Project Report
ATC-283

CSKETCH Image Processing Library

J. Morgan
S. Troxel

21 August 2002

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

Prepared for the Federal Aviation Administration,
Washington, D.C. 20591

Abstract

The **CSKETCH** image processing library is a collection of C++ classes and global functions which comprise a development environment for meteorological algorithms. The library is best thought of as a 'tool-kit' which contains many standard mathematical and signal processing functions often employed in the analysis of weather radar data. A tutorial-style introduction to the library is given, complete with many examples of class and global function usage. Included is an in-depth look at the main class of the library, the *SKArray* class, which is a templatized and encapsulated class for storing numerical data arrays of one, two, or three dimensions. Following the tutorial is a complete reference for the library which describes all publicly-available class data members and class member functions, as well as all global functions included in the library.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

1. Introduction

The CSKETCH Image Processing Library is a library consisting of C++ classes and functions which together constitute a development environment for signal processing and weather data analysis algorithms. The core class of the library is the *SKArray* class, an encapsulated array class for storing numerical data in rectangular arrays. The data may be one, two, or three dimensional; details are handled internally and are transparent to the user, so for example, calls to global functions have the same syntax regardless of array dimension number and size. In addition, the class has been templatized, using C++ 'template' methods, to minimize the lines of code needed for the implementation. For example, when writing a new global function, only one 'templatized' version need be written; the user then need only include prototypes for the versions of the function (short, double, etc.) that are explicitly needed. Also, C++ class methods have been widely used in the *SKArray* class to make manipulation of *SKArrays* almost as easy as manipulation of standard numerical data types (float, short, etc.) As an example, *SKArrays* A and B can be added and stored in *SKArray* C simply by coding $C = A + B$.

In addition to the *SKArray* class, the core of the CSKETCH library consists of a set of global functions for operating on *SKArrays*. The most widely-used functionality is that of functional template correlation (FTC), which is the main method used by various weather algorithms (MIGFA, AMDA, etc.) to identify likely gust fronts, microbursts, etc. Besides FTC, the core functions include mathematical morphology functions (Erosion, Dilation, Closure, and Opening, for both gray-scale and binary images). A set of standard matrix operations (singular value decomposition, LU decomposition, matrix inversion, etc.) is included, as well as many more general-purpose signal processing operations. They are all described in this document.

Besides the *SKArray* class, a number of other helper classes are included in the library. One of the most useful is the *SKResamp* class which can be used to resample polar format data into Cartesian format. There is an *SKArrayPad* class which can be used to create margined images, to minimize edge effects in various signal-processing operations. The *SKRegionInfo* class can be used to compile statistics (area, approximate length, center of gravity, etc.) of various subregions within an *SKArray's* data buffer. All component classes of the CSKETCH library are also described in this document.

The document begins with a tutorial-style introduction to the CSKETCH image processing library. The tutorial begins with some simple usage examples for the major CSKETCH library classes, namely the array class, the functional template class, the resampler class, and the region analysis class. These examples are meant to show the ease with which multi-dimensional numerical arrays of data can be created, manipulated, and analyzed within the CSKETCH framework. Moreover, these examples provide quick reference points for programmers just starting out in the CSKETCH environment. Following the class usage examples are some examples of global function packages for the CSKETCH library. Finally, since the *SKArray* class is so widely used, the tutorial concludes with an in-depth look at the inner working of the *SKArray* class. This information will prove useful to the experienced CSKETCH user who is looking to optimize algorithm performance through the most efficient manipulation of *SKArrays*.

The remainder of the document is a reference-style compendium of all classes and global functions in the CSKETCH library. For small classes, any relevant global functions are described inside the class documentation. An example of such a class would be the *SKChain* class which is used for encoding gust front chains for MIGFA and the zero-crossing line for AMDA. For large classes, such as the *SKArray* class, the global functions are described later in a separate section. In addition, the global function section contains many useful functions which don't have an association to any of the component classes. Examples of this type of function would be the analytic geometry functions for finding the distance between two points, and for converting a vector from (u,v) to (range, theta) format (and vice-versa). The global functions have been converting a vector from (u,v) to (range, theta) format (and vice-versa). The global functions have been grouped according to general purpose, e.g. mathematical morphology, analytic geometry, fuzzy logic, etc.

2. CSKETCH Image Processing Library Tutorial

2.1 Introduction

The **CSKETCH** Image Processing Library is a collection of C++ classes and functions which together comprise an object-oriented development environment for image-based algorithms. **CSKETCH** classes include an encapulated array type useful for storing rectangular arrays (up to 3-dimensional) of numerical data of any data type (short, float, double, etc); a functional template class for performing functional template correlation operations; a resampler class for converting data from polar to cartesian format, or converting cartesian data at one resolution to a different resolution; a 'region' class for computing statistics about distinct regions of data lying inside a rectangular array of data; and more. Global functions in the library exist for many frequently-performed image and matrix operations, such as image dilation and erosion; median filtering; numerical differentiation; image statistics such as mean and standard deviation; matrix inversion; singular value decomposition and 'LU' decomposition for matrices; and more. Additionally the library contains many simple functions for standard operations of analytic geometry, fuzzy set theory, basic array arithmetic, etc.

The **CSKETCH** Library serves as a 'toolkit' for development of meteorological algorithms. A number of Lincoln algorithms have been built in C++ atop this toolkit, in particular the Machine Intelligent Gust Front Algorithm (MIGFA) and the Automated Microburst Detection Algorithm (AMDA). Typically an application relies heavily on calls to the **CSKETCH** Library and uses class objects from the library, while defining new classes of objects and new functions specific to that particular application. However, the same coding style conventions have been used for application-specific software, which allows for very clean, uniform code appearance.

2.2 Class Usage Examples

We will introduce the library via a set of code samples which illustrate the more common usages of **CSKETCH** library classes and global functions. We begin by demonstrating the ease with which images may be processed and manipulated within the **CSKETCH** environment, using the *SKArray* class.

2.2.1 SKArray Class Usage

The major definitions needed for using the *SKArray* class are found in the file *skarray.h.* Following is a short program which illustrates many of the most frequently used *SKArray* class member functions.

```
#include <skarray.h>
void main( void )
{
// Create a 3-by-3 array to hold floating-point data and set all its values to 5.0
SKArray<float> A( 3, 3 );
A.SetAllSliceValsTo( 5.0 );
// Create another array B and initialize its data buffer to be copy of A's data buffer
SKArray<float> B(3, 3);
B.CopyFrom( A );

// Compute the sum of arrays A and B and store the result in array C:
SKArray<float> C = A + B;

// Create an array D which stores the same numerical data as C, but in double precision:
SKArray<double> D = SKToDouble( C );

// Create an array E whose data is read from file '/usr/data/array.dat')
SKArray<float> E = SKArrayReadFromSketchFile( "/usr/data/array.dat");

// Now we wish to perform various signal processing operations on array E.
// First create a 5-by-5 kernel to be used for window filtering operations.

SKArray<short> kernel( 5, 5 );
kernel.SetAllSliceValsTo ( 0 );

// Now do a median filter of E, replacing each pixel by the 50th percentile value
// of all pixels within the kernel centered at each pixel.
SKArray<float> F = SKArrayMedianFilter( E, kernel, 0.5 );

// Since E is unchanged by the median filter (the output array F is new and was
// created in the call to SKArrayMedianFilter(), we can repeat this process on
// the original data in array E. This time use the 90th percentile of data:
SKArray<float> G = SKArrayMedianFilter( E, kernel, 0.9 );

// Now do gray-scale erosion and dilation of E. Again E is unchanged by
// these operations as new output arrays are returned each time:
SKArray< float> H = GrayScaleDilate( E, kernel );
SKArray<float> I = GrayScaleErode( E, kernel );

// The operations of Closure (Dilation followed by Erosion) and Opening (Erosion followed by
// Dilation) can be performed directly.
SKArray<float> J = GrayScaleClose( E, kernel );
SKArray<float> K = GrayScaleOpen( E, kernel );

// Compute the mean value and standard deviation of the data values in SKArray K:
float mean = SKArrayComputeMean( J );
float stdDev = SKArrayComputeStdDev( J );
}
```

2.2.2 SKFuncTemplate Class Usage

For users familiar with the image-processing operation known as functional template correlation (hereafter referred to as FTC), here's a simple example of creating and using both single-kernel and two-kernel (tandem) functional template objects:

```
// First specify the kernel, function table, and orientation angles for the template:
static char *kernel[] =
{
   "0 0 0 0 0 0 0",
   "0 1 1 1 1 1 0",
   "0 0 0 0 0 0 0",
   0,
};
```

Note that a value of '9' in any kernel will be replaced by 'NIL' in the call to the *SKFuncTemplate* constructor. This is exactly as was done in the original SKETCH system. Note also the terminating '0' in the last row of kernel data. This '0' is needed so the constructor knows where the end of data for the kernel occurs.

```
static char *funcTable[] =
{
   "(0 144) (33 111) (255 111)",
   "(0 0) (26 0) (27 2) (60 200) (72 128) (255 128)",
   0,
};
```

Again note the terminating '0' above, after the last row of data in the function table.

```
static char *angles = "0 45 90 135" ;

// Now create the functional template. Use the point (3,1) as the center point (rotation point) for the
// kernel.
SKFuncTemplate newTemplate( kernel, 3, 1, funcTable, angles );

// Given an input image (SKArray) called inputImage, create mask, score, and orient arrays the
// same size as inputImage. The call to SKArray class member function 'DupEmpty()' returns a
// new array, the same size as inputImage, but with an uninitialized data block.
SKArray<short> mask = inputImage.DupEmpty();
SKArray<short> score = inputImage.DupEmpty();
SKArray<short> orient = inputImage.DupEmpty();

// Setting mask to 1 at all pixels insures processing over every possible input pixel.
mask.SetAllSliceValsTo( 1 );

// Perform functional template processing at every pixel (processing is not performed where
// the mask has value NIL, or at locations where the rotated kernel would fall off the input image
// boundary; in both cases the score and orient arrays have value NIL at that pixel).
SKFuncTemplateMatch( inputImage, newTemplate, mask, score, orient );

// The array score now holds the pixel-by-pixel scores of the FTC match process, while orient holds
// the corresponding pixel-by-pixel orientation of the best match.

// Now a quick example of creating and applying a tandem (2-kernel) template.
```

```
static char *kernel1[] =
{
   "9 9 9 9 9 9 9 9",
   "9 9 9 9 9 9 9 9",
   "9 0 0 0 0 0 0 9",
   "9 0 1 1 1 1 0 9",
   "9 0 0 0 0 0 0 9",
   "9 9 9 9 9 9 9 9",
   "9 9 9 9 9 9 9 9",
   0,
};

static char *kernel2[] =
{
   "9 9 9 9 9 9 9 9",
   "9 9 9 9 9 9 9 9",
   "9 9 9 9 9 9 9 9",
   "9 9 9 9 2 2 9 9",
   "9 9 9 9 9 9 9 9",
   "9 9 9 9 9 9 9 9",
   "9 9 9 9 9 9 9 9",
   0,
};

static char *funcTable[] =
{
   "(0 192) (20 112) (255 112)",
   "(0 0) (48 0) (64 255) (80 255) (88 128) (255 128)",
   "(0 255) (40 255) (60 0) (230 0) (240 255) (255 255)",
   0,
};

static char *angles = "0 20 40 60 80 100 120 140 160" ;

// Create the tandem template. Use pixel (3,3) as center of rotation for kernels.
SKFuncTemplate tandemTemplate( kernel1, kernel2, 3, 3, funcTable, angles );

// Given (equally-sized) input images inputImage1 and inputImage2, create score, orient, and
// mask arrays of the same size.
SKArray<short> mask = inputImage1.DupEmpty();
SKArray<short> score= inputImage1.DupEmpty();
SKArray<short> orient= inputImage1.DupEmpty();

// Again set the mask to 1 everywhere for processing at all input pixels in range.
mask.SetAllSliceValsTo( 1 );

// Apply the tandem template.
SKFuncTemplateMatch( image1, image2, tmpl, mask, score, orient );

// As before, the score and orient arrays store the pixelwise match scores and orientations from the
// (tandem) FTC match process.
```

2.2.3 SKResamp Class Usage

One of the most frequently used processes in working with weather radar data is the conversion of data from Polar format to Cartesian format. We now give examples of this so-called 'resampling' process using the *SKResamp* class.

The full documentation for the *SKResamp* class can be found later in this document. Since our goal in this section is to provide simple examples of **CSKETCH** library functionality, we will jump right in with an example of creating a resampler and then using it to convert polar data to cartesian data.

```
// Build a resampler (essentially a lookup table) for converting polar data to cartesian.
// Assume 256 radials of data per 360 degree scan, 400 gates of data in a full polar image,
// a polar gate size of 250 meters, cartesian output sizes of 100 by 100 pixels, and finally an
// xsize and ysize of 500 meters for the cartesian output data.
SKResamp resamp( 256, 400, 250, 100, 100, 500, 500);
```

The above resampler will be able to completely fill 100-by-100 cartesian output arrays given the specified polar and cartesian data ranges (i.e. the resampler assumes 400 gates * 250 meters / gate = 100 km of raw polar data, while the cartesian image has extent of 100 pixels * 500 meters / pixel = 50 km in both the x and y-dimensions). The full *SKResamp* class description later in this document discusses e.g. what happens when cartesian data is requested for locations out of range of the supplied polar data.

Now that we have the resampler built, we show how easy it is to build cartesian data from polar input. For this we use the member function *SKResamp::Run()*. Two overloaded versions of this function exist.

```
void SKResamp::Run( short *in[], int naz, int ngates, SKArray<short>& out )
void SKResamp::Run( SKArray<float>& in, SKArray<float>& out )
```

The first version of *SKResamp::Run()* takes the input data in a very simple form, namely an array of pointers to (radials of) data. The radials are assumed to be laid out in order, e.g. *in[0]* represents the first radial of data (0 azimuth), *in[1]* represents the next radial of data, etc. This is intended to make it as simple as possible for other applications to take raw input data and resample it to Cartesian format. The output cartesian array out is filled using the lookup table which was generated by the *SKResamp* constructor call.

```
// Example of usage of first form of 'Run()' with the resampler object created above.
SKArray<short> cartesianImage( 100, 100 );
short *radials[ 256 ]; // pointers for the input radials of polar data

// After the 'radials' pointers have been filled e.g. by realtime read of radar data:
resamp.Run( radials, 256, 400, cartesianImage );
```

The second form of 'Run' is provided as a convenience to applications programmers and is generally intended for applications other than raw input (although it can also be used for that purpose). This version takes an *SKArray* object representing a fully initialized polar input array and resamples to an output *SKArray* object representing an output cartesian array. This version is very useful, for example, when one performs a functional template correlation process over a polar image and then wishes to convert the results to cartesian format. It would be inconvenient in this case to store the polar data as an array of radial pointers, as is required for the first version of '*Run()*'.

```
// Example of usage of second form of 'Run()' with the resampler object created above.
SKArray<short> polarImage( 256, 400 );
SKArray<short> cartesianImage( 100, 100 );

// After the 'polarImage' has been filled:
resamp.Run( polarImage, cartesianImage );
```

Of course, for the second version of '*Run()*' to work correctly, the parameters (e.g. number of azimuths, number of gates, gate size, etc.) of the *polarImage* must be consistent with the values which were used in creating the *SKResamp* object, *resamp*.

2.2.4 SKRegion Class Usage

Our final section of **CSKETCH** class usage examples focusses on the *SKRegionInfo* class and the related *SKRegion* structure. These structures are useful for analyzing various regions of data within a (2-dimensional) *SKArray*. A 'region' inside an *SKArray* is defined to be a collection of same-valued pixels inside that array. For example, the set of all pixels with value 1 form region 1 of the image, the set of all pixels with value 2 form region 2 of the image, etc. The region need not be connected, e.g. if the image data is

```
char *arrayData[] =
{
"0 1 0 0 0",
"0 0 0 0 1",
"0 0 0 0 0",
"0 2 2 2 0",
"0 2 2 0 2",
0,
};
```

then the set of all pixels with value 1 form a valid region even though the pixels are not all adjacent. The pixels with value 2 form a (connected) region. The pixels with value 0 are considered 'background' or 'dataless' pixels and region statistics will not be computed for those pixels. In general the *SKRegion* structure will store various attributes (length, area, etc.) of a distinct region inside of an *SKArray*.

For most applications the regions of interest will be be '8-connected', that is, each pixel in a distinct region touches the region on at least one of its 8 pixel neighbors. The definition of 8-connected will become clear after a few simple examples.

Consider a 2-dimensional *SKArray* whose data buffer has the values:

```
char *arrayData1[] =
{
 "10100002",
 "11111002",
 "01000102",
 "11000102",
 "00000002",
 "00000222",
 "00000222",
 "00022222",
 0,
};

SKArray<short> A( arrayData1 );
```

The set of all pixels with value '1' in the above array *A* consitute a distinct 8-connected region in this array (as any pixel with value 1 touches at least one other pixel with value 1 among its 8 horizontal, vertical, and diagonal neighbors). Similarly, the set of all pixels with value '2' consitute a distinct 8-connected region. The set of all pixels with value 0 is not considered a valid region; the **CSKETCH** region analysis code reserves the value 0 for 'background' pixels which are not of general interest.

Given this simple input array, we illustrate several of the region analysis utilities for studying the various regions in the array. There are a number of interesting parameters of a region which one may wish to study (e.g. area, approximate length, centroid, coordinates of a 'bounding box' of a region, etc.) All of the region statistics contained in the *SKRegion* structure are computed simultaneously for all regions in an *SKArray* by function *SKRegionSummary()*:

```
// Compute statistics for all regions in an SKArray A. The call to SKRegionSummary() returns a
// pointer to an SKRegionInfo object which in turn contains an array of SKRegion structures
// (one for each distinct region in the (short integer) input array A).
SKRegionInfo *regionInfo = SKRegionSummary( A);
```

Note that the input array to *SKRegionSummary()* must be of type short integer. Since there were 2 distinct regions inside array A, the returned *SKRegionInfo* object will contain an array of 2 *SKRegion* structures. The various region attributes for regions 1 and 2 can then be directly read from the *SKRegionInfo* array:

```
// Get the area of region 1 (number of pixels in the array with value 1):
int area=regionInfo->region[1].area;

// Get the x-coordinate of the center of gravity of region 2:
float x=regionInfo->region[2].xCenterGravity;

// Get the 'length' of region 1 (the length is taken to be the length of an approximating rectangle).
// The approximating rectangle is found by a least-squares process.
float length=regionInfo->region[1].length;

// The coordinates of a 'bounding box' for a region can be accessed from:
```

```
int xmin=regionInfo->region[1].xmin; int xmax=regionInfo->region[1].xmax;
int ymin=regionInfo->region[1].ymin; int ymax=regionInfo->region[1].ymax;
```

For a full list of the statistics computed and stored by the call to *SKRegionSummary()*, see the description for the *SKRegionInfo* class later in this document.

As a second example, consider a 2-dimensional *SKArray* whose data buffer has the values:

```
char *arrayData2[] =
{
 "1 0 1 0 1",
 "1 0 1 0 1",
 "1 0 1 0 1",
 "1 0 1 0 1",
 "1 0 1 0 1",
 0,
};

SKArray<short> B( arrayData2 );
```

In this array, the set of all pixels with value '1' does not constitute a distinct 8-connected region; for example, all the pixels in the 3rd column are isolated from all the pixels in the 1st column. For some applications, we may not care whether a region is connected or not. Other applications do consider regions to be separate if they are not connected. For this reason, the function *SKLabelRegions()* is provided. Given a short integer input *SKArray B*, *SKLabelRegions()* returns a 'labelled' version of the array, i.e. a new array which has distinct connected regions identified by a distinct integer. The original input array is unchanged. For instance, using the array *B* from above, if we create a new short integer *SKArray* called *labelled* via:

```
SKArray<short> labelled = SKLabelRegions( B );
```

then the output array *labelled* will look like:

```
1 0 2 0 3
1 0 2 0 3
1 0 2 0 3
1 0 2 0 3
1 0 2 0 3
```

That is, the 3 distinct connected regions of array *B* are actually given unique tags in the output array *labelled.* Again the original input array *B* is unchanged. If we now call *SKRegionSummary()* with the *labelled* array as input, we will get back an *SKRegionInfo* object containing an array of 3 *SKRegion* structures, one for each region labelled 1 through 3. By contrast, if we call *SKRegionSummary()* with the original array *B* as input, we will only have 1 *SKRegion* structure contained in the output *SKRegionInfo* object (for the non-connected region consisting of all pixels with value 1). Again, recall that the set of pixels with value 0 is considered 'background' or uninteresting values, so statistics for the set of pixels with value 0 are not computed by *SKRegionSummary()*.

It's worth pointing out here that *SKLabelRegions()* takes an input array and returns a new array where distinct, 8-connected regions are given their own labels. Two pixels belonging to the same region will have the same label in the output array even if they had different values in the input array. For example, consider the array *C* with data array:

```
char *arrayData3[] =
{
 "1 0 6 0 5",
 "2 0 7 0 4",
 "3 0 8 0 3",
 "4 0 9 0 2",
 "5 0 1 0 1",
 0,
};

SKArray<short> C( arrayData3 );
```

Again, the first, third, and fifth columns of this array correspond to distinct, 8-connected regions of nonzero data. Thus, the labelled output resulting from a call to *SKLabelRegions()* will again look like:

```
1 0 2 0 3
1 0 2 0 3
1 0 2 0 3
1 0 2 0 3
1 0 2 0 3
```

The **CSKETCH** library tutorial now continues with descriptions and examples of some of the library's more commonly used global functions.

2.3 Additional CSKETCH Function Packages

CSKETCH library and class member functions are largely grouped into 'packages' of related functions. Such packages include, but are not limited to, analytic geometry functions; image processing functions; 'fuzzy' weighting functions; 2-dimensional matrix functions; and basic mathematical functions. Example usage for a number of the image processing functions (e.g. *GrayScaleErode(), GrayScaleDilate(), SKMedianFilter()*, etc.) was shown in the first code sample in this document. In order to give a bit more more of the flavor of the **CSKETCH** library as a development environment for C++ applications, we will give some examples of the analytic geometry and basic array arithmetic 'packages'. The full set of all **CSKETCH** class member and global functions are described later in this document.

2.3.1 Analytic Geometry Functions

A number of very simple, but frequently used, functions of analytic geometry are encoded in the **CSKETCH** library as C++ global functions. We give examples of a few of them here; the full list and description of functions can be found later in this document.

One of the simplest problems arising in analytic geometry is, given two points *p1 = (x1, y1)* and *p2 = (x2, y2)* in the x-y plane, what is the direction of the vector from *p1* to *p2*. While this question is easy enough to answer, it can be slightly complicated by the choice of angle measuring scheme. For instance, in radar meteorology applications, north is usually taken to be 0 degrees, with angles increasing in the clockwise sense (e.g. east = 90 degrees, south = 180 degrees). However, in the 'mathematical' convention of measuring angles, east is 0 degrees and angles increase in the counterclockwise sense (north = 90 degrees, west = 180 degrees, etc.) For this reason, the **CSKETCH** analytic geometery package allows for the specification of the angle measuring scheme for functions where the angle measuring scheme makes a difference in the computations. A simple example is function *SKDirectionFrom()* which takes as input 2 (2-dimensional) points and an enumerated type indicating the scheme used to measure angles:

```
SKCoordI pt1, pt2;
pt1.x = 0; pt1.y = 0; pt2.x = 1; pt2.y = 0;
// Compute the angle from (0,0) to (1,0), assuming the meteorological angle convention:
float angle = SKDirectionFrom( pt1, pt2, SK_METEO_CONVENTION );
// The value of 'angle' is now 90.0.

// Now recompute the difference assuming the mathematical convention:
float angle = SKDirectionFrom( pt1, pt2, SK_MATH_CONVENTION );
// The value of 'angle' is now 0.0.
```

A useful mnemonic when dealing with 'math' vs. 'meteo' convention angle schemes is:

MATH + METEO = 90.0,

i.e. if we take an angle and measure it both in the mathematical and meteorological conventions, and then add the results, the sum will always come out to 90 degrees (modulo 360, of course).

As mentioned earlier, the **CSKETCH** library provides support for both of these measuring schemes via the enumerated type

```
enum    SKAngleConvention { SK_MATH_CONVENTION, SK_METEO_CONVENTION };
```

in cases where the angle measuring scheme makes a difference. In cases where the angle scheme makes no difference (e.g. when computing the distance between 2 points, a task performed by function *SKDistanceBetween()*) the support is not needed and one need not specify an angle measurement scheme. For example:

```
float distance = SKDistanceBetween( p1, p2 );
```

needs no argument of type *SKAngleConvention*. Of course, the value of *distance* given the two points *p1* and *p2* from above will be 1.0 regardless of the scheme we are using for measuring angles.

A very typical problem in meteorological applications is, given two angles, find the difference between them in the vector sense. For example, a vector of 0 degrees (i.e. north) and a vector of 315 degrees (northwest) have an angle of 45 degrees between them. The function *SKAngleDifference()* is provided to compute this vector angle difference:

```
float difference = SKAngleDifference( 315.0, 0.0 );
// The value of 'difference' is now 45.0.

difference = SKAngleDifference( 185.0, 0.0 );
// The value of 'difference' is now 175.0.
```

In some applications, two directions are known but one (or both) may be 180 degrees ambiguous. For such cases, the function *SKAngle180Difference()* is provided. This function considers both interpretations of both angle measurements and returns the smallest possible difference between the angles. For instance, if an angle measurement of 0 degrees is ambiguous, it could have value 0 or 180. Likewise a measurement of 100 could be 100 or 280 degrees. If we know the measurements to be unambiguous, we would use function *SKAngleDifference()* to compute their difference; if the measurements are ambiguous, we use *SKAngle180Difference()*:

```
float difference = SKAngleDifference( 180.0, 0.0 );
// The value of 'difference' is now 180.0.

difference = SKAngle180Difference( 180.0, 0.0 )
// The value of 'difference' is now 0.0; the minimum difference occurs when the ambiguous 180.0
// is treated as 0 and the ambiguous 0 is treated as 0.
```

The full set of analytic geometry functions, including all overloaded versions of these functions, is described later in this document.

2.3.2 Functions for Basic Array Arithmetic

There are many functions available in the **CSKETCH** library for performing basic array operations on *SKArrays*. Most of these are encoded as global operators; some are *SKArray* class member operators. Regardless of implementation, usage is straightforward and we illustrate some of these operators here.

```
// First create 2 3-by3 arrays A and B; set all values in A to 3.1 and all values in B to 5.9:
// (the concept of 'slice' of an SKArray will be discussed in the next section).
SKArray<float> A( 3, 3 );
SKArray<float> B( 3, 3 );
A.SetAllSliceValsTo( 3.1);
B.SetAllSliceValsTo( 5.9 );
```

```
// Now create new arrays C, D, and E to be respectively the sum, difference, and
// elementwise product of A and B:
SKArray<float> C = A + B;
SKArray<float> D = A - B;
SKArray<float> E = A * B;

//All elements in C have value 9.0; all elements of D have value -2.8; all elements of E have value 18.29.
```

There are overloaded versions of operators +, -, and * to add, subtract, or multiply all elements of an array by a single scalar. All elements can also be divided by a single scalar using operator /:

```
SKArray<float> F = A + (float) 1.3; //All elements in F have value 4.4.
SKArray<float> G = A - (float) 1.3 //All elements in G have value 1.8.
SKArray<float> H = A * (float) 1.3; //All elements in H have value 4.03.
SKArray<float> I = A / (float) 1.3; //All elements in I have value 2.3846154.
```

There are also a full set of assignment operators for *SKArrays* (=, +=, -=, *=, and /=). The full list of such operators and global functions for *SKArray* manipulation is given later in this document.

We now turn to the final section of this tutorial-style introduction to the **CSKETCH** image processing library -- an in-depth look at the **SKArray** class.

2.4 A Detailed look at the SKArray class

Since the *SKArray* class is by far the most widely-used class in the **CSKETCH** image processing library, the tutorial section of this document continues with an in-depth examination of this class. While some of this information may not be strictly needed by the casual user, this section provides details on many of the 'inner workings' of the class and can be useful e.g. in optimizing memory and computational performance for the class. It also allows for a much better understanding of what occurs 'behind the scenes' when manipulating *SKArray* objects.

The in-depth look at the *SKArray* class begins with a few notes on the layout of the associated data array of an *SKArray* object. This is followed by a discussion of the concept of 'nil' or 'missing' data values. With this background, we then delve into examples of the many types of constructors which can be used to create *SKArrays*. Next, we give some examples of class member usage. Then we turn to the inner workings of the class and discuss the topic of data block ownership for *SKArrays*, as well as the concept of 'deep' vs. 'shallow' copies, and how that concept applies to *SKArrays*. Given that background, we continue with a discussion of the many types of 'copy' operations provided for *SKArrays*. Finally, we introduce the concept of a 'slice' or 'data view' of an *SKArray*, and differentiate 'parent' vs. 'current' slice of an *SKArray*. The in-depth look at the SKArray class then concludes with a discussion of the creation and manipulation of 'padded' *SKArrays*, which are made possible via manipulation of distinct parent and current slices of an *SKArray*.

2.4.1 SKArray Class Overview

The *SKArray* class is the most widely-used class in the **CSKETCH** image processing library. The class is an 'encapsulated' array class for storing rectangular arrays of numerical data. The class has been templatized using C++ template capabilities, so the associated data array of an SKArray object can store any type of numerical data (short, int, float, double). Data arrays can be 1, 2, or 3-dimensional. In addition to the numeric data, the SKArray object contains information about the data, such as a time stamp, latitude and longitude location of the data, orientation of the data relative to a radar, etc. See the *SKArray* class description in the reference section of this document for a full listing of the class member variables and member functions.

2.4.2 SKArray Data Layout

The numerical data array of an *SKArray* is considered to be stored in a 'bottom-up' format; that is, the lower left corner of the array is considered to be the origin of the data. In image space this origin pixel has coordinates (x, y) = (0, 0). Row indices increase in the positive 'y' direction, e.g. the first row has y-coordinate 0, the second has y-coordinate 1, etc. This is in opposition to the LISP-based SKETCH system which has a 'top-down' format (origin at upper left pixel, row indices increasing in the 'downwards' y-direction). The 'bottom-up' format was chosen since it is the most common data format used in Lincoln meteorological algorithms.

2.4.3 Concept of 'nil' or 'missing' data values

The **CSKETCH** image processing library (and hence the *SKArray* class) supports the concept of 'missing' or 'bad' values. Referred to as 'nil' values, these missing values are set equal to a special value. Numerically, this key value is the most negative valid number for the type of the numerical data stored by the *SKArray* object. For instance, 'nil' for an *SKArray* of type float is the most negative valid floating-point number, 'nil' for an *SKArray* of type short is the most negative valid short integer, etc. Typically, 'nil' values are ignored in computations, or the presence of a 'nil' term in a mathematical expression renders the whole term 'nil' (for instance, the difference or sum or product of any valid number with a 'nil' is 'nil'). In functional template correlation (FTC) processes, 'nil' values of the input are ignored, i.e. they contribute nothing to the output FTC results. Also, for FTC purposes, 'nil' values in the processing mask indicate that no FTC matching should occur at that location.

For coding purposes, the value of nil for a given array type can be accessed either directly from the class definition or from a particular instance of the class:

```
// Access value of floating-point 'nil' via the class definition:
float floatNil = SKArray<float>::SK_NIL;

// Access value of floating-point 'nil' via a particular float SKArray:
SKArray<float> floatArray;
float floatNil = floatArray.SK_NIL;
```

Either of these methods should be used whenever the numerical value of 'nil' is needed, rather than hardcoding the value. This allows for much easier porting to machines with different storage formats for float, double, etc. data.

2.4.4 Constructor Examples

Before continuing with additional concepts and contents of the *SKArray* class, it will be useful to illustrate some of the class constructors. This will aid in gaining familiarity with C++ constructor syntax for those not experienced in C++ coding, and will quickly show some of the construction methods available for C++ users who want to begin experimenting with the **CSKETCH** library immediately.

2.4.4.1 Constructing arrays using the default constructor.

Arrays created by the default constructor are of size 1 by 1 by 1, so will likely need to be resized (either by the user or by some subsequent function call). The default constructor is useful for declaring an array whose size is not known at compile time:

```
// Create an SKArray which stores float data, using the default constructor.
SKArray<float> floatArray;

// Create a 1 by 1 by 1 SKArray which stores short data, using the default constructor.
SKArray<short> shortArray;
```

2.4.4.2 Constructing arrays of known size.

It is possible to create *SKArray* class objects with specified sizes for the associated data buffer.

```
// Create a '3D' SKArray of size 20 by 30 by 19 which stores float data.
SKArray<float> floatArray( 20, 30, 19);

// Create a '2D' array of size 20 by 30 with integer data. Array is really 3D with
// the default 'z' size of 1.
SKArray<int> intArray( 20, 30 );

// Create a '1D' array of size 20, with double-precision data. Array is really 3D
// with the default 'y' and 'z' sizes of 1.
SKArray<int> intArray( 20 );
```

Note that the arguments for this constructor need not be hardcoded numbers. For instance, if the variables 'xsize' and 'ysize' have values 15 and 20, the code segment:

```
SKArray<float> floatArray( xsize, ysize );
```

is valid code and will create a float *SKArray* of size 15 by 20.

2.4.4.3 Constructing arrays using the copy constructor.

Often it is desired to create an array of the same size and data type of a preexisting array. This is easily done with the copy constructor:

```
// Construct a floating-point SKArray of size 20 by 30 (and default z size of 1)
SKArray<float> floatArray1( 20, 30 );

// Construct another float array, same size as 'floatArray1'
SKArray<float> floatArray2( floatArray1 );
```

2.4.4.4 Constructing arrays using character data.

For construction using character data, strings of 'nil' in the input char data will automatically be converted to the correct numerical value of 'nil' for the desired type of data. Layout of the data array matches the character representation; for example, if an array is created using the character data below, then after initialization the pixel at (x, y) = (0,0) has value 2, the pixel at (x, y) = (0,1) has value 4, the pixel at (2,2) has value 'nil', etc.

```
char *arrayData =
{
"3  nil nil",
"4  -6  6",
"2  -3  7",
0, // Terminator
};

// Create a 3-by-3 array with short data. Strings of 'nil' automatically replaced
// with SKArray<short>::SK_NIL.
SKArray<short> shortArray( arrayData );

// Create a 3-by-3 array, same values as 'shortArray', but with float data.
// Strings of 'nil' automatically replaced with SKArray<float>::SK_NIL.
SKArray<float> floatArray( arrayData );
```

2.4.5 Simple Member Function Examples

The tutorial continues with some simple examples of member function usage. These examples are merely meant to demonstrate the syntax for C++ class member function calls. More complex examples will require further description of the inner workings of the *SKArray* class, which follows later.

```
// First construct a 3-by-3 SKArray with short data, using the character data
// 'arrayData' used in previous examples.
SKArray<short> shortArray( arrayData );

// Absolute value member function replaces all data elements of shortArray with
// their absolute values. Nil values are unchanged by the absolute value member
// function. Note member function calling syntax is similar to the syntax used for
// accessing structure data variables in C structures.
shortArray.Abs();
```

```
// 'Binarize' member function; values greater than or equal to the supplied
// threshold are set to 1; lesser values are set to 0. Nil elements will be set to 0
// also by this operation.
shortArray.Binarize( (short) 3 );

// Note in the above example that the literal constant '3' must be cast to the type
// of the array for which 'Binarize' is being called. If the value '3' were stored in
// some variable of type short, the cast would not be needed:
short threshold = 3;
shortArray.Binarize( threshold );
```

2.4.6 Data Block Ownership

Before continuing with the next set of examples, it will be useful to understand the concept of data block ownership within the *SKArray* class. This will make the examples of assignment operators and various 'Copy' operations more meaningful. It will also provide insight into C++ destructors for those new to C++.

When a new *SKArray* object is created, it does not necessarily own its own data block. An example of this is afforded by the assignment operator '=':

```
// This array, built by a constructor, owns its own data block.
SKArray<int> intArray1( 10, 10 );

// This array, initialized by the assignment operator, does not own its own block.
SKArray<int> intArray2 = intArray1;
```

The *SKArray* assignment operator creates a new *SKArray* object whose various member variables are initialized to those of the right-hand side array (*intArray1* in this case). However, the data pointer for *intArray2* is in fact set equal to that of *intArray1*; they both point to the same data block. In C++ parlance, the assignment operator is a 'shallow' copy operator. That is, only the basic structure itself is copied, not any additional allocated memory within the right-hand side array.

Since both arrays point to the same block of data, we must be careful not to destroy the data block if only one (but not both) of the arrays which reference this block are destroyed (e.g. go out of scope). This is accomplished via an internal counter for the data block structure; the data block knows how many references to it are being made, and the block will only actually be freed by the *SKArray* destructor, *SKArray::~SKArray()* when this count hits zero (e.g. no *SKArrays* are currently referencing the block). This is entirely transparent to the user, who need not keep track of this information; however knowledge of the block-sharing mechanism is useful in the examples that follow later.

One important effect of this 'sharing' of data blocks means that editing *intArray2*'s data values will also change the corresponding entries in *intArray1* (since these entries are in fact the same memory locations). If one wishes to modify a copy of *intArray1* without changing the data values of *intArray2*, a 'deep' copy operation should be employed, rather than a 'shallow' copy operator. Deep copy methods are discussed in the next section.

2.4.7 'Deep' vs. 'Shallow' copies

The concept of a 'shallow' copy was described in the preceding section. Again, for an *SKArray* object, a shallow copy means a structure copy only; no duplicate data block is created.

A copy operation in which not only the structure is copied, but also any additional allocated data, is called a 'deep' copy operation. In the case of *SKArray* objects, this means a new *SKArray* object is created and assigned to, and the copy array also gets its own data block. The data block of the copy array is initialized from that of the source array in a deep copy operation. Examples of both types of copies follow in the next section. For now, it's worth repeating that one should always create a 'deep' copy of an array when one wishes to edit a copy of an array without modifying the original; since the deep copy has its own data block, one can edit its data values without editing the data of the original 'source' array. However, if one does not need to keep the original data array unedited, a shallow copy is better since it is more efficient (there is no need to allocate a large block of memory for the new array, and there is also no need to copy a potentially large amount of numerical data into the new array's data buffer).

2.4.8 Copy Operations

This section describes some methods of making shallow and deep copies of an *SKArray*. Again, deep copies can be edited without changing the data of the source array; however, deep copies require more system overhead since a new data block must be allocated and the data values of the copy must be initialized from the source array. For efficiency, use shallow copies when the copy's data values need not be changed; however use a deep copy when the data values must be modified.

We begin with some examples of shallow copies.

As mentioned earlier, the assignment operator = is a shallow copy operation:

```
// Both shortArray1 and shortArray2 share a common data block.
SKArray<short> shortArray2 = shortArray1;
```

The copy constructor is also a shallow copy operation:

```
// First declare a 10-by-10 integer array. This array owns its own data block.
SKArray< int > intArray1( 10, 10 );

// Declare a new int array via the copy constructor; all of intArray2's member
// variables are initialized using from the corresponding values of intArray1.
// Both arrays share a common data block.
SKArray< int > intArray2( intArray1 );
```

The assignment operator = and the copy constructor are the only shallow copy operations for the SKArray class. The remaining examples show various different methods of creating deep copies.

First, any shallow copy can be made into a deep copy using the *DeepenShallowCopy()* member function:

```
// Declare a new, 10-by-10 float array, which owns its own data block:
SKArray<float> floatArray1( 10, 10 );

// Shallow copy shares floatArray1's data block:
SKArray<float> floatArray2 = floatArray1;

// Now deepen the shallow copy. This member function creates a new data block
// of the proper size for the calling array; thus it no longer shares a data block with
// any other array. The data from the old array's data block will be copied to the
// dew data block as well.
floatArray2.DeepenShallowCopy();
```

The *DeepenShallowCopy()* example was provided just to show how a shallow copy can be made into a deep copy. In general, however, a user will know up front when a copy array should own its own data, and a deep copy should be created in one step using one of several available methods.

The most common way of creating a deep copy is to use the 'DeepCopy' member function.

```
// Create a new, 10-by-10 short array which owns its own data block:
SKArray<short> shortArray1( 10, 10 );

// Create another short array of the same size as shortArray1; the new array
// will own its own data block, and the new array's data will be initialized using
// shortArray1's data block:
SKArray<short> shortArray2;
shortArray2.DeepCopy( shortArray1 );
```

Sometimes one wants a new array which is the same size as some existing array, but whose values need not be initialized to those of the source array. The *DupEmpty()* member function creates a new array such that:

(1) All member variables of the new array are set equal to those of the source array, as in a shallow copy; and

(2) The new array owns its own data block, which is the same size as that of the source array; however the data values are not copied from the source array.

The *DupEmpty()* operation saves the overhead of initializing the data values using the source's data-values; this can be a useful speedup if the source array is large and if new arrays of the same size as the original *SKArray*, but with different data values, are frequently needed. Use *DupEmpty()* if you need an array, the same size as an existing array, but whose data values will be completely different from those of the source array.

```
// Create a new 10-by-10 short array by performing a 'DupEmpty', using
// 'shortArray1' as source array. The new array owns its own data, but the data
// buffer's values are not copied from the source array.
SKArray<short> shortArray3;
shortArray3.DupEmpty( shortArray1 );
```

A final deep copy example will be provided for the case of performing a deep copy with a pad. This will be discussed in a later section.

2.4.9 Current vs. Parent Slice

In order to support the concepts of (1) 'subimages' within an image and (2) padded arrays, the *SKArray* class includes structures for describing 2 different 'slices' or 'views' of an array. The 'parent' member variable describes the actual, full block of memory allocated for an *SKArray's* data buffer. The 'slice_' member variable describes the 'current' view of the array; this latter slice is the one used by all **CSKETCH** library signal processing operations, member functions, etc. unless it is explicitly stated otherwise.

The current 'slice' of the array is specified by the origin, the size, and the steps of the slice. The origin of the slice is simply the (x, y) coordinates of the lower left corner of the slice; the size specifies the size in each dimension. The 'steps' of the slice allow for 'subsampling' of the parent image; for example, if the x-step and y-step of the slice are equal to two, then the current slice effectively 'subsamples' the parent slice by taking only every second pixel in both the x- and y-directions. In most applications, the steps will all equal one, which corresponds to taking a full, contiguous piece of the parent image as a subimage.

This section continues with examples of manipulating *SKArrays* via various '*Slice()*' functions; we then move on to a typical application of parent vs. current slices, namely 'padded' arrays.

2.4.10 Manipulation of SKArrays via the 'Slice'

```
// First create a float SKArray, size 10 by 10. At time of creation of this particular
// array, the parent slice and the current slice are in fact the same.
SKArray<float> floatArray( 10, 10 );
```

We will manipulate various different slices of the array using different overloaded versions of the *SKArray* member function '*Slice()*'. The first version of this function slices the array in a single specified dimension only; it takes the dimension and the size, origin, and step in that dimension as arguments. The other dimensions are left unaffected, e.g. the slice contains the full size and all data in the non-specified dimensions.

```
// Set the current slice to be just the first column of data of the full array.
// This is a slice in the x-direction of size 1, origin 0, and step 1:
SKArray<float> firstColumn = floatArray.Slice( SK_X, 1, 0, 1 );
```

Again, this slice contains the full data of the original array in the y dimension.

We can use this slice to set all the elements in the first column of *floatArray* to the value 3.0:

```
floatArray.SetAllSliceValsTo( 3.0 );
```

As mentioned above, the member function *SetAllSliceValsTo()* operates on the current slice of the array, not the full parent slice.

```
// Now set the current slice to be just the last row of floatArray. This is a slice in
// the y-dimension of size 1, origin 9, and step 1. The slice contains the full
// extent of data in the x-dimension.
SKArray<float> lastRow = floatArray.Slice( SK_Y, 1, 9, 1 );
```

We could now set all values in the last row of floatArray to a fixed value using *SetAllSlicevalsTo()* as we did just above for the first column.

A few final examples of the first version of '*Slice()*':

```
// First 2 columns of data:
SKArray<float> subarray1 = floatArray.Slice( SK_X, 2, 0, 1 );

// 4th, 5th, and 6th rows of data:
SKArray<float) subarray2 = floatArray.Slice( SK_Y, 3, 4, 1 );

// First row of data, but subsampled so only every 3rd pixel is seen:
SKArray<float> subarray3 = floatArray.Slice( SK_Y, 1, 0, 3 );
```

The second version of the '*Slice()*' *SKArray* class member function allows for selection of size, origin, and step in each of the x, y, and z dimensions. The size, origin, and step for the x-dimension must be supplied; additional arguments are the size, origin, and step for the y and z dimensions and are optional. The optional arguments default to values which will return the full extent of data in the y and z dimensions. Of course, this means that if want to specify non-default values for the z dimension, we must also first supply parameters for the y-dimension, even if we want to keep all of the data in the y dimension.

// Now set the current slice to be the lower left quadrant of the parent slice; the
// slice starts at (0, 0), extends to (4,4), and contains all pixels in the lower left
// quadrant since the steps are equal to 1:
SKArray<float> subarray4 = floatArray.Slice(4, 0, 1, 4, 0, 1);

// Take a subarray which is the first 4 columns of data.
SKArray<float> subarray5 = floatArray.Slice(4. 0, 1);

Note that the above slice is equivalent to

SKArray<float> subarray5 = floatArray.Slice(SK_X, 4, 0, 1);

(using the first verison of the slice function).

2.4.11 Constructing 'padded' arrays

Sometimes it is desired to create an array which is really a 'subarray' of a larger array. The extra space around the subarray is referred to as 'padding'. The **CSKETCH** library supports this operation using the *SKArrayPad* class. The *SKArrayPad* class allows for the specification of padding sizes (x, y, and z pad dimensions, with default pad sizes of 0 for each dimension). The pad class also allows for a number of different methods of initializing the padding around the subarray. One example of a filling operation is '**PAD_MIRROR**' which fills the padding in via the 'mirroring' operation from the LISP-based SKETCH system. Another filling operation, which is useful for padding polar arrays, is the '**WRAP_AZ_MIRROR_R**' operation which wraps polar data in the azimuthal direction and mirrors polar data in the radial direction. Yet another pad operation is '**PAD_FILL**' which simply fills in all pad pixels with a supplied fill value. The fill value must be specified in the pad constructor regardless of the fill method chosen, but the fill value will be ignored if a padding operation other than **PAD_FILL** is selected.

We now give some examples of constructing padded arrays. See documentation of the *SKArrayPad* class for the full description of the padding class.

// Create a pad for a float SKArray. The pad is of size 3 by 5 (and default size 0 in the z-dimension).
*// The pad is to be filled in via the **PAD_MIRROR** method (hence the padFill value of 0 is ignored).*
SKArrayPad<float> pad(PAD_MIRROR, 0, 3, 5);

// Now create a padded array using the prescribed pad. The 'slice' of the array
// created will be of size 20 by 30. The parent slice will be of size 26 by 40 (20 by
// 30 plus a pad of 3 on each side, and a pad of 5 on the top and the bottom). The
// pad will be filled in via the 'mirroring' operation when "PadFill" is called.
SKArray<float> paddedArray(pad, 20, 30);

// Initialize data in the subarray by other means, e.g 'CopyFrom()' member function.

// Now fill in the pad via the mirroring operation.

paddedArray.PadFill();

3. Class Descriptions

Description of the CSKETCH library classes begins on the following page.

class SKArray

Name *class SKArray*

Synopsis *#include <skarray.h>*

Hierarchy *WxBase->WxDisplayable->SKArray*

Description The *SKArray* class is the most widely-used class in the *CSKETCH* image processing library. The class is an 'encapsulated' array class for storing rectangular arrays of numerical data. The class has been templatized using C++ template capabilities, so the associated data array of an *SKArray* object can store any type of character or numerical data (char, short, int, float, double). Data arrays can be 1, 2, or 3-dimensional. In addition to the numeric data, the *SKArray* object contains information about the data, such as a time stamp, latitude / longitude location of the data, orientation of the data relative to a radar, etc.

The *SKArray* class inherits publicly from class *WxDisplayable,* which in turn inherits publicly from class *WxBase.*

Example See the *SKArray Class Tutorial* for extensive examples of member and global functions, operators, constructors, etc.

Constants

```
const int SKARR_MAX_DIM = 3;             // Max number of dimensions.
const int SKARR_MAX_NAME_LEN = 64;    // Max length of name.
```

Component Structures

```
struct SKArrayLayout
{
  int size[SKARR_MAX_DIM] ;
  int origin[SKARR_MAX_DIM] ;
  int step[SKARR_MAX_DIM] ;
  int stride[SKARR_MAX_DIM] ;
} ;

struct SKCoordI
{
  int x, y, z;
};

struct SKCoordF
{
  float x, y, z;
};

// Aliases for the above 'point' structures.
typedef SKCoordI SKPointI ;
typedef SKCoordF SKPointF ;
```

```
struct SKRefLoc
{
  SKCoordI arrayPos;
  SKCoordF worldPos;
};

struct SKImageInfo
{
  char  name[SKARR_MAX_NAME_LEN];         // Image Name (e.g. DBDZ)
  char  dataClass[SKARR_MAX_NAME_LEN];    // Image Data Class (DZ, V, ...)
  short  id;                              // Image ID
  LLTime    time;                         // month/day/year/hr/min/secs
  SKCoordSys  coordSys;                   // Cartesian or Polar
  float     binSize[SKARR_MAX_DIM];       // In meters (& deg. for Polar)
  float     orientation;                  // Degrees from magnetic North
  float     scaleFactor;                  // Used to convert int to real
  float     scaleOffset;                  // Used to convert int to real
  SKRefLoc   refLoc;                      // Reference location
  float     latitude;                     // Lat, Long, Alt for world(0,0,0)
  float     longitude;
  float     altitude;
};
```

Finally each SKArray contains as a member variable an instance of the SKArrayPad class to encode padding information (if any padding is desired) for the given SKArray. Consult documentation for class SKArrayPad for detailed description of supported padding operations.

Enumerations

enum SKType { SK_CHAR, SK_SHORT, SK_INT, SK_FLOAT, SK_DOUBLE };
This enumeration is used for dynamic identification of the templatized SKETCH array types.

enum SKPadOp { PAD_NOP, PAD_FILL, PAD_MIRROR, PAD_WRAP_AZ_MIRROR_R, PAD_WRAP_POLAR } ;
This enumeration is used to identify the methods used in filling in the padding (or margin) for padded arrays. Padded arrays are sometimes needed to deal with edge effects in the image processing code). See documentation for the helper class *SKArrayPad* for a full description.

enum SKCoordSys { SK_CARTESIAN, SK_POLAR };
This enumeration is used to indicate whether an array is storing Cartesian or Polar data.

class SKArray

enum SKDim { SK_X, SK_Y, SK_Z };
Simple type to indicate x, y, or z-dimension. For example, the function call '*array.Size(SK_Z)*' returns the z-size of array.

Constructors

SKArray<T> SKArray();
Default constructor. Array will have x,y,z sizes = (1,1,1). Data array (of only one element) is not initialized.

SKArray<T> SKArray(int xSize, int ySize = 1, int zSize = 1);
Create an *SKArray* with supplied x, y, and z sizes. Default sizes are 1 in each dimension. Associated data array will store numerical data of type *T*. Data array is not initialized.

SKArray<T> SKArray(const SKArray& sourceArray);
Copy constructor. All member variables (size, number of dimensions, numerical data type (float, short, etc.)) are copied from *sourceArray*. The new array's data buffer has the same size as the *sourceArray*. Data values are also copied from *sourceArray*'s data buffer.

SKArray<T> SKArray(SKArrayPad<T>& pad, int xSize, int ySize = 1,
int zSize = 1);
Create an *SKArray* with padding as specified in the *pad* argument. See class *SKArrayPad*. Parent array is the 'big' array (size = specified x, y, z size plus the pad size). Constructor returns the 'slice' array which is the 'internal' array with size = specified x, y, z size.

*SKArray<T> SKArray(char *initData[]);*
Create an array with numerical data of type *T*, using character data.

Destructors

~SKArray()
The SKArray destructor deallocates the block of memory used to hold the data. The destructor checks this block of memory to see if there are any other references to it. Only if there are no other references is the block truly deleted. The array 'header' structure is deallocated in all cases.

Type conversion

template< class T >
SKArray<short> SKToShort(SKArray<T>& input);

template< class T >
SKArray<int> SKToInt(SKArray<T>& input);

template< class T >
SKArray<float> SKToFloat(SKArray<T>& input);

template< class T >

SKArray<double> SKToDouble(SKArray<T>& input);

These 4 templatized functions allow for the conversion of an *SKArray* of any numeric type (short, int, float, double) to any other type. *NIL* values of the input array are converted to the *NIL* value of the output type.

Data conversion is accomplished by pixelwise casting to the output type. Thus, for example, calling *SKToInt* with a float array gives an output array where all data values have been truncated. Note that the input array itself is not modified.

Assignment operators

SKArray<T>& SKArray<T>::operator = (const SKArray<T> &rhs)
The array assigment operator performs a shallow copy. All of the right-hand-side array's member variables are copied to the left-hand-side operand, along with the data pointer (but not the data itself). Increases the reference count of the (shared) data block to prevent freeing of data from underneath an array which has not yet gone out of scope. Returns the left-hand-side array whose members have been assigned the values of the right-hand-side array.

SKArray<T>& SKArray<T>::operator += (T val)
Add a scalar value of the same type as the array to each data element in the array. If an input pixel value is *nil* the output is also *nil* at the same pixel. Returns the left-hand-side array whose data elements have been increased by *val.*

SKArray<T>& SKArray<T>::operator -= (T val)
Subtract a scalar value of the same type as the array from each data element in the array. If an input pixel value is *nil* the output is also *nil* at the same pixel. Returns the left-hand-side array whose data elements have been decreased by *val.*

*SKArray<T>& SKArray<T>::operator *= (float val)*
Multiply an array by a floating-point scalar value. If an input pixel value is *nil* the output is also *nil* at the same pixel. Returns the left-hand-side array whose data elements have been multiplied by *val.*

SKArray<T>& SKArray<T>::operator /= (T val)
Divide each element of the array by a scalar value of the same type. If an input pixel value is *nil* the output is also *nil* at the same pixel. Returns the left-hand-side array whose data elements have been divided by *val.*

Indexing operators

T& SKArray<T>::operator () (int i, int j, int k)
Array indexing operator. The operator returns a reference (of type T) to the value at location (x, y, z) = (i, j, k) in the array. This means it can be used as a left hand value or a right hand value.

Logical operators

int SKArray<T>::operator == (SKArray<T> & rhs)
Function to determine array equality (e.g. arrays have same number of dimensions, same slice sizes, and same slice values). Returns 1 if the arrays are equal, 0 if they differ.

int SKArray<T>::operator == (T val)
Function to determine if the array data members are equal to a specified value. Returns 1 if the array data members are equal to val, 0 if they differ.

Other operators

SKArray<T> operator + (SKArray<T> &lhs, SKArray<T> &rhs);
Add two *SKArrays* and return the result in a new *SKArray*. The *lhs* array may be smaller than *rhs* as the output array is sized the same as the *lhs*. The two *SKArrays* passed into this routine are unchanged. When adding, if either element or both elements are *NIL* then the result for that element is *NIL*.

SKArray<T> operator + (SKArray<T> &lhs, T rhs)
Add the scalar value *rhs* to all array elements. The *SKArray* passed into this routine is unchanged. A new array containing the input data values with the scalar added to each data element is returned. When adding, if the *lhs* pixel is *NIL* then the result for that element is *NIL*.

SKArray<T> operator - (SKArray<T> &lhs, SKArray<T> & rhs)
Subtract two *SKArrays* and return the result in a new *SKArray*. Looping is driven by *lhs* size(s). The *lhs* array may be smaller than rhs as the output array is sized the same as the *lhs*. The two *SKArrays* passed into this routine are unchanged. When subtracting, if either element or both elements are *NIL* then the result for that element is *NIL*.

SKArray<T> operator - (SKArray<T> &lhs, T rhs)
Subtract the scalar value *rhs* from all array elements. The *SKArray* passed into this routine is unchanged. A new array containing the input data values with the scalar subtracted from each data element is returned. When subtracting, if the *lhs* pixel is *NIL* then the result for that element is *NIL*.

*SKArray<T> operator * (SKArray<T> &lhs, SKArray<T> &rhs)*
Multiply two *SKArrays* and return the result in a new *SKArray*. The two *SKArrays* passed into this routine are unchanged. When multiplying, if either element or both elements are *NIL* then the result for that element is *NIL*. The *lhs* array may be smaller than *rhs* as the output array is sized the same as the *lhs*.

*SKArray<T> operator * (SKArray<T> &, float rhs)*
Multiply all array elements by a floating point scalar value. If the array value is *NIL*, simply store a *NIL* in the output array. The *SKArray* passed into this routine is unchanged. A new array containing the input data values multiplied by the scalar *rhs* is returned.

SKArray<T> operator / (SKArray<T> &lhs, T rhs)
Divide all array elements by a non zero scalar value. If the array value is *NIL*, simply store a *NIL* in the output array. The *SKArray* passed into this routine is unchanged. A new array containing the input data values divided by the scalar is returned.

SKArray<T> operator & (SKArray<T> &lhs, T rhs)
Performs a bitwise comparison of each pixel in an *SKArray* against a supplied bitmask, returning a copy of the input *SKArray* with original pixel values set to *NIL* wherever the pixel value does not match (in a bitwise AND sense) the bitmask. Otherwise, the original input pixel value is copied to the output. The input *SKArray* passed into this routine is unchanged. When ANDing, if the lhs pixel is *NIL* then the result for that element in the output is *NIL*. Only supports short and int arrays.

ostream& operator << (ostream&lhs, SKArray<T>&rhs)
Overloaded left shift ostream operator to output *SKArray* elements. Outputs data from top down, so that ASCII representation will match up with graphical representation. Returns a reference to the ostream that was passed into this function.

Public member functions

void Abs();
Replace all pixel values in the current array (except *NIL* values) with their absolute values.

void AdvectImage(SKArray<float>& xvec, SKArray<float>&yvec);
Advect the current array using two supplied *SKArrays*. The *xvec* array indicates (in a pixelwise sense) the amount to advect in the x direction, and the *yvec* array indicates (in a pixelwise sense) the amount to advect in the y direction. The value at each pixel in the advected image is computed as follows. Given the coordinates (x, y) of a pixel in the output image, the amounts to advect in the x and y directions are, respectively, *xvec(x, y)* and *yvec(x, y)*. Compute new coordinates *xadv = x - xvec(x, y)* and *yadv = y - yvec(x, y)*. The value of the advected image at pixel *(x, y)* is set equal to the value of the original image at *(xadv, yadv)*. This method of advection ('backwards' advection, e.g. given a pixel in the current time, find which pixel it most likely came from in the prior time) insures that the advected image is completely filled in. So-called 'forward' advection (moving pixels in the 'prior' image forward by amounts indicated in the xvec and yvec arrays) can produce 'holes' in the output image as not necessarily every pixel in the output will be advected to by some prior pixel. Thus the 'backwards' advection is preferable in most cases.

Note that no new *SKArrays* are returned by this function. Rather, the current array's (unadvected) data buffer is replaced with a buffer of advected data values.

*SKArray Apply(T (*function)(SKArray<T>&, T *dPtr, void *arg),*

*void *arg) ;*
*SKArray Apply(T (*function)(SKArray<T>&, T *dPtr, int x, int y, void *arg),*
*void *arg);*

Functions to apply the user-specified function to all elements in the current array slice, returning a new array of the same size. The *Apply* functions are very useful for window-filtering type operations, such as median filtering, which loop over a kernel for every pixel in an input image. With the *Apply* functions, the user need only write the function which does the processing over a single window (e.g. finding the median of a window of data in the case of median filtering). The *Apply* functions automate looping over all pixels of an input image, sequentially applying the user-defined window function for every pixel in the input.

A word about the supplied input function is in order. In the first version of *Apply* the user-supplied function takes as arguments an *SKArray*, a data pointer *dPtr*, and a void * pointer. When *Apply* calls the supplied function at a location (x, y), the *dPtr* will point to the pixel at (x, y) in the original input image. The user-supplied function must be written with this scheme in mind. For example, in the case of median filtering, *dPtr* would likely point to the center pixel of the window over which the median is to be taken (unless, of course, the user wanted to position the window somewhat differently and wrote the code accordingly). Thus the user-supplied median function would have to be written from the point of view that it takes the median over a window centered at the input location handed to it via the *dPtr*. The void* arg input is intended to point to additional arguments (possibly supplied in a C++ structure) to the user-supplied function.

Two versions are supplied in case the user-supplied function depends on the (x, y) location of the pixel currently being processed (e.g. the function may contain branching or conditional logic based on the (x, y) location). The second version of *Apply* allows for (x, y) pixel location arguments in the user-defined function; *Apply* will call the user's supplied function with the appropriate (x, y) values for each pixel automatically.

void Binarize(T threshold) ;
Convert array to binary. Data values less than *threshold* are set to zero; values greater than or equal to *threshold* are set to 1. *NIL* pixels are also set to 0.

void ClipMax(T max);
Values in the array which are greater than *max* are set to *max.*

void ClipMin(T min);
Values in the array which are less than *min* are set to *min.*

void ClipMinMax(T min, T max);

Values in the array which are less than *min* are set to *min*. Values in the array which are greater than *max* are set to *max*.

void CopyImgInfo(SKImageInfo &src);
Member function to set the image data values to those passed in. The current array's *imgInfo* member variable is initialized using values from the supplied *SKImageInfo* object.

void CopyFrom(SKArray<T>&src);
Copy data values of the current array from those in the data buffer of *src*. The *src* and destination arrays must have the same data type. The slice sizes of the *src* and destination arrays must be equal, but their respective parent arrays need not be (e.g. one of the arrays could be padded).

SKArray DeepCopy();
Return a 'full' copy of the array, e.g. an array of the same dimensions, data type, etc. which also owns its own data block the same size as that of the current array. The data buffer of the newly-created array is initialized using the data values in the buffer of the current array.

SKArray DeepCopy(SKArrayPad<T> &pad);
Returns a padded array with the padding as specified by *pad*. The *slice* (and the *parentSlice*) of the returned array have the same number dimensions, data type, etc. as the current array. The slice of the new array has the same size as the slice of the current array; the parent slice sizes of the returned array are equal to the slice sizes plus the *pad* sizes. The slice values of the newly-created array are initialized using the data values in the slice of the current array. Finally, the padding is filled in via the method specified in the *pad* argument. See documentation for the *SKArrayPad* class for a list of supported padding options.

void DeepenShallowCopy(void);
Function to ensure that the array object has its own private copy of the data block it uses. If it doesn't (e.g. it shares the block with other arrays), a new block is created and the data of the common block is copied to the new block. Note that the entire block (parent + slice) is copied, not just the slice.

SKArray<T> DupEmpty(void);
Return a duplicate copy of the current array object without actually copying the data values. The new copy has its own private data block allocated, but its contents are uninitialized.

void GetSKType(SKType &skType);
Function to dynamically determine the instantiated *SKArray* object type. The recognized types are (enumerated type *SKType*):
SK_CHAR, SK_SHORT, SK_INT, SK_FLOAT, and *SK_DOUBLE.*

The input argument *skType* is passed by reference and its value is set to the *SKType* of the *SKArray* object.

void InitImgData(void);
Member function to initialize the image data (member variable *SKArray<T>::imgInfo*) to default values. Used by all of the constructors. See documentation for the *SKImageInfo* class for a full description if the image info class.

*void Load(T *data[], int nrows, int ncols);*
Load array with data from an array of pointers to strings. Each array element can be thought of as a pointer to a row of data values.

void Mirror(void) ;
The padded region of the array is initialized using a 'mirroring' scheme. The mirroring is performed first over rows, then over columns, as follows.

Mirroring scheme for the padded rows is done first. The left most and right most pad members that will be intialized during the column mirroring below are ignored at this point.

```
   -------------------------   P = padded region
---->|  ...| P | P | ....      E = data element
|  --------------------------
| -->|  ...| P | P | ....
| |  -------------------------   <<<<< (image values 'mirrored' across this line)
| --|   ...| E | E | ....
|    -------------------
-----|  |.. | E | E | ....
   -------------------
```

Mirroring scheme for columns to the left and right is accomplished last. Note: The pad elements ignored in the row mirroring will now be initialized by using the value contained in their mirror which happens to be an initialized pad element. (This happens in the corners of the array).

```
-------------------------   P = padded region
| P | P | E | E | ....      E = data element
-------------------
 |  |  ^  ^
 |  |__|  |
 |________|
```

void PadFill(T fillVal);
Fill the padded region with the specified value. The padded region is the region between the parent and the current slice.

SKArray<T> ParentSlice() ;
Return the 'parent' slice of the array (the slice corresponding to the full original dimensions of the current slice plus the pad (if there is a pad).

void RampFill(T val);
Fill the array slice with a ramp of values. Useful for diagnostics. The array is filled over x, then y, then z dimensions. The first pixel is given value *val*, the second 2**val*, the third 3**val*, etc.

SKArray Reverse(SKDim dim);
Function to reverse the order of the elements of the array in a specified dimension. This is accomplished by simply changing strides and steps for the specified dimension. Useful in debugging 'top-down' vs. 'bottom-up' implementations.

SKArray Rotate(float deg, int cx, int cy) ;
Rotate the current (2-D) array by *deg* degrees in the clockwise direction around the center point *(cx,cy)*, producing an output array of the same size. Values of the input array that end up outside the new array boundaries are omitted. Values of the output array that have no corresponding input value are filled with *SK_NIL*. Currently, this routine has not been extensively optimized, as it's only used when constructing functional templates.

template void SetAllSliceValsTo(T scalar) ;
Function to set all values in an array's current slice to the supplied *scalar.*

void SetElementsInInterval(T markVal, T low, T high,
T outlierMarkVal = SKArray<T>::SK_NIL);
Function to set all elements of the array which lie within some interval to a mark value, possibly setting outliers to a different mark value. Specifically, pixels with a value between *low* and *high* (inclusive) are set equal to *markVal.* If *outlierMarkVal* is not *NIL*, then pixels which lie outside the interval from *low* to *high* are set to *outlierMarkVal.* If the outlier mark is *NIL*, then pixels outside the interval are left unchanged. The default value of optional argument *outlierMarkVal* is *NIL.*

Note that the current array object's data buffer is written over with the new values. To create a new output array while leaving the source array intact, use the global version of this function (same name).

*void SetSliceVals (char *initData[]) ;*
Allow the initialization of arrays via character arrays. Useful for initializing everything in an array except the padding. Or, to reuse an array for other tests. See the *SKArray Class Tutorial* for usage examples.

SKArray Slice(SKDim dim, int size, int origin, int step);
SKArray Slice(int size0, int offset0, int step0,
int size1 = 0, int offset1 = 0, int step1 = 0,
int size2 = 0, int offset2 = 0, int step2 = 0);
Functions for altering the current slice (or 'view') of an array. First version allows for altering only a single specified dimension; other dimensions are not altered. The second version allows full 3D specification of the desired slice. The x-parameters (size, offset, and step in the x-dimension for the slice) must be specified; the slice parameters for the other dimensions are optional and default to values which will not change the slice in the y and z dimensions. See the *SKArray Class Tutorial* for usage examples.

float Sum();
Return the sum of all (non-NIL) pixels in the array, returning the result as a float.

void WrapAzMirrorR(void) ;
Fills the pad of a padded array by mirroring in the X-dimension and wrapping in the Y-dimension. This is a useful operation when performing operations on polar arrays, where range is X, and azimuth is Y. 'Wrapping' the data in the Y-dimension allows processing to proceed normally in the region of azimuth wraparound (the north-mark in radar image processing applications). See documentation for the *SKArrayPad* class for description of *SKArray* pads.

void WrapPolar(T fillVal) ;
Pad-fill the array in the X-dimension and wrap it in the Y-dimension. The padding in the X-dimension is filled in with value *fillVal.* This is a useful operation when performing operations on polar arrays, where range is X, and azimuth is Y. 'Wrapping' the data in the Y-dimension allows processing to proceed normally in the region of azimuth wraparound (the north-mark in radar image processing applications). See documentation for the *SKArrayPad* class for description of *SKArray* pads.

See descriptions of classes from which this class inherits (see **Hierarchy**, above) for additional virtual member functions that are defined by this class.

Access functions

float GetBinSize(int dimension);
Return the bin size of the specified array dimension in meters/pixel. The bin size is also known as the resolution (e.g. 231.5 meters/pixel). Use the predefined constants (actually enum type *SKDim*) *SK_X*, *SK_Y*, and *SK_Z* to indicate the desired dimension.

void SetBinSize(int dimension, float value);
Set the bin size of the specified array dimension in meters/pixel to the supplied value. The bin size is also known as the resolution (e.g. 231.5 meters/pixel). Use the predefined constants (actually enum type *SKDim*) *SK_X*, *SK_Y*, and *SK_Z* to indicate the desired dimension.

float GetConfirmingFactor( );
Returns the value of the *confirmingFactor* member variable. This variable is only meaningful for *SKArrays* which are interest images used in feature detection.

void SetConfirmingFactor(float factor);
Set the value of the *confirmingFactor* member variable. This variable is only meaningful for *SKArrays* which are interest images used in feature detection.

void GetCoordSys(SKCoordSys& coordSys);
Get the current coordinate system for the array. Allowed values are *SK_CARTESIAN* and *SK_POLAR* (enumerated type *SKCoordSys*). Input *coordSys* is passed by reference and assigned the current value of the coordinate system for the array.

void SetCoordSys(SKCoordSys coordSys);
Set the current coordinate system for the array. Allowed values are *SK_CARTESIAN* and *SK_POLAR* (enumerated type *SKCoordSys*).

*void GetDataClass(char *s);*
Get the data 'class'. The data class is a qualitative name tag for the type of array data (e.g DZ, V, SN). The CSketch display mechanism uses the data class to lookup the color table in the color map.

*void SetDataClass(char *s);*
Set the data 'class'. The data class is a qualitative name tag for the type of array data (e.g DZ, V, SN). The CSketch display mechanism uses the data class to lookup the color table in the color map.

short DataReady(void);
Return the *dataReadyFlag* of the current array (1 = data ready, 0 = not ready). The term 'data ready' is used to qualify the state of the data contained in the array. A system might initialize many *SKArrays* at one time, but each might be filled at a different time. The member variable *dataReadyFlag* can be used to determine whether or not an array has been loaded with timely data or not.

void SetDataReady(short flag);
Set the *dataReadyFlag* of the current array (1 = data ready, 0 = not ready). The term 'data ready' is used to qualify the state of the data contained in the array. A

system might initialize many *SKArrays* at one time, but each might be filled at a different time. The member variable *dataReadyFlag* can be used to determine whether or not an array has been loaded with timely data or not.

*T *   GetDataPtr(void);*
Return the pointer to the current slice of data. Note that the current slice need not equal the full parent slice, e.g. for padded arrays, so the slice data pointer need not equal the parent data pointer.

float   GetDisconfirmingFactor( );
Returns the value of the *disconfirmingFactor* member variable. This variable is only meaningful for *SKArrays* which are interest images used in feature detection.

void   SetDisconfirmingFactor(float factor);
Set the value of the *disconfirmingFactor* member variable. This variable is only meaningful for SKArrays which are interest images used in feature detection.

int   GetElementSize();
Return the element size of the array (i.e. the size of each data pixel of the array) in bytes.

void   GetGlobalPos(float& latitude, float& longitude, float& altitude);
Get the global position. For now, this routine assumes that the global position corresponds to the position for world coord system (0,0,0). For ASR-9 applications this is sufficient. Mosaic'ed ITWS images may need something more complex. All 3 variables are passed by reference and their values set to those of the current array.

void   SetGlobalPos(float latitude, float longitude, float altitude);
Set the global position. For now, this routine assumes that the global position corresponds to the position for world coord system (0,0,0). For ASR-9 applications this is sufficient. Mosaic'ed ITWS images may need something more complex.

void   GetID(short& id);
Get the ID number of the array. Input argument *id* is passed by reference and its value is set to the id of the current array.

void   SetID(short id);
Set an ID number for the array.

void   GetImageInfo(SKImageInfo &info);
Get the values of the *SKImageInfo* member variable of the current array. The member variable's fields are copied to the supplied info structure. The

SKImageInfo structure contains all of the information necessary to display the array.

void SetImageInfo(SKImageInfo &info);
Set the values of the *SKImageInfo* member variable of the current array. The member variable's fields are copied from the supplied info structure. The *SKImageInfo* structure contains all of the information necessary to display the array.

*void GetName(char *currName);*
Get the name of the array. A name can be associated to an array for convenience when creating displays, etc. It is assumed that currName points to a char buffer which is large enough to hold the name.

*void SetName(char *currName);*
Set the name of the array. A name can be associated to an array for convenience when creating displays, etc.

int GetNumDim(void);
Return the total number of dimensions of the SKArray.

void GetOrientation(float& orientation);
Get the current array orientation. The orientation is the angular offset from true north in degrees. The input argument *orientation* is passed by reference and its value set to the orientation of the current array.

void SetOrientation(float newOrientation);
Set the current array orientation to *newOrientation*. The orientation is the angular offset from true north in degrees.

void GetRefLocation(int& x, int& y, int& z,
float& wx, float& wy, float& wz);
Get the reference location of the array. The reference location is used to establish a relationship between pixel space and world space. The *x, y, z* arguments correspond to the reference point of the array in the *x, y, & z* dimensions. The *wx, wy, wz* arguments correspond to the reference point of the world coordinate system. All 6 variables are passed by reference and their values set to those of the current array.

void SetRefLocation(int x, int y, int z, float wx, float wy, float wz);
Set the reference location of the array. The reference location is used to establish a relationship between pixel space and world space. In WSP applications, where the center of the array most often corresponds to world location (0.0,0.0) (the radar location), the set routine will be called as:

SetRefLocation(xsize/2, ysize/2, 0, 0.0, 0.0, 0.0);

The *x, y, z* arguments give the reference point of the array in the *x, y, & z* dimensions. The *wx, wy, wz* arguments give the reference point of the world coordinate system.

void GetScaling(float& scaleFactor, float& scaleOffset);
Get the current scaling values for the data in the array. The scaling factor and scale offset allow for the conversion of data stored values to 'real-world' values via the equations:

real value = (float) (((data stored)/scaleFactor) - scaleOffset);
*stored Value = NINT(scaleFactor * (real value + scaleOffset));*

void SetScaling(float scaleFactor, float scaleOffset);
Set the current scaling values for the data in the array. The scaling factor and scale offset allow for the conversion of data stored values to 'real-world' values via the equations:

real value = (float) (((data stored)/scaleFactor) - scaleOffset);
*stored Value = NINT(scaleFactor * (real value + scaleOffset));*

int GetStep(int);
Return the step of the requested array dimension. Use the predefined constants (actually they are the enum type *SKDim*) *SK_X, SK_Y,* and *SK_Z* to specify the dimension.

void GetTime(LLTime & t);
Get the time of the array. See documentation for the *LLTime* class for specifics on manipulating time.

void SetTime(LLTime & t);
Set the time of the array. See documentation for the *LLTime* class for specifics on manipulating time.

int Size(int dimension);
Return the size of the requested array dimension. Use the predefined constants (actually they are the enum type *SKDim*) *SK_X, SK_Y,* and *SK_Z* to specify the dimension.

int Stride(int dimension);
Return the stride of the requested array dimension. Use the predefined constants (actually they are the enum type *SKDim*) *SK_X, SK_Y,* and *SK_Z* to specify the dimension.

Public data members

static const T SK_NIL;
Special value indicating a 'bad' or 'missing' pixel. Value corresponds to the most negative number of a given type, e.g. *SK_NIL* for a short *SKArray* is typically -32767.

static const T SK_MAXIMUM;
Value corresponds to the maximum positive number of a given type, e.g. *SK_MAXIMUM* for a short *SKArray* is typically +32768.

static const T SK_MINIMUM;
Value corresponds to the most negative number of a given type, e.g. *SK_MINIMUM* for a short *SKArray* is typically -32767.

Related global functions

Documented in *libskarr.a* library documentation.

See Also

Library skarr.

Document Revision Date

17 July, 2002

class SKArrayPad

Name *class SKArrayPad*

Synopsis *#include <skarray.h>*

Description Helper class used to pad *SKArrays* (e.g. to handle edge effects in certain image processing routines).

SKArrayPad objects are not actual data buffers which pad existing data buffers; rather they are objects which encode information about the padding which is desired for a particular array. *SKArrayPads* are specified by (1) their x, y, and z sizes which indicate the amount of padding in each dimension, (2) an optional 'fill' value and (3) an enumerated type which specifies how the padding should be filled in, if at all. Creation of an actual padded array requires calling an *SKArray* constructor with a pre-constructed *SKArrayPad* object as an argument. When this is done, the *SKArray* constructor builds an *SKArray* whose 'parent' slice is large enough to hold the desired data array plus the padding margins. The *SKArray*'s '*slice*' member variable is set to the area inside the padding.

Once the 'internal' array (the slice) data has been initialized, the correct 'padFill' operation can be called, filling in the pad area according to the methods described later in this section.

Example See the **CSKETCH Image Processing Library Tutorial** for examples of creating, filling, and manipulating padded *SKArray* objects using the *SKArrayPad* class.

Enumerations *enum SKPadOp { PAD_NOP, PAD_FILL, PAD_MIRROR,*
PAD_WRAP_AZ_MIRROR_R, PAD_WRAP_POLAR } ;
This enumeration is used to identify the methods used in filling in the padding (or margin) for padded arrays. The supported methods of pad filling are:

PAD_NOP: No filling operation is performed. The pad values in the padded array are left uninitialized. This operation may be used, for instance, if a kernel of an image processing function extends beyond an image's valid data buffer, but the function never needs to actually read kernel data outside the original image border.

PAD_FILL: The pad pixels of the padded array will be filled with a single value (supplied to the pad at the time of its construction).

PAD_MIRROR: The pad pixels are filled in via the *SKArray* '*Mirror*' member function. See documentation for the *SKArray* class for description of this *SKArray* class member function.

PAD_WRAP_AZ_MIRROR_R: The pad pixels are filled in via the *SKArray* '*WrapAzMirrorR*' member function. See documentation for the *SKArray* class for description of this *SKArray* class member function. This pad operation is only intended to be used for polar data arrays, where the first radial (first row of data in a 2D array) is in fact a neighbor of the last radial (last row of data).

PAD_WRAP_POLAR: The pad pixels are filled in via the *SKArray* '*WrapPolar*' member function. See documentation for the *SKArray* class for description of this *SKArray* class member function. This pad operation is only intended to be used for polar data arrays, where the first radial (first row of data in a 2D array) is in fact a neighbor of the last radial (last row of data).

Constructors

template <class T>
SKArrayPad<T>::SKArrayPad(SKPadOp padOpIn, T fillVal, int pad0 / = 0 */,*
int pad1 / = 0 */, int pad2 /* = 0 */)*
Templatized function to create an *SKArrayPad* object. Padded arrays of type 'T' (e.g. float, double, etc.) require a pad of the same type. The *padOpIn* variable must be set to one of the values of enumerated type *SKPadOp*, which are defined above. The *fillVal* is the value that will be used to fill the padding if the padding operation (*padOpIn*) is '*PAD_FILL* '. The remaining 3 (optional) arguments specify the padding in the x, y, and z-dimensions; they default to zero. These arguments specify the pad on each side of the array in a particular dimension, e.g. a pad size of 3 in the x dimension means a margin of width 3 on the left and right hand sides of an array will be created using this pad.

Destructors

template <class T>
SKArrayPad<T>::~SKArrayPad()
SKArrayPad class destructor. Currently a no-op as *SKArrayPad* contains no dynamically-allocated memory.

See Also

SKArray class member functions *WrapAzMirrorR*(), *WrapPolar*(), *RampFill*(), *PadFill*(), which are employed to perform the various pad-filling options described above.

Document Revision Date

19 August, 1998

class SKChain

Name

class SKChain

Class to store / represent 'generic' thin-line chains. AMDA uses this more generic version to store a zero-crossing line. MIGFA uses more specialized chains for gust front detections and predictions. Note that MIGFA detection chains (class GFChain) and prediction chains (class GFPredChain) inherit from this class.

Synopsis

#include <skchain.h>

Description

Class to store / represent 'generic' thin-line chains. AMDA uses this more generic version to store a zero-crossing line. MIGFA uses more specialized chains for gust front detections and predictions. Note that MIGFA detection chains (class GFChain) and prediction chains (class GFPredChain) inherit from this class.

Component Structures

```
class SKChainPoint
{
LLDListLink link;

public:
  short x, y ;

  // Local orientation of chain (orientation at this pixel).
  short orient;
};
```

Extremely simple 'helper' class for the *SKChain* class. The class contains public data elements *x, y,* and *orient*. The elements *(x, y)* give the location of a chain point. The *orient* gives the local orientation (measured in the meteorological or 'compass' sense) of the chain at the current chain point. The heart of an *SKChain* class instance is an *LLDList* of *SKChainPoints*.

Enumerations

enum SKChainSense { SK_CHAIN_FORWARD, SK_CHAIN_REVERSE };
Enumerated type for indicating whether a particular function should traverse the chain points in the 'forward' (start with first point) or 'reverse' (start with last point) orientation. Useful in MIGFA in some cases to avoid physically reversing the chain nodes. Note that some operations do require physical chain reversal, hence the member function *SKChain::Reverse*() described below.

Constructors

SKChainPoint::SKChainPoint()
Default constructor for the simple chain point class. Typically points are created one at a time as needed and placed on the list of points of an *SKChain* class instance.

SKChain::SKChain()

Default constructor for the *SKChain* class. Creates a new chain with an empty point list, and the chain score initialized to 0.0.

Destructors

~SKChain()
Destructor for the *SKChain* class. Note that each *SKChain* contains an *LLDList* of *SKChainPoints*. The list header only will be automatically deleted by the *LLDList* destructor. It is up to the application to delete all the individual points in the *LLDList* of *SKChainPoints*. This should occur prior to when the *SKChain* instance goes out of scope and is destructed.

Operators

ostream& operator << (ostream &os, SKChain &chain);
Overloaded ostream operator << for the *SKChain* class. Writes the chain to the supplied ostream. Starts by writing out the *(x,y)* coordinates of the first and last points of the chain, as well as the chain's *score*. Then the *(x,y)* coordinates of all points in the chain are written out, in the sequence they appear in the chain.

Public member functions

SKChain::Reverse()
Function to reverse the order of the points in an *SKChain's* point list; also the *start* and *end* member variables (which seperately store the coordinates of the chain start and end points) are swapped.

Public data members

float score;
Interest score for the chain. Often this is the sum of interest scores for all points in the chain, but some applications (e.g. MIGFA) may then modify this score based on other properties of the chain.

SKPointI start;
Structure which houses a pair of integers; for *SKChain* this stores the coordinates of the first point of the chain.

SKPointI end;
Structure which houses a pair of integers; for *SKChain* this stores the coordinates of the last point of the chain.

LLDList ptList;
The actual list of points in the chain (comprised of *SKChainPoint* objects). See also documentation for the *LLDList* class.

Related global functions

```
SKArray<short>  ExtendChains( SKArray<short> &thinned,
                              SKArray<short> &baseInterest,
                              SKArray<short> &baseOrient,
                              short extendThresh, short interestThresh,
                              short angleThresh );
```

Function to extend the chain features within the input array *thinned*. The *thinned* array is assumed to have had all shapes within the image reduced to chains which are only a single pixel wide, e.g by using *SKArray* class member function *SKArray<T>::Thin()*. The *baseInterest* and *baseOrient* images are the same size as *thinned* and are assumed to give pixelwise interest scores and orientations for points which could possibly be part of chain features (e.g. gust fronts for MIGFA and the zero-crossing line for AMDA).

The process is performed roughly as follows. First the endpoints of the chain features are identified. The chains are extended one pixel at a time, extending outwards from the endpoints. At the endpoint currently being processed, we first build a list of candidate points to append onto the end of the chain, using the helper function *BuildExtendSearchWindow()*. A point will be appended onto the end of the chain only if one of the candidate points has an interest score (from *baseInterest*) greater than *interestThresh* and an orientation which lies within *angleThresh* of the orientation of the original endpoint. In the case of multiple acceptable candidates, the one with the highest interest score is chosen. In case of 'ties', the first acceptable candidate point found will be used.

At the end of each round of processing, the new endpoints (all new points which were appended to some chain in the input image) are used as the endpoints for the next round of chain extension. In this way a chain can potentially grow by 1 pixel on each of its ends in each round of processing. The *extendThresh* parameter sets the maximum number of extensions which may be performed; thus each chain can be extended by no longer than 2 * *extendThresh* points.

For further details on the processing performed by this function, see the inline source documentation in file *chainExtend.C*.

The other functions contained in file *chainExtend.C* are helper functions *BuildExtendSearchWindow()* and *InitEndPointOrientations()*. These functions are not intended to be used by general-purpose users of the *CSKETCH* library, so they are not documented here. Detailed documentation of these functions can be found in the source file. Consult this documentation if detailed knowledge of the workings of these functions is desired.

void MarkChainEndAndJunctionPoints(SKArray<short> &thin,
LLDList &endPoints,LLDList &junctionPoints);

Function to take an *SKArray thin* which contains 'thinned' (e.g single-pixel wide) chain features, and mark those chain pixels which are endpoints and junction points of the overall chain 'graph'. The input array is binary, with values of 0 for non chain points and 1 for chain points. On output, non chain points will remain zero, chain junction points will have value 2, chain endpoints will have value 3, and chain 'interior' points will remain at 1.

The input to this array is in binary form, and is assumed to have a 1-pixel pad (zero-filled) around the edge, so no special edge processing is necessary.

In addition to marking the chain end points and junction points in the *thin* array, the input *LLDLists endPoints* and *junctionPoints* will be filled with lists of *SKChainPoint* structures; one such point for each end and junction point.

*void SKChainDeletePoints(SKChain *chain)*
Deletes, one by one, the *SKChainPoint* objects which comprise the *ptList* of the current chain. This is a useful cleanup function which should be called before an *SKChain* object goes out of scope, since the *SKChain* destructor does not delete the individual points in the *ptList* member variable.

void SKFillChainImage(SKArray<short>& chainImage,LLDList& chainList);
Function to take a (previously-allocated) *SKArray* (the *chainImage*) and 'fill' the image with data obtained from a list of *SKChains* (*chainList*). Specifically, if the point (x,y) is in any of the *SKChains* in *chainList*, then the pixel (x,y) in the *chainImage* is set to 1.

*void SKFillChainImage(SKArray<short>& chainImage, SKChain*chain)*
Overloaded version of the above function. In this case the *chainImage* is filled using the points from only a single *SKChain* (the *chain* argument).

See Also

class *LLDList*, class *GFChain*, class *GFPredChain*.

Document Revision Date

22 September, 1998

Name

class SKFuncTemplate

Synopsis

#include <skfunctemp.h>

Description

CSKETCH Library Functional Template Class (FTC) Definition. Functional template correlation is the main engine by which feature detection is accomplished for MIGFA, AMDA, and other meterological algorithms.

Example

See the **CSKETCH Image Processing Library Tutorial** for examples of declaring and using functional template correlation objects.

Constants

#define SKFUNCTEMP_MAX_ORIENT 25
This is the maximum number of kernel rotations for a single template.

#define SKFUNCTEMP_MAX_FUNC 10
This is the maximum number of scoring functions allowed for a single template.

Component Structures

```
struct SKFuncTemplatePoint
{
        int     xOffset ;
        int     yOffset ;
        int     ptrOffset0 ;     // Combination of above for speed
        int     funcIndex0 ;     // For diagnostics - ptr below is used for speed.
        short   *funcLookup0 ;
        int     ptrOffset1 ;     // Second set of info is for second kernel
        int     funcIndex1 ;     // in tandem template.
        short   *funcLookup1 ;
} ;
```

The most basic element of a functional template object is a functional template point. This structure encodes information about a single point of a single rotation for the kernel(s) of the functional template object.

The component fields of the *SKFuncTemplatePoint* are:

xOffset: Stores the x-coordinate of the template point as an offset relative to the pixel currently being processed by FTC. Thus, if *xOffset* for the current template point is -1, the template point is located one pixel to the left of the current processing pixel in the x direction.

yOffset: Stores the y-coordinate of the template point as an offset relative to the pixel currently being processed by FTC. Thus, if *yOffset* for the current template point is 1, the template point is located one pixel to the 'north' of the current processing pixel in the y direction.

ptrOffset0: The above x- and y-offsets for a template point specify an (x, y) pixel location for a template point, relative to the current processing pixel in an FTC operation. For speed, *ptrOffset0* stores a single offset for this pixel relative to the current processing pixel. That is, given a pointer to the current processing pixel, simply adding *ptrOffset0* to that pointer gives the address of the corresponding template pixel within the image being probed. Thus the template pixel can be found in the input image using a single addition rather than 2 adds and 2 multiplies. Note that the pointer offset depends on the x and y sizes of the image being probed (since the x and y strides of the image will differ), so these offsets cannot be computed at construction time. Rather they are computed via a call to *SKFuncTemplate::GenPtrOffsets*() as the first step to the FTC matching process.

See documentation of the *SKArray* class for an explanation of 2-dimensional image strides.

funcIndex0: Any particular functional template can employ multiple scoring functions for different regions within the functional template's kernel. The *funcIndex0* member variable gives the number of the scoring function to be used for this particular functional template point. This is mainly a diagnostic tool useful in determining whether a particular functional template was specified and built correctly; for efficiency, the funcLookup0 pointer, described next, is used to do the actual lookup task when scoring the input image pixel associated with the current template point.

funcLookup0: Points to the correct scoring function to be used for this template point. Recall that for FTC, input image values are expected to be scaled to the range 0 to 255. Each scoring function within the kernel is thus implemented as a 1 by 255 array of score values. The input image pixel is used to index into the scoring function, and the value of the scoring function at the corresponding index is the score for that input value. If there are n scoring functions for the template, the lookup table is an n by 255 table, with row j corresponding to the j'th scoring function. For a particular functional template point, the scoring function never changes; thus e.g. if the current template point uses scoring function 4, then funcLookup0 will be a pointer to the 4th row of the lookup table.

ptrOffset1, funcIndex1, and funcLookup1: These 3 member variables serve exactly the same purpose as *ptrOffset0, funcIndex0,* and *funcLookup0* in the case of a tandem template where 2 different images are probed simultaneously by 2 different kernels. In case the 2nd kernel also has a template point at the same x and y offsets as the first kernel, these member variables will be filled in with the corresponding information for the 2nd kernel and associated scoring function. If the 2nd kernel does not have a point at the same relative offsets, these variables will be set to NULL.

struct SKFuncTemplateOrient

```
{
  int angle ;                // In degrees.
  int nPoints ;
  short nPointsForFunc[ SKFUNCTEMP_MAX_FUNC ];
  SKFuncTemplatePoint *points ;
} ;
```

This structure encodes all information for a single orientation (angle of rotation) of a functional template object. By angle of rotation for a template, we really mean an angle of rotation for the kernel(s) of a functional template. When building a functional template 'orientation', the kernel is rotated, the new x,y offsets of the rotated kernel(s) are computed and scored, and function lookup information for each template point is filled in. In the case of tandem (dual-kernel) functional template objects, a single *SKFuncTemplateOrient* object contains the points for both kernels of the template at that orientation. This is easily done since each *SKFuncTempPoint* contains lookup information for up to 2 kernels.

The component fields of an *SKFuncTemplateOrient* are:

angle: The angle of rotation of this particular orientation of the template (i.e. the angle through which the kernel(s) of the template were rotated when this particular orientation was built).

nPoints: The number of *SKFuncTemplatePoints* associated with this particular rotation of the template. This number need not be the same as the number of points in unrotated kernel(s), due to rounding / truncation of pixel coordinates performed during the process of 'rotating' the kernel(s) through *angle* degrees.

nPointsForFunc: An array indicating, for each scoring function used by the template, how many points within the kernel use that scoring function. This is needed for the highly-optimized version of functional template matching encoded here.

points: An array of *SKFuncTemplatePoints*, the set of all functional template points for the current orientation of the template. The *SKFuncTemplatePoint* structure is described earlier in this section.

Constructors

(Note that the default constructor and copy constructor have been disabled as they are really not relevant for functional template objects).

```
SKFuncTemplate( char *kernelData[], int cx, int cy, char *funcs[],
                char *angles );
```

The *kernelData* input is a 2D character array of scoring function indices for the functional template. An *SKArray* representing the kernel will be constructed using the *SKArray(char **)* constructor (see documentation for the **SKArray** class for further elaboration). See also the CSketch Image Processing Library Tutorial for a description of the format of the 'kernelData' argument.

cx is the X-coordinate of the center of rotation for the kernel, relative to the lower left corner of the kernel. Need not coincide with the actual X-center of the kernel.

cy is the Y-coordinate of the center of rotation for the kernel, relative to the lower left corner of the kernel. Need not coincide with the actual Y-center of the kernel.

funcs is an array of character strings encoding each of the scoring functions indexed by the kernel. See the CSketch Image Processing Library Tutorial for a description of the format of the 'funcs' argument.

angles is a character string 'list' of discrete orientation angles for template rotation. Once again, see the CSKETCH Image Processing Library Tutorial for a description of the format of the 'angles' argument.

*SKFuncTemplate(char *kernelData0[], char *kernelData1[], int cx, int cy,*
*char *funcs[], char *angles);*

Two-kernel version of the constructor described immediately above. All variables are as for that constructor, except an additional argument (the character data for constructing the 2nd kernel) must be supplied. Also, it is worth noting that the *funcs* input encodes information for the scoring functions for both kernels. See the CSKETCH Image Processing Library Tutorial for an example of creating a 2-kernel or so-called 'tandem' functional template object using this constructor.

*SKFuncTemplate(SKArray<int> *kernel, int cx, int cy, char **funcs,*
*char *angles) ;*

Another version of the 1-kernel constructor. This version takes a pointer to an integer *SKArray* which actually stores the kernel for the ftc object. The kernel is assumed to have been filled in with the appropriate scoring function indices. This version is very useful e.g. in the case where the kernel indices are not known at compile time. With this version, the kernel can be created and filled in at run time, then the ftc object can be created with the dynamically-filled kernel.

The other arguments are exactly as in the other 1-kernel constructor (the first constructor described in this section).

*SKFuncTemplate(SKArray<int> *kernel0, SKArray<int> *kernel1, int cx, int cy, char **funcs , char *angles) ;*

Two-kernel version of the constructor described immediately above. All variables are as for that constructor, except an additional argument (the pointer to an integer *SKArray* which represents the 2nd kernel) must be supplied. Also, it is worth noting that the 'funcs' input encodes information for the scoring functions for both kernels. See the CSKETCH Image Processing Library Tutorial for an example of creating a 2-kernel or so-called 'tandem' functional template object using this constructor.

Again, this constructor is useful for building a tandem detector where the kernel sizes and / or indices are not known at compile time.

Destructors

SKFuncTemplate::~SKFuncTemplate()

Deallocates the template object and all associated memory.

Operators

friend ostream& operator << (ostream& os, SKFuncTemplate& t) ;

Print operator for dumping *SKFuncTemplate* information to an output stream. The following are written to the output stream:

The kernel(s) of the template (via operator << as defined for the *SKArray* class).

The list of orientations (rotation angles) for the template.

The individual orientations of each template are then written out. First the angle of the current orientation is written. Then, for each point in the current orientation, the x-offset, y-offset, and scoring function index. In the case of a tandem template, the second scoring function index (used with the second kernel) is also written out.

Finally, the scoring functions (in lookup table form) are printed. Recall each scoring function is essentially a 1 by 255 array. To avoid screen wraparound, only 8 of the 255 entries per scoring function are printed per line.

Public member functions

These member functions, while public, are class 'helper' functions used in functional template construction. Since they are not of use to application programmers, they are not documented here. See the in-line source documentation for detailed function descriptions.

*static int BuildFunctions(char *funcs[], short *funcLookup[]) ;*

*static void FillFuncLookup(char *func, short *funcLookup) ;*
*static char *GetFuncPoint(char *s, short *x, short *y) ;*
*SKArray<int> *CreateWorkKernel(SKArray<int>& kernel, int cx, int cy,*
int& workCenterX, int& workCenterY) ;
*static int OrientAnglesFromString(char *angStr, int *angles) ;*

Related Global Functions

friend void SKFuncTemplateMatch(SKArray<short>& image,
SKFuncTemplate& tmpl, SKArray<short>& mask,
SKArray<short>& scoreArr, SKArray<short>& orientArr) ;

Functional template match procedure for single-kernel templates. Inputs arrays *image*, *mask*, *scoreArr*, and *orientArr* must all have the same size. Functional template correlation is performed for every pixel in the input image which corresponds to a non-nil pixel in the *mask* (with the possible exception of pixels near the edge of image, where processing would cause the functional template kernel to probe out of bounds of image). The output *scoreArr* stores the maximum match score of the correlation process for each pixel processed; the output *orientArr* stores the orientation which produced the maximum match score for each pixel. See the in-line source documentation for more detailed description of the functional template match process.

friend void SKFuncTemplateMatch(SKArray<short>& image0,
SKArray<short>& image1, SKFuncTemplate& tmpl,
SKArray<short>& mask, SKArray<short>& scoreArr,
SKArray<short>& orientArr) ;

Functional template match procedure for two-kernel ('tandem') templates. Inputs arrays *image0*, *image1*, *mask*, *scoreArr*, and *orientArr* must all have the same size. Functional template correlation is performed for every pixel in the input images which correspond to non-nil pixels in the *mask* (with the possible exception of pixels near the edge of the images, where processing would cause the functional template kernel(s) to probe out of bounds of the images). The output *scoreArr* stores the maximum match score of the correlation process for each pixel processed; the output *orientArr* stores the orientation which produced the maximum match score for each pixel. See the in-line source documentation for more detailed description of the tandem functional template match process.

Document Revision Date

3 December, 1998

Name

class SKFuzzyFunc

Synopsis

#include <skfuzzyfunc.h>

Description

Class for representing a 'fuzzy function' as used in functional template correlation, averaging of interest images in AMDA, etc. While the *SKFuncTemplate* class has its own simple representation of a fuzzy function, this class allows for more general fuzzy functions. In the future this class may be used internally by the *SKFuncTemplate* class.

Constants

There are no defined constants for this class. However, by convention for AMDA, the number of 'bins' (the number of allowed x-values for the fuzzy function) defaults to 512. When the fuzzy function does in fact have less than, or equal to, 512 distinct x-bins, the fuzzy function lookup is implemented efficiently by a lookup table. If there are more than 512 bins, the fuzzy function value must be computed directly as each input pixel is encountered. Obviously this is a much less efficient process.

Constructors

template <class XType, class YType>
*SKFuzzyFunc<XType,YType>::SKFuzzyFunc(char *funcStr);*

Doubly-templatized constructor function. The XType is the data type of input values for the fuzzy function. The YType is the data type of output values for the fuzzy function. For example, the function could take short inputs and return float values. The *funcStr* specifies the inflection points and is in the format "(x1 y1) (x2 y2) ... (xn yn)". For example, for a fuzzy function which takes short arguments and returns floats, the string could look like "(3 10.0) (15 1.0) (255 0.0)".

Note that the constructor will build automatically an efficient lookup table for evaluating the fuzzy function if the supplied x-values in *funcStr* range between -255 and +255. Otherwise the function will be directly evaluated for given input values, which is a much slower process.

Destructors

template <class XType, class YType>
SKFuzzyFunc<XType,YType>::~SKFuzzyFunc();

Deletes the *SKFuzzyFunc* object and any associated memory.

Operators

friend ostream & operator << (ostream&, SKFuzzyFunc<XType,YType>&);

Output operator for the *SKFuzzyFunc* class. Just prints out the 'inflection points' of the fuzzy function.

Public member functions

SKArray<YType> Apply(SKArray<XType> &imgIn);

Function to apply the fuzzy function to every pixel in an input array. The returned output array is the same size as the input array. Pixels in the output array are computed by plugging each corresponding pixel from the input array into the fuzzy function.

Document Revision Date

2 December, 1998

class SKRegionInfo

Name

class SKRegionInfo

Synopsis

#include <skregion.h>

Description

Simple container class for storing information about all the 'regions' within an *SKArray*. The information about each particular region is stored in an *SKRegion* structure, described under **Component Structures** below. The *SKRegionInfo* class contains as member variables an array of *SKRegion* structures and the number of such structures in the array. The class also contains a constructor and a destructor.

A 'region' inside an *SKArray* is defined to be a collection of same-valued pixels inside that array. For example, the set of all pixels with value 1 form region 1 of the image, the set of all pixels with value 2 form region 2 of the image, etc. The region need not be connected, i.e. if the image data is

```
0 1 0 0 0
0 0 0 0 1
0 0 0 0 0
0 2 2 2 0
0 2 2 0 2
```

then the set of all pixels with value 1 form a valid region even though the pixels are not all adjacent. The pixels with value 2 form a (connected) region. The pixels with value 0 are considered 'background' or 'dataless' pixels and region statistics will not be computed for those pixels. In general the *SKRegion* structure will store various attributes (length, area, etc.) of a distinct (i.e. labelled) region inside of an *SKArray*.

Example

See the **CSKETCH Image Processing Library Tutorial** for examples of computing statistics for regions of data within an *SKArray*.

Constants

#define SKARR_MAX_REGIONS 7000
The maximum number of regions expected to be seen inside an *SKArray*. This definition is given in the MIGFA description document.

Component Structures

```
struct SKRegion
{
 // The region number.
 short regionNumber;

 // Area of the region (number of pixels in array having value =
 // regionNumber).
 int area;
```

```
  // Approximate length of region (length is taken to be the length of an
  // approximating rectangle).
  float length;

  // Center of gravity (in pixels) of the region.
  float xCenterGravity;
  float yCenterGravity;

  // Sums of x and y (pixel coordinates) for the region.  Also sums of their
  // squares.
  float sumX, sumY, sumXSquared, sumYSquared;

  // Additional intermediate statistics used in various computations.
  float xDev, xDevSquared, crossSum;

  // Coordinates of upper left hand corner and lower right hand corner
  // of a bounding box for the region.
  int xmin, ymin;
  int xmax, ymax;

  // Variance in horizontal and vertical directions.
  float varianceAboutXAxis, varianceAboutYAxis;

  // Properties of the rectangle used to approximate the region.
  float majorVariance, majorSlope, majorIntercept, majorCos;
  float minorVariance, minorSlope, minorIntercept, minorCos;
  float rectLength, rectWidth;
};
```

Constructors

SKRegionInfo::SKRegionInfo(int nRegions);
Constructor which takes as an argument the number of regions inside an *SKArray* object. Thus the returned *SKRegionClass* object will have an array of *SKRegion* structures of size *nRegions*.

Note that there is no default constructor for the *SKRegionInfo* class.

Destructors

~SKRegionInfo()
The *SKRegionInfo* class destructor frees all memory associated with an *SKRegionInfo* object; in particular the array of *SKRegion* structures is freed.

Operators

friend ostream& operator << (ostream&, SKRegionInfo&);
Print operator for dumping *SKRegionInfo* information to an output stream. First the total number of regions is printed out. Then, for each region in the *SKRegion* array, the region number, followed by an abbreviated list of statistics

for that region, are printed out. Currently the full set of statistics is not printed out, as it is a rather long list. If desired, this list can be added to in the future.

Public data members

int nRegions: The number of *SKRegion* structures contained in the array of such structures within the *SKRegionInfo* object.

*SKRegion *region:* The actual array of *SKRegion* structures.

Related global functions

The *SKRegionInfo* class is really a helper class for studying various distinct regions of data within an *SKArray* object. Thus most of the related functions are included in documentation for the CSKETCH Library. In particular, consult the 'Region Analysis' section of this document.

Document Revision Date

19 August, 1998

Name

class SKResamp

Synopsis

#include <skresamp.h>

Description

CSKETCH Library array resampler routines to handle conversion of polar radar data to cartesian. Resampling is effected via a lookup table; each (x, y) coordinate in a cartesian image is assigned a single range, azimuth cell in a polar image.
This is a 'full image' resampler, i.e. one must have a completely filled array of polar data before using the resampler. The resampler builds a full cartesian array given the full polar array.

Example

See the **CSKETCH Image Processing Library Tutorial** for examples of resampling polar *SKArrays* to cartesian *SKArrays* via the *SKResamp* class.

Constants

#define SK_MAX_RESAMP_AZ 512
The maximum number of azimuths supported by this resampler class; i.e. a polar array of data must contain no more than 512 azimuths if it is to be resampled to a cartesian grid using this resampler class.

Component Structures

```
struct SKResampMapCell
{
  short az;
  short gate;
};
```

Simple structure which specifies a pixel in a polar image (e.g. range, theta coordinates in a 2D array of polar data).

Constructors

Note that there is no default constructor for the *SKResamp* class.

```
SKResamp(int nazPer360, int ngates, float gateSize, int xbins, int ybins,
         float xsize, float ysize, float inScale = 1.0 , float inOffset = 0.0,
         float outScale = 1.0, float outOffset = 0.0);
```

Construct an *SKResamp* resampler. The following assumptions about the polar and cartesian data arrays are encoded in the arguments to the constructor:

nAzPer360 specifies the number of azimuths in a full polar array. Thus e.g. for WSP this argument is nominally 256 as there are 256 azimuths per full scan of radar (polar) data.

ngates specifies the total number of range gates in a full polar image.

gateSize specifies the size, in meters, of a single gate of polar data.

xbins specifies the maximum number of bins in the x-dimension of a cartesian array, which is to be built by resampling polar data.

ybins specifies the maximum number of bins in the y-dimension of a cartesian array, which is to be built by resampling polar data.

xsize specifies the x-size, in meters of a single pixel of cartesian data.

ysize specifies the y-size, in meters of a single pixel of cartesian data.

inScale, inOffset, offScale, and *outOffset* handle the case where a rescaling of the data is desired as the resampling is done. In this case, the input (polar) data is assumed to be scaled and offset via *inScale* and *inOffset.* The output (cartesian) data will be scaled and offset via *outScale* and *outOffset.* All 4 of these arguments are optional; the default value is scale=1, offset=0 for both input and output, e.g. no rescaling of the data will take place.

Destructors

~SKResamp()
An *SKResamp* object is destroyed, freeing up all dynamically-allocated memory associated with the object.

Public member functions

The only public member functions associated with this class are various '*Run*' functions for actually performing the polar-to-cartesian resampling. In all cases, '*Run*' fills the output cartesian array by finding, for each (x, y) coordinate in the cartesian array, the corresponding (range, azimuth) cell in the polar data. This is done efficiently via the lookup table built in the *SKResamp* constructor. The corresponding value in the polar array is then copied over to the corresponding slot in the cartesian output array.

The cartesian image to be filled by '*Run*' must have sizes smaller than the *xbins* and *ybins* arguments which were provided to the call to the *SKResamp* constructor. Cartesian pixels which are out of the range covered by the polar data will be set to *nil* (the lookup table provides for this when it is built).

*void SKResamp::Run(short *in[], int naz, int ngates, SKArray<short>& out)*
The first version of *Run* for this class takes the input data in a very simple form, namely an array of pointers to (radials of) data. This is intended to make it as simple as possible for other applications to take raw input data and resample it to Cartesian format. The output cartesian array *out* is filled using the lookup table which was generated by the *SKResamp* constructor call.

void SKResamp::Run(SKArray<short>& in, SKArray<short>& out)
The second form of '*Run*' is provided as a convenience to applications programmers and is generally intended for applications other than raw input (although it

can also be used for that purpose). This version takes a *SKArray* object representing a polar input array, and resamples to an output *SKArray* object representing an output cartesian array. This version is very useful, for example, when one performs a functional template correlation process over a polar image and then wishes to convert the results to cartesian format. It would be inconvenient in this case to store the polar data as an array of radial pointers, as is required for the first version of '*Run*'.

void SKResamp::Run(SKArray<float>& in, SKArray<float>& out)
This is an overloaded version of the second form of '*Run*', used when the input (and hence output) are float, not short, arrays.

See Also

Library skarr.

Document Revision Date

19 August, 1998

class SKStmResamp

Name

class SKStmResamp

Synopsis

#include <skstmresamp.h>

Description

Resampler object for resampling data from one Cartesian data array to another. The 2 arrays may have different sizes and / or data resolutions. It is assumed, however, that the center pixel of each array corresponds to the radar location (hence the center pixels of each array correspond to the same location in real world space). The class is named *SKStmResamp* as it was first written to allow for the resampling of storm motion u,v images to the finer resolution used by AMDA.

Constants

#define SK_MAX_VECTORS 512

This constant gives the maximum (one-dimensional) size of the output image which is to be filled in by resampling the input image. Thus, with a nominal value of 512, the output image can be no larger than 512 by 512.

Component Structures

```
struct SKVecMapCell
{
  short col;
  short row;
};
```

Simple structure which stores the coordinates of a pixel in the input image which correspond to a pixel in the output image, for the resampling process. For each pixel (x,y) in the output image, the *SKStmResamp* object stores a corresponding *SKVecMapCell* object that indicates which pixel in the input corresponds to (x,y) in the output image.

Constructors

SKStmResamp::SKStmResamp(SKArray<float> &in, SKArray<float> &out)

Constructor to set up an *SKStmResamp* object which can be used to resample from Cartesian input *in* to Cartesian output *out*. The *binSize* member variable for each array must be set prior to calling the constructor. The arrays may be of different (x,y) sizes or bin sizes, but again it is assumed that the center pixel of each array corresponds to the radar location. Also both cartesian images are assumed to have origin at the lower left corner.

The *SKStmResamp* object is essentially a lookup table; for each pixel in *out*, the resampler object stores the location of a pixel from *in* which maps to the pixel in *out* using the given resolutions. No averaging or combining of pixels is done; this is a simple, single-pixel resampler. For example, if a 100 by 100 array with resolution 750 m is to be resampled to a 300 by 300 array with resolution 250 m, the resultant output array will consist of a series of 3-by-3 blocks with duplicate data. That is, each pixel from the input image will map to exactly 9 pixels in the output, and no averaging with adjacent pixels from the input is done.

Destructors

~SKStmResamp()
The destructor destroys the lookup table as well as any other memory allocated by the *SKStmResamp* constructor.

Public member functions

void Resample(SKArray<float> &in, SKArray<float>& out);

Function for performing the actual resampling from Cartesian image *in* to Cartesian image *out*. Again, the resampler must have been built using images with the the same sizes, and the same bin sizes, as *in* and *out*, respectively.

See Also

class SKResamp, which resamples polar data to Cartesian data. Class *SKStmResamp* resamples Cartesian data to cartesian data of possibly different image size and resolution.

Document Revision Date

30 November, 1998

4. Analytic Geometry

4.1 Summary

Global functions for performing various analytic geometry calculations (e.g. distance between 2 points, vector difference between 2 angles, conversion of vector from u,v components to range, theta components, etc.) Many of the functions are overloaded and can take 2 point arguments as either a pair of *SKCoord* objects or as a set of 4 floats (*x1, y1*) and (*x2, y2*). Consult individual function descriptions to see which functions are overloaded and which overloaded versions are available.

4.2 Conventions

Throughout the *CSKETCH* analytic geometry functions, frequent use of an enumerated type, the *SKAngle-Convention* type, is made. This enumerated type takes on one of two values, *SK_MATH_CONVENTION* and *SK_METEO_CONVENTION*. The type is used to indicate whether supplied / returned values are considered specified in the mathematical convention (0 degrees = 'east', with angles increasing in the clockwise sense) or in the metorological or 'compass' convention (0 degrees = 'north', with angles increasing in the counterclockwise sense). Any function which depends on the system of angle measurement will always require an argument specifying the convention used for angle convention. As an example, consider the function *SKComponentsToVector*(), with a u-component of 1.0 and a v-component of 0.0. Thus the vector (u,v) represents a vector with direction due east and magnitude 1.0. In the meteorological sense, the vector components are (r, theta) = (1.0, 90.0). In the mathematical sense, the vector components are (r, theta) = (1.0, 0.0). The calling statements would appear as follows:

```
// Meteorological sense:

SKComponentsToVector( 1.0, 90.0, direction, speed, SK_METEO_CONVENTION );


// Mathematical sense:

SKComponentsToVector( 1.0, 0.0, direction, speed, SK_MATH_CONVENTION );
```

4.3 Functions

Analytic Geometry functions begin on the following page.

SKAngle180Difference()

Name

SKAngle180Difference()
Compute the difference between two angles, in the vector sense. The angles are assumed to be 180 degrees ambiguous, e.g. an angle of 10 can be interpreted as 10 degrees or 190 degrees (this situation occurs frequently e.g. in MIGFA). The minimum difference among all angle interpretations is returned.

Synopsis

```
#include <skanalyt.h>

float   SKAngle180Difference( float ang1, float ang2 );
```

Description

Function to compute the difference between two angles, *ang1* and *ang2*, in the vector sense. The angles are assumed to be 180 degrees ambiguous, e.g. an angle of 10 can be interpreted as 10 degrees or 190 degrees (this situation occurs frequently e.g. in MIGFA). This routine considers both interpretations of each supplied angle and returns the minimum (vector) angle difference between them. For example, if the first angle is 10 degrees (interpreted as 10 or 190) and the second angle is 160 (interpreted as 160 or 340), the 180 degree ambiguous angle difference is 30 (interpreting the first angle as 10 and the second as 340, the vector angle difference is 30 degrees).

Returns

The difference in degrees between the supplied, 180-degree ambiguous angles, in the vector sense. The difference is returned as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        float ang1, ang2, angDifference;
        ang1 = 10.0;
        ang2 = 195.0;

        // 180-degree-ambiguous angle difference, in vector sense, will be 5.0
        // degrees.
        angDifference = SKAngleDifference( ang1, ang2 );
}
```

See Also

SKAngleDifference()

Document Revision Date

3 August, 1998

SKAngleDifference()

Name

SKAngleDifference()
Compute the difference between two angles, in the vector sense.

Synopsis

```
#include <skanalyt.h>

float   SKAngleDifference( float ang1, float ang2 );
```

Description

Function to compute the difference between two angles, *ang1* and *ang2*, in the vector sense. For example, if the two angles are 10 and 340 degrees, the angle difference is 30 (i.e. the angle between two vectors, one at 10 degrees and the other at 340 degrees, both based at the origin, is 30 degrees.).

Returns

The difference in degrees between the supplied angles, in the vector sense. The difference is returned as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        float ang1, ang2, angDifference;
        ang1 = 35.5;
        ang2 = 235.5;

        // Angle difference, in vector sense, will be 160 degrees.
        angDifference = SKAngleDifference( ang1, ang2 );
}
```

See Also

SKAngle180Difference()

Document Revision Date

3 August, 1998

SKComponentsToVector)

Name

SKComponentsToVector()
Convert a vector specified by its x- and y-components to a magnitude and direction.

Synopsis

```
#include <skanalyt.h>

void SKComponentsToVector( float xComp, float yComp, float& direction,
                    float& speed, SKAngleConvention angleConvention );
```

Description

Function to take a vector specified by its x- and y-components, *xComp* and *yComp,* and compute the direction and speed of the vector. The inputs *direction* and *speed* are passed by reference and their contents are updated with the direction and speed of the vector. The desired *angleConvention* for measuring the direction must be specified, e.g. the vector (1,0) has direction 90.0 in the meteorological sense but direction 0.0 in the mathematical sense. See the **CSketch Image Processing Library Tutorial** description for details about the *SKAngleConvention* enumerated type.

Returns

Input variables *speed* and *direction,* which were passed by reference, are updated with the speed and direction of the supplied vector with components *xComp* and *yComp.*

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        float xComp, yComp;
        float direction, speed;

        xComp = 5.0;
        yComp = 0.0;

        // Update direction and speed of vector, measured in meteorological
        // sense.
        SKComponentsToVector( xComp, yComp, direction, speed,
                                SK_METEO_CONVENTION );
}
```

See Also

SKVectorToComponents()

Document Revision Date	3 August, 1998

SKDirectionFrom()

Name

SKDirectionFrom()
Return the direction from one specified point to another specified point, e.g. the angle of the vector from point1 to point2.

Synopsis

```
#include <skanalyt.h>

float   SKDirectionFrom( SKCoordI p1, SKCoordI p2,
                SKAngleConvention angleConvention );
float   SKDirectionFrom( float x1, float y1, float x2, float y2,
                SKAngleConvention angleConvention );
```

Description

Functions to return the direction from the first supplied point to the second supplied point. The first overloaded version of this function supplies the two points as two *SKCoordI* structures. The second overloaded version supplies each point via two float arguments (i.e. the first point is *(x1, y1)* and the second point is *(x2, y2))*.

The **SKCoordI** structure is a simple structure storing an x, y, and z-coordinate; the first version of this function considers only the x and y coordinates of each point and computes the direction from p1 to p2 in the x-y plane. Likewise, the second version of this function computes the direction from the point *(x1, y1)* to the point *(x2, y2)* in the x-y plane.

For each version, the final argument is the desired *angleConvention* for measuring the direction. This setting is important, e.g. the vector from (0,0) to (1,0) has direction 90.0 in the meteorological sense but direction 0.0 in the mathematical sense. See the **CSKETCH Image Processing Library Tutorial** description for details about the *SKAngleConvention* enumerated type.

Returns

In all cases, the direction (in degrees) from point1 to point2 is returned as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordI pt1, pt2;
        float x1, y1, x2, y2;
        float dir1, dir2;

        // First overloaded version; direction in mathematical sense.
        pt1.x = 0;  pt1.y = 0;
        pt2.x = 0;  pt2.y = 5;
        dir1 = SKDirectionFrom( pt1, pt2, SK_MATH_CONVENTION );
        // Second overloaded version; direction in meteorologial sense.
        x1 = 0.0;  y1 = 0.0;
        x2 = 0.0;  y2 = 5.0;
        dir2 = SKDirectionFrom( x1, y1, x2, y2, SK_METEO_CONVENTION );
}
```

Document Revision Date

3 August, 1998

SKDirectionMoved()

Name

SKDirectionMoved()
Disambiguate a 180-degree ambiguous angle measurement based on the direction of the vector from one point to a second point.

Synopsis

```
#include <skanalyt.h>

float   SKDirectionMoved( SKCoordl p1, SKCoordl p2, float dir,
                 SKAngleConvention angleConvention );
float   SKDirectionMoved( float x1, float y1, float x2, float y2, float dir,
                 SKAngleConvention angleConvention );
```

Description

Functions to disambiguate a 180-degree ambiguous angle measurement (the input *dir*) based on the direction of the vector from one point to a second point. Specifically, given two points *p1* and *p2* in the x-y plane, we compute the angle of the vector from *p1* to *p2*. If this vector lies within 90 degrees (in the vector sense) of the vector with angle *dir*, then we return the value of *dir* unchanged. If the two vectors differ by more than 90 degrees, we return (*dir* + 180) (mod 360), as (*dir* + 180) mod 360 will lie within 90 degrees of the vector from *p1* to *p2* in this case.

The first overloaded version of this function supplies the two points as two *SKCoordl* structures. The second overloaded version supplies each point via two float arguments (i.e. the first point is *(x1, y1)* and the second point is *(x2, y2)*).

Note that at the time this function is called, *angleConvention* should be set to the correct convention in which *dir* was initially set or computed. See the **CSKETCH** top-level library description for details about the *SKAngleConvention* enumerated type.

Returns

In all cases, the disambiguated direction *dir* (in degrees) is returned as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordl pt1, pt2;
        float x1, y1, x2, y2;
        float dir1, dir2;

        // First overloaded version; direction in mathematical sense.
        // 180 degree ambiguous measurement of 130 degrees (in math sense)
        // will be disambiguated to 310 degrees in this case.
```

```
    pt1.x = 0;  pt1.y = 0;
    pt2.x = 0;  pt2.y = 5;
    dir1 = SKDirectionMoved( pt1, pt2, SK_MATH_CONVENTION );

    // Second overloaded version; direction in meteorologial sense.
    // 180 degree ambiguous measurement of 130 degrees (in meteo sense)
    // will be disambiguated to 130 degrees (i.e. unchanged) in this case.
    x1 = 0.0;  y1 = 0.0;
    x2 = 0.0;  y2 = 5.0;
    dir2 = SKDirectionMoved( x1, y1, x2, y2,
            SK_METEO_CONVENTION );
}
```

Document Revision Date

3 August, 1998

SKDistanceBetween()

Name

SKDistanceBetween()
Functions to compute the distance between 2 points in the x-y plane.

Synopsis

```
#include <skanalyt.h>

float   SKDistanceBetween( SKCoordI p1, SKCoordI p2 );
float   SKDistanceBetween( float x1, float y1, float x2, float y2 );
```

Description

Functions to compute the distance betweend two points p1 and p2 in the x-y plane. The first overloaded version of this function takes the two point arguments as *SKCoordI* structures; the second overloaded version takes the two points specified by their (x, y) coordinates (i.e. the first point has coordinates *(x1, y1)* and the second point has coordinates *(x2, y2))*. Both versions simply compute the distance in the x-y plane between the 2 supplied points and return this distance as a float.

Returns

The distance between the 2 supplied points, as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordI pt1, pt2;
        float x1, y1, x2, y2, distance;

        // First overloaded version.
        pt1.x = 3;  pt1.y = 0;
        pt2.x = 0;  pt2.y = 4;
        distance = SKDistanceBetween( pt1, pt2 );

        // Second overloaded version.
        x1 = 3.0;  y1 = 0.0;
        x2 = 0.0;  y2 = 4.0;
        distance = SKDistanceBetween( x1, y1, x2, y2 );
}
```

Document Revision Date

3 August, 1998

SKDistanceMoved()

Name

SKDistanceMoved()
Functions to take 2 points p1 and p2, and a direction dir, and compute the distance travelled along direction dir in travelling from p1 to p2.

Synopsis

```
#include <skanalyt.h>

float   SKDistanceMoved( SKCoordI p1, SKCoordI p2, float dir,
                SKAngleConvention angleConvention );
float   SKDistanceMoved( float x1, float y1, float x2, float y2, float dir,
                SKAngleConvention angleConvention );
```

Description

Functions to take 2 points p1 and p2 and compute the distance travelled along direction *dir* in travelling from p1 to p2. This is the length of the projection of the vector from pt1 to pt2 onto the line through pt1 with direction "dir". This is equivalent to computing the distance from p2 to the line thru p1 with direction perpendicular to the supplied direction *dir.* We must supply an *angleConvention* parameter as this distance is different for different angle conventions (for example, a mathematical *dir* of 150 will generally yield a different result than a meteorological *dir* of 150).

Note the required distance is more easily computed as the length of the projection of the vector from p1 to p2 onto a unit vector of direction *dir.* The equivalent computation described above was the method used in the original SKETCH system and is the method described in the SKE Library description document, so was used here. The 2 methods lead to equivalent formulas.

The first overloaded version of this function takes the two point arguments as *SKCoordI* structures; the second overloaded version takes the two points specified by their (x, y) coordinates (i.e. the first point has coordinates *(x1, y1)* and the second point has coordinates *(x2, y2)).* Both versions simply compute the distance described in the paragraph above and return this distance as a float.

Returns

The distance travelled along direction *dir*, when moving from the first supplied point to the second supplied point. This distance is returned as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordI pt1, pt2;
        float x1, y1, x2, y2;
        float dir1, dir2, dist1, dist2;
```

```
    // First overloaded version; direction in mathematical sense.
    pt1.x = 0;  pt1.y = 0;
    pt2.x = 0;  pt2.y = 5;
    dir1 = 40.0;
    dir1 = SKDistanceMoved( pt1, pt2, dir1, SK_MATH_CONVENTION );

    // Second overloaded version; direction in meteorological sense.
    x1 = 0.0;  y1 = 0.0;
    x2 = 0.0;  y2 = 5.0;
    dir2 = 50.0;
    dist2 = SKDistanceMoved( x1, y1, x2, y2,
            SK_METEO_CONVENTION );
}
```

Document Revision Date

17 July, 2002

SKExternalAngle()

Name

SKExternalAngle()
Function to compute the external angle (in degrees) of two intersecting line segments determined by three points. The second of the three points is considered the intersection point of the 2 line segments.

Synopsis

```
#include <skanalyt.h>

float SKExternalAngle( SKCoordI pt0, SKCoordI pt1, SKCoordI pt2 )
```

Description

Function to compute the external angle (in degrees) of two intersecting line segments determined by three points, *p0*, *p1*, and *p2*. The point *p1* is considered the middle point, i.e. we take the line segments from *p0* to *p1* and *p2* to *p1* and compute the external angle determined by these. Angle convention is not applicable here (e.g. a vector at 45 degrees in any system, intersecting a vector at 150 degrees in any system, will always have an external angle of 255 degrees).

Returns

The external angle determined by the three points; this angle (expressed in degrees) is returned as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordI p0, p1, p2;
        float externalAngle;

        p0.x = 0;  p0.y = 0;
        p1.x = 0;  p1.y = 1;
        p2.x = 1;  p2.y = 0;

        // The three points determine an external angle of 315 degrees.
        externalAngle = SKExternalAngle( p0, p1, p2 );
}
```

Document Revision Date

3 August, 1998

SKFlipDirection()

Name

SKFlipDirection()
Function to flip an input direction by 180 degrees. Makes sure the returned direction lies between 0 and 360.

Synopsis

```
#include <skanalyt.h>

float SKFlipDirection( float dir )
```

Description

Function to flip an input direction by 180 degrees. Makes sure the returned direction lies between 0 and 360.

Returns

The flipped direction, returned as a float.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        float flippedDir;

        // 30 degrees flips to 210; 210 to 30 (not 490).
        flippedDir = SKFlipDirection( 30.0 );
        flippedDir = SKFlipDirection( 210.0 );
}
```

Document Revision Date

3 August, 1998

SKIntersectionOfVectors()

Name

SKIntersectionOfVectors()
Function to take two rays (each specified by a base point point and a direction) and determine whether the two rays are converging.

Synopsis

```
#include <skanalyt.h>

short SKIntersectionOfVectors( SKCoordI p0, float direction0,
                               SKCoordI p1, float direction1,
                               SKAngleConvention angleConv );
```

Description

Function to take two rays (each specified by a base point point and a direction) and determine whether the two rays are converging (i.e. determine whether the rays determined by propagating the base points forward only along the appropriate directions will intersect). Since this function calls several other analytic geometry functions which require an *SKAngleConvention* indicator, one must be supplied to this function. It is assumed that *direction0* and *direction1* were both measured with the corresponding angle convention.

Returns

A short with value 1 if the rays are converging, 0 if they are not.

Note

This routine attempts to handle "ill-conditioned" problems, e.g. when the rays are very nearly parallel, when two rays are "chasing" each other, etc. Consult the in-line documentation for details of these degenerate cases.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordI p0, p1;
        float dir0, dir1;
        short intersecting;

        // First ray based at (0,0), 30 degrees (math convention).
        p0.x = 0;   p0.y = 0;   dir0 = 30.0;

        // Second ray at (1,0), 340 degrees (math convention).
        p1.x = 1;   p1.y = 0;   dir1 = 340.0;

        // Are the rays converging?
        intersecting = SKIntersectionOfVectors( p0, dir0, p1, dir1,
        SK_MATH_CONVENTION );
}
```

Document Revision Date *17 July, 2002*

SKPointApproachingLocation()

Name

SKPointApproachingLocation()
Function to determine whether the point at loc1, travelling with the specified direction, is approaching or receding from the point at loc2.

Synopsis

```
#include <skanalyt.h>

bool    SKPointApproachingLocation( SKCoordI loc1, SKCoordI loc2, float dir,
                                         SKAngleConvention angleConvention );
```

Description

This function determines whether the point at *loc1*, travelling with direction *direction*, is approaching or receding from the point at *loc2*. It is approaching if the angle of the vector from *loc1* to *loc2* is within 90 degrees of *direction*, otherwise it is receding. This function needs the *angleConvention* variable as it calls *SKDirectionFrom()*, which returns different results according to the angle measuring scheme. It is assumed that *direction* was initially measured with the scheme indicated by *angleConvention*.

Returns

Returns boolean TRUE if the first point is approaching the second point, otherwise FALSE.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordI loc1, loc2;
        float dir;
        bool approaching;

        // First point at (1, 1) moving with direction 35 degrees (mathematical
        // sense).  Second point at (2, 1).
        loc1.x = 1;   loc1,y = 1; dir = 35.0;
        loc2.x = 2;   loc2.y = 1;

        // Is first point approaching second location?
        approaching = SKPointApproachingLocation( loc1, loc2, dir,
                                               SK_MATH_CONVENTION );
}
```

Document Revision Date

4 August, 1998

SKTranslateXYPosition()

Name

SKTranslateXYPosition()
Function to take an input point (pixel), and translate the point in a supplied direction by a supplied distance.

Synopsis

```
#include <skanalyt.h>
SKCoordI SKTranslateXYPosition( SKCoordI point, float dir, float dist,
                                SKAngleConvention angleConvention );
```

Description

Function to take an input point (pixel), and translate the point in a supplied direction by a supplied distance. Returns the translated pixel (integer coordinates) in an SKCoordI structure.

Returns

Returns the translated pixel (integer coordinates) in an SKCoordI structure.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        SKCoordI point, translatedPoint;
        float direction, distance;

        // First point at (1, 1) moving with direction 35 degrees (meteorological
        // sense).
        point.x = 1;   point.y = 1; direction = 35.0;

        // Translate the point thru a distance of 11.4 pixels.  Output will be
        // truncated to integer values to give a true pixel coordinate.
        distance = 11.4;
        translatedPoint = SKTranslateXYPosition( point, direction,
                                distance, SK_METEO_CONVENTION );
}
```

Document Revision Date

4 August, 1998

SKVectorToComponents()

Name

SKVectorToComponents()
Convert a vector specified by its magnitude and direction to its x- and y-components.

Synopsis

```
#include <skanalyt.h>

SKCoordF SKVectorToComponents( float theta, float range,
                        SKAngleConvention angleConvention );
```

Description

Given a vector in (*range, theta*) format, convert it to its (*x, y*) components. Must account for whether *theta* was measured in the mathematical sense or the meteorological sense (as indicated by the *angleConvention* argument).

Returns

The equivalent vector expressed as (floating-point) x- and y-components. The result is returned as an *SKCoordF* structure.

Example

```
#include <skanalyt.h>

int main( int argc, char *argv[] )
{
        float theta, range;
        SKCoordF components;
        theta = 30.0;
        range = 5.0;

        // Given a vector with magnitude 5.0 and direction 30.0 degrees (in the
        // meteorological sense), find the x- and y-components of the vector.
        components = SKVectorToComponents( theta, range,
                        SK_METEO_CONVENTION );
}
```

See Also

SKComponentsToVector()

Document Revision Date

4 August, 1998

5. Array Arithmetic

5.1 Summary

CSKETCH array arithmetic functions. Some common array arithmetic functions such as *Min*(), *Max*(), and *Magnitude*() of the elements of an *SKArray*. Also includes more complex functions such as *SKMedianFilter*() and *SkLsqDerivFilter*(). Note that other common arithmetic functions which are encoded as C++ operators are discussed in the *SKArray* class description., elsewhere in this document. Such arithmetic operators include +, +=, -, -=,etc.

5.2 Functions

Array Arithmetic functions begin on the following page.

Magnitude()

Name

Magnitude()

Function to return a new SKArray containing the pixelwise magnitudes of the vectors stored in input SKArrays xVec and yVec.

Synopsis

#include <skarrayarith.h>

SKArray<float> Magnitude(SKArray<float>& xVec, SKArray<float>& yVec);

Description

Function to return a new SKArray containing the pixelwise magnitudes of the vectors stored in input *SKArrays xVec* and *yVec*. The input arrays *xVec* and *yVec* are considered to be holding x and y components of an array of vectors; e.g. the vector at pixel (12, 15) has x-component = *xVec*(12, 15) and y-component = *yVec*(12, 15). The magnitude at each pixel is the square root of the x-component squared plus the y-component squared. This function first checks to see if the input arrays *xVec* and *yVec* are equal in dimension, size, and stride.

Returns

An *SKArray<float>* which stores the pixelwise magnitudes of all the vectors with x-components stored in *xVec* and y-components stored in *yVec*.

Document Revision Date

5 August, 1998

Max()

Name

Max()

Functions to return one of more new or edited SKArrays containing the pixelwise maximum values of a pair of SKArrays. See documentation below for further elaboration on this overloaded function.

Synopsis

```
#include <skarrayarith.h>

template<class T>
SKArray<T> Max( SKArray<T>& in1, SKArray<T>& in2, int anyNILIsNIL );
template<class T>
void Max( SKArray<T> *in1, SKArray<T>& in2,
          SKArray<T> *in3, SKArray<T>& in4,int anyNILIsNIL );
```

Description

The 2-array version of *Max()* works as follows. A new *SKArray* of the same type as the 2 input arrays is created. That array will contain the pixelwise maximum values of the 2 input arrays. The *anyNILIsNIL* flag controls how *NIL* values in either input is handled. If *anyNILIsNIL* == 1 (TRUE), then if either of the pixels is *NIL*, the result is NIL. If *anyNILIsNIL* == 0 (FALSE), and the pixel from the first input is *NIL*, then the value of the corresponding comparison pixel is copied to the output *SKArray* (this pixel value may itself be *NIL*). If neither comparison pixel in *NIL*, then the output pixel is simply the maximum of the 2 compared pixels.

The 4-array version of *Max()* returns two modified arrays whose contents are adjusted as follows. Refer to the documentation of the 2-array version of *Max()* described above. The 4-array version of *Max()* handles its first 2 input arrays exactly as the 2-array version of *Max()* handles its 2 input arrays, with the exception that in the 4-array version the first input (*in1*) is overwritten with maximum values (or *nil's*, which may occur at some pixels). In the 2-array version, a new output array was created so none of the inputs are edited. The *anyNILIsNIL* flag serves exactly the same purpose for this comparison as it served in the 2-array version.

In addition to the editing of *in1*, the "companion" array *in3* may also be edited according to the values of *in4*. However, any editing of *in3* is controlled by the results of comparing *in1* pixels to *in2* pixels. The bottom line of the editing process is:

(1) If the pixel at *in1* is unchanged, then the corresponding pixel in *in3* is also unchanged.
(2) If the pixel in *in1* is set directly to *NIL* (because of the *anyNILIsNIL* flag), then the corresponding pixel in *in3* is also set directly to *NIL*.

(3) If the pixel in *in1* is overwritten with the corresponding pixel of *in2*, then the corresponding pixel in *in3* is also overwritten with the corresponding pixel in *in4*. Note in this case, the result pixel could be *NIL* in none, one, or both of *in1* and *in3*; this depends solely on the values of *in2* and *in4* which were used for the overwriting process.

Returns

The input arrays *in1* and *in3* are edited as described above.

Warning

The 4-array version of *Max()* edits the input arrays *in1* and *in3*; this is different from the behavior of the 2-array version of *Max()*, which does not edit any of its inputs.

Document Revision Date

6 August, 1998

Min()

Name

Min()
Function to return a new SKArray containing the pixelwise minimum values of a pair of input SKArrays.

Synopsis

#include <skarrayarith.h>

template<class T>
SKArray<T> Min(SKArray<T>& in1, SKArray<T>& in2, int anyNILIsNIL);

Description

Function to return a new *SKArray* containing the pixelwise minimum values of a pair of input *SKArrays*. This global function assumes that arrays *in1* and *in2* are the same size arrays. That is, the number of dimensions is equal, and each dimension's sizes and steps are equal. This stipulation must hold since the purpose of the function is to compare each element in array *in1* to each element in array *in2* and find the minimum at each pixel.

Returns

An *SKArray* which stores the pixelwise minimum values of the two input *SKArrays in1* and *in2*. The returned *SKArray* has the same data type as the two input arrays (which must have the same type themselves, e.g. both inputs are float *SKArrays*, or both are integer *SKArrays*, etc.)

The Boolean input *anyNILIsNIL* controls how *NIL* values are handled. If *anyNILIsNIL* == 1 (TRUE), then if either of the pixels is *NIL*, the result is *NIL*. If *anyNILIsNIL* == 0 (FALSE), and the value at a particular pixel in the first input in1 is *NIL*, then the value of the corresponding comparison pixel from in2 is stored in the output array (this value may itself be *NIL*).

Document Revision Date

5 August, 1998

SKCopyMaskedElements()

Name

SKCopyMaskedElements()
Copy values from one SKArray to another SKArray, at pixel locations where a third 'mask' array has value equal to some prescribed value.

Synopsis

```
#include <skarrayarith.h>

template <class T>
void SKCopyMaskedElements( SKArray<T>& output, SKArray<T>& input,
                           SKArray<short>& mask, short maskValue /* = 1 */ );
```

Description

This function loops simultaneously over the *input*, *output*, and *mask* arrays. At pixels where the value in the *mask* array is equal to *maskValue*, the corresponding pixel value in *input* is copied to the corresponding location in *output*. The *maskValue* is an optional argument -- if no value is supplied for it in the call to *SKCopyMaskedElements()*, it defaults to 1.

Returns

The modified *output* array. Pixels in the *output* array are set equal to the corresponding pixel values in the *input* array, at locations where the *mask* array has value equal to the *maskValue*.

Document Revision Date

5 August, 1998

SKDerivativeFilter()

Name

SKDerivativeFilter()
Compute a one-dimensional numerical derivative over a specified window length at all pixels in an input SKArray.

Synopsis

```
#include <skarrayarith.h>

template <class T>
SKArray<float> SKDerivativeFilter( SKArray<T>& input, short dimension,
                                   short windowWidth );
```

Description

CSKETCH Derivative Filter function. For each applicable pixel in input, computes a numerical derivative of the data in the array, using a window of width *windowWidth* in dimension specified by *dimension*. Output array is shortened by (*windowWidth* - 1) in the dimension of the filtering. This function is no longer frequently used since the writing of *SKLsqDerivFilter()*, because the least-squares filter can handle missing data values in the input; *SKDerivativeFilter()* cannot handle missing values, i.e. there must be no *nil* pixels in the input array to this function. See the in-line source documentation for further discussion of this function.

Returns

A float *SKArray* which stores the pixelwise derivatives of the input data array.

Warning

This function cannot handle *nil* values in its *input* array; thus if the function is to be used, the developer must make sure that the supplied *input* does not have any *nil* values.

See Also

SKLsqDerivFilter()

Document Revision Date

5 August, 1998

SKDivideByElements()

Name

SKDivideByElements()
Function to return an array that has each element set to the quotient of the corresponding numerator array element divided by the denominator array element.

Synopsis

#include <skarrayarith.h>

template <class T>
SKArray<T> SKDivideByElements(SKArray<T>& num, SKArray<T>& den);

Description

Global function to return an array that has each element set to the quotient of the corresponding numerator array element divided by the denominator array element. If either of the numerator or denominator pixel is *nil*, or if the denominator pixel is 0, then the corresponding output pixel is also *nil.*

Returns

An *SKArray* which is the same type (float, int, etc.) as the input *num* and *den* *SKArrays*, which must themselves be of the same type.

See Also

SKArray<T>& SKArray<T>::operator /= (T) (scalar division operator for SKArrays; divides each element of the array by the right-hand side scalar).

Document Revision Date

6 August, 1998

SKFirstDerivativeSum()

Name

SKFirstDerivativeSum()
Function which calls SKDerivativeFilter and then SKSumFilter so that a given pixel in the output image stores the sum of the first derivatives of points within a small window of the corresponding pixel in the input image.

Synopsis

#include <skarrayarith.h>

SKArray<float> SKFirstDerivativeSum(SKArray<short>& input,
short dimension, short windowWidth);

Description

CSKETCH First Derivative Sum Filter function. Calls *SKDerivativeFilter()* and then *SKSumFilter()* so that a given pixel in the output image stores the sum of the derivatives of points within a small window of the corresponding pixel in the input image.

Since both the derivative filter and the sum filter produce a "shorter" output image in the direction of the filtering, this routine first creates an expanded image (expanded by 2 * (*windowWidth* - 1) in the direction of the filtering). The expanded array has its margin filled in via the *SKArray* class member function *SKArray<T>::Mirror()*. Since each filter operation shortens the output by *windowWidth* - 1 in the filter direction, the output image of the full *SKFirstDerivativeSum()* operation is sized the same as the original input.

Returns

A float *SKArray* which stores the pixelwise first derivative sums of the input data array.

Warning

This function cannot handle *nil* values in its *input* array, as it calls *SKDerivativeFilter()* and *SKSumFilter()*, which both cannot handle *nil* values. Thus if the function is to be used, the developer must make sure that the supplied *input* does not have any *nil* values.

See Also

SKDerivativeFilter(), SKSumFilter(), SKArray<T>::Mirror(), SKLsqDerivFilter()

Document Revision Date

7 August, 1998

SKLsqDerivFilter()

Name

SKLsqDerivFilter()
Compute a one-dimensional numerical derivative over a specified window length at all pixels in an input SKArray, using a least-squares method.

Synopsis

```
#include <skarrayarith.h>

template <class T>
SKArray<float> SKLsqDerivFilter( SKArray<T>& input, short dimension,
                        short windowHalfWidthX, short windowHalfWidthY,
                        float minGoodFraction, float minCorrelation);
```

Description

CSKETCH Least Squares Derivative Filter Function. For each pixel of the *input* image, a window whose half-widths are *windowHalfWidthX* and *windowHalf-WidthY* is used to determine the derivative along the specified dimension. The size of the window is reduced symmetrically near the edges, so that the output image is the same size as the input image. The input *minGoodFraction* specifies the minimum fraction of valid (i.e. non-*nil*) pixels in a window to return a non-nil value for the pixel currently being processed. In particular, this function can handle *nil* values in the *input* image; the SKDerivativeFilter() function cannot. The input *minCorrelation* specifies the minimum correlation coefficient to return a non-nil value.

Returns

A float *SKArray* which stores the pixelwise derivatives of the input data array, obtained using the least-squares method.

See Also

SKDerivativeFilter()

Document Revision Date

6 August, 1998

SKMedianFilter()

Name

SKMedianFilter()
SKArray median filter function.

Synopsis

#include <skarrayarith.h>

template <class T>
SKArray<T> SKMedianFilter(SKArray<T>& input, short winX, short winY, float fraction, SKPadOp padOp)

Description

CSKETCH median filter function. A 2-D window (window half sizes are *winX* and *winY*) is centered on each pixel of the input image. All values that fall within this window are added to an array. The array is then sorted. The "middle" element of the array is used to determine the median value that is placed in the corresponding element of the output array. *Nil* pixels are ignored, but a specified fraction of the pixels within the window must be non-*nil* in order for the output image to have a non-*nil* value at the corresponding pixel. The input *SKArray* is expected to be NON-PADDED. Padding is done internally via the specified *padOp*. See documentation for the helper class *SKArrayPad* for a full description of supported padding options.

Returns

A new *SKArray* of the same type as the input array; each element of the output array is the median of a 2D window of data centered at the corresponding pixel in the input array.

Document Revision Date

6 August, 1998

SKScaleArrayToBounds()

Name

SKScaleArrayToBounds()
Function to take an input array and scale a copy of the array to a specified range.

Synopsis

```
#include <skarrayarith.h>

template <class T, class V>
SKArray<T> SKScaleArrayToBounds( SKArray<T>& in1,  short num_bins,
                                 V high, V low )
```

Description

Function to take an input array *in1* and scale a copy of the array to a specified range. Specifically, the number of (integer) bins, *num_bins,* for the output array is passed in, as are *high* and *low* bounds for the scaling. Original input pixels which are less than the *low* bound are set to zero in the output; input pixels which are greater than the *high* bound are set to (*num_bins* - 1); input pixels between *low* bound and *high* bound are ramped linearly from 0 to (*num_bins* - 1) (and converted to the correct output type <T>). Note that this function is doubly-templatized; the output array of type *SKArray<T>* is an *SKArray* of the same data type (T) as the input *in1*, while the *high* and *low* bounds for the scaling may be of a different data type (V). For example, one could specify float values for the low and high boundaries even when scaling an *SKArray* of type short.

Returns

The scaled *SKArray*, which has the same data type as the *in1* input *SKArray.*

Document Revision Date

6 August, 1998

SKSumFilter()

Name

SKSumFilter()
For each pixel in an input SKArray, compute a one-dimensional sum of pixel-values over a specified window length; results stored in an output SKArray.

Synopsis

#include <skarrayarith.h>

```
template<class T>
SKArray<T> SKSumFilter( SKArray<T>& input, short dimension,
                        short windowWidth );
```

Description

CSKETCH Sum Filter function. For each applicable pixel in *input*, computes a sum of the data in the array, over a window of width *windowWidth* in dimension specified by *dimension*. Output array is shortened by (*windowWidth* - 1) in the dimension of the filtering. This function cannot handle missing data values in the *input*, i.e. there must be no *nil* pixels in the *input* array. See the in-line source documentation for further discussion of this function.

Returns

An *SKArray* of the same type as the input *SKArray*, which stores the pixelwise window sums of the input data array.

Warning

This function cannot handle *nil* values in its *input* array; thus if the function is to be used, the developer must make sure that the supplied *input* does not have any *nil* values.

Document Revision Date

6 August, 1998

6. Fuzzy Sets

6.1 Summary

This section includes various CSKETCH utilities for fuzzy logic applications. Most of the functions in this section are simple functions whose graph lies in the range from 0.0 to 1.0. Examples are *SKRisingRamp*() and *SKFallingRamp*(), which rise (respectively fall) from 0.0 to 1.0 (resp 1.0 to 0.0) over a specified x range. The functions are typically used to generate weights with value between 0.0 and 1.0, to be used in computing various weighted averages used by the algorithms. In addition, this function contains *SKFuzzyWeightedAverage*(), which is used to compute a fuzzy weighted average of images. See documentation for class *SKFuzzyFunc* for further elaboration.

6.2 Functions

Fuzzy logic functions begin on the following page.

SKFallingRamp()

Name

SKFallingRamp()

Compute a particular value of a function whose graph is a 'falling ramp'. Given a single input x-value, the corresponding y-value on the graph is computed and returned.

Synopsis

```
#include <skfuzzysets.h>

float SKFallingRamp( float xvalue, float *inflectionPoints );
```

Description

Function whose graph is a 'falling ramp'. The *inflectionPoints* argument specifies two points on the x-axis, namely x0 and x1. For points x with x < x0, the corresponding y-value is 1.0. When x > x1 the y-value is 0.0. From x = x0 to x = x1 the graph is an decreasing straight line, going from the point (x0, 1.0) to (x1, 0.0). Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Returns

Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Example

```
#include <skfuzzysets.h>

int main( int argc, char *argv[] )
{
        float infPts[2], y;

        // Falling ramp falls from y=1 at x=0 to y=0 at x=5.
        infPts[0] = 0;   infPts[1] = 5;

        // Compute y-values at x = -1, 0.5, and 10
        y = SKFallingRamp( -1.0, infPts );
        y = SKFallingRamp( 0.5, infPts );
        y = SKFallingRamp( 10.0, infPts );
}
```

Document Revision Date

6 August, 1998

SKFallingS()

Name

SKFallingS()
Compute a particular value of a function whose graph is a 'falling S'. Given a single input x-value, the corresponding y-value on the graph is computed and returned.

Synopsis

#include <skfuzzysets.h>

*float SKFallingS(float xvalue, float *inflectionPoints);*

Description

Function whose graph is a 'falling S'. The *inflectionPoints* argument specifies three points on the x-axis, namely x0, x1, and x2. For points x with x < x0, the corresponding y-value is 1.0. When x > x2 the y-value is 0.0. From x = x0 to x = x1 the graph is a downwards parabola; the value at x = x1 is 0.5. From x = x1 to x = x2 the graph is an upwards parabola; the value at x = x1 is again 0.5 for continuity, while the value at x = x2 is 0.0. Thus from x = x0 to x = x2 the graph resembles a reverse letter 's'.Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

This function is computed simply as 1.0 - *SKRisingS(xvalue, inflectionPoints);*

Returns

Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Example

```
#include <skfuzzysets.h>

int main( int argc, char *argv[] )
{
        float infPts[3], y;

        // Falling s falls from y=1 at x=0 to y=0 at x=5. The graph changes
        // from a downwards parabola to an upwards parabola at x = 2.
        infPts[0] = 0.0;   infPts[1] = 2.0;   infPts[2] = 5.0;

        // Compute y-values at x = -1, 0.5, and 10
        y = SKFallingS( -1.0, infPts );
        y = SKFallingS( 0.5, infPts );
        y = SKFallingS( 10.0, infPts );
}
```

See Also

SKRisingS()

Document Revision Date *17 July, 2002*

SKFuzzyWeightedAverage()

Name

SKFuzzyWeightedAverage()
Function to return the fuzzy weighted average of an array of interest images.

Synopsis

#include <skfuzzyavg.h>

*SKArray<short> SKFuzzyWeightedAverage(SKArray<short> *images[],*
*int numImgs, SKFuzzyFunc<short,float> *weightFunc[]);*

Description

Function to return the fuzzy weighted average of an array of interest images. There should be a corresponding fuzzy weight function in the *weightFunc* array for each of the input images in *images*. Each pixel in each input image is weighted by the corresponding weight function prior to averaging to find the output value at that pixel. By using *SKFuzzyFunc* objects as the weighting functions, we can in theory have a different weight at every pixel of the input image. This differs e.g. from the averaging process in MIGFA which has only a choice of 2 weights at each pixel.

Returns

The weighted average of all the input *SKArrays*. This is returned as an *SKArray*<short>; values are converted to short after averaging by truncation.

See Also

SKAverageInterestImages(), SKAverageInterestImagesExceptMin().

Document Revision Date

2 December, 1998

SKRampPlateau()

Name

SKRampPlateau()
Compute a particular value of a function whose graph is a 'ramp plateau'. Given a single input x-value, the corresponding y-value on the graph is computed and returned.

Synopsis

```
#include <skfuzzysets.h>

float SKRampPlateau( float xvalue, float *inflectionPoints );
```

Description

Function whose graph is a 'ramp plateau'. The *inflectionPoints* arguments specifies four points on the x-axis, namely x0, x1, x2, and x3. The y-value is zero when x <= x0; y increases linearly from 0.0 to 1.0 for x0 <= x < x1; the graph then plateaus, i.e. y is always = 1.0 for x1 <= x < x2; the graph then ramps down linearly to 0.0 for x2 <= x < x3; and finally the graph is zero when x >= x3. Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Returns

Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Example

```
#include <skfuzzysets.h>

int main( int argc, char *argv[] )
{
        float infPts[4], y;

        // Ramp plateau rises from y=0 at x=0 to y=1 at x=5; it plateaus
        // (i.e. y =1) from x=5 to x=7;  then falls linearly to y=0 at x=10.
        infPts[0] = 0;   infPts[1] = 5;   infPts[2] = 7;   infPts[3] = 10;

        // Compute y-values at x = -1, 0.5, and 10
        y = SKRampPlateau( -1.0, infPts );
        y = SKRampPlateau( 0.5, infPts );
        y = SKRampPlateau( 10.0, infPts );
}
```

Document Revision Date

17 July, 2002

SKRisingRamp()

Name

SKRisingRamp()
Compute a particular value of a function whose graph is a 'rising ramp'. Given a single input x-value, the corresponding y-value on the graph is computed and returned.

Synopsis

```
#include <skfuzzysets.h>

float SKRisingRamp( float xvalue, float *inflectionPoints );
```

Description

Function whose graph is a 'rising ramp'. The *inflectionPoints* argument specifies two points on the x-axis, namely x0 and x1. For points x with x < x0, the corresponding y-value is 0. When x > x1 the y-value is 1.0. From x = x0 to x = x1 the graph is an increasing straight line, going from the point (x0, 0.0) to (x1, 1.0). Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Returns

Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Example

```
#include <skfuzzysets.h>

int main( int argc, char *argv[] )
{
        float infPts[2], y;

        // Rising ramp rises from y=0 at x=0 to y=1 at x=5.
        infPts[0] = 0;   infPts[1] = 5;

        // Compute y-values at x = -1, 0.5, and 10
        y = SKRisingRamp( -1.0, infPts );
        y = SKRisingRamp( 0.5, infPts );
        y = SKRisingRamp( 10.0, infPts );
}
```

Document Revision Date

6 August, 1998

SKRisingS()

Name

SKRisingS()

Compute a particular value of a function whose graph is a 'rising S'. Given a single input x-value, the corresponding y-value on the graph is computed and returned.

Synopsis

```
#include <skfuzzysets.h>

float SKRisingS( float xvalue, float *inflectionPoints );
```

Description

Function whose graph is a 'rising S'. The *inflectionPoints* argument specifies three points on the x-axis, namely x0, x1, and x2. For points x with x < x0, the corresponding y-value is 0. When x > x2 the y-value is 1.0. From x = x0 to x = x1 the graph is an upwards parabola; the value at x = x1 is 0.5. From x = x1 to x = x2 the graph is a downwards parabola; the value at x = x1 is again 0.5 for continuity, while the value at x = x2 is 1.0. Thus from x = x0 to x = x2 the graph resembles a letter 's'. Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Returns

Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Example

```
#include <skfuzzysets.h>

int main( int argc, char *argv[] )
{
        float infPts[3], y;

        // Rising s rises from y=0 at x=0 to y=1 at x=5. The graph changes
        // from an upwards parabola to a downwards parabola at x = 2.
        infPts[0] = 0.0;   infPts[1] = 2.0;   infPts[2] = 5.0;

        // Compute y-values at x = -1, 0.5, and 10
        y = SKRisingS( -1.0, infPts );
        y = SKRisingS( 0.5, infPts );
        y = SKRisingS( 10.0, infPts );
}
```

Document Revision Date

6 August, 1998

SKSPlateau()

Name

SKSPlateau()
Compute a particular value of a function whose graph is an 's plateau'. Given a single input x-value, the corresponding y-value on the graph is computed and returned.

Synopsis

```
#include <skfuzzysets.h>

float SKSPlateau( float xvalue, float *inflectionPoints );
```

Description

Function whose graph is an 's plateau'. The *inflectionPoints* argument specifies six points are specified on the x-axis, namely x0, x1, x2, x3, x4, and x5. When x < x2 the graph is the 'rising s' described by *SKRisingS(xvalue, x0, x1, x2).* At x = x2 the graph plateaus, i.e. y = 1.0 for x2 <= x < x3. When x >= x3 the graph is the 'falling s' described by *SKFallingS(xvalue, x3, x4, x5).* Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Returns

Given the input x = *xvalue*, the corresponding yvalue on the graph of this function is computed and returned.

Example

```
#include <skfuzzysets.h>

int main( int argc, char *argv[] )
{
        float infPts[6], y;

        // Graph is a 'rising s' for 0 <=x <= 4; plateaus at y=1 from x=4 to
        // x=5; and is a 'falling s' from x=5 to x=9.
        infPts[0] = 0;  infPts[1] = 1;  infPts[2] = 4;
        infPts[3] = 5;  infPts[4] = 8;  infPts[5] = 9.

        // Compute y-values at x = -1, 0.5, and 10
        y = SKSPlateau( -1.0, infPts );
        y = SKSPlateau( 0.5, infPts );
        y = SKSPlateau( 10.0, infPts );
}
```

See Also

SKRisingS(), SKFallingS()

Document Revision Date

6 August, 1998

7. Image Processing

7.1 Summary

General image processing functions.

7.2 Functions

Image Processing functions begin on the following page.

SKConvolutionOf()

Name

SKConvolutionOf()

*Routine to compute the 'convolution' of an input image and a kernel. In actuality, what is computed is the convolution of the input image with the matrix whose (i,j) entry is the (-i, -j) entry of the kernel. By definition the true convolution of an image with a kernel at pixel (i,j) is Sum(k) Sum(l) ((image(i+k, j+l)) * kernel(-k, -l)). So what is computed here is really the dot product of the kernel with an equally-sized patch of the input image.*

Synopsis

```
#include <skimageproc.h>

SKArray<T> SKConvolutionOf( SKArray<T>& input,
                            SKArray<float> &kernel );
```

Description

Routine to compute the 'convolution' of an input image and a kernel. In actuality, what is computed is the convolution of the the image with the matrix whose (i,j) entry is the (-i, -j) entry of the *kernel*. By definition the true convolution of an image with a *kernel* at pixel (i,j) is

*Sum(k) Sum(l) ((image(i+k, j+l)) * kernel(-k, -l)).*

So what is computed here is really the dot product of the *kernel* with an equally-sized patch of the *input* image.

Returns

A new *SKArray*, the output convolved array.

Warning

The sizes of the *kernel* are assumed to be odd; an 'assert' will fail otherwise.

Document Revision Date

18 November, 1998

SKGaussianKernel()

Name

SKGaussianKernel()
Function to compute a (2-dimensional) Gaussian kernel.

Synopsis

#include <skimageproc.h>

SKArray<float> SKGaussianKernel(float xPeakWidth, float yPeakWidth,
float xOffset, float yOffset);

Description

Function to compute a Gaussian kernel, with the peak widths specified by *xPeakWidth* and *yPeakWidth,* and the peak location at *xOffset* and *yOffset* relative to the center of the returned image.

Returns

A new *SKArray,* the 2D gaussian kernel.

Note

Note that the x- and y-sizes of the returned kernel are not known at the time of the call to this function. Rather, we first compute a cutoff value below which we consider the kernel to have value 0. This is based on the constant SK_GAUSS_INTEGRAL_OUTSIDE_KERNEL (nominally defined to be 0.01). The 'extent' of the kernel can be computed based on this cutoff vale and the peak widths. The idea is that the contribution to the infinite integral of the gaussian surface over the region of the x-y plane outside the kernel extent should be less than SK_GAUSS_INTEGRAL_OUTSIDE_KERNEL. Then we must add *xOffset* and *yOffset* to the kernel sizes to account for the shift of the peak away from (0, 0).

Document Revision Date

18 November, 1998

SKMarkMissingOf()

Name

SKMarkMissingOf()
Function to mark pixels as missing (i.e. set them to NIL) if too few pixels in a window surrounding the current pixel do not have values within a specified range of the current pixel.

Synopsis

#include <skimageproc.h>

template<class T>
*T SKMarkMissingOf(SKArray<T>& input, T *xPtr, void *args);*

Description

Function to mark pixels as missing (i.e. set them to NIL) if too few pixels in a window surrounding the current pixel do not have values within a specified range of the current pixel. Processing at a given output pixel is as follows:

A window of data is centered at the corresponding input pixel (which is pointed to by *xPtr*). The *args* argument is a pointer to a structure of type *SKMarkMissingOfArgs* which houses the 4 remaining arguments: a kernel (type *SKArray<short> *)* which represents the data window; two floats, the *lowerRange* and *upperRange* for comparison to the center pixel of the window of data; and the *minValidCount* (a short). A count is made of pixels within the window which lie in the range *centerPixelValue + lowerRange* up to *centerPixelValue + upperRange* (inclusive). Thus *lowerRange* must be less than, or equal to, *upperRange*. If this count exceeds the supplied *minValidCount*, then the corresponding pixel in the output image is set to the original center pixel value of the input image. If the count is too small, the corresponding output pixel is set to *NIL*.

Returns

The result of applying the 'mark missing of' operation at the current pixel; e.g. if *xPtr* corresponds to pixel (x1, y1) in the input image, then *outputPixel* will be the value at (x1, y1) in the final output image. The *outputPixel* will be *NIL* if the input pixel was an outlier, otherwise it will be the original value of the input pixel.

Note

Note that this function only returns the output pixel for the pixel of the input which is currently being processed. To perform the *SKMarkMissingOf()* function at all locations inside an array, use the *SKArray<T>::Apply()* member function.

Note that the center pixel of the window is NOT allowed to count towards the number of pixels within range. For example, if the window is 3 by 3 and the *minValidCount* is 4, then 4 of the eight non-central pixels must be within range. Rather than adding an inefficient check to see if the pixel currently being tested is the center pixel, this routine simply increments the *minValidCount* by 1; thus the center pixel's contribution is effectively thrown out (the center pixel value will always be in range).

This function requires three additional args beyond the standard 'Apply' args; namely the lowerRange, upperRange, and minValidCount. The kernel for the function and the 3 additional args will be packed into a structure defined in skimageproc.h. A pointer to the struct will then be cast to void* before the call to this function, so that the 'Apply' function will work with this function.

See Also

SKArray<T>::Apply()

Document Revision Date

18 November, 1998

SKShrinkArray()

Name

SKShrinkArray()
Function to produce a shrunk version of a given input array.

Synopsis

```
#include <skimageproc.h>

template<class T>
SKArray<T> SKShrinkArray( SKArray<T>& input,
                          short newXSize, short newYSize );
```

Description

Function to produce a shrunk version of a given input array. Each pixel in the output image replaces all pixels contained in a rectangular subwindow of the input image. The output pixel is the median value of the input pixels in the sub-window. Note that this function copies scaling information from the input array to the output array as the input's data may have been scaled.

Returns

A new *SKArray*, the shrunk array.

Warning

The specified *newXSize* and *newYSize* must divide exactly the X and Y sizes (respectively) of the original *input* image, e.g. a 10-by-10 array may be shrunk to a 5-by-5 or 5-by-2 but not a 3-by-3 or 3-by-5. If the new sizes do not exactly divide the original sizes, an 'assert' statement will fail.

Document Revision Date

18 November, 1998

8. General Mathematical Functions

8.1 Summary

General-purpose mathematical functions defined in CSKETCH, such as *Abs*(), *Min*(), *Max*(), etc. Also includes some standard array mathematical functions such as *SKArrayComputeMedian*().

8.2 Functions

General purpose mathematical function descriptions begin on the following page.

ABS()

Name

ABS()
Inline templatized function to return the absolute value of a single scalar argument

Synopsis

```
#include <skmath.h>

template <class T>
inline T ABS( T a );
```

Description

Inline templatized function to return the absolute value of a single scalar argument.

Returns

The absolute value of the supplied numerical argument. The returned value has the same numerical type (float, int, etc.) as the supplied input.

Document Revision Date

7 August, 1998

MAX()

Name

MAX()

Inline templatized function to return the maximum of two supplied scalar arguments. The two arguments must have the same data type; if they do not, cast one (or both) so the types agree.

Synopsis

```
#include <skmath.h>

template <class T>
inline T MAX( T a, T b );
```

Description

Inline templatized function to return the maximum of two supplied scalar arguments. The two arguments must have the same data type; if they do not, cast one (or both) so the types agree. For example:

```
// Won't compile, due to conflicting types:
float a = 1.0;
float b = MAX( a, 0 );

// Will compile because of casting:
float a = 1.0;
float b = MAX( a, (float) 0 );
```

Returns

The maximum of the two supplied arguments. The maximum is returned as the same type (T) as the two input arguments.

Document Revision Date

10 August, 1998

MIN()

Name

MIN()
Inline templatized function to return the minimum of two supplied scalar arguments. The two arguments must have the same data type; if they do not, cast one (or both) so the types agree.

Synopsis

#include <skmath.h>

template <class T>
inline T MIN(T a, T b);

Description

Inline templatized function to return the minimum of two supplied scalar arguments. The two arguments must have the same data type; if they do not, cast one (or both) so the types agree. For example:

```
// Won't compile, due to conflicting types:
float a = 1.0;
float b = MIN( a, 0 );

// Will compile because of casting:
float a = 1.0;
float b = MIN( a, (float) 0 );
```

Returns

The minimum of the two supplied arguments. The minimum is returned as the same type (T) as the two input arguments.

Document Revision Date

10 August, 1998

MOD()

Name

MOD()
Function to return a mod b, e.g. 8 mod 6 = 2.

Synopsis

#include <skmath.h>

float MOD(float a, float b);

Description

Function to return 'a' mod 'b', e.g 8 mod 6 = 2. Correctly handles negative numbers as well, e.g. -9 mod 4 = 3. The returned value lies in the range from 0 to b, even when b is negative (e.g. 7 mod -4 = -1).

Returns

The value of *a* mod *b*, returned as a float.

Document Revision Date

10 August, 1998

SGN()

Name

SGN()
Inline templatized function to return the sign of a single supplied scaler. Returns 1 if argument is >= 0.0, otherwise returns -1.

Synopsis

```
#include <skmath.h>

template <class T>
inline short SGN( T a )
```

Description

Inline templatized function to return the sign of a single supplied scaler. Returns 1 if argument *a* is >= 0.0, otherwise returns -1.

Returns

The sign of the supplied argument. The value 1 or -1 is returned as the same type (float, short, etc.) as the input argument.

Document Revision Date

10 August, 1998

SKArrayComputeMean()

Name

SKArrayComputeMean()
Function to obtain the mean value of an array. Mean value is returned as a float in all cases.

Synopsis

#include <skmath.h>

template<class T>
float SKArrayComputeMean(SKArray<T>& input);

Description

Function to obtain the mean value of the *input* SKArray. The mean of the array is the sum of all non-*NIL* pixels in the array divided by the number of non-*NIL* pixels. The mean is returned as a floating-point number. If all pixels are *NIL*, the returned mean is (float) *NIL*.

The function handles 1, 2, and 3D *SKArrays*.

Returns

The mean value of the array is returned as a float .

Warning

In cases where all input values are *NIL*, the function returns the floating-point value of *NIL*, not the value of *NIL* corresponding to the input array's data type. This is bacause the mean of a set of numbers is in general not a perfect integer so should always be returned as a float.

Document Revision Date

10 August, 1998

SKArrayComputeMedian()

Name

SKArrayComputeMedian()
Function to obtain the median value of an array. Median value is returned as the same type (int, float, etc.) as the input SKArray.

Synopsis

#include <skmath.h>

template<class T>
float SKArrayComputeMedian(SKArray<T>& input);

Description

Templatized function to take an *input SKArray* and return the median of the non-*nil* values in the current "slice" of the *SKArray*. If all pixels are *NIL*, the returned median is the correct value of *NIL* for the type of data in the *input* array.

Currently only handles 1D and 2D arrays.

Returns

The median value of the *input* array is returned, as the same type (int, float) etc. as the *input* array.

Document Revision Date

17 August, 1998

SKArrayComputeStdDev()

Name

SKArrayComputeStdDev()
Overloaded functions to compute and return the standard deviation of the non-nil values of an SKArray. The standard deviation is returned as a float in all cases.

Synopsis

```
#include <skmath.h>

template<class T>
float SKArrayComputeStdDev( SKArray<T>& input );

template<class T>
float SKArrayComputeStdDev( SKArray<T>& input, float mean );
```

Description

Function to compute and return the standard deviation of the non-nil values of an SKArray. The first version is not supplied the mean value so it must be computed; the second overloaded version takes the mean as an additional argument. If all (or all but one) pixel is *NIL*, the returned standard deviation is (float) *NIL*.

The function handles 1, 2, and 3D *SKArrays*.

Returns

The standard deviation of the array's data values is returned as a float .

Warning

In cases where all input values are *NIL*, the function returns the floating-point value of *NIL*, not the value of *NIL* corresponding to the input array's data type. This is because the standard deviation of a set of numbers is in general not a perfect integer so should always be returned as a float.

Document Revision Date

10 August, 1998

SKArrayDecode()

Name

SKArrayDecode()

Function to undo the encoding done in SKArrayEncode. That is, convert the array values from scaled values to actual 'real world' values. The formula is:

actualValue = scaledValue / scaleFactor - scaleOffset

Synopsis

#include <skmath.h>

template<class T>
void SKArrayDecode(SKArray<T>& array);

Description

Function to undo the encoding done in *SKArrayEncode()*. That is, convert the *array* values from scaled values to actual 'real world' values. The formula is:

actualValue = scaledValue / scaleFactor - scaleOffset

The *scaleFactor* and *scaleOffset* are obtained from the array object itself (member variables '*scale*' and '*offset*'). Hence these variables must already have been set. *NIL* values are left unchanged. The array's data buffer is overwritten with the unscaled data, and the array's '*scale*' and '*offset*' are set to 1 and 0, respectively.

Currently only handles 1D and 2D *SKArrays*.

Returns

The input *array* values are overwritten with the 'decoded' (e.g. real-world) values, and the *scale* and *offset* member variables of array are set to 1 and 0, respectively.

Document Revision Date

17 August, 1998

SKArrayDecodePixel()

Name

SKArrayDecodePixel()
Function to return the decoded value of a (possibly) scaled input pixel of the input array.

Synopsis

#include <skmath.h>

template<class T>
float SKArrayDecodePixel(SKArray<T>& array);

Description

Function to return the decoded value of a (possibly) scaled input pixel of the input *array*. The pixel value at location (*x*, *y*) in the input *array* is decoded according to the '*scale*' and '*offset*' member variables of the input array. The decoded value is returned as a floating-point number. If the input value is *NIL*, (floating-point) *NIL* is returned. If the array is unscaled, the actual pixel value is returned, but as a float. The equation for the decoding is:

actualValue = scaledValue / scaleFactor - scaleOffset

Currently only handles 1D and 2D *SKArrays*.

Returns

The decoded (e.g. unscaled) value of the pixel at location (*x*, *y*) in *array* is returned as a float.

Document Revision Date

17 August, 1998

SKArrayEncode()

Name

SKArrayEncode()

Function to encode an array of 'true' (unscaled) values into scaled values, via the formula

*scaledValue = (actualValue + scaleOffset) * scaleFactor.*

Note that the desired scaling factors for the array must have been previously set (e.g. via function SKArray<T>::SetScaling()).

Synopsis

#include <skmath.h>

template<class T>
void SKArrayEncode(SKArray<T>& array);

Description

Function to encode an array of 'true' (unscaled) values into scaled values, via the formula

*scaledValue = (actualValue + scaleOffset) * scaleFactor.*

The *scaleFactor* and *scaleOffset* are obtained from the array's *scale* and *offset* member variables. This function treats the input as if it were unscaled (member variables *scale* = 1 and *offset* = 0). To rescale an array which has already been scaled by a different *scale* and *offset*, use function *SKArrayRescale()*.

Currently only handles 1D and 2D *SKArrays*.

Returns

The input *array* values are overwritten with the 'encoded' values computed as described above.

Document Revision Date

17 August, 1998

SKArrayEncodePixel()

Name

SKArrayEncode()

Function to encode a single 'real-world' pixel value into the scaled value stored inside an SKArray via the formula

*scaledValue = (actualValue + scaleOffset) * scaleFactor.*

Note that the desitred scaling factors for the array must have been previously set (e.g. via function SKArray<T>::SetScaling()).

Synopsis

#include <skmath.h>

template<class S, class T>
S SKArrayEncodePixel(SKArray<S>& array, T pixelValue)

Description

Doubly-templatized function to take an input *array* and a scalar *pixelValue* (considered to be an unscaled or 'real-world' value) and scale the *pixelValue* according to the *scale* and *offset* of the input array. Input *pixelValue* is of type T; the returned scaled value is of type S, as is the input *array* from which the scale factors are taken. For instance, if we scale a float *pixelValue* according to scale factors from an *SKArray*<short> *array*, the result is returned as a short. The decoding formula is

*scaledValue = (pixelValue + scaleOffset) * scaleFactor.*

The *scaleFactor* and *scaleOffset* are obtained from the array's *scale* and *offset* member variables. This function will work for 1D, 2D, and 3D arrays as we only need reference the array's '*scaleOffset*' and '*scaleFactor*' member variables; no values are read from the data buffer.

Returns

The encoded value of the supplied *pixelValue*, returned with the same data type as the input *array*.

Document Revision Date

24 August, 1998

SKArrayRescale()

Name

SKArrayRescale()
Function to rescale (and reoffset) an SKArray which may or may not have already been scaled and offset. (The 'scale' and 'offset' of the array will be 1 and 0, respectively, if the array has not previously been scaled.)

Synopsis

```
#include <skmath.h>

template<class T>
void SKArrayRescale( SKArray<T>& array, float newScaleFactor,
                                        float newScaleOffset )
```

Description

Function to rescale (and reoffset) an *SKArray* which may or may not have already been scaled and offset. (The '*scale*' and '*offset*' of the array will be 1 and 0, respectively, if the array has not previously been scaled.)

The unscaling / rescaling pixel operations are performed as floats to preserve digits; only when the newly-rescaled value is stored back in the array do we cast it to type T (the type of data in the original input array). The rescaling is NOT done via a call to *SKArrayDecode*() followed by a call to *SKArrayEncode*(), as that method could lose digits.

The original input *array's* data buffer is overwritten with the newly-rescaled data, and the array's '*scale*' and '*offset*' member variables are updated accordingly.

Currently only handles 1 and 2D arrays, not 3D.

Returns

The input *array* values are overwritten with the rescaled values, and the *scale* and *offset* member variables of array are set to *newScaleFactor* and *newScaleOffset*, respectively.

Document Revision Date

24 August, 1998

SKArraysDifferAtPixels()

Name

SKArraysDifferAtPixels()
Function to take two SKArrays (presumably with the same type data) and print (to the standard output) all coordinates at which the arrays differ by more than the supplied tolerance.

Synopsis

```
#include <skmath.h>

void SKArraysDifferAtPixels( SKArray<T>& input1, SKArray<T>& input2,
                              double tolerance );
```

Description

Function to take two *SKArrays*, *input1* and *input2* (presumably with the same type data) and print (to the standard output) all coordinates at which the arrays differ by more than the supplied *tolerance*. In addition to the coordinates, the values of each array at the conflicting pixels will be printed. If checking for exact equality of, say, int or short arrays, simply choose a tolerance of less than 1.0.

Note that the current slice values of each array are compared, thus not necessarily the full parent slices of each array.

Handles 1, 2, and 3D SKArrays.

Returns

The pixel locations at which *input1* and *input2* fail to match within *tolerance* are printed to the standard output. In addition to the coordinates, the values of each array at the conflicting pixels are printed.

Document Revision Date

17 August, 1998

9. Mathematical Morphology

9.1 Summary

Standard mathematical morphology functions for image processing. Dilation, Erosion, Open, and Close for both GrayScale and Binary images.

9.2 Functions

Mathematical morphology functions begin on the following page.

BinaryClose()

Name

BinaryClose()
Function to perform a binary 'closure' (binary 'dilate' followed by binary 'erode' operation) on a supplied input image.

Synopsis

```
#include <skarray.h>

template <class T>
SKArray<T> BinaryClose( SKArray<T> &in, SKArray<T> &kernel,
                                    T threshold );
```

Description

The binary close operation finds, for each location in the input array, the binary dilation by the structuring element (kernel), followed by the binary erosion with the structuring element. A symmetrical kernel is assumed.

The input array *in* is first binarized according to the supplied *threshold* (values >= the *threshold* are set to 1, values < *threshold* (and *nil* values) are set to zero). Note that this modifies the input array *in* which is passed by reference.

A new *SKArray* of the same size and type of the input array is returned. The new *SKArray* holds the closed (i.e. Dilated and Eroded) array.

Note that the sizes of the kernel must be odd in each dimension, e.g. 3-by-3, 3-by-5, etc. This function is implemented to handle 1, 2, or 3D arrays but has not been tested on 3-dimensional data.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the closed (i.e. Dilated and Eroded) array.

Warning

The input array *in* will be binarized according to the supplied *threshold.* Pass a copy of an input array to this function, rather than the array itself, if you wish to keep the input data unchanged.

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

BinaryDilate(), BinaryErode(), BinaryOpen()

Document Revision Date

19 August, 1998

BinaryDilate()

Name

BinaryDilate()
Function to perform a binary dilation on a supplied input image.

Synopsis

```
#include <skarray.h>

template <class T>
SKArray<T> BinaryDilate( SKArray<T> &in, SKArray<T> &kernel,
                         T threshold );
```

Description

The binary dilation operation finds, for each location in the input array *in*, the maximum value of binary image values within the region of support of the *kernel*. The input array *in* is first binarized according to the supplied *threshold* (values >= the *threshold* are set to 1, values < *threshold* (and *nil* values) are set to zero). Note that this modifies the input array which is passed by reference.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the binary dilated array.

Warning

The input array *in* will be binarized according to the supplied *threshold*. Pass a copy of an input array to this function, rather than the array itself, if you wish to keep the input data unchanged.

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

BinaryClose(), BinaryErode(), BinaryOpen()

Document Revision Date

19 August, 1998

BinaryErode()

Name

BinaryErode()
Function to perform a binary erosion on a supplied input image.

Synopsis

#include <skarray.h>

template <class T>
SKArray<T> BinaryErode(SKArray<T> &in, SKArray<T> &kernel,
T threshold);

Description

The binary erosion operation finds, for each location in the input array *in*, the minimum value of binary image values within the region of support of the *kernel*. The input array *in* is first binarized according to the supplied *threshold* (values >= the *threshold* are set to 1, values < *threshold* (and *nil* values) are set to zero). Note that this modifies the input array which is passed by reference.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the binary eroded array.

Warning

The input array *in* will be binarized according to the supplied *threshold.* Pass a copy of an input array to this function, rather than the array itself, if you wish to keep the input data unchanged.

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

BinaryClose(), BinaryDilate(), BinaryOpen()

Document Revision Date

19 August, 1998

BinaryOpen()

Name

BinaryOpen()
Function to perform a binary 'open' (binary 'erode' followed by binary 'dilate' operation) on a supplied input image.

Synopsis

#include <skarray.h>

```
template <class T>
SKArray<T> BinaryOpen( SKArray<T> &in, SKArray<T> &kernel,
                        T threshold );
```

Description

The binary open operation finds, for each location in the input array, the binary erosion by the structuring element (kernel), followed by the binary dilation with the structuring element. A symmetrical kernel is assumed.

The input array *in* is first binarized according to the supplied *threshold* (values >= the *threshold* are set to 1, values < *threshold* (and *nil* values) are set to zero). Note that this modifies the input array *in* which is passed by reference.

A new *SKArray* of the same size and type of the input array is returned. The new *SKArray* holds the opened (i.e. Eroded and Dilated) array.

Note that the sizes of the kernel must be odd in each dimension, e.g. 3-by-3, 3-by-5, etc. This function is implemented to handle 1, 2, or 3D arrays but has not been tested on 3-dimensional data.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the opened (i.e. Eroded and Dilated) array.

Warning

The input array *in* will be binarized according to the supplied *threshold*. Pass a copy of an input array to this function, rather than the array itself, if you wish to keep the input data unchanged.

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

BinaryClose(), BinaryDilate(), BinaryErode()

Document Revision Date

19 August, 1998

GrayScaleClose()

Name

GrayScaleClose()
Function to perform a gray scale 'closure' (gray scale 'dilate' followed by gray scale 'erode' operation) on a supplied input image.

Synopsis

```
#include <skarray.h>

template <class T>
SKArray<T> GrayScaleClose( SKArray<T> &in, SKArray<T> &kernel );
```

Description

The gray scale close operation finds, for each location in the input array, the gray scale dilation by the structuring element (kernel), followed by the gray scale erosion with the structuring element. A symmetrical kernel is assumed.

A new *SKArray* of the same size and type of the input array is returned. The new *SKArray* holds the (gray scale) closed (i.e. Dilated and Eroded) array.

Note that the sizes of the kernel must be odd in each dimension, e.g. 3-by-3, 3-by-5, etc. This function is implemented to handle 1, 2, or 3D arrays but has not been tested on 3-dimensional data.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the (gray scale) closed (i.e. Dilated and Eroded) array.

Warning

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

GrayScaleDilate(), GrayScaleErode(), GrayScaleOpen()

Document Revision Date

19 August, 1998

GrayScaleDilate()

Name

GrayScaleDilate()
Function to perform a gray scale dilation on a supplied input image.

Synopsis

#include <skarray.h>

template <class T>
SKArray<T> GrayScaleDilate(SKArray<T> &in, SKArray<T> &kernel);

Description

The gray scale dilation operation finds, for each location in the input array *in*, the maximum of each sum of a *kernel* value and superimposed image value within the region of support of the *kernel*. This maximum is assigned to the corresponding location in the output image. The newly-created array, the gray-scale dilated array, is returned.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the gray-scale dilated array.

Warning

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

GrayScaleClose(), GrayScaleErode(), GrayScaleOpen()

Document Revision Date

19 August, 1998

GrayScaleErode()

Name

GrayScaleErode()
Function to perform a gray scale erosion on a supplied input image.

Synopsis

```
#include <skarray.h>

template <class T>
SKArray<T> GrayScaleErode( SKArray<T> &in, SKArray<T> &kernel );
```

Description

The gray scale erosion operation finds, for each location in the input array *in*, the minimum of each sum of a *kernel* value and superimposed image value within the region of support of the *kernel*. This minimum is assigned to the corresponding location in the output image. The newly-created array, the gray-scale eroded array, is returned.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the gray-scale eroded array.

Warning

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

GrayScaleClose(), GrayScaleDilate(), GrayScaleOpen()

Document Revision Date

19 August, 1998

GrayScaleOpen()

Name

GrayScaleOpen()
Function to perform a gray scale 'open' (gray scale 'erode' followed by gray scale'open' operation) on a supplied input image.

Synopsis

#include <skarray.h>

template <class T>
SKArray<T> GrayScaleOpen(SKArray<T> &in, SKArray<T> &kernel);

Description

The gray scale open operation finds, for each location in the input array, the gray scale erosion by the structuring element (kernel), followed by the gray scale dilation with the structuring element. A symmetrical kernel is assumed.

A new *SKArray* of the same size and type of the input array is returned. The new *SKArray* holds the (gray scale) opened (i.e. Eroded and Dilated) array.

Note that the sizes of the kernel must be odd in each dimension, e.g. 3-by-3, 3-by-5, etc. This function is implemented to handle 1, 2, or 3D arrays but has not been tested on 3-dimensional data.

Returns

A new *SKArray* of the same size and type of the input array *in* is returned. The new *SKArray* holds the (gray scale) opened (i.e. Eroded and Dilated) array.

Warning

The supplied *kernel* is expected to have symmetrical data about its center. i.e. in coordinates (x, y) relative to the center of the kernel, if kernel(x, y) = val, then kernel(-x, -y) must = val.

See Also

GrayScaleClose, GrayScaleDilate(), GrayScaleErode()

Document Revision Date

19 August, 1998

10. Matrix Operations

10.1 Summary

Standard matrix operations, where a 2-D *SKArray* is considered a 'matrix' in the mathematical sense. Operations include matrix inversion, 'LU' decomposition and back substitution, and singular value decomposition.

10.2 Functions

Matrix operation descriptions begin on the following page.

SKArrayInvert()

Name

SKArrayInvert()
Function to invert a matrix (2D SKArray).

Synopsis

#include <skmatrix.h>

SKArray<float> SKArrayInvert(SKArray<float>& input);

Description

Function to invert a matrix. This method first computes the LU decomposition of a matrix and then solves for the inverse column-by-column by solving Ax = e(i), where e(i) is the i'th column of the identity matrix.

Returns

The inverse matrix, as an *SKArray<float>*.

Note

The first step of the inversion process is to compute the LU decomposition of the *input* array. This will overwrite *input* with the LU decomposition. If this is not desired, a deep copy (e.g. as returned by *SKArray<T>::DeepCopy()*) should be made of the *input* and the copy passed in to this routine.

The inversion function may be numerically unstable for arrays which are nearly singular. If the potential for a badly-conditioned matrix exists, then a singular value decomposition / perturbation method should be applied to the input to produce a better conditioned matrix. See function *CreateOeCovarianceMatrix()* (in the MIGFA code repository) for an example.

Since all of the CSKETCH 'matrix' functions (*SKArrayInvert, SKArrayLUBacksub, SKArrayLUDecomp, and SKArraySVDecomp*) are taken from 'Numerical Recipes in C', the 0th row and column of all returned arrays is to be ignored, e.g. if the inverse of an n-by-n array is desired, that array must first be copied to an (n+1) by (n+1) array with zeroes in the 0th row and column. A new (n+1 by n+1) array will be returned; the n-by-n inverse will be returned in the n-by-n subarray with subscripts 1 through n of the larger array.

Warning

As mentioned above in the *Note* section, the input array must be prepadded with a row and column of zeroes to account for 'Numerical Recipes in C' idiosyncracies.

Acknowledgment

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, Numerical Recipes in C: The Art of Scientific Computing, 2nd ed. (Cambridge University Press, Cambridgeshire, U.K., 1992).

Document Revision Date

17 July, 2002

SKArrayLUBacksub()

Name

SKArrayLUBacksub()
Function to perform the the back-substitution needed to solve a system of linear equations after LU decomposition.

Synopsis

#include <skmatrix.h>

void SKArrayLUBacksub(SKArray<float>& LUArray,
SKArray<short>& index, SKArray<float>& vector);

Description

Function to perform the the back-substitution needed to solve a system of linear equations after LU decomposition. E.g. to solve Ax = b we first reduce A to LU form, via *SKArrayLUDecomp*(). The results of that function call can then be passed to this function to solve systems of linear equations. Input *vector* (the 'b' vector or the right hand side of the set of linear equations) is overwritten by the solution vector x. Specifically, the returned LU matrix as well as the *index* vector returned by *SKArrayLUDecomp*() are needed by *SKArrayLUBacksub*().

Returns

The 'right hand side' *vector* (the vector 'b' in Ax = b) is overwritten with the solution vector x.

Note

Since all of the CSKETCH 'matrix' functions (*SKArrayInvert, SKArrayLUBacksub, SKArrayLUDecomp, and SKArraySVDecomp*) are taken from 'Numerical Recipes in C', the 0th row and column of input arrays is to be ignored, e.g. for this function, the 0th row of the column input *vector* must be zero (e.g. if the desired right-had side is (1, 1, 1), then the actual input vector should be (0, 1, 1, 1).

Warning

As mentioned above in the *Note* section, the input *vector* must be prepadded with a row of zeroes to account for 'Numerical Recipes in C' idiosyncracies.

Acknowledgement

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, Numerical Recipes in C: The Art of Scientific Computing, 2nd ed. (Cambridge University Press, Cambridgeshire, U.K., 1992).

Document Revision Date

18 November, 1998

SKArrayLUDecomp()

Name

SKArrayLUDecomp()
Routine to take a matrix A and compute the 'LU' decomposition of the matrix.

Synopsis

#include <skmatrix.h>

*void SKArrayLUDecomp(SKArray<float>& array, SKArray<short>& index, SKArray<float>& scales, int *numInterchanges);*

Description

Routine to take a matrix A and compute the 'LU' decomposition of the matrix A (i.e. A = L * U where L is lower diagonal and U is upper diagonal). This decomposition can then be used to efficiently solve linear equations of the type Ax = b.

Returns

All 4 input arguments are updated. The input *array* is overwritten with the LU decomposition of A ('L' is stored in the lower triangular half and 'U' is stored in the upper triangular half). The *index* array records the row permutations effected on the *input* array in the process of computing the LU decomposition (this routine uses partial pivoting for numerical stability). The *scales* array stores the factors by which each row is scaled during the LU decomposition process. Finally, *numInterchanges* is set to the parity of the number of row intechanges needed in the LU process -- +1 for an even number and -1 for an odd number of interchanges.

Note

The *input* array will be overwritten with its LU decomposition. If this is not desired, a deep copy (e.g. as returned by *SKArray<T>::DeepCopy()*) should be made of the *input* and the copy passed in to this routine.

The returned *index* variable will be needed by any subsequent calls made to *SKArrayLUBacksub*(). This is the routine which uses the LU decomposition of an array A to efficiently solve the matrix equation Ax = b.

Since all of the CSKETCH 'matrix' functions (*SKArrayInvert, SKArrayLUBacksub, SKArrayLUDecomp, and SKArraySVDecomp*) are taken from 'Numerical Recipes in C', the 0th row and column of all returned arrays is to be ignored, e.g. if the inverse of an n-by-n array is desired, that array must first be copied to an (n+1) by (n+1) array with zeroes in the 0th row and column. A new (n+1 by n+1) array will be returned; the n-by-n inverse will be returned in the n-by-n subarray with subscripts 1 through n of the larger array.

Warning

As mentioned above in the *Note* section, the input array must be prepadded with a row and column of zeroes to account for 'Numerical Recipes in C' idiosyncracies.

Acknowledgement

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, Numerical Recipes in C: The Art of Scientific Computing, 2nd ed. (Cambridge University Press, Cambridgeshire, U.K., 1992).

Document Revision Date

17 July, 2002

SKArraySVDecomp()

Name

SKArraySVDecomp()
Routine to take a matrix A and compute the singular value decomposition of the matrix.

Synopsis

#include <skmatrix.h>

```
void SKArraySVDecomp( SKArray<double>& array,
                      SKArray<double>& wmatrix,
                      SKArray<double>& vmatrix,
                      SKArray<double>& scales );
```

Description

Routine to compute the singular value decomposition of a matrix. The SVD of a matrix A is a way of 'factoring' the matrix into the form
A = U * W * hermitian(V), where the matrices U, W, and V have significant mathematical properties. This routine overwrites the input matrix A to be the output matrix U. Pre-allocated matrices for W and V are filled in by this routine. Also, a pre-allocated array 'scales', for holding various scale factors employed by the routine, is filled in by this routine.

Returns

All 4 inputs are edited by the call to this function. The input *array* is overwritten with the U matrix of the SVD of A (so if A will be needed later, pass in a deep copy of A rather than A itself). The pre-allocated *wmatrix* and *vmatrix* arrays are filled in with the W and V matrices of the SVD. Finally, 'scales' is filled in with various scale factors used in computing the SVD.

Note

The input *array* will be overwritten with the 'U' matrix of the SV decomposition. If this is not desired, a deep copy (e.g. as returned by *SKArray<T>::DeepCopy()*) should be made of *array* and the copy passed in to this routine.

Since all of the CSKETCH 'matrix' functions (*SKArrayInvert, SKArrayLUBacksub, SKArrayLUDecomp, and SKArraySVDecomp*) are taken from 'Numerical Recipes in C', the 0th row and column of all input and returned arrays is to be ignored, e.g. if the inverse of an n-by-n array is desired, that array must first be copied to an (n+1) by (n+1) array with zeroes in the 0th row and column. A new (n+1 by n+1) array will be returned; the n-by-n inverse will be returned in the n-by-n subarray with subscripts 1 through n of the larger array. Likewise the 0th row and column of all input arrays must be initialized to 0 and the true data of an n-by-n array moved to an n-by-n subarray with indices 1 through n rather than 0 through n-1.

Warning

As mentioned above in the *Note* section, the input array must be prepadded with a row and column of zeroes to account for 'Numerical Recipes in C' idiosyncracies.

Acknowledgement

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, Numerical Recipes in C: The Art of Scientific Computing, 2nd ed. (Cambridge University Press, Cambridgeshire, U.K., 1992).

Document Revision Date

17 July, 2002

11. Miscellaneous Functions

11.1 Summary

Some miscellaneous useful functions for operating on *SKArrays.*

11.2 Functions

Miscellaneous function description begins on the following page.

SetElementsInInterval()

Name

SetElementsInInterval()
Function to set all elements of the array which lie within some interval to a mark value, possibly setting outliers to a different mark value.

Synopsis

```
#include <skarray.h>

template<class T, class V>
void SetElementsInInterval( SKArray<T> & input,
                            const SKArray<V> & mask,
                            T markVal, V low, V high,
                            T outlierMarkVal = SKArray<T>::SK_NIL );
```

Description

The *SetElementsInInterval* functions are used to reset some (possibly all) elements of the *input* array to 'mark' and / or 'outlierMark' values, based on the pixel values of the *mask* array. Specifically, if the pixel with coordinates (x, y) in the *mask* image has a value between *low* and *high* (inclusive), then the pixel at (x, y) in the *input* image is set equal to *markVal.* If *outlierMarkVal* is not *NIL,* and the pixel at (x, y) in the *mask* image lies outside the interval from *low* to *high,* then the pixel at (x, y) in the *input* image is set to *outlierMarkVal.* If the outlier mark is *NIL,* then pixels in the *input* image will not be changed at locations (x, y) where the mask pixel value is not in the interval from *low* to *high.* The default value of optional argument *outlierMarkVal* is *NIL.* See the ***CSKETCH Image Processing Library Tutorial*** for usage examples.

Returns

The *input* array, which is passed in by reference, may have some of its data values modified as described above.

See Also

Member function *SKArray<T>::SetElementsInInterval().*

Document Revision Date

21 *September, 1998*

SetMissingToNearest()

Name

SetMissingToNearest()
Function which attempts to fill a missing (nil) array value by computing a value based on the values of the nearest neighbors. This function does this for one pixel; it may be used to attempt to fill in all missing array values via function SKArray<T>::Apply().

Synopsis

#include <skarray.h>

```
T SetMissingToNearest( SKArray<T>& input, T* in_xp, int x, int y,
                       void*grayScale);
```

Description

This function is called to attempt to find a missing array value by computing a value based on its nearest neighbors. This function does not necessarily find a value for every *NIL* value. It only will return a value if one has been found within the limits of the search. If no replacement candidates are found, the function returns the appropriate value of *NIL* for the type of data of the input array. Otherwise, the function returns the replacement value (but does not acually set the pixel to this replacement value).

The *x* and *y* inputs give the *(x, y)* coordinates of the missing value in the input array; the pointer *in_xp* points to the *NIL* data element in the *input* array. As mentioned above, only this one *NIL* value will attempt to be filled; the member function *SKArray<T>::Apply()* can be used with this function to attempt to fill all *NIL* values. In the event that multiple nearest non-*NIL* neighbors are found (i.e. the choice for a replacement value is ambiguous) then either the average value or a rounded average value will replace the missing value. If the *grayScale* argument is not *NULL*, the average is used; otherwise a rounded average is used.

Returns

The candidate replacement value for the *NIL* value. Again, this function itself will not automatically overwrite the missing value in the *input* array with the replacement value.

See Also

Member function *SKArray<T>::Apply()*.

Document Revision Date

21 *September, 1998*

SKArrayNearEqual()

Name

SKArrayNearEqual()
This function determines whether all pixels in the data buffers of two arrays are numerically equal within a supplied tolerance.

Synopsis

#include <skarray.h>

int SKArrayNearEqual(SKArray<T>& input1, SKArray<T>& input2,
double tolerance);

Description

This function determines whether all pixels in the data buffers of two arrays are numerically equal within the supplied *tolerance.*

Returns

Returns 0 if the *SKArrays* are not nearly equal (pixelwise), 1 if the *SKArrays* are pixelwise equal within the supplied *tolerance.*

Document Revision Date

22 *September, 1998*

SKThin()

Name

SKThin()

CSKETCH image thinning implementation. All distinct, connected shapes within the input image are reduced to chains a single pixel wide.

Synopsis

#include <skarray.h>

SKArray<short> SKThin(SKArray<short>& input);

Description

Functional Template Implementation of Levialdi's homotopic thinning, with an additional post thinning step. All of the shapes in an input image are skeletonized, reducing them to chains a single pixel wide.

Returns

A new *SKArray<short>*, the thinned array.

Document Revision Date

22 *September, 1998*

SliceSizeEqual()

Name

SliceSizeEqual()
This function determines if two arrays have equal slice sizes (i.e. have the same number of array dimensions and the same size in each dimension).

Synopsis

#include <skarray.h>

int sliceSizeEqual (const SKArray<T> &, const SKArray<T> &);

Description

This function determines if two arrays have equal slice sizes (i.e. have the same number of array dimensions and the same size in each dimension).

Returns

Returns 0 if the *SKArrays* are not equal in slice size, 1 if the *SKArrays* are equal in slice size.

Document Revision Date

22 *September, 1998*

12. Region Analysis

12.1 Summary

A set of functions for generating statistics on various 'regions' within the data array of a (1D or 2D only) integer *SKArray*. The data must be short, or integer, because of the way regions are identified within *SKArrays*. A region inside an *SKArray* is defined as the set of all pixels within the data buffer which have the same value. Thus all pixels with value 1 comprise region 1, all pixels with value 2 comprise region 2, etc. Note that individual regions need not be connected (by connected we mean that all pixels of the region touch at least one other pixel of the region, whether horizontally, vertically, or diagonally). However, region analysis can be constrained to connected regions only by use of the *SKLabelRegions*() function, which takes an input *SKArray* and returns a new array with all distinct, connected regions assigned a distinct label (i.e. all pixels in each region have a common value, and that value is unique to that region). Function *SKLabelRegions()* is described in this section. The main driver function for region analysis, *SKRegionSummary*(), does not assume that all regions are connected; rather it assumes all pixels with the same value belong to the same region. If region analysis is desired to take place only on connected regions, first create a new array via a call to *SKLabelRegions()* and then pass the new array into function *SKRegionSummary()*.

12.2 Conventions

The region analysis functions work closely with the *SKRegionInfo* class and the associated *SKRegion* structure. Specifically, function *SKRegionSummary()* returns a pointer to an *SKRegionInfo* object. The returned *SKRegionInfo* object contains an array of *SKRegion* objects, one for each region in the input *SKArray*. This array of *SKRegion* objects is the member variable known as '*region*' in the *SKRegionInfo* object. Statistics for the *i'th* region of data within an *SKArray* will be stored in the *i'th* element of the array of *SKRegion* objects. Again, the *i'th* region of an *SKArray* is the (connected or disconnected) set of all points with value *i*. Thus, if we set

*SKRegionInfo *regionInfo = SKRegionSummary(inputArray);*

we would access the length of region 7 of *inputArray* as

(regionInfo->region)[7].length;

Simliarly, the area of region 2 would be accessed as

(regionInfo->region)[2].area;

12.3 Functions

Region analysis functions begin on the following page.

EliminatePoorShapes()

Name

EliminatePoorShapes()
Function to eliminate all connected regions inside the input array which fail a minimum length criterion.

Synopsis

#include <skregion.h>

```
SKArray<short> EliminatePoorShapes( SKArray<short>& input,
                    short inputThresh, float lengthThresh, short fillOpt );
```

Description

Function to eliminate all regions inside the input array which fail a length criterion. Specifically, a copy of the *input* array is first thresholded to eliminate background noise (values strictly less than *inputThresh* are set to 0, values greater than or equal to *inputThresh* are set to 1). The thresholded image is then labelled into distinct, connected regions by function *SKLabelRegions*(). Region analysis is then performed via a call to *SKRegionSummary()*. Regions which fail a minimum length criterion (length < *threshLength*) are deleted by setting all their pixels to zero. The argument *fillOpt* indicates whether some additional image preprocessing should be done prior to the region length thresholding. If *fillOpt* is true the binarized, thresholded interest image will be dilated using a 3 by 3 elliptical kernel, then closed with a 5 by 5 elliptical kernel.

Returns

The final thresholded and length thresholded image. Distinct regions which passed the length threshold will all be marked in the output image (e.g. all pixels in the acceptable regions will have the same value). All other pixels (e.g. those belonging to regions which are too short) will be set to 0.

Document Revision Date

3 December, 1998

SKLabelRegions()

Name

SKLabelRegions()

Routine to take a (short) input SKArray and mark all distinct, connected, non-zero regions of data in the input data with distinct short integer labels.

Synopsis

#include <skregion.h>

SKArray<short> SKLabelRegions(SKArray<short>& input)

Description

This routine takes a (short) *input* array and marks all distinct, connected, nonzero regions of data in the *input* array with distinct labels. The *input* array values are not changed -- rather the routine creates a new *SKArray<short> output* array (the same size as the *input* array), copies the *input* data to the *output* array, and then performs the marking on the *output* array. Values of zero in the input image are considered 'background' values and are ignored; there is no region 0.

Returns

A newly-created *SKArray*, the same size as the *input* array. In effect, all distinct, connected, nonzero regions of *input* are copied to the corresponding pixels in *output*, and assigned a unique label (e.g. a common pixel value for each pixel in the region).

Document Revision Date

24 August, 1998

SKRegionArea()

Name

SKRegionArea()

Function to compute the area of each region in the input array. Area of a region is defined to be the number of pixels belonging to that region.

Synopsis

#include <skregion.h>

*void SKRegionArea(SKArray <short>& array, SKRegionInfo *regionInfo);*

Description

Function to compute the area of each region in the input array. Area of a region is defined to be the number of pixels belonging to that region. Note that the regions need not be connected. If statistics are desired for connected regions within an *SKArray<short>,* be sure to first call *SKLabelRegions*() to get a new *SKArray* in which all pixels within connected regions share the same pixel value.

The input *SKRegionInfo* object must have been constructed with a sufficient number of *SKRegion* objects prior to the call to this image. For this reason, it is preferable to simply call function *SKRegionSummary()* rather than *SKRegion-Area(). SKRegionSummary()* will compute many additional statistics about regions, and thus executes slightly more slowly, but *SKregionSummary()* is more robust in that it dynamically determines how many *SKRegion* objects will be needed inside the *SKRegionInfo* object to store statistics about all regions.

Implementation

For efficiency, this function and all related *SKRegion* functions compute region statistics 'in parallel' for all regions within an *SKArray.* This is necessary as there may be several hundred regions of interest within, e.g. a MIGFA interest image. Without the parallel implementation, algorithm latency requirements might not be met.

Returns

The *i'th* object in the array of *SKRegion* structures (stored inside the *SKRegion-Info* object) is filled in with the area (number of pixels) of region *i.* Region *i* of the input *array* is defined to be the set of all pixels with value *i.*

Document Revision Date

3 December, 1998

SKRegionBoundingRect()

Name

SKRegionBoundingRect()

Function to compute, for each region in the labelled input array, the coordinates of a bounding box for the region. Specifically, the lower left and upper right corners of the box are computed and stored in corresponding SKRegionInfo objects.

Synopsis

#include <skregion.h>

```
void SKRegionBoundingRect( SKArray <short>& array,
                           SKRegionInfo *regionInfo )
```

Description

Function to compute, for each region in the labelled input array, the coordinates of a bounding box for the region. Specifically, the lower left and upper right corners of the box are computed and stored. Note that the regions need not be connected. If bounding boxes are desired for connected regions within an *SKArray<short>,* be sure to first call *SKLabelRegions*() to get a new *SKArray* in which all pixels within connected regions share the same pixel value.

The input *SKRegionInfo* object must have been constructed with a sufficient number of *SKRegion* objects prior to the call to this image. For this reason, it is preferable to simply call function *SKRegionSummary()* rather than *SKRegion-Boundingrect(). SKRegionSummary()* will compute many additional statistics about regions, and thus executes slightly more slowly, but *SKregionSummary()* is more robust in that it dynamically determines how many *SKRegion* objects will be needed inside the *SKRegionInfo* object to store statistics about all regions.

Implementation

For efficiency, this function and all related *SKRegion* functions compute region statistics 'in parallel' for all regions within an *SKArray.* This is necessary as there may be several hundred regions of interest within, e.g. a MIGFA interest image. Without the parallel implementation, algorithm latency requirements might not be met.

Returns

The *i'th* object in the array of *SKRegion* structures (stored inside the *SKRegion-Info* object) is filled in with the bounding box (encoded as the lower left and upper right corners of the box) for region *i*. Region *i* of the input *array* is defined to be the set of all pixels with value *i*.

Document Revision Date

3 December, 1998

SKRegionCenterOfGravity()

Name

SKRegionCenterOfGravity()
Function to compute, for each region in the labelled input array, the coordinates of the center of gravity of the region.

Synopsis

#include <skregion.h>

```
void SKRegionCenterOfGravity( SKArray <short>& array,
                              SKRegionInfo *regionInfo );
```

Description

Function to compute the center of gravity of each region in the input array. Note that the regions need not be connected. If statistics are desired for connected regions within an *SKArray<short>*, be sure to first call *SKLabelRegions*() to get a new *SKArray* in which all pixels within connected regions share the same pixel value.

The input *SKRegionInfo* object must have been constructed with a sufficient number of *SKRegion* objects prior to the call to this image. For this reason, it is preferable to simply call function *SKRegionSummary()* rather than *SKRegion-CenterOfGravity()*. *SKRegionSummary()* will compute many additional statistics about regions, and thus executes slightly more slowly, but *SKregionSummary()* is more robust in that it dynamically determines how many *SKRegion* objects will be needed inside the *SKRegionInfo* object to store statistics about all regions.

Implementation

For efficiency, this function and all related *SKRegion* functions compute region statistics 'in parallel' for all regions within an *SKArray*. This is necessary as there may be several hundred regions of interest within, e.g. a MIGFA interest image. Without the parallel implementation, algorithm latency requirements might not be met.

Returns

The *i'th* object in the array of *SKRegion* structures (stored inside the *SKRegion-Info* object) is filled in with the center of gravity of region *i*. Region *i* of the input *array* is defined to be the set of all pixels with value *i*.

Document Revision Date

3 December, 1998

SKRegionSummary()

Name

SKRegionSummary()
Top-level routine for computing region statistics. Calls all SKRegion functions in the proper sequence (e.g. center of gravity, area, and other region statistics must be known prior to computing length, etc.) See documentation for the SKRegion class for a description of regions within an SKArray's data buffer as well as a list of all the statistics computed / saved by this routine.

Synopsis

#include <skregion.h>

*SKRegionInfo *SKRegionSummary(SKArray<short>& input)*

Description

Top-level routine for computing region statistics. Regions within an *SKArray<short>* are defined to be the set of all pixels with a common value; thus the set of all pixels with value 1 is region 1, the set of all pixels with value 2 is region 2, etc. Note that regions need not be connected; e.g. if the only pixels with value 1 are at the four corners of an *input* image, then region 1 is still a valid region made up of 4 non-contiguous points. See documentation for the *SKRegion* class for a list of all the statistics computed / saved by this routine.

Implementation

For efficiency, this function and all related *SKRegion* functions compute region statistics 'in parallel' for all regions within an *SKArray*. This is necessary as there may be several hundred regions of interest within, e.g. a MIGFA interest image. Without the parallel implementation, algorithm latency requirements might not be met.

Returns

A pointer to a newly-allocated *SKRegionInfo* object. Internal to the *SKRegion-Info* object is an array of *SKRegion* structures. One such structure will be allocated for each distinct region in the *input* array. Each structure will be filled in with statistical information about a corresponding region. The correspondence between regions and *SKRegion* structures can be obtained from the index of the *SKRegion* structure in the array: the statistics for the *i*th region are stored in the *i*th *SKRegion* structure.

Note

SKRegionSummary() is in a sense the 'master' routine for region analysis. It calls all other major region analysis functions, *SKRegionLength(), SKRegionArea()*, etc. For efficiency purposes, these 'slave' functions make certain assumptions about the array of *SKRegion* structures inside the *SKRegionInfo* object that they are supplied as an argument (i.e. results that are pre-computed in *SKRegionSummary()* prior to the call to a slave function). For this reason, one should not call the 'slave' functions directly; this might involve using uninitialized fields of the *SKRegion* structures and garbage output could result. Only a few of the simpler region analysis functions should be called directly by a user. These include

SKLabelRegions(), *SKRegionArea()*, *SKRegionBoundingRect()*, and *SKRegion-CenterOfGravity()*. Additionally, *EliminatePoorShapes()* may be called directly since it performs its own region analysis (via calls to *SKLabelRegions()* and *SKRegionSummary()*). Other than these functions, none of the region analysis functions should be called directly by a user, so only these functions are described in the 'Region Analysis' section of this document. Consult the in-line source code documentation for further elaboration on the details of the 'in-parallel' region analysis computations.

Document Revision Date

13 October, 1998

13. Weather Radar Tools

13.1 Summary

A set of functions for operating on *SKArrays* which are especially useful in the context of weather radar data analysis.

13.2 Functions

Weather radar data function description begins on the following page.

SKArraySimpleHistogram()

Name

SKArraySimpleHistogram()
A simple (i.e. not particularly general) function to compute a histogram of (the current slice of) an input array. Currently only supported for SKArray<short>.

Synopsis

#include <skwrt.h>

SKArray<int> SKArraySimpleHistogram(SKArray<short> &input);

Description

A simple (i.e. not particularly general) function to compute a histogram of (the current slice of) an input array. For now only supported for *SKArray<short>*. The function will allocate and return an integer *SKArray* which represents the histogram. The size of the histogram's data buffer will be equal to the maximum value of the (short) input array, plus 1; the histogram will be filled so that

histogram(0) = number of occurrences of "0" in *input* array,
histogram(1) = number of occurrences of "1" in *input* array,
etc., up to the maximum value of the input array.

Returns

An SKArray of type int which represents the histogram, computed as above.

Warning

This routine is not well-optimized since it is not anticipated to be used very frequently. Currently handles only 1D and 2D *SKArrays.*

Document Revision Date

30 September, 1998

SKAverageInterestImages()

Name

SKAverageInterestImages()
Function to compute the pixelwise average of a list of input images..

Synopsis

#include <skwrt.h>

SKArray<short> SKAverageInterestImages(LLNIDList &images,
short interestThreshold);

Description

Function to compute the pixelwise average of a list of input images. For MIGFA these images are 'interest'images, i.e. images returned by the various gust front detectors.

The *interestThreshold* (nominally 128 for MIGFA) is used to delineate 'confirming' (positive) evidence of the existence of a particular feature, vs. 'disconfirming' (negative) evidence of the existence of that feature. If a particular pixel value of an image is greater than, or equal to, the *interestThreshold*, then that pixel value will be multiplied by the image's *confirmingWeight* member variable in computing the weighted average at that pixel. Similarly, if the pixel value is below the *interestThreshold* it will be multiplied by the image's *disconfirmingFactor.* Note that each different image in the image list has its own confirming and disconfirming factors which should have been set prior to this function call.

Pixelwise averages are converted to type short for output by rounding. Values of the input images which are *nil*, e.g. missing, are ignored in the averaging process. If at a particular location all input images are *nil*, then the corresponding pixel in the averaged image will also be *nil.*

This function assumes all input images are of type short int, as interest images have short int data. The function can be templatized if averages for images with different data types are needed.

Returns

A 2D *SKArray<short>* which stores the pixelwise averages of the input images.

See Also

Function *SKAverageInterestImagesExceptMin*().

Document Revision Date

30 September, 1998

SKAverageInterestImagesExceptMin()

Name

SKAverageInterestImagesExceptMin()
Function to compute the pixelwise average of a list of input images, excluding the minimum pixel value across all images.

Synopsis

#include <skwrt.h>

```
SKArray<short> SKAverageInterestImagesExceptMin( LLNIDList &images,
                                   short interestThreshold );
```

Description

Function to compute the pixelwise average of a list of input images, excluding the minimum pixel value across all images. For example, if at location (x, y) = (5, 10) of the output image, the input images have values 10, 7, and 6, then only the 10 and 7 value will be used in computing the weighted average for the output pixel at (5, 10).

The *interestThreshold* (nominally 128 for MIGFA) is used to delineate 'confirming' (positive) evidence of the existence of a particular feature, vs. 'disconfirming' (negative) evidence of the existence of that feature. If a particular pixel value of an image is greater than, or equal to, the *interestThreshold*, then that pixel value will be multiplied by the image's *confirmingWeight* member variable in computing the weighted average at that pixel. Similarly, if the pixel value is below the *interestThreshold* it will be multiplied by the image's *disconfirmingFactor*. Note that each different image in the image list has its own confirming and disconfirming factors which should have been set prior to this function call.

Pixelwise averages are converted to type short for output by rounding. Values of the input images which are *nil*, e.g. missing, are ignored in the averaging process. If at a particular location all input images are *nil*, then the corresponding pixel in the averaged image will also be *nil*.

This function assumes all input images are of type short int, as interest images have short int data. The function can be templatized if averages for images with different data types are needed.

Returns

A 2D *SKArray<short>* which stores the pixelwise averages of the input images.

See Also

Function *SKAverageInterestImages*().

Document Revision Date

30 September, 1998

SKCreateCircularMask()

Name

SKCreateCircularMask()
Function to build circular masks for MIGFA, AMDA, etc. The circle is centered at the true center of the output cartesian image.

Synopsis

#include <skwrt.h>

```
void SKCreateCircularMask( SKArray<short> *mask, SKResamp *resamp,
                float maskRadKM, int nAzimuths, int nGates,
                float gateSizeMeters, short maskValue );
```

Description

Function to build circular masks for MIGFA, AMDA, etc. The circle is centered at the true center of the output cartesian image. A pointer to a pre-created *SKResamp* object must be passed in to this routine. The *maskRadKM* is the desired size, in KM, of the circle to be drawn in *mask*. The circle will be filled with value *maskValue*. The *nAzimuths* and *nGates* specify the size of typical polar images being used by the current application (e.g. MIGFA). Finally the *gateSizeMeters* is the size of a gate of polar data, in meters. The circular mask is filled in by creating a polar image, size *nAzimuths* by *nGates*, with all values set to *maskValue* for distances within *maskRadKM* of the origin. All other values are set to *NIL*. After this is performed, the resampler is run on the fabricated polar data, with output to *mask*.

Returns

The *mask* array's data buffer is filled in with the desired circular mask. Note that the *mask* array itself must have been allocated prior to calling this routine.

See Also

Class *SKResamp*

Document Revision Date

29 September, 1998

SKFillSector()

Name

SKFillSector()
Function to take a (short) SKArray and fill in a sector of data with the supplied fillValue. The start and end angle of the sector, as well as the sector's origin and radius are supplied as parameters. The start and end angles are specified in the SKETCH sense ((image) north = 0 degrees, increasing in the clockwise direction).

Synopsis

```
#include <skwrt.h>

void SKFillSector( SKArray<short>& input, double startAngle,
                   double endAngle, double xorigin, double yorigin,
                   short fillValue, double radius );
```

Description

Function to take a (short) *SKArray* and fill in a sector of data with the supplied *fillValue*. The start and end angle of the sector, as well as the sector's origin and radius are supplied as parameters. The start and end angles are specified in the CSKETCH sense ((image) north = 0 degrees, increasing in the clockwise direction).

Returns

Pixels in the *input* array which are within the specified sector are set to *fillValue.*

Warning

Note that a radius value of 0.0 is equivalent to setting the radius to *SKArray<float>::SK_MAXIMUM*, i.e. all radius values are considered in range of the sector.

All supplied angles are first modded to the range -180.0 to +180.0. This is necessary for correct angle range checking. For example, if filling the entire 330 degree sector from 90 degrees all the way around to 60 degrees, an angle of 75 degrees is out of bounds. If this angle were given as 435 degrees, rather than 75, the logic for determining which angles are in bounds would fail.

Document Revision Date

22 September, 1998

SKIntegrateZDimension()

Name

SKIntegrateZDimension()
Function to take a 3-dimensional input image and return a 2-dimensional image obtained by summing along the z direction of the 3-d image.

Synopsis

#include <skwrt.h>

template <class T>
SKArray<T> SKIntegrateZDimension(SKArray<T>& input);

Description

Function to take a 3-dimensional *input SKArray* and return a 2-dimensional *SKArray* obtained by summing along the z direction of the 3-D image. Templatized over *input* image data type. *NIL* values in the input image are ignored; if every input pixel in an entire ‘pillar’ is *NIL*, then the output at the (x,y) location of that pillar is set to *NIL*.

Returns

The 2-D integrated (i.e. summed) array.

Document Revision Date

29 September, 1998

SKPercentOfMassInZDimension()

Name

SKPercentOfMassInZDimension()
Function to compute, for each "pillar" in a 3-D SKArray, the z-coordinate at which the requested percentile of "mass" is attained.

Synopsis

#include <skwrt.h>

template <class T>
SKArray<short> SKPercentOfMassInZDimension(SKArray<T>& input,
float requestedPercent, float resolution);

Description

Function to compute, for each 'pillar' in a 3-D *SKArray*, the z-coordinate at which the requested percentile of 'mass' is attained. That is, at a given (x,y) location, the 'pillar' is summed over z to find the total mass. Then a threshold is computed as requestedPercentile * totalMass. The column is resummed starting at z = 0, continuing until the sum exceeds the threshold. Roughly speaking, the z-coordinate (denoted zthreshold) at which the sum exceeded the threshold is stored at the corresponding (x,y) location in the output image. More accurately, a *resolution* argument is supplied which can refine the simple pixel estimate given above. What is actually stored in the output image is

*SKROUND(resolution * (zthreshold + ((threshold - prev_sum) / curr)));*

where *prev_sum* is the sum just prior to the point where the sum exceeded the threshold, and *curr* is the pixel value which caused the sum to exceed the threshold.

For an example see the Example section for this function (below).

Returns

A 2-dimensional (short) *SKArray* which stores, at each (x, y) location, the quantity described above in the Description section (roughly the pixel at which the sum exceeds the requested percentile).

Example

As an example, consider a pillar whose entries are 1,1, and 0. Suppose the desired percentile is 0.9. The total mass is 2 so the threshold is 0.9 * 2 = 1.8. The sum exceeds the threshold at *z = zthreshold* = 1. In this case *prevSum* would be 1 and the *curr* value is 1. If the resolution were 1, a zcoord equal to 2 would be stored. If the resolution were 100, however, a z-coordinate of *SKROUND(100 + ((1.8 - 1)/1)) = 180* would be stored. Later dividing by the resolution would indicate that the threshold was achieved at z = 1.8.

See also the appropriate test points in the SKE library description to see the effect of changing the resolution.

Document Revision Date	*23 September, 1998*

SKPercentProbabilityDist()

Name

SKPercentProbabilityDist()
Function to find the pixel value in the input SKArray corresponding to the requested percentile of all the pixel values in input. For example, if the requested percent is 90.0, this routine returns the minimum pixel value in the input array which is greater than 90% of all other pixel values.

Synopsis

```
#include <skwrt.h>

float SKPercentProbabilityDist( SKArray<short>& input,
                                float requestedPercent );
```

Description

Function to find the pixel value in *input* corresponding to the *requestedPercent* percentile of all the pixel values in *input.*

A histogram of all the pixel values in *input* is created. Note that *input* is assumed to have data of type short, and that the minimum value is expected to be zero (the *input* array may first be scaled by the calling routine to ensure that this is so). The bin size of the histogram is one, so each discrete value in the (short) input array gets its own bin.

Since the histogram only counts non-*nil* pixels, a sum of the values of the histogram gives the number of non-*nil* pixels. This number is multiplied by the *requestedPercentile* to get a *targetCount.* We then sum over the values in the histogram array until we first exceed the *targetCount.* The current index in the histogram array when we first exceed the *targetCount* gives the index in the histogram of the desired percentile value. Because the minimum value in the *input* was zero,and because the histogram was constructed with bin size equal to one, this index is also the actual value of the requested percentile in the (possibly scaled) input array.

Since the input array may have been scaled and offset, this value is then 'decoded' based on the scale and offset of the input array to give the 'real-world' value of the desired percentile value.

Returns

The real-world (e.g. unscaled) value in the input array which corresponds to the *requestedPercent* percentile in the input. For example, if *requestedPercent* is 90.0, this routine returns the minimum pixel value which is greater than 90% of all other pixel values in the input array.

Document Revision Date

29 September, 1998

SKRadiallyAlignedOrientation()

Name

SKRadiallyAlignedOrientation()
Function to take an input 2D SKArray of 'orientations' (angles) and determine how close each orientation is to the corresponding radial angle.

Synopsis

#include <skwrt.h>

SKArray<short> SKRadiallyAlignedOrientation(SKArray<short>& orient, SKCoordI center);

Description

Function to take an input array of "orientations" (angles) and determine how close each orientation is to the corresponding radial angle. For example, in a 9-by-9 image the center pixel is (4, 4). The pixel at (8, 8) has a radial angle of 45 degrees relative to the center. If the value (orientation) at pixel (8, 8) is 50, then this function will store "5" in the output image at pixel (8, 8).

This function is typically used in suppressing radially-aligned interest in feature detectors, as such features are often caused by out-of-trip weather returns.

In this function the *center* is supplied as an argument so that these radial angle differences can be computed relative to a point other than the array's true center point, if desired.

Returns

The 2-D array of angle differences (pixelwise input orientation differenced against pixelwise radial angle).

Warning

All angles are assumed to be 180 degrees ambiguous, so angle differences are computed by function *SKAngle180Difference*(), not function *SKAngleDifference*().

See Also

Functions *SKAngle180Difference*() and *SKAngleDifference*().

Document Revision Date

30 September, 1998

SKReplaceMissingWithMedian()

Name

SKReplaceMissingWithMedian()
Function to perform the 'ReplaceMissingWithMedian' function at a single pixel within an input image. To replace all missing values in an image with a median value this function should be used in conjunction with the SKArray class member function "Apply()". See example below.

Synopsis

```
#include <skwrt.h>

template <class T>
T SKReplaceMissingWithMedian( SKArray<T> &input, T *xPtr, void *kPtr );
```

Description

Function to perform the 'ReplaceMissingWithMedian' function at a single pixel. If the current input pixel is (pointed to by *xPtr*) is *nil*, then this pixel value is replaced by the median of all non-*nil* input pixels within the supplied window (kernel) pointed to by *kPtr*. If all the pixels in the window are *nil*, then the input pixel remains *nil*. If the current input pixel is non-*nil*, it is unchanged. To replace all missing values in an image with a median value this function should be used in conjunction with the SKArray class member function "Apply()".

Returns

The value to replace the pixel at *xPtr* with (if it is to be replaced, i.e. if it is missing).

Example

```
// Tests for SKReplaceMissingWithMedian().

// Input array. Nil values to be replaced with median of a window.
static char *medianInputData1[] =
{
  "nil  2  3  4  5  6  7  8  9 10",
  "  2  3  4  5  6  7  8  9  9  9",
  "  1  1  2  4  4  6  6  8  8  8",
  "nil  1  3  5  7  9  7  9  7  9",
  "nil  3  4  5  6  8 nil 7  7  7",
  "  2  3  4  5  6  6 nil nil nil 10",
  "  3  4  5  6  7  6 nil nil nil nil",
  "  1  1  3  3  5  5 nil nil nil  8",
  "  1  3  1  3  5  7  5  7  7  3",
  "nil nil 9  3  6  5 nil  6  8 nil",
  0,
};

// Create a kernel for the median operation (window will be 3 by 3).
```

```
SKArray<short> kernel(3,3);

SKArray<short> output;
SKArrayPad<short> pad( PAD_MIRROR, 0, 1, 1 );

// Input must be padded.
SKArray<short> medianInput( pad, 10, 10 );

medianInput.SetSliceVals( medianInputData1 );
medianInput.Mirror();

// Replace missing values.
output = medianInput.Apply( SKReplaceMissingWithMedian, (void*)&kernel );

// Output will look like this:
static char *medianTruthData1[] =
{
  " 2  2  3  4  5  6  7  8  9 10",
  " 2  3  4  5  6  7  8  9  9  9",
  " 1  1  2  4  4  6  6  8  8  8",
  " 1  1  3  5  7  9  7  9  7  9",
  " 2  3  4  5  6  8  7  7  7  7",
  " 2  3  4  5  6  6  6  7  7 10",
  " 3  4  5  6  7  6  6 nil 8  8",
  " 1  1  3  3  5  5  6  7  7  8",
  " 1  3  1  3  5  7  5  7  7  3",
  " 1  3  9  3  6  5  6  6  8  7",
  0,
};

// Note one pixel is still missing as all values within the 3 by 3 kernel were
// missing, so no median replacement value could be found.
```

Document Revision Date

29 September, 1998

SKWeightedMergeTilts()

Name

SKWeightedMergeTilts()
Function to merge two tilts (arrays, in polar (r, theta) format) into a single tilt. Use data from higher tilt in close to the radar, (i.e. r < mergeBoundNearKM), use data from low tilt away from radar (i.e. r > mergeBoundFarKM) and use linearly ramped average of data from both tilts in between.

Synopsis

```
#include <skwrt.h>

template <class T>
SKArray<T> SKWeightedMergeTilts( SKArray<T>& tiltLow, SKArray<T>&
                                 tiltHigh, float mergeBoundNearKM, float
                                 mergeBoundFarKM );
```

Description

Function to merge two tilts (arrays, in polar (r, theta) format) into a single tilt. Use data from higher tilt in close to the radar, (i.e. r < mergeBoundNearKM), use data from low tilt away from radar (i.e. r > mergeBoundFarKM) and use linearly ramped average of data from both tilts in between.

Returns

The output (polar) *SKArray* of merged data.

Warning

In regions where the merged data is just the raw low or high tilt data, missing values in the raw data become missing values in the merged data. In regions where there is weighted averaging, the tilt with the higher weight 'wins' in the case of *nil* data. For example, if the low tilt weight is 0.2 and the high tilt weight is 0.8, a missing value in the high tilt will translate to a *nil* value in the output image regardless of the low tilt data. With the same weights and a *nil* low tilt value and non-*nil* high tilt value, the unweighted non-*nil* high tilt value will become the output value.

Document Revision Date

29 September, 1998