The Shapes of Multi-Dimensional Arrays

Vincent Reverdy

September 17th, 2020

Table of contents

Introduction

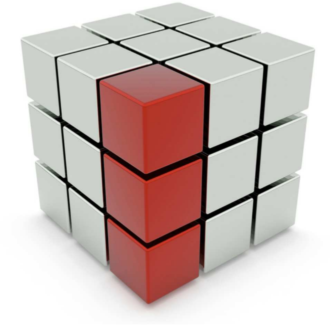
Multidimensional arrays and linear algebra

$$\begin{pmatrix} 0.5 \\ 1.2 \\ 4.1 \end{pmatrix}$$

1D

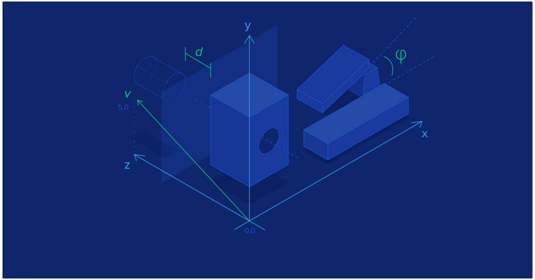
$$\begin{pmatrix} 0.5 & 4.5 & -0.3 \\ 1.2 & -0.9 & 1.4 \\ 4.1 & -3.4 & 0.1 \end{pmatrix}$$

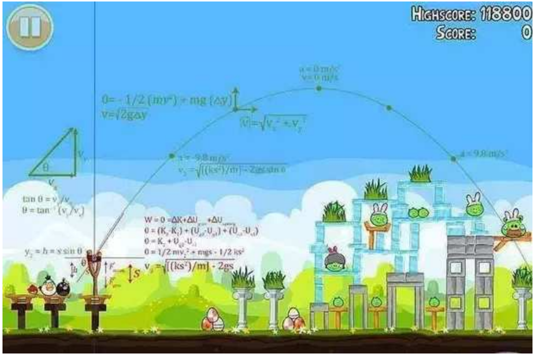
2D



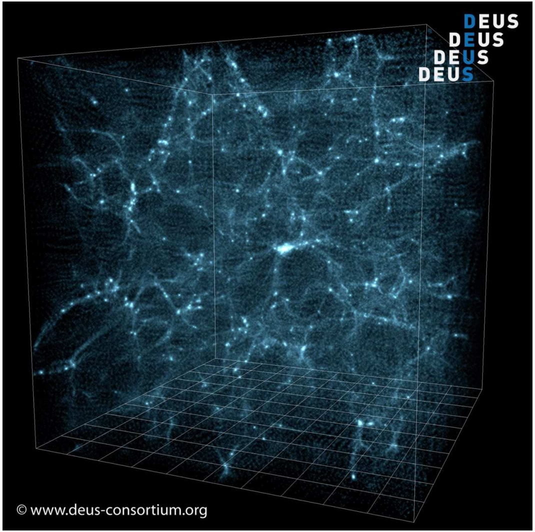
3D

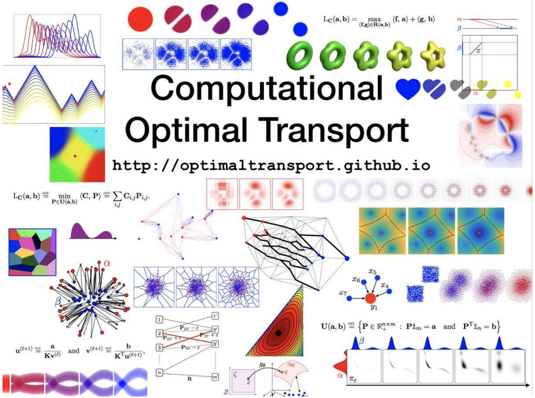
Computer graphics and videogames: small fixed-size vectors and matrices



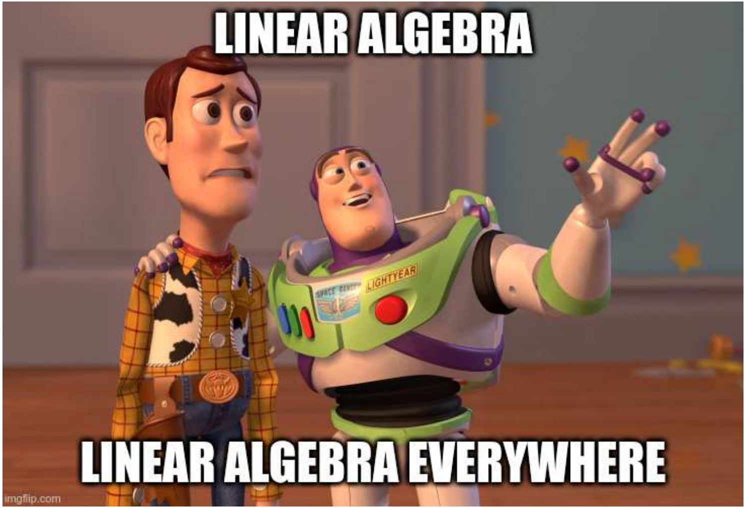


Personal use case with optimal transport: up to 6D dynamic arrays with billions of elements





Summary: multidimensional stuff are useful



Standardization

A standardization effort

On `mdspan`

- P0332: Relaxed Incomplete Multidimensional Array Type Declaration
- P0009: `mdspan`: A Non-Owning Multidimensional Array Reference
- P1684: `mdarray`: An Owning Multidimensional Array Analog of `mdspan`

On linear algebra in general

- P1417: Historical lessons for C++ linear algebra library standardization
- P1166: What do we need from a linear algebra library?
- P1385: A proposal to add linear algebra support to the C++ standard library
- P1635: A Design for an Inter-Operable and Customizable Linear Algebra Library

On BLAS (Basic Linear Algebra Subprograms)

- P1673: A free function linear algebra interface based on the BLAS
- P1674: Evolving a Standard C++ Linear Algebra Library from the BLAS

Miscellaneous

- P1416: Linear Algebra for Data Science and Machine Learning
- P1770: On vectors, tensors, matrices, and hypermatrices

Naming

Originally proposed in P0009 (with array instead of span)

- sci_array
- numeric_array
- vla_array
- multidimensional_array
- multidim_array
- mdarray/md_array
- multiarray/multi_array (multi means something different in multi-containers)

Alternatives

- ndarray/nd_array: because the rank is finite and known at compile-time (used in mathematics and in python)
- hyperarray/hyper_array: because the mathematical generalization of a matrix is called an hypermatrix (same with hypercubes, hyperspheres, . . .)

Layered approach

Standardization by layers

- non-owning references to arrays
- basic algorithms/operations
- owning arrays
- overloaded operators
- geometric aspects
- ...

The mdspan proposal: synopsis

Proposed **mdspan** synopsis as of P0009R10

```
namespace std {
  // [mdspan.extents], class template extents
  template<ptrdiff_t... Extents>
    class extents;

  // [mdspan.layout], Layout mapping policies
  class layout_left;
  class layout_right;
  class layout_stride;

  // [mdspan.accessor.basic]
  template<class ElementType>
    class accessor_basic;

  // [mdspan.basic], class template mdspan
  template<class ElementType, class Extents, class LayoutPolicy = layout_right,
           class AccessorPolicy = accessor_basic<ElementType>>
    class basic_mdspan;

  template<class T, ptrdiff_t... Extents>
    using mdspan = basic_mdspan<T, extents<Extents...>>;

  // ...
}
```

The mdspan proposal: extents

Proposed extents synopsis as of P0009R10

```
template<ptrdiff_t... Extents>
class extents {
public:
  using index_type = ptrdiff_t;

  // [mdspan.extents.cons], Constructors and assignment
  constexpr extents() noexcept = default;
  constexpr extents(const extents&) noexcept = default;
  constexpr extents& operator=(const extents&) noexcept = default;

  template<ptrdiff_t... OtherExtents>
    constexpr extents(const extents<OtherExtents...>&) noexcept;
  template<class... IndexType>
    constexpr extents(IndexType...) noexcept;
  template<class IndexType>
    constexpr extents(const array<IndexType, rank_dynamic()>&) noexcept;
  template<ptrdiff_t... OtherExtents>
    constexpr extents& operator=(const extents<OtherExtents...>&) noexcept;

  // [mdspan.extents.obs], Observers of the domain multidimensional index space
  static constexpr size_t rank() noexcept;
  static constexpr size_t rank_dynamic() noexcept;
  static constexpr index_type static_extent(size_t);
  constexpr index_type extent(size_t) const noexcept;

private:
  static constexpr size_t dynamic_index(size_t) noexcept; // exposition only
  array<index_type, rank_dynamic()> dynamic_extents_{}; // exposition only
};
```

The `mdspan` proposal: `layout`

Proposed `layout_left` synopsis as of P0009R10

```cpp
struct layout_left {
  template<class Extents>
  class mapping {
    using index_type = typename Extents::index_type; // exposition only
  public:
    constexpr mapping() noexcept = default;
    constexpr mapping(const mapping&) noexcept = default;
    constexpr mapping(const Extents&) noexcept;
    template<class OtherExtents> constexpr mapping(const mapping<OtherExtents>&) noexcept;

    constexpr mapping& operator=(const mapping&) noexcept = default;
    template<class OtherExtents> constexpr mapping& operator=(const mapping<OtherExtents>&) noexcept;

    constexpr Extents extents() const noexcept { return extents_; }
    constexpr index_type required_span_size() const noexcept;
    template<class... Indices> index_type operator()(Indices...) const noexcept;
    static constexpr bool is_always_unique() noexcept { return true; }
    static constexpr bool is_always_contiguous() noexcept { return true; }
    static constexpr bool is_always_strided() noexcept { return true; }
    constexpr bool is_unique() const noexcept { return true; }
    constexpr bool is_contiguous() const noexcept { return true; }
    constexpr bool is_strided() const noexcept { return true; }
    index_type stride(size_t) const noexcept;
    template<class OtherExtents> constexpr bool operator==(const mapping<OtherExtents>&) const noexcept;
    template<class OtherExtents> constexpr bool operator!=(const mapping<OtherExtents>& rhs) const noexcept

  private:
    Extents extents_{}; // exposition only
  };
};
```

The mdspan proposal: accessor

Proposed accessor_basic synopsis as of P0009R10

```
template<class ElementType>
  struct accessor_basic {
    using offset_policy = accessor_basic;
    using element_type = ElementType;
    using reference = ElementType&;
    using pointer = ElementType*;

    constexpr typename offset_policy::pointer
      offset(pointer p, ptrdiff_t i) const noexcept;

    constexpr reference access(pointer p, ptrdiff_t i) const noexcept;

    constexpr pointer decay(pointer p) const noexcept;
  };
```


The mdspan proposal: mdspan (1/3)

Proposed mdspan synopsis as of P0009R10

```
template<class ElementType, class Extents, class LayoutPolicy, class AccessorPolicy>
class basic_mdspan {
public:

  // Domain and codomain types
  using extents_type = Extents;
  using layout_type = LayoutPolicy;
  using accessor_type = AccessorPolicy;
  using mapping_type = typename layout_type::template mapping_type<extents_type>;
  using element_type = typename accessor_type::element_type;
  using value_type = remove_cv_t<element_type>;
  using index_type = ptrdiff_t ;
  using difference_type = ptrdiff_t;
  using pointer = typename accessor_type::pointer;
  using reference = typename accessor_type::reference;

  // [mdspan.basic.cons], basic_mdspan constructors, assignment, and destructor
  constexpr basic_mdspan() noexcept = default;
  constexpr basic_mdspan(const basic_mdspan&) noexcept = default;
  constexpr basic_mdspan(basic_mdspan&&) noexcept = default;
  template<class... IndexType>
    explicit constexpr basic_mdspan(pointer p, IndexType... dynamic_extents);
  template<class IndexType, size_t N>
    explicit constexpr basic_mdspan(pointer p, const array<IndexType, N>& dynamic_extents);
  constexpr basic_mdspan(pointer p, const mapping_type& m);
  constexpr basic_mdspan(pointer p, const mapping_type& m, const accessor_type& a);
  template<class OtherElementType, class OtherExtents, class OtherLayoutPolicy, class OtherAccessorPolicy>
    constexpr basic_mdspan(
      const basic_mdspan<OtherElementType, OtherExtents, OtherLayoutPolicy, OtherAccessorPolicy>& other
    );

  // ...
};
```

The mdspan proposal: mdspan (2/3)

Proposed **mdspan** synopsis as of P0009R10

```
template<class ElementType, class Extents, class LayoutPolicy, class AccessorPolicy>
class basic_mdspan {
public:
  // ...
  constexpr basic_mdspan& operator=(const basic_mdspan&) noexcept = default;
  constexpr basic_mdspan& operator=(basic_mdspan&&) noexcept = default;
  template<class OtherElementType, class OtherExtents, class OtherLayoutPolicy, class OtherAccessorPolicy>
    constexpr basic_mdspan& operator=(
      const basic_mdspan<OtherElementType, OtherExtents, OtherLayoutPolicy, OtherAccessorPolicy>& other
    ) noexcept;

  // [mdspan.basic.mapping], basic_mdspan mapping domain multidimensional index to access codomain element
  constexpr reference operator[](index_type) const noexcept;
  template<class... IndexType>
    constexpr reference operator()(IndexType... indices) const noexcept;
  template<class IndexType, size_t N>
    constexpr reference operator()(const array<IndexType, N>& indices) const noexcept;
  // ...
};
```

The mdspan proposal: mdspan (3/3)

Proposed **mdspan** synopsis as of P0009R10

```cpp
template<class ElementType, class Extents, class LayoutPolicy, class AccessorPolicy>
class basic_mdspan {
public:
  // ...
  accessor_type accessor() const;

  static constexpr int rank() noexcept;
  static constexpr int rank_dynamic() noexcept;
  static constexpr index_type static_extent(size_t r) noexcept;

  constexpr Extents extents() const noexcept;
  constexpr index_type extent(size_t r) const noexcept;
  constexpr index_type size() const noexcept;
  constexpr index_type unique_size() const noexcept;

  // [mdspan.basic.codomain], basic_mdspan observers of the codomain
  constexpr span<element_type> span() const noexcept;
  constexpr pointer data() const noexcept;

  static constexpr bool is_always_unique() noexcept;
  static constexpr bool is_always_contiguous() noexcept;
  static constexpr bool is_always_strided() noexcept;

  constexpr mapping_type mapping() const noexcept;
  constexpr bool is_unique() const noexcept;
  constexpr bool is_contiguous() const noexcept;
  constexpr bool is_strided() const noexcept;
  constexpr index_type stride(size_t r) const;
};
```

The `mdspan` proposal: summary

`mdspan` is the product of

- an element type `ElementType`
- an `Extents` to describe its shape
- a `LayoutPolicy` to specify the mapping between the indices and storage
- an `AccessorPolicy` to specify how individual and contiguous elements can be accessed in memory

Simplifications...

- it is non-owning
- it has been designed for contiguous memory storage

The `mdspan` proposal: specifying the shape

Example `mdspan` declaration

```cpp
// As of P0009R10
using myspan = std::mdspan<double, extents<3, std::dynamic_extent, 7>>;

// With P0332: Relaxed Incomplete Multidimensional Array Type Declaration
using myspan = std::mdspan<double[][3][7]>; // already allowed
using myspan = std::mdspan<double[3][][7]>; // currently not allowed
```

Standardization

- The Library Evolution Working Group (LEWG) was largely in favor of `std::mdspan<double[3][][7]>`
- The Evolution Working Group (EWG) was against a language change because of unintended side effects in other contexts

Zero-sized arrays

- *"Its value N specifies the array bound, i.e., the number of elements in the array; N shall be greater than zero."*
- $\Rightarrow$ `double[3][7][0]` not allowed
- However `std::array<double, 0>` is a compiling special case

The shapes of multi-dimensional arrays

What are we looking for?

A way to specify the shapes of multi-dimensional arrays

Goals

- Genericity: cover as much as possible of the parameter space
- Performance: both in terms of computing time and memory
- Expressivity: easy to and understand, read, and write in a concise way

Design

The shapes of multi-dimensional arrays

What are we looking for?

A way to specify the shapes of multi-dimensional arrays

Design goals: the GPE principle

- **G**enericity: cover as much as possible of the parameter space
- **P**erformance: both in terms of computing time and memory
- **E**xpressivity: easy to and understand, read, and write in a concise way

However...

One cannot have everything at the same time

Software design

Software architecture

The art of balancing and compromising between genericity, performance, and expressivity.

Design goals

- **G**enericity: cover as much as possible of the parameter space
- **P**erformance: both in terms of computing time and memory
- **E**xpressivity: easy to and understand, read, and write in a concise way

Design complexity

- Focusing on only one axis drastically simplifies the problem (for example focusing on expressivity in a high level language)
- Focusing on two axes can still lead to reasonable level of complexity (for example performance and expressivity)
- Addressing the three axes at the same time is what can make the design a real challenge

C++

The standard C++ library has to handle the three axes.

$\Rightarrow$ Still no `std::matrix` in 2020. . . .

What do we want (ideally) for a n-dimensional array?

Expressivity (pseudo-code)

```
std::ndarray<double> a;
std::ndarray<double[0]> b;
std::ndarray<double[3][4][5]> c;
std::ndarray<double[3][][5]> d;
std::ndarray<double[3][][5], properties...> d;
```

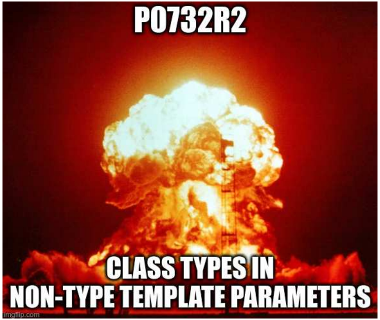
Performance

No computing-time/memory overhead of generic solutions compared to handcrafted solutions designed and optimized for specific scenarios

Genericity

- Arbitrary memory layouts
- Arbitrary indexing schemes
- Arbitrary finite ranks
- Combinations of static and dynamic dimensions
- Zero-sized case

What do we need? (hint: C++20)

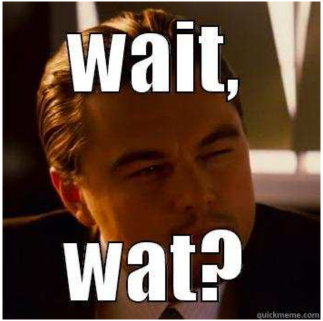


EDSL

What if. . .

. . . the following was perfectly possible in C++20. . .

- `std::ndarray<double, shape[4]()[3][5]> myarray;`
- or even: `std::ndarray<contents<double>[4]()[3][5]> myarray;`



Reverse-engineering from the expressivity goal

With shape:

`std::ndarray<double, shape[4]()[3][5]> myarray;`

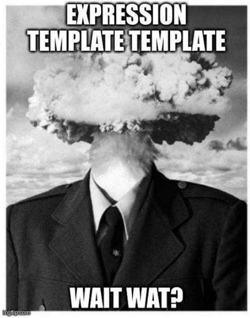
- shape cannot be a type, it has to be a variable
- `std::ndarray` should be a class template of the form `template <class, auto>`
- shape should be a variable of a type with `operator()` and `operator[]`
- `operator()`/`operator[]` should return a value of the same type so that `operator()`/`operator[]` can be called again recursively until all dimensions are specified

With contents:

`std::ndarray<contents<double>[4]()[3][5]> myarray;`

- contents cannot be a type, it has to be a variable template
- `std::ndarray` should be a class template of the form `template <auto>`
- `contents<T>` should be a variable of a type with `operator()` and `operator[]`
- `operator()`/`operator[]` should return a value of the same type so that `operator()`/`operator[]` can be called again recursively until all dimensions are specified

Introducing expression template template



Principle

- `shape` builds an expression template
- The result of the expression template is then injected as a template parameter

Warning

- Takes EDSL (Embedded Domain Specific Languages) to a whole new level

Building the EDSL: tags and shaper constructors

Tagging operators to know what operator has been called

```
// A tag marking the function call operator
template <class... Args> struct function_call_operator {};

// A tag marking the subscripting operator
template <class... Args> struct subscripting_operator {};
```

Shaper class template

```
// The type of a shape remembering the sequence of operations it was made from
template <class... Ops> class shaper {
    // Types
    public:
    using type = shaper<Ops...>;
    using index_sequence = std::index_sequence_for<Ops...>;

    // Lifecycle
    public:
    constexpr shaper() noexcept = default;
    template <class... Args> constexpr shaper(shaper<Args...> other, std::size_t i) noexcept
    : _data(other.to_array(i)) {}

    // Implementation details
    private:
    std::array<std::size_t, sizeof...(Ops)> _data;

    // ...
};
```

Building the EDSL: shaper operations

Shaper class template

```cpp
// The type of a shape remembering the sequence of operations it was made from
template <class... Ops> class shaper {
    // ...
    // Operators
    public:
    constexpr auto operator()() const noexcept {
        return shaper<Ops..., function_call_operator<>>(*this, -1);
    }
    constexpr auto operator()(std::size_t i) const noexcept {
        return shaper<Ops..., function_call_operator<std::size_t>>(*this, i);
    }
    constexpr auto operator[](std::size_t i) const noexcept {
        return shaper<Ops..., subscripting_operator<std::size_t>>(*this, i);
    }
    // Management
    public:
    static constexpr std::size_t rank() noexcept {
        return sizeof...(Ops);
    }
    constexpr std::size_t at(std::size_t i) const noexcept {
        return _data[i];
    }
    template <class... X> constexpr auto to_array(X... x) const noexcept {
        return to_array(index_sequence(), x...);
    }
    template <std::size_t... I, class... X> constexpr auto to_array(std::index_sequence<I...>, X... x) const noexcept {
        return std::array<std::size_t, sizeof...(Ops) + sizeof...(X)>{{std::get<I>(_data)..., x...}};
    }
};

// Default shaper instance
inline constexpr shaper shape;
```

Building the EDSL: indexed dynamic extent

Indexing helper

```
// Index constant alias template
template <std::size_t I> using index_constant = std::integral_constant<std::size_t, I>;

// Index variable template
template <std::size_t I> inline constexpr index_constant<I> index;
```

A dynamic extent

```
// A dynamic extent
template <auto...> struct indexed_dynamic_extent;

// A dynamic extent with no default value
template <std::size_t Index> struct indexed_dynamic_extent<Index> {
    using is_constant = std::false_type;
    constexpr auto operator[](index_constant<Index>) const noexcept {
        return *this;
    }
};

// A dynamic extent with a default value for initialization
template <std::size_t Index, std::size_t Value> struct indexed_dynamic_extent<Index, Value>
: std::integral_constant<std::size_t, Value> {
    using is_constant = std::false_type;
    constexpr auto operator[](index_constant<Index>) const noexcept {
        return *this;
    }
};
```

Building the EDSL: indexed static extent

A static extent

```
// A static extent with its value
template <std::size_t Index, std::size_t Value>
struct indexed_static_extent
: std::integral_constant<std::size_t, Value> {
    using is_constant = std::true_type;
    constexpr auto operator[](index_constant<Index>) const noexcept {
        return *this;
    }
};
```

Building the EDSL: indexed extent maker

Indexed extent maker

```
// A helper to build an indexed extent from a tagged operator
template <class, std::size_t, auto...>
struct indexed_extent_maker;

// Converts the function call operator into a dynamic extent
template <std::size_t I, auto... Args>
struct indexed_extent_maker<function_call_operator<>, I, Args...> {
    using type = indexed_dynamic_extent<I>;
};

// Converts the function call operator into a dynamic extent with default value
template <std::size_t I, std::size_t V, auto... Args>
struct indexed_extent_maker<function_call_operator<std::size_t>, I, V, Args...> {
    using type = indexed_dynamic_extent<I, V>;
};

// Converts the subscripting operator into a static extent
template <std::size_t I, std::size_t V, auto... Args>
struct indexed_extent_maker<subscripting_operator<std::size_t>, I, V, Args...> {
    using type = indexed_static_extent<I, V>;
};

// Alias template to build an indexed extent from a tagged operator
template <class Op, std::size_t I, auto... Args>
using make_indexed_extent = typename indexed_extent_maker<Op, I, Args...>::type;
```

Building the EDSL: extent policy

Extent policy class template

```cpp
// The extents, static and dynamic, of a N-dimensional array
template <class... Extents>
struct extent_policy
: Extents... {
    using index_sequence = std::index_sequence_for<Extents...>;
    using Extents::operator[]...;
    static constexpr std::size_t rank() noexcept {
        return sizeof...(Extents);
    }
};

// Extent policy maker declaration
template <auto, class, class>
struct extent_policy_maker;

// Specialization of extent policy maker for a shape, a shaper, and an index sequence
template <auto S, class... Ops, std::size_t... I>
struct extent_policy_maker<S, shaper<Ops...>, std::index_sequence<I...>> {
    using type = extent_policy<make_indexed_extent<Ops, I, S.at(I)>...>;
};

// Make an extent policy type from a shape variable
template <auto S, class T = decltype(S), class I = typename T::index_sequence>
using make_extent_policy = typename extent_policy_maker<S, typename T::type, I>::type;
```

Building the EDSL: extend

Extend class template

```cpp
// A class to specify the dynamic size along an axis
template <std::size_t Index>
class indexed_extender
{
    // Constants
    public:
    using is_extender = std::true_type;

    // Lifecycle
    public:
    constexpr indexed_extender() noexcept = default;
    explicit constexpr indexed_extender(std::size_t size) noexcept
    : _value(size) {
    }

    // Assignment and access
    public:
    constexpr indexed_extender operator=(std::size_t size) const noexcept {
        return indexed_extender(size);
    }
    constexpr operator std::size_t() const {
        return _value;
    }
    constexpr auto operator[](index_constant<Index>) const noexcept {
        return *this;
    }

    // Implementation details
    private:
    std::size_t _value;
};
```

Building the EDSL: extenders

Extenders class template

```
// A class grouping extenders
template <class...>
struct extenders;

// A class grouping extenders specialized for extenders
template <std::size_t... I>
struct extenders<indexed_extender<I>...>
: indexed_extender<I>... {
    using indexed_extender<I>::operator[]...;
    explicit constexpr extenders(indexed_extender<I>... i) noexcept
    : indexed_extender<I>(i)... {
    }
};

// Extenders deduction guide
template <std::size_t... I>
extenders(indexed_extender<I>... i) -> extenders<indexed_extender<I>...>;

// Extend variable template
template <std::size_t I>
inline constexpr indexed_extender<I> extend(-1);
```

Building the EDSL: ndarray

Ndarray class template

```
// Generic ndarray declaration
template <class...>
class basic_ndarray;

// Simplified ndarray definition
template <class Type, class ExtentPolicy>
class basic_ndarray<Type, ExtentPolicy>
{
    // Types and constants
    public:
    using value_type = Type;
    using extent_policy = ExtentPolicy;

    // Lifecycle
    public:
    template <class... Extenders>
    explicit constexpr basic_ndarray(Extenders... e) noexcept;

    // Access
    public:
    constexpr extents<extent_policy::rank()> shape();

    // ...
};

// Ndarray alias template
template <class Type, auto Shape, class... Args>
using ndarray = basic_ndarray<Type, make_extent_policy<Shape>, Args...>;
```


Illustration

Usage in practice

```
int main(int argc, char* argv[])
{
    // A ndarray of:
    // Dimension-0: a static size of 3
    // Dimension-1: a dynamic size whose initial value is given in constructor
    // Dimension-2: a static size of 4
    // Dimension-3: a dynamic size with a default initial value of 2
    // Dimension-4: a static size of 3
    // And then the constructor initializes the dimension 1 to size 5
    ndarray<int, shape[3]()[4](2)[3]> myarray(extend<1> = 5);
    return 0;
}
```

Extents

The problem with extents

Static only

When all dimensions are static, extents can just be an integer sequence ⇒ `sizeof(extents)` should be the size of an empty struct.

Dynamic only

When all dimensions are dynamic, extents can just be a runtime array of size `rank` ⇒ `sizeof(extents)` = `sizeof(std::size_t)` × `rank`.

Mixed static/dynamic

Well, it's a little bit more complicated...

Optimization goal

- Memory footprint of only dynamic dimensions
- Optimal runtime access of a particular dimension
- Bonus: optimized compilation time

Challenges

Access

- `extents[3]`: dynamic access to the extent along axis 3
- `extents[n]`: dynamic access to the extent along axis n (runtime variable)
- `extents[index<3>]`: static access to the extent along axis 3
- `extents[index<N>]`: static access to the extent along axis N (constant expression)

Challenges

- `extents[n]`: avoiding $\mathcal{O}(n)$ scaling for static dimensions
- `extents[index<N>]`: avoiding $\mathcal{O}(N)$ scaling in compilation-time

Log-tuple trick: introduction

Straightforward approach

`get<N>(tuple)` has to iterate over the first `N` types.

Advanced approach

There is a way to exploit overload resolution to have $\mathcal{O}(\log(N))$ compile-time access.

Indexing

```
// Index constant type
template <std::size_t I>
struct index_constant: std::integral_constant<std::size_t, I> {};

// Index constant variable template
template <std::size_t I>
inline constexpr index_constant<I> index = {};
```

Log-tuple trick: elements

Element wrappers

```
// A basic element wrapper
template <class T>
struct tuple_element_wrapper {
    constexpr tuple_element_wrapper(const T& x): value(x) {}
    // Other constructors to be defined
    T value;
};

// An indexed tuple element
template <std::size_t I, class T>
struct tuple_element: tuple_element_wrapper<T> {
    constexpr tuple_element(const T& x): tuple_element_wrapper<T>(x) {}
    constexpr T& operator[](index_constant<I>) {
        return static_cast<wrapper<T>&>(*this).value;
    }
    constexpr const T& operator[](index_constant<I>) const {
        return static_cast<const wrapper<T>&>(*this).value;
    }
};
```

Log-tuple trick: tuple

Tuple

```cpp
// Base class declaration
template <class Sequence, class... T>
struct tuple_base;

// Base class specialization for index sequence
template <std::size_t... I, class... T>
struct tuple_base<std::index_sequence<I...>, T...>
: tuple_element<I, T>... {
    using index_sequence = std::index_sequence<I...>;
    using tuple_element<I, T>::operator[]...;
    constexpr tuple_base(const T&... x): tuple_element<I, T>(x)... {}
    // Other constructors to be defined
};

// Actual tuple implementation
template <class... T>
struct tuple: tuple_base<std::index_sequence_for<T...>, T...> {
    using base = tuple_base<std::index_sequence_for<T...>, T...>;
    using base::base;
    using base::operator[];
};
template <class... T>
tuple(const T&...) -> tuple<T...>;
```

Result

`mytuple[index<3>]` leverages overload resolution to access the element at compile-time.

Runtime element access of a static sequence: introduction

The problem

Considers a tuple $\mathcal{T}$ of types `Types...` (where, for instance each type can be either a static or dynamic extent). How to design a runtime subscript operator for this tuple, so that when provided with an index i and a generic lambda Λ, it would return the result of the lambda applied to the i-th element of the tuple: $\Lambda(\mathcal{T}_i)$?

The trivial approach

```
i == 0 ? lambda(std::get<0>(tuple)) :
  i == 1 ? lambda(std::get<1>(tuple)) :
    i == 2 ? lambda(std::get<2>(tuple)) :
      i == 3 ? lambda(std::get<3>(tuple)) :
        // ...
          lambda(std::get<N - 1>(tuple));
```

The trivial non-branching approach

```
(i == 0) * lambda(std::get<0>(tuple)) +
  (i == 1) * lambda(std::get<1>(tuple)) +
    (i == 2) * lambda(std::get<2>(tuple)) +
      (i == 3) * lambda(std::get<3>(tuple)) +
        // ...
          (i == N - 1) * lambda(std::get<N - 1>(tuple));
```

Runtime element access of a static sequence: complexity

Access complexity

The preceding strategies are $\mathcal{O}(N)$-access strategies. Can we do better?

Dichotomic approach (for $\mathrm{N} = 6$)

```
i < 3
  ? i < 2
    ? i == 0
      ? lambda(std::get<0>(tuple))
      : lambda(std::get<1>(tuple))
    : lambda(std::get<2>(tuple))
  : i < 5
    ? i == 3
      ? lambda(std::get<3>(tuple))
      : lambda(std::get<4>(tuple))
    : lambda(std::get<5>(tuple));
```

Access complexity

The preceding strategy scales as $\mathcal{O}(\log(N))$.

Runtime element access of a static sequence: variadic implementation

Expressivity goal

`overload_sequence(F...)[i](Args...)` should return the result of the i-th function `F` on the arguments `Args...`.

Starting with the log-tuple trick

```
// Index constant helper
template <std::size_t I>
struct index_constant: std::integral_constant<std::size_t, I> {};

// Index constant variable template
template <std::size_t I>
inline constexpr index_constant<I> index = {};

// Element wrapper
template <std::size_t I, class F>
struct overload_sequence_element: F {
    static constexpr index_type<I> index = {};
    constexpr overload_sequence_element(const F& f): F(f) {}
    constexpr F& operator[](index_constant<I>) {return static_cast<F&>(*this);}
    constexpr const F& operator[](index_constant<I>) const {return static_cast<const F&>(*this);}
};
```

Runtime element access of a static sequence: overload sequence

Overload sequence implementation

```cpp
// Overload sequence base declaration
template <class...>
struct overload_sequence_base;

// Overload sequence base definition
template <std::size_t... I, class... F>
struct overload_sequence_base<std::index_sequence<I...>, F...>
: overload_sequence_element<I, F>... {
    using index_sequence = std::index_sequence<I...>;
    using overload_sequence_element<I, F>::operator[]...;
    constexpr overload_sequence_base(const F&... args)
    : overload_sequence_element<I, F>(args)... {}
    static constexpr std::size_t size() {return sizeof...(F);}
};

// Overload sequence
template <class... F>
struct overload_sequence:
overload_sequence_base<std::index_sequence_for<F...>, F...> {
    using base = overload_sequence_base<std::index_sequence_for<F...>, F...>;
    using base::base;
    using base::operator[];
    constexpr overload_sequence_selector<overload_sequence&> operator[](std::size_t i) {
        return overload_sequence_selector<overload_sequence&>(*this, i);
    }
    constexpr overload_sequence_selector<overload_sequence&> operator[](std::size_t i) const {
        return overload_sequence_selector<const overload_sequence&>(*this, i);
    }
};
template <class... F>
overload_sequence(const F&...) -> overload_sequence<F...>;
```

Runtime element access of a static sequence: dichotomic access

Overload sequence selector implementation

```cpp
// Dichotomic access implementation
template <class Sequence>
struct overload_sequence_selector {
    constexpr overload_sequence_selector(Sequence seq, std::size_t i): sequence(seq), index(i) {}
    template <class... Args>
    constexpr void operator()(Args&&... args) {
        this->operator()<0, std::decay_t<Sequence>::size() - 1>(std::forward<Args>(args)...);
    }
    template <int Min, int Max, int Mid = (Min + Max) / 2, class... Args>
    constexpr void operator()(Args&&... args) {
        if constexpr (Min < Max) {
            if (index < Mid) {this->operator()<Min, Mid - 1>(std::forward<Args>(args)...);}
            else if (index > Mid) {this->operator()<Mid + 1, Max>(std::forward<Args>(args)...);}
            else {this->operator()<Mid, Mid>(std::forward<Args>(args)...);}
        } else {
            sequence[index_type<Mid>{}](std::forward<Args>(args)...);
        }
    }
    Sequence sequence;
    std::size_t index;
};
```

Runtime element access of a static sequence: result

Result

For a family of lambdas indexed by i and each associated with an element $\mathcal{T}_i$ of a tuple $\mathcal{T}$, `overload_sequence` provides a runtime generic access in $\mathcal{O}(\log(N))$.

Application

For an heterogeneous sequence of `dynamic_extent` and `static_extent` it provides a $\mathcal{O}(\log(N))$ runtime access with branching.

Beyond logarithmic complexity

Overload sequence selector implementation

```
// Dichotomic access implementation
template <class... Extents>
struct extents {
    static constexpr std::array<bool, sizeof...(Extents)> is_static = {Extents::is_static...};
    static constexpr std::array<std::size_t, sizeof...(Extents)> table = // see after;
    std::array<std::size_t, dynamic_rank> storage = // see after;
    constexpr std::size_t operator[](std::size_t i) {
        return is_static ? table[i] : storage[table[i]];
    }
};
```

Where:

- `storage` contains the dynamic extents known at runtime
- `table` contains:
  - `static_extent::value` if the i-th extent is static
  - the index `j` where the element is stored in `storage` if the i-th extent is dynamic

Access complexity

For an heterogeneous sequence of `dynamic_extent` and `static_extent` it provides a $\mathcal{O}(1)$ runtime access with a branching and a level of indirection.

Optimizing extents

Summary

- Optimizing dynamic-only extents is easy
- Optimizing static-only extents is easy
- Optimizing hybrid extents is tricky

Axes of optimization

- In space: storing only dynamic extents at runtime
- In time: using `overload_sequence` or indirection to optimize runtime access
- In compilation time: using and reusing the log-tuple technique

`mdspan`

The current implementation of `extents.extent(i)` seems to scale as $\mathcal{O}(N)$ (to be confirmed).

Going beyond

Enriching the EDSL: sky is the limit

Layout information

```
std::ndarray<double, +shape[4]()[3][5]>
std::ndarray<double, -shape[4]()[3][5]>
std::ndarray<double, !shape[4]()[3][5]>
```

Symmetries

```
std::ndarray<double, shape[diagonal(4)]()[3][5]>
std::ndarray<double, shape[4]()[diagonal(3)][5]>
std::ndarray<double, shape<triangular>[4]()[3][5]>
```

Axis-based parallelization information

```
std::ndarray<double, shape[distributed(4)]()[3][5]>
std::ndarray<double, shape[distributed(4)]()[3][vectorized(5)]>
std::ndarray<double, shape[4]()[3][vectorized<gpu>(5)]>
```

Operations on shapes

```
std::ndarray<double, shape[4] * shape() * shape[5]>
```

With multidimensional subscript operator

P2128

- P2128: Multidimensional subscript operator
- `operator[](Args&&...)`

Evolution

- `shape[4]()[3][5]` ⇒ `shape[4][][3][5]` (requires zero-parameter operator)
- Other opportunities to enrich the mini-language by combining `operator()` and `operator[]` for different meanings

Towards a generic extent concept

Possible components

- `min`: the minimum allowed size under which it cannot shrink
- `initial`: the initial size at construction
- `threshold`: the maximum number of elements stored without external allocation
- `max`: the maximum allowed size over which is cannot be extended (can be infinite)

Concrete examples

- `static_extent` ⇒ `min == initial == threshold == max`
- `dynamic_extent` ⇒ `min == 0, max == infinity, initial == 0, threshold == 0`
- `fixed_extent` ⇒ `min == 0, max == n, initial == 0, threshold == n`

Generic NTTPs opened a Pandora's box

Metalanguages

- Being able to inject objects as template parameters in C++20 is a game changer for Embedded Domain Specific Languages
- Class templates can now be seen as true mini-compilers: `template <auto> class edsl_compiler ;`

Kind genericity

- The next revolution to come is probably the genericity regarding kinds
- Currently no way to specify universal template parameters (values, types, templates. . .)
- P1985: Universal template parameters
- `template <template auto X>` where `X` can be a value, a type, a template. . .

Conclusion

Summary

Summary

- Linear algebra is everywhere
- Very complex when trying to balance between Genericity, Performance, and Expressivity
- Existing effort with `mdspan`
- C++20 with arbitrary NTTPs allows expressive ways to specify multidimensional shapes
- Expression template template to create mini-languages
- Far more powerful than relaxed incomplete multidimensional array type declaration
- Hybrid dynamic and static extent particularly tricky to implement
- Log-tuple trick useful in many situations
- Extending the EDSL: layout information, symmetries, axis-based parallelization, shape operations
- Conceptifying a generalized extent
- Class NTTPs opened a Pandora's box for EDSL
- Kind genericity will take EDSLs even further

Thank you for your attention

Any question?

Acknowledgments

This work has been made possible thanks to PSL Research University. The author would also like to thank Joël Falcou from the Laboratoire de Recherche en Informatique for the fruitful collaboration on this topic.

