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	`vctFixedSizeVector<_elementType, _size>`	Fixed size vector
	`- vctDouble1 vctDouble2 vctDouble3 vctDouble4 ...`	... of doubles
	`- vct1 vct2 vct3 vct4 vct5 vct6 ...`	... of doubles (implicit)
	`- vctFloat1 vctFloat2 vctFloat3 vctFloat4 vctFloat5 ...`	... of floats
	`- vctInt1 vctInt2 vctInt3 vctInt4 vctInt5 vctInt6 ...`	... of ints
	`- vctChar1 vctChar2 vctChar3 vctChar4 vctChar5 ...`	... of chars
	`vctFixedSizeMatrix<_elementType, _rows, _cols>`	Fixed size matrix
	`- vctDouble2x2 vctDouble3x3 vctDouble2x3 ...`	... of doubles
	`- vct2x2 vct3x3 vct4x4 vct2x3 vct2x4 vct3x4 ...`	... of doubles (implicit)
	`- vctFloat2x2 vctFloat3x3 vctFloat4x4 vctFloat3x2 ...`	... of floats
	`- vctInt2x2 vctInt3x3 vctInt4x4 vctInt3x4 ...`	... of ints
	`vctFixedSizeMatrix<_elementType, _rows, _cols, _order>`	With specific storage order
	`- vctFixedSizeMatrix<double, 3, 3, VCT_ROW_MAJOR> m1;`	- row major [§] (default, $\simeq$ vct3x3)
	`- vctFixedSizeMatrix<double, 3, 3, VCT_COL_MAJOR> m2;`	- col major (Fortran like)
	`vctDynamicVector<_elementType>`	Dynamic vector
	`- vctDoubleVec, vctVec`	... of doubles, doubles (implicit)
	`- vctFloatVec, vctIntVec`	... of floats, ints
	`vctDynamicMatrix<_elementType>`	Dynamic matrix
	`- vctDoubleMat, vctMat`	... of doubles, doubles (implicit)
	`- vctFloatMat, vctIntMat`	... of floats, ints
Constructors	`vct4 v1;`	Default constructor (undefined content)
	`vct4 v2(0.0); vct4 v3(0.1, 0.2, 0.3, 0.4);`	Assign values, all or individually
	`vct4 v4(v1);`	Copy constructor (copy elements)
	`vct2x2 m1;`	Default constructor (undefined content)
	`vct2x2 m2(0.0); vct2x2 m3(0.1, 0.2, 0.3, 0.4);`	Assign values, all or individually (by rows)
	`vct2x2 m4(m1);`	Copy constructor (copy elements)
	`vctVec v1;`	Default constructor, size is zero
	`vctVec v2(12);`	Set size (undefined content)
	`vctVec v3(2, 0.0); vctVec v4(2, 0.1, 0.2);`	Set size & assign values, all or individually
	`vctVec v5(v1);`	Copy constructor (copy elements)
	`vctMat m1;`	Default constructor, size is zero
	`vctMat m2(12, 7);`	Set size; rows and colums (undefined content)
	`vctMat m3(6, 4, VCT_ROW_MAJOR);`	Set size and row major [§] (undefined content)
	`vctMat m4(5, 7, VCT_COL_MAJOR);`	Set size and column major (undefined content)
	`vctMat m5(2, 3, 0.0);`	Set size & assign value to all
	`vctMat m6(3, 5, 0.0, VCT_ROW_MAJOR);`	Set size, row major [§] & assign value to all
	`vctMat m7(2, 3, 0.0, VCT_COL_MAJOR);`	Set size, column major & assign value to all
	`vctMat m8(2, 2, 0.1, 0.2, 0.3, 0.4);`	Set size and assign values by rows
	`vctMat m9(m4);`	Copy constructor (copy elements)
Assignments	`c1.Assign(c2); c1 = c2`	Assign from similar container
	`c.Assign(s0, s1, s2, ...)`	Assign from multiple variables using `va_arg`
	e.g.: `v.Assign((double)0, 1.3, .34, 0.0);`	*Input must be explicitly cast to element type!*
	`c.SetAll(value); c = value;`	Assign `value` to all elements
	`vctRandom(c, s_min, s_max;)`	Fill with random values in $[s_{min}, s_{max}]$
	`bool succeeded = c1.FastCopyOf(c2);`	Shallow copy using `memcpy` [**]
	`bool succeeded = c1.FastCopyOf(c2, false);`	Shallow copy without safety checks
Identity (Eye for I)	`MatrixType::Eye()`	`Eye` for dynamic & fixed size matrices using:
	`- vct4x4 m = vct4x4::Eye();`	- fixed size matrix typedef
	`- vctInt3x3 m = vctFixedSizeMatrix<int, 3, 3>::Eye();`	- fixed size matrix class definition
	`- vctMat m = vctMat::Eye(9);`	- dynamic matrix typedef and dimension (9×9)
	`- vctIntMat = vctDynamicMatrix<int>::Eye(6);`	- 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)	`ContainerType::size_type size;`	Size type (`unsigned int`)
	`ContainerType::index_type index;`	Index type (`unsigned int`)
	`ContainerType::stride_type stride;`	Stride type (`int`)
	`ContainerType::value_type element;`	Type of contained elements
	`ContainerType::reference ref;`	Reference on element (`value_type &`)
	`ContainerType::const_reference constRef;`	Const reference (`const value_type &`)
	`ContainerType::pointer ptr;`	Pointer on element (`value_type *`)
	`ContainerType::const_pointer constPtr;`	Const pointer (`const value_type *`)
	`ContainerType::NormType norm;`	Norm type (`double`)
	`ContainerType::AngleType angle;`	Angle type (`double`)
Iterators (STL like)	`ContainerType::iterator iter;`	Iterator type
	`ContainerType::const_iterator iter;`	Const iterator type
	`ContainerType::reverse_iterator iter;`	Reverse iterator type
	`ContainerType::const_reverse_iterator iter;`	Const reverse iterator type
	`iter = c.begin(); iter = c.end();`	Iterator on first/last element
	`iter = c.rbegin(); iter = c.rend();`	Reverse iterator on last/first element
Memory layout	`size = c.size();`	Length for vectors, rows × cols for matrices
	`nbRows = m.rows(); nbCols = m.cols();`	Number of rows/columns
	`b = c.empty();`	Test if size is zero
	`b = m.IsSquare();`	Test if matrix is square
	`b = m.IsSquare(5);`	Test if square matrix of a given size (e.g. 5)
	`stride = v.stride();`	Stride between elements
	`rowStride = m.row_stride();`	Stride between rows
	`colStride = m.col_stride();`	Stride between columns
	`ptr = c.Pointer();`	Pointer on first element
	`ptr = v.Pointer(i);`	Pointer on i^{th} vector element
	`ptr = m.Pointer(row, col);`	Pointer on $[row, col]$ matrix element
Test indices	`bool b = c.ValidIndex(index);`	Index is lesser or equal to size
	`bool b = m.ValidIndex(row, col);`	Both row and column indices are correct
	`bool b = m.ValidRowIndex(row);`	Row index is valid
	`bool b = m.ValidColIndex(col);`	Column index is valid
Storage	`bool b = m.IsRowMajor();`	Stored row first (C/C++ like - default)
	`bool b = m.IsColMajor();`	Stored column first (Fortran like)
	`bool b = m.IsCompact();`	Uses a compact block of memory , i.e., all elements are in one contiguous block.
	`bool b = m.IsFortran();`	Both compact and column major
	`bool storageOrder = m.StorageOrder();`	Returns storage order (`true` for row major, `false` for column major)
Dynamic vector resizing	`v.SetSize(newSize);`	Destructive size change
	`v.resize(newSize);`	Non destructive size change
Dynamic matrix resizing	`m.SetSize(rows, cols, storageOrder);`	Destructive size change
	`m.resize(rows, cols);`	Non destructive size change
Access vector parts	`v.X() = x; y = v.Y(); v.Z() = z; w = v.W();`	Access elements by name
	`v.XY() = v2; v3 = v.XYZ(); v4 = v.XYZW();`	Access subvectors by name
	`v.XZ(); v.XW(); v.YZ(); v.YW(); v.ZW();`	Access subvectors other than "head"
	`v.YZW();`	
Access matrix parts	`v = m.Row(index); v = m.Column(index);`	Reference to a given row/column
	`v = m.Diagonal(); mt = m.TransposeRef();`	Reference to the diagonal/transpose
Matrix permutations	`m.ExchangeRows(r1, r2);`	Swap the values between two rows/columns
	`m.ExchangeColumns(c1, c2);`	
Elementwise tests	`b = c.IsPositive();`	All elements are greater than zero
	`b = c.IsNonNegative();`	All elements are greater than or equal to zero
	`b = c.IsNegative();`	All elements are lesser than zero
	`b = c.IsNonPositive();`	All elements are lesser than or equal to zero
	`b = c.All();`	All elements are different from zero
	`b = c.Any();`	At least one element is different from zero
Formatted output	`std::string s = c.ToString();`	Output to `std::string`
	`std::cout << c; c.ToStream(std::cout);`	Output to a C++ stream

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	`c.SumOf(c1, cs2);` `c.Add(cs1);`	`c = c1 + cs2;` `c += cs1;`
$-$	Subtraction	`c.DifferenceOf(c1, cs2);` `c.Subtract(cs1);`	`c = c1 - cs2;` `c -= cs1;`

$\times$	Container scalar multiplication	`c1.ProductOf(c2, s);`
		`c1.ProductOf(s, c2);`
		`c.Multiply(s1);`
			`c1 = c2 * s;`
			`c1 = s * c2;`
			`c *= s1;`

	Matrix vector multiplication	`v2.ProductOf(m, v1);` `v2.ProductOf(v1, m);`	`v2 = m * v1;` `v2 = v1 * m;`
	Matrix matrix multiplication [§]	`m.ProductOf(m1, m2);`	`m = m1 * m2;`
	Vector outer product	`m.OuterProductOf(v1, v2);`	
	Elementwise multiplication $c_1[i] = c_2[i] * c_3[i]$ and $c_1[i] *= c_2[i]$	`c1.ElementwiseProductOf(c2, c3);` `c1.ElementwiseMultiply(c2);`	
$\div$	Container scalar division	`c1.RatioOf(c2, s);` `c.Divide(s);`	`c1 = c2 / s;` `c /= s;`
	Elementwise division $c_1[i] = c_2[i]/c_3[i]$ and $c_1[i] /= c_2[i]$	`c1.ElementwiseRatioOf(c2, c3);` `c1.ElementwiseDivide(c2);`	
$\times$ $\wedge$	Cross product (size 3 only) `%` operator has high precedence	`v1.CrossProductOf(v2, v3);` `v1 = vctCrossProduct(v2, v3);`	`v1 = v2 % v3;`
$\circ$ $\bullet$	Dot product (inner product)	`s = v1.DotProduct(v2);` `s = vctDotProduct(v1, v2);`	`s = v1 * v2;`
$+\times$	Accumulation	`c2.AddProductOf(s, c1);`	`c2 += (s * c1);`
$=$	Equal	`b = c1.Equal(cs2);` `cb = c1.ElementwiseEqual(cs2);`	`b = (c1 == cs2);`
$\neq$	Not equal	`b = c1.NotEqual(c2);` `cb = c1.ElementwiseNotEqual(c2);`	`b = (c1 != cs2);`
$\simeq$	Approximately equal	`b = c1.AlmostEqual(c2, tolerance);`	
$<$	Lesser	`b = c1.Lesser(cs2);` `cb = c1.ElementwiseLesser(cs2);`	
$\leq$	Lesser or equal	`b = c1.LesserOrEqual(cs2);` `cb = c1.ElementwiseLesserOrEqual(cs2);`	
$>$	Greater	`b = c1.Greater(cs2);` `cb = c1.ElementwiseGreater(cs2);`	
$\geq$	Greater or equal	`b = c1.GreaterOrEqual(cs2);` `cb = c1.ElementwiseGreaterOrEqual(cs2);`	

$min(c_i, u)$	Clip above upper bound	`c.ClipAbove(s);`
		`c2.ClippedAboveOf(c1, s);`
		`c2.ClippedAboveOf(s, c1);`	

$max(c_i, l)$	Clip below lower bound	`c.ClipBelow(s);`
		`c2.ClippedBelowOf(c1, s);`
		`c2.ClippedBelowOf(s, c1);`	

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

$m_{i,j}$	Access matrix elements	`s = m.Element(row,col);`
		`s = m.at(row, col);` ¶
		`v = m.Row(row);`
		`v = m.Column(col); v = m.Diagonal();`
			`s = m[row][col];`
			`s = m(row, col);` ¶
			`v = m[row];`

†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	`cmnPI`, `cmnPI_2`, `cmnPI_4`	π, $\pi/2$, $\pi/4$

Typedefs for rotations & frames	`vctRot2` (a.k.a. `vctMatRot2`), `vctAnRot2`
	`vctRot3` (a.k.a. `vctMatRot3`), `vctQuatRot3`
	`vctAxAnRot3`, `vctRodRot3`
	`vctFrm3` (a.k.a. `vctMatFrm3`)
	`vctQuatFrm3`
		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	`rotation = frame.Rotation();` `translation = frame.Translation();`	Rotation reference (`&` and `const &`) Translation reference (`&` and `const &`)
Identity constants	`id = TransformationType::Identity();` e.g.: `vctRot3 id = vctRot3::Identity();`	Identity (static method per class)
Inverse & normalization	`invTr.InverseOf(tr); invTr = tr.Inverse();` `tr.InverseSelf();`	Inverse transformation
	`norTr.NormalizedOf(tr); norTr = tr.Normalized();` `tr.NormalizedSelf();`	Normalized transformation ¶
	`b = tr.IsNormalized(tolerance);` ††	

Equality
See equivalence for non unique representations
	`b = tr1.Equal(tr2);`
	`b = (tr1 == tr2);`
	`b = tr1.AlmostEqual(tr2);` ††
	`b = tr1.AlmostEqual(tr2, tolerance);`
		Exactly equal (elementwise)
		
		Almost equal (elementwise)
		With user specified tolerance

Equivalence *across non-unique representations, such as axis-angle*	`b = tr1.AlmostEquivalent(tr2);` †† `b = tr1.AlmostEquivalent(tr2, tolerance);`	Equivalent transformations With user specified tolerance
Conversions between rotations † or frames	`TypeA rotA(rotB);` `TypeA rotA; rotA.From(rotB);`	If `rotB` is normalized convert to `TypeA` otherwise throw an exception
	`TypeA rotA(rotB, VCT_NORMALIZE);` `TypeA rotA; rotA.FromNormalized(rotB);`	Convert to `TypeA` from a normalized copy of `rotB`
	`TypeA rotA(rotB, VCT_DO_NOT_NORMALIZE);` `TypeA rotA; rotA.FromRaw(rotB);`	Convert to `TypeA` with neither validation nor modification of `rotB`

Apply & apply inverse **	`tr.ApplyTo(in, out);`
	`out = tr.ApplyTo(in); out = tr * in`
	`tr.ApplyInverseTo(in, out);`
	`out = tr.ApplyInverseTo(in);`
		Apply to either a vector, a matrix or a transformation of the same type. Note that the methods with `(in, out)` 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.