ERC CISST Software
cisstVector User Guide

Ofri Sadowsky, Anton Deguet

Date: 2006/01/30 04:41:54

Copyright © 2005 Johns Hopkins University (JHU). All rights reserved. No part of this document may be redistributed in either printed, electronic, or any other form or by any means, either including the original content or a modified form of this document, without the express permission of the copyright holders.

Contents

1 Introduction **3**

2 Class families **6**

2.1 Vectors . . . 7

2.1.1 Fixed-size vectors . . . 8

2.1.2 Dynamic vectors . . . 18

2.1.3 Vector architecture summary . . . 25

2.2 Matrices . . . 26

2.2.1 Fixed-size matrices . . . 26

2.2.2 Dynamic matrices . . . 32

2.3 Matrix and vector slices . . . 32

2.3.1 New definitions for row and column subvectors . . . 33

2.3.2 Defining submatrix and subvector types . . . 34

2.3.3 A few more tricks . . . 35

2.4 Transformations . . . 36

2.4.1 Rotation representations . . . 36

2.4.2 Representation conversions and normalization . . . 37

2.4.3 Applicable rotation classes . . . 39

2.4.4 Frames . . . 40

2.5 Other classes . . . 41

2.5.1 Barycentric vectors . . . 41

2.5.2 Quaternions . . . 42

3 Method and function interfaces **44**

3.1 Class-scope types . . . 45

3.2 Common interfaces . . . 47

3.2.1 Constructors . . . 47

3.2.2 Assignments . . . 50

3.3 Vectors . . . 53

3.4 Matrices . . . 53

3.5 Transformations . . . 53

4 Memory management **54**

5 Interfacing other packages **55**

Chapter 1

Introduction

This document is a tutorial for the cisstVector library API. It covers the stucture of most of the class system and the function interface, and includes some usage examples. This tutorial does not include build instructions for the library, a function-by-function documentation, or an explanation of the library internals. These are covered in separate documents. The goal of the document is to provide enough background for novice and advanced users of the library, so that they are able to

- understand the terminology of the library;
- understand the API-level documentation (e.g., doxygen-generated web pages);
- locate the API functions they need;
- make efficient use of the features of the library, according to their programming needs;
- build interfaces between the cisstVector library and other programming packages.

The tutorial is organized as follows. Chapter 2 presents the families of classes in the library and discusses their structure and relations. Chapter 3 explains the method and function API principles and presents a few usage examples. Chapter 4 explains with more details the principles of managing memory by library objects. This is a more advanced section. Chapter 5 demonstrates how to build interfaces between cisstVector objects and other libraries and packages. Examples include the C++ Standard Template Library (STL) and numerical libraries such as C LAPACK.

The focus in this guide is on explaining the library API (application programming interface), which includes concepts, types and methods. There are a number of internal classes defined in the library, which are not part of the API, and should not be accessed directly by the library's users. Most of them are presented by name, with a brief explanation of their purpose, but without full documentation. Also, we do not include here a full documentation of the API methods, as this guide is meant to supplement a separate reference document.

Naming convention

The naming convention for objects, classes, functions, macros, and so on in the cisstVector library is the same as in the CISST package. The basic rules are:

1. Library-scope names begin with a three-letter library identifier. The library identifier of cisstVector is `vct`. For function, class, and type names, the library identifier is in lowercase, and the actual meaningful name begins with a capital letter. For example: `vctCrossProduct` (function), `vctDynamicVector` (class). For constants, the library identifier is in uppercase. For example, `VCT_ROW_MAJOR`.

2. Compile-time constants, such as macros and `enum` symbolic names, are all uppercase, with an underscore separating words. For example, `VCT_CONTAINER_TRAITS_TYPEDEFS` (macro), `SIZE` (enum).

3. Class members, including both functions (methods) and variables (data members), all begin with a capital letter, and words begin with a capital letter. For example, `vctDynamicVector::SumOfElements` (method).

4. Local variables and objects, defined within the scope of a function definition or declaration, begin with a lowercase letter, and new words begin in uppercase. For example
   ```
   vctDouble3 translationFromBase;
   ```

5. Template parameters start with an underscore, followed by a lowercase letter, and new words begin with capitals. For example, `_elementType`.

6. Library classes often define class-scope (or *local*) types. We draw this idea from the design of STL, but it is a key idea in generic programming. The main purpose of the local types is to define new types which have a specific relationship with the type of the external class. Immediate examples from STL are `Container::value_type` to relate to the type of each element within a container, and `Container::iterator` to define the type of an iterator over the elements.

   The names local type system are often based on equivalent structures in STL, and in such cases are given equivalent name. For example, the cisstVector classes usually define `value_type` and `iterator` with the same functionality as above. In some cases, this naming convention is extended to cover cases that do not exist in STL, such as `pointer`, which complements the STL `reference` type in defining a pointer to an element. However, in many other cases the library requires a special type which is not a container-generic type. In such cases, the name of the type starts with a capital letter, new words also start with capitals, and the name ends with the word `Type`. For example, `ThisType`, `BaseType`, and, in matrix classes, `RowRefType`.

7. Each class defines a local type named `ThisType`, which refers to the type of the class itself. This is done for convenience reasons: the name `ThisType` is often shorter than

the class name, especially when the class is templated, and it provides a more generic and uniform interface for the classes. In addition, if class `B` is derived from class `A`, then `B` defines as its base type in a local typedef. Here's an example (the template parameters of the base type are deliberately omitted).

```
template<class _elementType>
class vctDynamicMatrix
    : public vctDynamicMatrixBase</* ... */>
{
public:
// ...
    typedef vctDynamicMatrixBase</* ... */> BaseType;
    typedef vctDynamicMatrix<_elementType> ThisType;
// ...
};
```

Chapter 2

Class families

This chapter outlines the list of classes in the cisstVector library divided into families. It provides the declaration of each class (i.e., the list of template parameters and class name), but not the definition or the list of methods. The purpose of the chapter is to make the reader familiar with class names and concepts used in the library, without going into implementation details or specific operations. Operation names that appear in this section serve to illustrate the use of the class. They are presented informally, and we assume their meaning is intuitive.

From a functional aspect, the classes in cisstVector are divided into three main categories: vectors, matrices, and transformations. This document follows this division. However, objects are classified according to multiple criteria which define a structure of class families. The criteria are:

- The time when the object size is determined. We differentiate objects whose size is defined as a "literal" in the program, that is, an explicit and fixed number, and objects whose size is determined during the execution of the code, for example if it is part of the user input. We use the words `FixedSize` in the name of the first kind of objects, and the word `Dynamic` in the second kind. Examples of fixed-size objects include a 3-element vector to represent a point in space, and a 4×4 matrix to represent a homogeneous transformation. An example of a dynamic object is an image matrix read from a file, where the file header specifies the pixel dimensions of the image.

- Allocation of new memory. The library contains objects that allocate memory for their contents, such as `Vector` and `Matrix`. It also contains objects that overlay existing memory cells, and just apply a vector or matrix structure to them. We use the word `Ref` to identify the second kind. For example, `VectorRef`, `MatrixRef`. Allocating objects are intuitive and simple, for example, the same 4×4 matrix from the previous item. An example of an overlaying object would be to refer to a row or a column of that matrix as a vector object, and performing vector operations on it, while occupying the same memory cells allocated by the matrix.

Name	Dynamic/ FixedSize	const	Memory management
`vctFixedSizeVector` `vctFixedSizeMatrix`	fixed size	no	allocate
`vctFixedSizeVectorRef` `vctFixedSizeMatrixRef`	fixed size	no	overlay
`vctFixedSizeConstVectorRef` `vctFixedSizeConstMatrixRef`	fixed size	yes	overlay
`vctDynamicVector` `vctDynamicMatrix`	dynamic	no	allocate
`vctDynamicVectorRef` `vctDynamicMatrixRef`	dynamic	no	overlay
`vctDynamicConstVectorRef` `vctDynamicConstMatrixRef`	dynamic	yes	overlay

Table 2.1: API-level vector and matrix classes in cisstVector.

- Immutability. When overlaying existing memory with new objects, care must be taken not to modify its `const` attribute. That is, writing to any memory cell occupied by a const object should not be allowed. This only applies to overlaid objects, and we use the word `Const` to denote that. For example, `ConstVectorRef`, `ConstMatrixRef`. If the 4×4 matrix object from before is declared as `const`, then only `Const` overlay vectors for it can be created.

Summing up all the possible combinations for vectors and matrices, we identify 12 main classes in our libraries, summarized in Table 2.1.

To reduce the size of the code and the management work for it, we designed class hierarchies for the vector and matrix objects. These will be described next in 2.1 and 2.2. Further development of the subject of overlay vectors and matrices, specifically focused on *container slices* is in 2.3.

Built on top of the general vector and matrix classes, there is a collection of special purpose classes. These are used mostly to represent transformations, such as rotations and rigid frames. The transformation classes are described in 2.4. We conclude this chapter with a few special-purpose objects defined in the cisstVector library in 2.5.

2.1 Vectors

The main division of vector and matrix class families is along the fixed-size/dynamic line. This part presents the architecture for the fixed-size vector family first, and follows with the main changes applied to define the dynamic vector family.

The next subdivision is between allocated and overlaid vectors. Allocated vectors are more intuitive to understand, as they stand as objects for themselves. The first class presented in each subsection will be the allocated vector type. Then we will explain how to use the different overlaid vector types.

The last subdivision is between mutable and immutable objects. Immutable operations are shared among all the vector representations, while a memory block overlaid with a mutable vector interface must be mutable itself.

When examining all the different vector types, we can see large similarities between them. These similarities are factored into base classes, and the actual vector types are subclasses derived from them. At the end of each subsection, we show how the common features are grouped into base classes, and how the API-level classes are generated from them.

2.1.1 Fixed-size vectors

vctFixedSizeVector

The most accessible class in the fixed-size vector family is `vctFixedSizeVector`. It is declared as a class templated by the type of vector element and the length (or size) of the vector:

```
template<class _elementType, unsigned int _size>
class vctFixedSizeVector;
```

Instantiating a fixed-size vector object is simple and straightforward. For example, to create a 3D vector of double elements, write

```
vctFixedSizeVector<double, 3> p1;
```

For vectors of size 1 to 6, with element types of `char`, `int`, `float` and `double`, the library defines simple and short names through `typedef` statements. For example:

```
typedef vctFixedSizeVector<int, 1> vctInt1;
typedef vctFixedSizeVector<char, 2> vctChar2;
typedef vctFixedSizeVector<double, 3> vctDouble3;
typedef vctFixedSizeVector<float, 4> vctFloat4;
```

The simple names will be used throughout the examples here, as they are easier to read and to write than the full template instantiations. In addition, the library provides even shorter names for vectors of type double, which is assumed to be the "default" element type:

```
typedef vctFixedSizeVector<double, 1> vct1;
typedef vctFixedSizeVector<double, 2> vct2;
typedef vctFixedSizeVector<double, 3> vct3;
// ...
```

This guide will sometimes use these short names, especially when the element type is of no importance.

Vectors can be initialized or assigned with values in multiple ways, for example:

```
// Initialize all the elements with the same value
vctFloat5 vertex(1.0f);
// Initialize elements directly in vector constructor
vctInt3 areaCode(4, 1, 0);
// Elementwise initialization applies for any vector
// length.  NOTE: the initialization parameters must be
// cast to, or have, the same type as vector elements.
vctFixedSizeVector<double, 8> weights(
    1.5, 3.25, 6.0, double(vertex[0]), 24.0, 48.0, 96.0, 192.0);
// Copy one vector to another, including type conversion
vctFloat3 areaCodeF( areaCode );
// Assign new values to an existing vector.  NOTE: the
// parameters must have the same type as vector elements.
vertex.Assign(-7.33f, 0.0f, 1.17f, 5.62f, float(areaCode[1]) );
```

For a more complete list of initialization methods for vectors, see Chapter 3.

Operations with vectors include some that take one operand, typically the target object, and some that take multiple operands. Most single-operand operations are immutable, as they compute either a scalar or vector outcome of a vector. For example,

```
const vct4 v4(1.0, -2.0, 3.5, -4.25);
// compute the norm of a vector
double n = v4.Norm();
// compute the sum of the elements of a vector
double s = v4.SumOfElements();
// negate a vector
vct4 minus_v4 = v4.Negation();
// take the absolute values of the elements
vct4 abs_v4 = v4.Abs();
```

A small number of operations are applied to a single vector and store the result in the same vector, i.e., they are mutable. Usually, the names of these operations end with the word `Self`. For example,

```
vct4 minus_abs_v4( abs_v4 );
minus_abs_v4.NegationSelf();
```

Operations with multiple operands can be between vectors and vectors, or between vectors and scalars. For example,

```
vct3 position(0.0);
vct3 translation(-3.0, 10.0, 210.0);
// add another vector to the target vector.
position.Add(translation);
// subtract a scalar from all elements of a vector
translation.Subtract(2.5);
// find the difference between two vectors and store
// the result in a third
vct3 secondPosition(6.0, 3.7, 9.66);
vct3 displacement;
displacement.DifferenceOf(position, secondPosition);
// compute the dot product of vectors
double dotProd = secondPosition.DotProduct(translation);
```

Normally, the cisstVector library requires that all the vector operands have the same element type (or scalar type), and the operand sizes must match. For the fixed-size vector family, a compiler error will be generated if the sizes of the operands don't match. For example,

```
vct3 p1(4.1, -2.1, 6.23);
vct4 p2_homogeneous(-7.5, 21.3, 9.09, 1.0);
// size mismatch! compilation error!
double dot = p1.DotProduct(p2_homogeneous);
```

The first elements of a vector are given the special names `X()`, `Y()`, `Z()`, and `W()`. Subvectors consisting of the first elements are also given special names: `XY()`, `XYZ()`, `XYZW()`. One way to solve the former compilation error is by dividing by the homogeneous coordinate:

```
double dot = p1.DotProduct( p2_homogeneous.XYZ() /
    p2_homogeneous.W() );
```

Most of the examples so far included operations specified by an explicit name, as opposed to a typical notation of arithmetic symbols such as `+`, `-`, `*`, etc. The cisstVector library defines named methods for all of the operations, and we encourage using them. One reason for using named methods is that the C++ syntax is poor regarding the meanings of operators. Some common vector operations, such as norm, absolute value, cross product and dot product, have no natural corresponding C++ operator. When more complex algebraic objects are used, some operators lose their meaning, and others become ambiguous. For example, we choose to represent the dot product operation between vectors using the operator `*` , but then the operator `/` is no longer the inverse operation of `*` (dot product has no inverse), and we are still short of having an operator symbol for element-by-element (or *elementwise*) product, which in Matlab, for example, is defined as `.*` .

In addition, C++ typically requires overloaded operators to return new object instances, which is sometimes inefficient, due to the memory allocation and element copying involved.

The most important use for overloaded operators is to store intermediate results of an expression which the user does not want to create a named object for. In the last code example, `p2_homogeneous.XYZ()` is a 3-element vector, and it is divided by a scalar to obtain a temporary `vctFixedSizeVector`, which is then used as an operand in the dot product function. The same expression can be written concisely as

```
double dot = p1 * (p2_homogeneous.XYZ() / p2_homogeneous.W());
```

Whether or not this is a clear syntax depends on the reader's taste. Named functions, although generating longer source lines, are at least unambiguous and have fewer side effects than overloaded operators.

Additional to the arithmetic and algebraic operations on vectors, the cisstVector library includes simple direct *accessor* operations. The most straightforward one is the subscript operator `[]`, which was used in some of the examples above. Another one, worth mentioning, is the named method `Pointer(int index = 0)`, which returns an address of an element rather than the element itself. The two expressions

```
&(v[2])
```

and

```
v.Pointer(2)
```

yield equal values, but the second form is more readable to us, and more useful in complex expressions. In addition to `operator []` the library includes a method named `Element`, as part of our principle to encourage the use of named methods. Its use is identical to `operator []`:

```
int g = v.Element(0);
v.Element(2) = 8;
```

`vctFixedSizeVector` also provides an STL-compatible interface, which includes internal type definitions such as `value_type`, `size_type` and `difference_type`, and a collection of iterators. These will be described in detail in Chapter 3. For now, we note that `value_type` is an internal type defined as the template type parameter `_elementType`. It may be used in examples ahead.

A more complete listing of vector operations is in Chapter 3. To summarize this part, one can say that fixed-size vectors are the most common token in the cisstVector library. They can be defined with any size and element type, and there is a large variety of operations that can be applied to them. The next paragraphs explain how more sophisticated types of vectors can be created and manipulated.

vctFixedSizeVectorRef

Recall the example of normalizing homogeneous coordinates, given in the previous subsection. Here's a variant of it:

```
vct4 position_h(-15.0, 42.6, 18.18, 2.0);
vct3 position = position_h.XYZ() / position_h.W();
```

This example illustrates the use of overlaid vectors, as the `XYZ()` method returns an overlaid vector object. This means that the user can perform any operation on `position_h.XYZ()` as a 3-element vector, yet the actual memory cells involved are the ones allocated for `position_h`. An obvious example would be

```
position_h.XYZ().SumOfElements()
```

which returns the sum of the first three elements in the vector. However, mutable operations can be applied as well:

```
vct3 initialPos( /* ... */ );
vct3 translation( /* ... */ );
vct4 finalPos_h(0.0, 0.0, 0.0, 1.0);
finalPos_h.XYZ().SumOf(initialPos, translation);
```

This code stores the sum of the two input vectors directly into the elements of `finalPos_h`, without going through any intermediate storage. The reason is that the `XYZ()` method returns an object of type `vctFixedSizeVectorRef`, which is an overlay vector object.

The declaration of `vctFixedSizeVectorRef` is as follows.

```
template<class _elementType, unsigned int _size, int _stride>
class vctFixedSizeVectorRef;
```

The `_elementType` and `_size` parameters have the same meaning as in `vctFixedSizeVector`; but the class introduces a new template paremeter: `_stride`. The `_stride` parameter defines the spacing between adjacent elements of the overlaid vector, counted in element units. In the `XYZ()` example above, the stride is equal to 1, because the new vector is overlaid on part of a vector that has been allocated continuously. As we will see soon, overlay vectors with different stride values can be defined for more sophisticated cases.

Objects of class `vctFixedSizeVectorRef` can be instantiated directly. They are set to reference to a memory address of their first element by passing a pointer to the constructor or to a method named `SetRef`. For example:

```
// transformParams contains 3 Euler angles in
// elements 0 to 2, and 3 translation components
// in elements 3 to 5.
vctDouble6 transformParams;
// create an overlay vector for the translation
vctFixedSizeVectorRef<double, 3, 1>
  transformTranslation( transformParams.Pointer(3) );
// create an overlay vector for the Euler angles
vctFixedSizeVectorRef<double, 3, 1> transformEulerAngles;
transformEulerAngles.SetRef( transformParams.Pointer(0) );
```

From here on, the vectors `transformTranslation` and `transformEulerAngles` can be manipulated just like any 3-element vector. Note, though, that because they are not declared as `vctFixedSizeVector` objects, a copy of the overlaid vector will be created and passed to functions that take a `vctFixedSizeVector` parameter by value, such as

```
void f(vctDouble3 p);
```

which means that the function will not have direct access to the elements of the overlaid vector. Also, because the stride value may be different from 1, there is no direct way to cast a `vctFixedSizeVectorRef` to a `vctFixedSizeVector`. That is, a function with the signature

```
void g(vctDouble3 & p);
```

cannot take an overlay vector as its argument.

The most notable case of strides different than 1 involves accessing rows or columns of matrices as overlaid vectors. Normally, the elements of a matrix are stored in one contiguous memory block in row-major order. That is, the elements of a row lie next to each other in memory, or, in other words, the stride between row elements is 1. Yet the stride between the elements of a column is equal to the number of columns in the matrix:

```
// create a 4x4 matrix object
vctDouble4x4 xformMatrix;
// set reference to the third row as a vector
vctFixedSizeVectorRef<double, 4, 1> rowRef;
rowRef.SetRef(xformMatrix.Pointer(2, 0));
// set a reference to the fourth column as a vector.
// The stride is equal to 4.
vctFixedSizeVectorRef<double, 4, 4> colRef;
colRef.SetRef(xformMatrix.Pointer(0, 3));
```

It is possible now to perform any vector operations on the row and column vector objects. One can even refer to slices of these vectors, e.g.,

```
rowRef.XYZ().Norm()
```

and

```
colRef.XYZ().Assign(10.0, 5.0, 2.0); .
```

Note that these slice operations will return overlaid subset vectors that have the same stride as their superset vectors, i.e., 1 for the slice of `rowRef` and 4 for the slice of `colRef`.

It should be emphasized that the stride in a fixed-size vector, passed as a template parameter, is part of the type signature of the vector, and cannot be changed once the variable has been declared. That is, the users can set the `colRef` variable in the previous example to overlay any column of the matrix, or any other memory address, but they cannot change its stride, for example, to make it overlay a row of the matrix instead of a column.

To summarize this part, we saw uses of the class `vctFixedSizeVectorRef` for overlaying existing vector structures and slicing through other containers. The complete set of operations on `vctFixedSizeVectorRef` is identical to that of `vctFixedSizeVector`, except for

their constructors, operator `=`, and the `SetRef` method. See the section about methods for more information.

More on the use of overlay vectors is in Section 2.3.

vctFixedSizeConstVectorRef

Consider the following case.

```
// Declare and initialize a const 3x3 matrix
const vctDouble3x3 identity(
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0 );
// Create an overlay vector for a matrix row
vctFixedSizeVectorRef<double, 3, 1>
    rowVector(identity.Pointer(1, 0));
```

As said before, `vctFixedSizeVectorRef` provides all the operations available for `vctFixedSizeVector`, including mutable operations. This means that if the above example was legal, it would provide a mutable access to elements of an immutable object. Fortunately, the example is not legal. That is because the constructors and `SetRef` methods of `vctFixedSizeVectorRef` only take non-const pointer parameters of type `value_type *` (recall that `value_type` is the internal name for the vector's element type). The `Pointer` method on a const matrix, on the other hand, can only return a `const_pointer`, that is, `const value_type *` .

To deal with overlaying immutable vectors, the library defines the class `vctFixedSizeConstVectorRef`, with the following declaration.

```
template<class _elementType, unsigned int _size, int _stride>
class vctFixedSizeConstVectorRef;
```

The template parameters are the same as in `vctFixedSizeVectorRef`, and the behavior of the class is also the same, defining constructors and `SetRef` methods in a similar manner. The difference is that `vctFixedSizeConstVectorRef` defines only `const` methods, that is, methods that do not change the target object, while `vctFixedSizeVectorRef` includes non-const methods.

As we can see, the immutable operations are shared among all the vector classes, and the mutable operations are shared between allocating and overlaid vectors. These common features call for factorization in the form of base types, presented next.

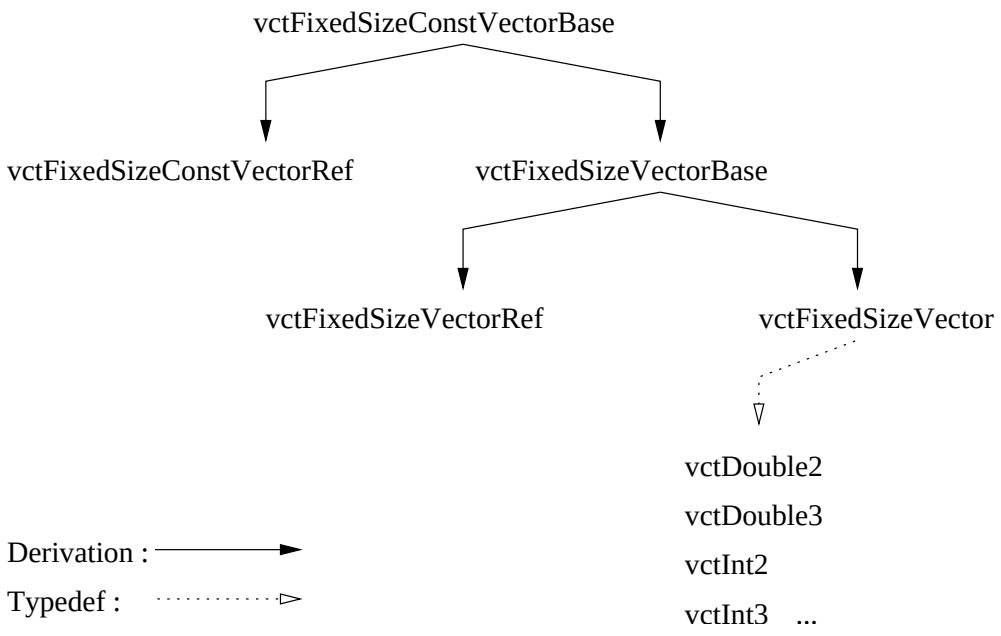


Figure 2.1: Class family for fixed size vectors

Fixed-size vector base classes

Because the immutable operations are common for all the vector objects, they have been factored to a base class at the root of the family: `vctFixedSizeConstVectorBase`. From it, we derive the mutable class `vctFixedSizeVectorBase`, which inherits all the immutable operations, and adds the mutating operations. These two classes are not API-level classes, and the user should not try to instantiate them. The API level classes are the ones in the previous subsections.

The class hierarchy for fixed-size vectors is illustrated in Figure 2.1. It listed next in brief, and then each class is explained in more detail.

- `vctFixedSizeConstVectorRef` is derived directly from `vctFixedSizeConstVectorBase`, and allows to overlay any memory block with an immutable vector interface.
- `vctFixedSizeVectorRef` is derived from `vctFixedSizeVectorBase`, and allows to overlay any mutable memory block with a mutable vector interface.
- `vctFixedSizeVector` is derived from `vctFixedSizeVectorBase`, and allocates its own memory. It is mutable unless the vector object is declared as `const`.

vctFixedSizeConstVectorBase The class declaration is as follows.

```
template<unsigned int _size, int _stride, class _elementType,
         class _dataPtrType>
class vctFixedSizeConstVectorBase;
```

Note that this declaration enables to define the number (size) and type of elements as template parameters. This is expected in any vector library. The `_stride` parameter, as explained before, specifies the memory spacing between consecutive vector elements in `_elementType` units. The `_dataPtrType` parameter enables to create both allocating and overlaying objects, as will be explained and demonstrated below.

`vctFixedSizeConstVectorBase` has one data member named `Data` of type `_dataPtrType`, which provides access to the elements. This is the only data member stored in any fixed-size vector object. The only difference between the API-level vector classes is in the specialization of the template arguments.

Accessor operations are defined at this level as follows:

```
const_pointer Pointer(size_type index = 0) const {
    return Data + STRIDE * index;
}
const_reference operator[](size_type index) const {
    return *(Pointer(index));
}
```

Recall that the `Pointer` method returns a pointer to an element (in this case to a const data). Here we see that it is computed from the base memory address `Data` plus an offset of `STRIDE * index` (`STRIDE` is an internally defined enum, equal to the template argument `_stride`). The operator `[]` "dereferences" that pointer. The return type of `operator []` is declared as `const_reference`, which is resolved as a C++ reference to a `const` element of the container, and is named according to a corresponding type in STL containers. The return type of `Pointer` is declared as `const_pointer`, resolved as a *pointer*, that is, the memory address of, a const element of the container. It is defined after the STL naming convention, even though it is not an STL interface.

All the other operations in the vector API are built around these and similar accessor methods, whose list is specified in Chapter 3. `vctFixedSizeConstVectorBase` also defines all the immutable operations on vectors, such as `Norm` etc.

vctFixedSizeVectorBase As said above, `vctFixedSizeVectorBase` extends `vctFixedSizeConstVectorBase` by adding mutating operations such as

```
this->Add(vector)
this->SumOf(vector, vector)
this->SetAll(scalar)
```

etc. Besides that, it is declared and defined in exactly the same way as `vctFixedSizeConstVectorBase`.

vctFixedSizeConstVectorRef The class definition is as follows.

```
template<class _elementType, unsigned int _size, int _stride>
class vctFixedSizeConstVectorRef : public vctFixedSizeConstVectorBase<
    _size, _stride, _elementType,
    typename vctFixedSizeVectorTraits<_elementType, _size, _stride>
           ::pointer >
{ /* ... */ };
```

`vctFixedSizeConstVectorRef` specializes `vctFixedSizeConstVectorBase` by setting `_dataPtrType` to be a pointer. The explanation of `vctFixedSizeVectorTraits` will be deferred. For now, we will simply indicate that
`vctFixedSizeVectorTraits<_elementType, _size, _stride>::pointer`
resolves to `_elementType *`. From here we can see that a `vctFixedSizeConstVectorRef` keeps a pointer to a memory location, where elements of type `_elementType` are stored, and wraps `_size` such elements in increments of `_stride` with a vector interface. Note that even though `_dataPtrType` is non-const, only const operations are provided by the base class.

vctFixedSizeVectorRef `vctFixedSizeVectorRef` specializes `vctFixedSizeVectorBase` in the same way that `vctFixedSizeConstVectorRef` specializes `vctFixedSizeConstVectorBase`, that is, by defining `_dataPtrType` as `_elementType *`. The difference between this class and the previous is that `vctFixedSizeVectorRef::SetRef` (and the constructors) can take only a non-const pointer argument, thus protecting const objects from being illegally modified. The operations available on a `vctFixedSizeVectorRef` object are the ones defined in `vctFixedSizeVectorBase`, that is, all the immutable and mutating operations.

vctFixedSizeVector The class definition is as follows.

```
template<class _elementType, unsigned int _size>
class vctFixedSizeVector : public vctFixedSizeVectorBase<
    _size, 1, _elementType,
    typename vctFixedSizeVectorTraits<_elementType, _size, 1>
        ::array >
{ /* ... */ };
```

`vctFixedSizeVectorTraits<_elementType, _size, 1>::array` resolves to
`_elementType[_size]` ,
that is, the `Data` member is now actually defined as a fixed-size array, allocated where the object is, and being part of it. If a fixed-size vector object is created on the stack, then its array member is allocated on the stack too. Note that the stride here is 1, because the elements are stored continuously in the memory.

As said before, types are specialized from `vctFixedSizeVector` by defining specific element type and size: `vctDouble2`, `vctDouble3`, `vctDouble4`, `vctInt2`, and so on.

2.1.2 Dynamic vectors

The main difference between the fixed-size vectors described in the previous section and dynamic vectors is in the allocation method. The size of a fixed-size vector is determined through template arguments at compilation time, and its memory block is allocated in the space of the object. The size of a dynamic vector is specified as a function parameter at runtime, and its memory block is allocated dynamically using `new`. Similarly, the stride of a dynamic vector is defined as a function parameter, compared to a template parameter in fixed-size vectors.

The class family of dynamic vectors is parallel to that of the fixed-size family. At the root of the family is the generic immutable class `vctDynamicVectorConstBase`. We derive the class of mutable operations `vctDynamicVectorBase` from it, inheriting all the immutable operations and adding mutable operations. These are not API-level classes, and should not be instantiated by the user. The API-level classes are:

- `vctDynamicVector`, derived from `vctDynamicVectorBase`, and allocates its own memory, which is mutable if the vector object is not declared as `const`.
- `vctDynamicVectorRef`, derived from `vctDynamicVectorBase`, and allows to overlay any mutable memory block with a mutable vector interface.
- `vctDynamicConstVectorRef`, derived directly from `vctDynamicConstVectorBase`, and allows to overlay any memory block with an immutable vector interface.
- `vctReturnDynamicVector`, a special case derived from `vctDynamicVector`, which is used when a dynamic vector object is returned by value from a function.
- Types specialized from `vctDynamicVector` by defining specific element type: `vctDoubleVec`, `vctloatVec`, and so on.

The dynamic vector class hierarchy is illustrated in Figure 2.2. Next, we explore the family in more detail. Since the API of dynamic vectors is very similar to that of fixed-size vectors, we will not go through as many examples and explanations as in the previous section. The principles are the same, and we will focus mostly on the differences.

vctDynamicVector

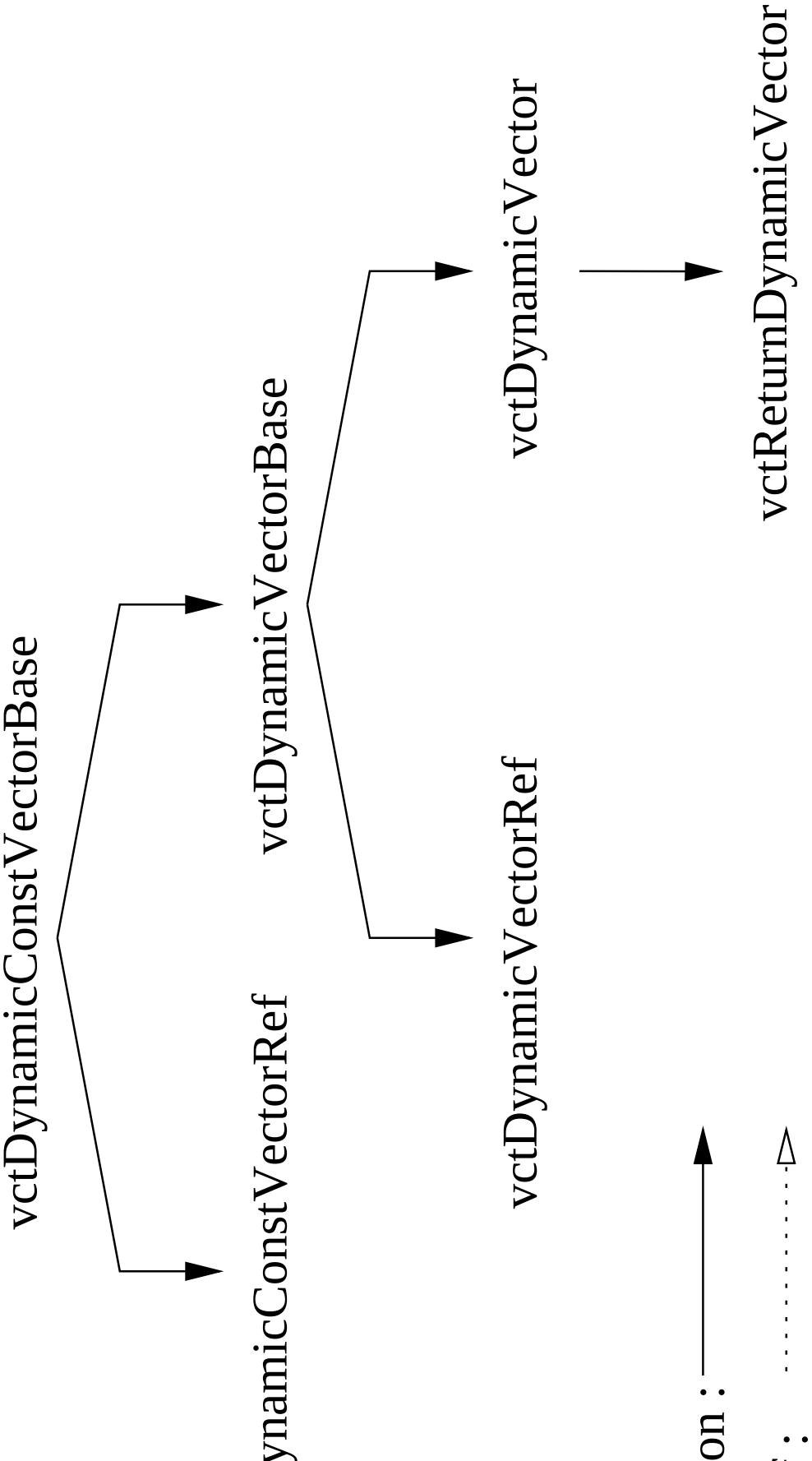
This is the most accessible class in the family. Instantiating a dynamic vector object is straightforward and simple:

```
vctDynamicVector<double> v(15);
```

where 15 is the number of elements to allocate.

The class declaration is

```
template<class _elementType> class vctDynamicVector;
```


vctDynamicConstVectorBase
vctDynamicConstVectorRef
vctDynamicVectorBase
vctDynamicVectorRef
vctDynamicVector
vctReturnDynamicVector
on :
t :

`vctDynamicVector` dynamically allocates memory using `new` and releases the memory using `delete`. The stride between the elements is always 1.

Here are some examples of using `vctDynamicVector`.

```
// define a short name for the vector type
typedef vctDynamicVector<double> VectorType;
// read the vector length from user input
int length;
std::cin >> length;
// create a vector with the input length
VectorType v1(length);
// create an empty vector, then set its size to length
VectorType v2, v3;
v2.SetSize(length);
// set all the values in v1 to the given scalar
v1.SetAll(2.0);
// assign random values to the elements of v2
int index;
for (index = 0; index < v2.size(); ++index)
    v2[index] = rand();
// scale each element of v2 by the corresponding element in v1
v2.ElementwiseMultiply(v1);
// assign v2 to v3
v3 = v2;
// compute the difference of vectors
v1 = v3 - v2;
// compute vector norm
double norm = v1.Norm();
```

Basically, `vctDynamicVector` supports all the operations defined for `vctFixedSizeVector`. Like `vctFixedSizeVector`, the constructors and assignment operators defined in `vctDynamicVector` can take any member in the dynamic vector family, and allow element type conversions. It is always safe and legal, though not necessarily efficient, to use `vctDynamicVector` objects.

The size of a `vctDynamicVector` can be passed to the constructor, to the `SetSize` method, or to the `resize` method. `SetSize` allocates new memory for the vector and discards the old data, while `resize` copies the old data to the new block allocated. Each of these operations actually allocates new memory, unless the new size is equal to the current size (even if the size of the vector is shrunk).

`vctDynamicVector` defines a copy constructor which creates a second instance of its input. That is, it allocates new memory for the copy instance, and then copies all the elements from the input. This definition of a copy constructor is the only way to ensure a safe pass-by-value

mechanism in all cases, but is not runtime efficient. In most cases, the programmer can pass a dynamic vector object to a function by reference, preferably const. `vctDynamicVector` can also be returned by value from functions, though for that purpose it is more efficient to use `vctReturnDynamicVector` objects, which we describe later.

Specialized names for dynamic vectors For the simplicity of use, the library defines a few short names for some dynamic vector types. For example:

```
typedef vctDynamicVector<double> vctDoubleVec;
typedef vctDynamicVector<double> vctVec;
typedef vctDynamicVector<float> vctFloatVec;
typedef vctDynamicVector<int> vctIntVec;
```

Note specifically that the "default" vector type is a dynamic vector of doubles.

vctDynamicVectorRef

`vctDynamicVectorRef` provides an overlay vector whose size and stride are determined during runtime. Its declaration is

```
template<class _elementType> class vctDynamicVectorRef;
```

To set the memory covered by `vctDynamicVectorRef`, one has to specify the size, location and stride of the elements. This can be done through a constructor, or through a call to the method `SetRef`. Here are some examples.

```
// variables defining the dimensions of a matrix
const int nRows = 30;
const int nCols = 20;
// define a dynamic matrix of integers with this size
vctDynamicMatrix<int> m(nRows, nCols);
// Overlay the middle 12 elements of the 5th row, starting
// at index (4,4) and ending at (4,15) (zero-based) with
// a vector.  The stride is 1, because the matrix elements
// are stored continouously.
vctDynamicVectorRef<int>
    subvector(12, m.Pointer(4, 4), 1);
// Change the overlay to the first 3 elements of the 7th
// column.  The stride is the number of columns.
subvector.SetRef(3, m.Pointer(0, 6), nCols);
```

The length of a `vctDynamicVectorRef` is passed as a parameter to the constructor or to the `SetRef` method. In either case, no actual memory is allocated for a `vctDynamicVectorRef`

except for the few data members containing the size, stride, and starting point. Note that with dynamic overlay vectors, the stride is a variable that can be changed in runtime. The choice whether it is worthwhile is at the programmer's hands.

The operations available on a `vctDynamicVectorRef` object are the ones available for `vctDynamicVector`, that is, all the immutable and mutating operations.

vctDynamicConstVectorRef

`vctDynamicConstVectorRef` provides overlay access to const memory objects. It is declared like `vctDynamicVectorRef`:

```
template<class _elementType> class vctDynamicConstVectorRef;
```

and the same semantics for constructors and `SetRef`. The class supports all the immutable operations for vectors.

vctReturnDynamicVector

In some cases, a function has to return a newly allocated dynamic vector object. For example, when `operator +` is applied between two vectors, the return value cannot be any of the operands. When a function is defined as follows

```
vctDynamicVector<int> f( /* some input */ )
{
  vctDynamicVector<int> result;
  // initialize, store values, etc.
  return result;
}
```

the C++ compiler issues a call to the return type's copy constructor. In the case of dynamic vectors, this means creating a second instance of the returned object, which wastes time on memory allocation and copying.

The class `vctReturnDynamicVector` was created to answer this problem. Instead of a copy mechanism, it uses ownership transfer. In this sense, it is a specialized class made for the purpose of returning a dynamic vector value. Its declaration is

```
template<class _elementType>
class vctReturnDynamicVector : public vctDynamicVector<_elementType>;
```

The declaration shows that `vctReturnDynamicVector` includes all the methods of `vctDynamicVector`, and also performs dynamic memory allocation. Our library defines a complete set of rules for copying and assigning between an ordinary dynamic vector and a return dynamic vector. The rules are mostly inconsequential for the API end user, and will not be presented here. From the end user's point of view, there are two rules of thumb, which should work pretty much anywhere:

1. `vctDynamicVector` can be used everywhere. Especially, when declaring a variable, it should normally be a `vctDynamicVector`. The user does not need to worry about functions that return `vctReturnDynamicVector`, because it will be transparently assigned to a `vctDynamicVector`.

2. When writing a function that returns a dynamic vector by value, the user has the option to return a `vctReturnDynamicVector`. For example:

```
vctReturnDynamicVector<int> UserFunction( /* some input */ )
{
  vctDynamicVector<int> result;
  // initialize, store values, etc.
  return vctReturnDynamicVector<int>(result);
}
```

   Note that the local variable defined in the function is still of type `vctDynamicVector`, and the type converstion is done only at the end of the function, and explicitly. This is the safest way to use this class.

Dynamic vector base classes

As it is with the fixed-size vectors, the library defines a class hierarchy for the dynamic vector family, which is illustrated in Figure 2.2. Here we will explain its structure from the root to the leaves, as we did with the fixed-size family. We will not add many details about the API classes, as these have been explained above.

vctDynamicConstVectorBase The class declaration is as follows.

```
template<class _vectorOwnerType, typename _elementType>
class vctDynamicConstVectorBase;
```

The template parameter `_vectorOwnerType` replaces the parameter `_dataPtrType` in the fixed-size vector family. The owner class defines operations and members such as memory allocation and deallocation, ownership management, accessors (e.g., `operator []`), and iterators. The cisstVector library defines two classes of vector owners: `vctDynamicVectorOwner` and `vctDynamicVectorRefOwner`. The first allocates memory, and the second overlays memory. The owners are not API-level classes, and will not be discussed in detail in this document. The template parameter `_elementType` identifies the element type of the vector. Other parameters, such as size and stride (which we have seen in the fixed-size family), are not template parameters. They are generally controlled by the vector owner, and their use is introduced in the derived classes.

`vctDynamicConstVectorBase` has one data member named `Vector` of type `_vectorOwnerType`. The element accessor operations of `vctDynamicConstVectorBase` are based on the owner:

```
const_pointer Pointer(index_type index = 0) const {
    return Vector.Pointer(index);
}
const_reference operator[](index_type index) const {
    return *Pointer(index);
}
```

Like `vctFixedSizeConstVectorBase`, this class defines all the immutable operations, such as `Norm`, `SumOfElements`, and so on. The method `Pointer` returns a pointer to an element of the vector. In the example above, it is an immutable element pointer.

vctDynamicVectorBase `vctDynamicVectorBase` extends `vctDynamicConstVectorBase` in the same way that `vctFixedSizeVectorBase` extends `vctFixedSizeConstVectorBase`, that is, by adding mutable operations such as

```
this->Add(vector)
this->SumOf(vector, vector)
this->SetAll(scalar)
```

etc. It takes exactly the same parameters as its base type.

vctDynamicConstVectorRef The class definition is as follows.

```
template<class _elementType>
class vctDynamicConstVectorRef
    : public vctDynamicConstVectorBase<
        vctDynamicVectorRefOwner<_elementType>, _elementType>
{ /* ... */ };
```

`vctDynamicConstVectorRef` has only one template parameter: the element type. It specializes `vctDynamicConstVectorBase` by setting the owner type to `vctDynamicVectorRefOwner`. By that, it defines a dynamic overlaid vector structure, whose length and stride are given through function parameters in runtime. The operations available for `vctDynamicConstVectorRef` are the immutable operations defined in `vctDynamicConstVectorBase`.

vctDynamicVectorRef The class definition is as follows.

```
template<class _elementType>
class vctDynamicVectorRef
    : public vctDynamicVectorBase<
        vctDynamicVectorRefOwner<_elementType>,
        _elementType>
{ /* ... */ };
```

`vctDynamicVectorRef` extends `vctDynamicVectorBase` the same way that `vctDynamicConstVectorRef` extends `vctDynamicConstVectorBase`, that is, by defining the owner to be the overlaid type `vctDynamicVectorRefOwner`. It provides all the mutable operations inherited from its base class, and can be set to a memory address as described before.

vctDynamicVector The class definition is as follows.

```
template<class _elementType>
class vctDynamicVector
    : public vctDynamicVectorBase<
        vctDynamicVectorOwner<_elementType>,
        _elementType>
{ /* ... */ };
```

`vctDynamicVectorOwner` is in charge of allocating, deallocating and transfering ownership of a memory block. All the operations are inherited from `vctDynamicVectorBase`, including mutable and immutable operations.

2.1.3 Vector architecture summary

As we have seen, both the fixed-size and the dynamic vector families have parallel structures. At the root is a `ConstVectorBase` class, defining immutable operations. Next is a `VectorBase` class, defining the mutable operations. The two are not API-level classes. Following are overlay classes: `ConstVectorRef` and `VectorRef`, derived respectively from the two base classes. Finally, there is an allocating `Vector` class. A similar structure exists in the matrix classes, which are presented in 2.2

We have introduced the concept of strides, which we use for overlaying non-contiguous memory cells with a vector interface. A vector is a one-dimensional object, and has one size and one stride parameters. As we will see in 2.2, matrix classes take two size and two stride parameters.

We described the division of the class families into fixed-size and dynamic containers. In the first kind, the sizes and strides are given as template parameters, and in the second they are passed as function parameters and stored as data members.

Finally, we discussed a few issues regarding to the allocation and returning of vector objects. We introduced the concept of ownership transfer in dynamic vector. This concept is also used with the dynamic matrices.

The architecture of the matrix class families is similar to that of the vector classes. Therefore, we will not repeat the presentation we did here for matrices. Instead, we will focus in 2.2 on the unique features in the structures of the matrix families.

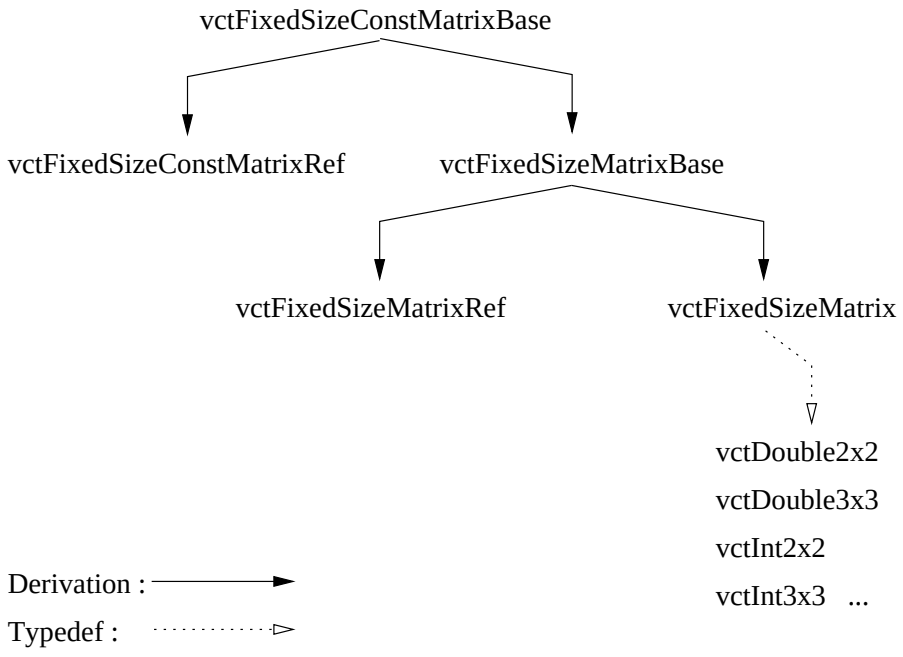


Figure 2.3: Class family for fixed size matrices

2.2 Matrices

The matrix class families were designed under similar guiding principles to the vector class families. There is a main division between fixed-size and dynamic matrix classes, which are identified by the beginning of the class name: `vctFixedSize` and `vctDynamic`. In each family, there are two base classes, with the names ending in `ConstMatrixBase` and `MatrixBase`, which define the immutable and mutable operations on matrix objects. From these we derive API classes, which include allocating classes (`Matrix`) and overlay classes (`ConstMatrixRef` and `MatrixRef`). An example illustration of the class hierarchy for fixed-size matrices is in Figure 2.3

This section goes briefly through the matrix class families, and give more details about the unique features of matrix objects, compared to vector objects. Matrices are more complicated objects than vectors, and thus they do include some additional features.

2.2.1 Fixed-size matrices

vctFixedSizeMatrix

The most accessible class in the fixed-size matrix family is `vctFixedSizeMatrix`. It is declared as a class templated by the type of matrix element, the number of rows, the number of columns, and the storage order:

```
template<class _elementType, unsigned int _rows, unsigned int _cols,
    bool _rowMajor>
class vctFixedSizeMatrix;
```

Like fixed-size vectors, there are simple names for fixed-size matrices of sizes 2×2 to 4×4, with all the combinations in between. The "default" element type is double, and the library defines short names for matrices of double elements. For example

```
typedef vctFixedSizeMatrix<char, 2, 2> vctChar2x2;
typedef vctFixedSizeMatrix<int, 3, 4> vctInt3x4;
typedef vctFixedSizeMatrix<float, 4, 2> vctFloat4x2;
typedef vctFixedSizeMatrix<double, 3, 3> vctDouble3x3;
typedef vctFixedSizeMatrix<double, 3, 3> vct3x3;
```

We will use the simple names and the short names occasionally in this document.

Storage order The elements of a `vctFixedSizeMatrix` objects are allocated in a fixed-size array of size `_rows * _cols`. A row-major storage order means that the elements are stored row by row, each row occupying a continuous memory block. Column-major storage means that the elements are stored column by column. Normally in C and C++ code, two-dimensional arrays are assumed to be stored in row-major order; but in languages such as Fortran, and in packages imported from Fortran, such as C-LAPACK, the storage order is column-major. Our library enables to define both orders. The default storage order is row-major, identified by the constant `VCT_ROW_MAJOR`. Column-major storage is identified by the constant `VCT_COL_MAJOR`. The two are defined in the file vctForwardDeclarations.h.

Note here that *all the elements* of a `vctFixedSizeMatrix` lie in one continuous memory block, regardless of the storage order. We call this memory configuration *compact* storage. We will come back to this definition later.

Accessors The matrix classes include accessor methods equivalent to the accessors in the vector classes, except that the matrix accessors usually take to indexes to identify an element of the matrix. The method `Pointer` simply takes two parameters, for the row and the column, as we saw in the examples in the previous section. The overloaded operator `[]` takes only one index, and returns an overlay vector (`vctFixedSizeVectorRef` or `vctFixedSizeConstVectorRef`) for a row of the matrix. To access an element using `[]` notation, use two pairs of brackets. The library also includes the named method `Element` that takes two indexes. One may find it potentially more efficient, as it does not create an overlaid vector object. Here are some accessor examples.

```
vct3x3 m;
// Access the 3rd element of the 2nd row through the Pointer
// method
```

```
*(m.Pointer(1, 2)) = 5.0;
// Assign zeros to the 1st row
m[0] = vct3(0.0);
// Access the 1st element of the 3rd row through []
double x = m[2][0];
// Access an element through the method Element
m.Element(2, 2) = 6.25;
```

Note that the order of indexing is always row-first, following the C/C++ convention regardless of the actual storage order. This is made possible in the case of `operator []` by setting the stride of the overlay row vector. More on this is in the part about matrix slices.

Matrix operations Matrices provide the same set of elementwise operations as vectors: addition, subtraction, elementwise multiplication and division, sum of elements, and so on. Operations can be performed between matrices of equal sizes, and between matrices and scalars. The definition of such operations is straightforward and similar to the definition for vectors. A list of operations is given in Chapter 3. Here we would like to highlight products of matrices by matrices and matrices by vectors.

Matrix and vector products A product involving a matrix and a vector can be a *matrix-vector product*, i.e., $\mathbf{M} \cdot \mathbf{v}$, or a *vector-matrix product*, i.e., $\mathbf{v}^T \cdot \mathbf{M}$. In this notation, $\mathbf{v}$ is always a column vector. In the first case, the number of columns in $\mathbf{M}$ must be equal to the number of elements in $\mathbf{v}$, and the result is a vector whose size is the number of rows in $\mathbf{M}$. In the second case, the number of elements in $\mathbf{v}$ must be equal to the number of rows in $\mathbf{M}$, and the result is a row vector whose size is equal to the number of columns in $\mathbf{M}$. A matrix-matrix product, $\mathbf{M}_1 \cdot \mathbf{M}_2$ requires that the number of columns in $\mathbf{M}_1$ to be equal to the number of columns in $\mathbf{M}_2$, and the result is a matrix whose number of rows is the same as $\mathbf{M}_1$'s, and number of columns is equal to $\mathbf{M}_2$'s.

An important advantage of the fixed-size matrix family is that the sizes of a matrix are defined in its type. Therefore, it is possible to check for matching sizes in the matrix product function declaration, and fail to generate code in a case of mismatch. Although compilation errors may be more difficult to decypher, this can serve as an important safety mechanism against programming errors.

The matrix-vector, vector-matrix, and matrix-matrix product operations are defined through methods working on the target object. For example,

```
// compute matrix-matrix product
vct3x4 m1;
vct3x3 m2;
vct3x4 m3;
m3.ProductOf(m2, m1);
```

```
// compute matrix-vector product
vct4 v1;
vct3 v2;
v2.ProductOf(m1, v1);
// compute vector-matrix product
vct3 v3;
vct4 v4;
v4.ProductOf(v3, m1);
```

In addition, the library includes an overloaded `operator *` which can take matrix-matrix, matrix-vector and vector-matrix operands. As usual, we generally encourage using the named methods to avoid the creation of temporary objects. In the case of long matrix-matrix products, the order of applying the operations can have a significant impact on the performance, and should be analyzed carefully.

Overlay matrix classes

Like the vector class families, the matrix classes also define overlay matrix types. For the fixed-size matrices, these are `vctFixedSizeMatrixRef` and `vctFixedSizeConstVectorRef`. As we have explored overlay classes before, we will focus here on the non-const overlay.

vctFixedSizeMatrixRef The class declaration is

```
template<class _elementType, unsigned int _rows,
    unsigned int _cols, int _rowStride, int _colStride>
class vctFixedSizeMatrixRef;
```

The declaration includes two size template parameters and two stride template parameters. For clarity, we will note here that we define the row-stride as *a stride of one row*, or the stride between rows, that is, the stride between adjacent elements in a column of the matrix. The column-stride is *a stride of one column*, or the stride between columns, i.e., between adjacent elements in a row. For example, in a 4×6 matrix stored in row-major order, the row stride is 6, and the column stride is 1. If the same matrix is stored in column-major order, the row stride is 1 and the column stride is 4.

Recall that `vctFixedSizeMatrix` used a compact memory layout, i.e., all the elements were in one continuous memory block. This is not necessarily the case with an overlay matrix. If we take columns 2 to 5 in the 4×6 matrix (that is, the third to sixth columns), we can overlay them with the following code:

```
// Sorry, no simple name defined here
vctFixedSizeMatrix<int, 4, 6> intMatrix;
// This is the overlay: 4 rows, 4 columns,
```

```
// row stride is 6 (number of row elements
// in intMatrix), column stride is 1 (row-major
// storage)
vctFixedSizeMatrixRef<int, 4, 4, 6, 1>
    submatrix( intMatrix.Pointer(0, 2);
```

From here on, any operation on `submatrix` is equivalent to an operation on `intMatrix` in the corresponding columns.

Note that this definition is very powerful. In the general case, it allows to overlay a non-contiguous set of rows and columns, as long as the differences between them are always the same. Normally, however, a user would wish to overlay a contiguous sub-block in a matrix. In order to simplify the overlay process, *ALL* the fixed-size matrix classes define internal enums named `ROWSTRIDE` and `COLSTRIDE`, which hold the values of the row and column strides. The simpler way to define an overlay matrix is therefore:

```
// Define the type of the matrix you use
typedef vctFixedSizeMatrix<int, 4, 6> MatrixType;
// Define a matrix object
MatrixType mat;
// Define a submatrix variable using the parent container
// definitions
vctFixedSizeMatrixRef<int, 4, 4, MatrixType::ROWSTRIDE,
    MatrixType::COLSTRIDE> submatrix( mat.Pointer(0, 2) );
```

While this definition appears longer, it is much more readable than the previous, and much safer. It is harder to be confused about which template parameter is which, and the modularity is improved, because the submatrix is defined in terms of the parent container.

More on the subject of accessing submatrices is in Section 2.3.

Matrix base classes

As we have mentioned before, matrix class families have a similar structure to vector class families. For fixed-size matrices, there are two base classes: `vctFixedSizeConstMatrixBase`, which defines defines defines immutable operations, and `vctFixedSizeMatrixBase`, which adds the mutable operations. Here we will discuss their definitions briefly. The principles of this design are essentially the same as for the fixed-size vectors.

At the root of the fixed-size matrix class tree is `vctFixedSizeConstMatrixBase`, declared as follows.

```
template<unsigned int _rows, unsigned int _cols, int _rowStride,
         int _colStride, class _elementType, class _dataPtrType>
class vctFixedSizeConstMatrixBase;
```

The template parameters include the sizes of the matrix, the strides, the element type, and a _dataPtrType. The last element may be a reference pointer or an allocated array, as will be described below. The class has one data member:

```
_dataPtrType Data;
```

which is specialized by the derived classes. More details about it are in 2.1.1.

Accessor methods are defined at this level for immutable access:

```
// return a pointer to an element of the matrix indexed by
// row and column
const_pointer
Pointer(size_type rowIndex, size_type colIndex) const
{
    return Data + ROWSTRIDE * rowIndex + COLSTRIDE * colIndex;
}
// return a vector overlay for a row of the matrix, identified
// by index
ConstRowRefType operator[](size_type index) const
{
    return ConstRowRefType(Data + ROWSTRIDE * index);
}
// return a reference to an element of the matrix indexed
// by row and column
const_reference
Element(size_type rowIndex, size_type colIndex) const
{
    return *(Pointer(rowIndex, colIndex));
}
```

In this example, we introduce a new object type: ConstRowRefType. This is just an alias for an overlay vector type, and will be explained in 2.3. Recall that we have already suggested to use the named method Element instead of the overloaded operator []. Here we see why it may be more efficient. Because the column stride of the matrix may be different than 1, the overloaded operator must return a correct overlay vector type as an object, and the second index resolution will be done by it. While many compilers may generate inlined code for this, it is not guaranteed, and in the case of dynamic matrices, it is less likely to happen.

As with the vector families, vctFixedSizeMatrixBase extends vctFixedSizeConstMatrix-Base by adding the mutable operations. vctixedSizeMatrixRef specializes vctFixedSize-ConstMatrixBase by defining _dataPtrType to be _elementType *, thus creating a const overlay vector. vctFixedSizeMatrixRef extends vctFixedSizeMatrixBase similarly, defining a mutable overlay vector. And vctFixedSizeMatrix extends vctFixedSizeMatrixBase by defining dataPtrType to be _elementType[_rows * _cols], thus creating an allocating class.

2.2.2 Dynamic matrices

By now, we have covered most of the background material for the understanding of the dynamic matrix classes in cisstVector. In previous sections we discussed allocating vs. overlay objects, mutable and immutable classes, strides, owners, storage order, and return objects. There is very little to add to that regarding dynamic matrices. We therefore refer the reader to the previous sections of this guide for detailed information. Here we will just outline the class hierarchy for dynamic matrices, as we did for other class families.

At the root of the family is `vctDynamicConstMatrixBase`, which defines immutabe operations on dynamic matrices. Its declaration is

```
template<class _matrixOwnerType, typename _elementType>
class vctDynamicConstMatrixBase;
```

As with dynamic vectors, the owner defines the memory allocation method, accessors, and iterators. There are two types of owners: `vctDynamicMatrixOwner` and `vctDynamicMatrixRefOwner`. The first is a memory allocating owner, and the second is an overlay owner. The dimensions of the matrix are defined through function parameters in the specialized derived classes.

Next in the hierarchy is `vctDynamicMatrixBase`, which extends `vctDynamicConstMatrixBase` by adding mutable operations. It takes the same template parameters as its base class. Following are the overlay classes `vctDynamicConstMatrixRef` and `vctDynamicMatrixRef`, specializing `vctDynamicConstMatrixBase` and `vctDynamicMatrixBase` by using a `vctDynamicMatrixRefOwner`. They can be set to overlay specific memory locations in constructors or `SetRef` methods, which take the sizes and strides as parameters.

Finally comes `vctDynamicMatrix` which allocates memory. Here the user specifies the sizes and storage order in function parameters, and these define the strides. The memory allocated for a `vctDynamicMatrix` objectis always compact, i.e., one of the strides is always 1. `vctDynamicMatrix` is inherited by `vctReturnDynamicMatrix` which is specialized for avoiding reallocation of memory for function return values.

2.3 Matrix and vector slices

When dealing with matrices, one often wants to refer to rows or columns in the matrix as vector objects. As we have seen, our overlay vector classes allow this type of access. Some examples were given in Section 2.2, using the overloaded `operator []`. Others were given in the discussion of overlay vectors. Here, we would like to provide a more systematic and convenient guide for accessing substructures in vector and matrix objects known as *slices*. Our definition of slices will be any vector or matrix overlay with regular intervals, or strides, between the elements. Examples include row and column vectors in a matrix, block submatrices, and subvectors with regular strides. The examples shown here will include mostly fixed-size vectors and matrices, and a few dynamic matrix examples.

2.3.1 New definitions for row and column subvectors

Recall that our first examples of overlay row and column vectors in a matrix required a specification of the stride based on the matrix dimensions:

```
typedef vctDouble3x3 RotationMatrixType;
RotationMatrixType rot;
// Explicit definition of a column vector
vctFixedSizeVectorRef<double, 3, RotationMatrixType::ROWSTRIDE>
    rotColumn0( rot.Pointer(0, 0) );
// A slight shortcut
vctFixedSizeVectorRef<double, 3, RotationMatrixType::ROWSTRIDE>
    rotColumn1( rot.Column(1) );
```

Since overlaying matrix rows and columns with vectors is a very ubiquitous operation, we simplified the process by having the matrix base classes define internal types for row and column overlays. The following definitions are for fixed-size matrices, taken from `vctFixedSizeConstMatrixBase`. Similar definitions exist for the dynamic matrices, with the exception that sizes and strides are not included in the type definition.

```
/*! The type indicating a row of this matrix accessed by (const)
  reference */
typedef vctFixedSizeConstVectorRef<_elementType, COLS, COLSTRIDE>
    ConstRowRefType;
/*! The type indicating a row of this matrix accessed by
  (non-const) reference */
typedef vctFixedSizeVectorRef<_elementType, COLS, COLSTRIDE>
    RowRefType;
/*! The type indicating a column of this matrix accessed by (const)
  reference */
typedef vctFixedSizeConstVectorRef<_elementType, ROWS, ROWSTRIDE>
    ConstColumnRefType;
/*! The type indicating a column of this matrix accessed by
  (non-const) reference */
typedef vctFixedSizeVectorRef<_elementType, ROWS, ROWSTRIDE>
    ColumnRefType;
```

Now, to overlay a row or a column, one only needs to write

```
MatrixType m;
MatrixType::RowRefType rowVec = m.Row(rowIndex);
MatrixType::ColumnRefType colVec = m.Column(colIndex);
```

This works just as well for dynamic matrices as it does fo fixed-size. Indeed, nothing in this code indicates what `MatrixType` may be.

2.3.2 Defining submatrix and subvector types

In Section 2.2 we presented a somewhat elaborate way of defining submatrices. Like row and column vectors, submatrices are fairly ubiquitous, and we want to have a convenient way of defining them.

For fixed-size matrices, we want to specify the sizes of the submatrix as template parameters. The only way to do that is through the definition of templated internal classes. Here's an example taken from the class definition of `vctFixedSizeConstMatrixBase`.

```
template<unsigned int _subRows, unsigned int _subCols>
class ConstSubmatrix
{
public:
    typedef vctFixedSizeConstMatrixRef<value_type, _subRows,
        _subCols, ROWSTRIDE, COLSTRIDE> Type;
};
```

The internal submatrix type uses the parent container's strides for its own. To define a submatrix object, the code is [1]

```
typedef vct4x4 MatrixType;
MatrixType m;
MatrixType::ConstSubmatrix<3,3>::Type topLeft(m, 0, 0);
```

In this example, we use a constructor of `vctFixedSizeConstMatrixRef` that takes a matrix object and the indexes of the first element of the submatrix rather than the examples shown before, which were initialized with an actual pointer. This can be seen as more convenient, especially for dynamic matrices, where the stride values are obtained from the parent matrix object instad of passed as parameters:

```
typedef vctDynamicMatrix<double> DoubleMatrixType;
DoubleMatrixType m(20, 12);
// define a submatrix of size 6x6 at the bottom-left block
// of m
DoubleMatrixType::Submatrix::Type bottomLeft(m, 14, 0, 6, 6);
```

Similar internal definitions were done for creating subvectors. For example, `vctFixedSize-VectorBase` defines the following

```
template<unsigned int _subSize>
class Subvector {
```

[1] Some compilers, e.g., Microsoft .NET, cannot digest the complexity of the definition of `topLeft`, and require explicit type definitions. We will disregard this here.

```
public:
    typedef vctFixedSizeVectorRef<value_type, _subSize, STRIDE> Type;
};
```

And here's a corresponding usage example

```
typedef vctFixedSizeVector<int, 9> LongVectorType;
typedef LongVectorType::Subvector<3>::Type SubvectorType;
LongVectorType v9;
// create a subvector occupying element 3 to 5
SubvectorType middleBatch(v9, 3);
```

2.3.3 A few more tricks

We conclude this section with a few programming tricks based on slice definitions. They are presened as short examples with short explanations.

Matrix transpose

All the matrix types define internal overlay types named `RefTransposeType` and `ConstRefTransposeType`. For fixed-size matrices, they are defined by swapping the row and column sizes, and the row and column strides, so that rows of the parent matrix become columns of the overlay, and vice versa. In dynamic matrices this needs to be done in runtime. We provide the method `TransposeRef` to initialize a const or non-const overlay transposed matrix on an existing matrix object.

Turning vectors into matrices

Turning vectors into matrices is a useful feature. One example is when one wants to compute an outer product of two vectors. This can be done by creating an overlay matrix of one vector as a single-column matrix, and the other as a single-row matrix. We provide the methods `AsColumnMatrix` and `AsRowMatrix` in the vector families for this purpose. Otherwise, one can simply define an overlay as follows.

```
vctDynamicVector<double> parametersVector(20);
vctDynamicMatrixRef<double>
    parametersMatrix(4, 5, // rows and columns
        5, 1, // row stride and column stride
        parametersVector.Pointer() );
```

Turning matrices into vectors

When a matrix object is compact, i.e., all of its elements are in one continuous memory block, it can be overlaid with a vector. For example,

```
vctDouble4x4 m;
vctFixedSizeVectorRef<double, 16, 1> elements(m.Pointer());
```

Reverse order

A reverse order overlay can be defined using negative strides.

```
vctInt6 forwardVec;
vctFixedSizeVectorRef<int, 6, -1>
    reverseVec(forwardVec.Pointer(5));
```

2.4 Transformations

Transformations are an essential element in working with spatial geometry, as opposed to abstract linear algebra operations. The basis for most transformations can be rooted in the algebra, but their semantics is more specialized than the framework of general vectors and matrices. We will explore some of it in this section.

The cisstVector library defines transformation objects for 2D and 3D geometry. It focuses on the family of rigid transformations, which is of special interest in applications such as robotics, computer graphics and computer vision. Rigid transformations include two essential elements: translations and rotations. Those are combined in structures called *frames*. Rotations can be represented in multiple ways, and the cisstVector library is designed to be scalable in the direction of rotation representations. The collection of rotation classes, and their incorporation into frames, are the main topics of this section.

2.4.1 Rotation representations

Rotations in two and three dimensional spaces can be represented in a large variety of ways. Some representations for two dimensions are: rotation angle, 2×2 orthogonal rotation matrix, and unit-length complex numbers. For three dimensional rotations we can mention: axis-angle pair, Euler angles (in various order combinations), 3×3 orthogonal rotation matrix, unit quaternions, and Rodriguez vector, which encodes the axis of rotation in its direction and the angle in its magnitude.

In the cisstVector library, we define a class for each representation method we use. Thus, for 2D rotations, we have classes such as `vctMatrixRotation2` and `vctAngleRotation2`.

For 3D rotations, we have classes such as `vctMatrixRotation3`, `vctQuaternionRotation3`, `vctAxisAngleRotation3`, and `vctRodriguezRotation3`. These lists may grow if we choose to add new representations; but the name of a rotation class will always have the format `vct`[Rep]`Rotation`#, where [Rep] is replaced by a name of the representation, and # indicates the dimension of the space where the rotation applies.

In spite of this large variety, the number of representations which are useful in an algebraic framework is comparatively small. By this we mean that operations such as composing, inverting, and applying to vectors are difficult to implement. Euler angles are an extreme example, as the inverse of a set of angles is not a simple function of the set (if we want to preserve the order), and so is composition of rotations. In axis-angle representation, the inverse is easier to compute (negate the angle or the axis), but composition is still difficult. Because of this, some rotation representations have been completely developed, that is, we provide a complete set of operations compatibe with the interface we defined, while others have only been partially developed. However, we attempted to provide at least a complete conversion framework between representations, that is, there is a way to convert any representation to any other representation in the library.

Several rotation representations have to be normalized for correct computational results. For example, rotation matrices must be orthogonal with a positive unit determinant; rotation quaternions must have unit norm; and the axis in axis-angle rotation must have unit length. We will discuss the normalization further in 2.4.2.

Listing rotation class names

A listing of rotation class names is expected to change from time to time due to the introduction or deprecation of representations. The current list is given in Table 2.2, and includes class names, templated by `_elementType`, e.g.,

```
template <class _elementType> class vctMatrixRotation3;
```

Normally, they are specialized by defining `_elementType` as a floating-point type, i.e., float or double. The "default" specialization uses double as the element type, and was given a short name for ease of use. We will occasionally use the short name in the examples given in this guide.

2.4.2 Representation conversions and normalization

Due to the increasing number of rotation representations, it is useful to define a unified syntactic structure for converting one representation to another. A complete graph of conversion, containing methods to convert each type `A` to type `B` and vice versa may be desirable, but is also highly complex and not necessarily efficient. Instead, we assume it's sufficient to have a connected conversion graph, that is, there is a path of conversions between each pair

Class name	Short name
vctAngleRotation2	vctAnRot2
vctAxisAngleRotation3	vctAxAnRot3
vctMatrixRotation2	vctMatRot2
vctMatrixRotation3	vctMatRot3
vctQuaternionRotation3	vctQuatRot3
vctRodriguezRotation3	vctRodRot3

Table 2.2: Rotation classes in the cisstVector library

of types, but it may involve creating intermediate objects that cen be converted to the new type. For example, if we had an Euler angles representation and we wished to convert it to axis-angle pair, we might have had to convert it first to a rotation matrix, and then convert the matrix to axis-angle.

Following the policy of writing mutating operations as methods on the mutated target object, all the rotation classes define a collection of method named `From`, which convert their input into the representation of, and store the result in the target object. For example,

```
// initialize an axis-angle rotation object by specifying the
// axis as a vector and the angle in radians
const double rootHalf = sqrt(0.5);
vctAxAnRot3 axAnRot( vct3(rootHalf, rootHalf, 0.0), PI / 3.0 );
// create a quaternion rotation initialized as the identity
// transformation
vctQuatRot3 quatRot = vctQuatRot3::Identity();
// convert the axis-angle to quaternion
quatRot.From( axAnRot );
// create a matrix rotation initialized with the axis-angle
// rotation.
vctMatRot3 matrixRot( axAnRot );
```

In the example, we used the class static method `Identity` to refer to the identity transformation, i.e., a function that maps each value to itself. All the rotation classes define a static method by this name, and all the rotation objects are initialized by default to the identity transformation. The example shown is therefore redundant syntactically, yet useful for presenting the subject.

By default, the `From` operation requires a normalized input object, and will throw an exception or crash due to unsatisfied `assert` if the input is not normalized. All the rotation classes define the following methods related to normalization:

- `IsNormalized(value_type tolerance)` – returns `true` if the data is normalized up to the given tolerance.

- `NormalizedOf(const ThisType & other)` – sets the target object to a normalized version of the input.
- `NormalizedSelf()` – normalizes the target object and stores the result back in it.
- `Normalized()` – returns a normalized version of the target object as a new object.

In some cases, the library user may know that the data to convert is already normalized and want to save the time spent on testing or renormalization. In other cases the user may wish to assign a rotation object with values that are not normalized to the current tolerance, e.g., when the elements of a rotation matrix are computed from Euler angles, and only when they are stored as a matrix they can be normalized. The rotation classes define methods named `FromRaw` to handle unnormalized data. Naturally, they must be used with care.

2.4.3 Applicable rotation classes

As we noted above, not all the rotation representations are applicable in an efficient algebraic structure. We have chosen a subset of the representations and developed their methods so that they can be composed and applied to vectors.

For two dimensional rotations, we currently provide one applicable class, which is `vctMatrixRotation2`. For three dimensional rotations we currently have two applicable classes: `vctMatrixRotation3` and `vctQuaternionRotation3`. Each of the rotation classes is derived from a more general library class: the `MatrixRotation` classes from matrices, and the `QuaternionRotation` class from quaternion (quaternions will be presented in Section 2.5)[2]. Therefore, they inherit all the operations which are defined for the base types. For example, in most cases, a `vctMatrixRotation3` object can be treated as a 3-by-3 matrix without any conversion. The sizes of the matrices are 2×2 in 2D rotations and 3×3 for 3D rotations. The quaternion always has four elements.

Any applicable rotation class defines the following methods:

- `Inverse()` – return a new object which is the inverse of the target object.
- `InverseSelf()` – store the inverse of the current rotation in the target object.
- `InverseOf(const ThisType & otherRotation)` – store the inverse of `otherRotation` in the target object. NOTE: Classes in the cisst package define `ThisType` internally as the type of the class.
- `ApplyTo(const Vector & input, Vector & output)` – apply the rotation of the target object to the input vector, and store the result in the output vector. NOTE: the input and output vector types are templated.

[2]The actual derivation of the rotation classes involves yet another level of abstraction, which is spared from the readers of this guide.

- `ApplyTo(const Vector & input)` – return a new vector which results from applying the rotation in the target object to the input vector. NOTE: the input vector type is templated.
- `ApplyTo(const ThisType & input, const ThisType & output)` – compose the target object on the left-side of the input rotation, and store the result in the output rotation object.
- `ApplyTo(const ThisType & input)` – compose the target object on the left-side of the input rotation, and return the result as a new object.
- `ApplyInverseTo( /* same types as the above examples */ )` – apply the inverse of the target object to the input object, and store the result in the output object or return a new object. The semantics is the same as the previous methods.
- `operator *` overloaded for composing rotations of the same type.

Note that the operations between rotation objects are restricted to rotation objects of the same type. We do not provide a direct composition of a matrix rotation over a quaternion rotation, for example. The methods that apply rotation to a vector are templated by the vector type, which enables, for example, to read the input from, or write the result to, an overlay vector. The template notation is not shown in this list for simplicity reasons.

The syntactic structure of applying rotations differs from our previously stated convention of mutating the target object through its methods. Here we see immutable methods on the target object, which write to a mutable argument object. We created this exception in order not to overload the vector classes with all the possible routines related to transformations. For once, our class structure is already complicated enough, and we did not want to introduce more dependencies in it.

Having defined rotations, we can now turn to frames.

2.4.4 Frames

Frame is our synonym to a rigid transformation, that is, one that includes rotation and translation. We already discussed rotation classes and described their generic interface in the cisstVector library. Our frame classes are all specialized from one base class, `vctFrameBase`, declared as follows:

```
template<class _rotationType> class vctFrameBase;
```

The template parameter `_rotationType` is a name of an applicable rotation class, e.g., `vctMatrixRotation3` or `vctQuaternionRotation3`. `vctFrameBase` obtains type-related information from the rotation class, such as element type and the dimension of the space.

For practical use, the library defines several types with short names as specializations of the generic frame base. For example,

```
typedef vctFrameBase<vctQuaternionRotation3<double> > vctQuatFrm3;
typedef vctFrameBase<vctMatrixRotation3<double> > vctMatFrm3;
typedef vctFrameBase<vctMatrixRotation3<double> > vctFrm3;
typedef vctFrameBase<vctMatrixRotation3<float> > vctFloatMatFrm3;
```

A frame object enables access to two members through accessor methods: `Rotation()`, which is of type `_rotationType`, and `Translation()`, which is a vector. Each member can be manipulated through the methods of its class.

Frames provide a similar application interface as rotations, with methods such as `InverseOf`, `ApplyTo`, `ApplyInverseTo`, etc. Note that the implementation of these methods depends on the definition of methods with similar names in the rotation parameter class. Therefore, only applicable rotation classes can be used to specialize a frame. As we said, the library defines specializations for most of them.

2.5 Other classes

The library contains a few specializations of the vector classes. Currently these include barycentric vectors and quaternions. The are presented briefly here.

2.5.1 Barycentric vectors

Barycentric vectors can be viewed as a redundant representation of vectors. We define a barycentric vector of size n as an n-element vector whose sum of elements is 1. Some works require that the elements be non-negative as well, but we allow more flexibility on this issue. Barycentric vectors are often used as a parametrization of the n-dimensional space with respect to an n-dimensional simplex, such as a line segment, a triangle, or a tetrahedron.

The library defines the templated class

```
template<class _elementType, unsigned int _size>
class vctBarycentricVector
    : public vctFixedSizeVector<_elementType, _size>
{ /* ... */ };
```

A `vctBarycentricVector` object inherits all the methods in `vctFixedSizeVector`, and extends them with methods for testing the status of the barycentric coordinates. Example methods include:

- `IsBarycentric(value_type tolerance)` – tests if the sum of the elements is 1, up to the given tolerance.
- `IsInterior(value_type tolerance)` – tests if the vector represents an *interior point* of the simplex, which is if all its elements are strictly positive, above the given tolerance.

- `IsVertex(value_type tolerance)` – tests if the vector represents a vertex of the simplex, which is if one of its elements is equal to 1, and the sum of elements is 1, up to the given tolerance.

Because the library allows direct access to the elements of a vector, there is no way to enforce the constraint on the sum of elements. Instead, we provide the `IsBarycentric` method to test the constraint, and the method `ScaleToBarycentric`, which performs "normalization" of the vector to satisfy the constraint.

Note that besides this API extension, `vctBarycentricVector` does not truly define a new concept. Rather, it specializes an existing vector class and adds interfaces that are not obvious or even necessary in an ordinary vector.

2.5.2 Quaternions

Quaternions are an algebraic group of four-element tuples, which can be viewed as combinations of vectors and scalars. We define the structure of a quaternion as (x, y, z, r), which represents the value

$$\mathbf{q} = r + xi + yj + zk$$

where i, j, and k are imaginary units with the multiplication rules:

$$\begin{aligned} ij &= k \\ jk &= i \\ ki &= j \\ i^2 = j^2 = k^2 &= -1 \end{aligned}$$

x, y and z are the *imaginary components* of the quaternion, and r is the *real component*. Like some libraries, and unlike the conventional mathematical notation, we store the real component of a quaternion last instead of first. This allows convenient access to the imaginary components through the named methods `X()`, `Y()`, and `Z()`.

the quaternion class in cisstVector is declared as

```
template <class _elementType> class vctQuaternion;
```

and is derived from `vctFixedSizeVector` of size 4.

Specific operations on quaternions include: a redefinition of multiplication, which overrides "vector product" (i.e., dot product); definition of a conjugate: $\hat{\mathbf{q}} = r - xi - yj - zx$; definition of an inverse (which does not exist for vectors); and a resulting definition of division as multiplication by inverse. Example methods include:

- `ConjugateOf(const ThisType & otherQuaternion)` – compute the conjugate of the other quaternion and store in this quaternion.

- `PreMultiply(const ThisType & other)` – multiply the input quaternion *on the left* of this quaternion, and store the result in this quaternion. Quaternion product is non-commutative.
- `QuotientOf(const ThisType & quat1, const ThisType & quat2)` – divide quat1 by quat2 and store the result in this quaternion.

In addition, unit quaternions, that is, quaternions whose norm is 1, can be used to represent rotations.

Chapter 3

Method and function interfaces

This chapter outlines the main functional API of the cisstVector library. It starts with a high-level review of the common taxonomy of library methods, and follows with a list of actual methods which are common in all classes. Methods that are specific to certain classes are presented in the end.

The notation used for explaining the methods is semi-formal. Almost all the methods that take arguments are templated or template-related, and often the use of full template notation obscures the explanation of the methods. Instead of providing the templated form, we will use the following short notation.

1. All the variable names are in lowercase letters, with underscore separating "words".
2. (a) Input variable names end with `in` or with `in#`, where `#` is an index number.
   (b) Output variable names end with `out`. Typically, there is only one output variable, but if there are more, they will be indexed.
   (c) Variables which are both input and output end with `io`.
3. (a) Vector variables begin with `v`.
   (b) Matrix variables begin with `m`.
   (c) Scalar variables begin with `s`.
   (d) Index variables, which are typically integers, start with `j`.
   (e) Iterator variables begin with `i`.
   (f) Vector or matrix elements are denoted with `e[j]` or `e[j][k]`.
   (g) A generic object (i.e., vector *or* matrix) is denoted with `t`.
   (h) Pointers to objects begin with `p` followed by the "type letter" of the object.
   (i) References begin with `r` followed by the type letter of the object.

(j) The *type* of the target object (`*this` – the one invoking the method) is denoted by `ThisType`.

4. Function return values are written down as variables, even though they may be temporary objects. If the return value is a reference or a pointer to one of the input variables, it is denoted by preceding `r` or `p` to the input variable's name.

Principles and conventions

Before listing the methods, we want to present some principles and conventions that can help in locating them and understanding their semantics.

1. As mentioned in Chapter 2, we are encouraging the use of named methods, as opposed to overloaded operators, due to a smaller number of potential side effects. Specifically, we attempt to provide a full collection of methods that operate on their input only without returning temporary objects.

2. Overloaded operators are provided as an extension to the named method interfaces, and typically consist of a function call to a named method. This means that the overloaded operators cover a *subset* of the available named methods.

3. Typically, the name of a method includes

   - a verb when the target object is modified based on its initial state. For example, the method `Add` will add a new vector or matrix to the target object.
   - a noun when the target object is set to something new, ignoring its initial state. For example, the method `SumOf` will set the vector or matrix as the sum of two other vectors or matrices.

   However, this is a general rule, with exceptions when convenience requires it. For example, the method `t_out.Assign(t_in)` is a verb, but it just assigns a value to the target object, and by the general rule should have been a noun.

3.1 Class-scope types

As presented in Chapter cap:class-families, the cisstVector classes often define class-scope type to facilitate generic interfaces. Here, we would like to present the common collection of these types. Note that classes may define additional types, but this list is common to all the cisstVector container classes.

- **Template independent types**. These types are defined regardless of the values of the container template arguments, and typically relate to size parametrizations. They include the following.

Type name	Description	Defined as	In STL
`size_type`	The number of elements of a vector, or the number of rows or columns of a matrix.	`unsigned int`	Yes
`index_type`	An index of an element in a vector, or a row or a column in a matrix.	`unsigned int`	Yes
`stride_type`	The stride between elements of a vector, or between rows or columns of a matrix.	`int`	No
`NormType`	The norm (e.g., length of a vector).	`double`	No
`AngleType`	An angle in radians (e.g., between two vectors).	`double`	No

- **Element dependent types**. These types depend on the `_elementType` template argument, and define types related to it. They include the following.

Type name	Description	Defined as	In STL
`value_type`	The type of the element in the container	`_elementType`	Yes
`const_reference`	A constant reference to an element in the container	`const _elementType &`	Yes
`const_pointer`	A pointer to a const element in the container	`const _elementType *`	No
`reference`	A non-constant reference to an element in the container	`_elementType &`	Yes
`pointer`	A pointer to a non-const element in the container	`_elementType *`	No

- **Iterators**. The container iterators are objects which can iterate over all the elements of the container in a sequential order. The iterator types are defined according to the STL convention, and include: `iterator`, `const_iterator`, `reverse_iterator`, and `const_reverse_iterator`. For an explanation of these, please refer to STL documentation. Here we will only note that the cisstVector iterators are self-contained objects which include knowledge about the strides and matrix dimensions of the container over which they iterate.

- **Class hierarchy**. For convenience, cisstVector classes define the local type `ThisType` to refer to their own type. This is often useful when the templated signature of the class name is long. Likewise, the classes define the local type `BaseType` to refer to their base class.

3.2 Common interfaces

All the operations described in this section are applicable for vectors and matrices. Most of the examples will be given for vectors, and it is assumed that the extension to matrices is natural.

3.2.1 Constructors

The non-API-level base classes in the vector and matrix families have no constructors, as they are not supposed to be instantiated directly. These are: `vctFixedSizeConstVectorBase`, `vctFixedSizeVectorBase`, `vctDynamicConstVectorBase`, `vctDynamicConstVectorBase`, `vctFixedSizeConstMatrixBase`, `vctFixedSizeMatrixBase`, `vctDynamicConstMatrixBase`, `vctDynamicMatrixBase`.

All the API-level classes have default constructors. As for parametrized constructors, dynamic containers always require passing a specification of the memory layout. This includes dimensions (size, rows, columns), strides, storage order, etc. Such parameters are not used in fixed-size containers, as they are given as part of the object type. Overlay containers may be initialized with the memory address of the overlaid data. Allocating containers may be initialized with element values or as copies of other containers. These constructor signatures are described next.

Default constructors

All the cisstVector API-level classes have a default constructor, i.e., a constructor that takes no parameters. For fixed-size objects, it leaves the new object uninitialized, that is, the object can contain any values, including illegal ones. Dynamic objects are initialized by default as empty containers with null buffers through the default constructors of their *owners*.

Copy constructors

Copy constructors copy values from one container to the target container. The conventional C++ copy constructor creates a "clone" of the input object, and has the following signature.

```
class C
{
public:
    C(const C & source);
};
```

In the cisstVector library, the concept of a copy constructor is modified. Its rationale is of creating a *replica* of structured data, not a clone of an object. In other words, cisstVector

copy constructors replicate the data elements of their input objects, which is not necessarily identical to replicating the input objects themselves. The straightforward example is when the input object is an overlay container, e.g., an overlay vector, and the target object is an allocating container:

```
vct3x3 matrix;
vct3 columnVector( matrix.Column(2) );
```

The `Column` method in this example returns a temporary overlay object of type `vctFixedSizeVectorRef</*...*/>`. However, `columnVector` is initialized with elements copied from the last column in `matrix`, and not with the pointer, which is the only data member of the temporary object.

The replication semantics implies that only allocating containers define copy constructors. These include `vctFixedSizeVector`, `vctFixedSizeMatrix`, `vctDynamicVector` and `vctDynamicMatrix`[1].

In addition, the replication semantics is extended to cover type conversion for the elements. We mentioned before that normally the library does not support operations between container objects with different element types. Therefore, it is essential to include at least type conversion mechanisms. One of them is through copy constructors. These are useful for both named and unnamed (temporary) objects.

```
vctDouble3 xVector(1.0, 0.0, 0.0);
vctDouble3 yVector(0.0, 1.0, 0.0);
vctFloat3 yVectorF( yVector );
// zVectorF is the cross product of two float vectors
vctFloat3 zVectorF( vctFloat3(xVector) % yVectorF );
```

Element value constructors

We have already presented numerous examples of initializing fixed-size vectors with element values. Here's a reminder.

```
// initialize a vector of threes
vctInt5 threes(3)
// Initialize a vector with values for each element
vctDouble3 xVector(1.0, 0.0, 0.0);
// A similar format for matrices - element order is
// row major
vctDouble3x3 magic(
    8.0, 1.0, 6.0,
    3.0, 5.0, 7.0,
    4.0, 9.0, 2.0);
```

[1] Transformation classes also allocate memory. They will be discussed separately.

This form of initialization is straightforward and intuitive. Here are the main guidelines for using it.

- Element value constructors are only available for allocating fixed-size vectors and matrices.
- A constructor that takes one element value initializes all the elements to that value.
- The general signature of multiple-element constructors is

  ```
  Container(const value_type element0, /* possibly a few more
  elements*/, const value_type elementK, ...);
  ```

  This means that the compiler passes any number of arguments provided by the caller, and the routine uses `va_arg` macros to retrieve the parameters. While this is mostly convenient from the user's point of view, the user must be aware that the method cannot deduce the number or types of the arguments in compilation time. Therefore, the method has to verify that the correct number of arguments was passed *in runtime*, and it assumes that all the arguments are promoted in the standard way to the `va_arg` promotion of `value_type`. In practice this means that the safest and most correct way to use multiple-element constuctors is to explicitly cast all the arguments to `value_type`, and the compiler will perform any type promotions necessary. If a mismatch between the size of the container and the number of arguments is found, the method throws an exception.
- For a few vector sizes, the library defines multiple-element constructors without `va_arg`. In other words, the library provides constructors with the signatures

  ```
  vctFixedSizeVector(value_type element0, value_type element1);
  vctFixedSizeVector(value_type element0, value_type element1,
      value_type element2);
  ```

  and so on. These are somewhat more efficient than the `va_arg` constructors, and permit passing arguments of other types than `value_type`, which they convert automatically. Nevertheless, we strongly encourage an explicit cast of the arguments in these cases as well. If a mismatch between the size of the container and the number of arguments is found, the method throws an exception.

Element array constructors

For convenient interface with conventional C arrays and objects defined in other libraries, the cisst library defines constructors that take element arrays as input. Vector and matrix classes can be initialized as the following examples show.

```
double cArray[4] = {1.0, 2.0, 3.0, 4.0};
vct4 vec4( cArray );
vctDynamicVector<double> dynVec(4, cArray);
vct2x2 mat2x2( cArray );
vctDynamicMatrix<double> dynMat(2, 2, cArray);
```

The following conditions are required for this initialization.

- The type of input elements and target container elements must be the same.
- The caller is responsible for the input block to contain the same number of elements as the target, and that the elements are packed in contiguous memory addresses.
- For matrices, there is an optional parameter that defines the storage order of the input. The default storage order is row major.

3.2.2 Assignments

We use the term *assignment* for the operation of assigning values to the elements of a container. Assignment performs the same operation a constructor performs, on an object that has already been created. Hence, assignment can also be used for type conversion.

Note specifically that the assignment methods *never* reallocate memory, chage container sizes, or change the reference to an overlaid container. They are *always* applied to the elements in the memory currently occupied by the container. Therefore, the source and target containers of assignment methods must have equal sizes. This is validated by the compiler when only fixed-size containers are involved, and in runtime when dynamic containers are invovled.

The cisst library defines the methods `Assign`, `SetAll`, and, in some cases, overloads the C++ assignment operator (`operator =`). Here are the different versions of these methods.

- `t.SetAll(s_in)` assigns the scalar value `s_in` to all the elements of the container `t`. `t` may be of any vector or matrix type.
- `t = s_in` is the same as `t.SetAll(s_in)`.
- `t.Assign(rt_in)` copies the element of the (const) container `rt_in`) into `t`. `t` and `rt_in` must both be either vectors or matrices of the same size; yet they may be dynamic or fixed-size, allocating or overlaid, independently of each other, and they may have different element types.
- `t = rt_in` is the same as `t.Assign(rt_in)`.

- `t.Assign(ps_in)` copies the elements of the C array contained in the memory beginning at address `ps_in` into `t`. It is the caller's responsibility to ensure that the input block contains the same number of elements as the target, and that they are correctly packed. If `t` is a matrix, a second parameter that defines the storage order of the input can optionally be passed. The default storage order is row major.

- `t.Assign(s_in0, s_in1, ...)` assigns the scalars `s_in0`, `s_in1`, and so on to the elements of `t`. As this method uses `va_arg`, the argument types must be the promotion of the element type of `t`. See the explanation about constructors for more information.

Element access

Container elements can be accessed by index, by pointer, or by iterator. Index access returns a "reference" to the element of that index: a single element in a vector or a row vector in a matrix. Pointer access returns an address of an element, which can be dereferenced or passed to any function that takes a pointer argument. Iterators provide sequential access to the elements, as if the container elements are next to each other. The cisstVector iterators follow the interface defintion of *random-access iterators*, as defined in the C++ Standard Template Library (STL) specification. An iterator can be incremented (`operator ++`), decremented (`operator --`), dereferenced (unary `operator *`), and so on.

The various element access methods in the cisstVector library are summarized (informally) in Table 3.1. Here are a few notes regarding the operations.

- Some operations perform a bounds check to see if the given index (or indexes) are within the container's bounds. If the check fails, the function throws an `std::out_of_range` exception.

- Forward iterators enumerate the elements of the container from the bottom (low index) to the top (high index). Reverse iterators enumerate them in the opposite direction.

- Matrix elements can be accessed directly using the methods that take two parameters, e.g., `Element(j,k)`, or with the conventional C/C++ notation `m[j][k]`. The first set of operations is slightly more efficient than the second, as it computes the actual address of the element in one expression, as opposed to creating a temporary row-overlay object in the second case.

- (Advanced) The vector and matrix operation engines use pointer arithmetic for a very efficient iteration over the elements of one or more containers. The method is beyond the scope of this guide. Unfortunately, implementing the same method in iterator objects may not be as efficient.

Syntax	*Target object*	*Return*
`v.Element(j)` `v[j]`	Vector	Reference to vector element
`v.at(j)` `v(j)`	Vector	Reference to vector element with bounds check

`v.Pointer(j)`
`&(v[j])`
`&(v.Element(j))`
	Vector	Address of vector element

`v.begin()`	Vector	Forward iterator at first element of v
`v.end()`	Vector	Forward iterator at end position (one past last) of v
`v.rbegin()`	Vector	Reverse iterator at first position (last element) of v
`v.rend()`	Vector	Reverse iterator at end position (before first element) of v
`m.Element(j,k)`	Matrix	Reference to matrix element
`m.at(j,k)` `m(j,k)`	Matrix	Reference to matrix element with bounds check
`m.at(j)`	Matrix	Reference to matrix element in sequential row-major order, with bounds check
`m[j]`	Matrix	Matrix row (overlay vector)
`m[j][k]`	Matrix	Reference to matrix element (k-th element of j-th row as overlay vector).
`m.Pointer(j,k)` `&(m.Element(j,k))`	Matrix	Address of matrix element
`m.begin()`, `m.end()`, `m.rbegin()`, `m.rend()`	Matrix	Matrix iterators, in sequential row-major order

Table 3.1: Element access operations in the cisstVector library

3.3 Vectors

3.4 Matrices

3.5 Transformations

Chapter 4

Memory management

Chapter 5

Interfacing other packages