FASTSERIES

FAST OBJECT IMAGING LIBRARY
USER'S MANUAL

FAST CAPTURE
FAST PROCESSING
FAST RESULTS

FASTSERIES PCI BOARD

FastVision
FastImage 1300
FastFrame 1300

FAST SERIES PMCS

FastMem
Fast4 1300
Fast I/O 1300

30002-00162

COPYRIGHT NOTICE

Copyright ã 2002 by Alacron Inc.

Document Name: Fast Object Imaging Library User's Manual

Document Number: 30002-00162

Revision History: 1.0 April 25, 2002

Trademarks:

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
Fax: 603-891-2745

Web Site:
http://www.alacron.com/

Email:
sales@alacron.com, or support@alacron.com

TABLE OF CONTENTS

MANUAL FIGURES & TABLES

OTHER ALACRON MANUALS

Alacron manuals cover all aspects of FastSeries hardware and software installation and operation. Call Alacron at 603-891-2750 and ask for the appropriate manuals from the list below if they did not come in your FastSeries shipment.

- 30002-00146 FastImage and FastFrame HW Installation Manual
- 30002-00148 ALFAST Runtime Software Programmer's Guide & Reference
- 30002-00150 FastSeries Library User's Manual
- 30002-00153 Fast I/O Hardware User's Manual
- 30002-00155 FastMem Hardware User's Manual
- 30002-00169 ALRT Runtime Software Programmer's Guide & Reference
- 30002-00170 ALRT, ALFAST & FASTLIB Software Installation Manual for Linux
- 30002-00171 ALRT, ALFAST, & FASTLIB Software Installation for Windows NT
- 30002-00173 FastMem Programmer's Guide & Reference
- 30002-00174 FastMem Hardware Installation Manual
- 30002-00176 FastImage 1300 Hardware User's Manual
- 30002-00180 Fast4 1300 Hardware User's Manual
- 30002-00183 FastImage 1300 Camera Integration User's Manual
- 30002-00184 FastSeries Getting Started Manual
- 30002-00185 FastVision Hardware User's Manual
- 30002-00186 FastVision Software User's Manual

I. OVERVIEW

Fast Object Imaging Library (FOIL) is intended to be used as a universal image-processing library with an extensive set of functions. FOIL provides a simple access to objects and an efficient environment for operations with vectors, matrices and image data. FOIL provides an object-oriented approach to image processing application development.

The library includes a set of main groups of all essential image-processing operations, such as:

- Basic operations (addition, subtraction, multiplication etc.) with objects;
- Convolution functions with different kernels;
- Standard operators like Sobel, Prewitt, Robert, and others;
- Image filtering;
- Image geometrical transforms (rotation, zooming, shifting, etc.);
- Image photometrical transforms (color systems transforms, look-up-table operations, others);
- Spectral analysis (FFT, etc.);
- Others.

The library can be used on the Alacron FastImage1300 Frame Gabber platform and under Microsoft Windows operating system (95/98/ME/NT/2000 or later).

The library can be used for a broad spectrum of applications:

- Video stream processing;
- Image properties adjustment (brightness and contrast control);
- Image enhancement;
- Image preparation;
- Others.

FOIL supports all main types of images:

- Black-and-white types;
- Grayscale types;
- Color types (with different color components encoding and color depth);
    - RGB
    - YUV

The package, which represents FOIL, contains the following information:

A. FOIL

The Information about the library itself:

1. Sources – Library source codes:

a) FOIL.cpp/.hpp – Defines the entry point for the library;

b) FOIL_Decl.cpp/.hpp – Main declarations and abstract classes;

c) FOIL_Vector.cpp/.hpp – Vector template definition and implementation;

d) FOIL_Matrix.cpp/.hpp – Matrix template definition and implementation;

e) FOIL_Image.cpp/.hpp – Image abstraction layer implementation;

f) FOIL_Exceptions.cpp/.hpp – Exceptions definition and implementation;

g) FOIL_RGBImage15.cpp/.hpp – RGB15 image class definition and implementation;

h) FOIL_RGBImage16.cpp/.hpp – RGB16 image class definition and implementation;

i) FOIL_RGBImage32.cpp/.hpp – RGB32 image class definition and implementation;

j) FOIL_RGBPlanar8.cpp/.hpp – RGBPlanar8 image class definition and implementation;

k) FOIL_RGBPlanar16.cpp/.hpp – RGBPlanar16 image class definition and implementation;

l) FOIL_YUB420.cpp/.hpp – YUV420 image class definition and implementation;

m) FOIL_YUV422.cpp/.hpp – YUV422 image class definition and implementation;

B. Binaries

Library binary codes (compiled versions for Windows and TriMedia platforms):

1. Debug

Debug versions (slow, but with data control and log active):

a) TriMedia – Library binary code for TriMedia:

LibFOIL.a;

b) Windows – Library binary code for Microsoft Windows 95/98/ME/NT/2000 or later:

FOIL.dll/.lib;

2. Release

Release version (fast, but no data control or log present):

a) TriMedia – Library binary code for TriMedia:

LibFOIL.a;

b) Windows – Library binary code for Microsoft Windows 95/98/ME/NT/2000 or later:

FOIL.dll/.lib;

3. Makefile

a) Makefile.tm - Building rules for TriMedia platform (debug version);

b) Makefile.tm_fast - Building rules for TriMedia platform (release version);

c) Makefile.msc - Building rules for Windows platform (debug version);

d) Makefile.msc_fast - Building rules for Windows platform (release version);

4. Objs.inc

Common file, which defines dependencies for building rules;

C. Libfastlib

The directory required for correct compilation of FOIL (derived from original Alacron FastSeries library):

1. libfastlib.a

Original FastSeries library binary module (for Philips TriMedia);

2. fastlib.h

Original FastSeries library header;

3. libfastlib.lib

Original FastSeries library binary module for Microsoft Windows 95/98/ME/NT/2000 or later.

4. FOIL (namespace)

The internal contents of the library (templates, classes, exceptions, namespaces,…):

a) Logging control
   - CInitialization (class) – used for log initialization;
   - EnableLogging (function) – used to control logging process;
   - DisableLogging (function) – used to control logging process;
   - GetLoggingStream (function) – used to control logging process;

b) Exception classes (derived from XException):
   - XOutOfBound;
   - XInvalidParameter
   - XInvalidDimensions;
   - XNonConformingViewport;
   - XEmptyObject;
   - XAssert;

c) Common types:
   - dimension_t;
   - byte_t;
   - word_t;
   - dword_t;
   - CComplex;

d) Storage classes:
   - TSimpleValueStorage
   - TComplexValueStorage

e) CDataHandler (class) – used to define new templates, internal use only

f) Image types:
   - ConvertImageFormat (function)
   - CYUV – represents abstract RGB color point, internal use only;

- CRGB – represents abstract YUV color point, internal use only;
- CGrayImage8, CGrayImage16, CGrayImage32, CGrayImageF – gray images;
- CRGBImage15, CRGBImage16, CRGBImage32, CRGBPlanar8, CRGBPlanar16 – RGB color images;
- CYUV420, CYUV422 – YUV color images;

g) Matrix types:
- CUnsignedCharMatrix, CMatrix8 – matrix of chars;
- CUnsignedShortMatrix, CMatrix16 – matrix of shorts;
- CIntMatrix, CMatrix32 – matrix of integers;
- CFloatMatrix, CMatrixF – matrix of floats;
- CComplexMatrix, CMatrixC – matrix of complex float pairs;

h) Vector types:
- CUnsignedCharVector, CVector8 – vector of chars;
- CUnsignedShortVector, CVector16 – vector of shorts;
- CIntVector, CVector32 – vector of integers;
- CFloatVector, CVectorF – vector of floats;
- CComplexVector, CVectorC – vector of complex float pairs.

The features of the library are:

- Vectors, matrices and image processing component support;
- Flexible classes and templates structure can be used to obtain objects with required properties;
- Dynamic allocation and release of the object memory at runtime;
- Viewport support (based on the strides mechanism);
- Exception tree allows the detection and process of any kind of mistakes and errors occurred at runtime;
- Naming convention compatibility with the original FastSeries library;
- Two platforms are supported:
  - Alacron FastImage1300 platform (based on TriMedia 1300);
  - Microsoft Windows operating system (95/98/ME/NT/2000 or later) platform;

Potential users of the library and this manual: managers, high-level application programmers and low-level application developers.

This manual is intended for:

- Managers – to know which advantages the system has, and when and how to use it;
- High-level application programmers – to learn base classes, understand the abstract data layer and function sorting ideas, review the best ways to use this library;
- Low-level application developers – to review the library functionality, ability to enhance it with the new features, to increase performance, improve data formats and calling conventions.

II. ENVIRONMENT REQUIREMENTS

The requirements for the hardware and software are:

- Microsoft Windows 95/98/ME/NT/2000 or later;
- Microsoft Visual C++ 5.0 (or later) compiler;
- Alacron FastImage 1300 Frame Grabber;
- Philips SDE 2.2 (compiler, linker and debugger);
- The FOIL package unpacked to any directory on a local hard drive;
- Path to include files and library files should be set in the makefile of the user's project.

The library and this document assume familiarity with:

- C/C++ programming language knowledge;
    - Classes and objects;
    - Templates;
    - Streams;
    - Exceptions;
    - Friends;
- Microsoft Windows (95/98/ME/NT/2000 or later) operating system;
- Microsoft Visual C++ 5.0 (or later) compiler;
- Familiarity with the SDE.

III. HOW TO USE THE LIBRARY

The step-by-step recommendations for the installation process of the library:

a) Switch off the computer;

b) Remove the cover from the case;

c) Install the Frame Grabber board into the PCI slot (see the corresponding manual);

d) Install the additional DC power cords to the Frame Grabber, if necessary (see the corresponding manual);

e) Install the power cord into the camera (see the corresponding manual);

f) Connect the camera and the Frame Grabber with the input/output data cables (see the corresponding manual);

g) Cover the case;

h) Switch on the power;

i) Boot the PC to Microsoft Windows (95/98/ME/NT/2000 or later) operating system (should already be installed on the computer);

j) Install the Frame Grabber drivers when requested (see the corresponding manual);

k) Install the Philips SDE 2.2 from the provided CDs (see the corresponding manual);

l) Unpack or copy the FOIL package to any directory on your hard drive, for instance, C:\FOIL;

m) Check the presence of all necessary files (refer to the file list described in section 2 “Overview”);

n) Create the simple project in the SDE (see the corresponding manual). For instance, you can use the source code from the example in this manual;

o) Add the following flags to the building rules of “makefile”:

`IC:/FOIL/Sources` *– for compiling process (tmcpp utility in compiling mode)*

`LC:/FOIL/Binaries/Debug/TriMedia -lfoil` *– for linking process (tmcpp utility in linking mode)*

p) Compile the example, using command line “NMAKE makefile”, where the “makefile” is a name of the project file. Note that you should have NMAKE utility already installed on your computer (for example, supplied with Microsoft Visual C++ 5.0 or later);

q) Run the example using command “TMRUN –d0 example”, where 0 is a number of TriMedia CPU on the frame grabber board (0-3), and “example” is the file name of the compiled binary target.

Note that the source code can be compiled with either Microsoft Visual C++ 5.0 compiler (or later) or Philips TriMedia C++ compiler. The binary code can be used (linked and executed) under Microsoft Windows (95/98/ME/NT/2000 or later) or TriMedia platforms.

IV. OBJECTIVE

This manual provides detailed descriptions and specifications for the Object Oriented FastSeries Library of vector, matrix and image functions. This library is based on the FastSeries Library by Alacron, Inc. This manual can be used to:

- Review the library structure;
- Understand data storage mechanisms;
- Learn classes functionality;
- Study application development;
- Go through the examples step-by-step.

The main properties of the Fast Object Imaging Library are:

- Structure of the image processing library is logical and simple;
- It is easy to find necessary methods – they have intuitive names;
- The development time of end-user applications is decreased;
- Methods are divided into groups by their functionality;
- The library routines implementation details and data storing principles are hidden;
- This library is compatible with the original FastSeries library (data formats and naming conventions).

V. RELATED INFORMATION

This document uses links to the following documents:

- FastSeries Library User's Manual, 30002-00150 (Alacron, Inc.)
- Philips TriMedia Documentation from SDE 2.2 (October 2000)

VI. SCOPE AND LIMITATIONS

This section explains all problems and limitations that appeared in the process of developing the structure of the library and the library itself.

A. Compatibility

The compatibility aspect consists of two parts:

- Binary data compatibility;

  All data reserved for vector/image storing should be compatible with the old version of the library. This means that all functions of the old library can be used in the new versions. I.e. the object-oriented library is an object shell for the old version of FastSeries library.

- Naming compatibility;

  Users, who remember the names of the functions of the old version of the library, should easily understand which methods to use in the new version of the library. i.e., the names should be identical. Many functions have a short form of a name (for instance, “meamgv”), which is not self-explanatory. These kinds of functions should be given two versions of names: the short one (same as in the old version of the library) and the new one – long, self-explanatory.

B. Platform-dependent limitations

Some specific limitations of Philips TriMedia platform are listed below:

- Project file should be presented as a standard “Makefile” file that can be interpreted by standard utilities, like “NMAKE” by Microsoft;
- The library should use memory allocation functions from the FastLib library (not from the standard TriMedia SDE libraries). This limitation affects performance;
- The compiled version of the library and examples should not contain debug information to provide fast execution and small executable size;
- Optimization should be enabled when generating the final executable code;
- Source code should not make any assumptions about sizes of basic types (short, int, etc.) because of target platforms differences;
- The library should not output any text to the screen (system console) – even in a case of a fatal error. The library should inform the host program using C++ exceptions (for fatal errors).

VII. CLASSES AND TEMPLATES

This chapter describes the process of the object model development (classes hierarchy and templates) and shows relations in the library. Here you can find the data storage concept, data abstraction layer and viewport data sharing (vector or image).

The Alacron FastSeries library uses:

- 1-D vector functions used in digital signal processing and graphics;
- 2-D matrix/image-processing functions that operate on miscellaneous image data.

Types of elements in vectors and matrices:

a) 8-bit unsigned integer – one byte 0..255 (unsigned char);

b) 16-bit unsigned integer – two bytes 0..65535 (unsigned short);

c) 32-bit floating point - IEEE standard 32 bit single precision floating point numbers;

d) Packed Binary Least-to-most (l2m)

Packed binary l2m are binary images, packing 8 binary pixels into a byte, ordered from least significant bit to most significant bit.

[d0 - pixel n; d1 - pixel n+1; …; d7 - pixel n+7]

e) Packed Binary Most-to-least (m2l)

Packed binary m2l are binary images, packing 8 binary pixels into a byte, ordered from most significant bit to least significant bit.

[d0 - pixel n+7; d1 - pixel n+6; …; d7 - pixel n]

The structure of the library supports all data types listed above and some other functionality:

- 1-D (vector) and 2-D (matrix/image) that represents complex floating point data (a pair of floating point values)
- Ability to define a subset of a vector or an image. This is called a “viewport”. It allows one to define a subvector in a vector and a submatrix in a matrix.

B. CDataHandler

The library operates on vectors and matrices. This is a main classification. The matrix and the vector entities have a lot in common. The entity of a matrix or a vector is a data-storage with some operations defined (for instance, adding one vector/matrix to another).

The most abstract entity is called CDataHandler. This is an abstract base class, which unites common data structures and functionality. Creation of objects of this class is prohibited (its constructor and destructor are located in a private section).

TVector and TMatrix are the templates with a base class parameter (CStorage classes). This means that a programmer can define objects with different functionality using a single template.

The hierarchy of classes is shown in figure 1

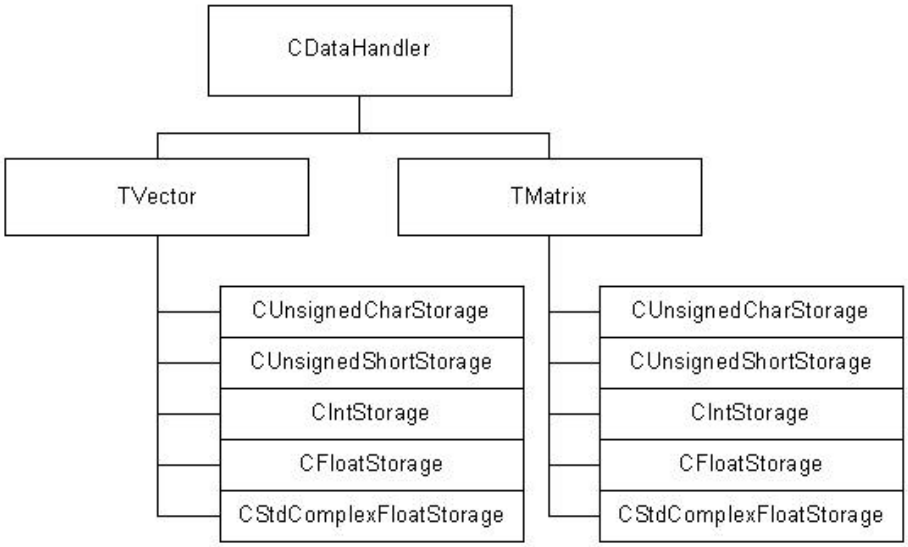


Figure 1 Classes hierarchy

C. Item Type

Both vector and matrix classes should have a parameter specified – the type of the base object. The type of storage and representation of each element of a data array (vector/matrix) is a special class. It defines:

- Format of data storage;
- Element size;
- Base operations (+, -, +=, [] and others).

Each type of the base item has a corresponding class (like CIntStorage, CComplexFloatStorage, CUnsignedIntegerStorage, etc.). These classes define all properties of vector/matrix elements.

Value storage definitions for generic FOIL types:

Long storage name	**Short storage name**
TSimpleValueStorage<unsigned char>	CUnsignedCharStorage
TSimpleValueStorage<unsigned short>	CUnsignedShortStorage
TSimpleValueStorage<int>	CintStorage
TSimpleValueStorage<float>	CfloatStorage
TComplexValueStorage<float, CComplex >	CStdComplexFloatStorage

D. Vector and Matrix Template

TVector and TMatrix are templates with a base class parameter (CStorage classes). This means that a programmer can derive objects with different functionality from a single template.

The data format is binary compatible with the FastSeries Library. The data is located in the dynamic memory area. Figures 2 and 3 show how vector and matrix objects are located in memory and how they refer to the data arrays in the dynamic memory.

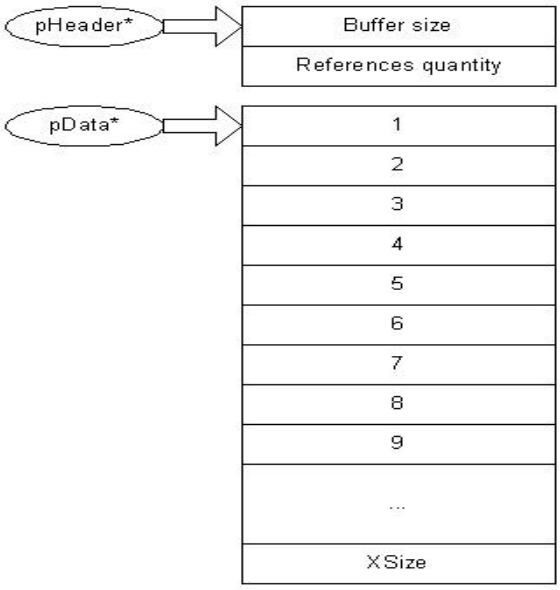


Figure 2 Memory allocation for a vector object

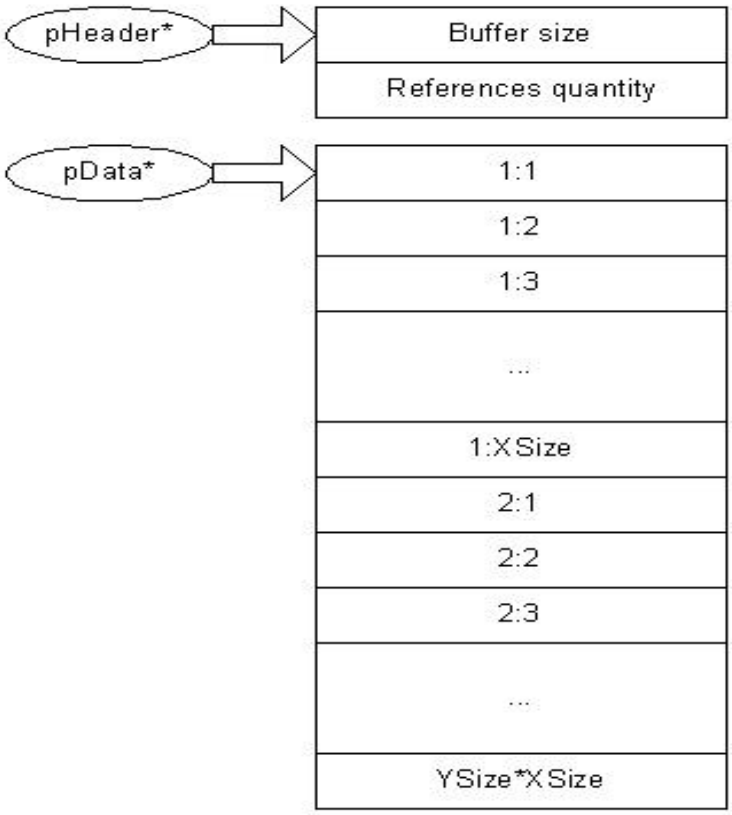


Figure 3 Memory allocation for a matrix object

E. Viewports

A viewport is a mechanism used to define subsets in existing objects. The viewport can define a sub vector (a small vector inside a large vector) in an existing vector or a sub matrix (a small matrix inside a large matrix) in an existing matrix. The viewport is an abstraction layer for data operations. Users will not use strides and pointers anymore (see FastSeries Library User's Manual by Alacron, Inc.).

Figures 4 and 5 show the subsets inside existing objects.

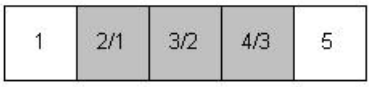


Figure 4 Vector object viewport

1	2	3	4	5
6	7	8	9	10
11	12	13/1	14/2	15
16	17	18/3	19/4	20
21	22	23	24	25

Figure 5 Matrix object viewport

The current implementation of viewports does not allow defining a subvector inside a matrix or submatrix inside a vector. It is possible to define a subset inside a subset (for instance, subvector inside subvector of vector).

Current object model allows one to define viewports quickly and easily. The idea is to store an object with the viewport description and vector/matrix data separately. Objects refer to allocated data and define subsets inside the data array. In fact, every vector or matrix is a viewport to data arrays.

Matrix members contain information about left top corner and width/height parameters, defining viewport to the referenced object (vector members contain only left position and number of elements stored). Objects that define new data (not a reference/viewport to existing data) allocate memory area for storing image/vector data and define a viewport, which is equal to the size of the original image/vector. This is done by the class constructor.

When creating a viewport to the existing image/vector, the constructor accepts reference to existing object and viewport parameters (left top corner and width-height for a matrix; start position and length for vector).

VIII. METHODS LIST

All templates, classes, types, data structures are declared in the “FOIL” namespace (Fast Object Imaging Library).

FOIL library defines the following exception classes:

- XException – common abstract exception used for inheriting new exceptions;
- XOutOfBound – The element index is out of bounds;
- XInvalidParameter – Bad call parameter;
- XInvalidDimensions (derived from XInvalidParameter) – Mismatch of parameters dimensions;
- XNonConformingViewport (derived from XInvalidDimensions) – Nonconforming viewport. It can not be processed;
- XAssert – Assertion in the debug version of the library;
- XEmptyObject – Object is empty (no data present). Access to an empty object is forbidden.

The what () method returns a string with the explanation of the exception cause.

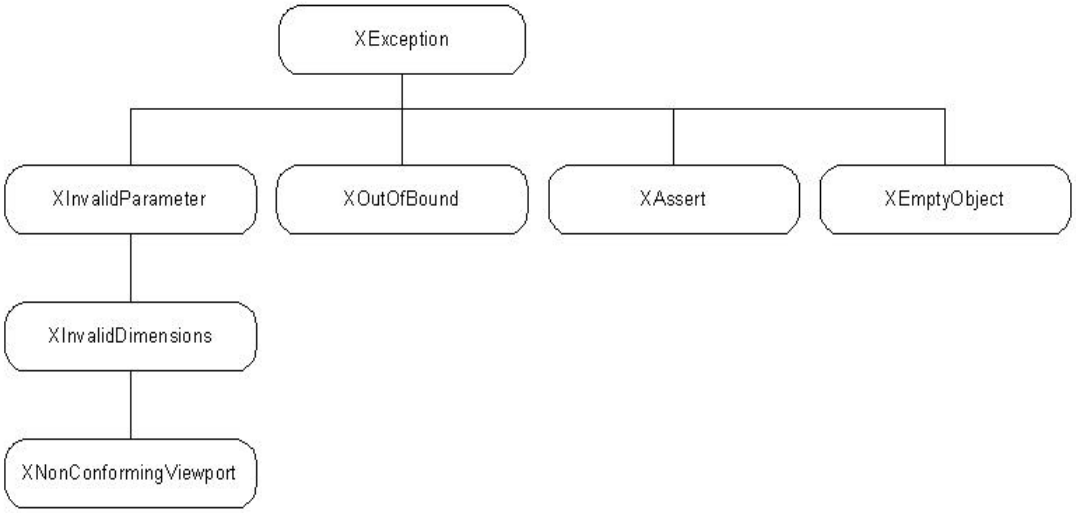


Figure 6 The scheme of used exceptions

The library contains the following types to be used in members and methods:

- dimension_t (unsigned short) – type, representing size of a vector or a matrix (in one dimension);
- color_t – type, representing the color element inside each image class (for instance, CRGB)
- Basic library types definition (types representing the type of data block element (used for compatibility with the FastSeries library)):
  - byte_t (unsigned char) – Unsigned 8-bit integer type;
  - word_t (unsigned short) – Unsigned 16-bit integer type;
  - dword_t (unsigned int) – Unsigned 32-bit integer type;
- CComplex (std::complex) – Complex float type for complex vectors and matrices.

You can define the “NDEBUG” compile-time symbol to control the debug functionality of the library. The definition of this switch means “do not debug”. The library compiled without this switch defined is a safe version for debugging purposes. All exception classes have string explanation. Parameters are checked before processing data. This reduces the performance of the library. If the switch is changed, a recompilation of the library is required.

The debugging version can be configured using the “NO_LOG” symbol. It switches off logging of all source parameters, processing information, and output results. If it is not defined, the logging is switched on.

The following functions affect the logging process. Logging is switched on by default:

Turns FOIL logging on. Returns false if failed.
bool FOIL_API EnableLogging();
Turns FOIL logging off.
void FOIL_API DisableLogging();

Some library methods may be called with the output object being the same as one of the input objects. Not all methods may be called in this manner. Methods that do not allow common input and output objects are:

- Conv3x3, Conv5x5, Conv7x7;
- Kirsch;
- Prewitt;
- Roberts;
- Sobel;
- Median, Median3x3, Median5x5, Median7x7;
- Rotate, Rotate8, Rotate16;
- Xgradient, Ygradient.

The following topics contain a full list of methods sorted by classes and templates, with calling convention, parameters and return values descriptions.

A. TVector

This chapter describes methods for classes produced using the TVector template.

The syntax of object creation using templates is lengthy and complicated. FOIL library contains special substitutes, which can be used in user programs:

Long name	**Aliases for common use**	**Platform independed types aliases**
Tvector<int, CUnsignedCharStorage>	CunsignedCharVector	CVector8
TVector<unsigned short, CUnsignedShortStorage>	CunsignedShortVector	CVector16
Tvector<int, CIntStorage>	CintVector	CVector32
Tvector<float, CFloatStorage>	CfloatVector	CVectorF
TVector<CComplex, CStdComplexFloatStorage>	CcomplexVector	CVectorC

1. Constructors and destructors

Creating methods for classes produced from the TVector template

Summary:	**Default constructor**
Syntax:	TVector ();
Arguments:	None
Description:	Creates a new vector object without attached data block. Object can’t be used until defining vector size and allocating appropriate data block.
Example:	FOIL::CIntVector MyVector;

Summary:	**Allocating constructor**
Syntax:	TVector (dimension_t number);
Arguments:	number - Vector dimension
Description:	Creates new vector object and attaches new allocated data block.
Example:	FOIL::CIntVector MyVector (10);

Summary:	**Filling constructor**
Syntax:	TVector (dimension_t number, Value fill_value);
Arguments:	number - Vector dimension fill_value - Source value for initial vector contents
Description:	Creates new vector object, attaches new allocated data block and fills it by given values.
Example:	FOIL::CIntVector MyVector (10, 0);

Summary:	**Copying constructor**
Syntax:	TVector(TVector& vector);
Arguments:	vector – Source vector
Description:	Creates new vector object, attaches new allocated memory block and copy source vector data to the new one.
Example:	FOIL::CIntVector MyVector1 (10, 0); FOIL::CIntVector MyVector2 (MyVector1);

Summary:	**Vector viewport definition constructor (from vector)**
Syntax:	TVector (TVector& vector, dimension_t offset, dimension_t stride, dimension_t number);
Arguments:	vector - Source vector offset – Viewport vector items will start from this offset stride – Specifies the stride between nearby elements of the viewport (should not equal 0) number - Number of items in viewport vector
Description:	Creates a special window named “viewport” to existing vector. Doesn’t allocate data blocks – new object links to data block of existing vector.
Example:	FOIL::CIntVector MyVector1 (10, 0); FOIL::CIntVector MyVector2 (MyVector1, 1, 1, 3);

Summary:	**Vector viewport definition constructor (from matrix)**
Syntax:	TVector (const TMatrix<Value, Storage>& matrix, dimension_t first_row = 0, dimension_t first_column = 0, dimension_t number = 0);
Arguments:	matrix – Source matrix first_row – The first row number of a viewport submatrix in the matrix first_column – The first column number of a viewport submatrix in the matrix number – The number of items in the vector viewport
Description:	Creates a special window named “viewport” to existing matrix. Doesn’t allocate data blocks – new object links to data block of existing matrix. If the matrix is a viewport, than the vector viewport should not be out of bounds of source data area mapped by the matrix viewport. In other words, the vector viewport should point to uninterrupted items sequence in the source matrix array. If the ‘number’ is zero then it will be treated as a most acceptable size of vector.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); FOIL::CIntVector MyVector (MyMatrix, 3, 3, 5);

Default destructor is used for releasing data memory used by the vector object. Object consists of the object data and the attached memory area. The attached memory area is released only when the destructor of the last referring object is called.

2. Operators

Summary:	Assignment operator
Syntax:	TVector& operator = (const TVector& b);

Arguments:	b – Source vector
Description:	Copies data block of source vector to the target one. Note: copying operations means destroying old data block of the destination vector (if one exists), creating the new data block and copying data from the data block of source vector.
Example:	FOIL::CIntVector MyVector1 (10, 0); FOIL::CIntVector MyVector2; MyVector2 = MyVector1;

Summary:	Array-access-like operator
Syntax:	Type& operator[](dimension_t index);
Arguments:	index – Source/destination element index
Description:	Can be used to access vector elements using array like operator [i]. This syntax form is simpler than the one that the library source code contains. “Type” is a type of vector items.
Example:	FOIL::CIntVector MyVector (10, 0); MyVector1[5] = 123;

3. Functions description

Summary:	**Gets item value**
Syntax:	Type GetItem(dimension_t index)
Arguments:	index – The index of vector item
Description:	Returns the value of vector item by the index. “Type” is a type of vector item.
Example:	FOIL::CIntVector MyVector (10, 0); int iItem = MyVector.GetItem (3);

Summary:	**Sets item value**
Syntax:	void SetItem(dimension_t index, const Type& pValue);
Arguments:	index – The index of vector item pValue – Value to store in the item
Description:	Sets the item in vector by the given index. “Type” is a type of vector item.
Example:	FOIL::CIntVector MyVector (10, 0); MyVector.SetItem (3, 123);

Summary:	**Gets number of elements in vector**
Syntax:	dimension_t GetNumber(void);
Arguments:	None
Description:	Gets elements quantity in vector. (0 if object has no attached data)
Example:	FOIL::CIntVector MyVector (10, 0); dimension_t iDimension = MyVector.GetNumber ();

Summary:	**Detects attached data presence**
Syntax:	bool IsEmpty (void);
Arguments:	None
Description:	Returns TRUE if object has no attached data, FALSE otherwise.
Example:	FOIL::CIntVector MyVector (10, 0); bool oEmpty = MyVector.IsEmpty ();

Summary:	**Gets viewport stride**
Syntax:	dimension_t GetStride(void);
Arguments:	None
Description:	Gets stride of the viewport (returns 1 for ordinary vectors)
Example:	FOIL::CIntVector MyVector1 (10, 0); FOIL::CIntVector MyVector2 (MyVector1, 1, 2, 3); dimension_t iStride = MyVector2.GetStride ();

Summary:	**Simple initialization**
Syntax:	void Initialize(dimension_t number);
Arguments:	number – Vector dimension
Description:	Initializes vector with the new data block – creates and attaches it to vector object. Old

	data block will be removed (if exists). New data block is uninitialized.
Example:	FOIL::CIntVector MyVector (10, 0); MyVector.Initialize (5);

Summary:	**Filling initialization**
Syntax:	void Initialize (dimension_t number, Value fill_value);
Arguments:	number – Vector dimension fill_value – Source value for initial matrix contents
Description:	Initializes vector with the new data block – creates and attaches it to vector object. Old data block will be removed (if exists). New data block is filled by value given.
Example:	FOIL::CIntVector MyVector (10, 0); MyVector.Initialize (5, 123);

Summary:	**Viewport initialization (from vector)**
Syntax:	void Initialize (TVector& vector, dimension_t offset, dimension_t stride, dimension_t number);
Arguments:	vector – Source vector offset – Viewport vector items will start from this offset stride – Element stride specified that any 'stride' element of a source vector is to be viewed by viewport number – Number of items in viewport vector
Description:	Attaches vector object to data block of existing vector. Old data block will be removed (if exists).
Example:	FOIL::CIntVector MyVector1 (10, 0); FOIL::CIntVector MyVector2; MyVector2.Initialize (MyVector1, 3, 1, 5);

Summary:	**Viewport initialization (from matrix)**
Syntax:	void Initialize (const TMatrix<Value, Storage>& matrix, dimension_t first_row = 0, dimension_t first_column = 0, dimension_t number = 0);
Arguments:	matrix – Source matrix first_row – The first row number of a viewport submatrix in the matrix first_column – The first column number of a viewport submatrix in the matrix number – The number of items in the vector viewport
Description:	Attaches vector object to data block of existing matrix. Old data block will be removed (if exists). If the matrix is a viewport, than the vector viewport should not be out of bounds of source data area mapped by the matrix viewport. In other words, the vector viewport should point to uninterrupted items sequence in the source matrix array. If the 'number' is zero then it will be treated as a most acceptable size of vector.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); FOIL::CIntVector MyVector; MyVector.Initialize (MyMatrix, 3, 3, 5);

Summary:	**Make object empty**
Syntax:	void Reset();
Arguments:	None
Description:	Detaches data block from the vector object.
Example:	FOIL::CIntVector MyVector1 (10, 0); MyVector1.Reset ();

a) Char

Methods operating on vectors with 8-bit integer elements.

Summary:	**Vector fix to 8 bit integer**
Syntax:	void vfix8 (const CFloatVector& a, CUnsignedCharVector* c, bool rounding);
Arguments:	a – Source float vector c – Destination unsigned char vector rounding – Rounding/truncating flag
Description:	Changes the elements of a floating point vector a from type float to type unsigned char (1 byte) and stores the results to the integer vector c. Rounding specifies whether to round or truncate.

	for i=0 to n-1
	{
	if rounding=false, c[i] = short((a[i]+sign(a[i])*0.5))
	else, c[i] = short(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 5.5); FOIL::CUnsignedCharVector MyVector2 (10); FOIL::CUnsignedCharVector MyVector3 (10); FOIL::vfix8 (MyVector1, &MyVector2, false); FOIL::vfix8 (MyVector1, &MyVector3, true);

b) Short

Methods operating on vectors with 16-bit integer elements.

Summary:	**Converts the elements of floating point vector to 8-bit packed integers in a high/low format**
Syntax:	void fixinta (const CFloatVector& a, CUnsignedShortVector* c);
Arguments:	a – Source float vector c – Destination unsigned short vector

Description:	Each element is fixed and the low-order 8 bits of the integer are taken as a positive magnitude. They are then stored as pairs into short integers in unsigned short vector c in least-significant then most-significant order:
	for i=0 to n-1 in steps of 2
	{
	c[i/2] = fix(a[i]) AND $FF OR 256*fix(a[i+1]) AND $FF
	}

Example:	FOIL::CFloatVector MyVector1 (10, 123.456); FOIL::CUnsignedShortVector MyVector2 (10); FOIL::fixinta (MyVector1, &MyVector2);

Summary:	**Converts the elements of a floating point vector to 8-bit packed integers in a low/high format.**
Syntax:	void fixintb (const CFloatVector& a, CUnsignedShortVector* c);
Arguments:	a – Source float vector c – Destination unsigned short vector

Description:	Each element is fixed and the low-order 8 bits of the integer are taken as a positive magnitude. They are then stored as pairs into short integers in short vector c in most-significant then least-significant order:
	for i=0 to n-1 in steps of 2
	{
	c[i/2] = 256*fix(a[i]) AND $FF OR fix(a[i+1]) AND $FF
	}

Example:	FOIL::CFloatVector MyVector1 (10, 123.456); FOIL::CUnsignedShortVector MyVector2 (10); FOIL::fixintb (MyVector1, &MyVector2);

Summary:	**Vector fix to short integer**
Syntax:	void vfix16 (const CFloatVector& a, CUnsignedShortVector* c, bool rounding);
Arguments:	a – Source float vector c – Destination unsigned short vector rounding – Rounding/truncating flag

Description:	Changes the elements of float vector a from type float to type unsigned short (2 bytes) and stores the results in vector c. Rounding specifies whether to round or truncate:
	for i=0 to n-1
	{
	if rounding=false, c[i] = short((a[i]+sign(a[i])*0.5))
	else, c[i] = short(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 5.5); FOIL::CUnsignedShortVector MyVector2 (10); FOIL::CUnsignedShortVector MyVector3 (10);

	FOIL::vfix16 (MyVector1, &MyVector2, false); FOIL::vfix16 (MyVector1, &MyVector3, true);

c) Int

Methods operating on vectors with 32-bit integer elements.

Summary:	**Integer logical bitwise AND**
Syntax:	void vand(const CIntVector& a, const CIntVector& b, CIntVector* c);
Arguments:	a – Source integer vector b – Source integer vector d – Destination integer vector

Description:	Forms the bitwise logical AND of the corresponding 32-bit integer elements of integer vectors a and b and stores the results into float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] AND b[i]
	}

Example:	FOIL::CIntVector MyVector1 (10, 123); FOIL::CIntVector MyVector2 (10, 456); FOIL::CIntVector MyVector3 (10); FOIL::vand (MyVector1, MyVector2, &MyVector3);

Summary:	**Integer logical bitwise AND**
Syntax:	CIntVector operator & (const CIntVector& a, const CIntVector& b);
Arguments:	a – Source integer vector b – Source integer vector

Description:	Forms the bitwise logical AND of the corresponding 32-bit integer elements of integer vectors a and b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] AND b[i]
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CIntVector MyVector1 (10, 123); FOIL::CIntVector MyVector2 (10, 456); FOIL::CIntVector MyVector3 (10); MyVector3 = MyVector1 & MyVector2;

Summary:	**Integer logical bitwise AND**
Syntax:	CIntVector& operator &= (CIntVector& a, const CIntVector& b);
Arguments:	a – Source integer vector b – Source integer vector

Description:	Forms the bitwise logical AND of the corresponding 32-bit integer elements of integer vectors a and b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] AND b[i]
	}

Example:	FOIL::CIntVector MyVector1 (10, 123); FOIL::CIntVector MyVector2 (10, 456); MyVector2 &= MyVector1;

Summary:	**Vector fix to integer**
Syntax:	void vfix32 (const CFloatVector& a, CIntVector* c, bool rounding);
Arguments:	a – Source float vector c – Destination integer vector rounding – Rounding/truncating flag
Description:	Changes the elements of float vector a to type int (4 bytes) and stores the results in integer vector c. rounding specifies whether to round or truncate

	for i=0 to n-1
	{
	if rounding=0, c[i] = long((a[i]+sign(a[i])*0.5))
	else, c[i] = long(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 5.5); FOIL::CIntVector MyVector2 (10); FOIL::CIntVector MyVector3 (10); FOIL::vfix32 (MyVector1, &MyVector2, false); FOIL::vfix32 (MyVector1, &MyVector3, true);

Summary:	**Vector scale and fix**
Syntax:	void vscal (const CFloatVector& a, float b, CIntVector* c, int nBits);
Arguments:	a – Source float vector b – Source float scalar c – Destination integer vector nBits – Source integer scalar

Description:	Scales the elements of float vector a by a power of 2, fixes the scaled values truncating towards 0.0, and stores the results into integer vector c. The power of 2 is chosen so that float scalar b falls within one-quarter to one-half the dynamic range specified by integer bit width nbits:
	k = nBits - int (log2(b))
	for i=0 to n-1
	{
	c[i] = fix (a[i]*(pow(2,k)))
	}

Example:	FOIL::CFloatVector MyVector1 (10, 5.5); FOIL::CIntVector MyVector2 (10); FOIL::vscal (MyVector1, 8.0, &MyVector2, 5);

Summary:	**Integer logical bitwise OR**
Syntax:	void vor(const CIntVector& a, const CIntVector& b, CIntVector* c);
Arguments:	a – Source integer vector a b – Source integer vector b c – Destination integer vector c

Description:	Forms the bitwise logical OR of the corresponding 32-bit integer elements of integer vectors a and b and stores the results into integer vector c:
	for i=0 to n-1
	{
	c[i] = a[i] OR b[i]
	}

Example:	FOIL::CIntVector MyVector1 (10, 0xE5); FOIL::CIntVector MyVector2 (10, 0x42); FOIL::CIntVector MyVector3 (10); FOIL::vor (MyVector1, MyVector2, &MyVector3);

Summary:	**Integer logical bitwise OR**
Syntax:	CIntVector operator | (const CIntVector& a, const CIntVector& b);
Arguments:	a – Source integer vector a b – Source integer vector b

Description:	Forms the bitwise logical OR of the corresponding 32-bit integer elements of integer vectors a and b and returns the results:
	for i=0 to n-1
	{
	result[i] = a[i] OR b[i]
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CIntVector MyVector1 (10, 0xE5); FOIL::CIntVector MyVector2 (10, 0x42); FOIL::CIntVector MyVector3 (10);

	MyVector3 = MyVector1 | MyVector2;

Summary:	**Integer logical bitwise OR**
Syntax:	CIntVector& operator |= (CIntVector& a, const CIntVector& b);
Arguments:	a – Source integer vector a b – Source integer vector b

Description:	Forms the bitwise logical OR of the corresponding 32-bit integer elements of integer vectors a and b and returns the results:
	for i=0 to n-1
	{
	result[i] = a[i] OR b[i]
	}

Example:	FOIL::CIntVector MyVector1 (10, 0xE5); FOIL::CIntVector MyVector2 (10, 0x42); MyVector2 |= MyVector1;

d) Float

Methods operating on vectors with float elements.

Summary:	**Blackman window multiply**
Syntax:	void blkman (const CFloatVector& a, CFloatVector* c);
Arguments:	a – vector of float values c – results vector c of float values

Description:	Multiples float vector a by a Blackman window to condition it for performing an FFT according to the algorithm:
	for i=0 to n-1
	{
	c[i] = a[i] * (0.42 - 0.5*cos(i*2*pi/n) + 0.08*cos(i*4*pi/n))
	}
	For other functions used for conditioning signals prior to performing an FFT, see the functions hamm and hann.

Example:	FOIL::CFloatVector MyVector1 (10, 12.3); FOIL::CFloatVector MyVector2 (10); FOIL::blkman (MyVector1, &MyVector2);

Summary:	**Dot product**
Syntax:	float dotpr (const CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source vector a b – Source vector b

Description:	Computes the dot product of two float vectors a and b and return a float scalar as result using algorithm:
	ainc = 0
	binc = 0
	for i=0 to n-1
	{
	c = sum (a[ainc] * b[binc])
	ainc = ainc + ai
	binc = binc + bi
	}

Example:	FOIL::CFloatVector MyVector1 (10, 12.3); FOIL::CFloatVector MyVector2 (10, 45.6); float fResult = FOIL::dotpr (MyVector1, MyVector2);

Summary:	**Converts the elements of packed unsigned short (low/high) vector to float format**
Syntax:	void fltinta (const CUnsignedShortVector& a, CFloatVector* c);
Arguments:	a – Source packed unsigned short vector c – Destination float vector
Description:	Converts the elements of pixel vector a from 8-bit packed integers in a high/low format to float values and stores them into vector c. Each 8-bit element in vector a is taken as a positive magnitude and floated. They are extracted in least-significant then most-

	significant order:
	for i=0 to n-1 in steps of 2
	{
	c[i] = float(a[i/2]).AND.$FF)
	c[i+1] = float(a[i/2]).AND.$FF00)/256
	}

Example:	FOIL::CUnsignedShortVector MyVector1 (10, 0xA3); FOIL::CFloatVector MyVector2 (10); FOIL::fltinta (MyVector1, &MyVector2);

Summary:	**Converts the elements of packed unsigned short (high/low) vector to float format**
Syntax:	void fltintb (const CUnsignedShortVector& a, CFloatVector* c);
Arguments:	a – Source packed unsigned short vector c – Destination float vector

Description:	Converts the elements of pixel vector a from 8-bit packed integers in a low/high format to float values and stores them into vector c. Each 8-bit element in vector a is taken as a positive magnitude and floated. They are extracted in most-significant then least-significant order:
	for i=0 to n-1 in steps of 2
	{
	c[i] = float(a[i/2]).AND.$FF00)/256
	c[i+1] = float(a[i/2]).AND.$FF)
	}

Example:	FOIL::CUnsignedShortVector MyVector1 (10, 0xA3); FOIL::CFloatVector MyVector2 (10); FOIL::fltintb (MyVector1, &MyVector2);

Summary:	**Hamming window multiply**
Syntax:	void hamm (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector a c – Destination float vector c

Description:	Multiplies float vector a by a Hamming window and stores the resulting float vector into c. The Hamming algorithm is:
	for i-0 to n-1
	{
	c[i] = a[i] * (0.54 - 0.46*cos(i*2*pi/n))
	}
	blkman and hann are two other functions for conditioning signals prior to performing an FFT.

Example:	FOIL::CFloatVector MyVector1 (10, 0xA3); FOIL::CFloatVector MyVector2 (10); FOIL::hamm (MyVector1, &MyVector2);

Summary:	**Hanning window multiply**
Syntax:	void hann (const CFloatVector& a, CFloatVector* c, bool flag);
Arguments:	a – Source float vector a c – Destination float vector c flag – Operation type flag

Description:	For flag = false, hann multiplies float vector a by an unnormalized Hanning window using the algorithm:
	for i=0 to n-1
	{
	c[i] = 0.5 * a[i] * (1.0 - cos(i*2*pi/n))
	}
	For flag = true, hann multiplies float vector a by a normalized Hanning window by the algorithm:
	for i=0 to n-1
	{
	c[i] = sqrt(2.0/3.0) * a[i] * (1.0 -cos(i*2*pi/n))
	}
	blkman and hamm are two other functions used for conditioning signals prior to

	performing an FFT.
Example:	FOIL::CFloatVector MyVector1 (10, 0xA3); FOIL::CFloatVector MyVector2 (10); FOIL::CFloatVector MyVector3 (10); FOIL::hann (MyVector1, &MyVector2,false); FOIL::hann (MyVector1, &MyVector3,true);

Summary:	**Histogram of float vector**
Syntax:	void hist (const CFloatVector& a, CFloatVector* c, float amax, float amin);
Arguments:	a – Source float vector a c – Destination float vector c amax – maximum value of histogram amin – minimum value of histogram

Description:	Constructs a histogram for the elements of float vector a and adds the results to the histogram in vector c. The number of bins in the histogram and the maximum and minimum values of the range of interest are specified by dimension of histogram vector, amax, and amin, respectively. The width of each bin is (amax - amin)/(bins number) except that values below amin or above amax are counted in the first and the last bin, respectively. Note that upon entry to hist, vector c contains an initial histogram. Upon return, vector c has been updated with the number of elements of vector a that fell in each bin according to the algorithm:
	for i=0 to n-1
	{
	if a[i] < amin, j = 0,
	else if a[i] >= amax, j = nc-1,
	else, j = int(nc*(a[I]-amin)/(amax-amin)),
	then c[j] = c[j] + 1.0
	}
	Note: If the destination vector c is a viewport, its stride should be equal to 1.

Example:	FOIL::CFloatVector MyVector1 (6); MyVector1[0] = 1.2; MyVector1[1] = 2.3; MyVector1[2] = 3.4; MyVector1[3] = 4.5; MyVector1[4] = 5.6; MyVector1[5] = 6.7; FOIL::CFloatVector MyVector2 (3, 0.0); FOIL::hist (MyVector1, &MyVector2, 2.0, 6.0);

Summary:	**Logical vector equal**
Syntax:	void lveq(const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Compares the corresponding elements of float vectors a and b and sets the corresponding elements of vector c to 1.0 if the element of a equals the element of b, to 0 if not:
	for i=0 to n-1
	{
	if a[i] == b[i], c[i] = 1.0, else c[i] = 0.0
	}

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); FOIL::CFloatVector MyVector2 (6, 12,34); MyVector1[2] = 999.999; MyVector2[4] = 888.888; FOIL::CFloatVector MyVector3 (6); FOIL::lveq (MyVector1, MyVector2, &MyVector3);

Summary:	**Logical vector greater than or equal**
Syntax:	void lvge(const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a

	b – Source float vector b c – Destination float vector c

Description:	Compares the corresponding elements of vectors a and b. If the element of a is greater than or equal to the element of b, the corresponding element of float vector c is set to 1.0. Otherwise, the element of c is set to 0.0:
	for i=0 to n-1
	{
	if a[i] >= b[i], c[i] = 1.0, else c[i] = 0.0
	}

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); FOIL::CFloatVector MyVector2 (6, 12,34); MyVector1[2] = 888.888; MyVector2[4] = 999.999; FOIL::CFloatVector MyVector3 (6); FOIL::lvge (MyVector1, MyVector2, &MyVector3);

Summary:	**Logical vector greater than**
Syntax:	void lvgt(const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	lvge compares the corresponding elements of float vectors a and b, and if the element of a is greater than the element of b, the corresponding element of float vector c is set to 1.0. Otherwise, the element of c is set to 0.0:
	for i=0 to n-1
	{
	if a[i] > b[i], c[i] = 1.0, else c[i] = 0.0
	}

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); FOIL::CFloatVector MyVector2 (6, 12,34); MyVector1[2] = 999.999; MyVector2[4] = 888.888; FOIL::CFloatVector MyVector3 (6); FOIL::lvgt (MyVector1, MyVector2, &MyVector3);

Summary:	**Logical vector not equal**
Syntax:	void lvne(const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Compares the corresponding elements of float vectors a and b, and if the element of a is not equal to the element of b, the corresponding element of float vector c is set to 1.0. Otherwise, the element of c is set to 0.0:
	for i=0 to n-1
	{
	if a[i] != b[i], c[i] = 1.0, else c[i] = 0.0
	}

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); FOIL::CFloatVector MyVector2 (6, 12,34); MyVector1[2] = 999.999; MyVector2[4] = 888.888; FOIL::CFloatVector MyVector3 (6); FOIL::lvne (MyVector1, MyVector2, &MyVector3);

Summary:	**Logical vector not**
Syntax:	void lvnot (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector a c – Destination float vector c
Description:	Computes the logical complement of float vector a and stores the results in float vector c according to the algorithm: for i=0 to n-1

	{
	if a[i] = 0.0, c[i] = 1.0, else c[i] = 0.0
	}

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); MyVector1[2] = 0.0; FOIL::CFloatVector MyVector2 (6); FOIL::lvnot (MyVector1, &MyVector2);

Summary:	**Maximum magnitude of a vector**
Syntax:	void maxmgv (const CFloatVector& a, float* c, dimension_t *lc);
Arguments:	a – Source float vector c – Destination float scalar containing maximum magnitude found lc – Destination integer scalar containing index of first element with maximum magnitude
Description:	Finds the first element in float vector a having the maximum magnitude (absolute value), stores its magnitude in c, and stores index of first element with maximum magnitude in lc.
Example:	FOIL::CFloatVector MyVector1 (6, 12.34); MyVector1[2] = -999.999; float fMaxMagnitude; dimension_t Index; FOIL::maxmgv (MyVector1, &fMaxMagnitude, &Index);

Summary:	**Maximum element of a vector**
Syntax:	void maxv (const CFloatVector& a, float *c, dimension_t *lc);
Arguments:	a – Source float vector c – Destination float scalar containing maximum element found lc – Destination integer scalar containing index of first element with maximum value
Description:	Finds the element in float vector a having the maximum value, stores that value in float scalar c, and stores index of first element with maximum value in lc.
Example:	FOIL::CFloatVector MyVector1 (6, 12.34); MyVector1[2] = 999.999; float fMaxElement; dimension_t Index; FOIL::maxv (MyVector1, &fMaxElement, &Index);

Summary:	**Mean of vector element magnitudes**
Syntax:	void meamgv (const CFloatVector& a, float *c);
Arguments:	a – Source float vector c – Destination float scalar containing mean magnitude of all elements

Description:	Computes the mean magnitude of all elements in float vector a and stores the result in float scalar c. The algorithm for the mean is:
	for i=0 to n-1
	{
	c = sum(abs(a[i])) / n
	}

Example:	FOIL::CFloatVector MyVector1 (4); MyVector1[0] = 1.2; MyVector1[1] = -2.3; MyVector1[2] = 3.4; MyVector1[3] = -4.5; float fMeanMagnitude; FOIL::meamgv (MyVector1, &fMeanMagnitude);

Summary:	**Mean value of vector elements**
Syntax:	void meanv (const CFloatVector& a, float *c);
Arguments:	a – Source float vector c – Destination float scalar containing mean value of all elements
Description:	Computes the mean value of all elements in float vector a and stores the result in float scalar c. The algorithm for the mean is: for i=0 to n-1

	{
	c = sum(a(i)) / N
	}

Example:	FOIL::CFloatVector MyVector1 (4); MyVector1[0] = 1.2; MyVector1[1] = -2.3; MyVector1[2] = 3.4; MyVector1[3] = -4.5; float fMeanValue; FOIL::meanv (MyVector1, &fMeanValue);

Summary:	**Mean of vector element squares**
Syntax:	void measqv (const CFloatVector& a, float *c);
Arguments:	a – Source float vector c – Destination float scalar containing mean value of the element squares

Description:	Computes the mean value of the squares of the elements of float vector a and stores the result into float scalar c. The algorithm for the computation is:
	for i=0 to n-1
	{
	c = sum(a[i] * a[i]) / n
	}

Example:	FOIL::CFloatVector MyVector1 (4); MyVector1[0] = 1.2; MyVector1[1] = -2.3; MyVector1[2] = 3.4; MyVector1[3] = -4.5; float fMeanSquares; FOIL::measqv (MyVector1, &fMeanSquares);

Summary:	**Minimum magnitude element of a vector**
Syntax:	void minmgv (const CFloatVector& a, float *c, dimension_t *lc);
Arguments:	a – Source float vector c – Destination float scalar containing minimum magnitude found lc – Destination integer scalar containing index of first element with minimum magnitude
Description:	Finds the first element having the minimum magnitude (absolute value) in float vector a, stores its magnitude in float scalar c, and stores index of first element containing minimum magnitude in lc.
Example:	FOIL::CFloatVector MyVector1 (6, 12.34); MyVector1[2] = 0.1; float fMinMagnitude; dimension_t Index; FOIL::minmgv (MyVector1, &fMinMagnitude, &Index);

Summary:	**Minimum element of a vector**
Syntax:	void minv (const CFloatVector& a, float *c, dimension_t *lc);
Arguments:	a – Source float vector c – Destination float scalar containing minimum value found lc – Destination integer scalar containing index of first element with minimum value
Description:	Finds the element in floatvector a having the minimum value, stores that value in float scalar c, and stores index of first element containing minimum value in lc.
Example:	FOIL::CFloatVector MyVector1 (6, 12.34); MyVector1[2] = 0.1; float fMinValue; dimension_t Index; FOIL::minv (MyVector1, &fMinValue, &Index);

Summary:	**Root mean square of vector elements**
Syntax:	void rmsqv (const CFloatVector& a, float* c);
Arguments:	a – Source float vector

	c – Destination float scalar containing mean value of the elements squares

Description:	Computes the square root of the mean value of the squares of the elements of float vector a and stores the result in float scalar c using the algorithm:
	for i=0 to n-1
	{
	c = sqrt(sum(a[i]*a[i]) / n)
	}

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); float fMeanSquare; FOIL::rmsqv (MyVector1, &fMeanSquare);

Summary:	**Scalar vector divide**
Syntax:	void svdiv (float a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float scalar b – Source float vector c – Destination float vector

Description:	Divides scalar a by float vector b putting the results in float vector c:
	for i=0 to n-1
	{
	c[i] = a / b[i]
	}
	Vector b should not contain zero elements.

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); FOIL::CFloatVector MyVector2 (6); float fArgument = 3.4; FOIL::svdiv (MyVector1, fArgument, &MyVector2);

Summary:	**Scalar vector divide**
Syntax:	CFloatVector operator / (float a, const CFloatVector& b);
Arguments:	a – Source float scalar b – Source float vector

Description:	Divides scalar a by float vector b and returns the results:
	for i=0 to n-1
	{
	result[i] = a / b[i]
	}
	Vector b should not contain zero elements.
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (6, 12.34); FOIL::CFloatVector MyVector2 (6); float fArgument = 3.4; MyVector2 = fArgument / MyVector1;

Summary:	**Sum of vector elements**
Syntax:	float sve (const CFloatVector& a);
Arguments:	a – Source float vector

Description:	Computes and returns the sum of the elements of float vector a computed by the algorithm:
	for i=0 to n-1
	{
	result = sum(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (3); MyVector1[0] = 1.2; MyVector1[1] = -2.3; MyVector1[2] = 3.4; float fSum; fSum = FOIL::sve (MyVector1);

Summary:	**Sum of vector element magnitudes**

Syntax:	float svemg (const CFloatVector& a);
Arguments:	a – Source float vector

Description:	Computes and returns the sum of the absolute values of the elements of float vector a using the algorithm:
	for i=0 to n-1
	{
	result = sum(abs(a[i]))
	}

Example:	FOIL::CFloatVector MyVector1 (3); MyVector1[0] = 1.2; MyVector1[1] = -2.3; MyVector1[2] = 3.4; float fSum; fSum = FOIL::svemg (MyVector1);

Summary:	**Sum of vector element squares**
Syntax:	float svesq (const CFloatVector& a);
Arguments:	a – Source float vector

Description:	Computes and returns the sum the squares of the elements of float vector a using the algorithm:
	for i=0 to n-1
	{
	result = sum(a[i]*a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (3); MyVector1[0] = 1.2; MyVector1[1] = -2.3; MyVector1[2] = 3.4; float fSum; fSum = FOIL::svesq (MyVector1);

Summary:	**Sum of vector signed element squares**
Syntax:	float svs (const CFloatVector& a);
Arguments:	a – Source float vector

Description:	Computes and returns the sum of the signed squares of the elements of float vector a using the algorithm:
	for i=0 to n-1
	{
	result = sum(a[i] * abs(a[i]))
	}

Example:	FOIL::CFloatVector MyVector1 (3); MyVector1[0] = 1.2; MyVector1[1] = -2.3; MyVector1[2] = 3.4; float fSum; fSum = FOIL::svs (MyVector1);

Summary:	**Vector absolute value**
Syntax:	void vabs (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Computes the absolute value of the elements of float vector a and stores the results into float vector c:
	for i=0 to n-1
	{
	c[i] = abs(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (3); FOIL::CFloatVector MyVector2 (3); MyVector1[0] = -1.2; MyVector1[1] = 2.3;

	MyVector1[2] = -3.4; FOIL::vabs (MyVector1, &MyVector2);

Summary:	**Vector add**
Syntax:	void vadd (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Adds the elements of float vectors a and b and stores the results into float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] + b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (3, 1.2); FOIL::CFloatVector MyVector2 (3, 3.4); FOIL::CFloatVector MyVector3 (3); FOIL::vadd (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector add**
Syntax:	CFloatVector operator + (const CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source float vector a b – Source float vector b

Description:	Adds the elements of float vectors a and b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] + b[i]
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (3, 1.2); FOIL::CFloatVector MyVector2 (3, 3.4); FOIL::CFloatVector MyVector3 (3); MyVector3 = MyVector1 + MyVector2;

Summary:	**Vector add**
Syntax:	CFloatVector& operator += (CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source float vector a b – Source float vector b

Description:	Adds the elements of float vectors a and b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] + b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (3, 1.2); FOIL::CFloatVector MyVector2 (3, 3.4); MyVector2 += MyVector1;

Summary:	**Vector anti-log base 10**
Syntax:	void valog10 (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Computes the anti-logarithm base 10 of each element of float vector a and stores the results in float vector c using the algorithm:
	for i=0 to n-1
	{
	c[i] = alog10(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (3, 1.2); FOIL::CFloatVector MyVector2 (3); FOIL::valog10 (MyVector1, &MyVector2);

Summary:	**Vector add and multiply**
Syntax:	void vam (const CFloatVector& a, const CFloatVector& b, const CFloatVector& c, CFloatVector* d);
Arguments:	a – Source float vector a b – Source float vector b c – Source float vector c d – Destination float vector d

Description:	Adds elements of float vectors a and b, multiplies that sum by the corresponding element of float vector c, and stores the results into float vector d:
	for i=0 to n-1
	{
	d[i] = (a[i] + b[i]) * c[i]
	}

Example:	FOIL::CFloatVector MyVector1 (3, 1.2); FOIL::CFloatVector MyVector2 (3, 3.4); FOIL::CFloatVector MyVector3 (3, 5.6); FOIL::CFloatVector MyVector4 (3); FOIL::vam (MyVector1, MyVector2, MyVector3, &MyVector4);

Summary:	**Vector arctangent**
Syntax:	void vatan (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Computes the arctangent of the elements of float vector a and stores the results in radians in float vector c according to the algorithm:
	for i=0 to n-1
	{
	c[i] = atan (a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (3, 12.34); FOIL::CFloatVector MyVector2 (3); FOIL::vatan (MyVector1, &MyVector2);

Summary:	**Vector two argument arctangent**
Syntax:	void vatan2 (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Computes the two argument arctangent of the corresponding elements of float vectors a and b and stores the results in radians in float vector c according to the algorithm:
	for i=0 to n-1
	{
	c[i] = atan2(a[i],b[i])
	}

Example:	FOIL::CFloatVector MyVector1 (3, 12.34); FOIL::CFloatVector MyVector2 (3, 56.78); FOIL::CFloatVector MyVector3 (3); FOIL::vatan2 (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector clip**
Syntax:	void vclip (const CFloatVector& a, float low_bound, float upper_bound, CFloatVector* c);
Arguments:	a – Source float vector upper_bound – Source float scalar low_bound – Source float scalar c – Destination float vector
Description:	vclip clips the value of each element in float vector a to be within the range specified by scalars low_bound and upper_bound. The results are stored in float vector c. The algorithm is: for i=0 to n-1 {

	if a[i] > upper_bound, tmp = upper_bound else tmp = a[i] if tmp < low_bound, tmp = low_bound c[i] = tmp }
Example:	FOIL::CFloatVector MyVector1 (10, 10.0); FOIL::CFloatVector MyVector2 (10); MyVector[4] = -999.999; MyVector[7] = 999.999; FOIL::vclip (MyVector1, -100.0, 100.0, &MyVector2);

Summary:	**Vector cosine**
Syntax:	void vcos (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector a c – Destination float vector c

Description:	Computes the cosine of each element of float vector a and stores the results in vector c
	for i=0 to n-1
	{
	c[i] = cos(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 0.5); FOIL::CFloatVector MyVector2 (10); FOIL::vcos (MyVector1, &MyVector2);

Summary:	**Vector distance**
Syntax:	void vdist (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Computes the square root of the sum of the squares of corresponding elements of float vectors a and b and stores the results into float vector c:
	for i=0 to n-1
	{
	c[i] = sqrt(a[i]*a[i] + b[i]*b[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 0.5); FOIL::CFloatVector MyVector2 (10, 0.9); FOIL::CFloatVector MyVector3 (10); FOIL::vdist (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector divide**
Syntax:	void vdiv (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Divides each element of float vector a by the corresponding element of float vector b and stores the result in float vector c. The algorithm is:
	for i=0 to n-1
	{
	c[i] = a[i] / b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (10, 0.5); FOIL::CFloatVector MyVector2 (10, 0.9); FOIL::CFloatVector MyVector3 (10); FOIL::vdiv (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector divide**
Syntax:	CFloatVector operator / (const CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source float vector a b – Source float vector b
Description:	Divides each element of float vector a by the corresponding element of float vector b

	and returns the result. The algorithm is:
	for i=0 to n-1
	{
	result[i] = a[i] / b[i]
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (10, 0.5); FOIL::CFloatVector MyVector2 (10, 0.9); FOIL::CFloatVector MyVector3; MyVector3 = MyVector1 / MyVector2;

Summary:	**Vector divide**
Syntax:	CFloatVector& operator /= (CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source float vector a b – Source float vector b

Description:	Divides each element of float vector a by the corresponding element of float vector b and returns the result. The algorithm is:
	for i=0 to n-1
	{
	result[i] = a[i] / b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (10, 0.5); FOIL::CFloatVector MyVector2 (10, 0.9); MyVector1 /= MyVector2;

Summary:	**Vector envelope**
Syntax:	void venvlp (const CFloatVector& a, const CFloatVector& b, const CFloatVector& c, CFloatVector* d);
Arguments:	a – Source float vector a b – Source float vector b c – Source float vector c d – Destination float vector d

Description:	Copies elements of float vector c to float vector d when the element is outside the range defined by corresponding elements of float vectors a and b; otherwise it copies 0.0 to the element of vector d. The algorithm is written:
	for i=0 to n-1
	{
	if (c[i] > a[i]) d[i] = c[i]
	else if (c[i] < b[i]) d[i] = c[i]
	else d[i] = 0.0
	}

Example:	FOIL::CFloatVector MyVector1 (3, 1.5); FOIL::CFloatVector MyVector2 (3, 0.5); FOIL::CFloatVector MyVector3 (3); MyVector3[0] = 0.0; MyVector3[1] = 1.0; MyVector3[2] = 2.0; FOIL::CFloatVector MyVector4 (3); FOIL::venvlp (MyVector1, MyVector2, MyVector3, &MyVector4);

Summary:	**Vector exponential**
Syntax:	void vexp (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Computes the natural exponential of each element of float vector a and stores the results in float vector c using the algorithm:
	for i=0 to n-1
	{
	c[i] = exp(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (3, 1.5); FOIL::CFloatVector MyVector2 (3); FOIL::vexp (MyVector1, &MyVector2);

Summary:	**Vector fill with constant**
Syntax:	void vfill (float a, CFloatVector* c);
Arguments:	a – Source float scalar c – Destination float vector

Description:	Fills the elements of float vector c with the float scalar in a.
	for i=0 to n-1
	{
	c[i] = a
	}

Example:	FOIL::CFloatVector MyVector1 (3); FOIL::vfill (123.456, &MyVector1);

Summary:	**Vector float byte integers**
Syntax:	void vflt8 (const CUnsignedCharVector& a, CFloatVector* c);
Arguments:	a – Source unsigned char vector c – Destination float vector

Description:	Converts the elements of unsigned char vector a to type float and stores the results in float vector c
	for i=0 to n-1
	{
	c[i] = float(a[i])
	}

Example:	FOIL::CUnsignedCharVector MyVector1 (5, 10); FOIL::CFloatVector MyVector2 (5); FOIL::vflt8 (MyVector1, &MyVector2);

Summary:	**Vector float short integers**
Syntax:	void vflt16 (const CUnsignedShortVector& a, CFloatVector* c);
Arguments:	a – Source unsigned short vector c – Destination float vector

Description:	Converts the elements of unsigned short vector a to type float and stores the results in float vector c
	for i=0 to n-1
	{
	c[i] = float(a[i])
	}

Example:	FOIL::CUnsignedShortVector MyVector1 (5, 10); FOIL::CFloatVector MyVector2 (5); FOIL::vflt16 (MyVector1, &MyVector2);

Summary:	**Vector float short integers**
Syntax:	void vflt32 (const CIntVector& a, CFloatVector* c);
Arguments:	a – Source int vector c – Destination float vector

Description:	Converts the elements of int vector a to type float and stores the results in float vector c
	for i=0 to n-1
	{
	c[i] = float(a[i])
	}

Example:	FOIL::CIntVector MyVector1 (5, 10); FOIL::CFloatVector MyVector2 (5); FOIL::vflt32 (MyVector1, &MyVector2);

Summary:	**Vector truncate to fraction**
Syntax:	void vfrac (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector

	c – Destination float vector

Description:	Extracts the fractional part of each element of float vector a and stores it in the corresponding element of float vector c using the algorithm
	for i=0 to n-1
	{
	c[i] = a[i] - float(int(a[i]))
	}

Example:	FOIL::CFloatVector MyVector1 (5, 12.34); FOIL::CFloatVector MyVector2 (5); FOIL::vfrac (MyVector1, &MyVector2);

Summary:	**Vector gather**
Syntax:	void vgathr (const CFloatVector& a, const CIntVector& b, CFloatVector* c);
Arguments:	a – Source float vector b – Source indices integer vector c – Destination float vector

Description:	Uses the elements of integer vector b as the indices (in the range 0 to n-1) by which to fetch the elements of float vector a for storage into float vector c. No check is made on the validity of the indices in vector b:
	for i=0 to n-1
	{
	c[i] = a[b[i]]
	}

Example:	FOIL::CFloatVector MyVector1 (3, 12.34); FOIL::CIntVector MyVector2 (3); FOIL::CFloatVector MyVector3 (3); MyVector2[0] = 2; MyVector2[1] = 1; MyVector2[2] = 0; FOIL::vfrac (MyVector1, MyVector2, &MyVector3);

Summary:	**Extract imaginaries of complex vector**
Syntax:	void vimag (const CComplexVector& a, CFloatVector* c);
Arguments:	a – Source complex vector c – Destination float vector

Description:	Forms a float vector c from the imaginary parts of complex vector a
	for i=0 to n-1
	{
	c[i] = imag(a[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (12.34, 56.78)); FOIL::CFloatVector MyVector2 (10); FOIL::vimag (MyVector1, &MyVector2);

Summary:	**Vector limit**
Syntax:	void vlim (const CFloatVector& a, float b, float c, CFloatVector* d);
Arguments:	a – Source float vector b – Source float scalar b c – Source float scalar c d – Destination float vector

Description:	Creates a float vector d with values of only float value c or -c depending on whether the corresponding element of float vector a is less than the threshold value b using the algorithm:
	for i=0 to n-1
	{
	if a[i] < b, d[i] = -c, else, d[i] = c
	}

Example:	FOIL::CFloatVector MyVector1 (4); FOIL::CFloatVector MyVector2 (4); MyVector1[0] = 0.5; MyVector1[1] = 1.5;

	MyVector1[2] = 2.5; MyVector1[3] = 3.5; FOIL::vlim (MyVector1, 2.0, 999.999, &MyVector2);

Summary:	**Vector natural logarithm**
Syntax:	void vlog (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Stores the natural logarithm of the elements of float vector a into float vector c
	for i=0 to n-1
	{
	c[i] = log(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (4, 12.34); FOIL::CFloatVector MyVector2 (4); FOIL::vlog (MyVector1, &MyVector2);

Summary:	**Vector base 10 logarithm**
Syntax:	void vlog10 (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Stores the base 10 logarithm of the elements of float vector a into float vector c
	for i=0 to n-1
	{
	c[i] = log10(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (4, 100.0); FOIL::CFloatVector MyVector2 (4); FOIL::vlog10 (MyVector1, &MyVector2);

Summary:	**Vector multiply and add**
Syntax:	void vma (const CFloatVector& a, const CFloatVector& b, const CFloatVector& c, CFloatVector* d);
Arguments:	a – Source float vector a b – Source float vector b c – Source float vector c d – Destination float vector d

Description:	Multiplies elements of float vectors a and b, adds that product to the corresponding element of float vector c, and stores the result into float vector d using the algorithm:
	for i=0 to n-1
	{
	d[i] = (a[i] * b[i]) + c[i]
	}

Example:	FOIL::CFloatVector MyVector1 (4, 1.2); FOIL::CFloatVector MyVector2 (4, 3.4); FOIL::CFloatVector MyVector3 (4, 5.6); FOIL::CFloatVector MyVector4 (4); FOIL::vma (MyVector1, MyVector2, MyVector3, &MyVector4);

Summary:	**Vector maximum of two vectors**
Syntax:	void vmax (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Select the maximum of corresponding elements of float vectors a and b and stores the result in float vector c:
	for i=0 to n-1
	{
	c[i] = max(a[i], b[i])
	}

Example:	FOIL::CFloatVector MyVector1 (2);

	FOIL::CFloatVector MyVector2 (2); FOIL::CFloatVector MyVector3 (2); MyVector1[0] = 1.0; MyVector1[1] = -4.0; MyVector2[0] = -2.0; MyVector2[1] = 3.0; FOIL::vmax (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector maximum magnitude of two vectors**
Syntax:	void vmaxmg (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Select the maximum of the absolute values of corresponding elements of float vectors a and b and stores the result in float vector c
	for i=0 to n-1
	{
	c[i] = max(abs(a[i]), abs(b[i]))
	}

Example:	FOIL::CFloatVector MyVector1 (2); FOIL::CFloatVector MyVector2 (2); FOIL::CFloatVector MyVector3 (2); MyVector1[0] = 1.0; MyVector1[1] = -4.0; MyVector2[0] = -2.0; MyVector2[1] = 3.0; FOIL::vmaxmg (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector minimum of two vectors**
Syntax:	void vmin (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Selects the minimum values of corresponding elements of float vectors a and b and stores the result in float vector c:
	for i=0 to n-1
	{
	c[i] = min(a[i], b[i])
	}

Example:	FOIL::CFloatVector MyVector1 (2); FOIL::CFloatVector MyVector2 (2); FOIL::CFloatVector MyVector3 (2); MyVector1[0] = 1.0; MyVector1[1] = -4.0; MyVector2[0] = -2.0; MyVector2[1] = 3.0; FOIL::vmin (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector minimum magnitude of two vectors**
Syntax:	void vminmg (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Selects the minimum of the absolute values of corresponding elements of float vectors a and b and stores the result in float vector c
	for i=0 to n-1
	{
	c[i] = min(abs(a[i]), abs(b[i]))
	}

Example:	FOIL::CFloatVector MyVector1 (2); FOIL::CFloatVector MyVector2 (2);

	FOIL::CFloatVector MyVector3 (2); MyVector1[0] = 1.0; MyVector1[1] = -4.0; MyVector2[0] = -2.0; MyVector2[1] = 3.0; FOIL::vminmg (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector multiply and scalar add**
Syntax:	void vmsa (const CFloatVector& a, const CFloatVector& b, float c, CFloatVector* d);
Arguments:	a – Source float vector a b – Source float vector b c – Source float scalar d – Destination float vector

Description:	Multiplies elements of float vectors a and b, adds that product to float scalar c, and stores the result into float vector d using the algorithm:
	for i=0 to n-1
	{
	d[i] = (a[i] * b[i]) + c
	}

Example:	FOIL::CFloatVector MyVector1 (2, 1.2); FOIL::CFloatVector MyVector2 (2, 3.4); FOIL::CFloatVector MyVector3 (2); FOIL::vmsa (MyVector1, MyVector2, 5.6, &MyVector3);

Summary:	**Vector multiply and subtract**
Syntax:	void vmsb (const CFloatVector& a, const CFloatVector& b, const CFloatVector& c, CFloatVector* d);
Arguments:	a – Source float vector a b – Source float vector b c – Source float scalar d – Destination float vector

Description:	Multiplies elements of float vectors a and b, subtracts from that product the corresponding element of float vector c, and stores the result into float vector d using the algorithm:
	for i=0 to n-1
	{
	d[i] = (a[i] * b[i]) - c[i]
	}

Example:	FOIL::CFloatVector MyVector1 (2, 1.2); FOIL::CFloatVector MyVector2 (2, 3.4); FOIL::CFloatVector MyVector3 (2); FOIL::vmsb (MyVector1, MyVector2, 5.6, &MyVector3);

Summary:	**Float vector multiply**
Syntax:	void vmul (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Multiplies the corresponding elements of float vectors a and b and stores the results into float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] * b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (2, 1.2); FOIL::CFloatVector MyVector2 (2, 3.4); FOIL::CFloatVector MyVector3 (2); FOIL::vmul (MyVector1, MyVector2, &MyVector3);

Summary:	**Float vector multiply**
Syntax:	CFloatVector operator * (const CFloatVector& a, const CFloatVector& b);

Arguments:	a – Source float vector a b – Source float vector b

Description:	Multiplies the corresponding elements of float vectors a and b and returns the results:
	for i=0 to n-1
	{
	result[i] = a[i] * b[i]
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (2, 1.2); FOIL::CFloatVector MyVector2 (2, 3.4); FOIL::CFloatVector MyVector3 (2); MyVector3 = MyVector1 * MyVector2;

Summary:	**Float vector multiply**
Syntax:	CFloatVector& operator *= (CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source float vector a b – Source float vector b

Description:	Multiplies the corresponding elements of float vectors a and b and returns the results:
	for i=0 to n-1
	{
	result[i] = a[i] * b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (2, 1.2); FOIL::CFloatVector MyVector2 (2, 3.4); MyVector2 *= MyVector1;

Summary:	**Vector negative absolute value**
Syntax:	void vnabs (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Stores the negative absolute values of the elements of float vector a into float vector c:
	for i=0 to n-1
	{
	c[i] = -abs(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10); FOIL::vnabs (MyVector1, &MyVector2);

Summary:	**Vector negate**
Syntax:	void vneg (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Stores the negative of the elements of float vector a into float vector c:
	for i=0 to n-1
	{
	c[i] = -a[i]
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10); FOIL::vneg (MyVector1, &MyVector2);

Summary:	**Vector negate**
Syntax:	CFloatVector operator - (const CFloatVector& a);
Arguments:	a – Source float vector
Description:	Returns the negative of the elements of float vector a: for i=0 to n-1 { result[i] = -a[i]

	} This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.
Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10); MyVector2 = -MyVector1;

Summary:	**Vector polynomial evaluation**
Syntax:	void vpoly (const CFloatVector& a, const CFloatVector& b, CFloatVector* c, int order);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c order – Source integer scalar

Description:	Evaluates the polynomial whose coefficients are provided in float vector a for each element of float vector b used as the independent variable and stores the results into float vector c. The coefficients are arranged in descending order in vector a. Argument order specifies the order of the polynomial stored in vector a and must be greater than or equal to 0. The evaluation algorithm is:
	for i=0 to n-1
	{
	for j=0 to order
	{
	c[i] = sum(a[j] * b[i]**(order-j))
	}
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10, 3.4); FOIL::CFloatVector MyVector3 (10); FOIL::vpoly (MyVector1, MyVector2, &MyVector3, 5);

Summary:	**Vector fill with ramp**
Syntax:	void vramp (float a, float b, CFloatVector* c);
Arguments:	a – Source float scalar a b – Source float scalar b c – Destination float vector c

Description:	Stores the ramp specified by float scalars a and b into float vector c using the algorithm:
	for i=0 to n-1
	{
	c[i] = a + (i * b)
	}

Example:	FOIL::CFloatVector MyVector1 (10); FOIL::vramp (1.2, 3.4, &MyVector1);

Summary:	**Vector reciprocal**
Syntax:	void vrcip (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Computes the reciprocal of float vector a putting the result in float vector c:
	for i=0 to n-1
	{
	c[i] = 1.0 / a[i]
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10); FOIL::vrcip (MyVector1, &MyVector2);

Summary:	**Extract reals of complex vector**
Syntax:	void vreal (const CComplexVector& a, CFloatVector* c);
Arguments:	a – Source complex vector

	c – Destination float vector

Description:	Copies the real parts of complex vector a to float vector c
	for i=0 to n-1
	{
	c[i] = real(a[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4); FOIL::CFloatVector MyVector2 (10); FOIL::vreal (MyVector1, &MyVector2);

Summary:	**Vector scalar add**
Syntax:	void vsadd (const CFloatVector& a, float b, CFloatVector* c);
Arguments:	a – Source float vector b – Source float scalar c – Destination float vector

Description:	Adds scalar b to each element of float vector a and stores the results into float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] + b
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10); FOIL::vsadd (MyVector1, 3.4, &MyVector2);

Summary:	**Vector subtract and multiply**
Syntax:	void vsbm (const CFloatVector& a, const CFloatVector& b, const CFloatVector& c, CFloatVector* d);
Arguments:	a – Source float vector a b – Source float vector b c – Source float vector c d – Destination float vector d

Description:	Subtracts corresponding elements of float vector b from those of float vector a, multiplies that difference by the corresponding element of float vector c, and stores the result into float vector d using the algorithm:
	for i=0 to n-1
	{
	d[i] = (a[i] - b[i]) * c[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10, 1.2); FOIL::CFloatVector MyVector3 (10, 1.2); FOIL::CFloatVector MyVector4 (10); FOIL::vsbm (MyVector1, MyVector2, MyVector3, &MyVector4);

Summary:	**Vector scatter**
Syntax:	void vscatr (const CFloatVector& a, const CIntVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source integer vector b c – Destination float vector c

Description:	Fetches elements from float vector a and stores those elements into vector c using indices from integer vector b to specify locations in vector c in which to store:
	for i=0 to n-1
	{
	c[b[i]] = a[i]
	}
	If vector a is a viewport then it must have stride attribute equal to 1.

Example:	FOIL::CFloatVector MyVector1 (2, 1.2); FOIL::CIntVector MyVector2 (2); FOIL::CFloatVector MyVector3 (2); MyVector2[0] = 1; MyVector2[1] = 0;

	FOIL::vscatr (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector scalar divide**
Syntax:	void vsdiv (const CFloatVector& a, float b, CFloatVector* c);
Arguments:	a – Source float vector b – Source float scalar c – Destination float vector

Description:	Divides the elements of float vector a by scalar b and stores the result into float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] / b
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5); FOIL::vsdiv (MyVector1, 3.4, &MyVector2);

Summary:	**Vector scalar divide**
Syntax:	CFloatVector operator / (const CFloatVector& a, float b);
Arguments:	a – Source float vector b – Source float scalar

Description:	Divides the elements of float vector a by scalar b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] / b
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5); MyVector2 = MyVector1 / 3.4;

Summary:	**Vector scalar divide**
Syntax:	CFloatVector& operator /= (CFloatVector& a, float b);
Arguments:	a – Source float vector b – Source float scalar

Description:	Divides the elements of float vector a by scalar b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] / b
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); MyVector1 /= 3.4;

Summary:	**Vector sine**
Syntax:	void vsin (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector a c – Destination float vector c

Description:	Stores the sine of the elements of float vector a into to float vector c:
	for i=0 to n-1
	{
	c[i] = sin(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5); FOIL::vsin (MyVector1, &MyVector2);

Summary:	**Vector scalar multiply and add**
Syntax:	void vsma (const CFloatVector& a, float b, const CFloatVector& c, CFloatVector* d);
Arguments:	a – Source float vector a

	b – Source float scalar b c – Source float vector c d – Destination float vector d

Description:	Multiplies elements of float vector a by scalar b, adds the corresponding element of float vector c, and stores the result into float vector d using the algorithm:
	for i=0 to n-1
	{
	d[i] = (a[i] * b) + c[i]
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5, 3.4); FOIL::CFloatVector MyVector3 (5); FOIL::vsma (MyVector1, 5.6, MyVector2, &MyVector3);

Summary:	**Vector scalar multiply and scalar add**
Syntax:	void vsmsa (const CFloatVector& a, float b, float c, CFloatVector* d);
Arguments:	a – Source float vector b – Source float scalar b c – Source float scalar c d – Destination float vector

Description:	Multiplies elements of float vector a by scalar b, adds float scalar c, and stores the result into float vector d using the algorithm:
	for i=0 to n-1
	{
	d[i] = (a[i] * b) + c
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5); FOIL::vsmsa (MyVector1, 3.4, 5.6, &MyVector2);

Summary:	**Vector scalar multiply and subtract**
Syntax:	void vsmsb (const CFloatVector& a, float b, const CFloatVector& c, CFloatVector* d);
Arguments:	a – Source float vector a b – Source float scalar c – Source float vector c d – Destination float vector d

Description:	Multiplies elements of float vector a by scalar b, subtracts the corresponding element of float vector c, and stores the result into float vector d using the algorithm:
	for i=0 to n-1
	{
	d[i] = (a[i] * b) - c[i]
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5, 3.4); FOIL::CFloatVector MyVector3 (5); FOIL::vsmsa (MyVector1, 5.6, MyVector2, &MyVector3);

Summary:	**Vector scalar multiply**
Syntax:	void vsmul (const CFloatVector& a, float b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float scalar c – Destination float vector

Description:	Multiplies the elements of float vector a by scalar b and stores the result into float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] * b
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5); FOIL::vsmul (MyVector1, 3.4, &MyVector2);

Summary:	**Vector scalar multiply**
Syntax:	CFloatVector operator * (const CFloatVector& a, float b);
Arguments:	a – Source float vector b – Source float scalar

Description:	Multiplies the elements of float vector a by scalar b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] * b
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5); MyVector2 = MyVector1 * 3.4;

Summary:	**Vector scalar multiply**
Syntax:	CFloatVector operator * (float b, const CFloatVector& a);
Arguments:	a – Source float vector b – Source float scalar

Description:	Multiplies the elements of float vector a by scalar b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] * b
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5); MyVector2 = 3.4 * MyVector1;

Summary:	**Vector scalar multiply**
Syntax:	CFloatVector& operator *= (CFloatVector& a, float b);
Arguments:	a – Source float vector b – Source float scalar

Description:	Multiplies the elements of float vector a by scalar b and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] * b
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); MyVector1 *= 3.4;

Summary:	**Vector square**
Syntax:	void vsq (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Stores the square of each element of float vector a into to float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] * a[i]
	}

Example:	FOIL::CFloatVector MyVector1 (5, 3.4); FOIL::CFloatVector MyVector2 (5); FOIL::vsq (MyVector1, &MyVector2);

Summary:	**Vector square root**
Syntax:	void vsqrt (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Stores the square root of each element of float vector a into to float vector c:
	for i=0 to n-1
	{
	c[i] = sqrt(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (5, 3.4); FOIL::CFloatVector MyVector2 (5); FOIL::vsqrt (MyVector1, &MyVector2);

Summary:	**Vector signed square**
Syntax:	void vssq (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector c – Destination float vector

Description:	Multiplies each element of float vector a by the absolute value of the element and stores the result into to float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] * abs(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (5, -3.4); FOIL::CFloatVector MyVector2 (5); FOIL::vssq (MyVector1, &MyVector2);

Summary:	**Vector subtract**
Syntax:	void vsub (const CFloatVector& a, const CFloatVector& b, CFloatVector* c);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

Description:	Subtracts the corresponding elements of float vector b from those of float vector a and stores the result in float vector c:
	for i=0 to n-1
	{
	c[i] = a[i] - b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5, 3.4); FOIL::CFloatVector MyVector3 (5); FOIL::vsub (MyVector1, MyVector2, &MyVector3);

Summary:	**Vector subtract**
Syntax:	CFloatVector operator - (const CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source float vector a b – Source float vector b

Description:	Subtracts the corresponding elements of float vector b from those of float vector a and returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] - b[i]
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5, 3.4); FOIL::CFloatVector MyVector3 (5); MyVector3 = MyVector1 - MyVector2;

Summary:	**Vector subtract**
Syntax:	CFloatVector& operator -= (CFloatVector& a, const CFloatVector& b);
Arguments:	a – Source float vector a b – Source float vector b
Description:	Subtracts the corresponding elements of float vector b from those of float vector a and

	returns the result:
	for i=0 to n-1
	{
	result[i] = a[i] - b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5, 3.4); MyVector1 -= MyVector2;

Summary:	**Vector swap**
Syntax:	void vswap (CFloatVector* a, CFloatVector* b);
Arguments:	a – Source and destination vector a b – Source and destination vector b

Description:	Swaps the elements of float vector a with the corresponding elements of float vector b:
	for i=0 to n-1
	{
	temp = b[i]
	b[i] = a[i]
	a[i] = temp
	}

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (5, 3.4); FOIL::vswap (&MyVector1, &MyVector2);

Summary:	**Vector tangent**
Syntax:	void vtan (const CFloatVector& a, CFloatVector* c);
Arguments:	a – Source float vector a c – Destination float vector c

Description:	Computes the tangent of each element of float vector a (in radians) and stores the result into float vector c:
	for i=0 to n-1
	{
	c[i] = tan(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (5, 0.1); FOIL::CFloatVector MyVector2 (5); FOIL::vtan (MyVector1, &MyVector2);

Summary:	**Vector threshold**
Syntax:	void vthresh (const CFloatVector& a, float b, CFloatVector* c);
Arguments:	a – Source float vector b – Source float scalar c – Destination float vector

Description:	Limits the value of the elements of float vector a to be greater than or equal to the threshold in scalar b and stores the result into float vector c:
	for i=0 to n-1
	{
	if a[i] < b, c[i] = b else, c[i] = a[i]
	}

Example:	FOIL::CFloatVector MyVector1 (3); FOIL::CFloatVector MyVector2 (3); MyVector1[0] = 0.5; MyVector1[1] = 1.5; MyVector1[2] = 2.5; FOIL::vthresh (MyVector1, 1.0, &MyVector2);

Summary:	**Convolution**
Syntax:	void conv (const CFloatVector& a, const CFloatVector& b, CFloatVector* c, int ndec);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c

	ndec – Source integer scalar

Description:	Performs the convolution of float vectors a and b and stores the results in float vector c. The ndec decimation factor specifies what portion of the output value are actually computed. A value of ndec =3 indicates that only every third possible output value is computed. The algorithm is:
	for i=0 to nc-1, j=0 to nb-1
	{
	c[i] = sum(a[ndec*i + j*ia] * b[nb-(j-1)*ib])
	}
	The number of items in vector a must be equal to ((nc-1)*ndec + nb), where nb and nc are dimenstions of vectors b and c.

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (3, 3.4); FOIL::CFloatVector MyVector3 (3); FOIL::vthresh (MyVector1, MyVector2, &MyVector3, 1);

Summary:	**Correlation**
Syntax:	void corr (const CFloatVector& a, const CFloatVector& b, CFloatVector* c, int ndec);
Arguments:	a – Source float vector a b – Source float vector b c – Destination float vector c ndec – Source integer scalar

Description:	Performs the correlation of float vectors a and b and stores the results in float vector c. The ndec decimation factor specifies what portion of the output value are actually computed. A value of ndec =3 indicates that only every third possible output value is computed. The algorithm is:
	for i=0 to nc-1, j=0 to nb-1
	{
	c[i] = sum(a[ndec*i + j*ia] * b[j*ib])
	}
	The number of items in vector a must be equal to ((nc-1)*ndec + nb), where nb and nc are dimenstions of vectors b and c.

Example:	FOIL::CFloatVector MyVector1 (5, 1.2); FOIL::CFloatVector MyVector2 (3, 3.4); FOIL::CFloatVector MyVector3 (3); FOIL::vthresh (MyVector1, MyVector2, &MyVector3, 1);

Summary:	**Inverse float to complex FFT (not in place)**
Syntax:	void rfftbi (const CComplexVector& a, CFloatVector* c);
Arguments:	a – Source complex vector c – Destination float vector
Description:	Computes inverse real-to-complex FFT on the data in vector a, and stores the results into vector c. A call must have been previously made to the function bldwts in order to build a weight vector required by the FFT functions. bldwts need only be called once, with an argument specifying the maximum FFT length required in calls to the this routine. If vector a or c is a viewport then it must have stride attribute equal to 1. The size of vector a must be equal to size of vector c and must be a power of 2.
Example:	FOIL::CComplexVector MyVector1 (8, FOIL::CComplex (1.2, 3.4)); FOIL::CFloatVector MyVector1 (8); bldwts (8); FOIL::rfftbi (MyVector1, &MyVector2);

Summary:	**Inverse float to complex FFT (in place)**
Syntax:	void rffti(CComplexVector* a, CFloatVector* c);
Arguments:	a – Source complex vector (data will be detached from this vector) c – Destination float vector
Description:	Computes inverse complex-to-float FFT on the detached data from complex vector a and attach them to float vector c. All previous data in c will be lost. Vector a is a complex vector in packed format. The size of vector a must be a power of 2. If vector a is a viewport then it must have ‘stride’ attribute equal to 1.

Example:	FOIL::CComplexVector MyVector1 (8, FOIL::CComplex (1.2, 3.4)); FOIL::CFloatVector MyVector1 (8); bldwts (8); FOIL::rffti (MyVector1, &MyVector2);

e) Complex

Methods operating on vectors with complex float elements.

Summary:	**Complex dot product**
Syntax:	CComplex cdotpr (const CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector a b – Source complex vector b
Description:	Computes the dot product of two complex vectors a and b and return a complex scalar as result using the algorithm: real© = sum(real(a[i]) * real(b[i]) - imag(a[i]) * imag(b[i])) imag© = sum(real(a[i]) * imag(b[i]) + imag(a[i]) * real(b[i]))
Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (4.5, 6.7)); CComplex cResult = FOIL::cdotpr (MyVector1, MyVector2);

Summary:	**Complex vector addition**
Syntax:	void cvadd (const CComplexVector& a, const CComplexVector& b, CComplexVector* c);
Arguments:	a – Source complex vector a b – Source complex vector b c – Destination complex vector c

Description:	Adds complex vectors a and b and stores the results in complex vector c:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i]) + real(b[i])
	imag(c[i]) = imag(a[i]) + imag(b[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (4.5, 6.7)); FOIL::CComplexVector MyVector3; FOIL::cvadd (MyVector1, MyVector2, &MyVector3);

Summary:	**Complex vector addition**
Syntax:	CComplexVector operator + (const CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector a b – Source complex vector b

Description:	Adds complex vectors a and b and returns the results:
	for i=0 to n-1
	{
	real(result[i]) = real(a[i]) + real(b[i])
	imag(result[i]) = imag(a[i]) + imag(b[i])
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (4.5, 6.7)); FOIL::CComplexVector MyVector3; MyVector3 = MyVector1 + MyVector2;

Summary:	**Complex vector addition**
Syntax:	CComplexVector& operator += (CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector a b – Source complex vector b
Description:	Adds complex vectors a and b and returns the results: for i=0 to n-1

	{
	real(result[i]) = real(a[i]) + real(b[i])
	imag(result[i]) = imag(a[i]) + imag(b[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (4.5, 6.7)); MyVector2 += MyVector1;

Summary:	**Complex vector combine**
Syntax:	void cvcomb (const CFloatVector& a, const CFloatVector& b, CComplexVector* c);
Arguments:	a – Source float vector a representing real part of complex vector b – Source float vector a representing imaginary part of complex vector c – Destination complex vector

Description:	Uses the elements of float vectors a and b to form complex vector c. The real elements of c are taken from vector a; the imaginary elements are taken from vector b:
	for i=0 to n-1
	{
	real (c[i]) = a[i]
	imag (c[i]) = b[i]
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CFloatVector MyVector2 (10, 3.4); FOIL::CComplexVector MyVector3 (10); FOIL::cvcomb (MyVector1, MyVector2, &MyVector3);

Summary:	**Complex vector conjugate**
Syntax:	void cvconj (const CComplexVector& a, CComplexVector* c);
Arguments:	a – Source complex vector c – Destination complex vector

Description:	Conjugates the element of complex vector a and stores the results in complex vector c:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i])
	imag(c[i]) = -imag(a[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::cvconj (MyVector1, &MyVector2);

Summary:	**Complex vector complex scalar multiply**
Syntax:	void cvcsml (const CComplexVector& a, const CComplex& b, CComplexVector* c);
Arguments:	a – Source complex vector a b – Source complex scalar c – Destination complex vector

Description:	Multiplies elements of complex vector a by complex scalar b and stores the result into complex vector c:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i])*real(b) - imag(a[i])*imag(b)
	imag(c[i]) = real(a[i])*imag(b) + imag(a[i])*real(b)
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::CComplex MyScalar (5.6, 7.8); FOIL::cvcsml (MyVector1, MyScalar, &MyVector2);

Summary:	**Complex vector complex scalar multiply**
Syntax:	CComplexVector operator * (const CComplexVector& a, const CComplex& b);
Arguments:	a – Source complex vector a b – Source complex scalar
Description:	Multiplies elements of complex vector a by complex scalar b and returns result:

	for i=0 to n-1
	{
	real(result[i]) = real(a[i])*real(b) - imag(a[i])*imag(b)
	imag(result[i]) = real(a[i])*imag(b) + imag(a[i])*real(b)
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::CComplex MyScalar (5.6, 7.8); MyVector2 = MyVector1 * MyScalar;

Summary:	**Complex vector complex scalar multiply**
Syntax:	CComplexVector operator * (const CComplex& b, const CComplexVector& a);
Arguments:	a – Source complex vector a b – Source complex scalar

Description:	Multiplies elements of complex vector a by complex scalar b and returns result:
	for i=0 to n-1
	{
	real(result[i]) = real(a[i])*real(b) - imag(a[i])*imag(b)
	imag(result[i]) = real(a[i])*imag(b) + imag(a[i])*real(b)
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::CComplex MyScalar (5.6, 7.8); MyVector2 = MyScalar * MyVector1;

Summary:	**Complex vector complex scalar multiply**
Syntax:	CComplexVector& operator *= (CComplexVector& a, const CComplex& b);
Arguments:	a – Source complex vector a b – Source complex scalar

Description:	Multiplies elements of complex vector a by complex scalar b and returns result:
	for i=0 to n-1
	{
	real(result[i]) = real(a[i])*real(b) - imag(a[i])*imag(b)
	imag(result[i]) = real(a[i])*imag(b) + imag(a[i])*real(b)
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplex MyScalar (5.6, 7.8); MyVector1 *= MyScalar;

Summary:	**Complex vector divide**
Syntax:	void cvdiv (const CComplexVector& a, const CComplexVector& b, CComplexVector* c);
Arguments:	a – Source complex vector (dividend) b – Source complex vector (divisor) c – Destination complex vector (quotient)

Description:	Divides complex vector a by complex vector b and stores the results in complex vector c. The vector b may not contain elements with both real and complex values of 0.0:
	for i=0 to n-1
	{
	real(c[i]) = (real(a[i]) * real(b[i]) + imag(a[i]) * imag(b[i])) / (real(b[i])**2 + imag(b[i])**2)
	imag(c[i]) = (imag(a[i]) * real(b[i]) - real(a[i]) * imag(b[i])) / (real(b[i])**2 + imag(b[i])**2)
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexVector MyVector3 (10); FOIL::cvdiv (MyVector1, MyVector2, &MyVector3);

Summary:	**Complex vector divide**
Syntax:	CComplexVector operator / (const CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector (dividend) b – Source complex vector (divisor)

Description:	Divides complex vector a by complex vector b and returns the results. The vector b may not contain elements with both real and complex values of 0.0:
	for i=0 to n-1
	{
	real(result[i]) = (real(a[i]) * real(b[i]) + imag(a[i]) * imag(b[i])) / (real(b[i])**2 + imag(b[i])**2)
	imag(result[i]) = (imag(a[i]) * real(b[i]) - real(a[i]) * imag(b[i])) / (real(b[i])**2 + imag(b[i])**2)
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexVector MyVector3 (10); MyVector3 = MyVector1 / MyVector2;

Summary:	**Complex vector divide**
Syntax:	CComplexVector& operator /= (CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector (dividend) b – Source complex vector (divisor)

Description:	Divides complex vector a by complex vector b and returns the results. The vector b may not contain elements with both real and complex values of 0.0:
	for i=0 to n-1
	{
	real(result[i]) = (real(a[i]) * real(b[i]) + imag(a[i]) * imag(b[i])) / (real(b[i])**2 + imag(b[i])**2)
	imag(result[i]) = (imag(a[i]) * real(b[i]) - real(a[i]) * imag(b[i])) / (real(b[i])**2 + imag(b[i])**2)
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); MyVector2 /= MyVector1;

Summary:	**Complex vector exponential**
Syntax:	void cvexp (const CFloatVector& a, CComplexVector* c);
Arguments:	a – Source float vector a c – Destination complex vector c

Description:	Computes the complex exponential of float vector a and stores the results in complex vector c according to the algorithm:
	for i=0 to n-1
	{
	real(c[i]) = cos(a[i])
	imag(c[i]) = sin(a[i])
	}

Example:	FOIL::CFloatVector MyVector1 (10, 1.2); FOIL::CComplexVector MyVector2 (10); FOIL::cvexp (MyVector1, &MyVector2);

Summary:	**Complex vector magnitude squared**
Syntax:	void cvmags (const CComplexVector& a, CFloatVector* c);
Arguments:	a – Source complex vector c – Destination float vector
Description:	Computes square of the magnitude of each element of complex vector a and stores the results in float vector c. The square of the magnitude is: for i=0 to n-1 { c[i] = real(a[i])**2 + imag(a[i])**2

	}
Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CFloatVector MyVector2 (10); FOIL::cvmags (MyVector1, &MyVector2);

Summary:	**Complex vector multiply**
Syntax:	void cvmul (const CComplexVector& a, const CComplexVector& b, CComplexVector* c, bool mode);
Arguments:	a – Source complex vector a b – Source complex vector b c – Destination complex vector c mode – Type of multiplication

Description:	For mode=true, performs a normal complex multiply of each element of two complex vectors a and b and stores the results in complex vector c:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i]) * real(b[i]) - imag(a[i]) * imag(b[i])
	imag(c[i]) = real(a[i]) * imag(b[i]) + imag(a[i]) * real(b[i])
	}
	For mode=false, multiplies vector b by the conjugate of vector a:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i]) * real(b[i]) + imag(a[i]) * imag(b[i])
	imag(c[i]) = real(a[i]) * imag(b[i]) - imag(a[i]) * real(b[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexVector MyVector3 (10); FOIL::CComplexVector MyVector4 (10); FOIL::cvmul (MyVector1, MyVector2, &MyVector3, true); FOIL::cvmul (MyVector1, MyVector2, &MyVector4, false);

Summary:	**Complex vector multiply**
Syntax:	CComplexVector operator * (const CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector a b – Source complex vector b

Description:	Performs complex multiply of each element of two complex vectors ‘a’ and ‘b’ and returns the results:
	for i=0 to n-1
	{
	real(result[i]) = real(a[i]) * real(b[i]) - imag(a[i]) * imag(b[i])
	imag(result[i]) = real(a[i]) * imag(b[i]) + imag(a[i]) * real(b[i])
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexVector MyVector3 (10); MyVector3 = MyVector1 * MyVector2;

Summary:	**Complex vector multiply**
Syntax:	CComplexVector& operator *= (CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector a b – Source complex vector b

Description:	Performs complex multiply of each element of two complex vectors ‘a’ and ‘b’ and returns the results.
	for i=0 to n-1
	{
	real(result[i]) = real(a[i]) * real(b[i]) - imag(a[i]) * imag(b[i])
	imag(result[i]) = real(a[i]) * imag(b[i]) + imag(a[i]) * real(b[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); MyVector2 *= MyVector1;

Summary:	**Complex vector negate**
Syntax:	void cvneg (const CComplexVector& a, CComplexVector* c);
Arguments:	a – Source complex vector c – Destination complex vector

Description:	Negates the elements of complex vector a and stores the results in complex vector c:
	for i=0 to n-1
	{
	real(c[i]) = -real(a[i])
	imag(c[i]) = -imag(a[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::cvneg (MyVector1, &MyVector2);

Summary:	**Complex vector negate**
Syntax:	CComplexVector operator - (const CComplexVector& a);
Arguments:	a – Source complex vector

Description:	Negates the elements of complex vector a and returns the results:
	for i=0 to n-1
	{
	real(result[i]) = -real(a[i])
	imag(result[i]) = -imag(a[i])
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); MyVector2 = -MyVector1;

Summary:	**Complex vector reciprocal**
Syntax:	void cvrcip (const CComplexVector& a, CComplexVector* c);
Arguments:	a – Source complex vector c – Destination complex vector

Description:	Computes the reciprocal of the elements of complex vector a and stores the results in complex vector c:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i]) / (real(a[i])**2 + imag(a[i])**2)
	imag(c[i]) = -imag(a[i]) / (real(a[i])**2 + imag(a[i])**2)
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::cvrcip (MyVector1, &MyVector2);

Summary:	**Complex vector scalar multiply and add**
Syntax:	void cvsma (const CComplexVector& a, const CComplex& b, const CComplexVector& c, CComplexVector* d);
Arguments:	a – Source complex vector a b – Source complex vector b c – Source complex vector c d – Destination complex vector d
Description:	Multiplies elements of complex vector a by complex scalar b, adds the corresponding element of complex vector c, and stores the result into complex vector d: for i=0 to n-1 { real(d[i]) = real(a[i])*real(b) - imag(a[i])*imag(b) + real(c[i])

	imag(d[i]) = real(a[i])*imag(b) + imag(a[i])*real(b) + imag(c[i]) }
Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexVector MyVector3 (10, FOIL::CComplex (9.0, 1.2)); FOIL::CComplexVector MyVector4 (10); FOIL::cvrcip (MyVector1, MyVector2, MyVector3, &MyVector4);

Summary:	**Complex vector subtract**
Syntax:	void cvsub (const CComplexVector& a, const CComplexVector& b, CComplexVector* c);
Arguments:	a – Source complex vector a b – Source complex vector b c – Destination complex vector c

Description:	Subtracts complex vector b from complex vector a and stores the results in complex vector c:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i]) - real(b[i])
	imag(c[i]) = imag(a[i]) - imag(b[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexVector MyVector3 (10); FOIL::cvsub (MyVector1, MyVector2, &MyVector3);

Summary:	**Complex vector subtract**
Syntax:	CComplexVector operator - (const CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector a b – Source complex vector b

Description:	Subtracts complex vector b from complex vector a and returns the results:
	for i=0 to n-1
	{
	real(result[i]) = real(a[i]) - real(b[i])
	imag(result[i]) = imag(a[i]) - imag(b[i])
	}
	This operation causes data blocks reallocating and copying and is not recommended for use for large vectors.

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexVector MyVector3 (10); MyVector3 = MyVector1 - MyVector2;

Summary:	**Complex vector subtract**
Syntax:	CComplexVector& operator -= (CComplexVector& a, const CComplexVector& b);
Arguments:	a – Source complex vector a b – Source complex vector b

Description:	Subtracts complex vector b from complex vector a and returns the results:
	for i=0 to n-1
	{
	real(result[i]) = real(a[i]) - real(b[i])
	imag(result[i]) = imag(a[i]) - imag(b[i])
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10, FOIL::CComplex (5.6, 7.8)); MyVector2 -= MyVector1;

Summary:	**Rectangular to polar conversion**
Syntax:	void polar (const CComplexVector& a, CComplexVector* c);
Arguments:	a – Source complex vector a

	c – Destination complex vector c

Description:	Converts elements of complex vector a from rectangular to polar form and stores the results in complex vector c. The algorithm for the conversion is:
	for i=0 to n-1
	{
	real(c[i]) = sqrt(real(a[i])**2 + imag(a[i])**2)
	imag(c[i]) = atan2(imag(a[i]), real(a[i]))
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::polar (MyVector1, &MyVector2);

Summary:	**Polar to rectangular conversion**
Syntax:	void rect (const CComplexVector& a, CComplexVector* c);
Arguments:	a – Source complex vector a c – Destination complex vector c

Description:	Converts the elements of complex vector a from polar to rectangular form and stores the results in complex vector c. The algorithm for the conversion is:
	for i=0 to n-1
	{
	real(c[i]) = real(a[i]) * cos(imag(a[i]))
	imag(c[i]) = real(a[i]) * sin(imag(a[i]))
	}

Example:	FOIL::CComplexVector MyVector1 (10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector2 (10); FOIL::rect (MyVector1, &MyVector2);

Summary:	**Forward complex FFT (not in place)**
Syntax:	void cfftbf (const CComplexVector& a, CComplexVector *c);
Arguments:	a – Source complex vector a c – Destination complex vector c
Description:	Computes a forward complex FFT on the data in complex vector a and stores the results into complex vector c. A call must have been previously made to the function bldwts in order to build a weight vector required by the FFT functions. bldwts need only be called once, with an argument specifying the maximum FFT length required in calls to this routine. Note: if vector ‘a’ or ‘c’ is a viewport then it must have ‘stride’ attribute equal to 1 and element count of both vectors must be a power of 2.
Example:	FOIL::CComplexVector MyVector1 (8, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector1 (8); bldwts (8); FOIL::cfftbf (MyVector1, &MyVector2);

Summary:	**Inverse complex FFT (not in place)**
Syntax:	void cfftbi (const CComplexVector& a, CComplexVector *c);
Arguments:	a – Source complex vector a c – Destination complex vector c
Description:	Computes a inverse complex FFT on the data in complex vector a and stores the results into complex vector c. A call must have been previously made to the function bldwts in order to build a weight vector required by the FFT functions. bldwts need only be called once, with an argument specifying the maximum FFT length required in calls to this routine. Note: if vector ‘a’ or ‘c’ is a viewport then it must have ‘stride’ attribute equal to 1 and element count of both vectors must be a power of 2.
Example:	FOIL::CComplexVector MyVector1 (8, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexVector MyVector1 (8); bldwts (8); FOIL::cfftbi (MyVector1, &MyVector2);

Summary:	**Forward float to complex FFT (not in place)**

Syntax:	void rfftbf (const CFloatVector& a, CComplexVector *c);
Arguments:	a – Source float vector c – Destination complex vector
Description:	Computes a forward real-to-complex FFT on the data in vector a, and stores the results into vector c. A call must have been previously made to the function bldwts in order to build a weight vector required by the FFT functions. bldwts need only be called once, with an argument specifying the maximum FFT length required in calls to the this routine. If vector a or c is a viewport then it must have 'stride' attribute equal to 1. The size of vectior a must be equal to size of vector c and must be a power of 2.
Example:	FOIL::CFloatVector MyVector1 (8, 1.2); FOIL::CComplexVector MyVector1 (8); bldwts (8); FOIL::rfftbf (MyVector1, &MyVector2);

Summary:	**Forward complex FFT (in place)**
Syntax:	void cfftf (CComplexVector* c);
Arguments:	c – Source/destination complex vector
Description:	Computes forward complex FFT on the data in complex vector c and stores the results back into vector c. The results are not scaled. (They may be scaled using 'cfftsc' to multiply by 1/n.) The size of vector c must be a power of 2. If vector c is a viewport then it must have 'stride' attribute equal to 1.
Example:	FOIL::CComplexVector MyVector1 (8, FOIL::CComplex (1.2, 3.4)); bldwts (8); FOIL::cfftf (&MyVector1);

Summary:	**Inverse complex FFT (in place)**
Syntax:	void cffti (CComplexVector* c);
Arguments:	c – Source/destination complex vector
Description:	Computes inverse complex FFT on the data in complex vector c and stores the results back into vector c. The size of vector c must be a power of 2. If vector c is a viewport then it must have stride attribute equal to 1.
Example:	FOIL::CComplexVector MyVector1 (8, FOIL::CComplex (1.2, 3.4)); bldwts (8); FOIL::cffti (&MyVector1);

Summary:	**Complex FFT scale**
Syntax:	void cfftsc (CComplexVector* c);
Arguments:	c – Source/destination complex vector
Description:	Scales the FFT in complex vector c by dividing the float and imaginary part of each element by n and storing the results back into vector c. Here n is number of complex items in complex vector.
Example:	FOIL::CComplexVector MyVector1 (8, FOIL::CComplex (1.2, 3.4)); FOIL::cfftsc (&MyVector1);

Summary:	**Forward float to complex FFT (in place)**
Syntax:	void rfftf(CFloatVector* a, CComplexVector* c);
Arguments:	a – Source float vector (data will be detached from this vector) c – Destination complex vector
Description:	Computes forward float-to-complex on the detached data from float vector a and attach them to complex vector c. All previous data in c will be lost. The results are not scaled, and they may be scaled using rfftsc to multiply by 1/(2*n). The 'rfftf' function uses a standard packed format for complex results of a float FFT. The forward FFT of an n element float valued vector produces n complex results. But due to symmetry only (n/2+1) complex results are independent. The first n/2 complex results are returned in vector c, except that, since symmetry dictates that the imaginary portions of the 0^{th} and n/2th complex results must be zero, the float portion of the n/2th complex result can be (and is) returned in place of the imaginary part of the 0^{th}(first) complex entry. The size of vector a must be a power of 2. If vector 'a' is a viewport then it must have 'stride' attribute equal to 1.

Example:	FOIL::CFloatVector MyVector1 (8, 1.2); FOIL::CComplexVector MyVector2 (); FOIL::rfftf (&MyVector1, &MyVector2);

Summary:	**Float FFT scale and format**
Syntax:	void rfftsc(CComplexVector* a, int iflag, int iscale);
Arguments:	c – Source/destination complex vector iflag – Source integer scalar iscale – Source integer scalar

Description:	Takes the results of a float to complex forward FFT and packs or unpacks it into a more conventional complex format. In addition, the results may be scaled by a factor of 1/(2*n) or 1/(4*n). If vector a is a viewport then it must have stride attribute equal to 1.
	iflag formatting flag
	• iflag = 0, no packing
	• iflag = 2, unpack into n/2 complex elements.
	• iflag = 3, unpack into n/2+1 complex elements
	• iflag = -2, pack from n/2 complex elements to real fft packed format
	• iflag = -3, pack from n/2+1 complex elements to real fft packed format
	iscale scale flag
	• iscale = 0, no scaling
	• iscale = 1, scale elements of a by 1/(2*n)
	• iscale = -1, scale elements of a by 1/(4*n)
	
	Flag is processed using following algorithm:
	{
	if iflag is 2, then real (A[0]) = real (A[0]), imag (A[0]) = 0.0
	if iflag is 3, then real (A[n/2]) = imag (A[0]), imag (A[n/2]) = 0.0 real (A[0]) = real (A[0]), imag (A[0]) = 0.0
	if iflag is -2, then real (A[0]) = real (A[0]), imag (A[0]) = 0.0
	if iflag is -3, then real (A[0]) = real (A[0]), imag (A[0]) = real (A[n/2])
	}

Example:	FOIL::CFloatVector MyVector1 (8, 1.2); FOIL::CComplexVector MyVector2 (); FOIL::rfftf (&MyVector1, &MyVector2); FOIL::rfftsc (&MyVector2, 2, 0);

B. TMatrix

This chapter describes methods for classes produced using the TMatrix template.

The syntax of object creation using templates is lengthy and complicated. FOIL library contains special substitutes, which can be used in user programs:

Long name	**Aliases for common use**	**Platform independed types aliases**
TMatrix<unsigned char, CUnsignedCharStorage>	CUnsignedCharMatrix	CMatrix8
TMatrix<unsigned short, CUnsignedShortStorage>	CUnsignedShortMatrix	CMatrix16
TMatrix<int, CIntStorage>	CIntMatrix	CMatrix32
TMatrix<float, CFloatStorage>	CFloatMatrix	CMatrixF
TMatrix<CComplex, CStdComplexFloatStorage>	CComplexMatrix	CMatrixC

1. Constructors and destructors

Creating methods for classes produced from TVector template

Summary:	**Default constructor**
Syntax:	TMatrix();
Arguments:	None
Description:	Creates a new matrix object without attached data block. Object can't be used until

	defining matrix size and allocating appropriate data block.
Example:	FOIL::CIntMatrix MyMatrix;

Summary:	**Allocating constructor**
Syntax:	TMatrix(dimension_t number_of_rows, dimension_t number_of_columns);
Arguments:	number_of_rows - Number of rows in the matrix number_of_columns - Number of columns in the matrix
Description:	Creates new matrix object and attaches new allocated data block.
Example:	FOIL::CIntMatrix MyMatrix (10, 10);

Summary:	**Filling constructor**
Syntax:	TMatrix(dimension_t number_of_rows, dimension_t number_of_columns, Value fill_value);
Arguments:	number_of_rows - Number of rows in the matrix number_of_columns - Number of columns in the matrix fill_value - Source value for initial matrix contents
Description:	Creates new matrix object, attaches new allocated data block and fills it by given values.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0);

Summary:	**Copying constructor**
Syntax:	TMatrix(const TMatrix& matrix);
Arguments:	matrix – Source matrix
Description:	Creates new matrix object, attaches new allocated memory block and copy source matrix data to the new one.
Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CIntMatrix MyMatrix2 (MyMatrix1);

Summary:	**Matrix viewport constructor**
Syntax:	TMatrix(const TMatrix& matrix, dimension_t first_row, dimension_t first_column, dimension_t number_of_rows, dimension_t number_of_columns);
Arguments:	matrix - Source matrix first_row - The first row number of submatrix in matrix first_column - The first column number of submatrix in matrix number_of_rows - The number of rows in the submatrix number_of_columns - The number of columns in the submatrix
Description:	Creates a special window named “viewport” to existing matrix. Doesn’t allocate data blocks – new object links to data block of existing matrix.
Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CIntMatrix MyMatrix2 (MyMatrix1, 2, 2, 5, 5);

Default destructor is used for releasing data memory used by the matrix object. Objects consist of an object data and an attached memory area. Attached memory area is released only when the last referring object destructor is called.

2. Operators

Summary:	**Assignment operator**
Syntax:	const TMatrix& operator = (const TMatrix& matrix);
Arguments:	matrix – Source matrix
Description:	Copies data block of source matrix to the target one. Note: copying operations means destroying old data block of the destination matrix (if one exists), creating the new data block and copying data from the data block of source matrix.
Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CIntMatrix MyMatrix2; MyMatrix2 = MyMatrix1;

Summary:	**Array-access-like operator**
Syntax:	Type& operator[](dimension_t index)
Arguments:	index - Source/destination element index

Description:	Accesses matrix elements using array like operator [i]. This syntax form is simpler than the one that the library source code contains. “Type” is a type of matrix items.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); MyVector1[5][3] = 123;

3. Functions description

Summary:	**Gets item value**
Syntax:	Value GetItem(dimension_t row, dimension_t column);
Arguments:	row - Row number of required item column - Column number of required item
Description:	Returns the value of matrix item by the row and column indexes. “Type” is a type of vector item.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); int iItem = MyMatrix.GetItem (3, 5);

Summary:	**Sets item value**
Syntax:	void SetItem(dimension_t row, dimension_t column, const Value& pValue);
Arguments:	row - Row number of required item column - Column number of required item pValue - Value to store in the item
Description:	Sets the item in matrix by the given row and column indexes. “Type” is a type of matrix item.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); MyMatrix.SetItem (3, 5, 123);

Summary:	**Get number of rows**
Syntax:	dimension_t GetRowsNumber();
Arguments:	None
Description:	Gets number of rows in the matrix. (0 if object has no attached data)
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); dimension_t iRows = MyMatrix.GetRowsNumber ();

Summary:	**Get number of columns**
Syntax:	dimension_t GetColumnsNumber();
Arguments:	None
Description:	Gets number of columns in the matrix. (0 if object has no attached data)
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); dimension_t iColumns = MyMatrix.GetColumnsNumber ();

Summary:	**Get number of elements in matrix**
Syntax:	dimension_t GetNumber();
Arguments:	None
Description:	Gets elements quantity in matrix. (0 if object has no attached data)
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); dimension_t iDimension = MyMatrix.GetNumber ();

Summary:	**Detects attached data presence**
Syntax:	bool IsEmpty (void);
Arguments:	None
Description:	Returns TRUE if object has no attached data, FALSE otherwise.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); bool oEmpty = MyMatrix.IsEmpty ();

Summary:	**Get row stride**
Syntax:	dimension_t GetRowStride();
Arguments:	None
Description:	Gets row stride of the matrix or matrix viewport
Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CIntMatrix MyMatrix2 (MyMatrix1, 2, 2, 5, 5);

	dimension_t iRowStride = MyMatrix2.GetRowStride ();

Summary:	**Simple initialization**
Syntax:	void Initialize(dimension_t number_of_rows, dimension_t number_of_columns);
Arguments:	number_of_rows - Number of rows in matrix number_of_columns - Number of columns in matrix
Description:	Initializes matrix with the new data block – creates and attaches it to matrix object. Old data block will be removed (if exists). New data block is uninitialized.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); MyMatrix.Initialize (5, 5);

Summary:	**Filling initialization**
Syntax:	void Initialize(dimension_t number_of_rows, dimension_t number_of_columns, Value fill_value);
Arguments:	number_of_rows - Number of rows in matrix number_of_columns - Number of columns in matrix fill_value - Source value for initial matrix contents
Description:	Initializes matrix with the new data block – creates and attaches it to matrix object. Old data block will be removed (if exists). New data block is filled by value given.
Example:	FOIL::CIntMatrix MyMatrix (10, 10, 0); MyMatrix.Initialize (5, 5, 123);

Summary:	**Viewport initialization**
Syntax:	void Initialize(const TMatrix& matrix, dimension_t first_row, dimension_t first_column, dimension_t number_of_rows, dimension_t number_of_columns);
Arguments:	matrix - Source matrix first_row - The first row number of submatrix in matrix first_column - The first column number of submatrix in matrix number_of_rows - The number of rows of submatrix number_of_columns - The number of columns of submatrix
Description:	Attaches matrix object to data block of existing matrix. Old data block will be removed (if exists).
Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CIntMatrix MyMatrix2; MyMatrix2.Initialize (MyMatrix1, 3, 3, 5, 5);

Summary:	**Make object empty**
Syntax:	void Reset();
Arguments:	None
Description:	Detaches data block from the matrix object.
Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); MyMatrix1.Reset ();

Summary:	**Fill matrix**
Syntax:	void Fill(Value val);
Arguments:	val – value to be used for filling
Description:	Loads each pixel of matrix with the given value
Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); MyMatrix1.Fill (123);

a) Char

Methods operating on matrices with 8-bit integer elements.

Summary:	**Logical AND of 8-bit images**
Syntax:	void And8(const CMatrix8& a, const CMatrix8& b, CMatrix8 *c);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix c - Destination unsigned char matrix
Description:	Performs a logical “and” of each element of image ‘a’ with the corresponding element in image ‘b’, placing the output at image ‘c’. The algorithm is:

	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j] & b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x1E); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x4A); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); FOIL::And8 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Logical AND of 8-bit images**
Syntax:	CMatrix8 operator & (const CMatrix8& a, const CMatrix8& b);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix

Description:	Performs a logical “and” of each element of image ‘a’ with the corresponding element in image ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] & b[ib*i+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x1E); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x4A); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 & MyMatrix2;

Summary:	**Logical AND of 8-bit images**
Syntax:	CMatrix8& operator &= (CMatrix8& a, const CMatrix8& b);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix

Description:	Performs a logical “and” of each element of image ‘a’ with the corresponding element in image ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] & b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x1E); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x4A); MyMatrix2 &= MyMatrix1;

Summary:	**Logical AND of 8-bit images with complement**
Syntax:	void AndNot8(const CMatrix8& a, const CMatrix8& b, CMatrix8 *c);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix c - Destination unsigned char matrix

Description:	Performs a logical “and” of the complement of each element of image ‘a’ with the corresponding element in image ‘b’, placing the output at image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i++)
	for (j = 0; j < nc; j++)
	c[ic*i+j] = ~a[ia*i+j] & b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x1E); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x4A); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); FOIL::AndNot8 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Average of 8-bit image**
Syntax:	void Ave8(const CMatrix8& a, const CMatrix8& b, float c, CMatrix8* d);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix c – Source float scalar

	d – Destination unsigned char matrix

Description:	Performs a “weighted average” of the two input byte images. Each element of image ‘a’ is scaled by ‘c’ and added to the corresponding element in image ‘b’, which has been scaled by 1.0 - ‘c’. The resultant image is placed at image ‘d’.
	The algorithm is:
	wt1 = *c; // weighting fraction for latest image
	wt2 = 1.0 - wt1;
	for (i = 0; i < nr; i ++)
	{
	for (j = 0; j < nc; j ++)
	{
	d[i*id+j] = (unsigned char)(a[i*ia+j] * wt1 + b[i*ib+j] * wt2);
	}
	}

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x1E); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x4A); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); FOIL::Ave8 (MyMatrix1, MyMatrix2, 0.5, &MyMatrix3);

Summary:	**Histogram equalization of 8 bit image**
Syntax:	void HistoEqual8(const CMatrix8 &a, const CIntVector& b, CMatrix8* c);
Arguments:	a – Source unsigned char matrix b – Source int vector c – Destination unsigned char matrix

Description:	Performs a histogram equalization on the 8-bit image located at ‘a’. The histogram should be located at ‘b’. The output image is placed at image ‘c’. The number of items of vector ‘b’ specifies the number of bins of the histogram and must be equal to 256. If the vector ‘b’ is a viewport then its ‘stride’ attribute must be equal to 1.
	The algorithm is:
	unsigned char tran_fn[MAX8BIT + 1]; // find the total pixel count
	pixel_count = nr * nc; // create image transform lut
	for (i = 0, total = 0; i <= MAX8BIT; i++)
	{
	total += b[i]; // calc. cumulative histogram
	tran_fn[i]=(unsigned char)(MAX8BIT*(float)((float)total/(float)pixel_count) + 0.5);
	}
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = tran_fn[a[ia*i+j]];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x1E); FOIL::CIntVector MyVector (256, 10); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); FOIL::HistoEqual8 (MyMatrix1, MyVector, &MyMatrix2);

Summary:	**Calculate histogram for 8 bit image**
Syntax:	void Histogram8(const CMatrix8& a, CIntVector *c);
Arguments:	a – Source unsigned char matrix c – Destination int vector

Description:	Calculates the histogram for the 8-bit image located at ‘a’. The histogram will be placed at ‘c’. The number of items of vector ‘c’ specifies the number of bins of the histogram and must be equal to 256. If the vector ‘c’ is a viewport then its ‘stride’ attribute must be equal to 1.
	The algorithm is:
	for (i = 0; i < 256; i ++)
	c[i] = 0;
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[a[ia*i+j]] ++;

Example:	FOIL::CUnsignedCharMatrix MyMatrix (10, 10, 0x1E); FOIL::CIntVector MyVector (256); FOIL::Histogram8 (MyMatrix, &MyVector);

Summary:	**Convert non-interlaced to interlaced image**
Syntax:	void Interlace(const CMatrix8& a, CMatrix8* odd, CMatrix8* even);
Arguments:	a – Source unsigned char matrix odd – Destination unsigned char matrix even – Destination unsigned char matrix
Description:	Takes the non-interlaced image stored in 'a', and breaks it up into odd and even field, storing the result in 'odd' and 'even'. Note that the number of rows in 'a' is twice the number of rows in 'odd' and 'even'.
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x1E); FOIL::CUnsignedCharMatrix MyMatrix2 (5, 10); FOIL::CUnsignedCharMatrix MyMatrix3 (5, 10); FOIL::Interlace (MyMatrix1, &MyMatrix2, &MyMatrix3);

Summary:	**Perform lookup on 8-bit image to 8-bit image**
Syntax:	void Lut8(const CMatrix8& a, const CVector8& b, CMatrix8* c);
Arguments:	a – Source unsigned char matrix b – Source unsigned char vector c – Destination unsigned char matrix

Description:	Modifies the 8-bit input image 'a' based on the lookup table which is passed to the function at 'b'. The 8-bit output image will be placed at 'c'. If the vector 'b' is a viewport then its 'stride' attribute must be equal to 1.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = b[a[ia*i+j]];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 123); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); FOIL::CIntVector MyVector (10, 1); FOIL::Lut8 (MyMatrix1, MyVector, &MyMatrix2);

Summary:	**Compute complement of 8-bit image**
Syntax:	void Not8(const CMatrix8& a, CMatrix8* c);
Arguments:	a – Source unsigned char matrix c – Destination unsigned char matrix

Description:	Performs a logical complement of each element of image 'a', placing the output at image 'c'.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = ~a[ia*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 123); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); FOIL::Not8 (MyMatrix1, &MyMatrix2);

Summary:	**Compute complement of 8-bit image**
Syntax:	CMatrix8 operator ~ (const CMatrix8& a);
Arguments:	a – Source unsigned char matrix

Description:	Performs a logical complement of each element of image '. This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = ~a[ia*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 123); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); MyMatrix2 = ~MyMatrix1;

Summary:	**Logical OR of 8-bit images**

Syntax:	void Or8(const CMatrix8& a, const CMatrix8& b, CMatrix8 *c);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix c – Destination unsigned char matrix

Description:	Performs a logical “or” of each element of image ‘a’ with the corresponding element in image ‘b’, placing the output at image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j] | b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 12); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 35); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); FOIL::Or8 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Logical OR of 8-bit images**
Syntax:	CMatrix8 operator | (const CMatrix8& a, const CMatrix8& b);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix

Description:	Performs a logical “or” of each element of image ‘a’ with the corresponding element in image ‘b’ and returns the result. This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] | b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 12); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 35); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 | MyMatrix2;

Summary:	**Logical OR of 8-bit images**
Syntax:	CMatrix8& operator |= (CMatrix8& a, const CMatrix8& b);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix

Description:	Performs a logical “or” of each element of image ‘a’ with the corresponding element in image ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] | b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 12); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 35); MyMatrix2 |= MyMatrix1;

Summary:	**Reflect 8-bit image**
Syntax:	void Reflect8(const CMatrix8& a, CMatrix8* c, flip_t mode);
Arguments:	a – Source unsigned char matrix c – Destination unsigned char matrix mode – Type of reflection
Description:	Flips the input 8-bit image, ‘a’, about the vertical, horizontal or diagonal axis. The axis of reflection is given by the input parameter, ‘mode’. The output image is stored at ‘c’. The type of axis of reflection: enum flip_t { vertical=0, horizontal=1, diagonal=2 };
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 12); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); MyMatrix1[3][7] = 3; FOIL::Reflect8 (MyMatrix1, &MyMatrix2, 0);

Summary:	**Rotate 8-bit image**
Syntax:	void Rotate8(const CMatrix8& a, float b, CMatrix8* c);

Arguments:	a – Source unsigned char matrix b – Source float scalar c – Destination unsigned char matrix
Description:	Performs a rotation of the 8-bit image, ‘a’, about the image center. The angle of rotation is given in degrees by ‘b’, which rotates the image in a counter-clockwise direction with respect to the x-axis. The output image is stored at ‘c’.
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 12); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); MyMatrix1[3][7] = 3; FOIL::Rotate8 (MyMatrix1, 45, &MyMatrix2);

Summary:	**Threshold 8-bit**
Syntax:	void Thr8(const CMatrix8& a, unsigned char b, CMatrix8* c);
Arguments:	a – Source unsigned char matrix b – Source threshold value c – Destination unsigned char matrix
Description:	Thresholds the 8-bit image ‘a’ against the threshold value ‘b’. The output generated is 8-bit image ‘c’, with value decimal 255 for pixels of ‘a’ that equal or exceed the threshold, and 0 for pixels that are less than the threshold.
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 12); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); MyMatrix1[5][5] = 3; FOIL::Thr8 (MyMatrix1, 10, &MyMatrix2);

Summary:	**Threshold 8-bit to least-to-most packed binary**
Syntax:	void Thr8l2m(const CMatrix8& a, unsigned char b, CMatrix8* c);
Arguments:	a – Source unsigned char matrix b – Source threshold value c – Destination unsigned char matrix
Description:	Performs an 8-bit thresholding of image ‘a’ against the threshold value ‘b’, placing the packed binary (ordered least-to-most) in image ‘c’. The number of columns in the matrix ‘a’ should be eight times more than colums number in the matrix ‘c’.
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 80, 0x15); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); MyMatrix1[5][5] = 0x03; FOIL::Thr8l2m (MyMatrix1, 0x10, &MyMatrix2);

Summary:	**Threshold 8-bit to most-to-least packed binary**
Syntax:	void Thr8m2l(const CMatrix8& a, unsigned char b, CMatrix8* c);
Arguments:	a – Source unsigned char matrix b – Source threshold value c – Destination unsigned char matrix
Description:	Performs an 8-bit thresholding of image ‘a’ against the threshold value ‘b’, placing the packed binary (ordered most-to-least) in image ‘c’. The number of columns in the matrix ‘a’ should be eight times more than colums number in the matrix ‘c’.
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 80, 0x15); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10); MyMatrix1[5][5] = 0x03; FOIL::Thr8m2l (MyMatrix1, 0x10, &MyMatrix2);

Summary:	**Convert interlaced to non-interlaced image**
Syntax:	void Uninterlace(const CMatrix8& odd, const CMatrix8& even, CMatrix8* c);
Arguments:	odd – Source unsigned char matrix even – Source unsigned char matrix c – Destination unsigned char matrix
Description:	Takes as input two fields of an interlaced image (odd and even), and synthesizes a non-interlaced image in ‘c’. The odd rows of ‘c’ are built from ‘odd’ and the even rows from ‘even’. Note that the number of rows in ‘c’ is twice the number of rows in ‘odd’ and ‘even’.
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x15);

	FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x26); FOIL::CUnsignedCharMatrix MyMatrix3 (20, 10); FOIL::Uninterlace (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Logical XOR 8-bit images**
Syntax:	void Xor8(const CMatrix8& a, const CMatrix8& b, CMatrix8* c);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix c - Destination unsigned char matrix

Description:	Performs the exclusive-or function between 8-bit images ‘a’ and ‘b’, placing the result in image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x15); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x26); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); FOIL::Xor8 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Logical XOR 8-bit images**
Syntax:	CMatrix8 operator ^ (const CMatrix8& a, const CMatrix8& b);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix

Description:	Performs the exclusive-or function between 8-bit images ‘a’ and ‘b’ and returns the result. This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x15); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x26); FOIL::CUnsignedCharMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 ^ MyMatrix2;

Summary:	**Logical XOR 8-bit images**
Syntax:	CMatrix8& operator ^= (CMatrix8& a, const CMatrix8& b);
Arguments:	a – Source unsigned char matrix b – Source unsigned char matrix

Description:	Performs the exclusive-or function between 8-bit images ‘a’ and ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 0x15); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0x26); MyMatrix2 ^= MyMatrix1;

Summary:	**Convert float image to 8-bit integer**
Syntax:	void Fix8(const CMatrixF& a, CMatrix8* c);
Arguments:	a – Source float matrix c – Destination unsigned char matrix

Description:	Converts a float image stored at ‘a’ to an 8-bit image, which will be placed at ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = (unsigned char) a[ia*i+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10);

	FOIL::Fix8 (MyMatrix1, &MyMatrix2);

Summary:	**Zoom 8 bit image**
Syntax:	void Zoom8(const CMatrix8& a, CMatrix8* c);
Arguments:	a – Source unsigned char matrix c – Destination unsigned char matrix
Description:	Expands or shrinks 8 bit image 'a' to 8 bit image 'c' using bi-linear interpolation.
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 12); FOIL::CUnsignedCharMatrix MyMatrix2 (20, 20); MyMatrix1[5][7] = 26; FOIL::Zoom8 (MyMatrix1, &MyMatrix2);

b) Short

Methods operating on matrices with 16-bit integer elements.

Summary:	**Logical AND of 16 bit images**
Syntax:	void And16(const CMatrix16& a, const CMatrix16& b, CMatrix16 *c);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix c – Destination unsigned short matrix

Description:	Performs a logical "and" of each element of image 'a' with the corresponding element in image 'b', placing the output at image 'c'.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j] & b[ib*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x56); FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); FOIL::And16 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Logical AND of 16 bit images**
Syntax:	CMatrix16 operator & (const CMatrix16& a, const CMatrix16& b);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix

Description:	Performs a logical "and" of each element of image 'a' with the corresponding element in image 'b' and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] & b[ib*i+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x56); FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 & MyMatrix2;

Summary:	**Logical AND of 16 bit images**
Syntax:	CMatrix16& operator &= (CMatrix16& a, const CMatrix16& b);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix

Description:	Performs a logical "and" of each element of image 'a' with the corresponding element in image 'b' and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] & b[ib*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x56); MyMatrix2 &= MyMatrix1;

Summary:	**Logical AND of 16 bit images with complement**
Syntax:	void AndNot16(const CMatrix16& a, const CMatrix16& b, CMatrix16 *c);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix c – Destination unsigned short matrix

Description:	Performs a logical "and" of the complement of each element of image 'a' with the corresponding element in image 'b', placing the output at image 'c'.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = ~a[ia*i+j] & b[ib*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x56); FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); FOIL::AndNot16 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Average of 16 bit image**
Syntax:	void Ave16(const CMatrix16& a, const CMatrix16& b, float c, CMatrix16* d);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix c – Source float scalar d – Destination unsigned short matrix
Description:	Performs a "weighted average" of the two input 16 bit images. Each element of image 'a' is scaled by 'c' and added to the corresponding element in image 'b', which has been scaled by 1.0 - 'c'. The resultant image is placed at image 'd'.
Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x56); FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); FOIL::Ave16 (MyMatrix1, MyMatrix2, 0.5, &MyMatrix3);

Summary:	**Convert float image to 16 bit integer**
Syntax:	void Fix16(const CMatrixF& a, CMatrix16* c);
Arguments:	a – Source float matrix c – Destination unsigned short matrix

Description:	Converts a float image stored at 'a' to an 16-bit image, which will be placed at 'c'.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = (unsigned short) a[ia*i+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 3.4); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); FOIL::Fix16 (MyMatrix1, &MyMatrix2);

Summary:	**Histogram equalization of 10 bit image**
Syntax:	void HistoEqual10(const CMatrix16 &a, const CIntVector& b, CMatrix16* c);
Arguments:	a – Source unsigned short matrix b – Source int vector c – Destination unsigned short matrix
Description:	Performs a histogram equalization on the 10-bit image located at 'a'. The function must be passed the histogram for image 'a'. The user must place the histogram at 'b'. The output image is placed at location 'c'. The number of items of vector 'b' specifies the number of bins of the histogram and must be equal to 1024. If the vector 'b' is a viewport then its 'stride' attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 34); FOIL::CIntVector MyVector (1024, 3); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); FOIL::HistoEqual10 (MyMatrix1, MyVector, &MyMatrix2);

Summary:	**Histogram equalization of 12 bit image**

Syntax:	void HistoEqual12(const CMatrix16 &a, const CIntVector& b, CMatrix16* c);
Arguments:	a – Source unsigned short matrix b – Source int vector c – Destination unsigned short matrix
Description:	Performs a histogram equalization on the 12-bit image located at ‘a’. The function must be passed the histogram for image ‘a’. The user must place the histogram at ‘b’. The output image is placed at location ‘c’. The number of items of vector ‘b’ specifies the number of bins of the histogram and must be equal to 4096. If the vector ‘b’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 34); FOIL::CIntVector MyVector (4096, 3); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); FOIL::HistoEqual12 (MyMatrix1, MyVector, &MyMatrix2);

Summary:	**Histogram equalization of 14 bit image**
Syntax:	void HistoEqual14(const CMatrix16 &a, const CIntVector& b, CMatrix16* c);
Arguments:	a – Source unsigned short matrix b – Source int vector c – Destination unsigned short matrix
Description:	Performs a histogram equalization on the 14-bit image located at ‘a’. The function must be passed the histogram for image ‘a’. The user must place the histogram at ‘b’. The output image is placed at location ‘c’. The number of items of vector ‘b’ specifies the number of bins of the histogram and must be equal to 16384. If the vector ‘b’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 34); FOIL::CIntVector MyVector (16384, 3); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); FOIL::HistoEqual14 (MyMatrix1, MyVector, &MyMatrix2);

Summary:	**Histogram equalization of 16 bit image**
Syntax:	void HistoEqual16(const CMatrix16 &a, const CIntVector& b, CMatrix16* c);
Arguments:	a – Source unsigned short matrix b – Source int vector c – Destination unsigned short matrix
Description:	Performs a histogram equalization on the 16-bit image located at ‘a’. The function must be passed the histogram for image ‘a’. The user must place the histogram at ‘b’. The output image is placed at location ‘c’. The number of items of vector ‘b’ specifies the number of bins of the histogram and must be equal to 65536. If the vector ‘b’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 34); FOIL::CIntVector MyVector (65536, 3); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); FOIL::HistoEqual16 (MyMatrix1, MyVector, &MyMatrix2);

Summary:	**Calculate histogram for 10 bit image**
Syntax:	void Histogram10(const CMatrix16& a, CIntVector *c);
Arguments:	a – Source unsigned short matrix c – Destination int vector
Description:	Calculates the histogram for the 10-bit image located at ‘a’. The histogram will be placed at ‘c’. The number of items of vector ‘c’ specifies the number of bins of the histogram and must be equal to 1024. If the vector ‘c’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix (10, 10, 34); FOIL::CIntVector MyVector (1024); FOIL::Histogram10 (MyMatrix, &MyVector);

Summary:	**Calculate histogram for 12 bit image**
Syntax:	void Histogram12(const CMatrix16& a, CIntVector *c);
Arguments:	a – Source unsigned short matrix c – Destination int vector

Description:	Calculates the histogram for the 12-bit image located at ‘a’. The histogram will be placed at ‘c’. The number of items of vector ‘c’ specifies the number of bins of the histogram and must be equal to 4096. If the vector ‘c’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix (10, 10, 34); FOIL::CIntVector MyVector (4096); FOIL::Histogram12 (MyMatrix, &MyVector);

Summary:	**Calculate histogram for 14 bit image**
Syntax:	void Histogram14(const CMatrix16& a, CIntVector *c);
Arguments:	a – Source unsigned short matrix c – Destination int vector
Description:	Calculates the histogram for the 14-bit image located at ‘a’. The histogram will be placed at ‘c’. The number of items of vector ‘c’ specifies the number of bins of the histogram and must be equal to 16384. If the vector ‘c’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix (10, 10, 34); FOIL::CIntVector MyVector (16384); FOIL::Histogram14 (MyMatrix, &MyVector);

Summary:	**Calculate histogram for 16 bit image**
Syntax:	void Histogram16(const CMatrix16& a, CIntVector *c);
Arguments:	a – Source unsigned short matrix c – Destination int vector
Description:	Calculates the histogram for the 16-bit image located at ‘a’. The histogram will be placed at ‘c’. The number of items of vector ‘c’ specifies the number of bins of the histogram and must be equal to 65536. If the vector ‘c’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CUnsignedShortMatrix MyMatrix (10, 10, 34); FOIL::CIntVector MyVector (65536); FOIL::Histogram16 (MyMatrix, &MyVector);

Summary:	**Perform lookup on 8-bit integer to 16 bit image**
Syntax:	void Lut8s(const CMatrix8& a, const CVector16& b, CMatrix16 *c);
Arguments:	a – Source unsigned char matrix b – Source unsigned short vector c – Destination unsigned short matrix

Description:	Modifies the 8-bit input image ‘a’ based on the 16-bit lookup table which is passed to the function at ‘b’. The 16-bit output image will be placed at ‘c’. If the vector ‘b’ is a viewport then its ‘stride’ attribute must be equal to 1.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	{
	c[ic*i+j] = b[a[ia*i+j]];
	}

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 34); FOIL::CUnsignedShortVector MyVector (256, 3); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); FOIL::Lut8s (MyMatrix1, MyVector, &MyMatrix2);

Summary:	**Compute complement of 16 bit image**
Syntax:	void Not16(const CMatrix16& a, CMatrix16* c);
Arguments:	a – Source unsigned short matrix c – Destination unsigned short matrix

Description:	Performs a logical complement of each element of image ‘a’, placing the output at image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = ~a[ia*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); FOIL::Not16 (MyMatrix1, &MyMatrix2);

Summary:	**Compute complement of 16 bit image**
Syntax:	CMatrix16 operator ~ (const CMatrix16& a);
Arguments:	a – Source unsigned short matrix

Description:	Performs a logical complement of each element of image 'a' and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = ~a[ia*i+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); MyMatrix2 = ~MyMatrix1;

Summary:	**Logical OR of 16 bit image**
Syntax:	void Or16(const CMatrix16& a, const CMatrix16& b, CMatrix16* c);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix c – Destination unsigned short matrix

Description:	Performs a logical "or" of each element of image 'a' with the corresponding element in image 'b', placing the output at image 'c'.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j] | b[ib*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x34); FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); FOIL::Or16 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Logical OR of 16 bit image**
Syntax:	CMatrix16 operator | (const CMatrix16& a, const CMatrix16& b);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix

Description:	Performs a logical "or" of each element of image 'a' with the corresponding element in image 'b' and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] | b[ib*i+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x34); FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 | MyMatrix2;

Summary:	**Logical OR of 16 bit image**
Syntax:	CMatrix16& operator |= (CMatrix16& a, const CMatrix16& b);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix

Description:	Performs a logical "or" of each element of image 'a' with the corresponding element in image 'b' and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] | b[ib*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x34); MyMatrix2 |= MyMatrix1;

Summary:	**Reflect 16-bit image**
Syntax:	void Reflect16(const CMatrix16& a, CMatrix16* c, flip_t mode);
Arguments:	a – Source unsigned short matrix c – Destination unsigned short matrix mode – Source reflection mode
Description:	Flips the input 16-bit image, 'a', about the vertical, horizontal or diagonal axis. The axis of reflection is given by the input parameter, 'mode'. The output image is stored at 'c'. The type of axis of reflection enum flip_t { vertical=0, horizontal=1, diagonal=2 };
Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); MyMatrix1[5][7] = 0x34; FOIL::Reflect16 (MyMatrix1, &MyMatrix2, 0);

Summary:	**Rotate 16-bit image**
Syntax:	void Rotate16(const CMatrix16& a, float b, CMatrix16* c);
Arguments:	a – Source unsigned short matrix b – Source float scalar c – Destination unsigned short matrix
Description:	Performs a rotation of the 16-bit image, 'a', about the image center. The angle of rotation is given in degrees by 'b', which rotates the image in a counter-clockwise direction with respect to the x-axis. The output image is stored at 'c'.
Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10); MyMatrix1[5][7] = 0x34; FOIL::Rotate16 (MyMatrix1, 45, &MyMatrix2);

Summary:	**Logical XOR 16 bit images**
Syntax:	void Xor16(const CMatrix16& a, const CMatrix16& b, CMatrix16* c);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix c – Destination unsigned short matrix

Description:	Performs the exclusive-or function between 16 bit images 'a' and 'b', placing the result in image 'c'.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x34); FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); FOIL::Xor16 (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Logical XOR 16 bit images**
Syntax:	CMatrix16 operator ^ (const CMatrix16& a, const CMatrix16& b);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix

Description:	Performs the exclusive-or function between 16 bit images 'a' and 'b' and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x34);

	FOIL::CUnsignedShortMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 ^ MyMatrix2;

Summary:	**Logical XOR 16 bit images**
Syntax:	CMatrix16& operator ^= (CMatrix16& a, const CMatrix16& b);
Arguments:	a – Source unsigned short matrix b – Source unsigned short matrix

Description:	Performs the exclusive-or function between 16 bit images ‘a’ and ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0x34); MyMatrix2 ^= MyMatrix1;

Summary:	**Zoom 16 bit image**
Syntax:	void Zoom16(const CMatrix16& a, CMatrix16* c);
Arguments:	a – Source unsigned short matrix c – Destination unsigned short matrix
Description:	Expands or shrinks 16 bit image ‘a’ to 16 bit image ‘c’ using bi-linear interpolation.
Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 0x12); FOIL::CUnsignedShortMatrix MyMatrix2 (20, 20); FOIL::Zoom16 (MyMatrix1, &MyMatrix2);

c) Int

Methods operating on matrices with 32-bit integer elements.

Summary:	**Sum 16 bit image to 32 bit image**
Syntax:	void Sum16i(const CMatrix16& a, CMatrix32* c);
Arguments:	a – Source unsigned short matrix c – Destination int matrix

Description:	Sums the 16 bit image data input from image ‘a’ with the 32 bit image ‘c’. The result is saved to ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] += a[ia*i+j];

Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0); FOIL::Sum16i (MyMatrix2, &MyMatrix1);

Summary:	**Sum 16 bit image to 32 bit image**
Syntax:	CMatrix32& operator += (CMatrix32& c, const CMatrix16& a);
Arguments:	a – Source unsigned short matrix c – Source int matrix

Description:	Sums the 16 bit image data input from image ‘a’ with the 32 bit image ‘c’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] += a[ia*i+j];

Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0); MyMatrix1 += MyMatrix2;

Summary:	**Sum 8-bit image to 32 bit image**
Syntax:	void Sum8i(const CMatrix8& a, CMatrix32* c);
Arguments:	a – Source unsigned char matrix c – Destination int matrix

Description:	Sums the 8-bit image data input from image a with the 32 bit image ‘c’. The result is saved to ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] += a[ia*i+j];

Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CUnsignedShortMatrix MyMatrix2 (10, 10, 0); FOIL::Sum8i (MyMatrix2, &MyMatrix1);

Summary:	**Sum 8-bit image to 32 bit image**
Syntax:	CMatrix32& operator += (CMatrix32& c, const CMatrix8& a);
Arguments:	a – Source unsigned char matrix c – Source int matrix

Description:	Sums the 8-bit image data input from image a with the 32 bit image ‘c’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = c[ic*i+j] + a[ia*i+j];

Example:	FOIL::CIntMatrix MyMatrix1 (10, 10, 0); FOIL::CUnsignedCharMatrix MyMatrix2 (10, 10, 0); MyMatrix1 += MyMatrix2;

d) Float

Methods operating on matrices with float elements.

Summary:	**Convert 16 bit integer image to float**
Syntax:	void Float16(const CMatrix16& a, CMatrixF* c);
Arguments:	a – Source unsigned short matrix c – Destination float matrix

Description:	Converts a 16-bit image stored at ‘a’ to a float image, which will be placed at ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 1); FOIL::CFloatMatrix MyMatrix2 (10, 10, 2.3); FOIL::Float16 (MyMatrix1, &MyMatrix2);

Summary:	**Sum 8-bit integer image to float**
Syntax:	void Sum8f(const CMatrix8& a, CMatrixF* c);
Arguments:	a – Source unsigned char matrix c – Destination float matrix

Description:	Sums the 8-bit image data input from image ‘a’ with float image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] += (float) a[ia*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 1); FOIL::CFloatMatrix MyMatrix2 (10, 10, 2.3); FOIL::Sum8f (MyMatrix1, &MyMatrix2);

Summary:	**Sum 8-bit integer image to float**
Syntax:	CMatrixF& operator += (CMatrixF& c, const CMatrix8& a);
Arguments:	a – Source unsigned char matrix c – Source float matrix
Description:	Sums the 8-bit image data input from image ‘a’ with float image ‘c’ and returns the result. The algorithm is: for (i = 0; i < nr; i ++)

	for (j = 0; j < nc; j ++) result[ic*i+j] += (float) a[ia*i+j];
Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 1); FOIL::CFloatMatrix MyMatrix2 (10, 10, 2.3); MyMatrix2 += MyMatrix1;

Summary:	**Perform lookup on 8-bit image to float image**
Syntax:	void Lut8f(const CMatrix8& a, const CVectorF& b, CMatrixF* c);
Arguments:	a – Source unsigned char matrix b – Source float vector c – Destination float matrix

Description:	Modifies the 8-bit input image 'a' based on the lookup table which is passed to the function at 'b'. The float output image will be placed at 'c'. If the vector 'b' is a viewport then its 'stride' attribute must be equal to 1.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = b[a[ia*i+j]];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 1); FOIL::CFloatMatrix MyMatrix2 (10, 10, 2.3); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::Lut8f (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Convert 8-bit integer image to float**
Syntax:	void Float8(const CMatrix8& a, CMatrixF* c);
Arguments:	a – Source unsigned char matrix c – Destination float matrix

Description:	Converts an 8-bit image stored at 'a' to a float image, which will be placed at 'c'.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] = a[ia*i+j];

Example:	FOIL::CUnsignedCharMatrix MyMatrix1 (10, 10, 1); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Lut8f (MyMatrix1, &MyMatrix2);

Summary:	**Absolute value of difference**
Syntax:	void AbsDiff(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Computes the absolute value of the difference between two input images.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = fabs (a[i*ia+j]-b[i*ib+j]);

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::AbsDiff (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Sum of absolute values**
Syntax:	void AbsSum(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Computes the sum of the absolute values between two input images.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = fabs (a[i*ia+j]) + fabs (b[i*ib+j]);

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::AbsSum (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Add images**
Syntax:	void Add(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Computes the arithmetic sum of two input images. The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = a[i*ia+j]+b[i*ib+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::Add (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Add images**
Syntax:	CMatrixF operator + (const CMatrixF& a, const CMatrixF& b);
Arguments:	a – Source float matrix b – Source float matrix

Description:	Computes the arithmetic sum of two input images and returns the result. The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j]+b[i*ib+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 + MyMatrix2;

Summary:	**Add images**
Syntax:	CMatrixF& operator += (CMatrixF& a, const CMatrixF& b);
Arguments:	a – Source float matrix b – Source float matrix

Description:	Computes the arithmetic sum of two input images and returns the result. The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j]+b[i*ib+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); MyMatrix2 += MyMatrix1;

Summary:	**Add scalar to image**
Syntax:	void AddScalar(const CMatrixF& a, float b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float scalar c – Destination float matrix

Description:	Adds to each element of image ‘a’ the scalar given by argument ‘b’, placing the output at image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = a[i*ia+j] + b;

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10);

	FOIL::AddScalar (MyMatrix1, 5.6, &MyMatrix2);

Summary:	**Add scalar to image**
Syntax:	CMatrixF operator + (const CMatrixF& a, float b);
Arguments:	a – Source float matrix b – Source float scalar

Description:	Adds to each element of image ‘a’ the scalar given by argument ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j] + b;
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); MyMatrix2 = MyMatrix1 + 5.6;

Summary:	**Add scalar to image**
Syntax:	CMatrixF operator + (float b, const CMatrixF& a);
Arguments:	a – Source float matrix b – Source float scalar

Description:	Adds to each element of image ‘a’ the scalar given by argument ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j] + b;
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); MyMatrix2 = 5.6 + MyMatrix1;

Summary:	**Add scalar to image**
Syntax:	CMatrixF& operator += (CMatrixF& a, float b);
Arguments:	a – Source float matrix b – Source float scalar

Description:	Adds to each element of image ‘a’ the scalar given by argument ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j] + b;

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); MyMatrix1 += 5.6;

Summary:	**Average of float image**
Syntax:	void Ave(const CMatrixF& a, const CMatrixF& b, float c, CMatrixF* d);
Arguments:	a – Source float matrix b – Source float matrix c – Source float scalar d – Destination float matrix

Description:	Performs a “weighted average” of the two input float images. Each element of image ‘a’ is scaled by ‘c’ and added to the corresponding element in image ‘b’, which has been scaled by 1.0 - ‘c’. The resultant image is placed at image ‘d’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	{
	for (j = 0; j < nc; j ++)

	{
	d[i*id+j] = a[i*ia+j] * c + b[i*ib+j] * (1.0 - c);
	}
	}

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::Ave (MyMatrix1, MyMatrix2, 0.3, &MyMatrix3);

Summary:	**General 2D convolution**
Syntax:	void Conv2D(const CMatrixF& a, const CMatrixF& knl, CMatrixF* c);
Arguments:	a – Source float matrix knl – Source float kernel matrix c – Destination float matrix
Description:	Performs a two-dimensional convolution of the input kernel, 'knl', with the input image 'a'. The output image is placed at 'c'.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyKernel (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Conv2D (MyMatrix1, MyKernel, &MyMatrix2);

Summary:	**3x3 convolution**
Syntax:	void Conv3x3(const CMatrixF& a, CMatrixF* c, const CMatrixF& knl);
Arguments:	a – Source float matrix c – Destination float matrix knl – Source float kernel matrix
Description:	Performs a 3 by 3 convolution of image a with the 3x3 kernel located in 'knl', placing the output at image c.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyKernel (3, 3, 56.78); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Conv3x3 (MyMatrix1, &MyMatrix2, MyKernel);

Summary:	**5x5 convolution**
Syntax:	void Conv5x5(const CMatrixF& a, CMatrixF* c, const CMatrixF& knl);
Arguments:	a – Source float matrix c – Destination float matrix knl – Source float kernel matrix
Description:	Performs a 5 by 5 convolution of image a with the 5x5 kernel located in 'knl', placing the output at image c.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyKernel (5, 5, 56.78); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Conv5x5 (MyMatrix1, &MyMatrix2, MyKernel);

Summary:	**7x7 convolution**
Syntax:	void Conv7x7(const CMatrixF& a, CMatrixF* c, const CMatrixF& knl);
Arguments:	a – Source float matrix c – Destination float matrix knl – Source float kernel matrix
Description:	Performs a 7 by 7 convolution of image a with the 7x7 kernel located in 'knl', placing the output at image c.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyKernel (7, 7, 56.78); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Conv7x7 (MyMatrix1, &MyMatrix2, MyKernel);

Summary:	**Convolution using FFT on float image**
Syntax:	void ConvFFT(const CMatrixF& a, const CMatrixF& knl, CMatrixF* c);
Arguments:	a – Source float matrix

	knl – Source float kernel matrix c – Destination float matrix
Description:	Performs a circular convolution of image ‘a’ with a kernel ‘knl’ by the multiplication of the two functions in the frequency domain. The output image is placed at ‘c’.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyKernel (5, 5, 56.78); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::ConvFFT (MyMatrix1, MyKernel, &MyMatrix2);

Summary:	**Divide images**
Syntax:	void Div(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Divides each pixel of image ‘a’ with the corresponding pixel located in ‘b’, placing the output at image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = a[i*ia+j]/b[i*ib+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::Div (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Divide images**
Syntax:	CMatrixF operator / (const CMatrixF& a, const CMatrixF& b);
Arguments:	a – Source float matrix b – Source float matrix

Description:	Divides each pixel of image ‘a’ with the corresponding pixel located in ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j]/b[i*ib+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 / MyMatrix2;

Summary:	**Divide images**
Syntax:	CMatrixF& operator /= (CMatrixF& a, const CMatrixF& b);
Arguments:	a – Source float matrix b – Source float matrix

Description:	Divides each pixel of image ‘a’ with the corresponding pixel located in ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j]/b[i*ib+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); MyMatrix1 /= MyMatrix2;

Summary:	**Calculate histogram of float image**
Syntax:	void Histogram(const CMatrixF& a, float low, float high, CIntVector* c);
Arguments:	a – Source float matrix low – Source float scalar high – Source float scalar

	c – Destination int vector
Description:	Calculates the histogram for the float image located at ‘a’. The histogram will be placed at ‘c’. Input parameters, ‘high’ and ‘low’, provide the function with the range of the input image data. The number of items of vector ‘c’ specifies the number of bins of the histogram. If the vector ‘c’ is a viewport then its ‘stride’ attribute must be equal to 1.
Example:	FOIL::CFloatMatrix MyMatrix (10, 10, 12.34); FOIL::CFloatMatrix MyVector (256); FOIL::Histogram (MyMatrix, 10.0, 20.0, &MyVector);

Summary:	**Kirsch operator on float image**
Syntax:	void Kirsch(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Applies the kirsch gradient operator to the input image located at ‘a’. The gradient image will be placed at ‘c’.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Kirsch (MyMatrix1, &MyMatrix2);

Summary:	**Perform lookup on float image to float image**
Syntax:	void Lutf(const CMatrixF& a, float low, float high, const CVectorF& b, CMatrixF* c);
Arguments:	a – Source float matrix low – Source float scalar high – Source float scalar b – Source float vector c – Destination float matrix

Description:	Modifies the float input image ‘a’ based on the lookup table which is passed to the function at ‘b’. The float output image will be placed at ‘c’. Input parameters, high and low, provide the function with the range of the input image data. The number of items in vector ‘b’ specifies the number of bins of the lookup table. If the vector ‘b’ is a viewport then its ‘stride’ attribute must be equal to 1.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	{
	idx = (a[ia*i+j] - *low) / (*high - *low) * ((float)nbin);
	if (idx < 0)
	idx = 0;
	if (idx >= nbin)
	idx = nbin - 1;
	c[ic*i+j] = b[idx];
	}

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatVector MyVector (5, 2); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Lutf (MyMatrix1, 1.0, 100.0, MyVector, &MyMatrix2);

Summary:	**Compute magnitude of images**
Syntax:	void Magnitude(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Computes the square root of the sum of the squares of two images.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = sqrt (a[i*ia+j]*a[i*ia+j] * b[i*ib+j]*b[i*ib+j]);

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10);

	FOIL::Magnitude (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Maximum of images**
Syntax:	void Max(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Compares the two input images (‘a’ and ‘b’) to find the maximum pixel value at each pixel location. The maximum pixel values create the output image at ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = max (a[i*ia+j],b[i*ib+j]);

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::Max (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**3x3 median filter on float image**
Syntax:	void Median3x3(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Applies a 3 by 3 median filter to the float input image at ‘a’. The output image ‘c’ is created by replacing a pixel in ‘a’ with the median value of a nine pixel window centered at the particular pixel. The filtering begins where there is complete coverage of the 3x3 kernel. All border pixels are not filtered and passed directly to the output image.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Median3x3 (MyMatrix1, &MyMatrix2);

Summary:	**5x5 median filter on float image**
Syntax:	void Median5x5(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Applies a 5 by 5 median filter to the float input image at ‘a’. The output image ‘c’ is created by replacing a pixel in a with the median value of a 25 pixel window centered at the particular pixel. The filtering begins where there is complete coverage of the 5x5 kernel. All border pixels are not filtered and passed directly to the output image.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Median5x5 (MyMatrix1, &MyMatrix2);

Summary:	**7x7 median filter on float image**
Syntax:	void Median7x7(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Applies a 7 by 7 median filter to the float input image at ‘a’. The output image ‘c’ is created by replacing a pixel in ‘a’ with the median value of a 49 pixel window centered at the particular pixel. The filtering begins where there is complete coverage of the 7x7 kernel. All border pixels are not filtered and passed directly to the output image.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Median7x7 (MyMatrix1, &MyMatrix2);

Summary:	**Minimum of images**
Syntax:	void Min(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Compares the two input images (‘a’ and ‘b’) to find the minimum pixel value at each pixel location. The minimum pixel values create the output image at ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = min (a[i*ia+j],b[i*ib+j]);

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::Min (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Multiply image by scalar**
Syntax:	void MulScalar(const CMatrixF& a, float b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float scalar c – Destination float matrix

Description:	Multiplies each element of image ‘a’ with the scalar given by argument ‘b’, placing the output at image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i++)
	for (j = 0; j < nc; j++)
	c[i*ic+j] = a[i*ia+j] * b;

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::MulScalar (MyMatrix1, 5.6, &MyMatrix2);

Summary:	**Multiply image by scalar**
Syntax:	CMatrixF operator * (const CMatrixF& a, float b);
Arguments:	a – Source float matrix b – Source float scalar

Description:	Multiplies each element of image ‘a’ with the scalar given by argument ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i++)
	for (j = 0; j < nc; j++)
	result[i*ic+j] = a[i*ia+j] * b;
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); MyMatrix2 = MyMatrix1 * 5.6;

Summary:	**Multiply image by scalar**
Syntax:	CMatrixF& operator *= (CMatrixF& a, float b);
Arguments:	a – Source float matrix b – Source float scalar

Description:	Multiplies each element of image ‘a’ with the scalar given by argument ‘b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i++)
	for (j = 0; j < nc; j++)
	result[i*ic+j] = a[i*ia+j] * b;

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); MyMatrix1 *= 5.6;

Summary:	**Multiply image by scalar**
Syntax:	CMatrixF operator * (float b, const CMatrixF& a);
Arguments:	a – Source float matrix b – Source float scalar
Description:	Multiplies each element of image ‘a’ with the scalar given by argument ‘b’ and returns the result.

	The algorithm is:
	for (i = 0; i < nr; i++)
	for (j = 0; j < nc; j++)
	result[i*ic+j] = a[i*ia+j] * b;
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); MyMatrix2 = 5.6 * MyMatrix1;

Summary:	**Power spectrum on float image**
Syntax:	void PowerSpectrum(const CMatrixC& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix

Description:	Determines the power spectrum of the complex FFT located at 'a'. The output power spectrum is placed at 'c'.
	The algorithm is:
	for (i = 0; i < nr; i++)
	{
	for (j = 0; j < nc; j++)
	{
	real = a[i*ia + j].real;
	imag = a[i*ia + j].imag;
	c[i*ic + j] = real*real + imag*imag;
	}
	}

Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::PowerSpectrum (MyMatrix1, &MyMatrix2);

Summary:	**Prewitt operator on float image**
Syntax:	void Prewitt(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Applies the Prewitt gradient operator to the input image located at 'a'. The input image is convolved with the two Prewitt operators, which are listed below. The absolute values of the two edge images are summed to give the gradient image. The gradient image is placed at 'c'.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Prewitt (MyMatrix1, &MyMatrix2);

Summary:	**Reflect float image**
Syntax:	void Reflect(const CMatrixF& a, CMatrixF* c, flip_t mode);
Arguments:	a – Source float matrix c – Destination float matrix mode – Source reflection mode flag
Description:	Flips the input float image, 'a', about the vertical, horizontal or diagonal axis. The axis of reflection is given by the input parameter, mode. The output image is stored at 'c'. The type of axis of reflection enum flip_t { vertical=0, horizontal=1, diagonal=2 };
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Reflect (MyMatrix1, &MyMatrix2, 0);

Summary:	**Roberts operator on float image**
Syntax:	void Roberts(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Applies the Roberts gradient operator to the input image located at 'a'. The input image is convolved with the two Roberts operators. The absolute values of the two edge images are summed to give the gradient image. The gradient image is placed at

	‘c’.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Roberts (MyMatrix1, &MyMatrix2);

Summary:	**Rotate float image**
Syntax:	void Rotate(const CMatrixF& a, float b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float scalar c – Destination float matrix
Description:	Performs a rotation of the float image, ‘a’, about the image center. The angle of rotation is given in degrees by ‘b’, which rotates the image in a counter-clockwise direction with respect to the x-axis. The output image is stored at ‘c’.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Rotate (MyMatrix1, 45, &MyMatrix2);

Summary:	**Sobel operator on float image**
Syntax:	void Sobel(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Applies the Sobel gradient operator to the input image located at ‘a’. The input image is convolved with the two Sobel operators. The absolute values of the two edge images are summed to give the gradient image. The gradient image is placed at ‘c’.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Sobel (MyMatrix1, &MyMatrix2);

Summary:	**Subtract images**
Syntax:	void Sub(const CMatrixF& a, const CMatrixF& b, CMatrixF* c);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix

Description:	Computes the arithmetic difference of two input images ‘a’ and ’b’ and saves result to ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[i*ic+j] = a[i*ia+j]-b[i*ib+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::Sub (MyMatrix1, MyMatrix2, &MyMatrix3);

Summary:	**Subtract images**
Syntax:	CMatrixF operator - (const CMatrixF& a, const CMatrixF& b);
Arguments:	a – Source float matrix b – Source float matrix

Description:	Computes the arithmetic difference of two input images ‘a’ and ’b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j]-b[i*ib+j];
	This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::CFloatMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 - MyMatrix2;

Summary:	**Subtract images**
Syntax:	CMatrixF& operator -= (CMatrixF& a, const CMatrixF& b);
Arguments:	a – Source float matrix b – Source float matrix

Description:	Computes the arithmetic difference of two input images ‘a’ and ’b’ and returns the result.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[i*ic+j] = a[i*ia+j]-b[i*ib+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); MyMatrix1 -= MyMatrix2;

Summary:	**Sum float image to float image**
Syntax:	void Sum(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix

Description:	Sums the float image data input from image ‘a’ with the float image c.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] += a[ia*i+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::Sum (MyMatrix1, &MyMatrix2);

Summary:	**Sum float image to float image**
Syntax:	CMatrixF& operator += (CMatrixF& a, const CMatrixF& c);
Arguments:	a – Source float matrix c – Source float matrix

Description:	Sums the float image data input from image ‘a’ with the float image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	result[ic*i+j] = c[ic*i+j] + a[ia*i+j];

Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); MyMatrix2 += MyMatrix1;

Summary:	**Sum 16-bit image to float image**
Syntax:	void Sum16f(const CMatrix16& a, CMatrixF* c);
Arguments:	a – Source unsigned short matrix c – Destination float matrix

Description:	Sums the 16-bit image data input from image ‘a’ with float image ‘c’.
	The algorithm is:
	for (i = 0; i < nr; i ++)
	for (j = 0; j < nc; j ++)
	c[ic*i+j] += (float) a[ia*i+j];

Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 12); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); FOIL::Sum16f (MyMatrix1, &MyMatrix2);

Summary:	**Sum 16-bit image to float image**
Syntax:	CMatrixF& operator += (CMatrixF& c, const CMatrix16& a);
Arguments:	a – Source unsigned short matrix c – Source float matrix
Description:	Sums the 16-bit image data input from image ‘a’ with float image ‘c’. The algorithm is: for (i = 0; i < nr; i ++)

	for (j = 0; j < nc; j ++) result[ic*i+j] = c[ic*i+j] + (float) a[ia*i+j];
Example:	FOIL::CUnsignedShortMatrix MyMatrix1 (10, 10, 12); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78); MyMatrix2 += MyMatrix1;

Summary:	**X gradient operator on float data**
Syntax:	void Xgradient(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Convolves a row gradient operator with the input image located at 'a'. The X gradient image is placed at 'c'.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Xgradient (MyMatrix1, &MyMatrix2);

Summary:	**Y gradient operator on float data**
Syntax:	void Ygradient(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Convolves a column gradient operator with the input image located at 'a'. The Y gradient image is placed at 'c'.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::Ygradient (MyMatrix1, &MyMatrix2);

Summary:	**Zoom float image**
Syntax:	void Zoom(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Expands or shrinks float image 'a' to float image 'c' using bi-linear interpolation.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (20, 20); FOIL::Zoom (MyMatrix1, &MyMatrix2);

Summary:	**Float matrix transpose**
Syntax:	void mtrans(const CMatrixF& a, CMatrixF* c);
Arguments:	a – Source float matrix c – Destination float matrix
Description:	Transposes the elements of a float matrix 'a' and stores the results in 'c'. 'a' and 'c' must not overlay each other.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10); FOIL::mtrans (MyMatrix1, &MyMatrix2);

Summary:	**Float matrix multiply**
Syntax:	void mmul(const CMatrixF& a, const CMatrixF& b, CMatrixF *c, int ia, int ib, int ic);
Arguments:	a – Source float matrix b – Source float matrix c – Destination float matrix ia – Source int scalar ib – Source int scalar ic – Source int scalar
Description:	Multiplies the two float matrices 'a' and 'b' and stores the results in matrix 'c'. The strides arguments 'ia', 'ib', 'ic' are used to to specify the address increment between consecutive elements to be processed in the matrices 'a', 'b', 'c' respectively. If 'a', 'b' or 'c' are the viewports than they must point to whole source matrices rather than it submatrices.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 12.34); FOIL::CFloatMatrix MyMatrix2 (10, 10, 56.78);

	FOIL::CFloatMatrix MyMatrix3 (10, 10); FOIL::mmul (MyMatrix1, MyMatrix2, &MyMatrix3, 1, 1, 1);

e) Complex

Methods operating on matrices with complex float elements.

Summary:	**Complex image multiplication**
Syntax:	void cimul(const CMatrixC& a, const CMatrixC& b, CMatrixC* c, bool mode);
Arguments:	a – Source complex float matrix b – Source complex float matrix c – Destination complex float matrix mode – Multiplication type
Description:	Multiplies two complex matrices ‘a’ and ‘b’ and stores the results in complex matrix ‘c’. All matrices should point to the uninterrupted memory area (if they are viewports). The multiplication algorithm is: for i=0 to nrows-1, j=0 to ncols-1 if mode == 1 real(c[i,j]) = real(a[i,j])*real(b[i,j]) - imag(a[i,j]) * imag(b[i,j]) imag(c[i,j]) = real(a[i,j])*imag(b[i,j]) + imag(a[i,j]) * real(b[i,j]) else real(c[i,j]) = real(a[i,j])*real(b[i,j]) + imag(a[i,j]) * imag(b[i,j]) imag(c[i,j]) = -real(a[i,j])*imag(b[i,j]) + imag(a[i,j]) * real(b[i,j])
Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexMatrix MyMatrix2 (10, 10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexMatrix MyMatrix3 (10, 10); FOIL::cimul(MyMatrix1, MyMatrix2, &MyMatrix3, false);

Summary:	**Complex image multiplication**
Syntax:	CMatrixC operator * (const CMatrixC& a, const CMatrixC& b);
Arguments:	a – Source complex float matrix b – Source complex float matrix
Description:	Multiplies two complex matrices ‘a’ and ‘b’ and returns the results. All matrices should point to the uninterrupted memory area (if they are viewports). The multiplication algorithm is: for i=0 to nrows-1, j=0 to ncols-1 real(c[i,j]) = real(a[i,j])*real(b[i,j]) - imag(a[i,j]) * imag(b[i,j]) imag(c[i,j]) = real(a[i,j])*imag(b[i,j]) + imag(a[i,j]) * real(b[i,j]) Where ‘c’ is a result matrix. This operation causes data blocks reallocating and copying and is not recommended for use for large matrices.
Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexMatrix MyMatrix2 (10, 10, FOIL::CComplex (5.6, 7.8)); FOIL::CComplexMatrix MyMatrix3 (10, 10); MyMatrix3 = MyMatrix1 * MyMatrix2;

Summary:	**Complex image multiplication**
Syntax:	CMatrixC& operator *= (CMatrixC& a, const CMatrixC& b);
Arguments:	a – Source complex float matrix b – Source complex float matrix
Description:	Multiplies two complex matrices ‘a’ and ‘b’ and returns the result. All matrices should point to the uninterrupted memory area (if they are viewports). The multiplication algorithm is: for i=0 to nrows-1, j=0 to ncols-1 real(c[i,j]) = real(a[i,j])*real(b[i,j]) - imag(a[i,j]) * imag(b[i,j]) imag(c[i,j]) = real(a[i,j])*imag(b[i,j]) + imag(a[i,j]) * real(b[i,j]) Where ‘c’ is a result matrix.
Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexMatrix MyMatrix2 (10, 10, FOIL::CComplex (5.6, 7.8));

	MyMatrix1 *= MyMatrix2;

Summary:	**Forward complex 2D FFT (in place)**
Syntax:	void cfft2df(CMatrixC* a);
Arguments:	a – Destination complex float matrix
Description:	Computes a forward complex two dimensional FFT on the data in complex matrix ‘a’ and stores the results back into matrix ‘a’. The number of columns and number of rows of matrix ‘a’ must be a power of 2. If ‘a’ is a viewport than it must point to whole source matrix rather than it submatrix.
Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::cfft2df (&MyMatrix1);

Summary:	**Inverse complex 2D FFT (in place)**
Syntax:	void cfft2di(CMatrixC* a);
Arguments:	a – Destination complex float matrix
Description:	Computes a inverse complex two dimensional FFT on the data in complex matrix ‘a’ and stores the results back into matrix ‘a’. The number of columns and number of rows of matrix ‘a’ must be a power of 2. If ‘a’ is a viewport than it must point to whole source matrix rather than it submatrix.
Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::cfft2di (&MyMatrix1);

Summary:	**2-D real forward FFT on float image (in place)**
Syntax:	void rfft2df(CMatrixF* a, CMatrixC* c);
Arguments:	a – Destination float matrix c – Destination complex float matrix
Description:	Performs a two dimensional complex FFT on the detached data from float matrix ‘a’ and attach results to complex matrix ‘c’. Forward transform results are not scaled, and may multiplied by 1/(2*nr*nc) (refer to MulScalar), where nr is number of columns in matrix ‘a’, nc - number of rows. The ‘rfft2df’ function uses a standard packed format for storing complex results of a real 2D FFT. The forward FFT of an nr by nc element real valued matrix produces nr*nc complex results. But due to symmetry only nr*nc real results need be stored. The first two columns of result a contain two complex results stored in the rfft packed format (refer to vector rfft function). The remaining column pairs of a form complex results. The matrix function ‘uprft2’ may be used to unpack the result of ‘rfft2df’ into an nr by nc complex matrix, which when scaled produces the same result were the ‘cfft2d’ function used on a real valued complex input matrix. The number of columns and number of rows of matrix ‘a’ must be a power of 2. If ‘a’ is a viewport than it must point to uninterrupted memory area.
Example:	FOIL::CFloatMatrix MyMatrix1 (10, 10, 1.2); FOIL::CComplexMatrix MyMatrix2 (10, 10, FOIL::CComplex (3.4, 5.6)); FOIL::rfft2df (&MyMatrix1, &MyMatrix2);

Summary:	**2-D real inverse FFT on float image (in place)**
Syntax:	void rfft2di(CMatrixC* a, CMatrixF* c);
Arguments:	a – Destination complex float matrix c – Destination complex float matrix
Description:	Performs a two dimensional inverse complex FFT on the detached data from complex matrix ‘a’ and attach results to float matrix ‘c’. The number of columns and number of rows of matrix ‘a’ must be a power of 2. If ‘a’ is a viewport than it must point to uninterrupted memory area.
Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::CFloatMatrix MyMatrix2 (10, 10, 5.6); FOIL::rfft2di (&MyMatrix1, &MyMatrix2);

Summary:	**Unpack results of rfft2df**
Syntax:	void uprft2(const CMatrixC& a, CMatrixC* c);
Arguments:	a – Source complex float matrix c – Destination complex float matrix

Description:	Takes the results of the foward two dimensional real to complex FFT function (rfft2df) and unpacks the results from the packed format of 'rfft2df' to a full nr*nc element complex matrix, where nr is number of columns in matrix 'a', nc - number of rows. The resultant matrix is the same as what would result from using 'cfft2d' on a real valued complex input matrix. If 'a' or 'c' are the viewports than they must point to uninterrupt memory area.
Example:	FOIL::CComplexMatrix MyMatrix1 (10, 10, FOIL::CComplex (1.2, 3.4)); FOIL::CComplexMatrix MyMatrix2 (10, 10, FOIL::CComplex (5.6, 7.8));; FOIL::uprft2(&MyMatrix1, &MyMatrix2);

C. Image abstraction layer

The image abstraction layer is used to define the new entity—"image". The image can be 2 colors (black/white), grayscale (8-, 16-, 32-bit and float) and color. Color images can be compound and unified.

New image types are based on the TMatrix template type (inheritance and aggregation is used). Table 1 below represents the main image types that the library supports.

Name	Compound	Size	Packing	Comment
Gray8	No	8bit	No	Integer 8-bit grayscale image
Gray16	No	16bit	No	Integer 16-bit grayscale image
Gray32	No	32bit	No	Integer 32-bit grayscale image
Gray32F	No	32bit	No	Float 32-bit grayscale image
Complex Gray32F	No	2x 32bit	No	Complex float 32-bit grayscale image
RGBPlanar8	Yes	3x 8-bit	No	3x Integer 8-bit per channel color image (RGB)
RGBPlanar16	Yes	3x 16-bit	No	3x Integer 16-bit per channel color image (RGB)
RGB15	No	16bit	X1 R5 G5 B5	Integer 15-bit packed color image (RGB)
RGB16	No	16bit	R6 G5 B5	Integer 16-bit packed color image (RGB)
RGB32	No	32bit	X8 R8 G8 B8	Integer 32-bit packed color image (RGB)
YUV422	Yes	3x 8-bit	Y [rows]x[cols] U [rows/2]x[cols] V [rows/2]x[cols]	3x Integer 8-bit per channel color image (YUV)
YUV420	Yes	3x 8-bit	Y [rows]x[cols] U [rows/2]x[cols/2] V [rows/2]x[cols/2]	3x Integer 8-bit per channel color image (YUV)
Binary L2M	No	8bit	1 bit per pixel, packed 8 per byte (least to most)	Integer 16-bit packed black-and-white image
Binary M2L	No	8bit	1 bit per pixel, packed 8 per byte (most to least)	Integer 16-bit packed black-and-white image

Table 1 Image types supported by the FOIL library

Data storage format:

There are 2 different methods of data storing for color images: Compound and Unified.

The Compound type of data storing:

Each color channel uses a separate matrix (3 matrices for RGB or YUV color encoding). So, the information about each pixel is separated.

The Unified type of data storing:

Entire image uses only one matrix, and each element contains information about all color components of a pixel (for example: R, G and B). Some image types will have unused bit fields (marked as “X” in the table, for example RGB15).

The main feature of unified images is: it is impossible to perform operations on individual channels. For instance, adding green channels of two RGB16 images requires unpacking each byte before adding and packing bytes back after addition. The main drawbacks of packing/unpacking method are:

- Decreased execution speed;
- Additional memory blocks allocation is required.

Note: The FastSeries functions work only on unpacked images, therefore it is necessary first to unpack the whole image, then to process the whole image, then to pack it back. This operation requires an intermediate memory block.

1. 2 color (black/white images)

The black/white images are classes based on the matrices template.

a) Class name:

CMatrix8

b) Alias:

TMatrix<unsigned char, CUnsignedCharStorage>

2. Grayscale images

The grayscale images are classes based on the matrices template.

Class name	Alias	Full name
CMatrix8	CGrayImage8	TMatrix<unsigned char, CUnsignedCharStorage>
CMatrix16	CGrayImage16	TMatrix<unsigned short, CUnsignedShortStorage>
CMatrix32	CGrayImage32	TMatrix<int, CIntStorage>
CMatrixF	CgrayImageF	TMatrix<float, CFloatStorage>
CMatrixC	No	TMatrix<CComplex, CStdComplexFloatStorage>

Table 2. Types of grayscale images:

3. Color images

The color images are based on color information types.

Class name	Constructors	Methods
CRGB	CRGB (color_t r, color_t g, color_t b); – Constructs image information object from 3 color components CRGB (const CYUV& yuv); – Constructs image information object from another object	color_t R() – Take red component. color_t G() – Take green component. color_t B() – Take blue component.
CYUV	CYUV (color_t y, color_t u, color_t v); – Constructs image information object from 3 color components CYUV (const CRGB& rgb); – Constructs image information object from another object	color_t Y() – Take Y component. color_t U() – Take U component. color_t V() – Take V component.

Table 3 Types of color information

Class name	Full name	Compound
CRGBImage15	TMatrix<CRGB, CRGB15ValueStorage>	No
CRGBImage16	TMatrix<CRGB, CRGB16ValueStorage>	No
CRGBImage32	TMatrix<CRGB, CRGB32ValueStorage>	No
CRGBPlanar8	No	Yes
CRGBPlanar16	No	Yes
CYUV422	No	Yes
CYUV420	No	Yes

Table 4 Types of color images:

All classes for color images contain functions derived from the matrix template with the same syntax (like + operator and AndNot16 method).

There are two constructors: the simple constructor and constructor from color layers.

Examples below demonstrate the constructors for RGBPlanar8 image class:

a) Simple constructor.

CRGBPlanar8(dimension_t height, dimension_t width);

height – Height of the image

width – Width of the image

b) Constructor from color layers.

CRGBPlanar8(const CMatrix8& r_layer, const CMatrix8& g_layer, const CMatrix8& b_layer);

r_layer – Red layer

g_layer – Green layer

b_layer – Blue layer

You can use the following methods for compound images:

c) Function to get the value of the image item (pixel).

CRGB GetItem(dimension_t row, dimension_t column)

row - The row number of the required item

column - The column number of the required item

d) Function to set the value of the image item (pixel).

void SetItem(dimension_t row, dimension_t column, const CRGB& value);

row - The row number of the required item

column - The column number of the required item

value - Value to put in image

e) Get image dimension info.

dimension_t GetRowsNumber();

dimension_t GetColumnsNumber();

f) Access color channels (for RGB Images):

CMatrix16& RLayer();

CMatrix16& GLayer();

CMatrix16& BLayer();

g) Access color channels (for YUV Images):

CMatrix16& YLayer();

CMatrix16& ULayer();

CMatrix16& VLayer();

Unified images should be unpacked before processing. The algorithm is:

- o Unpack all color channels;
- o Process desired unpacked channel as matrix;
- o Pack channels back to the unified image.

The packing/unpacking routines defined for corresponding compound image classes:

h) Pack 8-bit channels to the 15-bit image.

void FOIL_API PackRGB(const CGrayImage8& r_image, const CGrayImage8& g_image, const CGrayImage8& b_image, CRGBImage15 rgb_image);*

r_image – Source unpacked Red channel

g_image – Source unpacked Green channel

b_image – Source unpacked Blue channel

rgb_image – Destination packed RGB image

i) Unpack 8-bit channels from the 15-bit image.

void FOIL_API UnpackRGB(const CRGBImage15& rgb_image, CGrayImage8 r_image, CGrayImage8* g_image, CGrayImage8* b_image);*

rgb_image – Source packed RGB image

r_image – Destination unpacked Red channel

g_image – Destination unpacked Green channel

b_image – Destination unpacked Blue channel

j) Pack 8-bit channels to the 16-bit image.

void FOIL_API PackRGB(const CGrayImage8& r_image, const CGrayImage8& g_image, const CGrayImage8& b_image, CRGBImage16 rgb_image);*

r_image – Source unpacked Red channel

g_image – Source unpacked Green channel

b_image – Source unpacked Blue channel

rgb_image – Destination packed RGB image

k) Unpack 8-bit channels from the 16-bit image.

void FOIL_API UnpackRGB(const CRGBImage16& rgb_image, CGrayImage8 r_image, CGrayImage8* g_image, CGrayImage8* b_image);*

rgb_image – Source packed RGB image

r_image – Destination unpacked Red channel

g_image – Destination unpacked Green channel

b_image – Destination unpacked Blue channel

l) Pack 8-bit channels to the 32-bit image.

void FOIL_API PackRGB(const CGrayImage8& r_image, const CGrayImage8& g_image, const CGrayImage8& b_image, CRGBImage32 rgb_image);*

r_image – Source unpacked Red channel

g_image – Source unpacked Green channel

b_image – Source unpacked Blue channel

rgb_image – Destination packed RGB image

m) Unpack 8-bit channels from the 32-bit image.

void FOIL_API UnpackRGB(const CRGBImage32& rgb_image, CGrayImage8 r_image, CGrayImage8* g_image, CGrayImage8* b_image);*

rgb_image – Source packed RGB image

r_image – Destination unpacked Red channel

g_image – Destination unpacked Green channel

b_image – Destination unpacked Blue channel

4. Conversion routine

The conversion routine can be used to change the format of the image from any to any.

Convert image from one format to another.

void ConvertImageFormat(const T1& from, T2* to, bool scaling = true);

from - source image of format to convert from.

to - pointer to the destination (result) image of format to convert to. Should not be NULL.

scaling – If true – color components are scaled to maximum available destination range. For instance, when scaling from 8-bit to 16-bit, 255 would become 65535. Scaling is not performed if at least one (source of destination) is of floating point type.

IX. HOW TO USE

The list below presents library headers:

- FOIL.hpp – Main include file. Any program that uses the FOIL library should include it;
- FOIL_Decl.hpp – FOIL declarations;
- FOIL_Exceptions.hpp – FOIL exceptions;
- FOIL_Image.hpp – Image types definitions and declarations;
- FOIL_Matrix.hpp – Matrix types definitions and declarations;
- FOIL_Vector.hpp – Vector types definitions and declarations;
- FOIL_RGBImage15.hpp – RGB15 image definitions and declarations;
- FOIL_RGBImage16.hpp – RGB16 image definitions and declarations;
- FOIL_RGBImage32.hpp – RGB32 image definitions and declarations;
- FOIL_RGBPlanar16.hpp – RGBPlanar16 image definitions and declarations;
- FOIL_RGBPlanar8.hpp – RGBPlanar8 image definitions and declarations;
- FOIL_YUV420.hpp – YUV420 image definitions and declarations;
- FOIL_YUV422.hpp – YUV422 image definitions and declarations;

The programs should include only “FOIL.hpp” header. All other headers will be included automatically.

Compiled library for Microsoft Windows (95/98/ME/NT/2000 or later) is presented as dynamic library FOIL.dll and FOIL.lib files. A program should be linked with the FOIL.lib library. At run time the FOIL.dll should be located either in the current directory or in the Windows system directory.

Compiled library for Philips TriMedia is presented as static library libFOIL.a. A program should be linked with this library.

X. APPENDIX – LIBRARY USAGE EXAMPLE

This example contains a listing of a simple application, with input data and output results. This application will help users to write their own applications.

A. Example description and task definition

The task for the application is:

- To read the existing color image;
- Transform it to grayscale;
- Rotate it;
- Scale it;
- Build the histogram;
- Write the results (image and histogram vector) back.

B. Source code listing

Listing 1 The example main source file:

```
/*
• Name:        Main.cpp
• See also:
• Description: Main module of the FOIL example
• Project:     Fast Object Imaging Library Example
*
• Copyright © 2002 Alacron, Inc.
*
• Created:     20-February-2002 Grigorij Nikodimov V1.00
*
• Revision history:
 *
 */

// Note:
// This example uses the PPM (Portable Picture Map) and PGM (Portable Gray Map) file
   formats
// You need to have the image viewer, which supports these file formats

// --------------------
// Include files section
// --------------------
#include <stdio.h>
#include "../FOIL/Sources/FOIL.hpp"

// Main function
int main
        (
int argc,
char* argv[]
        )
{
// Source image separated by channels
FOIL::CMatrix8 RChannel;
FOIL::CMatrix8 GChannel;
FOIL::CMatrix8 BChannel;
// Histogram vectors
```

```
FOIL::CIntVector Histogram ( 256, 0 );
// Grayscale images (converted from original image)
FOIL::CGrayImage8 Grayscale1;
FOIL::CGrayImage8 Grayscale2;
FOIL::CGrayImage8 Grayscale3;
// Input and output file handles
FILE* hInputFile = NULL;
FILE* hOutputFile = NULL;
// Image dimensions
FOIL::dimension_t SizeX = 0;
FOIL::dimension_t SizeY = 0;
// Miscellaneous variables
char cChar = 0;
FOIL::dimension_t Counter = 0;
FOIL::dimension_t CounterX = 0;
FOIL::dimension_t CounterY = 0;

try
        {
                // ---------------------------------------------
// Read source image file section
                // ---------------------------------------------

// Note: we do not check the existence, format and validity of input file
// Open source image file
hInputFile = fopen ( “input.ppm”,”rb” );

// Skip header
do
                {
cChar = fgetc ( hInputFile );
} while ( cChar != 0x0A );
// Read image dimensions
fscanf ( hInputFile, “%i %i\n”, &SizeX, &SizeY );
// Skip color depth
do
                {
cChar = fgetc ( hInputFile );
} while ( cChar != 0x0A );
// Initialize images with dimensions
RChannel.Initialize ( SizeY, SizeX );
GChannel.Initialize ( SizeY, SizeX );
BChannel.Initialize ( SizeY, SizeX );
Grayscale1.Initialize ( SizeY, SizeX );
Grayscale2.Initialize ( SizeY, SizeX );
Grayscale3.Initialize ( SizeY/2, SizeX/2 );
// Read image data
for ( CounterY=0 ; CounterY<SizeY ; CounterY++ )
                {
for ( CounterX=0 ; CounterX<SizeX ; CounterX++ )
                        {
// B channel
BChannel[ CounterY ][ CounterX ] = fgetc ( hInputFile );
// G channel
GChannel[ CounterY ][ CounterX ] = fgetc ( hInputFile );
// R channel
RChannel[ CounterY ][ CounterX ] = fgetc ( hInputFile );
                        }
                }

// Close source image file
fclose ( hInputFile );
```

```
        // ---------------------
// Process image
        // ---------------------

// Create new planar image (source image, represented as planar RGB)
FOIL::CRGBPlanar8 RGBPlanar ( RChannel, GChannel, BChannel );
// Convert planar image to grayscale
FOIL::ConvertImageFormat ( RGBPlanar,&Grayscale1 );
// Rotate grayscale image
FOIL::Rotate8 ( Grayscale1, 20, &Grayscale2 );
// Zoom grayscale image
FOIL::Zoom8 ( Grayscale2, &Grayscale3 );
// Build histogram on grayscale image
FOIL::Histogram8 ( Grayscale3, &Histogram );

        // ------------------------------------------------------------
// Write the output grayscale image to a file
        // ------------------------------------------------------------

// Note: we do not check the available space and write-protecting flags
// Open destination image file
hOutputFile = fopen ( "output.pgm","wb" );

// Write header
fprintf ( hOutputFile, "P5\n" );

// Write image dimensions
fprintf ( hOutputFile, "%i %i\n", SizeX/2, SizeY/2 );
// Write color depth
fprintf ( hOutputFile, "255\n" );

// Write image data
for ( CounterY=0 ; CounterY<SizeY/2 ; CounterY++ )
        {
for ( CounterX=0 ; CounterX<SizeX/2 ; CounterX++ )
                {
fputc ( Grayscale3[ CounterY ][ CounterX ], hOutputFile );
                }
        }

// Close destination image file
fclose ( hOutputFile );

        // ----------------------------------------
// Write the histogram to a file
        // ----------------------------------------

// Open destination histogram vector file
hOutputFile = fopen ( "output.txt","w" );

// Write histogram vector data
for ( Counter=0 ; Counter<256 ; Counter++ )
        {
fprintf ( hOutputFile, "Histogram[ %i ] = %i\n", Counter, ( unsigned int
)Histogram[ Counter ] );
        }

// Close destination histogram vector file
fclose ( hOutputFile );
        }
// Known exception
catch ( std::exception& x )
        {
```

```
printf (“Exception: %s\n”, x.what() );
exit ( 1 );
        }
// Unknown exception
catch ( ... )
        {
printf ( “Unknown exception\n” );
exit ( 2 );
        }

printf ( “Success\n” );
return ( 0 );
        }
```

Listing 1 The Example main Source File

C. ***Input data***

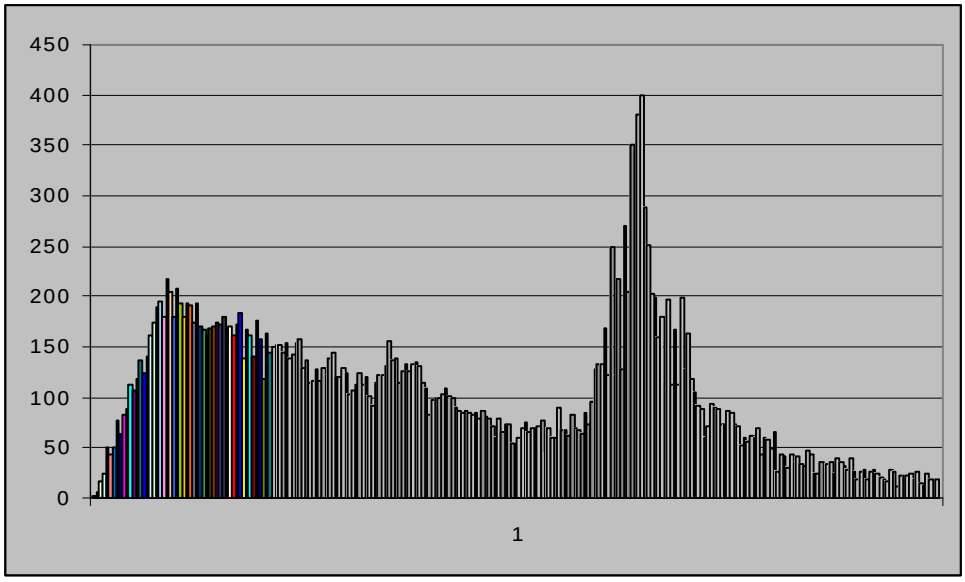
The size of input image is 394x340 pixels, 24bit color depth (RGB, 8-bit per channel)

Picture 1 Represents the source image.

D. ***Execution results***

The size of output image is 197x170 pixels, 8bit grayscale.

Picture 2 Represents the output result image.



Picture 3 Represents the graphical diagram of histogram vector:

H[0] = 1
H[1] = 5
H[2] = 17
H[3] = 25
H[4] = 51
H[5] = 43
H[6] = 50
H[7] = 77
H[8] = 63
H[9] = 83
H[10] = 88
H[11] = 112
H[12] = 107
H[13] = 118
H[14] = 136
H[15] = 124
H[16] = 140
H[17] = 162
H[18] = 175
H[19] = 190
H[20] = 195
H[21] = 180
H[22] = 217
H[23] = 204
H[24] = 180
H[25] = 209
H[26] = 194
H[27] = 180
H[28] = 193
H[29] = 191
H[30] = 175
H[31] = 193
H[32] = 171
H[33] = 167
H[34] = 161
H[35] = 168
H[36] = 171
H[37] = 174
H[38] = 172
H[39] = 180
H[40] = 155
H[41] = 170
H[42] = 161
H[43] = 172
H[44] = 183
H[45] = 139
H[46] = 166
H[47] = 161
H[48] = 141
H[49] = 176
H[50] = 157
H[51] = 118
H[52] = 164
H[53] = 144
H[54] = 150
H[55] = 151
H[56] = 151
H[57] = 144
H[58] = 153
H[59] = 139
H[60] = 143
H[61] = 153
H[62] = 158
H[63] = 130
H[64] = 137
H[65] = 114
H[66] = 116
H[67] = 128
H[68] = 116
H[69] = 129
H[70] = 122
H[71] = 138
H[72] = 145
H[73] = 120
H[74] = 120
H[75] = 130
H[76] = 123
H[77] = 103
H[78] = 106
H[79] = 113
H[80] = 124
H[81] = 112
H[82] = 120
H[83] = 102
H[84] = 91
H[85] = 115
H[86] = 122
H[87] = 121
H[88] = 131
H[89] = 155
H[90] = 136
H[91] = 138
H[92] = 114
H[93] = 126
H[94] = 134
H[95] = 126
H[96] = 133
H[97] = 135
H[98] = 131
H[99] = 115
H[100] = 108
H[101] = 82
H[102] = 97
H[103] = 100
H[104] = 99
H[105] = 104
H[106] = 108
H[107] = 102
H[108] = 99
H[109] = 90
H[110] = 86
H[111] = 84
H[112] = 86
H[113] = 85
H[114] = 82
H[115] = 85
H[116] = 78
H[117] = 86
H[118] = 81
H[119] = 78
H[120] = 71
H[121] = 60
H[122] = 79
H[123] = 66
H[124] = 73
H[125] = 73
H[126] = 55
H[127] = 55
H[128] = 60
H[129] = 70
H[130] = 75
H[131] = 65
H[132] = 70
H[133] = 70
H[134] = 72
H[135] = 77
H[136] = 45
H[137] = 70
H[138] = 60
H[139] = 60
H[140] = 90
H[141] = 68
H[142] = 67
H[143] = 62
H[144] = 82
H[145] = 69
H[146] = 68
H[147] = 63
H[148] = 84
H[149] = 73
H[150] = 95
H[151] = 127
H[152] = 134
H[153] = 133
H[154] = 168
H[155] = 122
H[156] = 249
H[157] = 122
H[158] = 218
H[159] = 128
H[160] = 270
H[161] = 204
H[162] = 351
H[163] = 345
H[164] = 481
H[165] = 400
H[166] = 289
H[167] = 252
H[168] = 202
H[169] = 199
H[170] = 160
H[171] = 180
H[172] = 116
H[173] = 196
H[174] = 112
H[175] = 166
H[176] = 113
H[177] = 198
H[178] = 127
H[179] = 164
H[180] = 119
H[181] = 105
H[182] = 91
H[183] = 88
H[184] = 60
H[185] = 72
H[186] = 93
H[187] = 90
H[188] = 88
H[189] = 74
H[190] = 74
H[191] = 87
H[192] = 84
H[193] = 74
H[194] = 71
H[195] = 52
H[196] = 60
H[197] = 57
H[198] = 62
H[199] = 60
H[200] = 69
H[201] = 43
H[202] = 60
H[203] = 59
H[204] = 49
H[205] = 65
H[206] = 27
H[207] = 43
H[208] = 41
H[209] = 30
H[210] = 43
H[211] = 35
H[212] = 41
H[213] = 34
H[214] = 31
H[215] = 46
H[216] = 44
H[217] = 24
H[218] = 25
H[219] = 36
H[220] = 32
H[221] = 34
H[222] = 35
H[223] = 24
H[224] = 39
H[225] = 35
H[226] = 32
H[227] = 28
H[228] = 39
H[229] = 27
H[230] = 18
H[231] = 27
H[232] = 29
H[233] = 18
H[234] = 27
H[235] = 29
H[236] = 25
H[237] = 20
H[238] = 18
H[239] = 16
H[240] = 28
H[241] = 27
H[242] = 12
H[243] = 23
H[244] = 23
H[245] = 22
H[246] = 25
H[247] = 19

H[248] = 26
H[249] = 15
H[250] = 14
H[251] = 25
H[252] = 18
H[253] = 12
H[254] = 18
H[255] = 0

Listing 2.. Represents the output result for histogram vector

XI. TROUBLESHOOTING

There are several things you can try before you call Alacron Technical Support for help.

_____ *Make sure the computer is plugged in. Make sure the power source is on.*
_____ *Go back over the hardware installation to make sure you didn't miss a page or a section.*
_____ *Go back over the software installation to make sure you have installed all necessary software.*
_____ *Run the Installation User Test to verify correct installation of both hardware and software.*
_____ *Run the user-diagnostics test for your main board to make sure it's working properly.*
_____ *Insert the Alacron CD-ROM and check the various Release Notes to see if there is any information relevant to the problem you are experiencing.*

The release notes are available in the directory: **\usr\alacron\alinfo**

_____ *Compile and run the example programs found in the directory:* ***\usr\alacron\src\examples***
_____ *Find the appropriate section of the Programmer's Guide & Reference or the Library User's Manual for the particular library and problem you are experiencing. Go back over the steps in the guide.*
_____ *Check the programming examples supplied with the runtime software to see if you are using the software according to the examples.*
_____ *Review the return status from functions and any input arguments.*
_____ *Simplify the program as much as possible until you can isolate the problem. Turning off any operations not directly related may help isolate the problem.*
_____ *Finally, first* ***save your original work****. Then remove any extraneous code that doesn't directly contribute to the problem or failure.*

XII. ALACRON TECHNICAL SUPPORT

Alacron offers technical support to any licensed user during the normal business hours of 9 a.m. to 5 p.m. EST. We offer assistance on all aspects of processor board and PMC installation and operation.

A. Contacting Technical Support

To speak with a Technical Support Representative on the telephone, call the number below and ask for Technical Support:

Telephone: **603-891-2750**

If you would rather FAX a written description of the problem, make sure you address the FAX to Technical Support and send it to:

Fax: **603-891-2745**

You can email a description of the problem to ***support@alacron.com***

Before you contact technical support have the following information ready:

_____ *Serial numbers and hardware revision numbers of all of your boards. This information is written on the invoice that was shipped with your products.*
_____ *Also, each board has its serial number and revision number written on either in ink or in bar-code form.*
_____ *The version of the ALRT, ALFAST, or FASTLIB software that you are using.*
_____ *You can find this information in a file in the directory:* ***\usr\alfast\alinfo***
_____ *The type and version of the host operating system, i.e., Windows 98.*
_____ *Note the types and numbers of all your software revisions, daughter card libraries, the application library and the compiler*
_____ *The piece of code that exhibits the problem, if applicable. If you email Alacron the piece of code, our Technical-Support team can try to reproduce the error. It is necessary, though, for all the information listed above to be included, so Technical Support can duplicate your hardware and system environment.*

B. Returning Products For Repair Or Replacement

Our first concern is that you be pleased with your Alacron products.

If, after trying everything you can do yourself, and after contacting Alacron Technical Support, you feel your hardware or software is not functioning properly, you can return the product to Alacron for service or replacement. Service or replacement may be covered by your warranty, depending upon your warranty.

The first step is to call Alacron and request a "Return Materials Authorization" (RMA) number.

This is the number assigned both to your returning product and to all records of your communications with Technical Support. When an Alacron technician receives your returned hardware or software he will match its RMA number to the on-file information you have given us, so he can solve the problem you've cited.

When calling for an RMA number, please have the following information ready:

_____ *Serial numbers and descriptions of product(s) being shipped back*
_____ *A listing including revision numbers for all software, libraries, applications, daughter cards, etc.*
_____ *A clear and detailed description of the problem and when it occurs*
_____ *Exact code that will cause the failure*
_____ *A description of any environmental condition that can cause the problem*

All of this information will be logged into the RMA report so it's there for the technician when your product arrives at Alacron.

Put boards inside their anti-static protective bags. Then pack the product(s) securely in the original shipping materials, if possible, and ship to:

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Clearly mark ***the outside of your package:***

Attention **RMA #80XXX**

Remember to include your return address and the name and number of the person who should be contacted if we have questions.

C. Reporting Bugs

We at Alacron are continually improving our products to ensure the success of your projects. In addition to ongoing improvements, every Alacron product is put through extensive and varied testing. Even so, occasionally situations can come up in the fields that were not encountered during our testing at Alacron.

If you encounter a software or hardware problem or anomaly, please contact us immediately for assistance. If a fix is not available right away, often we can devise a work-around that allows you to move forward with your project while we continue to work on the problem you've encountered.

It is important that we are able to reproduce your error in an isolated test case. You can help if you create a stand-alone code module that is isolated from your application and yet clearly demonstrates the anomaly or flaw.

Describe the error that occurs with the particular code module and email the file to us at:

support@alacron.com

We will compile and run the module to track down the anomaly you've found.

If you do not have Internet access, or if it is inconvenient for you to get to access, copy the code to a disk, describe the error, and mail the disk to Technical Support at the Alacron address below.

If the code is small enough, you can also:

FAX the code module to us at 603-891-2745

If you are faxing the code, write everything large and legibly and remember to include your description of the error.

When you are describing a software problem, include revision numbers of all associated software.

For documentation errors, photocopy the passages in question, mark on the page the number and title of the manual, and either FAX or mail the photocopy to Alacron.

Remember to include the name and telephone number of the person we should contact if we have questions.

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
FAX: 603-891-2745

Web site
http://www.alacron.com/

Electronic Mail
sales@alacron.com, or support@alacron.com