Contemporary C++ in HPC codes

Jakub Gałecki[1,2]

October 17th 2023, UQ Computers & Fluids Discussion Group

[1] Department of Aerodynamics, Warsaw University of Technology
[2] Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw

Introduction

Why C++?

- Lingua franca for HPC
- C interop
- Zero cost abstractions
- Rich ecosystem of existing (legacy) software
- Still growing & evolving!

Introduction

Why C++?

- Lingua franca for HPC
- C interop
- Zero cost abstractions
- Rich ecosystem of existing (legacy) software
- Still growing & evolving!

Why should I care if I'm coding in X?

- What facilities does X provide for the problems of interest?
- How can I improve X?

Introduction

Why C++?

- Lingua franca for HPC
- C interop
- Zero cost abstractions
- Rich ecosystem of existing (legacy) software
- Still growing & evolving!

Why should I care if I'm coding in X?

- What facilities does X provide for the problems of interest?
- How can I improve X?

Maintainability carries real-world cost

Introduction

Why C++?

- Lingua franca for HPC
- C interop
- Zero cost abstractions
- Rich ecosystem of existing (legacy) software
- Still growing & evolving!

Why should I care if I'm coding in X?

- What facilities does X provide for the problems of interest?
- How can I improve X?

Maintainability carries real-world cost

Naming: modern C++ = $\{11, 14\}$, contemporary C++ ≥ 17

Agenda

- Background & motivation
- Examples:
  - Miscellanea
  - MPI wrapper
  - Physics injection

Agenda

- Background & motivation
- Examples:
  - Miscellanea
  - MPI wrapper
  - Physics injection

Please interrupt if you have questions/comments

Agenda

- Background & motivation
- Examples:
  - Miscellanea
  - MPI wrapper
  - Physics injection

Please interrupt if you have questions/comments

Click to see code: 

The least-squares finite element method, L3STER

The least-squares principle

Consider the abstract boundary-value problem:

$$\begin{cases} \mathscr{A}(\mathsf{u}) = \mathsf{f} & \text{in } \Omega \\ \mathscr{B}(\mathsf{u}) = \mathsf{g} & \text{on } \partial\Omega \end{cases}$$

We define the least-squares functional:

$$\mathscr{J}(\mathsf{u};\mathsf{f},\mathsf{g}) = \frac{1}{2}\left(\|\mathscr{A}(\mathsf{u}) - \mathsf{f}\|^2_{\Omega,0} + \|\mathscr{B}(\mathsf{u}) - \mathsf{g}\|^2_{\partial\Omega,0}\right)$$

The least-squares principle

Find $\mathsf{u} \in \mathscr{V}$ such that

$$\forall_{\tilde{\mathsf{u}}\in\mathscr{V}}\ \mathscr{J}(\mathsf{u};\mathsf{f},\mathsf{g}) \leq \mathscr{J}(\tilde{\mathsf{u}};\mathsf{f},\mathsf{g})$$

Restrict $\mathscr{V}$ to a finite-dimensional $\mathscr{V}_h = \operatorname{span}\{\Phi_1, ..., \Phi_N\}$

$$u = \sum_{i=1}^{N} u_i \Phi_i, \text{ where } u_i \in \mathbb{R}$$

We now have a convex algebraic minimization problem, where the stationary point must satisfy

$$\left[\int_\Omega \left(\mathscr{A}^T \Phi_i\right) \mathscr{A} \Phi_j + \int_{\partial\Omega} \left(\mathscr{B}^T \Phi_i\right) \mathscr{B} \Phi_j\right] [u_j]$$
$$= \left[\int_\Omega \left(\mathscr{A}^T \Phi_i\right) f + \int_{\partial\Omega} \left(\mathscr{B}^T \Phi_i\right) g\right]$$

Alternative view: weak formulation where the derivatives of the result of the operator w.r.t. the unknowns serve as the trial functions

The domain contribution to the local matrix is given by

$$\mathsf{K} = \left[\int_\Omega \left(\mathscr{A}^T \Phi_i\right) \mathscr{A} \Phi_j\right] \approx \sum_{q=1}^{Q} w_q \left[\left(\mathscr{A}^T \Phi_i\left(x_q\right)\right) \mathscr{A} \Phi_j\left(x_q\right)\right] = \mathsf{A}^T \mathsf{W}_d \mathsf{A}$$

- ✓ There's no weak formulation!
- ✓ BCs are trivial to impose (replace $\mathscr{A}$ with $\mathscr{B}$)
- ✓ The system is SPD regardless of $\mathscr{A}$
- ✓ Error norm is built in
- ✗ $\mathscr{A}$ must be first-order and linear
- ✗ Quadrature order is usually higher than for the Galerkin method, e.g. for 3D convection-diffusion $Q = 8p^3$

Jiang, B. (1998). *The Least-Squares Finite Element Method*. Springer Berlin Heidelberg.
https://doi.org/10.1007/978-3-662-03740-9

Assembly kernel

Linear first-order operators can be expressed as

$$\mathscr{A} = A_0 + \sum_{i=1}^{dim} A_i \frac{\partial}{\partial x_i}$$

where $A_i : \Omega \times (0, T] \to \mathbb{R}^{E \times U}$, E being the number of equations and U the number of unknowns (E and U may not be equal).

All we need to define a set of physics is to provide these matrices!

Different physics in different domains can be handled automatically.

- **L**east-**S**quares **S**calable **S**pec**T**ral **E**lement f**R**amework
- Written in C++ 20
- General PDE solver – no equations included in the core
- High premium placed on passing parameters (mostly kernel dimensions) at compile-time
- Low entry barrier: equations + mesh = solution

Miscellaneous C++ bits for HPC

std::span

Span = pointer + size

Non-owning view over a contiguous memory region

Convenient API:

- Compile-time or dynamically sized
- Sub-spans
- Range constructor

std::span

Span = pointer + size

Non-owning view over a contiguous memory region

Convenient API:

- Compile-time or dynamically sized
- Sub-spans
- Range constructor

```
auto v           = std::vector{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
auto v_span      = std::span{v};
auto drop3       = v_span.subspan(3);
auto drop3_take6 = drop3.subspan(0, 6);
std::print("{}\n", drop3_take6);
std::sort(drop3_take6.begin(), drop3_take6.end());
std::print("{}\n", v);
```

std::span

Span = pointer + size

Non-owning view over a contiguous memory region

Convenient API:

- Compile-time or dynamically sized
- Sub-spans
- Range constructor

```
1 auto v           = std::vector{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
2 auto v_span      = std::span{v};
3 auto drop3       = v_span.subspan(3);
4 auto drop3_take6 = drop3.subspan(0, 6);
5 std::print("{}\n", drop3_take6);
6 std::sort(drop3_take6.begin(), drop3_take6.end());
7 std::print("{}\n", v);
```

```
[7, 6, 5, 4, 3, 2]
[10, 9, 8, 2, 3, 4, 5, 6, 7, 1]
```

`std::mdspan`

- Multi-dimensional span
- Non-owning view over a contiguous memory region, with multidimensional access information encoded
- API inspired by Kokkos
- Allows strided access and various memory layouts
- Can serve as common interface for matrix types from different libraries
- Finally multi-argument `operator[]`!
- Sorry, still no support on Godbolt :(

std::atomic_ref

```cpp
template<typename T>
class atomic_ref;
```

Atomic reference type

Objects can be safely concurrently accessed from multiple threads, provided all accesses are via `std::atomic_ref`

Supports the usual set of operations and memory orders

Now full support for floating-point types!

std::atomic_ref

```cpp
void atomic_increment(double& val, double inc) {
  std::atomic_ref{val}.fetch_add(inc /*, mem_order */);
}
```

std::atomic_ref

```cpp
void atomic_increment(double& val, double inc) {
  std::atomic_ref{val}.fetch_add(inc /*, mem_order */);
}
```

```
atomic_increment(double&, double):
        mov     rax, qword ptr [rdi]
.LBB0_1:
        movq    xmm1, rax
        addsd   xmm1, xmm0
        movq    rcx, xmm1
        lock            cmpxchg qword ptr [rdi], rcx
        jne     .LBB0_1
        ret
```

std::atomic_ref

```cpp
void atomic_increment(double& val, double inc) {
  std::atomic_ref{val}.fetch_add(inc /*, mem_order */);
}
```

```
atomic_increment(double&, double):
        mov     rax, qword ptr [rdi]
.LBB0_1:
        movq    xmm1, rax
        addsd   xmm1, xmm0
        movq    rcx, xmm1
        lock            cmpxchg qword ptr [rdi], rcx
        jne     .LBB0_1
        ret
```

Note the use of CTAD...

MPI in contemporary C++

MPI wrapper

What you stand to gain:

MPI wrapper

What you stand to gain:

- Communicator ownership semantics:
  - Own
  - View
  - Duplicate

MPI wrapper

What you stand to gain:

- Communicator ownership semantics:
  - Own
  - View
  - Duplicate
- Comm buffer type deduction – no more `void*`

MPI wrapper

What you stand to gain:

- Communicator ownership semantics:
  - Own
  - View
  - Duplicate
- Comm buffer type deduction – no more `void*`
- RAII for MPI requests (and other things)

MPI wrapper

What you stand to gain:

- Communicator ownership semantics:
  - Own
  - View
  - Duplicate
- Comm buffer type deduction – no more `void*`
- RAII for MPI requests (and other things)
- Request tied with payload to automate memory management

MPI wrapper

What you stand to gain:

- Communicator ownership semantics:
  - Own
  - View
  - Duplicate
- Comm buffer type deduction – no more `void*`
- RAII for MPI requests (and other things)
- Request tied with payload to automate memory management
- **Range-enabled MPI calls**

MPI wrapper

What you stand to gain:

- Communicator ownership semantics:
  - Own
  - View
  - Duplicate
- Comm buffer type deduction – no more `void*`
- RAII for MPI requests (and other things)
- Request tied with payload to automate memory management
- **Range-enabled MPI calls**

What you stand to lose:

MPI wrapper

What you stand to gain:

- Communicator ownership semantics:
  - Own
  - View
  - Duplicate
- Comm buffer type deduction – no more `void*`
- RAII for MPI requests (and other things)
- Request tied with payload to automate memory management
- **Range-enabled MPI calls**

What you stand to lose:

- API surface

Range:

- represents an iterable sequence
- exposes iterator/sentinel pair via `.begin()` / `.end()`

C++20 ranges crash course

Range:

- represents an iterable sequence
- exposes iterator/sentinel pair via `.begin()` / `.end()`

View:

- view over part or all of a range
- possibly with additional transformations applied on top
- standard views are lazily evaluated
- composable via pipeline operator

Range:

- represents an iterable sequence
- exposes iterator/sentinel pair via `.begin()` / `.end()`

View:

- view over part or all of a range
- possibly with additional transformations applied on top
- standard views are lazily evaluated
- composable via pipeline operator

Borrowed range:

- range which does not own the underlying sequence
- iterators to a borrowed range can safely outlive their parent range
- a range type can be queried to check whether it's borrowed

C++20 ranges crash course

Range:

- represents an iterable sequence
- exposes iterator/sentinel pair via `.begin()` / `.end()`

View:

- view over part or all of a range
- possibly with additional transformations applied on top
- standard views are lazily evaluated
- composable via pipeline operator

Borrowed range:

- range which does not own the underlying sequence
- iterators to a borrowed range can safely outlive their parent range
- a range type can be queried to check whether it's borrowed

Ranges facilities are transparent to the compiler – no run-time penalty

Range:

- represents an iterable sequence
- exposes iterator/sentinel pair via `.begin()` / `.end()`

View:

- view over part or all of a range
- possibly with additional transformations applied on top
- standard views are lazily evaluated
- composable via pipeline operator

Borrowed range:

- range which does not own the underlying sequence
- iterators to a borrowed range can safely outlive their parent range
- a range type can be queried to check whether it's borrowed

Ranges facilities are transparent to the compiler – no run-time penalty

Slideware note: `namespace sr = std::ranges`, `namespace sv = std::views`

```cpp
std::vector v{1, 3, 314, 7, 9, 42};
sr::sort(v);
auto even = v | sv::filter([](int i) { return i%2==0; });
std::print("Sorted even elements: {}\n", even);
```

```cpp
std::vector v{1, 3, 314, 7, 9, 42};
sr::sort(v);
auto even = v | sv::filter([](int i) { return i%2==0; });
std::print("Sorted even elements: {}\n", even);
```

```
Sorted even elements: [42, 314]
```

```cpp
std::vector v{1, 3, 314, 7, 9, 42};
sr::sort(v);
auto even = v | sv::filter([](int i) { return i%2==0; });
std::print("Sorted even elements: {}\n", even);
```

```
Sorted even elements: [42, 314]
```

```cpp
auto maxit = sr::max_element(std::vector{1, 2, 3, 4, 5});
std::print("Max element: {}\n", *maxit);
```

```cpp
std::vector v{1, 3, 314, 7, 9, 42};
sr::sort(v);
auto even = v | sv::filter([](int i) { return i%2==0; });
std::print("Sorted even elements: {}\n", even);
```

```
Sorted even elements: [42, 314]
```

```cpp
auto maxit = sr::max_element(std::vector{1, 2, 3, 4, 5});
std::print("Max element: {}\n", *maxit);
```

```
error: no match for 'operator*'
(operand type is 'std::ranges::dangling')
```

Range-enabled MPI wrapper

The arguments of various MPI calls get simplified:

- Communicator $\longrightarrow$ base object
- (pointer, size, type) $\longrightarrow$ range
- status $\longrightarrow$ ignored(?)
- request $\longrightarrow$ returned from call

Range-enabled MPI wrapper

The arguments of various MPI calls get simplified:

- Communicator $\longrightarrow$ base object
- (pointer, size, type) $\longrightarrow$ range
- status $\longrightarrow$ ignored(?)
- request $\longrightarrow$ returned from call

Passing an owning range to non-blocking calls:

- Compiler error, or
- Payload gets bundled with the request

Range-enabled MPI wrapper

The arguments of various MPI calls get simplified:

- Communicator $\longrightarrow$ base object
- (pointer, size, type) $\longrightarrow$ range
- status $\longrightarrow$ ignored(?)
- request $\longrightarrow$ returned from call

Passing an owning range to non-blocking calls:

- Compiler error, or
- Payload gets bundled with the request

Error handling via exceptions or `std::expected`

Example: halo exchange

Inputs:

- `std::vector` of in-neighbors (ranks)
- `std::flat_map` of out-neighbors (ranks) to corresponding message – vector of `double`s

Example: halo exchange

Inputs:

- `std::vector` of in-neighbors (ranks)
- `std::flat_map` of out-neighbors (ranks) to corresponding message – vector of `double`s

Algorithm:

1. Post nonblocking sends
2. Probe for incoming message sizes
3. Gather incoming messages into single `std::vector`

Example: halo exchange

```
std::vector<double> halo(MPI_Comm                                        comm,
                         const std::vector<int>&                         in_nbrs,
                         const std::flat_map<int, std::vector<double>>& out) {
  std::vector<MPI_Request> reqs;
  send(comm, reqs, out);
  const auto in_sizes = probe(comm, in_nbrs);
  auto       retval   = receive(comm, reqs, in_nbrs, in_sizes);
  MPI_Waitall(static_cast<int>(reqs.size()), reqs, MPI_STATUSES_IGNORE);
  return retval;
}
```

Example: halo exchange

```
std::vector<double> halo(MPI_Comm                                    comm,
                         const std::vector<int>&                     in_nbrs,
                         const std::flat_map<int, std::vector<double>>& out) {
  std::vector<MPI_Request> reqs;
  send(comm, reqs, out);
  const auto in_sizes = probe(comm, in_nbrs);
  auto       retval   = receive(comm, reqs, in_nbrs, in_sizes);
  MPI_Waitall(static_cast<int>(reqs.size()), reqs, MPI_STATUSES_IGNORE);
  return retval;
}
```

```
std::vector<double> halo(const MPIComm&                              comm,
                         const std::vector<int>&                     in_nbrs,
                         const std::flat_map<int, std::vector<double>>& out) {
  std::vector<MPIComm::Request> reqs;
  send(comm, reqs, out);
  const auto in_sizes = probe(comm, in_nbrs);
  auto       retval   = receive(comm, reqs, in_nbrs, in_sizes);
  // Waitall(reqs);
  return retval;
}
```

Example: halo exchange

```
1 void send(MPI_Comm                                          comm,
2           std::vector<MPI_Request>&                        reqs,
3           const std::flat_map<int, std::vector<double>>& out) {
4   for(const auto& [id, data] : out) {
5     MPI_Request req;
6     MPI_Isend(data.data(), data.size(), MPI_DOUBLE, id, 0, comm, &req);
7     reqs.push_back(req);
8   }
9 }
```

Example: halo exchange

```
void send(MPI_Comm                                          comm,
          std::vector<MPI_Request>&                         reqs,
          const std::flat_map<int, std::vector<double>>& out) {
  for(const auto& [id, data] : out) {
    MPI_Request req;
    MPI_Isend(data.data(), data.size(), MPI_DOUBLE, id, 0, comm, &req);
    reqs.push_back(req);
  }
}
```

```
void send(const MPIComm&                                    comm,
          std::vector<MPIComm::Request>&                    reqs,
          const std::flat_map<int, std::vector<double>>& out) {
  for(const auto& [id, data] : out) {
    reqs.push_back(comm.Isend(data, id, 0));
  }
}
```

Example: halo exchange

```
void send(MPI_Comm                                          comm,
          std::vector<MPI_Request>&                         reqs,
          const std::flat_map<int, std::vector<double>>& out) {
  for(const auto& [id, data] : out) {
    MPI_Request req;
    MPI_Isend(data.data(), data.size(), MPI_DOUBLE, id, 0, comm, &req);
    reqs.push_back(req);
  }
}
```

```
void send(const MPIComm&                                    comm,
          std::vector<MPIComm::Request>&                    reqs,
          const std::flat_map<int, std::vector<double>>& out) {
  for(const auto& [id, data] : out) {
    reqs.push_back(comm.Isend(data, id, 0));
  }
}
```

```
void send(const MPIComm&                                    comm,
          std::vector<MPIComm::Request>&                    reqs,
          const std::flat_map<int, std::vector<double>>& out) {
  sr::transform(out, std::back_inserter(reqs), [&](const auto& pair){
    return comm.Isend(pair.second, pair.first, 0);
  });
}
```

Example: halo exchange

```
std::vector<int> probe(MPI_Comm                 comm,
                       const std::vector<int>& in_nbrs) {
  std::vector<int> retval;
  for(int id : in_nbrs) {
    MPI_Status stat;
    MPI_Probe(id, 0, comm, &stat);
    MPI_Count sz;
    MPI_Get_elements(&status, MPI_DOUBLE, &sz);
    retval.push_back(sz);
  }
  return retval;
}
```

Example: halo exchange

```
std::vector<int> probe(MPI_Comm                comm,
                       const std::vector<int>& in_nbrs) {
  std::vector<int> retval;
  for(int id : in_nbrs) {
    MPI_Status stat;
    MPI_Probe(id, 0, comm, &stat);
    MPI_Count sz;
    MPI_Get_elements(&status, MPI_DOUBLE, &sz);
    retval.push_back(sz);
  }
  return retval;
}
```

```
std::vector<int> probe(const MPIComm&          comm,
                       const std::vector<int>& in_nbrs) {
  return in_nbrs
    | sv::transform([&](int id) { return comm.probeSize<double>(id, 0); })
    | sr::to<std::vector>;
}
```

Example: halo exchange

```
1  void receive(MPI_Comm                    comm,
2               std::vector<MPI_Request>& reqs,
3               const std::vector<int>&   in_nbrs,
4               const std::vector<int>&   in_sizes) {
5    std::vector<double> retval(std::reduce(in_sizes.begin(), in_sizes.end()));
6    for(int i = 0, offset = 0; i != in_nbrs.size(); ++i) {
7      MPI_Request req;
8      MPI_Irecv(retval.data()+offset, in_sizes[i], MPI_DOUBLE, in_nbrs[i], 0, comm, &req);
9      reqs.push_back(req);
10     offset += sz;
11   }
12   return retval;
13 }
```

Example: halo exchange

```cpp
void receive(MPI_Comm                    comm,
             std::vector<MPI_Request>& reqs,
             const std::vector<int>&   in_nbrs,
             const std::vector<int>&   in_sizes) {
  std::vector<double> retval(std::reduce(in_sizes.begin(), in_sizes.end()));
  for(int i = 0, offset = 0; i != in_nbrs.size(); ++i) {
    MPI_Request req;
    MPI_Irecv(retval.data()+offset, in_sizes[i], MPI_DOUBLE, in_nbrs[i], 0, comm, &req);
    reqs.push_back(req);
    offset += sz;
  }
  return retval;
}
```

```cpp
void receive(const MPIComm&                  comm,
             std::vector<MPIComm::Request>& reqs,
             const std::vector<int>&        in_nbrs,
             const std::vector<int>&        in_sizes) {
  std::vector<double> retval(std::reduce(in_sizes.begin(), in_sizes.end()));
  for(int offset = 0; const auto [id, sz] : sv::zip(in_nbrs, in_sizes)) {
    reqs.push_back(comm.Irecv(retval | sv::drop(offset), id, 0));
    offset += sz;
  }
  return retval;
}
```

Physics injection &
Passing compile-time parameters

Physics injection & compile-time parameters

What we mean:

- Equations
- Boundary conditions
- Simulation structure, if impactful at compile-time
- Dimensions, if they meaningfully affect computation (e.g. stencils)

Physics injection & compile-time parameters

What we mean:

- Equations
- Boundary conditions
- Simulation structure, if impactful at compile-time
- Dimensions, if they meaningfully affect computation (e.g. stencils)

What we don't mean:

- Physical parameters, e.g., viscosity, heat capacity, ...
- Simulation structure, if no compile-time optimization occurs

Physics injection & compile-time parameters

2 distinct approaches:

Physics injection & compile-time parameters

2 distinct approaches:

- Code generation:
  - Flexibility
  - Requires auxiliary tooling
  - Hostile to IDEs
  - e.g.: TCLB

Physics injection & compile-time parameters

2 distinct approaches:

- Code generation:
  - Flexibility
  - Requires auxiliary tooling
  - Hostile to IDEs
  - e.g.: TCLB
- Meta-programming:
  - Self-contained
  - Transparent to the IDE
  - Requires some knowledge of C++ from the user
  - Templates
  - More templates!
  - Unless...?

Parameter structs

We've always had aggregate initialization:

```
struct Params {
  int nx = 1, ny = 1, nz = 1;
  double nu = 1e-3;
  int n_rhs = 1, n_depends = 0;
};
Params p{2, 4, 1, 1e-3, 1, 2};
```

Parameter structs

We've always had aggregate initialization:

```
struct Params {
  int nx = 1, ny = 1, nz = 1;
  double nu = 1e-3;
  int n_rhs = 1, n_depends = 0;
};
Params p{2, 4, 1, 1e-3, 1, 2};
```

But now we have *designated* initialization:

```
Params p{.nx = 2, .ny = 4, .n_depends = 2};
```

- Reader friendly
- No need to touch sensible defaults

Compile-time parameter structs

We can now use any literal class type as a template parameter.

Combined with the following

`std::integral_constant`-esque utility

```
1 template<auto>
2 struct ConstexprWrapper {};
```

Compile-time parameter structs

We can now use any literal class type as a template parameter. Combined with the following `std::integral_constant`-esque utility

```
template<auto>
struct ConstexprWrapper {};
```

it gives us a beautiful, clear, and powerful interface for passing CT parameters:

```
constexpr auto params     = Params {2, 4, 1, 1e-3, 1, 2};
/* 1 */   auto simulation = makeSim(ConstexprWrapper<params>{});
/* 2 */   auto simulation = makeSim(WRAP_CT_PARAM(params));
/* 3 */   auto simulation = makeSim<params>();
```

Compile-time parameters

But we're not limited to simple structs:

```
// u, vort, p, grad T, T
constexpr auto sim_def = ProblemDef{
  defineDomain< 11 >(IDfluid, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
  defineDomain< 11 >(IDsolid, 7, 8, 9, 10)};
constexpr auto dbc_def = ProblemDef{
  defineDomain< 11 >(wall, 7, 8, 9)};
auto sys = makeAlgebraicSystem<sim_def, dbc_def>(mesh /*,...*/);
```

example from L3STER

Physics injection

Since we know how to elegantly pass compile-time parameters, we know how to set the size of operators, the discretization order, etc.

```
constexpr KernelParams diff3_params{.dimension = 3,
                                    .n_equations = 7,
                                    .n_unknowns = 4};
```

example from L3STER

Physics injection

Since we know how to elegantly pass compile-time parameters, we know how to set the size of operators, the discretization order, etc.

```
constexpr KernelParams diff3_params{.dimension = 3,
                                    .n_equations = 7,
                                    .n_unknowns = 4};
```

example from L3STER

The question remains – how do we provide the bodies of these operators?

Physics injection

Since we know how to elegantly pass compile-time parameters, we know how to set the size of operators, the discretization order, etc.

```
constexpr KernelParams diff3_params{.dimension = 3,
                                    .n_equations = 7,
                                    .n_unknowns = 4};
```

example from L3STER

The question remains – how do we provide the bodies of these operators?

Idea: rely on generic lambdas and structured bindings to provide a type-oblivious interface.

Physics injection

```
const auto diffusion3d_kernel = wrapDomainEquationKernel<diff3_params>(
  [](const auto& in, auto& out) {
        auto& [operators, rhs] = out;
        auto& [A0, Ax, Ay, Az] = operators;
        constexpr double k = 1.; // diffusivity
        constexpr double s = 1.; // source

        Ax(0, 1) = -k;
        Ay(0, 2) = -k;
        Az(0, 3) = -k;
        rhs[0]   = s;
        A0(1, 1) = -1.;
        Ax(1, 0) = 1.;
        A0(2, 2) = -1.;
        Ay(2, 0) = 1.;
        A0(3, 3) = -1.;
        Az(3, 0) = 1.;
        Ay(4, 3) = 1.;
        Az(4, 2) = -1.;
        Ax(5, 3) = -1.;
        Az(5, 1) = 1.;
        Ax(6, 2) = 1.;
        Ay(6, 1) = -1.;
  });
sys->assembleProblem(diffusion3d_kernel, std::views::single(solid_id));
```

example from L3STER

Conclusions

- Today's C++ is not the same language you held your nose at 20 years ago

Conclusions

- Today's C++ is not the same language you held your nose at 20 years ago
- It's (relatively) easy to pass compile-time information to an API and to the compiler

Conclusions

- Today's C++ is not the same language you held your nose at 20 years ago
- It's (relatively) easy to pass compile-time information to an API and to the compiler
- Most of the complexity can be hidden from the user

Conclusions

- Today's C++ is not the same language you held your nose at 20 years ago
- It's (relatively) easy to pass compile-time information to an API and to the compiler
- Most of the complexity can be hidden from the user

Discussion

jakub.galecki.dokt@pw.edu.pl