ERC CISST Software
Quick start for cisstVector

Anton Deguet

Date: 2007/11/12 19:51:36

Contents

1 Introduction

These examples provide a quick introduction to the features of cisstVector. The code is part of the CVS module `cisst/examples/vectorTutorial`. To compile your own code, remember to include `cisstVector.h`.

2 Manipulating fixed size vectors and frames

```
void ExampleFrame(void) {
    // create 3 points
    vct3 point000(0.0, 0.0, 0.0);
    vct3 point100(3.0, 0.0, 0.0);
    vct3 point110(2.0, 3.2, 0.0);

    // create a normalized vector along the axis X
    // using methods
    vct3 axisX;
    axisX.DifferenceOf(point100, point000);
    axisX.Divide(axisX.Norm());

    // create a normalized vector along the axis Z
    // using operators ’-’ for difference,
    // ’%’ for cross product, and ’/=’ for in-place
    // elementwise division.
    vct3 tmp = point110 - point000;
    vct3 axisZ = axisX % tmp;
    axisZ /= axisZ.Norm();

    // Using named methods instead of operators
    vct3 axisY;
    axisY.CrossProductOf(axisZ, axisX);
    axisY.NormalizedSelf();

    /* three ways to display the result: */
    // 1. Just output a vector
    cout << "X:␣" << axisX << endl;
    // 2. Output vector component by index
    cout << "Y:␣" << axisY[0]
         << "␣" << axisY[1]
         << "␣" << axisY[2] << endl;
    // 3. Output vector component by ‘‘name’’
    cout << "Z:␣" << axisZ.X()
         << "␣" << axisZ.Y()
         << "␣" << axisZ.Z() << endl;
    /**/

    // create a rotation along axis "tmp"
    tmp.Assign(1.3, -0.3, 1.7);
    tmp.NormalizedSelf();
    vctMatRot3 rotation(vctAxAnRot3(tmp, 3.1415 / 2.0));

    /* two ways to apply the rotation
       to vectors: */
    vct3 newAxisX, newAxisY, newAxisZ;
    // 1. Using operator ’*’
    newAxisX = rotation * axisX;
    // 2. Using named method ApplyTo
```

```
    rotation.ApplyTo(axisY, newAxisY);
    rotation.ApplyTo(axisZ, newAxisZ);

    /* verify that the transformed vectors are still
       an orthogonal basis.  Compute dot products
       in three ways. */
    // 1. Using operator * on two vectors
    double dotXY = newAxisX * newAxisY;
    // 2. Using global function vctDotProduct
    double dotYZ = vctDotProduct(newAxisY, newAxisZ);
    // 3. Using named method DotProduct
    double dotZX = newAxisZ.DotProduct(newAxisX);

    cout << "Dot products: " << dotXY << " "
         << dotYZ << " " << dotZX << endl;
    /**/

    // create a rigid transformation frame from
    // a rotation matrix and a translation vector
    vct3 translation(0.0, 3.2, 4.5);
    vctFrm3 frame(rotation, translation);
    // Apply the frame to a vector
    vct3 frameOnX = frame * axisX;
    cout << "Image of X: " << frameOnX << endl;

   // inverse of the frame
    vctFrm3 inverse;
    inverse.InverseOf(frame);
    // The product of a frame and its inverse
    // should be the identity (eye for rotation,
    // zero for translation).
    cout << "frame * inverse: " << endl << frame * inverse
         << endl;
    // Compare this with the actual identity frame
    cout << "Identity frame: " << endl
         << vctFrm3::Identity() << endl;
}
```

In the example, we used some fixed size vector of 3 doubles (`vct3`) and some of the methods and operators available for this class (`Norm()`, `DifferenceOf`, `CrossProductOf`, operators `-`, `%`, etc.).
We also introduced a rotation matrix and a frame which can be used with the cisst fixed size vectors (`vct3`, same as `vctDouble3`). For more information related to transformations, see the cisstVector Tutorial

3 Manipulating dynamic vectors and matrices

```
void ExampleDynamic(void) {
    // define our prefered types
```

```
    typedef vctDynamicVector<double> VectorType;
    typedef vctDynamicMatrix<double> MatrixType;

    // The dynamic vector library may throw exceptions,
    // (derived from std::exception)
    // so we place the operations in a try-catch block.
    try {
        // create an empty vector
        VectorType vector1;
        cout << "Size of vector1: " << vector1.size() << endl;

        // resize and fill the vector
        unsigned int index;
        vector1.SetSize(5);
        for (index = 0; index < vector1.size(); index++) {
            vector1[index] = index;
        }
        // look at product of elements
        cout << "Product of elements is 0? "
             << vector1.ProductOfElements() << endl;

        // create a matrix initialized with zeros
        MatrixType matrix1(7, vector1.size());
        matrix1.SetAll(0.0);

        // set the diagonal to 5.0
        matrix1.Diagonal().SetAll(5.0);

        // look at the sum/product of elements
        cout << "Sum of elements is 25? "
             << matrix1.SumOfElements() << endl;

        // multiply matrix1 by vector 2
        VectorType vector2(matrix1.rows());
        vector2.ProductOf(matrix1, vector1);

        // multiply vector1 directly
        VectorType vector3(vector1.size());
        vector3.ProductOf(5.0, vector1);

        // resize vector2 while preserving the data
        vector2.resize(vector3.size());

        // vector2 and vector3 are the same?
        VectorType difference;
        difference = vector3 - vector2;
        difference.AbsSelf();
        cout << "Maximum difference between v2 and v3: "
             << difference.MaxElement() << endl;
        // alternative solution
        cout << "Maximum difference between v2 and v3: "
```

```
                << (vector3 - vector2).MaxAbsElement() << endl;

    } // end of try block
    // catch block
    catch (std::exception Exception) {
        cerr << "Exception occured: " << Exception.what() << endl;
    }
}

```

In this example, we created a couple of dynamic vectors as well as a dynamic matrix. Dynamic containers are convenient for large collections of data, or when the number of elements is provided during runtime. Since the allocation is dynamic, it is important to check that the sizes of the operands are compatible. Our library throws exceptions, derived from the Standard Library `std::exception` class, on illegal operation arguments, such as unmatching vectors or out-of-range element access. This is why we use a `try` and `catch` structure.
The space allocated for a vector or a matrix can be changed in two ways. `SetSize` discards any old data and allocated memory in the specified size. `resize` preserves the old data by first allocating new space and then copying the elements from the old space to the new one. The `Diagonal` method is a first example of manipulating *container slices*, or *vector references*. The concept is demonstrated in the next example code.

4 Container slices and vector references

```
void ExampleReference(void) {
    // define our preferred type
    typedef vctDynamicMatrix<int> MatrixType;

    try {
        // create a matrix filled with zero
        MatrixType matrix(8, 6);
        matrix.SetAll(0);

        // create a reference to column 3 (4th column
        // from zero-base)
        MatrixType::ColumnRefType col3 = matrix.Column(3);
        col3.SetAll(2);

        // create a reference to row 0
        MatrixType::RowRefType row0 = matrix.Row(0);
        row0.SetAll(3);

        // create a reference to the last row
        MatrixType::RowRefType rowLast = matrix[matrix.rows() - 1];
        rowLast.SetAll(4);

        // create a reference to the diagonal
```

```
        MatrixType::DiagonalRefType diagonal = matrix.Diagonal();
        diagonal.SetAll(1);

        // create a sub matrix
        MatrixType::Submatrix::Type submatrix(matrix,
                                              /* from */ 3, 1,
                                              /* size */ 4, 2);
        submatrix += 6;

       // display the result
        cout << "Matrix␣modified␣by␣pieces:␣" << endl
             << matrix << endl;
        cout << "Trace:␣" << diagonal.SumOfElements() << endl;

    } catch (std::exception Exception) {
        cout << "Exception␣received:␣" << Exception.what() << cout;
    }
}
```

This example demonstrates the use of slices through a dynamic matrix. The term “slice” refers to a contiguous subregion of a larger vector or matrix. In the example, we directly address columns, rows, and a diagonal of the large matrix `matrix` through the methods `Column`, `Row` and `Diagonal`. We use the word `Ref` to indicate a vector or matrix object that doesn't allocate its own memory, but overlays another object's memory, such as a slice in a matrix. These slices have appropriately named types, which are `ColumnRefType`, `RowRefType`, and `DiagonalRefType`.

Next, we define a submatrix slice, using the type `MatrixType::Submatrix::Type` (the reason for this notation will be given soon). The constructor takes the location of the first element of the submatrix in the large matrix, and the dimensions of the submatrix. As we can see, we can operate on the submatrix just as we do on any matrix.

The next example shows how to use slices with fixed-size vectors and matrices. In the example, we allocate a 4×4 homogeneous transformation matrix, and relate to parts of it as a rotation component and a translation component.

```
void ExampleReferenceFixedSize(void) {
    // define our preferred type
    typedef vctFixedSizeMatrix<float,4,4> MatrixType;

    try {
        // create a matrix initialized as identity
        MatrixType matrix(0.0f);
        matrix.Diagonal().SetAll(1.0f);

        // create a rotation matrix of 30deg about the
        // X axis
        vctAxAnRot3 axRot30(vct3(1.0, 0.0, 0.0), (3.14159265 / 6.0));
        vctMatRot3 rot30( axRot30 );

```

```
        // Assign the rotation to the upper-left
        // 3x3 part of our matrix
        MatrixType::Submatrix<3,3>::Type rotSubmatrix(matrix, 0, 0);
        rotSubmatrix.Assign( rot30 );

        // Create reference to the last column
        MatrixType::ColumnRefType lastCol = matrix.Column(3);

        /**/
        // Create a slice of length 3 of the last column
        // NOTE: MSVC7 does not allow the following definition:
        //     MatrixType::ColumnRefType::Subvector<3>::Type translation;
        // but gcc does.
        typedef MatrixType::ColumnRefType ColumnType;
        typedef ColumnType::Subvector<3>::Type TranslationRefType;
        TranslationRefType translation(lastCol);
        translation.Assign(5.5f, -3.25f, 217.32f);
        /**/

        // Display the result
        cout << "final␣matrix:\n";
        cout << matrix << endl;

    } catch (std::exception Exception) {
        cout << "Exception␣received:␣" << Exception.what() << cout;
    }
}
```

Here, we see that in fixed-size objects, the size of the submatrix has to be given in template parameters, though the location of its first element is still passed as a regular function argument. In C++ it is impossible to have templated `typedef` statements. Instead, we declare a templated inner class: `MatrixType::Submatrix<3,3>`, and that inner class has an internal `typedef` of its own type as `Type`. Similarly, for the slice of the first three elements in the last column, we use the type

`MatrixType::ColumnRefType::Subvector<3>::Type`

Note that the element type in this example is `float`, while the rotation matrix `rot30` is `double`. We can assign vectors and matrices of different element types to each other, but normally we do not allow other operations between them. Also note that we explicitly define literals of type `float` using the suffix `f`. This may reduce the number of compiler warnings. Also, we consider it safer to use explicit casting of the arguments whenever they are passed in a variable-length list (`va_arg`, or an ellipsis in the function declaration). This has to do with the mechanism used in C and C++ for handling variable-length argument list. So generally, if we have a long vector `v` of type `double`, the following code may generate an error:

```
v.Assign(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Instead, use explicit literal type:

```
v.Assign(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0);
```

For more information regarding the different type of references available, refer to the cisstVector Tutorial .

5 Manipulating multidimensional arrays

Multidimensional arrays can be used to represent volumes (medical imaging), images with multiple layers (separating RGB channels) or any dataset of high dimension. `vctDynamicArray` can also be used for datasets of dimension 1 or 2 but for these, `vctDynamicVector` and `vctDynamicMatrix` might be a better choice.
Multidimensional arrays can only be found in the dynamic allocation flavor, i.e. there is no `vctFixedSizeNArray`. A multidimensional array type is defined by the type of elements stored as well as by the dimension, i.e. both of these are defined at compilation time and cannot be changed at runtime. As for the vectors and matrices, it is recommended to define an nArray type to use in your code:

```
typedef vctDynamicNArray<unsigned short, 3> InputVolumeType;
```

Since the dimension defines the number of indices required to randomly access an element as well as the number of sizes to resize an nArray, it is also very convenient to define both an index and size type:

```
typedef InputVolumeType::nsize_type SizeType;
typedef InputVolumeType::nindex_type IndexType;
```

The following code illustrates how to create, fill and operate on nArrays:

```
void ExampleNArray(void) {
    // Define a working volume and index types
    typedef vctDynamicNArray<unsigned short, 3> InputVolumeType;
    typedef InputVolumeType::nsize_type SizeType;
    typedef InputVolumeType::nindex_type IndexType;

    // The dynamic vector library may throw exceptions,
    // (derived from std::exception)
    // so we place the operations in a try-catch block.
    try {
        // Create a volume object and allocate memory
        SizeType volumeSize(128, 256, 256);
        InputVolumeType inputVolume(volumeSize);
        // alternative to set size and allocate
        inputVolume.SetSize(volumeSize);

        // Fill the memory with data
        vctRandom(inputVolume, 0, 10);

```

```
        // Random access (read) of elements
        IndexType zyxIndex(15, 120, 240);
        cout << inputVolume.Element(zyxIndex) << endl
             << inputVolume.at(zyxIndex) << endl
             << inputVolume[zyxIndex[0]] [zyxIndex[1]] [zyxIndex[2]]
             << endl;

        // Define a new volume type
        typedef vctDynamicNArray<double, 3> NormalizedVolumeType;

        // Type conversions from and existing volume, also defines the size
        NormalizedVolumeType newVolume(inputVolume);
        // alternative
        newVolume.Assign(inputVolume);

        // Find minimum and maximum
        double minElement, maxElement;
        newVolume.MinAndMaxElement(minElement, maxElement);
        cout << "Min and max " << minElement << " " << maxElement << endl;
        // "shift and bias"
        newVolume.Subtract(minElement); // Also available: operator -=
        newVolume.Divide(maxElement - minElement); // or operator /=

        // Operations with several operands
        NormalizedVolumeType noise, corrected;
        corrected.DifferenceOf(newVolume, noise);

        // Slice overlay: array.Slice(dimension, sliceIndex)
        newVolume.Slice(0, 0).SumOf(newVolume.Slice(0, 10),
                                    newVolume.Slice(0, 20));

        // Using a named object for slice overlay
        typedef NormalizedVolumeType::ConstSliceRefType InputSliceType;
        InputSliceType input1;
        input1.SliceOf(newVolume, 1, 15);
        // alternative
        input1.SetRef(newVolume.Slice(1, 15));

        // Layout manipulation: permutation of dimensions
        vctDynamicNArrayRef<double, 3> permutedVolume;
        permutedVolume.PermutationOf(newVolume, IndexType(1, 0, 2));
    } // end of try block
    // catch block
    catch (std::exception Exception) {
        cerr << "Exception occured: " << Exception.what() << endl;
    }
}
```

Multidimensional arrays also provide ways to create slices and other references. It is possible to:

- Use only a sub-array while keeping the dimension, i.e. create a window along each dimension
- Reduce the visible dimension, i.e. only consider n-1 dimensions
- Use the data with a permutation of indices, i.e. permuting the dimensions

In this example, we reduced the dimension using the method `SliceOf`. The type of a slice is defined using the same type of elements and subtracting one from the dimension. To ease the programmer's life, one can use `NormalizedVolumeType::SliceRefType` and `NormalizedVolumeType::ConstSliceRefType`. A more subtle way to slice an nArray is to use the square brackets (operator `[]`). This is similar to a matrix operator `[]` as both operators return a reference container with a lower dimension.
Finally, the method `PermutationOf` allows to view the nArray from a different "angle" by implicitly re-ordering the dimensions (please note that the data itself is not moved or copied as for all cisst "Ref" objects). This can be compared to the method `TransposeOf` for a matrix.

6 Using the C++ Standard Template Library

The different containers of cisstVector have been written to be compatible with the STL. They define different iterators as well as the methods required to manipulate these iterators.

```
void ExampleSTL(void) {
    typedef vctFixedSizeVector<double, 6> VectorType;

    VectorType vector1;
    int value = vector1.size();
    const VectorType::iterator end = vector1.end();
    VectorType::iterator iter;
    // fill with numbers using iterators
    for (iter = vector1.begin(); iter != end; ++iter) {
        *iter = value--;
    }
    cout << vector1 << endl;
    // sort using std::sort.  cool isn't it?
    std::sort(vector1.begin(), vector1.end());
    cout << vector1 << endl;
}
```

In this example, we demonstrated the use of an STL generic algorithm (`std::sort`) on a cisstVector container.

7 Using cisstCommon

This example requires to include `cisstCommon.h`.

```
void ExampleCommon(void) {
    // fill a vector with random numbers
    vctFixedSizeVector<double, 8> vector1, vector2;
    cmnRandomSequence & randomSequence = cmnRandomSequence::GetInstance();
    randomSequence.ExtractRandomValueArray(-100.0, 100.0,
                                           vector1.Pointer(), vector1.size());

    // to fill a matrix or vector one can also use vctRandom
    vctRandom(vector2, -100.0, 100.0);

    // perform some operations such as divide by zero
    vector2.SetAll(0.0);
    vector2.Divide(vector2.Norm());
    unsigned int index;
    // look for Not A Number
    for (index = 0; index < vector2.size(); index++) {
        if (CMN_ISNAN(vector2[index])) {
            cout << "vector[" << index << "] is NaN " << endl;
        }
    }

    // create a rotation based on an a normalized axis
    vctAxAnRot3 axisAngle(vct3(1.0, 0.0, 0.0), cmnPI / 2.0);
    vctQuatRot3 rot1(axisAngle);

    // modify this rotation with a new axis, not well normalized
    vct3 axis(1.0005, 0.0, 0.0);
    if (axis.Norm() > cmnTypeTraits<double>::Tolerance()) {
        cout << "Axis is not normalized wrt default tolerance" << endl;
    }
    cmnTypeTraits<double>::SetTolerance(0.001);
    // this method asserts that the axis is normalized
    axisAngle.Axis() = axis;
    axisAngle.Angle() = cmnPI / 2.0;
    rot1.From(axisAngle);
    cmnTypeTraits<double>::SetTolerance(cmnTypeTraits<double>::DefaultTolerance);
}
```

This example illustrates how to use the cisst `cmnRandomSequence` to fill a vector or matrix with random numbers.
The macro `CMN_ISNAN` allows to check if a variable is still a number. It is defined in `cmnPortability.h`.
The default tolerance is used in many methods of cisstVector (e.g. `IsNormalized()` for any transformation) and it might be useful to increase it for a given application. This should be used with caution. The class `cmnTypeTraits` contains some useful information per type (double, float, char, int, etc) such as `HasNaN`, `MinNegativeValue`, etc.

8 Using cisstNumerical

In this example, we are showing how to choose the storage order. In C/C++, the usual convention is to use a row major data storage, this means that the elements of a matrix are stored in a single block, row by row. This is the default behavior of cisstVector. In some cases, it is necessary to store the data column by column. This is the case whenever one wants to interface with an external library using a column major representation. cisstNumerical includes a collections of wrappers around FORTRAN routines which might require a column major storage order (see cisstNumerical Quickstart and the cisst cnetlib pages).

```
void ExampleNumerical(void) {
    // here we create the 3x3 matrix (column major!!)
    const unsigned int rows = 3, cols = 3;
    vctDynamicMatrix<double> m(rows, cols, VCT_COL_MAJOR);
    // fill with random numbers
    vctRandom(m, -10, 10);

    // display the current matrix
    cout << "Matrix:" << endl << m << endl;

    // create and solve the problem
    nmrSVDDynamicData svdData(m);
    nmrSVD(m, svdData);
    cout << "Singular Values: " << endl << svdData.S() << endl;
}
```

This example uses `vctRandom` to fill a 3 by 3 matrix. The matrix is declared using `VCT_COL_MAJOR`. For more information regarding the storage order, see the cisstVector Tutorial . To find the singular values of this matrix, we used the function `nmrSVD`. The class `nmrSVDDynamicData` allows to allocate some memory prior to the computation so that one can use the same memory for multiple problems of the same size. This can be particularly useful in a loop.

9 Writing your own functions

These examples are reserved to advanced programmers. They require a fairly good understanding of the C++ templates and the class hierarchy of cisstVector (refer to the cisstVector Tutorial). The terms *Const* and *Ref* refer to the cisstVector classes (e.g. `vctDynamicConstVectorRef`), not the C++ keyword `const` and symbol `&`.
It is important to understand that the declaration of a new templated function or method is much more complex than the call to this function or method. When you will call the method, the compiler will infer (i.e. deduce) the template parameters based on the type of the objects used as function parameters. This allows to create a very generic method while preserving the ease of use.

9.1 Using dynamic containers

The first four functions show different possible signatures to use for a function parameter. The fifth signature is for a function that takes two matrices containing the same type of elements, but each matrix can be either a *Reference* or not, *Const* or not.
The last example shows how to use a `vctReturnDynamicVector` to avoid a copy of all the elements of a vector when it is `return`ed (this approach is valid for a matrix as well).

```
// take any dynamic matrix as input parameter
// Const or not, Reference or not
template <class _matrixOwnerType, class _elementType>
void
FunctionDynamicA(vctDynamicConstMatrixBase<_matrixOwnerType,
                                           _elementType> & matrix)
{}

// take any non-const matrix as input parameter, Reference or not
template <class _matrixOwnerType, class _elementType>
void
FunctionDynamicB(vctDynamicMatrixBase<_matrixOwnerType,
                                      _elementType> & matrix)
{}

// only take a matrix as input parameter, can't use a Reference
template <class _elementType>
void
FunctionDynamicC(vctDynamicMatrix<_elementType> & matrix)
{}

// this shows how to restrict to a given type of elements (double)
template <class _matrixOwnerType>
void
FunctionDynamicD(vctDynamicConstMatrixBase<_matrixOwnerType, double> & matrix)
{}

// take any two dynamic matrices as input parameters
// Const or not, Reference or not, same type of elements
template <class _matrixOwnerType1, class _matrixOwnerType2, class _elementType>
void
FunctionDynamicE(vctDynamicConstMatrixBase<_matrixOwnerType1,
                                           _elementType> & matrix1,
                 vctDynamicConstMatrixBase<_matrixOwnerType2,
                                           _elementType> & matrix2)
{}

// function with a return type
template<class _vectorOwnerType, class _elementType>
vctReturnDynamicVector<_elementType>
FunctionDynamicF(const vctDynamicConstVectorBase<_vectorOwnerType,
                                                 _elementType> & inputVector) {
```

```
    typedef _elementType value_type;
    vctDynamicVector<value_type> resultStorage(inputVector.size());
    // ...... do something to resultStorage
    return vctReturnDynamicVector<value_type>(resultStorage);
}
```

9.2 Using fixed size containers

Fixed size containers are similar to the dynamic ones with a major exception, the size(s) and stride(s) must be specified at compilation time. This requirement adds a fair amount of template parameters.

```
// Take any fixed size vector as input parameter
// Const or not, Reference or not
template <unsigned int _size, int _stride, class _elementType,
          class _dataPtrType>
void
FunctionFixedSizeA(vctFixedSizeConstVectorBase<_size, _stride, _elementType,
                                                _dataPtrType> & vector)
{}

// take any non-const vector as input parameter, Reference or not
template <unsigned int _size, int _stride, class _elementType,
          class _dataPtrType>
void
FunctionFixedSizeB(vctFixedSizeVectorBase<_size, _stride, _elementType,
                                          _dataPtrType> & vector)
{}

// only take a vector as input parameter, can't use a Reference
template <class _elementType, unsigned int _size>
void
FunctionFixedSizeC(vctFixedSizeVector<_elementType, _size> & vector)
{}

// this shows how to restrict to a given type of elements (float)
template <unsigned int _size, int _stride, class _dataPtrType>
void
FunctionFixedSizeD(vctFixedSizeConstVectorBase<_size, _stride, float,
                                                _dataPtrType> & vector)
{}

// take any two fixed size vectors as input parameters
// Const or not, Reference or not, same size, same type of elements
template <unsigned int _size, class _elementType,
          int _stride1, class _dataPtrType1,
          int _stride2, class _dataPtrType2>
void
FunctionFixedSizeE(vctFixedSizeConstVectorBase<_size, _stride1, _elementType,
                                                _dataPtrType1> & vector1,
```

```
                      vctFixedSizeConstVectorBase<_size, _stride2, _elementType,
                                                   _dataPtrType2> & vector2)

{}
```

Since the fixed size containers are designed for fairly small containers (up to approximately 10 elements) and they use stack memory, there is no specific `return` type (as opposed to `vctReturnDynamicVector` or `vctReturnDynamicMatrix`).