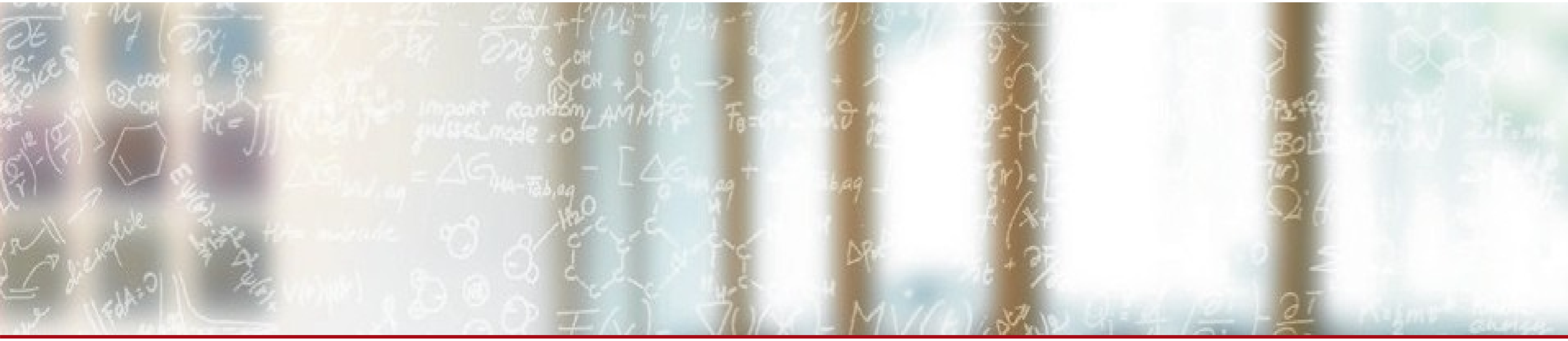
```
template<class InteractionType>
auto computeForces(std::span<const InteractionIndex<InteractionType>> indices,
                   const InteractionType* parameters,
                   const float3*          x,
                   float3*                f)
{
    for (const auto& index : indices)
    {
        dispatchInteraction(index, parameters, x, f);
    }
}
```

Looping over interaction types

- The C++ implementation uses a tuple, we can’t write a for-loop over it:

```
void computeAllTypes(const BondedInteractionData& idef,
                     const float3*                x,
                     float3*                      f)
{
    for (int i = 0; i < BondedInteractionData.size(); ++i)
    {
        // error: i is not a compile time constant
        computeForces(std::get<i>(idef).indices
                      std::get<i>(idef).parameters,
                      x, f);
    }
}
```

- Obviously we don’t want to unroll the loop by hand
- What we need is something like `for_each_tuple(f, tuple)` that calls `f` on each element of `tuple`



Part 4: Computing interactions for all bonded types in C++

or: can we loop over tuples?

How to implement `for_each_tuple(f, tuple)`?

- We can use `std::apply` as the first step to convert the tuple elements into a parameter pack:

```
std::apply(g, tuple) calls g(std::get<0>(tuple), std::get<1>(tuple), ...)
```

- g is called with all elements. How do we define g in terms of `f`, which we want to call with each tuple element?

```
template<class F, class... Ts>
void g(F&& f, const Ts&... args)
{

    // error: can't create variadic number of statements from single expression

    f(args)...;
}
```

How to implement `for_each_tuple(f, tuple)`?

- We can dispose of the return values with a helper function:

```
template<class... Ts>
void dispose_return_value(Ts&&...) { }

template<class F, class... Ts>
void g(F&& f, const Ts&... args)
{
    // error if f returns void. argument type cannot be void

    dispose_return_value(f(args)...);
}
```

How to implement `for_each_tuple(f, tuple)`?

- Remember the comma operator:

`(a, b)` : *Evaluate a, discard return value, then evaluate b and return the result.*

```
template<class... Ts>
void dispose_return_value(Ts&&...) { }

template<class F, class... Ts>
void g(F&& f, const Ts&... args)
{
    dispose_return_value((f(args), 0)...);
}
```

How to implement `for_each_tuple(f, tuple)`?

- But now, we don't really need the helper function any more. The comma expression returns zeros and we can just put them in an (unused) initializer list:

```cpp
template<class F, class... Ts>
void g(F&& f, const Ts&... args)
{
    [[maybe_unused]] std::initializer_list<int>{ (f(args), 0)... };
}
```

How to implement `for_each_tuple(f, tuple)`?

- We can now combine g with `std::apply` and implement the final `for_each_tuple`

```
template<class F, class... Ts>
void for_each_tuple(F&& func, std::tuple<Ts...>& tuple_)
{
    auto g = [f=func](auto&... args)
    {
        [[maybe_unused]] std::initializer_list<int>{ (f(args), 0)... };
    };

   std::apply(g, tuple_);
}
```

- g is implemented as a generic lambda function with `f` captured from the parent scope

Back to the original problem: looping over types

- Instead of manual loop-unrolling, we can now use `for_each_tuple`

```
void computeAllTypes(const BondedInteractionData& idef,
                     const float3*                x,
                     float3*                      f)
{
    auto computeOneType = [x, f](const auto& idefElement)
    {
        computeForces(idefElement.indices, idefElement.params, x, f);
    }

    computeOneType(std::get<0>(idef));
    computeOneType(std::get<1>(idef));
    ...
}
```

The final implementation

- Instead of manual loop-unrolling, we can now use `for_each_tuple`

```
void computeAllTypes(const BondedInteractionData& idef,
                     const float3*                x,
                     float3*                      f)
{
   auto computeOneType = [x, f](const auto& idefElement)
   {
       computeForces(idefElement.indices, idefElement.params, x, f);
   }

   for_each_tuple(computeOneType, idef);
}
```

Summary

- Replaced a C-union with distinct C++ types
- Reduced code duplication:
  - one single loop over interactions reused for all types
  - one single interaction dispatch per NCenter category,

    shared index, parameter and coordinate loads, shared force spread and stores
  - type specific force functions are reusable in the GPU implementation
- Extensibility: new types can be added by just appending the new C++ type to a type list, interaction loops and force reductions generated automatically
- New optimization opportunities: C++ type awareness means we can automatically generate code paths for syntetic aggregate types, e.g. combining one angle with two bonds to improve cache hits