Euclid

A C++ class template library providing orientational analysis and Euclidean vectors.

Andreas Harnack
(ah8 at freenet dot de)

Version 0.0.4

January 2009

Contents

1 Introduction **1**

2 Dimensional Analysis **3**
2.1 Physical Quantities . . . 3
2.2 Scales . . . 4
2.3 Euclidean Vectors . . . 4
2.4 Imaginary Number and 1 . . . 5
2.5 Physical Units reviewed . . . 5
2.6 Conclusion . . . 5
2.7 Naming Conventions . . . 5

3 Euclidean Vectors **7**

4 Implementation **8**
4.1 Vector Operations . . . 10
4.2 Scalar and Cross Product . . . 10
4.3 Constructors . . . 10
4.4 Base Type Access . . . 11
4.5 Component Access . . . 11
4.6 `fold`, `map` and `join` and Friends . . . 13
4.7 Norms . . . 13
4.8 The Comma Operator . . . 14
4.9 Diagonal Matrices and Matrices . . . 16
4.10 Exception Safety . . . 17
4.11 I/O . . . 17
4.12 File Layout . . . 17
4.13 Usage . . . 19
4.14 Euclidean Vectors and Physical Dimensions . . . 19

5 Dimensional Analysis and Matrices **21**
5.1 Space Complexity . . . 23
5.2 Conclusion . . . 23

List of Figures

Figure 1: An Euclidean vector template class, basic structure 9
Figure 2: An application example 12
Figure 3: The access operator 12

Chapter 1

Introduction

Dimensional analysis has been a topic for quite a while in the C++ community and is still discussed with interest. The need to integrate dimensional analysis into the C++ type system to support dimensional consistency checks during compile time seems to be widely accepted. However, so far the idea of dimensional analysis has been reduced to physical dimensions, while other aspects apparently are being ignored.

The library introduced here implements a set of class templates providing orientational analysis, i.e. consistency checks for values with a dimension attached to it indicating a direction in a plane or in space. We shall refer to such values as *coordinates*. Describing objects in terms of coordinates is a first step towards the world of Euclidean geometry and Euclidean vector spaces. Hence, it seems only natural to extend the concept of dimensions by providing additional templates to aggregate coordinate values into Euclidean vectors. The presented library does covers that as well.

Accepting coordinate values as being type distinct has some fundamental implications to the design of a vector class. Traditionally a vector is considered as being an array-like data structure, i.e. an aggregation of a possibly varying number of objects of the same type. That doesn't fit any more. Now the components have mutual distinct types. Consequently access operators can't be defined in the traditional way: their return type would depend on the index. The same applies to iterators accordingly.

An Euclidean vector space has a specific dimension, i.e. all members of a vector space have the same specific number of components. Neither insertion nor deletion or reordering do make any sense. Hence, Euclidean vectors are much better represented in a structure-like form.

Methods of template programming offer a way to implement such structures for an arbitrary yet fixed number of components. Component access is provided by type conversion, made possible by the unique type of each component.

The library presented here is not intended to be a Linear Algebra library. Although it deals with objects of Linear Algebra, it does not provide any

non-trivial algorithms. It only provides orientational analysis and type safety for those, who wish to write or use algorithms concerning algebraic problem.

Chapter 2

Dimensional Analysis

Let's think of a *quantity* as something that can be expressed in the form $m \cdot d$, where m shall be called a magnitude and d a dimension. The magnitude is some kind of numerical value (in the broadest sense). The dimension represents concepts including but not limited to physical units, scales, unit vectors and so on. Even the imaginary number does fit into that scheme.

Assuming we can add two magnitudes and multiply a magnitude by a scalar, then we can formally deduce the operation allowed over quantities as:

$$m_1 \cdot d + m_2 \cdot d = (m_1 + m_2) \cdot d$$
$$c \cdot (m \cdot d) = (c \cdot m) \cdot d$$

For the record: these are vector operations, so quantities of the same dimension form a vector space.

Multiplication of two quantities is given by:

$$(m_1 \cdot d_1) \cdot (m_2 \cdot d_2) = (m_1 \cdot m_2) \cdot (d_1 \cdot d_2)$$

Its meaning – if any – depends on the meaning of the multiplication defined for the dimensions involved and shall be discussed below.

2.1 Physical Quantities

For physical quantities the dimension is actually a unit[1]. Units can be divided and multiplied to form new units. In a system of units a combination of base units, raised to some powers, can be used to express all other units. For rational powers this opens the opportunity to express base units as prime numbers. Since any non-negative natural number can be split unambiguously into a product of prime factors, each fractional number represents a unit where the exponents

[1] In physics the term dimension has a different meaning. A physical dimension describes only the quality of an object, while a unit provides both, a qualitative as well as a quantitative reference.

of the prime factors correspond to the exponents of the respective base units. Hence, multiplication of physical units can be reduced to the multiplication of fractional numbers. This should be reasonably flexible; there's an infinitely high number of prime numbers, so there's also a potentially unlimited set of base dimensions. The only remaining problem is the normalization of fractional numbers, i.e. removing all common factors from nominator and denominator. The Euclidean algorithm does just that. It has already been demonstrated to be executable at compile time by template programming techniques.

It would be even more charming to use logarithms of prime numbers instead of the numbers themselves. In that case dimensions only needed to be add and subtracted without the need of normalization. But since logarithms are irregular real numbers this is rather certain to cause numerical problem, unless we could find a suitable encoding to get around these problems. I'm, however, not enough of a mathematician to address this one.

2.2 Scales

Dimensions can be used to represent scaling factors. In that case multiplying dimensions is just a common numerical multiplication. One special case is to interpret magnitude and dimension of a quantity as mantissa and exponent, herein the multiplication is carried out by adding the (logarithmic) exponents. Another option is to interpret magnitude and dimension as nominator and denominator of a fraction. This is a way to describe fixed point arithmetic. In that case multiplication would have the special form of:

$$(m_1 \cdot d) \cdot (m_2 \cdot d) = ((m_1 \cdot m_2)/d) \cdot d$$

About three decades ago the programming language FORTH was quite successful using that kind of arithmetic to get along without floating point arithmetic, which was quite inefficient on the hardware of the time. Maybe, if put into a separate library, it's still an interesting option for performance critical applications or to minimize rounding errors.

2.3 Euclidean Vectors

Dimensions can be unit vectors. In that case they shall be called orientational dimensions. That is the topic the presented library addresses. I've been using orientational dimensions for quite a while in projects involving graphical programming to protect x and y coordinates from being mixed up. This significantly reduced the error rate. This interpretation leads to a new kind of a vector type. We'll discuss the details in a moment.

Physical quantities and orientational dimensions are orthogonal concepts and therefore best go into different libraries, but they are linked to each other and can boost each other when used together. A physical quantity library should be able to deal with vectors as well as with scalars. This is essential for avoiding

things like the torque/energy ambiguity, a problem that can't be addressed by physical dimensions alone. The distinction between vector types and scalar types makes it trivial: the former is a vector, the later a scalar.

2.4 Imaginary Number and 1

To complete the picture it should be mentioned that the imaginary number is also a dimension with the special definition of $d_i \cdot d_i = -1$ while all dimensionless numerical values can be interpreted as having the dimension 1. That makes the complex numbers a special case of a some kind of generalized Euclidean vector, so `std::comlpex.h` is a good point to start implementing a dimensional analysis libraries.

2.5 Physical Units reviewed

In practice physical units often appear as a combination of a unit and a prefix. Such prefixes are scales. A library providing dimensional analysis for physical quantities should probably not be concerned with scaling. Many applications will do without explicit scaling if the base units are chosen appropriately. The specification of a set of base units is application specific and should consequently go into a separate library (or at least a separate module). The SI is an obvious candidate for a default option. Alternative option could be offered if well defined and general enough, like a unit system for astrophysics, a monetary systems, an imperial one and so on. If in doubt, users can always choose to define their own system.

2.6 Conclusion

Quantities are a quite general concept coming in different forms and shapes. To isolate different aspects into orthogonal concepts might be a serious problem, which possibly explains the length of the ongoing discussions. The following text focuses on orientational analysis only. Regarding the variety of aspects, an adequate handling of physical units is likely to require a set of libraries, that can be combined freely to fit specific application problems.

2.7 Naming Conventions

The library presented here deals with orientational analysis only. It has been developed before their relations to other aspects of dimensional analysis became apparent. Therefore, the usage of names and terms is somewhat historical and does not necessarily follow the standards as in place today and as defined above. Though the objects it deals with are quantities – with a dimension attached to it representing an orientation in space – the term 'quantity' itself is never

actually used. Instead they're called coordinates or simply dimensions, referring to the casual use of the term to indicate a direction in space. Both terms are used synonymously, though the later one contradicts with the more general use suggested above. Should the code ever make it into a general purpose library then the naming schemes will certainly be a candidate for some clean-up.

Chapter 3

Euclidean Vectors

Euclidean vector spaces can be used to describe objects in a plane or in space. Each member of such vector space can be written as the following sum:

$$x = x_1 e_1 + x_2 e_2 + \ldots + x_n e_n = \sum_{i=1}^{n} x_i e_i$$

The summands shall be called coordinates and are quantities as defined above, with the dimensions being a system of linear independent – typically orthogonal – unit vectors. As quantities they are potentially subject to dimensional analysis, specifically orientational analysis. A programming system supporting dimensional analysis will assign different types to values of different dimensions, so the plausibility of dimensional terms can be checked by static type checking during compile time.

Accepting coordinate values as being type distinct leads to a quite different understanding of a vector than traditionally found in programming. Most importantly, an Euclidean vector is not a container. Containers are array-like data structure, i.e. an aggregation of a possibly varying number of objects of the same type. Now the components have mutual distinct types. Consequently it is impossible to define access operators the traditional way: their return type would depend on the index. A similar problem occurs for iterators. On the other hand do Euclidean vectors have a specific dimension, i.e. a specific number of components. Typically neither insertion nor deletion do make any sense; if taken as a sum as above, not even the order of components matters. An aggregation of a fixed number of differently typed objects without any explicit attention to their order has all criteria typical for a structure, hence Euclidean vectors are much better represented in a structure-like form.

Each single coordinate can be interpreted as a vector with all other components set to zero. This view is supported whenever possible, though occasionally there might be conflicts with the component access scheme.

Chapter 4

Implementation

As discussed above, Euclidean vectors are best represented in a structure-like form. Methods of template programming offer a way to implement such structures for an arbitrary yet fixed number of components. The fundamental structure of the class templates defined is shown in Figure 1. For lack of a better name, all templates are put into a name space `euclid`.

The dimension class template is nothing but a wrapper around a base type, associating it with a dimension tag and forwarding the allowed operations. For orientational analysis, type tags are only compared for equality, so any type or non-type template parameter would do. The use of an integral type, however, is dictated by the vector class template. Vectors are defined recursively, so their dimension parameter needs to be countable. The indexing scheme has been chosen similar to arrays, so a two-dimensional vector contains components of dimension 0 and 1. Consequently a vector of dimension 0 doesn't exist (not even a null vector).

Neither iterators nor container-like access operators can be defined for aggregations of mutually type distinct objects. Type conversion is used instead to gain access to the components of a vector. This type conversion is one of the few, that is allowed to be applied implicitly, which offer the interesting option to allow vectors to appear where a coordinate is required. The type system will pick the right component automatically. Also, this access method is robust with respect to range violations; range violation will be detected at compile time.

Note that the code does neither contain nor will generate any loops into the target code. All recursively called vector operations are expanded at compile time, leading to a sequence of component-wise operations. This is not just time-efficient but, as long as the vector dimensions are reasonably small, even saves space on most architectures.

```
name space euclid
{
        // dimension class template
        template<typename T, unsigned int D> class dim
        {
                        T v;
                public:
                        // operations allowed for dim ...
        };

        // vector class template
        template<typename T, unsigned int D> class vec
        {
                        dim<T, D-1> d;
                        vec<T, D-1> v;
                public:
                        // element access
                        template <unsigned int I>
                        operator dim<T,I>() const { return dim<T, I>(v); }
                        operator dim<T,D-1>() const { return d; }

                        // operations allowed for vec ...
        };

        // vector class base case specialization
        template<typename T> class vec<T,1>
        {
                        dim<T,0> d;
                public:
                        // element access
                        operator dim<T,0>() const { return d; }

                        // operations allowed for vec ...
        };

        // never defined
        template<typename T> class vec<T,0>;
};
```

Figure 1: An Euclidean vector template class, basic structure

4.1 Vector Operations

The operations allowed for vectors and dimensions are primarily those typically defined by vector algebra, i.e. adding and comparing two values of the same dimension and multiplying a dimension with a scalar. The implementation only checks for correct dimensions, it does not put additional restrictions on base types; operations on objects with dimensionally correct types are legal as long as the corresponding operation for the underlying base type is legal. Dimensionally correctly typed objects shall be called *compatible*.

Arithmetical vector operations come in two flavours: as assignment operators and as binary operators. The former are implemented as members, the later as global functions. Whenever possible, binary operators are implemented in the canonical form. They come in two versions, selected by a compile time flag: In the default version the return type is (typically) determined by the right hand side's operand. This is simple, portable and should be sufficient for most situations. In the advanced version the return type is deduced from both operands, using type traits currently relying on the GNU `typeof` operator, so this version is likely to be far less portable.

Additionally, the dimension template implements the pre- and postfix increment and decrement operators as member functions.

4.2 Scalar and Cross Product

Vectors and dimensions implement a scalar product taking two objects of compatible types and returning a scalar.

Two- and three-dimensional vectors also define a cross product. Most textbooks will claim, that the cross product is defined for three-dimensional vectors only. This is not quite correct. The concept of a cross product can be extended to vector spaces of any dimension, but only for a three-dimensional space this is a binary operation evaluating to a vector of the same space. In all other cases there will be a different number of operands or the result will not be a vector. (The two-dimensional case resulting in a two-dimensional vector is an unary operation.) The reason, however, that no general solution is offered is given by the fact, that the complexity increases rapidly with the number of components. Since all operations are inlined without generating any loops, this would be prone to blow up the code.

Both, scalar and cross products come with or without type deduction using traits.

4.3 Constructors

Vectors and dimensions implement a default constructor, a copy constructor and an explicit constructor initializing them with a base type value. For vectors, all components are initialized to the same value. Calling the initialization constructors is the only way to inject scalar values into dimensions and vectors.

To not undermine the type safety only a few selected constructors are allowed to be called implicitly. These are the constructors to convert compatible vectors and dimensions respectively into each other. Converting a dimension into a vector assumes that a all other components are set to zero, converting a vector to a dimension picks the correct component. Though this is in accordance with the assumptions made above and allows for some syntactical sugar, I'm not entirely sure that this will not open the door for some hidden traps, so this feature might be going to change in the future.

For vectors there's finally a constructor to compose a vector inductively. It takes a vector of the next lower dimension and either the correct top level dimension or a scalar value. The latter is allowed since there are no ambiguities. This constructor is intended to implement functions that need to construct a result vector recursively.

4.4 Base Type Access

Occasionally it's unavoidable to access the base type value of a dimension; for an example please consider the subscription operator in Figure 2. I prefer to use the function call operator as element access function whenever possible. Access to numerical values is often needed in more or less complex expressions. Readability will suffer if expressions are getting too long, especially if they have to be broken into several lines. Having an operator instead of a named function has the advantage of being short and so often helps to improve readability. There is a choice of other operators for this purpose, but I feel that the function call fits best. After all, an access function is supposed to do exactly what a function does: returning a value.

Vectors offer element access through two named access functions: `head()` and `tail()`. These are, together with the composite constructors, intended to offer a way of extending the predefined set of functions and operators. New functions can be written by recursively decompose and compose the operands and return values respectively. End users just applying vectors and vector arithmetic are unlikely to ever need to call this functions directly.

4.5 Component Access

As outlined above, it is impossible to define a value dependant access operator for aggregations of type distinct objects. Instead, retrieving a vector component is done by explicit or implicit conversion to the dimension type of the component to be retrieved. Setting a single component can be achieved by explicitly or implicitly converting the dimension to a vector during initialization or by applying the assignment operator.

There is a version of the assignment operator that takes a dimension as an argument and assigns it to the corresponding component. This is quite intuitive though it violates the assumption, that a single dimension is a vector with all

```
template <class Char, class T = int> class Screen
{
        public:
                typedef typename euclid::dim<T,0> dimX;
                typedef typename euclid::dim<T,1> dimY;
                typedef typename euclid::vec<T,2> vect;

        private:
                vect sz;
                Char *img;

        public:
                dimX columns() const { return dimX(sz); }
                dimY lines() const { return dimY(sz); }
                vect size() const { return sz; }

                Char& operator[](vect pos) {
                        return img[dimX(sz)()*dimY(pos)() + dimX(pos)()]; }
};
```

Figure 2: An application example

other components set to zero. For that assumption to hold, the remaining components of the target vector should be cleared. That is somewhat unfortunate but still feels better than other operators I could think of.

As a short cut, the additive operators – both, in the assignment and in the binary form – are defined to accept dimensions as the right hand's side, avoiding the need to create a vector first.

All methods to access vector components described so far only provide access by value. Sometimes life would be simpler by having reference access instead. For such situations an experimental subscription operator is provided, taking a dimension as an argument. Selective, however, is not the value, but the type of the argument, the value is actually being ignored.

```
void foo(vect& size) { size[dimX()] *= 2; }
```

Figure 3: The access operator

A temporary object can be created wherever the type is needed, as in Figure 3. The compiler should be able to optimize it away.

For convenience, there is also a non-member `at` function. This can be instantiated directly, without having to create a temporary explicitly.[1]

[1] Internally it still creates a temporary object. Hopefully, that's going to change once I find a spare minute. :-)

4.6 `fold`, `map` and `join` and Friends

Iterators over type distinct aggregations are impossible for the same reasons as for a value dependant access operator. Consequently, standard generic algorithms won't work.[2]

Nevertheless there is a need to apply operations on all components of a vector. As an alternative the second order function templates `foreach`, `map`, `join` and `fold` are provided. All these are instantiated with an operator class providing an unitary or binary operation and apply an object of that class component-wise to a vector (two vectors in the case of `join`). They differ, however, in the way the return values are handled. The `foreach` function ignores them, but returns the function object itself. That's the candidate for state based algorithms. It comes in a `const` as well as in a non-`const` version, so it can be used to modify a vector. The `map` function template takes an unary operation and applies it component-wise to a vector, constructing a vector of results, `join` does the same with a binary operator on the respective components of two vectors, while `fold` applies an unary operator to the components of a vector and assembles the results by a binary operator returning a scalar.

For predicates, i.e. function objects returning a boolean value, the `forall` functions are available, allowing short-cut evaluation. These are specialized versions of the `fold` and `join` operators.

All second order functions are implemented as member functions to simplify recursion. For convenience a canonical non-member version is provided as well.

In some circumstances it's even necessary to convert a vector into a sequence, mostly due to the fact, that many Linear Algebra libraries implement vectors as arrays. For such cases a special `copy` member function is provided, taking an iterator as argument. For the reverse conversion an unary version of the `map` function is used.

4.7 Norms

The `norm` function is left there mainly for historical reasons. It is a convenient way to define vector norms, which now could as well be defined using `map` and `fold`. Other than the functors in the STL, the functors used by `norm` do not just provide a single function call operator, but two: an unary and a binary one. The *norm* module provides an example implementation for the p-vector norm.

For lack of a better place it also holds some implementations for standard functors useful in the context of vectors. Some are redundant to the STL but with the added bonus of allowing type deduction.

[2] Most of them wouldn't make sense anyway.

4.8 The Comma Operator

Certainly, the most controversial point will be the definition of a comma operator. I know, that this is depreciated and I'm well aware of the arguments. I still decided to provide one, mainly since I'm convinced that the problem is not the overloading, but a common misconception regarding the build-in version of the comma construct. Anyway, I don't want this to be a show stopper, and I'd gladly accept an alternative if we could find one. We will, however, need to find a solution for the following problem:

For an n-dimensional vector there should be a simple way to specify all n components in an intuitive manner. The obvious choice, an appropriate constructor, can't be defined for a recursive data structure, since there's no way to specify the number of arguments depending on a template parameter, let alone the correct type. The nested use of the composite constructor, however, that would be an option with the given implementation, is anything else but intuitive in every day use.

An alternative option is to provide a *lift* operator to compose vectors. A possible candidate is again the function call operator. With the vector template dimension parameter set to the default of $D = 1$, this would allow code like:

```
// pseudo code, needs to be a member function
template<typename T, unsigned int D>
        vec<T,D+1> operator()(vec<T,D> v0, T c0) {
                return vec<T,D+1>(c0, v0); }

vec<int,3> = vec<int>(1)(2)(3);
```

The leftmost expression would be the constructor for a one-dimensional vector while the following function calls are lifting an n-dimensional to an $n+1$-dimensional vector, initializing the newly added component with the scalar values passed along the way. Not exactly nice, but at least save.

The next intuitively obvious choice would be the add operator. Even without any changes to the current implementation it can be used for initializing:

```
typedef typename euclid::dim<T,0> dimX;
typedef typename euclid::dim<T,1> dimY;
typedef typename euclid::vec<T,2> vect;

vect x = vect() + dimX(1) + dimY(2);
```

With another version of the add operator it could be simplified to:

```
template<typename T, unsigned int D>
        vec<T,D+1> operator+(vec<T,D> v0, dim<T,D> d0) {
                return vec<T,D+1>(d0, v0); }

template<typename T, unsigned int D>
        vec<T,2> operator+(dim<T,0> d0, dim<T,1> d1) {
                return vec<T,2>(d0, d1); }

vect x(dimX(1) + dimY(2));
```

That doesn't look bad at all, but consider:

```
dimX x
dimY y
...
dimY d = x+y; // compiles smoothly;
              // creates a vector and picks y
```

The initialization of `d` is with the value of `y` most likely not what the programmer had in mind. Since the whole library is fostered by the idea of detecting such errors at compile time, it tends to undermine all the work that has been done so far.

Taking all binary operators and leaving out those already defined for vectors or dimensions, there only remain `<<`, `>>`, `&`, `^`, `|`, `&&`, `||` and the comma operator. Apart from the last one, only the `|` operator and perhaps the `&` operator might be an acceptable choice.

So why do I think the comma operator is acceptable despite the arguments against it? This is probably not the place to discuss it in detail, but I'll give the my main arguments:

1. The comma operator has the lowest priority, even lower than any assignment, so whenever it's used to compose a vector it will have to be placed into parentheses. This is an obvious syntactical marker and easy to spot.

2. A comma operator used in an expression to deliver some value is syntactically very similar to a function call (without the functor). The comma in a function call behaves very similar to the overloaded version of the comma operator and very differently to the build-in comma construct[3]: It does not mark a sequence point and the order of evaluation is explicitly undefined. A C++ programmer is used to it.

3. The overloaded version only matches vectors and dimensions, all other cases are not effected. Programmers using vectors will get used to the different meaning pretty soon, those not using it don't need to worry.

[3] Which actually is a separator rather then an operator: The build-in comma has no functional semantic whatsoever; it only allows a sequence of (expression-) statements to appear were otherwise syntactically just an expression would be acceptable.

4. There's always an option to fall back to the build-in version, if desired. All that's needed is to make sure that the operands don't have a dimension type. A simple way to achieve this is to call the base type access operator. (Provided, of course, the base type doesn't overload the comma operator, in which case it should provide a fall back option too.)

As I said, I expect a lot of discussion regarding this issue and I don't want to be it a show stopper. I'd accept any solution the community is prepared to accept.

4.9 Diagonal Matrices and Matrices

The dimension and vector library has been developed with applications in mind that deal with Euclidean geometry in one way or the other. This is typical for graphical interfaces. They often require scaling of vectors or mapping them from one vector space into another. Mathematically this is described by multiplying a vector with a matrix. For this purpose the library includes a diagonal matrix class template and a matrix class template.

In many situations a diagonal matrix will be the more appropriate choice. Apart from being simpler and more efficient, there are some fundamental as well as a rather pragmatic reasons. Fundamentally, applying dimensional analysis to any matrices is much trickier than it is for diagonal matrices, this will be discussed in the next chapter. The pragmatic reasons are given by the fact, that scaling might involve dividing a vector by a matrix, which is equivalent to multiplying it with the inverse of that matrix. Inverting a matrix, however, is trivial only for diagonal matrices.[4]

It should be pointed out that this is by no means intended to be a Linear Algebra library, so any non-trivial algorithm is considered to be beyond its scope. Consequently the operations provided for matrices – diagonal or not – are limited to the most elementary ones. Inverting an arbitrary matrix is definitely not elementary. The general matrix class is in here only to provide a general mapping operator. If you need to invert it or do any other non-trivial stuff, you'll have to use a proper Linear Algebra library.

The diagonal matrix class template could be implemented using the same recursive structure like the vector class template, only that components are dimensionless, i.e. they're scalars rather then dimensions. In fact, earlier versions of the library did it this way. However, without type distinct components a diagonal matrix is much more a traditional container then vectors are. Consequently, this idea has been given up and the diagonal class is implemented as an array conforming to STL standards, i.e. providing iterator access as well as a subscription operator.[5]

The equivalent structure of a recursive diagonal matrix template and the vector template suggests to implement one in terms of the other. Both ways have

[4] As long as the elements of the matrix are invertible.

[5] Inserting, removing and reordering are still pointless and not supported, though.

been tried in the past and proven to be possible. However, the discussion in the following chapter will show, that – despite of the similarities in implementation – vectors and matrices are different concepts. Hence, to retain flexibility as well as efficiency, both concepts have been implemented individually.

4.10 Exception Safety

The implementation is exception neutral and basic exception safe. Strong exception safety is considered as too expensive to be guaranteed. Base types will usually be numerical types with assignment operators that will not fail, hence a simple assignment statement is already strong exception safe. It might be useful to provide a non-throwing swap function for critical cases, but I haven't done that yet. After all, `<std::complex>` doesn't provide it either. Basic exception safety is simple to guarantee, since non of the classes relies on any invariant constrains or requires any resource management.

4.11 I/O

The library deliberately provides only very limited I/O support. It is restricted to simple output operators intended primarily for debugging. It might be reasonable to add a corresponding input operator, anything beyond that is considered to be application dependent. Application developers have all necessary tools to write their own I/O facilities.

4.12 File Layout

The current implementation contains the following files:

`euclid/types.h`:

A the forward declarations of the class templates defined in other modules, probably expendable.

`euclid/traits.h`:

The binary (and some unary) operators defined for dimensions, vectors and matrices come in two flavours: In the default version the return type is determined by the left hand side operator. The advanced version deduces the return type from both operators using type traits. This is the place, where these traits are defined. The current implementation relies on the GNU `typeof` operator and is therefore not guaranteed to be portable.

`euclid/dimension.h`:

Here lives the dimension class template, providing all the functionality for orientational analysis. It can be used as a stand-alone module. It only depends on type traits, if advanced type deduction is required.

`euclid/vector.h`:

This module provides both, the base case and the recursive vector class template. It depends on dimensions, since dimensions are needed for component access. Consequently, it also needs type traits, if advanced type deduction is required.

`euclid/diagonal.h`:

Euclidean vectors often need scaling. This is done by multiplying them with a matrix. The library offers two matrix class templates, the one restricted to diagonal matrices lives in this module. It depends on vectors, since it also contains the operators for multiplying vectors and matrices. The reasons for having a dedicated class for diagonal matrices have been discussed above.

`euclid/matrix.h`:

Her lives the general matrix class. It depends on vectors for the same reasons as above and on diagonal matrices since it also contains the conversion functions. The concept of matrices is independent of that of vectors and dimensions, so it would be a candidate to live in its own library.[6]

`euclid/norm.h`:

Both, vectors and diagonal matrices offer a `norm` function requiring dedicated norm-functor objects. An example implementation of the p-norm lives here. The module further offers some standard functors for a `fold`, `map` and `join`.

`euclid/cuboids.h`:

This is the least sophisticated module yet and doesn't really belong here. It uses vectors to implement the generalized version of the concept that in two-dimensional space is simply called a rectangle. It should go into a geometry library one layer above the vector class and is left here only as an example for possible vector class applications.

`euclid.h`:

A user friendly all-in-one header; includes all other header files defined in the library. Currently the I/O functions live here too.

[6] The matrix modules and even more so the test code for the matrix module has easily become the major part of the library.

`euclid/test/`:

The test directory: the most important bit in here is the fail-check test. It checks if language constructs which are syntactically legal yet semantically illegal are actually detected by the type system during compile time. It consists of a sample programme (`failcheck.cpp`) and a test script (`failcheck.sh`). The script scans the sample programme for test cases and tries to compile it with one test activate at a time. The compilation is supposed to fail for each try. The test script will succeed if and only if all tests have failed. The directory further contains a rather naïve test programme (`euclid.cpp`) basically left here since an additional test never hurts. A full scale regression test using the Boost Test library lives in its own sub-directory.

`euclid/test/boost/`:

The regression tests using the Boost Test library and required for Boost, just in case the library should ever make it in there.

`euclid/example/`:

A simple example application can be found here. It compiles with Qt3, but should be simple enough to be easily ported to more recent versions.

`euclid/doc/`:

All documentations, including this document.

4.13 Usage

Many applications, modules or classes will need vectors and dimension of one specific base type only. As already demonstrated in the examples above, the instances can easily be `typedef`ed to short and intuitive names, that can be used almost like built in types:

```
typedef ... scalar_type;
typedef typename euclid::dim<scalar_type,0> dimX;
typedef typename euclid::dim<scalar_type,1> dimY;
typedef typename euclid::vec<scalar_type,2> vect;
```

For classes, this can be be done using template parameters, see `cuboid.h` for an example.

4.14 Euclidean Vectors and Physical Dimensions

Euclidean vectors and physical dimensions are orthogonal concepts and should go smoothly along with each other. The vector templates forward any legal operation to the base type; there's nothing in the code to prevent the base

type to do their own checks. Consequently, there's nothing to prevent physical dimensions being used as an Euclidean vector's base type. The same applies to orientational dimensions. The checks of both templates would be conjunctive. Technically a vector with a quantity as a base type should equivalent to a quantity with a vector as a base type, though the later obviously reduces the number of check to be performed. Please note, however, that I never actually really tried this.

Chapter 5

Dimensional Analysis and Matrices

Having invested all that work into the implementation of an Euclidean vector class naturally suggests extending the concept to another class of algebraic objects: matrices. Matrix operations conform to vector axioms, so mathematically matrices are vectors. They are, however, a far more general concept then Euclidean vectors and need careful consideration when dimensional analysis is involved.

Vectors, as discussed above, are *uniform*. Their components are dimensionally different only with respect to their orientational dimensions. They might carry other dimensions as well, like physical units, but if they do, they'll be the same for all components. This implies, that uniform vectors can be written as a product of a unit, carrying all but the orientational dimensions, and a vector of components with only orientational dimensions. The orientational dimensions are carried by the components of the vector, not by the vector itself, hence the vector as a whole will still be dimensionless.

For matrices, thinks can be different, for example when representing linear equations in science and engineering. Matrix entries can have different dimensions with significant consequences for matrix operations. A good introduction can be found in [1], where the following examples are cited from.

Take the following three matrices of base units and try to square them:

$$\mathbf{X} = \begin{bmatrix} m & s \\ s & m \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} m & ms \\ m/s & m \end{bmatrix}, \quad \mathbf{Z} = \begin{bmatrix} 1 & s \\ 1/s & 1 \end{bmatrix}.$$

You'll find it impossible for $\mathbf{X}$, since the operation would involve summing up square metres with square seconds, which is undefined. However, $\mathbf{Y^2}$ and $\mathbf{Z^2}$ are defined, $\mathbf{Z^2}$ even prevents all dimensions, so that not just $\mathbf{Z}^3$,$\mathbf{Z}^4$... do exist, but also $\mathbf{Z} + \mathbf{Z}^2 + \mathbf{Z}^3$... and hence $\mathbf{e^Z}$; while $\mathbf{e^Y}$ does not.

Similar for operations involving row- or column matrices: A two-port electrical circuit might be described by the following equation, where v_k represents

a voltage and i_k an electrical current:

$$\begin{bmatrix} v_2 \\ i_2 \end{bmatrix} = \begin{bmatrix} a_1 & r_1 \\ 1/r_2 & a_2 \end{bmatrix} \times \begin{bmatrix} v_1 \\ i_1 \end{bmatrix}$$

To be dimensionally consistent any a_k needs to be dimensionless, any r_k an electrical resistance, i.e. having the dimension of a voltage divided by a current. Obviously not all operation commonly defined for (dimensionless) matrices hold in the presents of dimensions.

The possibility for their components to carry dimensions that differ in other aspects but their orientation in space makes a (non-uniform) one-dimensional matrix very different from an Euclidean vector. Extending the concept implemented in the Euclidean vector class templates to one-dimensional matrices seems to be feasible, for example by replacing the integral dimensional template parameter by, let's say, a type list. This, however, would be significantly more complex than an Euclidean vector and, in any case, conceptually Euclidean vectors seems to be general enough to live in its own library anyway.

Matrices, however, are important as operators to transform vectors between different vector spaces. What needs the dimensional structure of a matrix to be, that can be multiplied by a vector? Euclidean vectors are uniform, which implies that their components carry one an only one orientational dimension. Assuming x, y and z are unit vectors we could formally define:

$$\begin{bmatrix} 1 & x/y & x/z \\ y/x & 1 & y/z \\ z/x & z/y & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

These matrix would even preserve the dimension when being squared, like **Z** above. The dimensionless diagonal matrix is included as a special case. The problem with this approach is the unanswered question, how unit vectors are being divided.

An alternative option which avoids that problem is to define the product of two orientational dimension to be:

$$d_i \cdot d_j = \begin{cases} 1, & \text{if } d_i = d_j \\ 0, & \text{otherwise} \end{cases}$$

This is consistent with the definition of the scalar product. Now we could write:

$$\begin{bmatrix} 1 & xy & xz \\ yx & 1 & yz \\ zx & zy & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Squaring this matrix would preserve the dimension too and also includes the diagonal matrix as special case. Now a matrix can be interpreted as a vector of a vector, i.e. something like:

```
template<typename T, unsigned int D>
        class matrix: public vec< vec<T,D>, D> { ... };
```

Mathematically this is not quite sound since a vector is used as a scalar while a scalar is supposed to be a member of a field, but it's interesting to note, that with the vector class templates as currently defined and a reasonably powerful type deduction system most matrix operations will work as expected. This is certainly true for adding two matrices and multiplying a matrix with the base type. Multiplying a matrix with a vector can be treated as scalar multiplication. This is done component wise with both, component and multiplicand being vectors. The product of two vectors is a scalar, hence the result is a vector of the correct type. Another way to interpret it (and the better match) is the scalar product. Now the components of matrix and vector are multiplied pairwise, i.e. vectors with scalars leading to a set of vectors being summed up. The result is a vector again, but now being the product of the transposed matrix with the vector. The only operation that doesn't fit is matrix multiplication, collapsing to a single base type value.

All in all the discussion shows that matrices are much more complex then vectors and that the restriction to diagonal matrices is a reasonable choice for the time being.

5.1 Space Complexity

A library implementing a recursive data structure will never compile any loops into the target code. Instead, all recursions are expanded at compile time to sequences of statements. Vector and diagonal matrix operations have a linear space complexity, for arbitrary matrices it will be polynomial. Multiplying two 3×3 diagonal matrices requires 3 multiplications, multiplying two 3×3 arbitrary matrices already 27 multiplications plus 18 additions, all compiled sequentially into the target code. Is that feasible for anything beyond 2×2 matrices? Probably not, a matrix library is likely to provide only an additional type checking layer, but forward the actual computations to traditional dimensionless but iterative algebraic libraries.

5.2 Conclusion

The library presented here implements uniform Euclidean vectors. Extending the concept to non-uniform vectors and matrices would be an interesting step to go, but this is a major step and not done in a hurry. May be, the need for it will arise in future. The diagonal matrix class enclosed in the library is a kind of smallest common divisor between Euclidean vectors and matrices. Being restricted to the to the dimensionless elements of the main diagonal makes it dimensionally sound. The general matrix class now also available in the library doesn't contribute to dimensional analyses at all. It is a mere helper class to allow vector space transformation.

Bibliography

[1] Hart, George W. (George William): *Multidimensional analysis: algebras and systems for science and engineering*, New York, Springer (1995), ISBN: 0–387–94417–6.