

cisstVector: Vectors and matrices quick sheet (part 1 of 4)

Revision: 1.18 - Date: 2006/11/21 21:21:53

	Code †	Description
Classes and typedefs	<code>vctFixedSizeVector<_elementType, _size></code> - <code>vctDouble1 vctDouble2 vctDouble3 vctDouble4 ...</code> - <code>vct1 vct2 vct3 vct4 vct5 vct6 ...</code> - <code>vctFloat1 vctFloat2 vctFloat3 vctFloat4 vctFloat5 ...</code> - <code>vctInt1 vctInt2 vctInt3 vctInt4 vctInt5 vctInt6 ...</code> - <code>vctChar1 vctChar2 vctChar3 vctChar4 vctChar5 ...</code>	Fixed size vector ... of doubles ... of doubles (implicit) ... of floats ... of ints ... of chars
	<code>vctFixedSizeMatrix<_elementType, _rows, _cols></code> - <code>vctDouble2x2 vctDouble3x3 vctDouble2x3 ...</code> - <code>vct2x2 vct3x3 vct4x4 vct2x3 vct2x4 vct3x4 ...</code> - <code>vctFloat2x2 vctFloat3x3 vctFloat4x4 vctFloat3x2 ...</code> - <code>vctInt2x2 vctInt3x3 vctInt4x4 vctInt3x4 ...</code> <code>vctFixedSizeMatrix<_elementType, _rows, _cols, _order></code> - <code>vctFixedSizeMatrix<double, 3, 3, VCT_ROW_MAJOR> m1;</code> - <code>vctFixedSizeMatrix<double, 3, 3, VCT_COL_MAJOR> m2;</code>	Fixed size matrix ... of doubles ... of doubles (implicit) ... of floats ... of ints With specific storage order - row major § (default, \simeq <code>vct3x3</code>) - col major (Fortran like)
	<code>vctDynamicVector<_elementType></code> - <code>vctDoubleVec, vctVec</code> - <code>vctFloatVec, vctIntVec</code>	Dynamic vector ... of doubles, doubles (implicit) ... of floats, ints
	<code>vctDynamicMatrix<_elementType></code> - <code>vctDoubleMat, vctMat</code> - <code>vctFloatMat, vctIntMat</code>	Dynamic matrix ... of doubles, doubles (implicit) ... of floats, ints
	Constructors	<code>vct4 v1;</code> <code>vct4 v2(0.0); vct4 v3(0.1, 0.2, 0.3, 0.4);</code> <code>vct4 v4(v1);</code> <code>vct2x2 m1;</code> <code>vct2x2 m2(0.0); vct2x2 m3(0.1, 0.2, 0.3, 0.4);</code> <code>vct2x2 m4(m1);</code> <code>vctVec v1;</code> <code>vctVec v2(12);</code> <code>vctVec v3(2, 0.0); vctVec v4(2, 0.1, 0.2);</code> <code>vctVec v5(v1);</code> <code>vctMat m1;</code> <code>vctMat m2(12, 7);</code> <code>vctMat m3(6, 4, VCT_ROW_MAJOR);</code> <code>vctMat m4(5, 7, VCT_COL_MAJOR);</code> <code>vctMat m5(2, 3, 0.0);</code> <code>vctMat m6(3, 5, 0.0, VCT_ROW_MAJOR);</code> <code>vctMat m7(2, 3, 0.0, VCT_COL_MAJOR);</code> <code>vctMat m8(2, 2, 0.1, 0.2, 0.3, 0.4);</code> <code>vctMat m9(m4);</code>
Assignments	<code>c1.Assign(c2); c1 = c2</code>	Assign from similar container
	<code>c.Assign(s0, s1, s2, ...)</code> e.g.: <code>v.Assign((double)0, 1.3, .34, 0.0);</code>	Assign from multiple variables using <code>va_arg</code> <i>Input must be explicitly cast to element type!</i>
	<code>c.SetAll(value); c = value;</code> <code>vctRandom(c, s_min, s_max);</code>	Assign value to all elements Fill with random values in $[s_{min}, s_{max}]$
	<code>bool succeeded = c1.FastCopyOf(c2);</code> <code>bool succeeded = c1.FastCopyOf(c2, false);</code>	Shallow copy using <code>memcpy</code> ** Shallow copy without safety checks
Identity (Eye for <i>I</i>)	<code>MatrixType::Eye()</code> - <code>vct4x4 m = vct4x4::Eye();</code> - <code>vctInt3x3 m = vctFixedSizeMatrix<int, 3, 3>::Eye();</code> - <code>vctMat m = vctMat::Eye(9);</code> - <code>vctIntMat = vctDynamicMatrix<int>::Eye(6);</code>	Eye for dynamic & fixed size matrices using: - fixed size matrix typedef - fixed size matrix class definition - dynamic matrix typedef and dimension (9×9) - dynamic matrix class definition (6×6)

† **v** is for vector, **m** for matrix, **c** for a container (either vector or matrix) and **s** is for scalar.

§ `VCT_ROW_MAJOR` is the default for fixed size and dynamic matrices. Use `VCT_COL_MAJOR` to store by column.

**`FastCopyOf` fails and returns `false` if the containers are not compact or don't have the same memory layout (storage order).

cisstVector: Vectors and matrices quick sheet (part 2 of 4)

Revision: 1.18 - Date: 2006/11/21 21:21:53

	Code	Description
Member typedefs (STL like)	<pre>ContainerType::size_type size; ContainerType::index_type index; ContainerType::stride_type stride; ContainerType::value_type element; ContainerType::reference ref; ContainerType::const_reference constRef; ContainerType::pointer ptr; ContainerType::const_pointer constPtr; ContainerType::NormType norm; ContainerType::AngleType angle;</pre>	<p>Size type (unsigned int) Index type (unsigned int) Stride type (int) Type of contained elements Reference on element (value_type &) Const reference (const value_type &) Pointer on element (value_type *) Const pointer (const value_type *) Norm type (double) Angle type (double)</p>
Iterators (STL like)	<pre>ContainerType::iterator iter; ContainerType::const_iterator iter; ContainerType::reverse_iterator iter; ContainerType::const_reverse_iterator iter; iter = c.begin(); iter = c.end(); iter = c.rbegin(); iter = c.rend();</pre>	<p>Iterator type Const iterator type Reverse iterator type Const reverse iterator type Iterator on first/last element Reverse iterator on last/first element</p>
Memory layout	<pre>size = c.size(); nbRows = m.rows(); nbCols = m.cols(); b = c.empty(); b = m.IsSquare(); b = m.IsSquare(5);</pre>	<p>Length for vectors, rows × cols for matrices Number of rows/columns Test if size is zero Test if matrix is square Test if square matrix of a given size (e.g. 5)</p>
	<pre>stride = v.stride(); rowStride = m.row_stride(); colStride = m.col_stride();</pre>	<p>Stride between elements Stride between rows Stride between columns</p>
	<pre>ptr = c.Pointer(); ptr = v.Pointer(i); ptr = m.Pointer(row, col);</pre>	<p>Pointer on first element Pointer on i^{th} vector element Pointer on $[row, col]$ matrix element</p>
Test indices	<pre>bool b = c.ValidIndex(index); bool b = m.ValidIndex(row, col); bool b = m.ValidRowIndex(row); bool b = m.ValidColIndex(col);</pre>	<p>Index is lesser or equal to size Both row and column indices are correct Row index is valid Column index is valid</p>
Storage	<pre>bool b = m.IsRowMajor(); bool b = m.IsColMajor(); bool b = m.IsCompact(); bool b = m.IsFortran(); bool storageOrder = m.StorageOrder();</pre>	<p>Stored row first (C/C++ like - default) Stored column first (Fortran like) Uses a compact block of memory , i.e., all elements are in one contiguous block. Both compact and column major Returns storage order (true for row major, false for column major)</p>
Dynamic vector resizing	<pre>v.SetSize(newSize); v.resize(newSize);</pre>	<p>Destructive size change Non destructive size change</p>
Dynamic matrix resizing	<pre>m.SetSize(rows, cols, storageOrder); m.resize(rows, cols);</pre>	<p>Destructive size change Non destructive size change</p>
Access vector parts	<pre>v.X() = x; y = v.Y(); v.Z() = z; w = v.W(); v.XY() = v2; v3 = v.XYZ(); v4 = v.XYZW(); v.XZ(); v.XW(); v.YZ(); v.YW(); v.ZW(); v.YZW();</pre>	<p>Access elements by name Access subvectors by name Access subvectors other than "head"</p>
Access matrix parts	<pre>v = m.Row(index); v = m.Column(index); v = m.Diagonal(); mt = m.TransposeRef();</pre>	<p>Reference to a given row/column Reference to the diagonal/transpose</p>
Matrix permutations	<pre>m.ExchangeRows(r1, r2); m.ExchangeColumns(c1, c2);</pre>	<p>Swap the values between two rows/columns</p>
Elementwise tests	<pre>b = c.IsPositive(); b = c.IsNonNegative(); b = c.IsNegative(); b = c.IsNonPositive(); b = c.All(); b = c.Any();</pre>	<p>All elements are greater than zero All elements are greater than or equal to zero All elements are lesser than zero All elements are lesser than or equal to zero All elements are different from zero At least one element is different from zero</p>
Formatted output	<pre>std::string s = c.ToString(); std::cout << c; c.ToStream(std::cout);</pre>	<p>Output to std::string Output to a C++ stream</p>

cisstVector: Vectors and matrices quick sheet (part 3 of 4)

Revision: 1.18 - Date: 2006/11/21 21:21:53

	Description	Method or Function [†]	Operator
+	Addition	<code>c.SumOf(c1, cs2);</code> <code>c.Add(cs1);</code>	<code>c = c1 + cs2;</code> <code>c += cs1;</code>
-	Subtraction	<code>c.DifferenceOf(c1, cs2);</code> <code>c.Subtract(cs1);</code>	<code>c = c1 - cs2;</code> <code>c -= cs1;</code>
×	Container scalar multiplication	<code>c1.ProductOf(c2, s);</code> <code>c1.ProductOf(s, c2);</code> <code>c.Multiply(s1);</code>	<code>c1 = c2 * s;</code> <code>c1 = s * c2;</code> <code>c *= s1;</code>
	Matrix vector multiplication	<code>v2.ProductOf(m, v1);</code> <code>v2.ProductOf(v1, m);</code>	<code>v2 = m * v1;</code> <code>v2 = v1 * m;</code>
	Matrix matrix multiplication [§]	<code>m.ProductOf(m1, m2);</code>	<code>m = m1 * m2;</code>
	Vector outer product	<code>m.OuterProductOf(v1, v2);</code>	
	Elementwise multiplication $c_1[i] = c_2[i] * c_3[i]$ and $c_1[i] * = c_2[i]$	<code>c1.ElementwiseProductOf(c2, c3);</code> <code>c1.ElementwiseMultiply(c2);</code>	
÷	Container scalar division	<code>c1.RatioOf(c2, s);</code> <code>c.Divide(s);</code>	<code>c1 = c2 / s;</code> <code>c /= s;</code>
	Elementwise division $c_1[i] = c_2[i] / c_3[i]$ and $c_1[i] / = c_2[i]$	<code>c1.ElementwiseRatioOf(c2, c3);</code> <code>c1.ElementwiseDivide(c2);</code>	
× ^	Cross product (size 3 only) % operator has high precedence	<code>v1.CrossProductOf(v2, v3);</code> <code>v1 = vctCrossProduct(v2, v3);</code>	<code>v1 = v2 % v3;</code>
○ ●	Dot product (inner product)	<code>s = v1.DotProduct(v2);</code> <code>s = vctDotProduct(v1, v2);</code>	<code>s = v1 * v2;</code>
+×	Accumulation	<code>c2.AddProductOf(s, c1);</code>	<code>c2 += (s * c1);</code>
=	Equal	<code>b = c1.Equal(cs2);</code>	<code>b = (c1 == cs2);</code>
≠	Not equal	<code>cb = c1.ElementwiseEqual(cs2);</code> <code>b = c1.NotEqual(c2);</code>	<code>b = (c1 != cs2);</code>
≈	Approximately equal	<code>cb = c1.ElementwiseNotEqual(c2);</code> <code>b = c1.AlmostEqual(c2, tolerance);</code>	
<	Lesser	<code>b = c1.Lesser(cs2);</code>	
≤	Lesser or equal	<code>cb = c1.ElementwiseLesser(cs2);</code> <code>b = c1.LesserOrEqual(cs2);</code> <code>cb = c1.ElementwiseLesserOrEqual(cs2);</code>	
>	Greater	<code>b = c1.Greater(cs2);</code>	
≥	Greater or equal	<code>cb = c1.ElementwiseGreater(cs2);</code> <code>b = c1.GreaterOrEqual(cs2);</code> <code>cb = c1.ElementwiseGreaterOrEqual(cs2);</code>	
$\min(c_i, u)$	Clip above <u>upper</u> bound	<code>c.ClipAbove(s);</code> <code>c2.ClippedAboveOf(c1, s);</code> <code>c2.ClippedAboveOf(s, c1);</code>	
$\max(c_i, l)$	Clip below <u>lower</u> bound	<code>c.ClipBelow(s);</code> <code>c2.ClippedBelowOf(c1, s);</code> <code>c2.ClippedBelowOf(s, c1);</code>	

[†]Operand types: **b** boolean, **s** scalar, **v** vector, **m** matrix, **c** container (**v** or **m**), **cs** container *or* scalar, **cb** container of booleans.

[§]Verifies that output base pointer is different from input pointers, throws `std::runtime_error` otherwise.

cisstVector: Vectors and matrices quick sheet (part 4 of 4)

Revision: 1.18 - Date: 2006/11/21 21:21:53

	Description	Method or Function [†]	Operator
$\ c\ _2$	ℓ_2 norm	<code>s = c.Norm();</code>	
$\ c\ _2^2$	ℓ_2 norm square	<code>s = c.NormSquare();</code>	
$\ c\ _1$	ℓ_1 norm	<code>s = c.L1Norm();</code>	
$\ c\ _\infty$	ℓ_∞ norm	<code>s = c.LinfNorm();</code>	
$\sum_i c[i]$	Sum of elements	<code>s = c.SumOfElements();</code>	
$\prod_i c[i]$	Product of elements	<code>s = c.ProductOfElements();</code>	
$\sum_i m[i, i]$	Trace	<code>s = m.Trace();</code>	
$\min_i c[i]$	Minimum	<code>s = c.MinElement();</code>	
$\max_i c[i]$	Maximum	<code>s = c.MaxElement();</code>	
$\min_i c[i] $	Minimum of absolute values	<code>s = c.MinAbsElement();</code>	
$\max_i c[i] $	Maximum of absolute values	<code>s = c.MaxAbsElement();</code>	
	Minimum and maximum	<code>c.MinAndMaxElement(s_min, s_max);</code>	
$c/\ c\ $	Normalization	<code>c.NormalizedSelf(); b = c.IsNormalized(); c2.NormalizedOf(c1); c2 = c1.Normalized();</code>	<code>c /= c.Norm();</code>
$ c_i $	Elementwise absolute values	<code>c.AbsSelf(); c2.AbsOf(c1); c2 = c1.Abs();</code>	
$\lfloor c_i \rfloor$	Elementwise floor	<code>c.FloorSelf(); c2.FloorOf(c1); c2 = c1.Floor();</code>	
$\lceil c_i \rceil$	Elementwise ceil	<code>c.CeilSelf(); c2.CeilOf(c1); c2 = c1.Ceil();</code>	
$-$	Negation	<code>c.NegationSelf(); c2.NegationOf(c1); c2 = c1.Negation();</code>	<code>c2 = -c1;</code>
v_i	Access vector elements	<code>s = v.Element(i); s = v.at(i); ¶</code>	<code>s = v[i]; s = v(i); ¶</code>
$m_{i,j}$	Access matrix elements	<code>s = m.Element(row, col); s = m.at(row, col); ¶ v = m.Row(row); v = m.Column(col); v = m.Diagonal();</code>	<code>s = m[row][col]; s = m(row, col); ¶ v = m[row];</code>

[†]Operand types: **b** boolean, **s** scalar, **v** vector, **m** matrix, **c** container (**v** or **m**), **cs** container *or* scalar, **cb** container of booleans.

¶Performs a bound check and throws `std::out_of_range` if necessary.

cisstVector: Transformations quick sheet

Revision: 1.18 - Date: 2006/11/21 21:21:53

	Code	Description
Global constants	<code>cmnPI, cmnPI_2, cmnPI_4</code>	$\pi, \pi/2, \pi/4$
Typedefs for rotations & frames	<code>vctRot2</code> (a.k.a. <code>vctMatRot2</code>), <code>vctAnRot2</code> <code>vctRot3</code> (a.k.a. <code>vctMatRot3</code>), <code>vctQuatRot3</code> <code>vctAxAnRot3</code> , <code>vctRodRot3</code> <code>vctFrm3</code> (a.k.a. <code>vctMatFrm3</code>) <code>vctQuatFrm3</code>	Matrix/angle rotation SO(2) Matrix/quaternion rotation SO(3) Axis-angle/Rodriguez rotation SO(3) Frame based on matrix rotation SO(3) Frame based on quaternion rotation SO(3)
Frame components	<code>rotation = frame.Rotation();</code> <code>translation = frame.Translation();</code>	Rotation reference (& and <code>const &</code>) Translation reference (& and <code>const &</code>)
Identity constants	<code>id = TransformationType::Identity();</code> e.g.: <code>vctRot3 id = vctRot3::Identity();</code>	Identity (static method per class)
Inverse & normalization	<code>invTr.InverseOf(tr); invTr = tr.Inverse();</code> <code>tr.InverseSelf();</code>	Inverse transformation
	<code>norTr.NormalizedOf(tr); norTr = tr.Normalized();</code> <code>tr.NormalizedSelf();</code>	Normalized transformation ¶
	<code>b = tr.IsNormalized(tolerance);</code> ††	
Equality <i>See equivalence for non unique representations</i>	<code>b = tr1.Equal(tr2);</code> <code>b = (tr1 == tr2);</code> <code>b = tr1.AlmostEqual(tr2);</code> †† <code>b = tr1.AlmostEqual(tr2, tolerance);</code>	Exactly equal (elementwise) Almost equal (elementwise) With user specified tolerance
Equivalence across non-unique representations, such as axis-angle	<code>b = tr1.AlmostEquivalent(tr2);</code> †† <code>b = tr1.AlmostEquivalent(tr2, tolerance);</code>	Equivalent transformations With user specified tolerance
Conversions between rotations † or frames	<code>TypeA rotA(rotB);</code> <code>TypeA rotA; rotA.From(rotB);</code>	If <code>rotB</code> is normalized convert to <code>TypeA</code> otherwise throw an exception
	<code>TypeA rotA(rotB, VCT_NORMALIZE);</code> <code>TypeA rotA; rotA.FromNormalized(rotB);</code>	Convert to <code>TypeA</code> from a normalized copy of <code>rotB</code>
	<code>TypeA rotA(rotB, VCT_DO_NOT_NORMALIZE);</code> <code>TypeA rotA; rotA.FromRaw(rotB);</code>	Convert to <code>TypeA</code> with neither validation nor modification of <code>rotB</code>
Apply & apply inverse **	<code>tr.ApplyTo(in, out);</code> <code>out = tr.ApplyTo(in); out = tr * in</code> <code>tr.ApplyInverseTo(in, out);</code> <code>out = tr.ApplyInverseTo(in);</code>	Apply to either a vector, a matrix or a transformation of the same type. Note that the methods with <code>(in, out)</code> avoid the creation of a temporary variable.

†TypeA and TypeB can be `vctAnRot2` or `vctMatRot2` for 2D rotations and `vctMatRot3`, `vctQuatRot3`, `vctAxAnRot3` or `vctRodRot3` in 3D.

¶The normalization methods do not modify the angle of `vctAnRot2` and `vctAxAnRot3` nor `vctRodRot3`.

**The rotations `vctAnRot2`, `vctAxAnRot3` and `vctRodRot3` do not support the `Apply*` methods nor the operator `*`.

††Most methods requiring a tolerance use `cmnTypeTraits<value_type>::Tolerance()` by default.