MathsLibrary reference document

By Leonard Andrew Spencer, 2017

# Classes

**Vector 3**

**Matrix 10**

# Vector

template<size\_t DIM> class Vector

Vectors are spatial vectors, represented as an array of the vector’s components.

Vector2, Vector3, and Vector4 are used as aliases for Vector<2>, Vector<3> and Vector<4>

## Template Parameters

DIM Dimensions of the vector.

## Member Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Member** | **Type** | **Access** |  |
| m\_component | float[DIM] | protected | Components of the vector. |

## Member Functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Page** |  |
| (constructor) | 4 | Initializes vector with components passed as arguments |
| (destructor) | 4 | Destroys the vector |
| operator[] | 4 | Access specified component |
| operator float\* | 4 | Cast vector as array of floats, allowing access to its components |
| operator Vector<D> | 5 | Cast vector as vector of different dimensions by creating a new vector |
| dot | 5 | Returns product of dot multiplication |
| cross | 5 | Returns product of cross multiplication for Vector<3> and Vector<4> |
| magnitudeSquared | 6 | Returns square of vector’s magnitude |
| magnitude | 6 | Returns vector’s magnitude |
| normalise | 6 | Scales vector to a unit vector |
| compareMagnitude | 6 | Compares the vector’s magnitude to the absolute value of a float |
| isZeroVector | 9 | Checks if all components are zero |
| isInfinite | 9 | Check if any components are infinite |
| isNAN | 9 | Checks if any components are NaN |

## Non-member Functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Page** |  |
| operator+ | 7 | Add vectors |
| operator- | 7 | Subtract vector |
| operator\* | 7 | Scale vector by float |
| operator> | 8 | Vector magnitude greater than |
| operator< | 8 | Vector magnitude less than |
| operator>= | 8 | Vector magnitude greater than or equal to |
| operator<= | 8 | Vector magnitude less than or equal to |
| operator== | 8 | Vector components are equal |
| operator!= | 8 | Vector components are not equal |

## Vector::Vector

1. Vector(); *default constructor*
2. template<typename... Args>  
   Vector(typename std::enable\_if<(sizeof...(Args)+1 == DIM), float>::type x, Args... args);

Constructs a Vector object, initializing its components depending on constructor used.  
(1) constructs a zero vector, with all components equal to 0.0f.  
(2) initializes components with arguments passed, starting with x. Enable\_if is used so that each Vector class compiles with a constructor taking as many arguments as it has components

### Template Parameters

… Args Parameter pack allowing enable\_if to check total number of arguments

### Function Parameters

x Float to be used as the first component of the vector.

args Function parameter pack containing additional vector components

## Vector::~Vector

~Vector();

Destroys the vector.

## Vector::operator[]

1. float& operator[](size\_t n);
2. const float& operator[](size\_t n) const;

Returns a reference to the component at position n. (2) returns a constant reference

### Function Parameters

n Position of component in array. The first component is at position 0

### Return Value

Reference to float. If constant version is used, constant reference to float

### Exceptions

Throws range exception if n is greater than or equal to vector size

## Vector::operator float\*

explicit operator float\*();

Casts vector as a pointer to its first component, allowing it to be treated as an array

### Return Value

float\* pointing to first component of the vector.

## Vector::Vector<D>

template<size\_t D>

explicit operator Vector<D>(;

Casts vector as a vector of different dimensions. The components of this vector are copied to a newly constructed vector. If the new vector has more dimensions, the components for those dimensions are left as 0.

### Template Parameters

D Dimensions of vector class being cast to

### Return Value

Vector<D> object with components copied from this vector for all shared dimensions

## Vector::dot

float dot(const Vector<DIM>& b) const;

Calculates the dot product of this vector and vector b of same dimensions. The dot product is calculated by multiplying the corresponding components of each vector, and adding the products together.

### Function Parameters

b Reference to the other vector of the dot product

### Return Value

float equal to dot product of this vector and b;

## Vector::cross

template <std::size\_t D = DIM>

typename std::enable\_if<D == 3||D==4, Vector<D>>::type cross(const Vector<D>& b) const;

Calculates the cross product of this vector and vector b of the same dimensions. As cross products are only well defined for 3 dimensional vectors, this function only exists for Vector<3> and Vector<4> (with the Vector<4> being treated as a 3D vector in homogeneous coordinates with a w element of 0)

### Template Parameters

D Dimensions of this vector. Used by enable\_if to prevent this function from being compiled for Vector classes of invalid dimensions

### Function Parameters

b Reference to the other vector of the cross product

### Return Value

Vector<D> equal to cross product of this vector and b. If this is a Vector<4>, its final component will have a value of 0.

## Vector::magnitudeSquared

float magnitudeSquared() const

Calculates the square of the vector’s magnitude. This is equal to the sum of the square of each component. This value can be used to compare the vector’s magnitude to other values without having to calculate a square root.

### Return Value

float equal to the square of this vector’s magnitude

## Vector::magnitude

float magnitude() const;

Calculates the magnitude of this vector. This is done by taking the square root of magnitudeSquared.

### Return Value

Float equal to this vector’s magnitude

## Vector::normalise

bool normalise();

Converts this vector to its normalised form, by scaling it to a magnitude of 1. The inverse of the vector’s magnitude is calculated, and all components are multiplied by this value. If the vector’s magnitude is zero, infinite, or NaN, instead nothing happens.

### Return Value

True if vector can be normalized. False if zero vector, any component is infinite or not a number, or an overflow makes magnitude infinite.

## Vector::compareMagnitude

int compareMagnitude(float f) const;

Compares the vector’s magnitude with the absolute value of the float passed. This is done by comparing magnitudeSquared to the square of f.

### Function Parameters

f value being compared to the vector’s magnitude

### Return Value

Returns 0 if the vector’s magnitude is equal to the absolute value of the float. Otherwise, returns 1 if the magnitude is greater, and -1 if the float is greater.

## isZeroVector

bool isZeroVector() const;

Checks if all components of vector are zero.

### Return Value

Returns true if all components are zero, false if any are non-zero.

## isInfinite

bool isInfinite() const;

Checks if any components of vector are infinite

### Return Value

Returns true if any component is infinite, false otherwise.

## isNAN

## bool isNAN() const;

Checks if any component of the vector is not a number

### Return Value

Returns true if any component is not a number, false otherwise.

## operator+ (Vector)

template<size\_t DIM>

Vector<DIM> operator+(const Vector<DIM>& a, const Vector<DIM>& b);

Adds two vectors with the same dimensions together. Each component of the returned vector is equal to the sum of the corresponding components of the vectors

### Template Parameters

DIM dimensions of the vectors

### Function Parameters

a, b Vectors to be added

### Return Value

Vector equal to the sum of a and b

## operator- (Vector)

template<size\_t DIM>

Vector<DIM> operator-(const Vector<DIM>& a, const Vector<DIM>& b);

Subtracts one vector from another. Each component of the returned vector is equal to the difference between the corresponding components of the vectors.

### Template Parameters

DIM dimensions of the vectors

### Function Parameters

a Vector being subtracted from

b Vector to be subtracted from a

### Return Value

Vector equal to the difference between a and b

## operator\* (Vector)

1. template<size\_t DIM>  
   Vector<DIM> operator\*(const Vector<DIM>& v, float f)
2. template<size\_t DIM>  
   Vector<DIM> operator\*(float f, const Vector<DIM>& v)

Multiplies a vector by a float. Each component of the returned vector is equal to the corresponding component in the vector multiplied by the float.

### Template Parameters

DIM Dimensions of the vector

### Function Parameters

v Vector to be multiplied

f Value to multiply the vector by

### Return Value

Vector equal to v scaled by f

## operator>,< (Vector)

1. template<size\_t DIM>  
   bool operator>(const Vector<DIM>& a, const Vector<DIM>& b);
2. template<size\_t DIM>  
   bool operator<(const Vector<DIM>& a, const Vector<DIM>& b);

Compares the magnitude of two vectors, using the squares of their magnitudes.

### Template Parameters

DIM dimensions of the vectors

### Function Parameters

a, b Vectors to be compared

### Return Value

1. True if the magnitude of a is greater than that of b, false otherwise
2. True if the magnitude of a is less than that of b, false otherwise

## operator==,!= (Vector)

1. template<size\_t DIM>  
   bool operator==(const Vector<DIM>& a, const Vector<DIM>& b)
2. template<size\_t DIM>  
   bool operator!=(const Vector<DIM>& a, const Vector<DIM>& b)

Compares the components of two vectors. Vectors are equal if all corresponding components are equal.

### Template Parameters

DIM dimensions of the vectors

### Function Parameters

a, b Vectors to be compared

### Return Value

1. Returns true if the corresponding components of a and b are equal, false otherwise
2. Returns true if any component in a is not equal to a the corresponding component in b, false otherwise

# Matrix

template<size\_t ORDER> class Matrix

Matrix is a template class for square matrices. The elements of the matrix are stored in column-major order.

Matrix2, Matrix3, and Matrix4 are aliases for Matrix<2>, Matrix<3> and Matrix<4>

## Template Parameters

ORDER Order of the matrix

## Member Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Member** | **Type** | **Access** |  |
| m\_axis | Vector<ORDER>[ORDER] | protected | Columns of matrix. In union with m\_element |
| m\_element | Float[ORDER][ORDER] | protected | Elements of matrix. In union with m\_axis |

## Member Functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Page** |  |
| (constructor) | 10 | Initializes matrix with elements passed as arguments |
| (destructor) | 10 | Destroys the matrix |
| operator[] | 10 | Access specified column |
| operator float\* | 10 | Cast matrix as pointer to float |
| operator\* | 11 | Multiplies vector or matrix by this matrix |
| setIdentity | 11 | Sets matrix as identity matrix |
| setRotate | 11 | Sets Matrix<2> as transformation matrix rotating by given angle |
| setRotateX | 11 | Sets Matrix<3> or Matrix<4> as transformation matrix rotating around X axis |
| setRotateY | 11 | Sets Matrix<3> or Matrix<4> as transformation matrix rotating around Y axis |
| setRotateZ | 11 | Sets Matrix<3> or Matrix<4> as transformation matrix rotating around Z axis |
| setEulerRotate | 12 | Sets Matrix<3> or Matrix<4> as transformation matrix rotating to an orientation defined by Euler angles |
| setTaitBryanRotate | 12 | Sets Matrix<3> or Matrix<4> as transformation matrix rotating to an orientation defined by Euler angles given in Tait-Bryan notation |
| getEulerOrientation | 14 | Calculates Euler angles for the orientation of this matrix’s axes |
| getTaitBryanOrienation | 14 | Calculates Euler angles given in Tait-Bryan notation for the orientation of this matrix’s axes |
| calculateInverse | 12 | Calculates inverse matrix for this matrix |
| transformByInverse | 13 | Multiplies vector or matrix by the inverse of this matrix |

## Protected Member Functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Page** |  |
| invertTransform | 13 | Performs row operations on matrix, passed as array of floats in column major order, equivalent to multiplying it by the inverse of this matrix |
| areColumnsEqual | 14 | Checks if two matrix columns or vectors, passed as float arrays, are equal |
| swapRows | 11 | Swaps rows of matrix passed as float array |
| multiplyRow | 11 | Multiplies all elements in matrix row by a float |
| addRow | 11 | Adds matrix row to another matrix row |

## Matrix::Matrix

1. Matrix(); *default constructor*
2. template<typename... Args>  
   Matrix(typename std::enable\_if<(sizeof...(Args)+1 == (ORDER\*ORDER)), float>::type xx, Args... args);

Constructs a Matrix object. (1) constructs an empty matrix, with all elements initialized to 0.0f. (2) initializes elements based on the arguments passed, starting with xx. Enable\_if is used so that each Matrix class compiles with a constructor taking as many arguments as it has elements.

### Template Parameters

…Args Parameter pack allowing enable\_if to check total number of arguments

### Function Parameters

xx Float to be used as the first component of the vector.

Args Function parameter pack containing additional elements

## Matrix::~Matrix

~Matrix()

Destroys the matrix

## Matrix::operator[]

1. Vector<ORDER>& operator[](size\_t n);
2. const Vector<ORDER>& operator[](size\_t n) const;

Returns a reference to the column at index n. (2) returns a constant reference

### Function Parameters

n Position of column in matrix. The first column is at index 0

### Return Value

1. Reference to Vector<ORDER>
2. Constant reference to Vector<ORDER>

### Exceptions

Throws range exception if n is greater than or equal to matrix order

## Matrix::operator float\*

explicit operator float\*();

Casts matrix as pointer to its first element. As a column-major matrix, the element at column i and row j can be found at index [i\*ORDER + j].

### Return Value

float\* pointing to first element of matrix

## Matrix::operator\*

1. Vector<ORDER> operator\*(const Vector<ORDER>& v) const;
2. Matrix<ORDER> operator\*(const Matrix<ORDER>& m) const;

Multiplies a vector or matrix by this matrix. Since the columns of a matrix can be referenced as vectors, (2) calls (1) for each column of matrix m.

### Function Parameters

v Vector to be multiplied by this matrix

m Matrix to be multiplied by this matrix

### Return Value

1. A new Vector<ORDER> object, equal to v transformed by this matrix
2. A new Matrix<ORDER> object, equal to m transformed by this matrix

## Matrix::setIdentity

void setIdentity();

Sets the matrix to the identity matrix of its order. An identity matrix has all diagonal elements equal to 1, and all other elements 0.

## Matrix::setRotate

template<size\_t ORD = ORDER>  
typename std::enable\_if<ORD == 2, void>::type setRotate(float angle);

Sets this Matrix<2> to a transformation matrix rotating by angle.

### Template Parameters

ORD Order of the matrix. Used by enable\_if to prevent this function from being compiled for Matrix classes of invalid order

### Function Parameters

angle Angle of rotation

## Matrix::setRotateX,setRotateY,setRotateZ

1. template<size\_t ORD = ORDER>  
   typename std::enable\_if<ORD == 3 || ORD == 4, void>::type setRotateX(float angle);
2. template<size\_t ORD = ORDER>  
   typename std::enable\_if<ORD == 3 || ORD == 4, void>::type setRotateY(float angle);
3. template<size\_t ORD = ORDER>  
   typename std::enable\_if<ORD == 3 || ORD == 4, void>::type setRotateZ(float angle);

Sets this Matrix<3> or Matrix<4> to a transformation matrix rotating by angle. (1) rotates around the X axis, (2) rotates around the Y axis, and (3) rotates around the Z axis.

### Template Parameters

ORD Order of the matrix. Used by enable\_if to prevent this function from being compiled for Matrix classes of invalid order

### Function Parameters

angle Angle of rotation.

## Matrix::setEulerRotate,setTaitBryanRotate

1. template<size\_t ORD = ORDER>  
   typename std::enable\_if<ORD == 3 || ORD == 4, void>::type setEulerRotate(float alpha, float beta, float gamma);
2. template<size\_t ORD = ORDER>  
   typename std::enable\_if<ORD == 3 || ORD == 4, void>::type setTaitBryanRotate(float yaw, float pitch, float roll);

Sets this Matrix<3> or Matrix<4> to a transformation matrix rotating to the orientation specified by Euler angles. For (1) these are in Proper Euler form with z-x-z axes, and for (2) in Tate-Bryan form with z-y-x axes.

### Template Parameters

ORD Order of the matrix. Used by enable\_if to prevent this function from being compiled for Matrix classes of invalid order

### Function Parameters

alpha Angle between starting x axis and line of nodes

beta Angle between starting z axis and final z axis

gamma Angle between line of nodes and final x axis

yaw Rotation around z axis

pitch Rotation around y axis

roll Rotation around x axis

## Matrix::getEulerOrientation,getTaitBryanOrientation

1. template<size\_t ORD = ORDER>  
   typename std::enable\_if<ORD == 3 || ORD == 4, std::tuple<float, float, float, bool>>::type getEulerOrientation();
2. template<size\_t ORD = ORDER>  
   typename std::enable\_if<ORD == 3 || ORD == 4, std::tuple<float, float, float, bool>>::type getTaitBryanOrientation();

Calculate Euler angles for the orientation of a Matrix<3> or Matrix<4>. For (1) these are in Proper Euler form with z-x-z axes, and for (2) these are in Tait-Bryan form with z-y-x axes. In both cases, the axes of the matrix are copied and normalised. Based on the components of these axes, the angles can be calculated

### Template Parameters

ORD Order of the matrix. Used by enable\_if to prevent this function from being compiled for Matrix classes of invalid order

### Return Value

Returns a tuple of three float values, and a boolean indicating success. If the Euler angles could not be calculated (because the axes could not be normalized to an orthonormal basis), the floats will all be 0 and the boolean will be false

1. Returns a tuple containing the alpha, beta, and gamma angles. Alpha is within the range [-pi,pi] radians. Beta is within the range [0,pi] radians. Gamma is within the range[-pi,pi] radians, however if Beta is 0 or pi, Alpha and Gamma are gimbal locked, and Gamma is 0.
2. Returns a tuple containing the yaw, pitch, and roll angles. Yaw is within the range [-pi,pi] radians. Pitch is within the range [-pi/2,pi/2]. Roll is within the range [-pi,pi] radians, however if Pitch is –pi/2 or pi/2, Yaw and Roll are gimbal locked, and Roll is 0.

## Matrix::calculateInverse

bool calculateInverse(Matrix<ORDER>& dest);

If this matrix is invertible, copies the inverse of this matrix to dest. This is done by passing an identity matrix to invertTransform. If invertTransform returns true, the transformed identity matrix is copied to dest. This may be useful if multiple vectors need the same inverse transformation applied to them, as multiplying the vector by a matrix is quicker than inverting a matrix (O(n2) vs O(n3))

### Function Parameters

dest Reference to Matrix object the inverse matrix will be written to

### Return Value

Returns true if the matrix is invertible, or false if it is singular or too poorly conditioned to invert

## Matrix::transformByInverse

1. bool transformByInverse(Matrix<ORDER>& target);
2. bool transformByInverse(Vector<ORDER>& target);

If this matrix is invertible, transforms target by the inverse of this matrix. This is done by passing a copy of target to invertTransform. If invertTransform returns true, target will be overwritten by a transformed version of itself.

### Function Parameters

target Reference to Matrix or Vector to perform inverse transformation on

### Return Value

Returns true if the matrix is invertible, or false if it is singular or too poorly conditioned to invert

## Matrix::invertTransform

template<size\_t RESULT\_COLUMNS>  
bool invertTransform(float\* result);

Uses Gaussian elimination to convert a copy of this matrix to the identity matrix. Performs the same row operations on result. These row transformations are equivalent to multiplying a matrix by the inverse of this matrix, and so this function can be used to calculate that inverse matrix (if result is the identity matrix) or to transform a vector or matrix by the inverse of this matrix.

To perform this elimination, the function loops over each column in the matrix. If all values in a column are zero, the matrix is singular and the function returns false. Otherwise, the pivot element of the column is checked, and if its value is zero the row is swapped with a lower row that has a nonzero value in the column. Then, all elements in the row are divided by the value of the pivot element, changing the pivot element to 1. Finally, the row containing the current pivot element is subtracted from all other rows, until all over elements in the column have a value of zero. After this is done for all columns, the matrix will be reduced to the identity matrix.

### Template Parameters

RESULT\_COLUMNS Number of columns result has. 1 if result is a vector (treating a vector as a single column matrix) or ORDER if transforming a matrix

### Function Parameters

result Column major matrix or vector cast as float pointer. The order of this matrix, or dimensions of this vector, must be equal to ORDER. Even if invertTransform fails, the values in result may be modified.

### Return Value

Returns true if the matrix is reduced to the identity matrix, or false if the elimination algorithm fails at any step

## Matrix::areColumnsEqual

template<size\_t ROWS>

static bool areColumnsEqual(float\* first, float\* second);

Checks if two float arrays have equal values.

### Template Parameters

ROWS Number of elements to check in each array

### Function Parameters

first, second Pointer to the first element of arrays to compare

### Return Value

Returns true if the members of each array (up to ROW-1) are equal. False otherwise.

## Matrix::swapRows

template<size\_t ROWS, size\_t COLUMNS>

static void swapRows(float\*\* theMatrix, size\_t first, size\_t second);

Swaps the rows in a column major ROWSxCOLUMNS matrix.

### Template Parameters

ROWS Number of elements in each column

COLUMNS Number of columns in matrix.

### Function Parameters

theMatrix Pointer to first element of matrix to have rows swapped. If COLUMNS is 1, this could be a pointer to the first element of a Vector object

first, second Index of rows to be swapped

### Exceptions

Throws range exception if first or second is greater than or equal to ROWS

## Matrix::multiplyRow

template<size\_t ROWS, size\_t COLUMNS>

static void multiplyRow(float\*\* theMatrix, size\_t row, float factor);

Multiplies all elements in a row of a column major ROWSxCOLUMNS matrix by some given value.

### Template Parameters

ROWS Number of elements in each column

COLUMNS Number of columns in matrix.

### Function Parameters

theMatrix Pointer to first element of matrix to have rows swapped. If COLUMNS is 1, this could be a pointer to the first element of a Vector object

row Index of row to multiply

factor Value which multiplies all elements in row

### Exceptions

Throws range exception if row greater than or equal to ROWS

## Matrix::addRow

template<size\_t ROWS, size\_t COLUMNS>

static void addRow(float\*\* theMatrix, size\_t sourceRow, size\_t targetRow, float factor);

Adds each element in one row, multiplied by a constant factor, to the corresponding element in another row.

### Template Parameters

ROWS Number of elements in each column

COLUMNS Number of columns in matrix.

### Function Parameters

theMatrix Pointer to first element of matrix to have rows swapped. If COLUMNS is 1, this could be a pointer to the first element of a Vector object

sourceRow Index of row to take values from

targetRow Index of row to which values will be added

factor Value which multiplies all elements in row

### Exceptions

Throws range exception if sourceRow or targetRow greater than or equal to ROWS