January 5, 2014 **Inline Math Library** Elizabeth Lowry

Every class and function in this library is either inlined, templated, or both, so all the code is in header files. Unless you #define MATH_LIBRARY_SUPRESS_INLINE before including InlineMath.h.

1. Projects

There are two projects here - the Math library itself (named *Math*) and the unit test library (named *MathTest*).

MathTest includes a few very basic unit testing classes based on our class excercises. It compiles into an executeable that, when run, tests all the math library functionality listed in the assessment PDF and outputs text to the console indicating which tests passed and which ones failed (and why).

Math is composed almost entirely of header files. There is one all-inclusive .cpp file that defines a preprocessor flag for suppressing inlining for non-template functions and member functions of non-template classes, then includes *InlineMath.h*. If compiling as a static library, be sure to define the _LIB flag in the preprocessor settings of this project and include the *StaticMath.h* header file in yours. If compiling as a dynamic library, define the _DLL flag in the preprocessor settings of this project and include the *DynamicMath.h* header file in yours. Alternatively, you can just include *InlineMath.h* in your project instead of compiling this one to let your project inline everything for maximum efficiency. Whichever method you use, make sure you don't have _LIB, _DLL, or MATH_LIBRARY_SUPPRESS_INLINE defined when you include the appropriate header.

2. namespace Math

The *Math* namespace contains the *Matrix* and *Vector* template classes, along with constants *PI*, *DEGREES_IN_A_RADIAN*, *DEGREES_IN_A_CIRCLE*, and *RADIANS_IN_A_CIRCLE*, the values of which should be obvious. The *Math* namespace also includes several template functions:

- *double Degrees(const T& ac_rRadians)* and *double Radians<T>(const T& ac_rDegrees)* allow you to convert angles in radians or degrees (of any data type), respectively, into a double representing degrees or radians, also respectively.
- *T Interpolate(const T& ac_rPointA, const U& ac_rPointB, float a_fProgress)* returns a value interpolated between the first two parameters based on the third. The third parameter is optional - if unspecified, its default value is 0.5.
- *T Modulo(const T& ac_rDividend, const U& ac_rDivisor)* and *T& ModuloAssign(T& a_rDividend, const T& ac_rDivisor)* call the % or %= operators if both parameters are integral types, while calling std::fmod and returning/assigning its result if either parameter is a floating-point type.
- *T NearestPowerOfTwo(const T& ac_rValue)* returns the number closest to the parameter that is equal to 2 raised to a whole number power.
- *T Round(const T& ac_rValue)* returns the whole number closest to the parameter.
- *T Scroll(const T& ac_rValue, const T& ac_rMax, const T& ac_rMin)* returns the result of adding or subtracting the difference between the given max and min from the given value until it is less than the max and greater than or equal to the min. The third parameter is optional - if unspecified, its default value is 0.
- *T ComplexConjugate(const T& ac_rValue)* returns the complex conjugate of the given value. This library does not implement any complex number types, so the only provided implementation simply returns the given value. This function is called by the Matrix class to calculate a matrix's conjugate transpose, which is in turn called to check if a matrix is Hermitian. If you implement a complex number data type and create a Matrix of such values, make sure you define an explicit specialization of this function.

2.1 class Matrix

The *Matrix* class takes three template parameters - a data type (for which the arithmetic operators must be defined, or else it won't compile), an unsigned int indicating the number of rows, and an unsigned int indicating the number of columns. The third parameter is optional - if unspecified, the number of columns is the same as the number of rows. Thus, specifying type *Matrix< double, 3 >* is the same as specifying type *Matrix< double, 3, 3 >*.

A matrix can be constructed from another matrix of any type or dimensions. A matrix can also be constructed from an array with a number of elements of the matrix's data type equal to the number of rows times the number of columns, from a two-dimensional array of the matrix's data type with the same dimensions as the matrix itself, or from an appropriately sized array of appropriately sized row or column vectors (see the *Vector* class). Finally, a matrix can be constructed from a single value, which all elements will be set to, or from a pair of values, the first specifying the value of the elements along the diagonal and the second specifying the value of all other elements. The default constructor creates a matrix filled with zeros.

Every matrix class provides static functions *Zero()* and *Identity()*, which return const references to (respectively) a zero matrix and an identity matrix with dimensions equal to the smaller of the dimensions of the matrix type called from. Every matrix class also provides public constants *ROWS* and *COLUMNS*, the values of which should be obvious.

Elements in the array can be accessed using square bracket (for example, *T& rValue = oArray[uiRow][uiColumn];*) or by calling the member function *T& At(unsigned int a_uiRow, unsigned int a_uiColumn)*. Both methods come with const equivalents for use with const matrix objects.

Copies of rows or vectors in column form can be obtained using the *Row(unsigned int a_uiIndex)* and *Column(unsigned int a_uiIndex)* functions. Submatrices can be generated using the *MinusRow(unsigned int a_uiIndex)*, *MinusColumn(unsigned int a_uiIndex)*, and *MinusRowAndColumn(unsigned int a_uiRow, unsigned int a_uiColumn)* functions. The *Transpose()* and *ConjugateTranspose()* return the transpose and conjugate transpose, respectively, of the matrix (for non-complex data types, these are the same thing)..

The *Shift(unsigned int a_uiRight, unsigned int a_uiDown)* function shifts every element in the matrix by the given number of spaces. *Round()* adjusts every element in the matrix to the nearest whole number. Both functions return a reference to the matrix object.

The *IsInvertable()*, *IsOrthogonal()*, *IsIdempotent()*, and *IsHermitian()* functions all return boolean values, the significance of which should be obvious from the function names. In a non-square matrix, *IsInvertable()* returns true if a pseudo-inverse (left-inverse for a matrix with more rows than columns, right-inverse for a matrix with more columns than rows - see Wikipedia for what these terms mean).

The *Inverse()* function returns a matrix of floating-point type - float if the matrix contains floats, long double if the matrix contains long doubles, and double for all other types. If the matrix is invertible, then the resulting matrix will be its inverse (left, right, or true, depending on matrix dimensions); if the matrix is not invertible, then the resulting matrix will be a zero matrix. The *Inverse(bool& a_rbInvertable)* function returns the same, and sets the reference parameter to true if the matrix was invertable and false if it wasn't. A third version of the function takes a reference to a matrix of the appropriate size and type, sets its elements to those of the inverse, and returns true if possible; otherwise the function returns false and makes no changes to the matrix passed in by reference. Finally, the *Invert()* function sets the matrix's own elements to those of its inverse and returns true if it is a square invertible matrix, changing nothing and returning false otherwise - this function should be used with caution with integer data type matrices, since a matrix of whole numbers may have an inverse containing non-whole numbers which could be truncated.

The ==, !=, +, -, +=, and -= operators are defined for the Matrix class and accept Matrix objects of the same data type and dimensions. The *, /, *=, /=, %, and %= operators are defined for the Matrix class and accept scalar values of any type, though the result will never be a different type of matrix. The unary negation operator, -, is equivalent to multiplying by -1. The * and / operators also accept matrices of the same data type and of appropriate dimensions - for multiplication, the number of columns in the left operand must equal the number of rows in the right operand, while for division, the matrices must have the same number of rows (division means multiplying the left operand by the inverse of the right operand, with an exception thrown if the right operand is not invertible). Similarly, the *= and /= operators accept square matrices of the appropriate dimensions.

Outside the math namespace, the * and / operators are defined to accept scalar left operands and matrix right operands, returning matrices of the same type as the matrix operand. The << operator is defined for a std::ostream left operand and a matrix right operand, and outputs the matrix in format similar to "{ { 1, 2 }, { 3, 4 } }".

2.2 class Vector

The *Vector* class takes three template parameters - a data type, an unsigned int indicating size, and a boolean indicating whether or not the vector is a row vector. The bool is optional - if unspecified, the vector is a row vector.

The *Vector* class publicly inherits from the *Matrix* class of corresponding type and dimension (*Matrix< T, 1, N >* for *Vector< T, N >*, or *Matrix< T, N, 1 >* for *Vector< T, N, false >*). Some Matrix functionality is hidden, however.

Instead of using two sets of brackets to access elements, one is used - *oMyVector[1]* returns a reference to the first element in the vector whether it's a row vector or a column vector. Similarly, the *At* function only accepts one index parameter, instead of one each for row and column as in the parent class, and the *MinusRow*, *MinusColumn*, and *MinusRowAndColumn* functions are hidden in favor of a *MinusElement* function. The *Row() and Column()* functions return row and column version copies (respectively) of the entire vector. The *Shift* function, like the *At* and *MinusElement* functions, only accepts one index parameter.

A vector can be constructed like a matrix if given an object or reference of merely matrix type. Given another vector, however, it will copy values by element index, not by row and column indices, whether or not the vector's third template parameter is the same. Similarly, the +, -, +=, and -= operators will add by element index instead of by row and column if their operand is another vector - this means that row and column vectors can be added together as if they were the same type.

The vector class defines constants *SIZE and IS_ROW*, of obvious meaning, and a static function *Unit(unsigned int a_uiAxis)* that returns const references to unit vectors (*Unit(0)* meaning the x-axis for example). Functions *T Dot(const Vector& ac_roVector)* and *Vector Cross(const Vector& ac_roVector)* return the dot and cross products, respectively, of the vector with another vector.

The dot and cross product functions, like the addition and subtraction operators, can take parameters of either row or column type, as long as they have the same length and data type. For 2D vectors, the parameter of the cross product is ignored and the result is the current vector rotated 90 degrees, the same result one would get by crossing the vector with the z-axis. For dimensions higher than three, the result is the determinant of a matrix in which the elements of the first row are unit vectors, the elements of the second row are the elements of the current vector, the elements of the third are those of the parameter, and the rest of the rows are those of an identity matrix (see Wikipedia).

Finally, the *Magnitude()*, *MagnitudeSquared()*, and *Normal()* functions return the obvious, while the *Normalize()* function divides all elements of the vector by its magnitude and returns a reference to the vector.

3. namespace Color

The *Color* namespace contains the *ColorVector* class, the *Hex* union type, and a number of constants for different color values.

The *Hex* union type has members of three types: a *FourChannelInt* (typedef for a 32-bit integer) *i*, an anonymous struct containing four *Channel* (typedef for an 8-bit integer) members *a*, *r*, *g*, and *b*, and a *Channel* array of size four. Thanks to variations in system endianness, the elements of the array may not always correspond to the same parts of the int, but thanks to the magic of macros, the anonymous struct members always match up - *a* corresponds to the most significant byte, *r* to the second-most, *g* to the third-most, and *b* to the least significant. Thus, 0xFFFF7F00 will always refer to opaque orange.

The color constants are all of *Hex* type. Most of them are grouped into subsidiary namespaces - the constants in the *GrayScale*, *Red*, *Green*, and *Blue* namespaces contain different monochrome shades that can be combined using the bitwise OR operator to create different shades of opaque color, while the *Opacity* namespace contains constants for different transparency levels that can be combined with an opaque color using bitwise AND. The *VGA* and *ColorWheel* namespaces contain multiple named colors.

The *ColorVector* class is a *Vector< Channel, 4 >* with a set of public *Channel* reference members *a*, *r*, *g*, and *b* pointing to the elements 0-3, respectively, of the vector. Additionally, *ColorVector* class defines the bitwise operators &, |, ^, ~, <<, >>, &=, |=, ^=, <<=, and >>=, along with a conversion operator for converting to *Hex* type, constructors taking a *Hex* value or four *Channel* or float parameters (the floats are multiplied by 0xFF and converted to *Channel* values), and functions *fAlpha()*, *fRed()*, *fGreen()*, and *fBlue()* for returning the elements as floats in the range 0 - 1 instead of *Channel* values in the range 0x00 - 0xFF.

4. namespace Plane and namespace Space

The *Plane* and *Space* namespaces contain 2D and 3D versions, respectively, of the same types and functions.

4.1 Classes

The *PointVector* class is a *Vector< double, 2 >* or *Vector< double, 3 >*, depending on namespace, with public reference members *x*, *y*, and (in *Space*) *z* pointing to the corresponding elements in the vector. A *PointVector* has an additional constructor taking 2 or 3 (depending on namespace) double values to specify each element. The *PointVector* class for each namespace has a static function *Origin()* that returns a const reference to (0,0) or (0,0,0). Within each namespace, *Point* is a typedef for the *PointVector* class. Outside, *Plane::PointVector* has typedefs *PointVector2D* and *Point2D*, while *Space::PointVector* has typedefs *PointVector3D* and *Point3D*.

The *HomogeneousVector* class is a *Vector< double, 3 >* or *Vector< double, 4 >* with public reference members *x*, *y*, *z* (in *Space*), and *h* pointing to the corresponding elements in the vector. A point vector has an additional constructor taking a set of double values or a PointVector, along with an optional double value for *h* (set to 1 if unspecified). The *HomogeneousVector* class also offers an *Origin* function, this time returning (0,0,1) or (0,0,0,1), depending on namespace. Inside each namespace, *HomogeneousVector* has typedef *HVector*. Outside,, *Plane::HomogeneousVector* has typedefs *HomogeneousVector2D* and *HVector2D*, while *Space::HomogeneousVector* has typedefs *HomogeneousVector3D* and *HVector3D*.

The *HomogeneousVector* class offers a *Homogenize()* function that divides all elements by the *h* element (if it isn't zero) and overrides the multiplication and division operators for multiplying and dividing by matrices to call *Homogenize()* before applying the transformation. The *Magnitude()*, *MagnitudeSquared()*, *Normal()*, and

Normalize() functions are also overridden so that the magnitude calculated is that of the vector as a point vector (and converting to a point vector results in a vector with the homogenized values of the other elements).

4.2 Transformation-generating functions

Each namespace has typedefs for square matrices for transforming each type of vector. *Matrix< double, 2 >* or *Matrix< double, 3 >* is typedef'd to *PointTransform* (and, in turn, *Plane::PointTransform* to *PointTransform2D* and *Space::PointTransform* to *PointTransform3D*). *Matrix< double, 3 >* or *Matrix< double, 4 >*, used for transforming homogeneous vectors, is typedef'd to *Transform* (and *Plane::Transform* to *Transform2D* and *Space::Transform* to *Transform3D*). Since the vectors in use are row vectors, translations can be applied simply by using the *= operator.

The *PointScaling* and *Scaling* functions create transformation matrices that, when applied to the corresponding vector type, scale the vector coordinate. Each version can take a single value for uniform scaling, separate values for each coordinate element, or a *PointVector* object for each coordinate. The *HVector* version, *Scaling*, can take an optional *PointVector* parameter after the other(s) to specify a point other than the origin about which the scaling should be applied. For example, scaling (1,1) about the origin by 2 would yeild (2,2), while scaling the homogeneous version, (1,1,1), about the point (3,2) by 2 would yield (-1,0,1).

The *PointRotation* and *Rotation* functions create transformation matrices that, when applied, rotate a vector by a certain number of radians (*PointDegreeRotation* and *DegreeRotation* rotate by degrees). In the *Space* namespace, the rotation function takes an optional second parameter, a point vector specifying the axis about which the rotation should take place (the z-axis, if unspecified). The *HVector* versions, *Rotation* and *DegreeRotation*, take another optional parameter after that specifying a point vector other than the origin about which the rotation should be applied. Finally, all versions take a final optional boolean parameter specifying whether or not the rotation should be clockwise (false by default, meaning counter-clockwise).

The *Translation* functions generate *HVector* transformations that move the coordinates. The parameters are either a double for each coordinate's movement amount, a point vector specifying the movement amount, or a double indicating an elapsed time and an optional point vector indicating the velocity (an x unit vector, if unspecified).

Finally, the *Space* namespace offers two functions for projecting coordinates onto a screen. *PerspectiveProjection(double a_dNearDistance, double a_dFarDistance, const Point2D& ac_roScreenSize, const Point2D& ac_roScreenCenter)* and *ParallelProjection(double a_dNearDistance, double a_dFarDistance, const Point2D& ac_roScreenSize, const Point2D& ac_roScreenCenter)* both project points onto a screen on the plane z = -*a_dNearDistance*, centered at the origin, with a size given by *ac_roScreenSize*.

Points on the screen itself have their z coordinates mapped to zero, while those on the plane z = -a_dFarDistance have their z coordinates mapped to 1. Points projecting to the upper right corner of the screen have their x and y coordinates mapped to 0.5 while those projecting to the lower left corner have them mapped to -0.5. Then, the optional screen center coordinates (given in terms of the final coordinate system) are subtracted from the x and y coordinates - if the screen center is (0, 1), then we want the points one screen above the origin-centered screen. The end result is a set of coordinates indicating where on a display screen points should be drawn and their relative distance from the screen.

PerspectiveProjection mimics a lens, so transforming a point behind the "camera" at the origin results in a point that appears to be in front of it, much like a hologram in the physical world, so discard points in front of the near plane before transforming en masse. *ParallelProjection* doesn't have this problem, since it mimics the effect of a camera with infinite focal length.