Laboratory for Advanced Planning and simulation project

Lumen Carotid Segmentation Software: Programming user Manual

Alan Scheinine[a] , Piero Pili[b] and Fabrizio Murgia[b]

[a] RSC/High Performance Computing Systems Area, CRS4
[b] EIP/Geometric Modelling and Monte Carlo Simulations Area, CRS4

Lumen Carotid Segmentation Software: Programming user Manual

Alan Scheinine[a] , Piero Pili[b] and Fabrizio Murgia[b]

[a] Geometric Modelling and Monte Carlo Simulations Area, CRS4
[b] High Performance Computing Systems Area, CRS4

12 Giugno 2003

ABSTRACT

This document illustrates the segmentation source modules used to extract lumen contours from CT images. The modules, based on open source software, are:

Hash Library module: hash table;

Random Number generator module: George Marsaglia's The mother of all random number generators producing uniformly distributed pseudo random 32 bit values with period about 2^{250};

DXIO Library module: Data Explorer, interface module and DICOM readers;

Loop Library module: the package Loop contains classes for loops and tubes, and linear algebra for interpolation;

Smoothing Library module: the package Smoothing contains algorithms for several techniques, including smoothing a two-dimensional loop and extrapolating the ends of tubes at a bifurcation;

Filtering Library module: the image filters are independent of the segmentation program;

Contour The module for segmenting lumen;

In Appendix has been reported the GSNAKE-API and GSNAKE program that we have used for in own tool.

Contents

1 DXIO **9**

1.1 Format information from Data Explorer documentation 9

1.2 Description of the DXIO functions 10

1.3 DX_DICOM: Basic numbers and numerical vectors 13

2 HASH **15**

2.1 Synopsis 15

2.2 Description 15

2.3 Variables 15

2.4 Public_Functions 16

2.5 Private_Functions 17

2.6 HISTORY 18

3 RANDOM **19**

3.1 George Marsaglia's random number generators 19

4 LOOP **23**

4.1 Loop 23

4.2 Overview 23

4.3 class CyclicLoop 36

4.4 Basic constants and simple functions 37

4.5 Classes to hold arrays 38

4.6 Interpolation and matrix inversion 39

5 SMOOTHING **41**

5.1 Package Smoothing 41

5.2 class Smoothing 41

5.3 Function find_mapping 47

5.4 Program two_to_one 47

5.5 Portable Pixmap 52

6 FILTERING **54**

6.1 Image Filter 54

6.2 ContourGUI and CanvasContourGui 56

6.3 DICOM read 59

7 CONTOUR **60**

7.1 CONTOUR: Image Filter 60

7.2 Mouse-Contour Interaction 63

7.3 Introduction 69

7.4 Overview 69

Bibliography **71**

APPENDICES **72**

Appendix A GSNAKE **72**

A.1 About this manual 72

A.2 How to Avoid Reading this Manual 72

A.3 Design of the GSNAKE API 73

A.4 Class Design and Implementation 73

Appendix B GSNAKE Class Library Reference **75**

B.1 IMAGE : The raw image object 75

B.1.1 IMAGE Constructor 75

B.1.2 IMAGE Destructor 76

B.1.3 Resetting an IMAGE 76

B.1.4 Initializing image matrix 76
B.1.5 Initializing Gaussian template 77
B.1.6 Putting data into image matrix 77
B.1.7 Getting data from image matrix 78
B.1.8 Getting IMAGE row and col 78
B.1.9 Printing image data 79
B.1.10 Displaying image 79
B.1.11 Reading image data from file 79
B.1.12 Writing image data to file 80
B.1.13 Histogram conditioning of image 81
B.1.14 Image correlation 81
B.1.15 X window image generation 82
B.1.16 Cutting an image segment 82
B.1.17 Copying an image 83
B.1.18 Filling of an area of image 83
B.1.19 Example : Reading, writing and correlation of images . . 84
B.1.20 Example : Histogram specification of an image 85
B.1.21 Example : Cutting and copying of image segments 86
B.2 EDGE : Edge Gradient 87
B.2.1 EDGE constructor 88
B.2.2 EDGE destructor 88
B.2.3 Resetting an `EDGE` object 88
B.2.4 Retrieving magnitude and angle of an edge point 88
B.2.5 Storing magnitude and angle of an edge point 89
B.2.6 Displaying edge map 89
B.2.7 Calculating the edge map 90
B.2.8 Getting row and column of `EDGE` 91
B.2.9 Example : Edge computation 91
B.3 PYRAMID : Pyramid of Images 92
B.3.1 PYRAMID Constructor 93
B.3.2 PYRAMID Destructor 93

B.3.3 Resetting PYRAMID object 93
B.3.4 Generating pyramid images 94
B.3.5 Accessing edge map 94
B.3.6 Accessing Gaussian image 95
B.3.7 Getting pyramid level 95
B.3.8 Print Gaussian pyramid data 96
B.3.9 Displaying the pyramid of images 96
B.3.10 Duplicating a pyramid 96
B.3.11 Putting root image into PYRAMID 97
B.3.12 Example:Building of pyramid from image 97
B.3.13 Example:Accessing a particular level image 98
B.4 MATRIX : Simple matrix calculation 99
B.4.1 Print matrix data 100
B.4.2 Initializing matrix 100
B.4.3 Matrix transpose 101
B.4.4 Matrix addition 101
B.4.5 Matrix substraction 101
B.4.6 Matrix multiplication 102
B.4.7 Swapping rows of matrix 102
B.4.8 Matrix inversion 103
B.5 SNAXEL : The contour model unit 103
B.5.1 Computing mean position 104
B.5.2 Computing tangent vector 105
B.5.3 Computing normal vector 105
B.5.4 Calculating the `SNAXEL` angle 106
B.5.5 Showing snaxel 106
B.5.6 Interface row and column information 107
B.5.7 Interface snaxel energy information 107
B.5.8 Interface parameter information 108
B.5.9 Interface pointers to neighbouring snaxels 108
B.5.10 Example : Using mean position to calculate Internal energy 109

B.5.11 Example : Verifying vector calculations 111
B.6 CONTOUR : A deformable template object 112
B.6.1 `CONTOUR` constructor 113
B.6.2 `CONTOUR` destructor 113
B.6.3 Resetting a `CONTOUR` 113
B.6.4 Automatic initialization of closed contour 114
B.6.5 Automatic initialization of open contour 114
B.6.6 Manual initialization of a contour 115
B.6.7 File interface methods 115
B.6.8 Learning shape matrix 116
B.6.9 Computing normalizing constant for classification purpose 116
B.6.10 Computing the average vector distance l(U) between snaxels 117
B.6.11 Regenerating shape matrix 117
B.6.12 Conversion of Coordinates 118
B.6.13 Calculating the internal energy of an individual snaxel . . 118
B.6.14 Affine transformations of contour 119
B.6.15 Computing the centre of gravity of contour 119
B.6.16 Duplicating a contour 120
B.6.17 Retrieving and accessing of a contour content 120
B.6.18 Displaying contour shape 121
B.6.19 Showing a contour on another image 122
B.6.20 Example : Initialization of contours 122
B.6.21 Example : Affine transformations of contour 123
B.6.22 Example : Coordinate conversion of contour 125
B.6.23 Example : Duplication of contour 126
B.7 GHOUGH : Generalised Hough Transform Object 127
B.7.1 GHOUGH constructor 128
B.7.2 GHOUGH destructor 129
B.7.3 Resetting GHOUGH object 129
B.7.4 Finding one contour 129
B.7.5 Finding multiple contours 130

B.7.6 Example : Localization of a contour 131
B.7.7 Example : Localization of multiple contours 132
B.8 GSNAKE : Generalized Active Contour Mo-del 134
B.8.1 GSNAKE constructor 135
B.8.2 GSNAKE destructor 135
B.8.3 GSNAKE destructor 135
B.8.4 Generating pyramid images 136
B.8.5 Localization of contour 136
B.8.6 Fine localization of GSNAKE template 137
B.8.7 Minimization of GSNAKE 138
B.8.8 Marginalizing gsnake to get probablity 138
B.8.9 Calculating total energy of a gsnake 139
B.8.10 Calculating total energy of a snaxel 139
B.8.11 Calculating internal energy of a snaxel 140
B.8.12 Caluclating Internal energy at snaxel co-ordinates 140
B.8.13 Calculating external energy of a snaxel 140
B.8.14 Showing GSNAKE 141
B.8.15 Displaying gsnake and image 141
B.8.16 Manual deformation of a contour 142
B.8.17 Duplicating a GSNAKE 142
B.8.18 Getting external energy input type 143
B.8.19 Getting and writing the regularization parameter 143
B.8.20 Retrieving internal and external energy 144
B.8.21 Example : Localization and minimization of GSNAKE . 144
B.8.22 Example : Energy calculation I 146
B.8.23 Example : Energy calculation II 147
B.9 MODEL : Shape Learning Class 148
B.9.1 MODEL constructor 149
B.9.2 Learning shape matrix 149
B.9.3 Learning local deformation variances 150
B.9.4 Accessing the trained model 150

B.9.5 Example : Learning of shape matrix from different samples 151

B.10 CLASSIFY : Contour Classification 152

B.10.1 CLASSIFY constructor 153

B.10.2 CLASSIFY destructor 154

B.10.3 Loading a template into CLASSIFY 154

B.10.4 Classifying templates 155

B.10.5 Selecting the best matching contour 155

B.10.6 Getting the number of templates read 156

B.10.7 Getting the matching score 156

B.10.8 Printing templates score 156

B.10.9 Getting label of template 157

B.10.10 Example : Classifying various templates 157

Appendix C GSNAKE API Command Line Utilities 159

C.1 Image Processing utilities 159

C.1.1 Image generation: `imggen` 159

C.1.2 Image viewing: `imgshow` 160

C.1.3 Gaussian pyramid images generation: `imgpyramid` 161

C.1.4 Image conditioning : `imgcond` 162

C.1.5 Image learning : `imglearn` 163

C.2 GSNAKE utilities 163

C.2.1 Template generation : `gsinit` 163

C.2.2 Contour viewing: `gsshow` 164

C.2.3 Contour matching: `gsfit` 165

Appendix D STL 166

D.1 STL Introduction 166

D.2 Overall Description 167

D.3 Description of the functions 169

Chapter 1

DXIO

1.1 Format information from Data Explorer documentation

(Not every possibility is supported with this set of programs.)

The data is specified by an offset in bytes in the data section of the current file, an offset within the data section of another Data Explorer file, or by the keyword follows, indicating that the data begins immediately following the newline after the follows keyword. The offset is specified in bytes for both binary and text files.

Optional keywords before the data keyword specify the format and byte order of the data. The mode keyword before a data-location specification sets the default data encoding for all subsequent data clauses to be the most recently defined data encoding. The default data encoding is text (or ascii on all currently supported systems). The ieee keyword specifies the ANSI/IEEE standard 754 data format.

If binary (or ieee on all currently supported systems) is specified, the default byte order depends on the platform on which Data Explorer is running. On the DEC Alpha, the default byte order is lsb (least significant byte first). On all other platforms [written before Intel supported], the default byte order is msb (most significant byte first). The 'data mode' clause can be used outside an Array Object definition; see "Data Mode Clause" for more information.

```
object  number [class] array
        "name"
[type  [unsigned] byte ]
       signed byte
```

```
        unsigned short
        [signed] short
        unsigned int
        [signed] int
        hyper
        float
        double
        string

 [category   real     ]
           complex
 [rank number]
 [shape number ...]
 items number
 [  msb  ] [  text ] data [ mode ]   offset
    lsb       ieee                   file file,offset
              binary                 follows
              ascii
```

If byte, short, or int are not prefixed with either signed or unsigned, by default, bytes are unsigned, shorts are signed and ints are signed. For compatibility with earlier versions, char is accepted as a synonym for byte. Note: For string-type data, the Array rank should be 1 and the Array shape should be the length of the longest string plus 1.

1.2 Description of the DXIO functions

Constants for types The data types are flagged by the constants

DX_TYPE_NONE,

DX_TYPE_CHAR,

DX_TYPE_SIGNED_CHAR,

DX_TYPE_UNSIGNED_CHAR,

DX_TYPE_SHORT,

DX_TYPE_UNSIGNED_SHORT,

DX_TYPE_INT,

DX_TYPE_UNSIGNED_INT,

DX_TYPE_LONG,

DX_TYPE_UNSIGNED_LONG,

DX_TYPE_FLOAT,

DX_TYPE_DOUBLE, and

DX_TYPE_LONG_DOUBLE. Not all of these data types are implemented by DX files, in particular, the long types are not implemented.

Constants for endianess The endianess is flagged using

DX_UNKNOWN_ENDIAN,

DX_MSB, and

DX_LSB. These flags describe what is found in the header, but otherwise, no adjustment of the data is made for endianess and binary files created by these functions will not be portable.

loop_indices The struct

loop_indices passes a group of specialized variables as one argument of

make_loop_indices.

read_dx_header

read_dx_header reads a header file in Data Explorer format for a single array. The actual array may be in another file declared in the header file.

get_text

get_text fills-in a character array with the text of the header information.

read_dx_type

read_dx_type opens a file and parses the DX format header simply to determine the type of the data array. It can be useful when one wants to know what type of data array needs to be malloc'ed.

Reading DX data

read_dx_char,

read_dx_signed_char,

read_dx_unsigned_char,

read_dx_short,

read_dx_unsigned_short,

read_dx_int,

read_dx_unsigned_int,

read_dx_float and

read_dx_double

These functions read the header and the data in DX format.

Note the that pointer that is returned itemers to a newly malloc'ed array that the user will need to free when no longer needed.

free_dx_data Serves as an interface between C++ and C. When data is read from a file, a malloc'ed array is returned.

free_dx_data permits a C++ to use free() to return the space to the heap.

make_loop_indices

make_loop_indices fills-in the structure

loop_indices.

open_data_file

open_data_file positions file pointer to the beginning of the data.

float_is_zero

float_is_zero tests if a float type value is effectively zero, returning 1 (true) if the argument is close to zero.

fix_relative_name

fix_relative_name checks for a leading slash in the pathname outfile. If there is no leading slash, the substring of filename up to and including the last slash is pitemixed to the outfile character arrry.

reassign_file_pointer

reassign_file_pointer closes the header file and opens a second file for writing the data.

strstr_word

strstr_word finds first match that is a distinct word.

distinct_word

distinct_word checks that a string is not imbedded.

seek_type

seek_type seeks the keyword that describes the DX data type.

to_dx_type

to_dx_type converts from integer to string for a given DX type. Used for writing the header.

Writing the header

write_dx_header writes the header. The format of the header is based on the assumption that the data to be written has the x-direction varying the fastest; which is the opposite of the DX default convention.

Writing header and data Writing everything, both the header and the data array is done with the functions

write_dx_char,

write_dx_signed_char,

write_dx_unsigned_char,

write_dx_short,

write_dx_unsigned_short,

write_dx_int,

write_dx_unsigned_int,

write_dx_float and

write_dx_double.

Reading and writing one data item A DX file to be read may not have been created with this set of functions and as a consequence, the order of the data may not have the x-direction varying most rapidly. So for reading data, a specialized loop is used that calculates the array index for each data item and the data items are read one at a time. Though it is not absolutely necessary to write the data one item at a time, such functions have been implemented; moreover, writing one item at a time for formatted data allows line feeds to be inserted for easier reading. These functions are

1.3 DX_DICOM: Basic numbers and numerical vectors

LimitRange The class LimitRange is used for the conversion between primitive numerical types. The role is similar to a static cast. Since it operates on one number at a time, the conversion is not efficient. Nonetheless, the class may be useful for type conversion of fields between different steps of processing while avoiding compiler warnings. Here are some examples of the functions

```
static T limit_range(char s);
static T limit_range(unsigned int s);
static T limit_range(double s);
```

The primitive numerical types that can be used for the function parameter or return value are *char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double* and *long double.*

LocalNumTNTVect The class LocalNumTNTVect inherits from the TNT class **Vector**. A few more functions have been added and some functions have been changed to increase efficiency. This class definition is identical to that used for Nast++. This is a vector for primitive numerical types rather than a vector of an arbitrary class. Note that the size does not change when allocation is done beyond the range of the internal array. Such an assignment is an error. The size is set when the class is constructed or by using **newsize(int)**.

Chapter 2

HASH

2.1 Synopsis

The hash table library module is rthe file hash.c - hash table with a KEY that is an array of integers and a VALUE that is one integer. The include file is in flosys3d directory includefort_general. The code of the user should have #include<hash.h>. The compiled functions are in the flosys3d library libsmlutl.a.

2.2 Description

Functions for a hash table. The key is a vector of integers. For each hash table, the number of elements in the vector is fixed when the hash table is created. Associated with each key is a value of type integer. The hash table automatically grows or shrinks as needed.

- Variables
- Public Functions
- Private Functions

2.3 Variables

The type definition *HTable* and the variable *hash_ifdebug* are specifically defined for this package. Other variables listed below are example declarations.

HTable HTable *htable; A typical declaration for a hash table named *htable*.

num_key_elmnts int num_key_elmnts; A variable set equal to the number of elements in the key. The value can range from 1 to 12.

key int key[NUM_OF_ELEMENTS]; This array could contain one key.

fd FILE *fd; A binary file that has already been opened.

key_count int key_count; The number of items in the key list returned by <CODE> get_keys().

value_count int value_count; The number of items in the value list returned by get_values().

key_vec int **key_vec; An array of pointers to the base address of each key returned by get_keys (htable, & key_count, & key_vec). The space occupied by the list can be destroyed using free(key_vec).

value_vec int *value_vec; An array of integer values returned by get_values (htable, & value_count, &value_vec). The space occupied by the list can be destroyed using free(value_vec).

hash_ifdebug int hash_ifdebug; If set to 1, some print statements are activated. Declared in hash.h.

2.4 Public_Functions

new_hash htable = new_hash(num_key_elmnts)

HTable * new_hash(int num_key_elmnts)

Returns a newly created hash table for which the keys always have *num_key-_elmnts* elements. Returns NULL if not successful.

destroy_hash destroy_hash(htable) void destroy_hash(HTable *htable)

Frees the space occupied by *htable*.

hash_insert status = hash_insert(htable, key, value) int hash_insert(HTable *htable, int *key, int value) Returns 0 if key[] was not in the hash table (key was never used or the key was deleted prior to this call). Returns 1 if key[] was in the hash table and the value was replaced. Returns a negative number if an error occurs.

hash_lookup status = hash_lookup(htable, key, & value) int hash_lookup (HTable *htable, int *key, int *value) Returns 0 and sets *value* if key[] is found. Returns 1 if key[] is not found. Returns a negative number if an error occurs.

hash_delete status = hash_delete(htable, key) int hash_delete(HTable *htable, int *key) Returns 0 if key[] was deleted from the hash table by this call. Returns 1 if nothing was done: item was never inserted or was delete previously. Returns a negative number if an error occurs.

get_keys status = get_keys(htable, & key_count, & key_vec)

int get_keys(HTable *htable, int *key_count, int ***key_vec) Creates an array of pointers to type integer *key_vec* which contains <EM>key_count</EM> items. The first key returned has base address key_count[0]. Returns 0 if successful.

get_values status = get_values(htable, & value_count, & value_vec) int get_values (HTable *htable, int *value_count, int **value_vec) Creates an array of integers *value_vec* which contains *value_count* items. Returns 0 if successful.

write_hash status = write_hash(htable, fd) int write_hash(HTable *htable, FILE *fd) Writes a hash table *htable* to file *fd*. Returns 0 if successful.

read_hash htable = read_hash(fd) HTable *read_hash(FILE *fd)

Returns a hash table read from file *fd*. The hash table space is newly created as if new_hash() had been called. Returns NULL if not successful.

2.5 Private_Functions

hash01 static unsigned int hash01(int *key_int_list, int num_key_elmnts, unsigned int mask) Given a key, returns a hash index.

hash02 static unsigned int hash02(int *key_int_list, int num_key_elmnts, unsigned int mask, unsigned short expand_factor) Given a key, returns a hash index using a method different from hash01.

make_tablesize static void make_tablesize(int new_tablesize, HTable *htable)

Given a suggested table size *new_tablesize*, finds a table size that is a power of 2 and sets the hash table variables *table_size*, *mask*, *not_mask*, *rehash_upper* and *rehash_lower*.

compare_keys static int compare_keys(int *key1, int *key2, int num_key_elmnts) Return 1 if two keys are equal, otherwise returns 0.

rehash int rehash(HTable *htable) Determines if the table size should be changed. Returns 0 if nothing is done. Returns 1 if rehashing is done. Returns negative if there is an error.

no_rehash_insert static int no_rehash_insert(int *key, int value, HTable *htable) Used in the function rehash to insert values into the resized hash table.

write_hash_entry static int write_hash_entry (HashEntry *entry, long int base, FILE *fd)

Writes one structure of type HashEntry to file *fd*. The variable *base* is used to convert the pointer to the key array to an integer with zero offset. Returns the number of words written.

read_hash_entry static int read_hash_entry (HashEntry *entry, long int base, FILE *fd)

Reads one structure of type HashEntry from file *fd*. The variable *base* is used to create a pointer to the key array. Returns the number of words read.

2.6 HISTORY

The KEY is an integer vector because originally this set of hash routines was written for geometric objects used in finite element descriptions. For example, a triangular surface could be stored as the indices of the three vertices, with the VALUE being the index that identifies the triangle.

Chapter 3

RANDOM

This library module furnishes the random numbers generator algorithm used by the segmentation tool.

3.1 George Marsaglia's random number generators

George Marsaglia's the mother of all random number generators producing uniformly distributed pseudo random 32 bit values with period about 2^{250}. The text of Marsaglia's posting is appended at the end of the function.

The arrays mother1 and mother2 store carry values in their first element, and random 16 bit numbers in elements 1 to 8. These random numbers are moved to elements 2 to 9 and a new carry and number are generated and placed in elements 0 and 1. The arrays mother1 and mother2 are filled with random 16 bit values on first call of mother by another generator. mStart is the switch. Returns: A 32 bit random number is obtained by combining the output of the two generators and returned in *pSeed. It is also scaled by $2^{32}-1$ and returned as a double between 0 and 1

SEED: The inital value of *pSeed may be any long value

Bob Wheeler 8/8/94

Marsaglia's comments

Yet another RNG

Random number generators are frequently posted on the network; my colleagues and I posted ULTRA in 1992 and, from the number of requests for releases to use it in software packages, it seems to be widely used.

I have long been interested in RNG's and several of my early ones are used as system generators or in statistical packages.

So why another one? And why here?

Because I want to describe a generator, or rather, a class of generators, so promising I am inclined to call it *The Mother of All Random Number Generators* and because the generator seems promising enough to justify shortcutting the many months, even years, before new developments are widely known through publication in a journal.

This new class leads to simple, fast programs that produce sequences with very long periods. They use multiplication, which experience has shown does a better job of mixing bits than do +,- or exclusive-or, and they do it with easily-implemented arithmetic modulo a power of 2, unlike arithmetic modulo a prime. The latter, while satisfactory, is difficult to implement. But the arithmetic here modulo 2^{16} or 2^{32} does not suffer the flaws of ordinary congruential generators for those moduli: trailing bits too regular. On the contrary, all bits of the integers produced by this new method, whether leading or trailing, have passed extensive tests of randomness.

Here is an idea of how it works, using, say, integers of six decimal digits from which we return random 3-digit integers. Start with n=123456, the seed.

Then form a new $n = 672 * 456 + 123 = 306555$ and return 555.
Then form a new $n = 672 * 555 + 306 = 373266$ and return 266.
Then form a new $n = 672 * 266 + 373 = 179125$ and return 125.

and so on. Got it? This is a multiply-with-carry sequence $x(n) = 672 * x(n-1) + carrymodb = 1000$, where the carry is the number of b's dropped in the modular reduction. The resulting sequence of 3-digit x's has period 335,999. Try it.

No big deal, but that's just an example to give the idea. Now consider the sequence of 16-bit integers produced by the two C statements:

$k = 30903 * (k\&65535) + (k >> 16);$
$return(k\&65535);$

Notice that it is doing just what we did in the example: multiply the bottom half (by 30903, carefully chosen), add the top half and return the new bottom.

That will produce a sequence of 16-bit integers with period ¿ 2^29, and if we concatenate two such:

$k = 30903 * (k\&65535) + (k >> 16);$
$j = 18000 * (j\&65535) + (j >> 16);$
$return((k << 16) + j);$

we get a sequence of more than 2^{59} 32-bit integers before cycling.

The following segment in a (properly initialized) C procedure will generate more than 2^{118} 32-bit random integers from six random seed values i,j,k,l,m,n:

$k = 30903 * (k\&65535) + (k >> 16);$
$j = 18000 * (j\&65535) + (j >> 16);$
$i = 29013 * (i\&65535) + (i >> 16);$
$l = 30345 * (l\&65535) + (l >> 16);$
$m = 30903 * (m\&65535) + (m >> 16);$
$n = 31083 * (n\&65535) + (n >> 16);$
$return((k + i + m) >> 16) + j + l + n);$

And it will do it much faster than any of several widely used generators designed to use 16-bit integer arithmetic, such as that of Wichman-Hill that combines congruential sequences for three 15-bit primes (Applied Statistics, v31, p188-190, 1982), period about 2^{42}.

I call these multiply-with-carry generators. Here is an extravagant 16-bit example that is easily implemented in C or Fortran. It does such a thorough job of mixing the bits of the previous eight values that it is difficult to imagine a test of randomness it could not pass:

$x[n] = 12013x[n-8] + 1066x[n-7] + 1215x[n-6] + 1492x[n-5] + 1776x[n-4] + 1812x[n-3] + 1860x[n-2] + 1941x[n-1] + carrymod2^{16}$.

The linear combination occupies at most 31 bits of a 32-bit integer. The bottom 16 is the output, the top 15 the next carry. It is probably best to implement with 8 case segments. It takes 8 microseconds on my PC. Of course it just provides 16-bit random integers, but awfully good ones. For 32 bits you would have to combine it with another, such as

$x[n] = 9272x[n-8] + 7777x[n-7] + 6666x[n-6] + 5555x[n-5] + 4444x[n-4] + 3333x[n-3] + 2222x[n-2] + 1111x[n-1] + carrymod2^{16}$.

Concatenating those two gives a sequence of 32-bit random integers (from 16 random 16-bit seeds), period about 2^{250}. It is so awesome it may merit the Mother of All RNG's title.

The coefficients in those two linear combinations suggest that it is easy to get long-period sequences, and that is true. The result is due to Cemal Kac, who extended the theory we gave for add-with-carry sequences: Choose a base b and give r seed values x[1],...,x[r] and an initial 'carry' c. Then the multiply-with-carry sequence

$x[n] = a1 * x[n-1] + a2 * x[n-2] + ... + ar * x[n-r] + carrymodb$

where the new carry is the number of b's dropped in the modular reduction, will have period the order of b in the group of residues relatively prime to $m = ar * b^r + ... + a1b^1 - 1$. Furthermore, the x's are, in reverse order, the digits in the expansion of k/m to the base b, for some 0¡k¡m.

In practice $b = 2^{16}$ or $b = 2^{32}$ allows the new integer and the new carry to be the bottom and top half of a 32- or 64-bit linear combination of 16- or 32-bit integers. And it is easy to find suitable m's if you have a primality test: just search through candidate coefficients until you get an m that is a safeprime-both m and (m-1)/2 are prime. Then the period of the multiply-with-carry sequence will be the prime (m-1)/2 (It can't be m-1 because $b = 2^{16}$ or 2^{32} is a square).

Here is an interesting simple MWC generator with period¿ 2^{92}, for 32-bit arithmetic:

$x[n] = 1111111464 * (x[n-1] + x[n-2]) + carrymod2^{32}$.

Suppose you have functions, say top() and bot(), that give the top and bottom halves of a 64-bit result. Then, with initial 32-bit x, y and carry c, simple statements such as $y = bot(1111111464 * (x + y) + c)\ x = y\ c = top(y)$ will, repeated, give over 2^{92} random 32-bit y's.

Not many machines have 64 bit integers yet. But most assemblers for modern CPU's permit access to the top and bottom halves of a 64-bit product.

I don't know how to readily access the top half of a 64-bit product in C. Can anyone suggest how it might be done? (in integer arithmetic)

George Marsaglia geo@stat.fsu.edu

Chapter 4

LOOP

4.1 Loop

The package Loop contains classes for loops and tubes, and linear algebra for interpolation.

4.2 Overview

Class hierarchy

The base of the array classes is BasicArrayBase. This class does not contain an array, it simple contains size and length data members and methods to access these data. Note that size and length are independent. The size is the size of the allocated storage, whereas, the length is the current number of entries, possibly less than the size.

The only class that has BasicArrayBase as an immediate base class is WithTopology. All other classes that represent arrays inherit BasicArrayBase by inheriting WithTopology. The only time BasicArrayBase appears in a more derived class is when an algorithm uses the static function

```
BasicArrayBase::generic_mod(i, length)
```

The function *generic_mod* is different from *mod* in the treatment of negative numbers. Toroidal boundary conditions are implemented so that generic_mod(-1, length) is equal to (length - 1), and so on.

The class BasicArrayBase contains the virtual functions.

```
virtual void renew_basic_array() = 0;
virtual void renew_basic_array(int size_in) = 0;
virtual int length() const;
virtual void setLength(int length_in);
```

The next step in the array hierarchy, the class WithTopology, does not contain space for an array. It simple implements either a linear of cyclic topology. As well as having a method for setting the topology, there are the methods

```
virtual void lockTopology();
virtual void unlockTopology();
```

Though a lock can be unlocked, the locking mechanism can be used as an indication that initialization has been done. The WithTopology constructor starts as **Lup::LINEAR_ARRAY** and unlocked. In an algorithm that creates an array, by locking (possibly first setting the topology to **Lup::CYCLIC_ARRAY**) later steps in the algorithm cannot undo the initialization by mistake. By the way, the namespace for constants is "Lup" instead of "Loop" because there is a class named "Loop".

One other generally useful function is introduced in WithTopology, the function *my_mod*.

```
virtual int my_mod(int i);
```

The function *my_mod* returns *generic_mod(i, length)* where *length* is the value set for the particular instantation of the class.

The next stap in the array hierarchy is the class that contains an allocationed array

```
template<class T> class BasicArray
```

which inherits from class WithTopology. The primitive array within this class can be any type, T. The class WithTopology is not used explicitly elsewhere in this package, but rather, its methods are accessed through BasicArray or classes derived from BasicArray. By the way, though the array access operators

```
virtual T& operator[](int i)
virtual const T& operator[](int i) const
```

are defined, yet the primitive array T* is public because sometimes very fast access may be needed.

In BasicArray the virtual functions

```
void renew_basic_array()
void renew_basic_array(int size_in)
```

are defined, they destroy the contents when changing the size of the primitive array. In addition, there is defined the function *resize(int newsize)* which preserves entries from 0 to (newsize - 1). The virtual function of BasicArrayBase, *setLength(int length_in)*, is redefined to use *resize(int length_in)*. This does not guarantee that size() == length() because *resize* is not called if the allocated space is already sufficient.

There is also a class

```
template<class T> class ReadOnlyArray
```

that derives from

```
BasicArray<T>
```

but the class ReadOnlyArray is not particularly useful for several reasons: the array T* is public, the assignment operator changes the content in any case, and the non-constant operator array access operator that is redefined from BasicArray, that is,

```
T& BasicArray<T>::operator[](int i)
```

must return something.

The class BasicArray is analogous to the class **vector** of the standard template library. One difference is that BasicArray includes the option of setting a cyclic topology. Another difference is that the length is distinct from the size. The latter difference can also be found in the CORBA class **sequence**. Moreover, the BasicArray also does not automatically increase in size, unlike the STL **vector** when *push_back()* is used. In many cases, BasicArray is used to represent a loop (a contour) which has a cyclic topology. In the case of a cyclic loop, access above or below the array limit is allowed, the wrap-around is automatic. As a consequence, an index that is outside the conventional array bounds can still be useful. But that index would change its meaning, it would change the element to which it points, if the length of the loop changes. More specifically, the loops are used to form tubes in which for a given loop an adjacent loop has points to the given loop. Of course, a loop needs to change its length (the number of points) for algorithms that refine the points of a loop or cancel points that are not included in a longitude of a tube, but such changes must be done coherently with the adjacent loop. The problem is the question of indexing when there is wrap-around. So changing the length (that is, the number of points) is an procedure that must be done carefully. In sum, the differences between a BasicLoop and an STL **vector** are motivated.

In the hierarchy of classes for loops, the BasicLoop is hidden in three other classes:

```
class FlatPointArray : public BasicArray<Doublet>
class PointArray : public BasicArray<Triplet>
class PatchedPointArray : public BasicArray<PatchedPoint>
```

It seems easier for the programmer (and with regard to the first generation C++ compilers, easier for the compiler) to use non-templated classes, or in this case, specific cases of the templated class BasicArray. Aside from loops, BasicArray is used as an array class for holding sets of loops that form a tube, for example

```
class LoopArray : public BasicArray<Loop>
class PatchedLoopArray : public BasicArray<PatchedLoop>
```

and simply integers for a longitudinal line

```
BasicArray<int> Longitude::point_array;
```

But class BasicArray does not appear explicitly outside of the package **Loop**, which is reasonable because if cyclic topology is not needed it is better to use a conventional class, such as the STL **vector**.

One point of view is that the classes FlatPointArray, PointArray and PatchedPointArray can be considered specific cases of BasicArray, in any case, arrays. A loop (a contour) is something more. As a consequence, there are the classes LoopBase, BasicLoop, FlatLoop, Loop, PatchedLoop and CyclicLoop to implement loops. Before describing the loops, the derived classes of BasicArray (those used for loops) will be described since they contain a few more functions than BasicArray.

A FlatPointArray has functions *setX(int, double)*, *setY(int, double)*, *setZ(int, double)*, *double getX(int)*, *double getY(int)* and *double getZ(int)*, but for Z the result is independent of the integer index. The class has a protected data member *double _zeta* since the type of the array, **Doublet** does not have a Z component. The default topology is **Lup::CYCLIC_ARRAY**.

A PointArray has the same functions for setting and getting the position of a point as does FlatPointArray, except that the Z value can vary from point to point. The default topology is **Lup::CYCLIC_ARRAY**.

The class PatchedPointArray has the same position data access methods as FlatPointArray and PointArray, and the initial topology is **Lup::CYCLIC_ARRAY**. In addition, it has the functions

```
virtual int getGot(int i) const
virtual int getConnection(int i) const
virtual void setGot(int i, int got_in)
virtual void setConnection(int i, int connection_in)
```

to access the "got" and "connection" values at each point.

The three classes FlatPointArray, PointArray and PatchedPointArray can each be constructed from the other.

All three loop classes FlatLoop, Loop and PatchedLoop inherit from the templated class BasicLoop, which in turn inherits from LoopBase, which in turn inherits from XYZPntLoop.

Starting at the bottom, XYZPntLoop is a pure abstract class that declares the following functions:

```
virtual int size() const = 0;
virtual int length() const = 0;
virtual void setLength(int length_in) = 0;
virtual int topology() const = 0;
virtual void setTopology(int topology_in) = 0;
virtual void remove_item(int idx) = 0;
virtual double x(int i) const = 0;
virtual double y(int i) const = 0;
virtual double z(int i) const = 0;
virtual void setX(int i, double x_in) = 0;
virtual void setY(int i, double y_in) = 0;
virtual void setZ(int i, double z_in) = 0;
```

Aside from *remove_item*, these functions have analogs in BasicArray or the classes derived from BasicArray. So far, a loop is hardly different from a BasicArray. Many lines of class definitions are needed for loops without creating anything new. The approach that has been taken is that FlatPointArray, PointArray and PatchedPointArray should be considered intelligent arrays (like STL **vector**) the go beyond BasicArray only in that they provide access to the attributes of their points with some convenience functions (for example, *setGot(int, int)* and *setConnection(int, int)*) whereas the loop classes is where a developer should put functions related to loops used to form tubes. The role of XYZPntLoop is implied by its name: the abstract virtual functions tell us that every loop has points with x, y, and z values, as well as the additional basic functions listed above.

The next class in the loop hierarch is LoopBase

```
class LoopBase : public virtual XYZPntLoop
```

which does not contain an array of points, and yet, contains many algorithms. The class LoopBase defines many abstract virtual functions which, when combined with the abstract virtual functions of XYZPntLoop, are enough for the definition of concrete algorithms. The concrete algorithms are static functions, to give just a few examples:

```
static int inside(XYZPntLoop* lb, double x, double y)
static int chirality(XYZPntLoop* lb, int ifdebug)
static void nokinks_3d(XYZPntLoop* lb,
                       int numsteps,
                       double tolerance,
                       int ifdebug)
static double median_gap_2d(XYZPntLoop* lb)
static void segment_interpolate(XYZPntLoop* lb, ... )
static double interval_distance(LoopBase* lb,
                                int beg,
                                int end,
                                int dimen)
static void gen_midpnt(LoopBase* lb,
                       int which,
                       double segment_distance,
                       double total_distance,
                       double *mid_point)
```

The actual class used for the first parameter, *lb*, will be a class for which the virtual functions have an actual implementation. The static functions have corresponding abstract virtual functions declared, without the first parameter. In a derived class the virtual functions which correspond to the static functions call the static functions with the first parameter {\em this}.

The next step in the hierarchy is BasicLoop, which is a templated class that inherits from LoopBase.

```
template<class T>
class BasicLoop : public LoopBase { ... }
```

All of the abstract virtual functions of XYZPntLoop and LoopBase are implemented. A few examples from the class BasicLoop are:

```
void remove_item(int idx) {
  int k;
  for(k=idx;k<(length()-1);k++) {
    point_array[k] = point_array[k+1];
  }
  setLength((length() - 1));
}
```

```
double x(int i) const { return point_array.getX(i); }

void setZ(int i, double z_in) { point_array.setZ(i, z_in); }

int inside(double x, double y) {
  return LoopBase::inside(this, x, y);
}

int chirality(int ifdebug) {
  return LoopBase::chirality(this, ifdebug);
}

void gen_midpnt(int which,
                double segment_distance,
                double total_distance,
                double* mid_point) {
  LoopBase::gen_midpnt(this, which,
                       segment_distance, total_distance,
                       mid_point);
}
```

Some algorithms for loops, such as

```
void setChirality(int chirality_in,
                  int ifdebug)
void force_order()
```

are implemented in BasicLoop, but many are implemented in LoopBase.

As was the case for array classes, the are specific classes for specific values of the template parameter of BasicLoop. Few new methods are introduced in these specific classes, just data access and type conversion.

Explain that they just have simple data access and conversion, AND PatchedLoop has a few other methods.

class FlatLoop : public BasicLoop(FlatPointArray)

interpolate_lsf

The function interpolate_lsf is a C function that interpolates a segment of a curve as a third-degree polynomial. The number of points used for defining the coefficients of the polynomial can be greater than the number of coefficients. For each call it returns three points within the interval between two points in the middle of a curve segment. This function is somewhat specialized as it was developed for smoothing a contour for segmentation.

As input the function is given a set of two-dimensional points. The number points is *2*extent* and the pair points whose midpoint is interpolated have indices *(extent - 1)* and *extent*. The coordinate system is rotated so that the line connecting *pnt[extent - 1]* and *pnt[extent]* is horizontal and the interpolation polynomial gives y values (height) along x in the rotated system. As well as providing an interpolated midpoint, it returns two points a distance *frac* (towards the middle) from the central points of the original curve. For example, *frac = 1/6* gives points on the final curve that are evenly spaced if points on the original curve are evenly spaced.

Several noteworthy aspects of this function are the following. The interpolated points are not displacements of the original points because a pair of original points are used to define the interval, inside of which. three interpolated points are placed. The interpolated points should not be considered valid if the variable *is_bad_point* is set to true. For some arrangements of original points, the interpolated value can be far from a smooth curve.

The criteria for setting *is_bad_point* is the following. The function is given as an argument the circumference. If the contour were a circle, then for a given distance between a pair of points we know the displacement of a midpoint of a cord to reach the circumference. The interpolation function starts with the midpoint and displaces it according to the local curvature, so this displacement would be the distance between the midpoint of the cord and the circumference if the segment of the curve was part of a circle. This is one way to estimate how much displacement is realistic. But the contour is not a circle and another estimate of a reasonable displacement is to consider it relative to the length of the cord, for example, the displacement would be equal to half the cord if the interpolated midpoint for a ninety degree angle. The criteria for a "bad" point is when the displacement is more than twice displacement due to a uniform circle *unless* that critical distance is less than one quarter of a cord length. In the latter case, the original definition of the critical distance is considered too severe and the interpolated point is considered "bad" if its displacement is greater than a quarter of a cord length. A further constraint is that the critical distance can never be greater than half the cord length. If an interpolated point is tagged as bad, the magnitude of its displacement is reduced to the critical distance.

The actual implementation is slightly different. For example, where half the cord length is used in the above algorithm, the actual value is 0.9 times half the cord length. The details of the algorithm is based on what gave reasonable smoothing for a certain group of contours. Rather than making more coherent or more generalized this particular function, it is recommended that other similar functions be written, with this one left as an example.

interpolate_lsf_2

The signature of the function is shown below.

```
  void interpolate_lsf_2(double *x_pnts, double *y_pnts,
                         int extent,
                         unsigned char *immobile,
                         double immobile_weight,
                         double *final_x, double *final_y,
                         double *xn, double *yn,
                         double *xp, double *yp,
                         int *is_bad_point,
                         double total_distance,
                         double frac,
                         double max_angle,
                         int ifdebug)
```

The function **interpolate_lsf_2()** interpolates a point that is already on the curve, rather than interpolating points that are midway between the points at *(extent - 1)* and *extent* as is the case for **interpolate_lsf()**. The number of points in the curve segment is 2*extent + 1.

Impact on caller: Was

```
   xneg = (1.0 - frac)*x\_pnts[extent - 1] + frac*x\_pnts[extent];
   yneg = (1.0 - frac)*y\_pnts[extent - 1] + frac*y\_pnts[extent];
```

and now changed to

```
  xneg = frac*x\_pnts[extent - 1] + (1.0 - frac)*x\_pnts[extent];
  yneg = frac*y\_pnts[extent - 1] + (1.0 - frac)*y\_pnts[extent];
  xpos = frac*x\_pnts[extent + 1] + (1.0 - frac)*x\_pnts[extent];
  ypos = frac*y\_pnts[extent + 1] + (1.0 - frac)*y\_pnts[extent];
```

So now center is at index extent and also definition of frac has changed.

Number of points was extent*2 but now is (extent*2)+1 .

Must give array and variable

unsigned char *immobile

double immobile_weight value of two means, as if point existed twice

```
----------------------------------------------------------
class LoopBase : public virtual XYZPntLoop
```

```
does not yet contain array

class XYZPntLoop

is pure abstract
```

How is cyclic loop different from patched loop ? How is remove_item implemented in each derived class ?

```
----------------------------------------------------------
$ grep -l BasicArray *.hh *.C
basic_loop.hh
flat_point_array.hh  case of BasicLoop
flat_point_array.C
```

point_array.hh case of BasicLoop point_array.C

patched_point_array.hh case of BasicLoop patched_point_array.C

longitude.hh longitude_list.hh longitude_list.C

loop

patched_loop.C class PatchedLoop : public BasicLoop(PatchedPointArray), public virtual PatchedPntLoop

virtual functions of basic array base, and the generic mod: /// minimize array size virtual void renew_basic_array() = 0; /// set array size virtual void renew_basic_array(int size_in) = 0; /// get formal length virtual int length() const; /// set formal length virtual void setLength(int length_in); //@}

```
  /**@name static methods */
  //@{

  /// different from mod because -1 maps to (length_in - 1)
  inline static int generic_mod(int i, int length_in) {
 ----------------------------------------------------------
  /**@name virtual methods of BasicArrayBase */
  //@{

  /// delete all elements of array
  void renew_basic_array() {
    setSize(1);
    delete[] things;
    things = new T[_size];
  }
  /// allocate array to size size_in
```

```
void renew_basic_array(int size_in) {
  setSize(size_in);
  delete[] things;
  things = new T[_size];
}
/// get formal length
int length() const { return(BasicArrayBase::length()); }
/// set formal length, but no greater than size()
void setLength(int length_in) {
  if(length_in > size()) {
    resize(length_in);
  }
  BasicArrayBase::setLength(length_in);
}
//@}
```

basic_array_base.C basic_array_base.hh void setSize(int size_in); //@}

```
/**@name data members */
//@{

/// size of allocated array
int _size;
/// formal length of array
int _length;
//@}
```

public:

```
/**@name usual methods */
//@{

/// default constructor
BasicArrayBase();
/// constructor
BasicArrayBase(int size_in);
/// copy operator
BasicArrayBase(const BasicArrayBase& object_in);
/// assignment
BasicArrayBase& operator=(const BasicArrayBase& object_in);
/// virtual destructor
virtual ~BasicArrayBase();
//@}
```

```
/**@name data member accesss */
//@{

inline int size() const { return _size; }
//@}

/**@name virtual functions */
//@{

/// minimize array size
virtual void renew_basic_array() = 0;
/// set array size
virtual void renew_basic_array(int size_in) = 0;
/// get formal length
virtual int length() const;
/// set formal length
virtual void setLength(int length_in);
//@}

/**@name static methods */
//@{

/// different from mod because -1 maps to (length_in - 1)
inline static int generic_mod(int i, int length_in) {
  int j;
  if(length_in == 0) {
    j = 0;
  }
  else {
    if(i >= length_in*2) {
      j = i%length_in;
    }
    else if(i >= length_in) {
      j = i - length_in;
    }
    else if(i >= 0) {
      j = i;
    }
    else {
      j = (length_in - 1) - ((abs(i + 1))%length_in);
    }
  }
  return j;
}
//@}
```

basic_array.hh

flat_point_array.C

flat_point_array.hh

loop_array.C

loop_array.hh

patched_loop_array.C

patched_loop_array.hh

patched_point_array.C

patched_point_array.hh

point_array.C

point_array.hh

read_only_array.hh

basic_loop.hh

compressible_loop.C

compressible_loop.hh

cyclic_loop.C

cyclic_loop.hh

flat_loop.C

flat_loop.hh

loop.C

loop.hh

loop_array.C

loop_array.hh

loop_base.C

loop_base.hh

loop_namespace.C

loop_namespace.hh

loop_point.hh

loop_soft_error.C

loop_soft_error.hh

patched_loop.C

patched_loop.hh

patched_loop_array.C

patched_loop_array.hh

xyz_pnt_loop.hh

4.3 class CyclicLoop

The class **CyclicLoop** is a **BasicLoop** with template parameter **Patched-PointArray** . That is to say, each point of the loop can have a connection to another loop.

function connect_pair

This function connects the closest points of two loops. The signature is the following.

```
        void CyclicLoop::connect_pair(PatchedLoop* loop1,
                                      PatchedLoop* loop2,
                                      RelaxMode relax_mode,
                                      int ifdebug)
```

functions find_closest

```
        int CyclicLoop::find_closest(int ibeg, int iend,
                                     Triplet& p)
```

For points in the interval ibeg to iend of a **CyclicLoop** , this function finds the point closest to the position p. The value of iend can be less than ibeg. In any case, the search for the closest points is done from ibeg to ibeg + 1, ibeg + 2, etc. with the possibility of passing the last index and continuing from the first index.

When the entire loop is to be considered for finding the closest loop point to a given point in space, the virtual functions of **LoopBase** can be used.

```
        int find_closest(double x, double y)
        int find_closest(double x, double y, double z)
```

Considering all the distances between nearest neighbors, returns the distance for with half the nearest-neighbor distances are larger and half are smaller. (In two dimensions or three dimensions.)

class **LoopBase**

double LoopBase::median_gap_2d(XYZPntLoop* lb);

double LoopBase::median_gap_3d(XYZPntLoop* lb);

Repeats the following algorithm until no points are closer than gap_critical or until the number of cycles is equal to the number of points. The algorithm is that for each point on a loop the distance to another point is calculated for eight points ahead of the given point. For all these comparisons the minimum distance is found. Given the closest two points, which should be eliminated? The distance between neighboring points is calculated for the pair before and after each of the two closest points, that is, the new gaps are found if one or the other point where eliminated. The point eliminated is the one whose absence creates the smallest new gap.

class **LoopBase**

void LoopBase::remove_close_pnts_2d(XYZPntLoop* lb, double gap_critical);

4.4 Basic constants and simple functions

Sorting A few classes, **LoopBase**, **PatchedLoop** and **LongitudeList**, need sorting. The C functions

```
void integer_sortwithaux(int *list,
                         datum *aux,
                         int list_size);
void sort_double(double *list,
                 int *aux,
                 int list_size);
void sort_dble(double *list,
               int list_size);
```

have been implemented. In addition, the classical general-purpose database structure **datum** is defined.

Constants Constants are defined in the class **Lup**. In particular, the values of **LINEAR_ARRAY CYCLIC_ARRAY OPEN_CURVE** and **CLOSED_CURVE** are defined. The file *loop_namespace.hh*, which has these definitions, is constructed so as to be meaningful when included in a C file, C++ file or even a Fortran file.

Points Three point classes are defined, **Doublet**, **Triplet** and **PatchedPoint**. A **PatchedPoint** has a variable *got* to say whether a connection is being used and a variable *connection* to indicate the index of a point on an adjacent loop. A patched point has only one connection so a longitudinal line along a *patch* (local manifold) of a tube is a singley-linked list.

LoopSoftError The class **LoopSoftError** has the static variables

```
static int _bounds_error;
static int _topology_error;
static int _read_only_error;
```

which function like the C *errno*. For example, an array bounds error does not always cause the error to print but a test can be done for an error condition by testing *bounds_error()*. An example of use is shown below.

```
PatchedLoop& PatchedTube::operator[](int i) {
  PatchedLoop& t = loop_array[i];
  if(LoopSoftError::bounds_error() != 0) {
    if(write_count < 10) {
      fprintf(stderr,"Error in array bound\n");
      fprintf(stderr,"line %d file %s\n", __LINE__, __FILE__);
      write_count++;
    }
  }
  return t;
}
```

The variable *write_count* is a static variable of the class **PatchedTube**. The first ten bounds errors for the class **PatchedTube** cause an error message to be written, which is enough to tell the developer or user that something is wrong. More lines written would have the disadvantage of hiding (for example, by causing to scroll-away) other useful error messages.

Data For testing, tube data can be found in the following subdirectories:

```
smooth_166dx
smooth_286dx
smooth_286dx_bis
smooth_286sn
smooth_386dx
smooth_386sn
smooth_anastomosi
```

4.5 Classes to hold arrays

BasicArrayBase The class **BasicArrayBase** does not contain an array, it just holds the size and length. The number of valid elements is *length*, whereas, the size of the allocated space is *size*.

WithTopology The class **WithTopology** inherits from **BasicArrayBase** and adds a topology, either linear or cyclic.

BasicArray The class **BasicArray** inherits from **WithTopology**. It contains allocated space of a primitive array of type **T**.

FlatPointArray This section will not describe arrays of loops, but only arrays of points. A **BasicArray** of **Doublet**'s is a **FlatPointArray**.

PointArray A **BasicArray** of **Triplet**'s is a **PointArray**.

PatchedPointArray A **BasicArray** of **PatchedPoint**'s is a **PatchedPointArray**.

ReadOnlyArray The class **ReadOnlyArray** is an array whose writing-into can be prevented. This class is not now used.

4.6 Interpolation and matrix inversion

Interpolation The following C functions are used for interpolation:

```
void interpolate_lsf(double *x_pnts,
                     double *y_pnts,
                     int extent,
                     double *final_x,
                     double *final_y,
                     double *xn, double *yn,
                     double *xp, double *yp,
                     int *is_bad_point,
                     double total_distance,
                     int num_pnts,
                     double frac,
                     int ifdebug)

void make_baseline(BaseLine *horz,
                   double x1, double y1,
                   double x2, double y2,
                   double *dist_half_cord)

void move_to_baseline(BaseLine *horz,
                      double *xpnts, double *ypnts,
                      int num_pnts)

void get_from_baseline(BaseLine *horz,
                       double *xpnts, double *ypnts,
                       int num_pnts)
```

The interpolation is a least squares fit to a third degree polynomial. For a short line segment, a baseline is defined that is approximately parallel to the line segment, then all the points are rotated so that the variable to be interpolated is the height from the baseline. The number of points is typically greater than the number of unknowns (the four coefficients of the polynomial) so, in effect, the interpolation is a least squares fit.

Invert Matrix For interpolation, matrix inversion is needed. The inversion is done using the Fortran-based Lapack library. The Lapack library functions are called by the C function

```
int invert_matrix(double *matrix,
                  int size,
                   int ifdebug);
```

Linear Vector Mapping The linear interpolation between two loops requires a definition of a mapping that contains both a vector and a matrix. This mapping is defined in the structure **linear_vector_mapping**.

Four Points The functions *interpolate_4* and *do_interpol* do interpolation for just four points, but these functions have not been tested and are not now used.

LimitRange The class **LimitRange** is used for the conversion between primitive numerical types. The role is similar to a static cast. Since it operates on one number at a time, the conversion is not efficient. Nonetheless, the class may be useful for type conversion of fields between different steps of processing while avoiding compiler warnings. Here are some examples of the functions

```
static T limit_range(char s);
static T limit_range(unsigned int s);
static T limit_range(double s);
```

The primitive numerical types that can be used for the function parameter or return value are *char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double* and *long double.*

NumberTyped The class **NumberTyped** inherits from **Number** and implements the virtual functions of **Number**. This class contains one protected data member: **T _value**. It implements all of the virtual functions of **Number**, using **LimitRange** when numeric conversions are necessary.

Chapter 5

SMOOTHING

5.1 Package Smoothing

The package Smoothing contains algorithms for several techniques, including smoothing a two-dimensional loop and extrapolating the ends of tubes at a bifurcation.

Many of the classes in the package Loop are used in this package.

5.2 class Smoothing

The class **Smoothing** contains algorithms for smoothing a contour.

Smoothing Parameters

Parameters for the smoothing are data members of the class. These parameters are given default values when the class is constructed (the function *void init_variables()*). The smoothing parameters and their associated set/get functions are the following.

```
double _segment_fraction;
double segment_fraction() const;
void segment_fraction(double d);

// Before interpolation, the numsteps parameter for nokinks_2d.
int _nokinks_numsteps_before;
int nokinks_numsteps_before() const;
```

```
// Sets _nokinks_numsteps_before to a value between 0 and 64.
void nokinks_numsteps_before(int numsteps_in);

// Before interpolation, the tolerance parameter for nokinks_2d.
double _nokinks_tolerance_before;
double nokinks_tolerance_before() const;
// Sets _nokinks_tolerance_before to a value between 1.0 and
   128.0.
void nokinks_tolerance_before(double tolerance_in);

// After interpolation, the numsteps parameter for nokinks_2d.
int _nokinks_numsteps_after;
int nokinks_numsteps_after() const;
// Sets _nokinks_numsteps_after to a value between 0 and 64.
void nokinks_numsteps_after(int numsteps_in);

// After interpolation, the tolerance parameter for nokinks_2d.
double _nokinks_tolerance_after;
double nokinks_tolerance_after() const;
// Sets _nokinks_tolerance_after to a value between 1.0 and
   128.0.
void nokinks_tolerance_after(double tolerance_in);

double _gap_factor;
double gap_factor() const;
void gap_factor(double d);

int _goal_num_pnts;
int goal_num_pnts() const;
void goal_num_pnts(int goal_in);

double _fraction_averaged_curvature;
double fraction_averaged_curvature() const;
void fraction_averaged_curvature(double fraction_averaged_in);

double _frac_min_curvature;
double frac_min_curvature() const;
void frac_min_curvature(double frac_min_in);

double _frac_max_curvature;
double frac_max_curvature() const;
void frac_max_curvature(double frac_max_in);
```

The parameter *_segment_fraction* is the fraction of the total loop length that is used in the smoothing done by the function **segment_interpolate()**.

The paramter *_gap_factor* is a fraction that multiplies the average distance to arrive at *gap_critical* that is used in the call to **remove_close_pnts_2d()**.

```
gap_med = loop_inout->median_gap_2d();
gap_critical = gap_factor()*gap_med;
loop_inout->remove_close_pnts_2d(gap_critical);
```

The removal of close points is done in the function **smooth_contour()**. The functions **median_gap_2d()** and **remove_close_pnts_2d()** are defined in the class **LoopBase** .

The parameter *_goal_num_pnts* is used in **smooth_contour()** and **arrange-_num_dens()**. It is the final number of points in the contour.

The parameters *_fraction_averaged_curvature*, *_frac_min_curvature* and *_frac-_max_curvature* are used in the algorithm **arrange_num_dens()**.

The *_fraction_averaged_curvature* is the fraction of the loop used to determine the local curvature. The function **arrange_num_dens()** creates a higher density of points where the curvature is higher. The problem is that a curve that zig-zags might have a high curvature locally but perhaps the short-range zig-zag is to be eliminated.

The parameters *_frac_min_curvature* and *_frac_max_curvature* are an indirect way to control the number density in the function **arrange_num_dens()**. For each point, a local curvature is defined. Then the local curvature values are confined to a range between *_frac_min_curvature* and *_frac_max_curvature.*

Smoothing::angle_per_dist

The function signature is the following.

```
double angle_per_dist(double x1, double y1,
                      double x2, double y2,
                      double x3, double y3);
```

Used in the function **arrange_num_dens()**.

Smoothing::find_distance

The function signature is the following.

```
double find_distance(double x1, double y1,
                     double x2, double y2);
```

Smoothing::smooth_contour

The function signature is the following.

```
  void smooth_contour(FlatLoop* loop_inout);

  /*
     For a loop defined in two dimensions, a FlatLoop,
     the function smooth_contour makes the curvature more uniform.
     The first step is
     nokinks_2d(_nokinks_numsteps_before,
                _nokinks_tolerance_before, ifdebug);
     followed by
     remove_close_pnts_2d(_gap_factor*median_gap_2d());

     Then for each point, ipnt, there is the call
     segment_interpolate(ipnt,
                         flat_loop,
                         seg_fraction*sum_distance_2d(),
                         sum_distance_2d(),
                         after_x, after_y,
                         bad_point_item,
                         xn_array_item, yn_array_item,
                         xp_array_item, yp_array_item,
                         0.16666, ifdebug);
     The fraction 0.16666 creates points spaced 1/3 of
     the original distance.

     The final steps are
     nokinks_2d(_nokinks_numsteps_after,
                _nokinks_tolerance_after, ifdebug);
     followed by
     remove_close_pnts_2d(_gap_factor*median_gap_2d());
  */
```

Smoothing::arrange_num_dens

The function signature is the following.

```
  void arrange_num_dens(FlatLoop* loop_ptr,
                        int ifdebug);
```

Smoothing::standard_order

The function signature is the following.

```
  void standard_order(FlatLoop* loop_ptr,
                      int ifdebug);

  // Rotate curve so that it starts at smallest x.

  num_pnts = loop.length();
  smallest_x = loop.x(0);
  index_smallest = 0;
  for(i=0;i<num_pnts;i++) {
    if(smallest_x > loop.x(i)) {
      smallest_x = loop.x(i);
      index_smallest = i;
    }
  }

  // Shifts points so that point at first_point has index 0.

  void setFirstPoint(int first_point) {
    loop.setFirstPoint(index_smallest);
    if(total_twist > 0.0) {
      j = index_smallest;
      for(i=0;i<num_pnts;i++)
      {
loop.setX(i) = xpnts[j];
  loop.setY(i) = ypnts[j];
  j++;
  if( j >= num_pnts) {
  j -= num_pnts;
}
      }
    }
    else {
      j = index_smallest;
      for(i=0;i<num_pnts;i++) {
  loop.setX(i) = xpnts[j];
  loop.setY(i) = ypnts[j];
  j--;
  if( j < 0) {
  j += num_pnts;
}
```

```
      }
    }

 // Reverse directions of rotation of points relative to indexing.
  void setChirality(int chirality_in,
                    int ifdebug)

  // Have points listed by counter-clockwize rotation.
  loop.setChirality(1, ifdebug);
```

Smoothing::smooth_one_contour_01

The function signature is the following.

```
  void smooth_one_contour_01(FlatLoop* loop_tmp,
                             int ifdebug);


  /*
     The actual smoothing can involve smoothing
     more than once.
     One procedure that seems to work well for
     generating contours of 128 points is
     implemented in smooth_one_contour_01.
  */
```

Smoothing::segment_interpolate

This static function has the following signature.

```
  static void segment_interpolate(int ipnt,
                                  XYZPntLoop* lb,
                                  double segment_distance,
                                  double total_distance,
                                  double& after_x,
                                  double& after_y,
                                  int& bad_point_item,
                                  double& xn_array_item,
                                  double& yn_array_item,
                                  double& xp_array_item,
                                  double& yp_array_item,
                                  double frac,
                                  int ifdebug);
```

5.3 Function find_mapping

The C language function **find_mapping** finds a linear mapping between two contours. The mapping is specified by the structure LinVMap defined in the package **Loop** in the file *lin_v_map.h.* The theory is described in comments in the file *find_mapping.h* in the package **Smoothing**. The mapping LinVMap consists of a center position, an average displacement vector, and a matrix that describes the change of displacement (relative to the average displacement vector) as a function of position (relative to the center position). The input to this function is a set of two-dimensional vectors with defined on a plane, one displacement vector for each point on the plane. Redefining the problem so that the plane position is relative to the average point position and the displacement is relative to the average displacement, the problem becomes one of linear algebra, that of solving an overdetermined system of linear equations.

5.4 Program two_to_one

The program **two_to_one** uses many functions of the packages Loop and Smoothing to interpolate a bifurcation. The program is highly specific, certain files of contours are "hardwired" into the program. Moreover, the program writes pixmap files to show what it has done. This program can be used as an example for developing other programs.

The program reads the environmental variable **LOOPHOME** to find the contours used for the specific cases treated by this program. You just need specifiy the directory of the package *Loop* because the test cases (the subdirectories smooth_166dx, smooth_386sn, etc.) are part of the *Loop* package.

The program takes the arguments -p *directory* and -b *number*, where *number* is between 0 and 6. (Option -b signifies "base name index".) The *directory* is the location if the *.ppm output files. (Option -p signifies "pictures".)

For reading the contours, the set of C programs in the file *Loop/rw_tubatura.c* are used. Most of the programs in *Loop/rw_tubatura.c* are general-purpose, but there are a few specialized functions for testing. In particular, **get_tube_filenames()** is specialized for the filename of the datasets in the package Loop. Another specialized function in *Loop/rw_tubatura.c* is **read_tubatura02()** which assumes there is one major bifurcation and one of the branches can have a second bifurcation. If a particular branch (tube) is empty, certain values in the C structure TubeSegment must be forced to zero, since the language C does not have default constructors. A sample code fragment for initializing the structure TubeSegment is shown below.

```
// The structure TubeSegment is a C type, so it does
// not have a default constructor.
// The number of loops must be set to zero because
// most sample cases do not have a second split
// called bS and bD.
TubeSegment toob_1, toob_a, toob_b, toob_bS, toob_bD;
toob_1.count = 0; toob_1.loops = NULL;
toob_a.count = 0; toob_a.loops = NULL;
toob_b.count = 0; toob_b.loops = NULL;
toob_bS.count = 0; toob_bS.loops = NULL;
toob_bD.count = 0; toob_bD.loops = NULL;
```

The example above deals with five tubes simply because the (specialized for testing) program **read_tubatura02()** returns a set of five tubes and the debugging program **check01()** expects five tubes.

The C structures of type TubeSegment are converted into C++ structures of type Tube, for example

```
Tube tube1_tmp(toob_1);
```

We see above that the conversion is done by using a constructor operator. The program **two_to_one()** uses Tube objects such as **Tube tube1** so after conversion the assignment operator is used, as shown below.

```
tube1 = tube1_tmp;
```

The C structures such as **TubeSegment toob_1** and **Tube tube1** are defined within braces "{}" so that they are temporary. Before exiting from the braces, **free_tube** must be called because **read_tubatura02()** mallocs arrays that would not be freed automatically when the structure **TubeSegment** goes out of scope. Recall that TubeSegment is a C structure and hence does not have a destructor. An example code fragment is shown below.

```
// read_tubatura02 mallocs arrays inside of the tubes which
// need to be freed before the tubes go out of scope
free_tube(&toob_1); free_tube(&toob_a); free_tube(&toob_b);
free_tube(&toob_bS); free_tube(&toob_bD);
```

At this point in the algorithm **two_to_one()** we have the C++ classes of type **Tube**. By convention, for this particular group of test cases, the **tube1** is the main tube in which the bifurcation begins after the last contour and **tubea** & **tubeb** are branches in which the first loop after the bifurcation is at index 0. What we really need for this algorithm is a class **PatchedLoop** because it contains links to points on a neighboring loop. The loops of the tube (of type class **Loop**) can be accessed with the square bracket operator so the necessary **PatchedLoop** object are instantiated with a PatchedLoop constructor then assignment.

```
PatchedLoop loop1;
PatchedLoop loop1_tmp(tube1[k]);
loop1 = loop1_tmp;
```

There are no links between loops initially, which is imposed with the following lines of code

```
loop1.setAllGot(-1);
loop1.setAllConnection(-1);
```

The length of each loop is calculated and placed in double precision variables dist1, dista, distb.

Now an assumption is made, that the pair of points furthest away from each other on the main branch are furthest from the bifurcation and are in a segment (arc) similar to the pair of branches that combine at the bifurcation. One of these points on loop1 is associated with the closes point of loopa and the other is associated with the closest point on loopb. The loop1 is then partitioned into two pieces for which the ratio of the arc lengths is nearly equal to the ratio of the perimeters of loopa and loopb. This partitioning is done ”step by step” starting from the two extreme outside points. The overall idea is to associate a part of loop1 to loopa and to associate the rest of loop1 to loopb.

The next step in the algorithm is involves complicated point indexing that will not be explained. The end result are two points on loop1, called end2beg3 and end3beg2 which signify where the estimated projected loopa ends on loop1 and loopb begins (**end2beg3**) and where the estimated projected loopb ends on loop1 and loopa begins (**end3beg2**).

The first crude extrapolation of loopa to the position of loop1 is a **PatchedLoop** called *first_extrap_a* which contains the piece of loop1 between end3beg2 and end2beg3 plus a circular arc between end2beg3 and end3beg2. The analogous description applies to the object *first_extrap_a*. The algorithm can be seen in the actual code. A detail concerning the loop sizes need to be explained. An error occurs if one tries to index beyond the ”length” of a **PatchedLoop**. In addition, when copying the loop or generating a cyclic loop, only the first ”length” points are used. The number of points in loop1, called size1, is considered to sufficient for first_extrap_a and first_extrap_b. Initially, the objects first_extrap_a and first_extrap_b are allocated space **size1** then the ”length” is set to **size1** while they are being constructed, as shown below.

```
first_extrap_a = PatchedLoop(size1);
first_extrap_b = PatchedLoop(size1);
first_extrap_a.setLength(size1);
first_extrap_b.setLength(size1);
```

After they have been constructed, first_extrap_a and first_extrap_b must have their lengths set (using **setLength(actual_size)**) to be equal to the actual number of points in the construction so that functions such as **CyclicLoop::connect_pair()** will work correctly.

One sees in the program **two_to_one()** that first_extrap_a and first_extrap_b are draw using the class **PortablePixMap**. After which, static functions from the class **CyclicLoop** are used, as shown below.

```
CyclicLoop::connect_pair(&loopa, &first_extrap_a,
                         CyclicLoop::RELAX, ifdebug);
CyclicLoop::check_connect_pair(&loopa, &first_extrap_a);

CyclicLoop::connect_pair(&loopb, &first_extrap_b,
                         CyclicLoop::RELAX, ifdebug);
CyclicLoop::check_connect_pair(&loopb, &first_extrap_b);
```

Since loopa and first_extrap_a are type PatchedLoop, they have the data members to indicate a point-by-point mapping between the two. That is, each point has a data member *got* and *connection*. (Ditto for loopb and first_extrap_b.) The mode of operation *CyclicLoop::RELAX* specified in the call of connect_pair() tends to distribute the connections uniformly at the expense of having the connections not being strictly nearest pairs of points.

The function **find_offset()** defines a set of points *lup* and a set of vectors *offset*, one vector for each point. In the code fragment shown below, *lup* is given the points of *loopa* and the vectors in *offset* are vectors from *loopa* to *first_extrap_a* for the points connected by the function **connect_pair**.

```
PatchedLoop::find_offset(&loopa,
                         &first_extrap_a,
                         &lup, &offset,
                         ifdebug);


find_mapping(&lup,
             &offset,
             &mapping);

// Subroutine free_loop frees the array inside of a closed loop.
free_loop(&lup); free_loop(&offset);
```

The function **find_mapping()** shown above is a C function and as a consequence *lup* and *offset* are C structures of type **ClosedLoop**. Since they are C structures, the data allocated within these structures needs to be freed explicitly with the function **free_loop()**. The function **find_mapping()** derives an average displacement vector and a matrix transformation (scaling and rotation) (both in *mapping*) that is a "best fit" to the set of offset vectors. (Ditto for loopb and first_extrap_b.)

The next code fragment is shown for loopa and first_mapped_a and is also done for loopb and first_mapped_b. Notice that as well as allocating space for the **PatchedLoop** first_mapped_a, the "length" needs to be set in order to tell **apply_map()** how many points on which it should operate. The function argument *loopa* remains constant. The mapping *mapping* is applied to the points of *loopa* to generate the points of *first_mapped_a*.

```
first_mapped_a = PatchedLoop(256);
first_mapped_a.setLength(256);
PatchedLoop::apply_map(&loopa,
                       &first_mapped_a,
                       &mapping,
                       ifdebug);
```

To summarize, the first loop of each branch after the bifurcation is extrapolated towards the loop of the main branch before the bifurcation. The extrapolation is based on an approximate partitioning of the loop of the main branch into two pieces. Each of the two pieces of the main branch has an arc added in the gap where the loop was cut so that the extrapolation is based on a mapping between closed loops.

The loops *first_mapped_a* and *first_mapped_b* should be intersecting loops, that is, the union of the two should be similar to the loop of the main branch. The next step in the algorithm (code not shown) is to identify the points of *first_mapped_a* inside *first_mapped_b* and vice versa. Also, an adjustment is made in case the original loops do not all have the same sense of rotation. The next step in the program **two_to_one()** is to draw the results of the operations performed thus far.

The end result that is desired is to have extrapolated loops for branches A and B which include (in one or the other) all of the points of the loop of the main branch. The part of the extrapolated loop of branch A (the **PatchedLoop** called *first_mapped_a*) this is not inside *first_mapped_b* will not use extrapolated points but rather will use points of the main branch loop, whereas, the extrapolated points will be used for points of *first_mapped_a* that are inside *first_mapped_b*. Now consider the main loop and the intersecting loops *first_mapped_a* and *first_mapped_b*. Where *first_mapped_a* and *first_mapped_b* intersect (in two places) is not necessarily a point of either *first_mapped_a* or *first_mapped_b* and moreover is not necessarily a point of the loop of the main branch. Yet there should be a point on the loop of the main branch that is very near the intersection of *first_mapped_a* and *first_mapped_b*. It is possible to define a linear mapping on *first_mapped_a* and *first_mapped_b* such that they intersect at common points at two places, moreover, point that are also on the loop of the main branch. The program (code not shown) defines a few points in *remap_1* and *remap_2* which define a mapping that assures the coincidence of points as described above. The following mapping is then applied. (Ditto for first_mapped_b.)

```
PatchedLoop::find_offset(&remap_1,
                         &remap_2,
                         &lup, &offset,
                         ifdebug);
find_mapping(&lup,
             &offset,
             &mapping);
free_loop(&lup); free_loop(&offset);
PatchedLoop::apply_map(&holder_a,
                       &first_mapped_a,
                       &mapping,
                       ifdebug);
```

As a final step, the number of points in each extrapolated loop is set to 128.

5.5 Portable Pixmap

Portable Pixmap is a type of Portable Anymap developed by Jef Poskanzer.

The class **PortableAnyMap** contains just height and width data. The class **Shift** contains just center position and scale data. These classes are base classes for the class **PortablePixMap** which draws contours in portable pixmap format. One function, setBackground(), needs the hashing functions defined in *include/HASH/hash.h* and compiled in *lib/libhash.a*.

The functions in this class are not the most general-purpose, but rather, the functions were written to show the results of operations on contours. For example, different contours can be indicated by using **draw_cross()**, **draw_square()**, **draw_diamond()**, and **draw_circle()**. The class **PortablePixMap** is also useful for drawing a two-dimensional color-encoded field using **setRed()**, **setGreen()**, **setBlue()** and **setColor()**. All of the functions mentioned in this paragraph have two versions. In one version, the position is given as a pair of integers and in the other version the position is given as a pair of doubles. The latter version uses the x and y shift values and scaling of the base class **Shift**.

The function **setBackground()** looks for the most-used pixel value and changes only those locations to the new background color.

The function **write_binary(picture_file)** opens for writing and then closes the file *picture_file.ppm*.

An example code fragment is shown below

```
#include "Smoothing/portable_anymap.hh"

PortablePixMap* A_ppm = new PortablePixMap(width, height);
x = (xmin + xmax)*0.5; // x center of contour
A_ppm->setCenterX(x);
y = (ymin + ymax)*0.5; // y center of contour
A_ppm->setCenterY(y);
scale = 1.0/(dmax - dmin); // Scale by inverse of contour size.
// Add space for border region.
scale -= (2.0*border)/(sqrt(width*height)*(dmax - dmin));
A_ppm->setScale(scale);

// Note that black is indicated by color (1, 1, 1) instead
// of (0, 0, 0).  The background is (0, 0, 0), which will be
// changed to (255, 255, 255) using B<setBackground()>.
for( i = 0; i < sizea; i++ ) {
  A_ppm->draw_square(loopa.x(i), loopa.y(i), 1, 1, 1);
}

// ... other contours are drawn

A_ppm->setBackground(255, 255, 255);
// The name given to B<write_binary()> does not include the
// suffix .ppm, which is always appended.
A_ppm->write_binary(picture_file);
fprintf(stdout,"Writing picture file %s.ppm\n",
        picture_file);

// Free space used for the pixmap.
delete A_ppm;
```

Chapter 6

FILTERING

6.1 Image Filter

The image filters are independent of the segmentation program. Any set of filters must implement the virtual functions of **BasicImgFilter** , which are

```
virtual int filterDoIt(int w, int h,
                       const std::vector<float>& sigwin_slice,
                       std::vector<float>& filtered_sigwin_slice);

virtual void open_parameter_window();

virtual void hide_parameter_window();
```

In addition, all filters can use the signal of the base class

```
void filter_changed();
```

The choice of which filters are active and the choices of the filter parameters are done by using a GUI that is implemented by a particular filter set.

The *filterDoIt()* function is called when the image changes or when the filter parameters change. This function implements the filtering. Since *filterDoIt()* must be called by the segmentation program for a particular slice or set of slices, the segmentation program needs to know when a filter parameter has changed, hence the signal *filter_changed()*.

The filters use the convention that the values in the input image

```
  const std::vector<float>& sigwin_slice
```

are between 0.0 and 255.0. This is motivated by the fact that the filters already written usually make this assumption. In particular, the maximum value is important when using the inversion filter. Even though the range is limited, information is not lost because the type is floating-point. The filters can be used on the original image before it is discretized.

Two sets of filters have been constructed to test the package: files *filter/img_filter01.hh* and *filter/img_filter01.hh*, and files *filter/img_filter02.hh* and *filter/img_filter02.hh*. The two classes **ImgFilter01** and **ImgFilter02** are nearly identical.

There are two sets of filters in order to test the ability to change the filter sets while the program is running. The class **ImgFilter01** (or **ImgFilter02**) contains the image filter parameters. In this example implementation, there is also the class **ImgFilter01Params** (or **ImgFilter02Params**) which implements the GUI for changing the parameters. (Despite the name, **ImgFilter01Params** does not store the parameters, that is done by **ImgFilter01**.)

The files *filter_window.hh* and *filter_window.hh* provide an example driver for image filters, **FilterWindow**.

The logic of providing more than one filter set is provided by the class **FiltersetManager** in the files *filter/filterset_manager.hh* and *filter/filterset_manager.C*. The possible filter sets cannot be dynamically loaded, but rather, all the possibilities are instantiated in the constructor of **FiltersetManager**. The class **FilterWindow**, which displays the image and controls filtering, is an arbitrary construction so the class **FiltersetManager** is not required to know anything about the class (such as **FilterWindow**) that uses it. The public methods of **FiltersetManager** are:

```
void create_filterset_popup(QWidget* parent);
int filterDoIt_manager(int w, int h,
                       const std::vector<float>& sigwin_slice,
                       std::vector<float>& filtered_sigwin_slice);
```

and the public data members are:

```
QPopupMenu* filterset_popup;
std::vector<BasicImgFilter*> filterset_vec;
```

and the signals:

```
void no_filter_chosen(const QString& m);
void filterset_changed();
```

The function *create_filterset_popup()* needs to be given the widget *parent* onto which the popup menu is placed. The class **FiltersetManager** creates the popup menu and keeps track of the id numbers. The function *filterDoIt_manager()* is a simple wrapper for *BasicImgFilter::filterDoIt()* which allows the **FiltersetManager** to decide which filterset to use. The signal *no_filter_chosen()* is an error message and the signal *filterset_changed()* indicates that the image should be filtered again because the filter set changed.

Though an ideal organization of C++ programs would have a functional wrapper for the filter sets, for now, the complete set of filters is made public in order for the class **FilterWindow** (or similar) to connect between its filtering and the signal *BasicImgFilter::filter_changed()* of each filter set. The popup menu for the filter set selection is public because the class **FilterWindow** (or similar) needs to insert it into its menubar.

6.2 ContourGUI and CanvasContourGui

The class **ContourGUI** is an abstract class of virtual functions for a graphical user interface for manipulating contours. Primarily, the functions concern the mouse. What is done depends on the mouse event received and the state. The state is defined in the implementation **CanvasContourGui** by the variable *ContourGUI::MouseMode _mouse-_mode*. The value of _mouse_mode is only set by using the function *CanvasContourGui::setMouseMode(ContourGUI::MouseMode mm)*. As well as assigning the state of the mouse interface, the function defines the comment that is displayed in a text window for the purpose of indicating to the user the significance of using the mouse.

The class **CanvasContourGui** has one constructor in which a pointer to the class **VisualSupport** is set. The class **VisualSupport** is an abstract class defined in the subdirectory *contour*. This class contains functions for the visualization of the contour and the image; a wider range of functions than just the mouse interface. In the original develop of this package, there were two implementations of **VisualSupport**, one for a Qt canvas and one for OpenGL (the Qt interface to OpenGL). Due to the limited time available, the OpenGL implementation was dropped, whereas, the Qt canvas and the Qt image manipulation routines were retained. Nonetheless, the effort to support two visualization tools served to define the functions in **VisualSupport** as an abstraction of a visualization support for the segmentation.

In practice, the visualization of the image and the contour is done with a Qt canvas implemented by **SliceCanvas** . This class implements all of the **VisualSupport** functions and contains a data member that is a pointer to **ContourGUI** which is actually a **CanvasContourGui**. The **VisualSupport** needed by the contructor of **CanvasContourGui** is *this*, the **SliceCanvas** which contains the pointer to **ContourGUI**. The canvas implementation has a helper class for the mouse interface, the class **FigureEditor**.

The **FigureEditor** uses the same instantiation of **ContourGUI** as the **SliceCanvas** class.

By the way, the abstract class **VisualSupport** has the functions *getMouseMode* and *setMouseMode*. The implementation in **SliceCanvas** is to call *_contour_gui-getMouseMode* and *_contour_gui-setMouseMode* where *_contour_gui* is an instantiation of **CanvasContourGui** .

Some of the **CanvasContourGui** data members for which a explanation may be useful are the following.

The mouse-contour interface needs to control the source of events with regard to two functions *void set_mouse_tracking(bool enable)* and *bool has_mouse_tracking() const*. For the canvas interface, the source of mouse events is the **FigureEditor**. The class **FigureEditor** is derived from **MouseEventSource** which is an abstract class defined in the file *contour/mouse_event_source.hh* with only those two functions. The class **CanvasContourGui** has the variable *MouseEventSource* _mouse_event_source* which actually points to an instantiation of **FigureEditor**. The overall idea is that if a different GUI is used, such as OpenGL, perhaps the necessary control of the source of events will be analogous and therefore rather than implementation-specific calls, the function calls will be made to the abstract class **MouseEventSource**.

The private logical variable *bool _contour_op_finalized* is for dealing with the case in which the user starts a new operation without finishing the old operation, which is rather easy to do. The user might be indicating that some points are to be held fixed, the press the button that starts the implementation of dragging points to new locations. For most operations, the second mouse button is used to indicate that a particular type of operation is finished, whereas in this scenario the user changed mode without pressing the second mouse button. If a new operation begins and _contour_op_finalized is false, then some final steps are done for the previous operation. In this regard, the background colors are of particular relevance. As an option, the user can request that the background display some fields that influence the active contour (the snake). Some of these field depend on the actual point positions, such as a threshold image value that is relative to local point positions. The calculation of these fields requires several seconds, so is only done when switching modes of operation, for example, when the second mouse button is pressed to indicate "done". Waiting twice for updating of the background field colors can be frustating for the user, so it is important that finalization be done once and only once when switching modes.

The variables *int highlighted_index, prev_highlighted_index* are used for highlighting points as the mouse moves from one point to another. The actual color of a point depends on its state, for example, whether it has been declared fixed (immobile). Being highlighted or not is orthogonal to being fixed or not. The actual color of a point when being drawn is known by the point, so a point needs to be informed as to whether it is should draw itself with the highlighted color choice. With regard to the mouse position, by definition only one point can be chosen at a given time and therefore only one point can be highlighted. The actual algorithm is complicated because the highlighting depends on the mouse_mode (the operation being performed), and moreover, the actual change of color required that the contour be redrawn.

head2 Adding an evolution parameter

Files that need to be changed:

contour/evolve_params.hh

contour/evolve_params.C

contour/vis_evolve_params.hh

contour/vis_evolve_params.C

contour/simple_contour.hh

contour/simple_contour.C

```
  double _tension;
  double _viscosity;
```

contour/evolve_params.hh: has the parameters as data members.

contour/evolve_params.C: implements constructor, copy constructor and assignment operator.

contour/vis_evolve_params.hh: declares method void set_VAR(VAR_TYPE VAR_in) and signal void signal_set_VAR() where VAR is the name of the variable which has type VAR_TYPE.

contour/vis_evolve_params.C: implements void set_VAR(VAR_TYPE VAR_in) Also, handles setting current->_VAR in either void setVEP(int i, double value) or void setVEP(int i, int value) and returning the value from VEPEntry getVEP(int i) const

contour/simple_contour.hh: declares references to the variables in EvolveParams _evolve_params; Either double& _VAR; or int& _VAR; The constructor and copy constructor defines the references _VAR(_evolve_params._VAR) Note that the constructor, copy constructor and assignment operator of SimpleContour set _vis_evolve_params._current to be the address of _evolve_params.

contour/simple_contour.C: in the function void init_vars() the initial value of each VAR& is set, the minimum and maximum of the variable is set in the data member VisEvolveParams _vis_evolve_params; The default value in _vis_evolve_params._default is set to the initial value. Also, the name of the variable as a QString is set in init_vars()

6.3 DICOM read

ImgSequence

The class **ImgSequence** reads a set of files that represent a DICOM image. The name of any DICOM file needs to be given to the constructor **ImgSequence(const char* dicom_in)**. The last part of the file name should be a number that the program strips off to get a prefix, then all files with the same prefix plus a number are ordered into a list of files. A specific slice is selected using **slice2image(int slice_number)** where the first slice number is zero. After the function call, the function **usSlice()** will return a reference of type

```
const std::vector<unsigned short>&
```

Only one slice is in memory at a given time, in order to minimize the memory usage. The user can make copies and store all the slices, if for example, the application requires thumbnail sketches of all the slices.

The actual reading of the DICOM files is done by functions defined in the package DX_DICOM because that package has a mature version of a DICOM reader. The DX_DICOM package uses the package CTN, but the functions of **ImgSequence** do not use any CTN functions or CTN constants because the DX_DICOM functions serve as wrappers to hide the details of CTN. However, the header files and libraries for DX_DICOM, CTN, and DXIO are needed during compilation. The package DXIO reads and writes IBM Data Explorer files for a simple lattice. Even though Data Explorer is not used, there is an error during linking because the library of DX_DICOM functions references functions in the package DXIO.

Chapter 7

CONTOUR

7.1 CONTOUR: Image Filter

The image filters are independent of the segmentation program. Any set of filters must implement the virtual functions of **BasicImgFilter**, which are

```
virtual int filterDoIt(int w, int h,
                       const std::vector<float>& sigwin_slice,
                       std::vector<float>& filtered_sigwin_slice);

virtual void open_parameter_window();

virtual void hide_parameter_window();

virtual void setMask(int w, int h,
                     const QBitArray& bit_array_in);
```

In addition, all filters can use the signal of the base class

```
void filter_changed(int id);
```

To simplify the many calls for emitting a signal, so that derived classes can ignore the filter ID, there is defined the function

```
void emit_filter_changed();
```

which causes *filter_changed(int id)* to be emitted.

Moreover, the class **BasicImgFilter** manages the filter identification number with the function

```
void setRunTimeID(int id);
```

All filters, including the base class **BasicImgFilter**

implement the copy constructor and assignment operator because the filters are placed in a vector and may be copied during the *push* onto the vector.

The choice of which filterset is active and the choices of the filter parameters are done by using a GUI. Each filterset implements its own GUI.

The *filterDoIt()* function implements the filtering. External routines should call this function (indirectly) by calling *filterDoIt_manager()*. Since *filterDoIt()* must be called by the segmentation program for a particular slice or set of slices, the segmentation program needs to know when a filter parameter has changed, hence the signal *filter_changed(int id)*.

While running the segmentation program, a user can change the filterset, and as a consequence, a FiltersetManager is in the middle of the communication between the filtersets and the segmentation program. In particular, the segmentation program calls **FiltersetManager::filterDoIt_manager()** which then calls filterDoIt() for the filter that is active. With regard to communication in the other direction, a particular filter calls *emit_filter_changed()* which then emits the signal *filter_changed(int id)*, however, this signal is caught by **FiltersetManager** and the segmentation part of the program should expect to receive **FiltersetManager::filter_changed()**. The **FiltersetManager** knows which filterset is active whereas the segmentation part of the program sees the same interface to the filter.

The filters use the convention that the values in the input image

```
const std::vector<float>& sigwin_slice
```

are between 0.0 and 255.0. This is motivated by the fact that the filters already written usually make this assumption. In particular, the maximum value is important when using the inversion filter. Even though the range is limited, information is not lost because the type is floating-point. The filters can be used on the original image before it is discretized. Moreover, in most cases the filters will work correctly with an arbitrary range of values for the field.

Two sets of filters have been constructed to test the package: files *filter/img_filter01.hh* and *filter/img_filter01.hh*, and files *filter/img_filter02.hh* and *filter/img_filter02.hh*. The two classes **ImgFilter01** and **ImgFilter02** are nearly identical. There are two sets of filters in order to test the ability to change the filter sets while the program is running. The class **ImgFilter01** (or **ImgFilter02**) contains the image filter parameters. In this example implementation, there is also the class **ImgFilter01Params** (or **ImgFilter02Params**) which implements the GUI for changing the parameters. (Despite the name, **ImgFilter01Params** does not store the parameters, that is done by **ImgFilter01**.)

There is a third filterset, class **ImgFilter03**, which is likely to be useful for segmentation. This filterset provides the possibility of inverting the grayscale (black become white, etc.); after which, there is the option of using a mask the blacks-out a part of the image; there is a median filter; there are gaussian smoothing filters with radii 3, 5, and 7; the penultimum filter is a gamma correction; and the last filter is a histogram equalization with a tranfer function that is a continuous range between the identity and full histogram equalization.

The mask that black-out a part of the image is set by *setMask()* of a given filter, a function that is accessed by *setMask_manager()* of the **FiltersetManager** so that only the mask of the active filter is changed. The mask actually specifies pixels that should not be masked and the **ImgFilter03** has a slider to select a radius beyond the specified pixels which should not be masked. The logic of this procedure is that the pixels of the mask can be the contour points, or the contour points plus some points to indicate the inside of the part selected from the image. This could be useful in the first pass of segmentation in which it is convenient to remove other features from the image. In particular, an rapid approximate segmentation can be done that generates contours that simply give an indication of the area of interest. The points of these contours can be given to the filter and the user can select (using the GUI of the filter) a radius that includes an image area slightly broader than the approximate contours. All of the slices (the images) can then be written using the masking of the filter.

There are at least two advantages to doing segmentation using slices that have first been masked. One advantage is that the dynamic range can be reduced and a greater range of values can be given to the gray levels most relevant for the segmentation. The actual segmentation is done with 8-bit data whereas the original data could have an amplitude as large as 12 bits. As an example, if an image includes bone whereas the area to be segmented is an artery, a first pass that eliminates the bone can help to reduce the range of values to those important for the artery so that in the second pass of segmentation the important range does not suffer information loss when going from a DICOM file to an 8-bit image or when going from a Data Explorer file of *unsigned short* to 8 bits. (A Data Explorer file can be of any primitive numerical type, the choice of *unsigned short* for segmentation has been made because it is sufficient for all medical images we expect to encounter.) A second advantage is realized when viewing the dataset in three dimensions. The first pass of segmentation, even if not precise, can clear away extraneous features that block the three-dimensional inspection of the relevant part.

The files *filter_window.hh* and *filter_window.hh* provide an example driver for image filters, **FilterWindow**.

The program logic of providing more than one filterset is demonstrated by the class **FiltersetManager** in the files *filter/filterset_manager.hh* and *filter/filterset_manager.C*. The possible filter sets cannot be dynamically loaded, but rather, all the possibilities are instantiated in the constructor of **FiltersetManager**. The class **FilterWindow**, which displays the image and controls filtering, is an arbitrary construction in the sense that the class **FiltersetManager** is not required to know anything about the class that uses it. The public methods of **FiltersetManager** are

```
void create_filterset_popup(QWidget* parent);
int filterDoIt_manager(int w, int h,
                       const std::vector<float>& sigwin_slice,
                       std::vector<float>& filtered_sigwin_slice);
int setMask_manager(int w, int h,
                    const QBitArray& bit_array_in);
```

and the public data members are

```
QPopupMenu* filterset_popup;
std::vector<BasicImgFilter*> filterset_vec;
```

and the signals

```
void no_filter_chosen(const QString& m);
void filterset_changed();
```

The function *create_filterset_popup()* needs to be given the widget *parent* onto which the popup menu is placed. The class **FiltersetManager** creates the popup menu and keeps track of the id numbers. The function *filterDoIt_manager()* is a simple wrapper for *BasicImgFilter::filterDoIt()* which allows the **FiltersetManager** to decide which filterset to use. The signal *no_filter_chosen()* is an error message and the signal *filterset_changed()* indicates that the image should be filtered again because the filter set changed.

7.2 Mouse-Contour Interaction

Different graphical user interfaces will have different algorithms for changing the contour when interacting with the mouse. For example, to zoom-in on a part of an image using a **QCanvas** the original image is enlarged and as a consequence the position of the mouse from a QMouseEvent is scaled proportionally. In contrast, for an OpenGL pixmap there is a zoom function **glPixelZoom()** to enlarge the display but the mouse coordinates from a QMouseEvent are based on the original size.

ContourGUI

The class **ContourGUI**

is an abstract class of GUI functions directly related to the contour.

The though this class need not be restricted to mouse events, in practice the mouse events provide the mouse-contour interaction.

One possible program organization would be to enable mouse events within certain methods which, for example, create an initial contour or move a point of the contour. Upon receiving the mouse event, the thread of control would return to the specific algorithm for creating a contour or moving a point.

This implementation uses a different program organization. The primary consideration for actual program organization is that for a given type of window, there is a specific class which receives mouse events inside the window. The methods of the specific class call methods of class **ContourGUI**.

For example, for a **QCanvas** mouse events are received by **QCanvasView**. The actual implementation for a canvas uses the derived class **FigureEditor** in which the functions

contentsMousePressEvent(),

contentsMouseMoveEvent() and

contentsMouseReleaseEvent()

call respectively

_contour_gui->**mouse_press_event()**,

_contour_gui->**mouse_move_event()** and

_contour_gui->**mouse_release_event()**

where *_contour_gui* is a pointer of type **ContourGUI**.

As a second example, for OpenGL pixmap mouse events are received by **QGLWidget**. The actual implementation for this type of window uses the derived class **MyGLDrawer** in which the functions

MousePressEvent(),

MouseMoveEvent() and

MouseReleaseEvent()

call respectively

_contour_gui->**mouse_press_event()**,

_contour_gui->**mouse_move_event()** and

_contour_gui->**mouse_release_event()**

where *_contour_gui* is a pointer of type **ContourGUI** .

The class **ContourGUI** defines an enumeration type MouseMode.

See the head2:enum MouseMode entry elsewhere in this document.

The class declares the virtual functions

virtual void mouse_move_event(QMouseEvent* e) = 0;

virtual void mouse_press_event(QMouseEvent* e) = 0;

virtual void mouse_release_event(QMouseEvent* e) = 0;

whose actual implementations depend on the window implementation.

The class also declares

virtual void setMouseMode(MouseMode mm) = 0;

virtual MouseMode getMouseMode() const = 0;

virtual QString getComment() const = 0;

virtual void init_operation() = 0;

The first method sets the mouse mode. The next two methods return the current values of the mouse mode or the comment shown to the user concerning the current mouse mode. The last method in the list does initialization for a procedure based on the current mouse mode.

The sequence of calls for interacting with the contour forms a rather long chain. The procedure of indicating some points to define an initial contour will be used as an example. There may be a class that implements a panel of buttons for initiating various types of interactions. The next step is simple, the class that implements the panel needs to call a function that initiates the state of the mouse interaction. By convention, that function is called **runByHand()** in a class that derives from **VisualSupport**. The function **runByHand()** is not a virtual function and therefore the class that contains the function's class must be explicit for the button's callback. By convention, the function is placed in a class that derives from **VisualSupport** because this is a fat class of the GUI. (The actual classes are **ContWind** and **SliceCanvas**.)

An implementation of **runByHand()** is the following

```
void SliceCanvas::runByHand() {
   setMouseMode(ContourGUI::mm_addPoints);
   _contour_gui->init_operation();
}
```

The call made to **setMouseMode()** happens to be a virtual function of **VisualSupport** for convenience and has the simple implementation of

```
void SliceCanvas::setMouseMode(ContourGUI::MouseMode mm) {
  _contour_gui->setMouseMode(mm);
}
```

Finally we have arrived at **ContourGUI**. The actual implementation of

ContourGUI::setMouseMode()

is shown below

```
void CanvasContourGui::setMouseMode(ContourGUI::MouseMode mm) {
  _mouse_mode = mm;
  QString comment;
  if(_mouse_mode == ContourGUI::mm_Ignore) {
    QString s("");
    comment = s;
  }
  // [ ... ] other mouse modes
  if(_mouse_mode == ContourGUI::mm_addPoints) {
    QString s("M1 : add point, M2 : done");
    comment = s;
  }
  _comment = comment;
  _visual_support->GUI_use_comment(comment);
}
```

The second operation invoked by **runByHand()** does any initialization that may be required. An example of an implementation is shown below.

```
void CanvasContourGui::init_operation() {
  if(_mouse_mode == ContourGUI::mm_Ignore) { }
  // [ ... ] other mouse modes
  if(_mouse_mode == ContourGUI::mm_addPoints) {
     _contours->refCurve().clear();
  }
}
```

The function **refCurve()** returns a reference to a workspace that will be used to store the points that are added.

Note that the code segments shown for classes **SliceCanvas** and **CanvasContourGui** are for the canvas implementation and are the same for the OpenGL implementation in the classes **ContWind** and **GLContourGui**.

Nothing more needs to be done until a mouse button is clicked over the window that shows the image. Notice that we bounce back to a **VisualSupport** class. But just for a moment to make the comment visible to the user.

Because **CanvasContourGui** and **GLContourGui** implement virtual functions of **ContourGUI**, the number of functions are limited to reduce the editing work needed when new contour operations are added. As possible, the graphical user interface affects the contours using a class derived from class **ContourGUI**. However, the fat GUI that inherits from VisualSupport has a pointer to the class that holds the contours whereas the class derived from **ContourGUI** does not have a pointer to the contours.

enum MouseMode

The mouse modes correspond to specific contour operations.

mm_Ignore

Ignore mouse events for the image window.

mm_initContours

Initialize contours.

mm_putSeed

Similar to initialize contours.

mm_passHere

Moves one point.

mm_fixHere

Selects one point that becomes fixed. It does not move despite forces from the contour algorithm.

mm_releaseHere

Releases a point that was fixed.

mm_addPoints

Used for generating a contour from points specified by hand.

```
class CanvasContourGui : public ContourGUI {
  public:
  CanvasContourGui(VisualSupport* visual_support_in);
  virtual function of ContourGUI
  void mouse_move_event(QMouseEvent* e);
  // virtual function of ContourGUI
  void mouse_press_event(QMouseEvent* e);
  // virtual function of ContourGUI
  void mouse_release_event(QMouseEvent* e);
  // virtual function of ContourGUI
  void setMouseMode(ContourGUI::MouseMode mm, QString comment);
  // virtual function of ContourGUI
  ContourGUI::MouseMode getMouseMode() const;
  // virtual function of ContourGUI
  QString getComment() const;
  private:
  VisualSupport* _visual_support;
  QCanvasItem* moving;
  QPoint moving_start;
  ContourGUI::MouseMode _mouse_mode;
  QString _comment;}

  VisContours* getVisContours();   // needed or not?
};

  class GLContourGui : public ContourGUI {
    public:
    GLContourGui(VisualSupport* visual_support_in);
  // virtual function of ContourGUI
  void mouse_move_event(QMouseEvent* e);
  // virtual function of ContourGUI
  void mouse_press_event(QMouseEvent* e);
  // virtual function of ContourGUI
  void mouse_release_event(QMouseEvent* e);
  // virtual function of ContourGUI
  void setMouseMode(ContourGUI::MouseMode mm);
  // virtual function of ContourGUI
  ContourGUI::MouseMode getMouseMode() const;
  /// virtual function of ContourGUI
  QString getComment() const;
  // virtual function of ContourGUI
  void init_operation();
  private:
  VisualSupport* _visual_support;
  ContourGUI::MouseMode _mouse_mode;
  QString _comment;
  QPoint moving_start;
  int moving_index;
}

  VisContours* getVisContours();
};
```

7.3 Introduction

Many public domain software packages use the GNU (Free Software Foundation, Inc.) tools, procedures and conventions for the compilation and installation of software. By following this convention, a new user knows the procedure for software installation. For example, in the most simple case the user can type

```
./configure
make
make check
make install
```

to have software installed under the directory */usr/local*, independently of the computer architecture; as long as the architecture has been forseen by the developer when preparing the **configure** shell script. In my own work, I often use the command

```
./configure --prefix=destination_directory
```

where *destination_directory* is the directory under which the software will be installed. For a very large number of public domain packages, I know that I can control the installation directory by using the "–prefix" option.

In addition to the convenience for the user, the GNU tools such as **autoconf** provide a convenient method for writing portable (multi-platform) software.

Test of C++

This document describes methods and tools for organizing multi-platform software. Rather than describing a wide range of possibilities, this document gives details concerning one specific procedure. By being specific, the recommendations of this document can be used as guidelines for a software development project. The recommendations go beyond the use of **autoconf** tools, for example, there is a description of **stow** for handling the installation of many packages under one hierarchy while retaining information about the package and version of the files.

7.4 Overview

The following subsections give a broadbrush description of the topics covered in this document.

ConfigTools

See the head1:Overview entry elsewhere in this document

In the initial years of CRS4, each architecture had an NFS mount to an architecture-specific hierarchy called */usr/crs4*. Later, there was added a single mountpoint */u/crs4* with architecture-specific subdirectories.

Plain Old Documentation

The source of this document is in the POD format, defined by Larry Wall. This format is used in the documentation of Perl modules. The format specifications can be displayed with the command "man perlpod". Conversion from POD format do Latex format is done with "pod2latex". The original pod2latex program has some errors, for example, the second-level header creates a "section" instead of a "subsection". As a consequence, a slightly modified version of pod2latex was used to created this document.

The fundamental advantage of pod2latex is that it protects from Latex translation the symbols '$' and '_' (underline). These two symbols are used very often in naming variables in shell scripts and Makefile files but for Latex these symbols have meanings related to mathematical formulas. More generally, by using the POD format a minimum of formatting symbols in the original text is sufficient for obtaining a nicely formatted Postscript file; using Latex as an intermediate and converting to Postscript with "dvips".

Genoa Active Message MAchine (GAMMA)

(http://www.disi.unige.it/project/gamma/)

Bibliography

[1] K. F. Lai, "Deformable Contours: Modeling, Extraction, Detection and Classification,"*Phd Thesis, Electrical Engineering* , University of Wisconsin-Madison, 1994.

[2] K. F. Lai & R. T. Chin, "On Regularization, Formulation and Initialization of the Active Contour Models (Snakes)," *Asian Conference on Computer Vision*, 1993, pp. 542-545.

[3] K. F. Lai & R. T. Chin, "Deformable Contours: Modeling and Extraction," *IEEE Int. Conf. on Comp. Vis. Patt. Recog.*, 1994, pp. 601-608.

[4] K. F. Lai & R. T. Chin, "On Classifying Deformable Contours Using the Generalized Active Contour Model," *Third Int. Conf. on Automation, Robotics & Comp. Vis.*, 1994, pp. 930-934.

[5] D. H. Ballard, "Generalizing the Hough Transform to Detect Arbitrary Shapes," *Pattern Recognition*, vol. 13, 1981, pp. 111-122.

[6] A. A. Amini, T.E. Weymouth & R. C. Jain, "Using Dynamic Programming for Solving Variational Problems in Vision," *IEEE Trans. Pat. Anal. Mach. Intell.*, vol. **PAMI**-12, no. 9, 1990, pp. 855-867.

[7] D. J. Williams & M. Shah, "A Fast Algorithm for ve Contours and Curvature Estimation," *Computer Vision, Graphics, age Processing*, vol. 55, 1992, pp. 14-26.

[8] P. J. Burt & E. H. Adelson,"The Laplacian Pyramid a Compact Image Code" *IEEE Trans. on Commun.*, vol. **COM**-31, o. 4, 1983, pp. 532-540.

Appendix A

GSNAKE

A.1 About this manual

This manual is meant as a programmers' reference to the GSNAKE API. It covers all the classes and explains the use, arguments and return values of each of the methods. All protected methods and members are not covered in the manual.

Throughout the manual, all class names are in CAPITAL and `typewriter` font, when the item being referred to is the object as an entity. When it is the member or data structure of the object that is being referred, the name will be in `typewriter` font.

At the end of the reference for each class, some example programs are provided. These are workable program segments for users to familiarize with the workings and application of the API.

The manual is organised as follows:

- Chapter 1 explains the design of the API.
- Chapter 2 contains the references to each class and their methods.
- Chapter 3 covers the various utilities provided with the API.

A.2 How to Avoid Reading this Manual

The examples after the reference of each class, as well as the utilities programs, provide a rather rich collection of applications one can perform using the GSNAKE API. One may start by looking at these programs, and treat this manual as what it is : a *reference* manual.

A.3 Design of the GSNAKE API

The GSNAKE API is a set of fundamental codes, based on the Generalised Active Contour Model [1], suitable for use in the area of feature extraction, image detection and classification, and motion analysis.

The API consists of a set of objects built in C++. These are described in the following section.

A.4 Class Design and Implementation

class IMAGE

`IMAGE` provides routines for manipulating and displaying image data. These includes histogram conditioning, image correlation and image smoothing.

class EDGE

`EDGE` computes edge map of an image. It consists of two `IMAGE` objects that marking the magnitude and direction of each edge point.

class PYRAMID

`PYRAMID` is built up of a series of `EDGE` and `IMAGE` objects in a pyramid form. The higher level `IMAGE` is a reduced version of the lower image in that both resolution and sample density are decreased. By computing and conditioning edge map from each level of `IMAGE`, we obtain `EDGE` objects.

class MATRIX

`MATRIX` provides advanced routines for matrix addition, substraction, multiplication, inversion and transpose. It is inherited with extra members and methods from `IMAGE class`.

class SNAXEL

A `SNAXEL` is a point on a contour. It consists of methods for snaxel angle and energy calculation.

class CONTOUR

`CONTOUR` is a linked list of `SNAXEL` originated at image origin or contour center. It consists of shape matrix and internal energy term to represent shape irregularities.

class GHOUGH

`GHOUGH` localizes `CONTOUR` of a particular shape from an `IMAGE` or `EDGE` by generalized Hough transform.

class GSNAKE

`GSNAKE` consists of `PYRAMID` for external energy calculation and `CONTOUR` for internal energy calculation. It localizes `CONTOUR` by generalized Hough transform, minimizes energy by dynamic programming with stratified line search algorithm, and regularizes parameters tradeoff by minmax criterion or local/global parameters selection strategy.

class MODEL

With sufficient image samples, `MODEL` trains a robust `CONTOUR` model with prior knowledge of shape matrix and local regularization parameters. It is inherited with extra members and methods from `GSNAKE`.

class CLASSIFY

`CLASSIFY` classifies various `CONTOUR` models and compute their corresponding score from an image. Classifying methods include marginalization of the distribution, MAP probability, match of deformable template and rigid template.

Appendix B

GSNAKE Class Library Reference

B.1 IMAGE : The raw image object

`IMAGE` is a class for manipulating and displaying image data. Image data can be stored, retrieved, copied, cut, and viewed for different purposes. In addition, histogram processing, image correlation and Gaussian kernel generation are also provided for image enhancement and smoothing purpose. An `IMAGE` object has the following structure :

```
CLASS IMAGE {
      protected :
         float *data;
         int row, col;

      public :
         XIMAGE *ximg;
}
```

Image data is built up of a matrix of size row x col, which can be generated as an X window image.

B.1.1 IMAGE Constructor

Synopsis

`IMAGE()`

Description

The constructor initializes an image data of matrix size 0 x 0, and sets the X window image pointer to NULL.

B.1.2 IMAGE Destructor

Synopsis

```
~IMAGE()
```

Description

The destructor frees the memory allocated to the image data and the X window image.

B.1.3 Resetting an IMAGE

Synopsis

```
reset()
```

Description

`reset` allows the reuse of an `IMAGE` object. The memory allocated previously is freed and the image data of matrix size 0 x 0 is initialized.

B.1.4 Initializing image matrix

Synopsis

```
int init(int _row, int _col);
```

Arguments

`_row`	The number of rows in an image.
`_col`	The number of columns in an image.

Returns

`NOERROR`	Memory allocation is successful.
`MEMORYERROR`	Otherwise.

Description

`init` allocates memory of type float and size _row x _col to image data.

B.1.5 Initializing Gaussian template

Synopsis

```
void initAsGauss()
```

Description

`initAsGauss` initializes the image data into a 3 x 3 Gaussian kernel which can be used for image smoothing. A larger Gaussian kernel can be obtained by correlating the template with itself. The content of kernel is as below :

0.049997	0.122466	0.049997
0.122466	0.299975	0.122466
0.049997	0.122466	0.049997

Table B.1: Gaussian generating kernel

B.1.6 Putting data into image matrix

Synopsis

```
void put(int m, int n, float val)
```

Arguments

`m`	Row coordinate.
`n`	Column coordinate.
`val`	Floating point data

Description

`put` stores the floating point data into location (m,n) of the image matrix. The user should ensure that the row and column coordinates are within the valid range.

B.1.7 Getting data from image matrix

Synopsis

```
float get(int m, int n)
```

Arguments

`m`	Row coordinate.
`n`	Column coordinate.

Description

`get` returns the floating point data at location (m,n) of the image data. The validity of the row and column coordinates are not checked.

B.1.8 Getting IMAGE row and col

Synopsis

```
int getRow()
int getCol()
```

Returns

Row or column size of the image data.

Description

`get` returns the row and column size of the image matrix.

B.1.9 Printing image data

Synopsis

```
void print()
```

Description

`print` displays onto the screen all the values of the image matrix.

B.1.10 Displaying image

Synopsis

```
void show(unsigned char blowup = 1, int num,
          int h_offset = 0, int v_offset)
```

Arguments

`blowup`	Image magnification factor.
`num`	The number of times the length of the window to be increased.
`h_offset`	Horizontal offset of the image against the window origin.
`v_offset`	Vertical offset of the image against the window origin.

Description

`show` displays the image data on an X window. When `h_offset` and `v_offset` are not specified or set to 0, an X window of the image size will be created. Otherwise, no window will be generated and the image will be shown on an existing window. The parameter `num` allows the user to create a larger window so that more than one image can be displayed on the same window. For instance, a second image can be shown next to the original image by calling the method `show` with `num` = 1 and `h_offset` = column of the original image.

B.1.11 Reading image data from file

Synopsis

```
int read(char *filename)
```

Arguments

`filename`	Image file name.

Returns

`NOERROR`	Successful read operation.
`MEMORYERROR`	Memory allocation failure.
`FILEIOERROR`	File I/O error.

Description

By using the appropriate filter, `read` can access the file of type SUN raster (_ras) or binary (_bin). Memory allocation is done automatically. However, reading of incorrect file format will cause memory allocation error. For a SUN raster colour image file, the image will probably need to be conditioned before it can be viewed clearly.

B.1.12 Writing image data to file

Synopsis

```
int write(char *filename,int filetype = _ras)
```

Arguments

`filename`	Image file name.
`filetype`	File type, either _bin or _ras.

Returns

`NOERROR`	Write is successful.
`MEMORYERROR`	Unable to create write buffers in memory.
`FILEIOERROR`	File interface error.

Description

`write` converts the floating point image data to unsigned characters and linearly mapped them to the full range of 0 to 255 before they are written to the file specified. If the file is of SUN raster format, the raster file header, consisting of 8 bytes of integers, will be written as below :

```
ras_magic    = 0x59a66a95    (Magic number of sun raster file)
ras_width    = column size   (Image column)
ras_height   = row size      (Image row)
ras_depth    = 8             (Depth of colour plane)
ras_length   = row * col     (Size in bytes of image)
ras_type     = 1             (Old or new format raster file)
ras_maptype  = 0             (Colour Map type)
ras_maplength= 0             (Colour Map length)
```

B.1.13 Histogram conditioning of image

Synopsis

```
void condition(double low_pct=0.9, double high_pct=0.95,
               double low_val=0.2, double high_val=0.9,
               double imm_pow=1.0)
```

Arguments

`low_pct, high_pct`	Percentage range which the image pixels are to be mapped from.
`low_val, high_val`	Intensity range over which the pixels are to be mapped to.
`imm_pow`	Exponential power used by the transformation function.

Description

`condition` performs image conditioning based on the histogram specification of an image. The transformation function is as following :

$$T_k = \begin{cases} \frac{C_k low\_val}{low\_pct} & ; C_k < low\_val \\ (high\_val - low\_val)(\frac{C_k - low\_pct}{high\_pct - low\_pct})^{imm\_pow} + low\_val & ; \\ low\_val < C_k < high\_val & \\ (1 - high\_val)(\frac{C_k - high\_pct}{1 - high\_pct})^{imm\_pow} + high\_val & ;\text{Otherwise} \end{cases} \quad \text{(B.1)}$$

where C_k is the culmulative distribution function of histogram. Notice that histogram equalization can be achieved by setting `low_pct` $= 0$, `high_pct` $= 1$, `low_val` $= 0$, `high_val` $= 1$ and `imm_pow` $= 1$.

B.1.14 Image correlation

Synopsis

```
IMAGE *correlate (IMAGE *InputImg, int RowStep, int ColStep,
                  char verbose=0)
```

Arguments

`*InputImg`	Pointer to the correlating image template.
`RowStep, ColStep`	Pixel spacing where the correlating template will be shifted each step.

Returns

Pointer to a new correlated image. Returns NULL if memory allocation failed.

Description

`correlate` performs pixel-to-pixel correlation by calculating the inner product between the image template and data. The template will be shifted horizontally and vertically by `ColStep` and `Rowstep` each time.

B.1.15 X window image generation

Synopsis

```
void generateX(unsigned char blowup=1)
```

Arguments

`blowup`	Image magnification factor.

Description

`generateX` creates an X image without displaying it on the screen.

B.1.16 Cutting an image segment

Synopsis

```
IMAGE *IMAGE::cut(int sx, int sy, int height, int length)
```

Arguments

`sx, sy`	Top left corner of the new copied image.
`height`	Height of the new copied image.
`length`	Length of the new copied image.

Returns

Pointer to a new copied image. Returns NULL if memory allocation fails.

Description

`cut` allows an image segment to be cut out from the image data. If the dimensions given are larger than the original image, `cut` will select the maximun possible dimension which originated at (`sx, sy`).

B.1.17 Copying an image

Synopsis

```
IMAGE *IMAGE::copy()
```

Returns

Pointer to a new copied `IMAGE` object. Returns NULL if memory allocation fails.

Description

`copy` duplicates the `IMAGE` object.

B.1.18 Filling of an area of image

Synopsis

```
int fill(float val, int sx=0, int sy=0, int length=0, int height=0)
```

Arguments

`val`	Pixel value.
`sx, sy`	Top left corner of the filled image segment.
`height`	Height of the filled image segment.
`length`	Length of the filled image segment.

Returns

`NOERROR`	Segment filled successfully.
`PARAMERROR`	Illegal parameter values used.

Description

`fill` initializes an image area to the given pixel value.

B.1.19 Example : Reading, writing and correlation of images

This program performs correlation between the input image and template. The correlated image will be displayed on an X window and written to an output file.

```
void testmain( char *imgfile,   /* input image file */
               char *tmpfile,   /* image template */
               char *outfile,   /* output image file */
               int mag )        /* magnification factor */
{

        IMAGE test1,test2;
        IMAGE *test3;

        /* Read test images */
        if (test1.read(imgfile))  exit(-1);

        printf("Showing test image : %s",imgfile);
        test1.show(mag);

        /* If user did not specify template image, use a gaussian
template */
        if (tmpfile) {
           if( test2.read(tmpfile) )  exit( -1 );
        }
        else
           test2.initAsGauss();

        test2.show(mag);

        printf("\nCorrelating image..");
        test3=test1.correlate(&test2,2,2,1);
        /* Show all three images horizontally */
        /* Allocate window space for three images */
        test1.show(mag,3,0,0);

        /* Show images with offsets */
        test2.show(mag,1,test1.getCol());
        test3->show(mag,1,test2.getCol()+test1.getCol());
        printf("\nPress enter to continue ");
        getchar();

        /* write result to file */
        if (outfile)
           test3->write(outfile) ;
        else
           test3->write("out.bin");

        printf("End of test.");
        xwin_close();
}
```

First, an input image will be read. If there is no template image, `initAsGauss` will create a 3 by 3 Gausian kernal. The resulting image, `test3`, is returned by `correlate` and displayed together with the input image and template by `show`.

B.1.20 Example : Histogram specification of an image

```
void testmain( char *imgfile,        /* input image file */
               int blowup,           /* magnification factor */
               float lp , float hp,  /* percentage range */
               float lv, float hv,   /* intensity range */
               float ep  )           /* exponential power */
{
        IMAGE myimage;

        if  ( !(myimage.read(imgfile)) ) {

           printf("Displaying image .... \n\n");
           printf("Histogram Specification parameters :\n");
           printf("\n    Low Percent:%f     High Percent:%f",lp,hp);
           printf("\n    Low Value  :%f     High Value  :%f",lv,hv);
           printf("\n    Exponent   :%f     Magnification:%d",ep,blowup);

           myimage.show(blowup,2);
           myimage.condition(lp,hp,lv,hv,ep);
           printf("Displaying conditioned image ... Press enter to continue\n");
```

```
            myimage.show(blowup,1,myimage.getCol()*blowup);
            getchar();
            xwin_close();
        }
}
```

Histogram specification is done by calling `condition`. With appropriate input parameters, it can either perform noise reduction or image enhancement. In this program, the conditioned and input image will be displayed side by side on an X window.

B.1.21 Example : Cutting and copying of image segments

```
void testmain( char *imgfile,   /* input image file */
               char *outfile,   /* output image file */
               int mag )        /* magnification factor */
{
        IMAGE test1;
        IMAGE *test2,*test3;

        int sx, sy, length, height;

        /* Read image file */
        if  (test1.read(imgfile) ) exit(-1);

        printf("Showing test1 : %s",imgfile);
        test1.show(mag);

        /* Using X windows to select an image segment */
        xwin_SelRegion( &sx,&sy,&length,&height);
        test2 = test1.cut(sx/mag ,sy/mag ,length/mag, height/mag);

        /* Copy image*/
        test3 = test2->copy();

        /* Show images in line */
        printf("\nShowing cut image and copied image ");
        test2->show(mag,2);
        test3->show(mag,1,test2->getCol());
        printf("\nPress enter to continue...");
        getchar();

        if (outfile) {
                printf("\nWriting to file %s",outfile);
                test3->write(outfile);
        }

        printf("End of test..");
        xwin_close();
}
```

This program uses X window interface, `xwin_SelRegion`, to select an image segment and then cut it. This segment is finally copied to another image segment.

B.2 EDGE : Edge Gradient

`EDGE` is a class for operating image intensity to produce an edge map marking the location, strength and direction of edge points. The intensity gradient vectors are computed by fitting planes in 2x2 windows using the least squares method. For example, for a 2x2 image window as shown in Table 1.2, we have :

f_{11}	f_{12}
f_{21}	f_{22}

Table B.2: 2 x 2 image window.

$$d_y = \frac{1}{2}\left\{(f_{21} + f_{22}) - (f_{11} + f_{12})\right\} \tag{B.2}$$

$$d_x = \frac{1}{2}\left\{(f_{12} + f_{22}) - (f_{11} + f_{21})\right\} \tag{B.3}$$

Thus the magnitude of intensity gradient at location f_{11} is given by $({d_x}^2 + {d_y}^2)^{\frac{1}{2}}$, while the angle is given by $tan^{-1}\frac{dy}{dx}$.

`EDGE` has the following defination :

```
CLASS EDGE {
      protected :
          IMAGE *Mag;        /* magnitude */
          IMAGE *Ang;        /* angle */
}
```

`EDGE` object stores magnitude and direction of the edge gradient in the form of two images. This make them easy to experiment with various functions provided by `IMAGE` class.

B.2.1 EDGE constructor

Synopsis

```
EDGE()
```

Description

The constructor initializes the edge magnitude and angle image pointers to NULL.

B.2.2 EDGE destructor

Synopsis

```
~EDGE()
```

Description

The destructor frees memory allocated to the edge magnitude and angle image pointers.

B.2.3 Resetting an `EDGE` object

Synopsis

```
void reset()
```

Description

`reset` allows the reuse of `EDGE` object. It frees memory allocated and set the edge magnitude and angle image pointers to NULL.

B.2.4 Retrieving magnitude and angle of an edge point

Synopsis

```
float getMag(int m, int n)
float getAng(int m, int n)
```

Arguments

`m, n`	Row and column coordinates of interest.

Returns

Edge magnitude or angle at location (m, n) of the edge matrix.

Description

These methods allow accessing of the edge data. Attempts to get data from invalid coordinate will result in garbage data.

B.2.5 Storing magnitude and angle of an edge point

Synopsis

```
void putMag(int m, int n, float val)
void putAng(int m, int n, float val)
```

Arguments

`m, n`	Row and Column coordinates of interest.
`val`	Data to be stored in location (m, n).

Description

These methods store data in location (m, n) of edge magnitude or angle.

B.2.6 Displaying edge map

Synopsis

```
void show (unsigned char magnify = 1, int h_offset=0,
                                       int v_offset=0)
```

Arguments

`magnify`	Magnification factor of the displayed image.
`h_offset`	Horizontal offset of the image against the window origin.
`v_offset`	Vertical offset of the image against the window origin.

Description

`show` generates and displays an X image for both the edge magnitude and angle images. `show` will raise an X window for the display if the horizontal and vertical offsets are both 0. Otherwise, it will draw the specified X image without generating an X window. In this case, he user will have to ensure that an X window is up before displaying the image.

B.2.7 Calculating the edge map

Synopsis

```
int compute(IMAGE *img, int verbose=1, double low_pct, double
            high_pct, double low_val, double high_val, double
            imm_pow)
```

Arguments

`*img`	Input image in which the edge is to be computed.
`verbose`	Verbose Flag. On = 1 and off = 0.
`low_pct, high_pct`	Percentage range which the image pixels are to be mapped from.
`low_val, high_val`	Intensity range over which the pixels are to be mapped to.
`imm_pow`	Exponential power of the mapping function.

Returns

`NOERROR`	No error.
`MEMORYERROR`	Memory allocation error.

Description

`compute` generates the edge gradient magnitude and angle images by least square estimation. To increase robustness, we perform image conditioning for edge magnitude. The parameters ($high\_pct = 0.95$, $low\_pct = 0.9$, $high\_val = 0.9$, $low\_val = 0.2$) were found empirically to produce good conditioning image.

B.2.8 Getting row and column of `EDGE`

Synopsis

```
int getRow(int MagData=1)
int getCol(int MagData=1)
```

Arguments

`MagData`	Indicates the choice of either magnitude (1) or angle (0) information.

Returns

Row and column size of the magnitude or angle images.

Description

These methods provide a means of getting the row and column information.

B.2.9 Example : Edge computation

```
void testmain( char *imgfile,   /* image file */
               int mag )        /* magnification factor */
{

   IMAGE myimage;
   EDGE myedge;

   if (imgfile) myimage.read(imgfile);
   else {
         /* Create a sample image of 3 rows of different intensities */

         int i,j,row,col;

         row = 90;
         col = 90;

        printf("\nCreating demo image...\n");

        myimage.init(row,col);

        for(i=0;i<row;i++)
                for(j=0;j<col;j++)
```

```
                        myimage.put(i,j,20);

        for(i=20;i<70;i++)
                for(j=20;j<70;j++)
                        myimage.put(i,j,70);

        for(i=30;i<60;i++)

                for(j=30;j<60;j++)
                        myimage.put(i,j,150);
   }

   myimage.show(mag);
   printf("\nShowing image. Press enter to see edge .");
   getchar();

   /* Computing and showing edge */
   /* Default parameters are used. Verbose mode */
   myedge.compute(&myimage, 1);
   myedge.show(mag);
   printf("\nPress enter to continue.");
   getchar();
   printf("End of test.\n");
}
```

This program creates `IMAGE` and `EDGE` classes. If no image file is specified, a demo image will be generated. By calling `compute` and `show`, the edge gradient is calculated from the input or sampple image and then displayed on an X window.

B.3 PYRAMID : Pyramid of Images

`PYRAMID` is built up of a series of edge map and Gaussian images, which are generated from the raw image, in the form of EDGE and IMAGE objects. Construction of Gaussian pyramid images [7] is done by image correlation with Gaussian generating kernel. In this case, the higher level correlates image is a reduced version of the lower level image in that both resolution and sample density are decreased. For instance, a Gaussian image at level l, g_l, can be expressed as :

$$g_l(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} w(m,n) g_{l-1}(2i+m, 2j+n) \tag{B.4}$$

where w(m, n) is a 3 x 3 Gaussian generating kernel. An `PYRAMID` object are defined as below :

```
class PYRAMID {

   protected :
        short numLevel ; /* number of level */
        EDGE **edgemap ; /* edge map at each level */
        IMAGE **gaussImg ; /* gaussian smoothed image */

      public :
        IMAGE *rawImg; /* raw image */

};
```

B.3.1 PYRAMID Constructor

Synopsis

`PYRAMID()`

Description

The constructor initializes the edge map, Gaussian pyramid and raw images to NULL, as well as sets the pyramid to level 0.

B.3.2 PYRAMID Destructor

Synopsis

`~PYRAMID()`

Description

The destructor frees the memory allocated to edge map, Gaussian pyramid and raw images.

B.3.3 Resetting PYRAMID object

Synopsis

`reset()`

`reset` allows the reuse of an `PYRAMID` object. It frees the memory allocated to the edge map and Gaussian images for each level of the pyramid.

B.3.4 Generating pyramid images

Synopsis

```
generate(short level, int verbose=0,double low_pct=0.9,
         double high_pct=0.95, double low_val=0.2,
         double high_val=0.9, double imm_pow=1.0,
         EextTYPE = _EDGE)
```

Arguments

`level`	The number of Gaussian pyramid levels to be generated.
`verbose`	An option to enable verbose operation mode.
`low_pct, high_pct`	Percentage range which the image pixels are to be mapped from.
`low_val, high_val`	Intensity range over which the pixels are to be mapped to.
`imm_pow`	Exponential power for the mapping function.
`EextTYPE`	Type of external energy, includes intensity (_INTENSITY), edge magnitude only (_EDGEMAG) and edge map (_EDGE) image energy.

Returns

`NOERROR`	Successful pyramid generation.
`MEMORYERROR`	Memory allocation failure.

Description

`generate` creates a pyramid of Gaussian images by correlating the raw image with a Gaussian kernel, and computes edge gradient based on the `EextTYPE` specified. Edge direction map will not be generated if `_EDGE` is not used. To account for robustness, we use image conditioning, `condition`, for the edge magnitude. The parameters ($high\_pct = 0.95$, $low\_pct = 0.9$, $high\_val = 0.9$, $low\_val = 0.2$) were found empirically to produce a good conditioning image.

B.3.5 Accessing edge map

Synopsis

```
EDGE *getEdge(short level_id)
```

Arguments

`level_id`	Pyramid level of interest.

Returns

Pointer to an edge image of level specified. Returns `NULL` if the level specified is invalid.

Description

`getEdge` allows accessing of an edge image of level specified, in which the level must be within the range of 0 to the highest possible pyramid level availble.

B.3.6 Accessing Gaussian image

Synopsis

```
IMAGE *getGauss(short level_id)
```

Arguments

`level_id`	Pyramid level of interest.

Returns

Pointer to a Gaussian image of level specified. Returns `NULL` if the level specified is invalid.

Description

`getGauss` allows accessing of a Gaussian image of level specified, in which the level must be within the range of 0 to the highest possible pyramid level availble.

B.3.7 Getting pyramid level

Synopsis

```
short getLevel(void)
```

Returns

Number of levels of Gaussian pyramid. The highest level is given as level-1, since the lowest level is 0.

B.3.8 Print Gaussian pyramid data

Synopsis

```
void print(int level)
```

Arguments

`level`	Pyramid level of interest.

Description

`print` displays onto the screen all the data of an edge image of level specified.

B.3.9 Displaying the pyramid of images

Synopsis

```
void show(unsigned magnify,int level=1)
```

Arguments

`magnify`	Image magnification factor.
`level`	Levels of the Gaussian pyramid required for display.

Description

`show` displays pyramid of images, include edge and Gaussian images, from the lowest level to the level specified.

B.3.10 Duplicating a pyramid

Synopsis

```
duplicate(PYRAMID *pyramid)
```

Arguments

`*pyramid`	Pointer to a source pyramid.

Returns

`NOERROR`	Successful duplication.
`MEMORYERROR`	Memory allocation failure.

Description

`duplicate` copies the source pyramid into `PYRAMID` itself.

B.3.11 Putting root image into PYRAMID

Synopsis

```
int putRawImg(char *filename)
void putRawImg( IMAGE *img )
```

Arguments

`*filename`	Souce image filename.
`*img`	Pointer to a source `IMAGE object`.

Returns

`NOERROR`	Successful read operation.
`MEMORYERROR`	Memory allocation failure.
`FILEIOERROR`	Unable to read from file.

Description

`putRawImg` reads or copies the source image as its raw image. If a raw image has already existed, `putRawImage` will destroy the current one and copies the source image as the new raw image.

B.3.12 Example:Building of pyramid from image

This program generates pyramid of images from an image file and shows them on an X window.

```
void testmain( char *imgfile,   /* image file */
               int level,       /* Gaussian pyramid level */
               int mag )         /* magnification factor */
```

```
{
        PYRAMID mypyramid;

        if ( mypyramid.putRawImg( imgfile ) ) {

                printf("Unable to read image file.");
                exit(-1);
        }

        else {
                /* generating pyramid to default conditioning */
                /* parameters in verbose mode */
                mypyramid.generate( level, 1 );
                mypyramid.show( mag, level );
                getchar();
                xwin_close();
        }
}
```

`generate` uses default histogram conditioning paramters and verbose 'on' mode to compute and generate edge map and Gaussian images at each level of pyramid, while `show` displays them in a hierarchical manner.

B.3.13 Example:Accessing a particular level image

```
void testmain( char *imgfile,   /* image file */
               int pymlevel,    /* pyramid levels */
               int i_level,     /* level of interest */
               int mag)         /* maginification factor */
{
        IMAGE myimage1,
              *myimage2;        /* IMAGE object */
        PYRAMID mypyramid;      /* PYRAMID object */
        EDGE *myedge;           /* EDGE object */

        if ( myimage1.read( imgfile ) ) {
                printf("\nUnable to get image.\n");
                exit(-1);
        }

        /* Placing image into pyramid object by using another
           image */
        mypyramid.putRawImg( &myimage1 );
        /* generating pyramid by default conditioning
           parameters */
        mypyramid.generate( pymlevel, 1 );

        if ( !i_level )
                i_level = mypyramid.getLevel() - 1;

        /* get edge map and Guassian images of interest */
        myimage2 = mypyramid.getGauss( i_level );
        myedge = mypyramid.getEdge( i_level );

        /* Showing gaussian image at level */
        if ( myimage2 && myedge) {

                printf("\n Showing level %d Gaussian image",
                              i_level);
                myimage2->show( mag*i_level );
                printf("\nPress enter to continue .");
                getchar();

                printf("\nShowing level %d edge map", i_level);
                myedge->show( mag*i_level );
                printf("\nPress enter to continue.");
                getchar();
        }
        else
                printf("\nLevel specified is invalid.");

        printf("\nEnd of test\n");
}
```

This example demonstrates how to access and display an edge map and Gaussian images of level of interest by `getGauss` and `getEdge`.

B.4 MATRIX : Simple matrix calculation

`Matrix` class provides advanced matrix manipulation routines which include addition, substraction, multiplication, tranpose and inversion for an IMAGE object. `Matrix` is inherited with extra methods from `IMAGE` class.

```
class MATRIX : public IMAGE {
        /* no extra variable */
};
```

B.4.1 Print matrix data

Synopsis

```
void MATRIX::dump(char *header = NULL)
```

Arguments

`header`	Title of matrix

Description

`print` display on the screen all the value of matrix data, as well as the title of matrix if it is specified.

B.4.2 Initializing matrix

Synopsis

```
MATRIX::init(short _row, short _col, char identity = 0)
```

Arguments

`_row, _col`	The size of matrix, _row x _col.
`identity`	A flag (1:on and 0:off) to indicate whether to create an identity matrix.

Returns

`NOERROR`	Successful memory allocation.
`MEMORYERROR`	Otherwise.

Description

`init` allocates memory of type float and size _row x _col to a matrix. If `identity` is on, an identity matrix will be created.

B.4.3 Matrix transpose

Synopsis

```
MATRIX * MATRIX::transpose(void)
```

Returns

Pointer to a transposed matrix. Returns NULL if memory allocation fails.

Description

`transpose` creates a new matrix of size col x row and transposes the image data from row to column and visa versa.

B.4.4 Matrix addition

Synopsis

```
MATRIX * MATRIX::operator+ (MATRIX& mtx)
```

Arguments

`mtx`	Matrix for addition.

Returns

Pointer to an added matrix. Returns NULL if memory allocation fails or the size of `mtx` is different from the size of original matrix.

Description

`+` performs addtion between the `mtx` and the original matrix.

B.4.5 Matrix substraction

Synopsis

```
MATRIX * MATRIX::operator- (MATRIX& mtx)
```

Arguments

`mtx`	Matrix for substraction.

Returns

Pointer to a substracted matrix. Returns NULL if memory allocation fails or the size of `mtx` is different from the size of original matrix.

Description

`-` substracts the original matrix from `mtx`.

B.4.6 Matrix multiplication

Synopsis

```
MATRIX * MATRIX::operator* (MATRIX& mtx)
```

Arguments

`mtx`	Matrix for multiplication.

Returns

Pointer to a multipled matrix. Returns NULL if memory allocation fails or the row of `mtx` is different from the column of original matrix.

Description

`*` performs multiplication between the `mtx` and the original matrix.

B.4.7 Swapping rows of matrix

Synopsis

```
MATRIX::swapRow(short i, short j)
```

Arguments

`i, j`	Rows to be swapped.

Returns

`NOERROR`	Successful operation.
`PARAMERROR`	Invalid row or out of range.

Description

`swapRow` swap the matrix data of row i and j.

B.4.8 Matrix inversion

Synopsis

```
MATRIX * MATRIX::inverse(void)
```

Returns

Pointer to a inversed matrix. Returns NULL if memory allocation fails or the original matrix data is of a singular matrix or not a square matrix.

Description

`inverse` applies Gaussian elimination algorithm to perform matrix inversion.

B.5 SNAXEL : The contour model unit

`SNAXEL` is a class for calculating tangent and normal vectors, internal, external and total energies of a snaxel. A series of SNAXEL objects can form a complete contour. It is defined as below :

```
class SNAXEL {

    protected :
        double row, col ;        /* snaxel coordinate */
        double alpha, beta ;     /* shape coefficients */
        double lambda ;          /* local regularization
                                    parameters */
        double Eint, Eext ;      /* internal/external energy */
        double Esnaxel;          /* snaxel energy */
        SNAXEL *next;            /* next snaxel */
        SNAXEL *prev;            /* previous snaxel */
};
```

Each SNAXEL object is internally connected with its two neighbouring snaxels so as to form a complete chain of contour. Besides, it consists of shape coefficients and local regularization parameter for the ease of energy calculation.

B.5.1 Computing mean position

Synopsis

```
void meanPosition( SNAKEMODE mode, double cg_row,
                   double cg_col, SNAXEL *head, SNAXEL *tail,
                   double *meanrow, double *meancol )
```

Arguments

`mode`	Snake mode of a contour - open or closed.
`cg_row, cg_col`	Coordinate of the centre of gravity of a contour.
`avglen`	Average length of snaxels.
`head`	Pointer to the head of a contour.
`tail`	Pointer to the tail of a contour.
`meanrow, meancol`	Coordinate of the mean position.

Description

`meanposition` calculates the mean position of snaxels and stores the coordinates in the locations pointed to by `meanrow` and `meancol`. The mean position can be referred to as the position of zero internal energy and calulated as following :

$$\alpha_i u_{i_\alpha} + \beta_i u_{i_\beta} \tag{B.5}$$

where u_{i_α} and u_{i_β} are the neighbouring snaxels in the contour-centered coordinate formed. α_i and β_i are the shape coefficients.

B.5.2 Computing tangent vector

Synopsis

```
void    getDirectionVec( SNAKEMODE mode,
                         SNAXEL *head, SNAXEL *tail,
                         double *Uv0_x, double *Uv0_y,
                         double *Uv1_x, double *Uv1_y)
```

Arguments

`mode`	Snake mode of a contour - open or closed.
`head`	Pointer to the head of a contour.
`tail`	Pointer to the tail of a contour.
`Uv0_x, Uv0_y`	Unit vector of neighbouring snaxel (alpha).
`Uv1_x, Uv1_y`	Unit vector of neighbouring snaxel (beta).

Description

`getDirectionVec` calculates two unit vectors $uv0 = (uv0\_x, uv0\_y)$ and $uv1 = (uv1\_x, uv1\_y)$, which can be used to compute the tangent vector. The tangent vector, $\mathbf{t}_i$, is given as follows :

$$\mathbf{t}_i = \frac{u_i - u_{i-1}}{\|u_i - u_{i-1}\|} + \frac{u_{i+1} - u_i}{\|u_{i+1} - u_i\|} \tag{B.6}$$

where u is the snaxel in contour-centered coordinate formed, $\frac{u_i - u_{i-1}}{\|u_i - u_{i-1}\|}$ is $uv0$ and $\frac{u_{i+1} - u_i}{\|u_{i+1} - u_i\|}$ is $uv1$.

B.5.3 Computing normal vector

Synopsis

```
void    getNormalVec( SNAKEMODE mode,
                      SNAXEL *head, SNAXEL *tail,
                      double *nv_x, double *nv_y )
```

Arguments

`mode`	Snake mode of a contour - open or closed.
`head`	Pointer to the head of a contour.
`tail`	Pointer to the tail of a contour.
`nv_x, nv_y`	Coordinate of normal vector.

Description

`getNormalVec` calculates the unit vector which is normal to the tangent vector of snaxel and stores the result in the locations pointed to by `nv_x` and `nv_y`.

B.5.4 Calculating the `SNAXEL` angle

Synopsis

```
double getNormalAng( SNAKEMODE mode, SNAXEL *head, SNAXEL *tail)
```

Arguments

`mode`	Snake mode of a contour - open or closed.
`head`	Pointer to the head of a contour.
`tail`	Pointer to the tail of a contour.

Returns

Angle of snaxel in radians.

Description

`getNormalAng` computes the angle which is normal to the tangent vector of a snaxel.

B.5.5 Showing snaxel

Synopsis

```
void show( IMAGE *img, short Old_Row, short Old_Col,
           int pt_Xoffset = 0, int pt_Yoffset = 0 )
```

Arguments

`*img`	Background image on which the SNAXEL is to be shown.
`Old_Col, Old_Row`	Coordinate of a snaxel before deformation.
`pt_Xoffset, pt_Yoffset`	Offset position of a snaxel.

Description

`show` displays a snaxel on top of the background image. This is done by resetting the dot at [`Old_col, Old_Row`] to the background image data, and then placing a dot at the current snaxel position. The offset position need to be specified if the background image occupies only certain portion of an X window. The user should ensure that the background image have been displayed before calling `show`.

B.5.6 Interface row and column information

Synopsis

```
double getRow(void)
double getCol(void)
void putRow(double _row)
void putCol(double _col)
```

Returns

Snaxel coordinate.

Description

These methods facilitate the retrieval and accessing of snaxel co-ordinate.

B.5.7 Interface snaxel energy information

Synopsis

```
double  getEint(void)
double  getEext(void)
double  getEsnaxel(void)
void  putEsnaxel( double _Esnaxel )
void  putEint( double _Eint )
void  putEext( double _Eext )
```

Returns

Internal, external or total snaxel energy.

Description

These methods facilitate the retrieval and accessing of snaxel energy information.

B.5.8 Interface parameter information

Synopsis

```
double  getAlpha(void)
double  getBeta(void)
double  getLambda(void)
void  putAlpha( double _alpha )
void  putBeta( double _beta )
void  putLambda( double _lambda )
```

Returns

Alpha (α), beta (β) or lambda (λ).

Description

These methods facilitate the retrieval and accessing of snaxel parameters.

B.5.9 Interface pointers to neighbouring snaxels

Synopsis

```
SNAXEL *getNext(void)
SNAXEL *getPrev(void)
SNAXEL *getNext(SNAKEMODE mode, SNAXEL *head)
SNAXEL *getPrev(SNAKEMODE mode, SNAXEL *tail)
void  putPrev( SNAXEL *_prev )
void  putNext( SNAXEL *_next )
```

Arguments

`mode`	Snake mode of a contour - opened or closed.	
`head`	Pointer to the head of a contour.	
`tail`	Pointer to the tail of a contour.	1—1
`_prev`	Pointer to the previous snaxel.	
`_next`	Pointer to the next snaxel.	

Returns

Pointer to previous or next snaxel.

Description

These methods facilitate the retrieval and accessing of the neighbouring snaxels. If snake mode is specified, the following will be done :

1. For an open snake
   - `getPrev` retrieves or accesses the previous snaxel if it is not the head of a contour. Otherwise, the third snaxel will be retrieved or accessed.
   - `getNext` retrieves or accesses the next snaxel if it is not the tail of a contour. Otherwise, the snaxel retrieved or accessed is that of two snaxels before.
2. for a closed snake
   - `getPrev` retrieves or accesses the previous snaxel if it is not the head of a contour. Otherwise, the tail of a contour will be retrieved or accessed.
   - `getNext` retrieves or accessed the next snaxel if it is not the tail of a contour. Otherwise, the head of a contour will be retrieve or accessed.

B.5.10 Example : Using mean position to calculate Internal energy

This example demonstrates how the internal energy of a snaxel is calculated. The internal energy measures the deviation of a snaxel from its mean position after deformation.

```
void testmain( char *confile,   /* standard contour file */
               int mag )        /* magnification factor */
{
  SNAXEL *sptr;            /* SNAXEL object */
  CONTOUR mycontour;       /* CONTOUR object */
  double l_inf;            /* infinity norm */
  double meanrow,          /* mean position of snaxel */
         meancol;
  double avglen;           /* average length of snaxel */
  double Eint;             /* Internal energy */
  double cg_row,           /* center of gravity (CG) */
         cg_col;
  register short i;

  /* Read and show the contour */
  if ( mycontour.read( confile ) )        exit( -1 );

  mycontour.display(mag);

  /* Compute the average distance between snaxels and CG*/
  mycontour.computeCG();
  avglen = mycontour.computeAvgLength();
  cg_row = mycontour.getCgRow();
  cg_col = mycontour.getCgCol();
  printf("\nCentre of gravity at row %d col %d\n",
             ROUNDOFF( cg_row ), ROUNDOFF( cg_col ));

  mycontour.imageCentered();

  /* Calculating the internal energy for each snaxel */
  for (i=0, sptr=mycontour.getHead(); sptr;
                          sptr=sptr->getNext(),i++)
  {
     if ( fabs(sptr->getAlpha() ) < VERY_SMALL &&
        fabs(sptr->getBeta() )  < VERY_SMALL )
        Eint = 1.0;

     /* compute shape coeff. (alpha & beta) with
     infinity norm */
     sptr->meanPosition( mycontour.getMode(), cg_row,
cg_col, mycontour.getHead(), mycontour.getTail(),
&meanrow, &meancol);
     l_inf = MAX( fabs( meanrow - sptr->getRow() ),
     fabs( meancol - sptr->getCol() ) );
     Eint =  SQR( l_inf / avglen );
     printf("\nSnaxel %d energy = %f",i,Eint);
  }

  printf("\nEnd of test.");
```

```
        printf("\nPress enter to continue.");
        getchar();
        xwin_close();
}
```

meanPosition measure the mean position of current snaxel, while infinity norm measures the deviation of a snaxel.

B.5.11 Example : Verifying vector calculations

```
void testmain( char *confile,   /* standard contour file */
               int mag )        /* magnification factor */
{
        SNAXEL *sptr;           /* SNAXEL object */
        CONTOUR mycontour;      /* CONTOUR object */
        double uv_x, uv_y,      /* tangent vector */
               uv0_x, uv0_y,
               uv1_x, uv1_y;
        double nv_x, nv_y;      /* normal vector */
        double Amodel,          /* tangent angle */
               Anorm;           /* normal angle */
        register short i;

        /* Read and show the contour */
        if ( mycontour.read( confile ) ) exit(-1);

        /* Compute the average distance between snaxels and CG*/
        mycontour.computeCG();
        mycontour.imageCentered();

        /* Calculating the internal energy for each snaxel */
        for (i=0, sptr=mycontour.getHead(); sptr;
          sptr=sptr->getNext(), i++) {
                sptr->getDirectionVec(mycontour.getMode(),
                                      mycontour.getHead(),
                                      mycontour.getTail(),
                                      &uv0_x,&uv0_y,&uv1_x,&uv1_y);

                /* tangent vector at current snaxel */
                uv_x = uv0_x + uv1_x;
                uv_y = uv0_y + uv1_y;

                Amodel = ATAN2( uv_x, uv_y );

                sptr->getNormalVec(mycontour.getMode(),
                              mycontour.getHead(),
                              mycontour.getTail(),
                              &nv_x,&nv_y);
                Anorm = ATAN2(nv_x,nv_y);

                /* for verification, angle should be 90 degree
                   apart */
                printf("\nAngle : %3.6f\t Normal : %3.6f",
                          ANGLE(Amodel), ANGLE(Anorm));
        }

        mycontour.display(mag);
        printf("Press enter to continue ");
        getchar();
        printf("\nEnd of test.\n\n");
        xwin_close();
}
```

This program calculates the tangent and normal angle of each snaxel. `getDirectionVec` calculates the tangent vector at each snaxel, while `getNormalVec` rotates the tangent direction by 90 degree so as to compute the normal vector.

B.6 CONTOUR : A deformable template object

`CONTOUR` is a collection of `SNAXEL` objects forming a deformable template. It has the following structure :

```
class CONTOUR {

    protected :
        SNAXEL *head;           /* head of snake */
        SNAXEL *tail;           /* tail of snake */
        SNAKEMODE mode;         /* opened or closed snake */
        short numsnaxel;        /* number of snaxel */
        double cg_row, cg_col;  /* center of gravity */
        double avglen;          /* ave. power (length) of snake */
        double sig_nu_sqr;      /* white noise variance on gradient
                           power */
        double Z;               /* normalizing constant */
        short direction;        /* [0/-1/+1] preset gradient
                                   direction */
};
```

`CONTOUR` is a linked list of `SNAXELS` originated at image origin or the reference point (`cg_col`, `cg_row`). Each contour can have a gradient direction (`direction`) that pointing outward (+1) or inward (-1). `avglen` refers to $l(U)$ in equation (3.9), `sig_nu_sqr` refers to σ_η^2 in equation (3.24), and `Z` refers to Z_i in equation (4.35) of [1].

B.6.1 `CONTOUR` constructor

Synopsis

```
CONTOUR(void)
```

Description

The constructor initializes the number of snaxels, white noise variance (σ_η^2) and contour direction to 0, sets normalizing constant to 1. and directs the head and tail to NULL.

B.6.2 `CONTOUR` destructor

Synopsis

```
~CONTOUR(void)
```

Description

The destructor frees the memory allocated to snaxels which form a complete chain of contour.

B.6.3 Resetting a `CONTOUR`

Synopsis

```
void reset()
```

Description

`reset` allows the reuse of an `CONTOUR` object by freeing up the memory allocated to snaxels and setting the head and tail to NULL.

B.6.4 Automatic initialization of closed contour

Synopsis

```
int init( short _cg_row, short _cg_col,
          double radius, short numpts = 16)
```

Arguments

`_cg_row, _cg_col`	Centre of circle.
`radius`	Radius of circle.
`numpts`	Number of snaxels.

Returns

`NOERROR`	Successful operation.
`MEMORYERROR`	Memory allocation failure.

Description

`init` creates a circle with `numpts` snaxels and sets its `mode` to _CLOSED, that is, a contour with its head and tail internally connected.

B.6.5 Automatic initialization of open contour

Synopsis

```
int init( short sx, short sy, short ex, short ey,
          short numpts = 16 )
```

Arguments

`sx, sy`	The starting point of a line.
`ex, ey`	The ending point of a line.
`numpts`	Number of snaxels.

Returns

`NOERROR`	Successful operation.
`MEMORYERROR`	Memory allocation failure.

Description

`init` creates a line with `numpts` snaxels and sets its `mode` to _OPENED, that is, a contour with unconnected head and tail.

B.6.6 Manual initialization of a contour

Synopsis

```
int init( IMAGE *ximg, unsigned char blowup = 1,
          INITMODE  theINIT = _CLICKMOUSE,
          SNAKEMODE _mode = _CLOSED,int spacing)
```

Arguments

`*ximg`	Reference image.
`blowup`	Image magnification factor.
`theINIT`	Contour initialization method, _DRAGMOUSE or _CLICKMOUSE
`_mode`	Snake mode of a contour, _OPENED or _CLOSED.

Returns

`NOERROR`	Successful operation.
`MEMORYERROR`	Memory allocation error.

Description

`init` creates a contour of either through clicking (`_CLICKMOUSE`) or dragging (`_DRAGMOUSE`) by mouse. By displaying the reference image on an X window, the user then specifies the desired snaxels. Since the snaxels are joined sequentially, the selection of the snaxels must be in order.

B.6.7 File interface methods

Synopsis

```
int read( char *filename )
int write( char *filename )
```

Arguments

`*filename`	Contour file.

Returns

`NOERROR`	Successful read or write operation.
`MEMORYERROR`	Memory allocation failure.
`FILEIOERROR`	Unable to read from or write to file.

Description

`read` and `write` provide file manipulation routines for accessing a contour file. The format, which is in text form, is as following :

Field	*Datatype*
Magic number	1010 (hex)
Mode	_OPENED (0) or _CLOSED (1)
Number of snaxels	Integer number
White noise variance	Exponential notation
Snaxel coordinates (column, row)	Double
Shape matrix (α, β)	Double
Regularization parameter (λ)	Double
Normalizing constant	Exponential notation.

B.6.8 Learning shape matrix

Synopsis

```
void computeShape()
```

Description

`computeShape` models and calculates the *shape matrix* of a contour as in (Eqn 3.3 of [1]). α and β are computed as following :

$$\begin{bmatrix} \alpha_\mathbf{i} \\ \beta_\mathbf{i} \end{bmatrix} = \begin{bmatrix} x_{i\alpha} & x_{i\beta} \\ y_{i\beta} & y_{i\beta} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{x_i} \\ \mathbf{y_i} \end{bmatrix} \tag{B.7}$$

B.6.9 Computing normalizing constant for classification purpose

Synopsis

```
void computeZ()
```

Description

`computeZ` compute normaling constant Z_i by Monte Carlo estimation. Z_i must be calculated during the training stage of classification. Supposes a family of contours $U \in \Omega_i$, then

$$Z_i = \sum_{U \in \Omega_i} \exp(-E_{int}(U)) \tag{B.8}$$

B.6.10 Computing the average vector distance l(U) between snaxels

Synopsis

```
double computeAvgLength()
```

Returns

Average distance of snaxels.

Description

`computeAvgLength` calculates the average distance between snaxels, $l(U)$, as following :

$$l(U) = \frac{1}{n} \sum_{i=1}^{n} \|u_{i+1} - u_i\|^2 \tag{B.9}$$

$l(U)$ is also the normalising constant used for internal energy calculation.

B.6.11 Regenerating shape matrix

Synopsis

```
int regenerate()
```

Returns

`NOERROR`	Successful operation.
`MEMORYERROR`	Memory allocation failure.
`PARAMERROR`	Parameter error.

Description

`regenerate` generate the complete chain U from *shape matrix* A by matrix inversion if the last two snaxels on U is available. This is possible by decomposing A into an $(n-2) \times (n-2)$ invertible submatrix A_r such that,

$$A_r U_r + b = 0 \tag{B.10}$$

where

$$b = \begin{bmatrix} \dots \\ 0 \\ \dots \\ -\beta_{n-2} u_{n-1} \\ u_{n-1} - \beta_n u_n \end{bmatrix} \tag{B.11}$$

Thus the regenerative contour U_r is,

$$U_r = -A_r^{-1} b \tag{B.12}$$

B.6.12 Conversion of Coordinates

Synopsis

```
void imageCentered()
void contourCentered()
```

Description

`imageCentered` defines a contour as the vector containing an ordered set of points, $V = [v_1, v_2, \dots, v_n]$, where each v_i has its origin at (0, 0) of an image. In contrast, `contourCentered` defines a contour as $U = [u_1, u_2, \dots, u_n]$, where each $u_i = v_i - g$ represents the displacement from the center of gravity g of a contour.

B.6.13 Calculating the internal energy of an individual snaxel

Synopsis

```
double EInternal (SNAXEL *sxptr)
```

Arguments

sxptr	Targeted snaxel.

Returns

Internal energy of the targeted snaxel.

Description

`EInternal` computes the internal energy of a snaxel as (Eqn 3.21 of [1]).

B.6.14 Affine transformations of contour

Synopsis

```
void rotate (double angle)
void translate (int dx, int dy)
void scale (double sx, double sy)
void dilate (double idx, double idy)
void affineTransform(double sx, double sy, double rt,
                double tx, double ty, double idx, double idy)
```

Arguments

`angle`	Rotation angle in degrees.
`dx, dy`	Translation vector.
`sx, sy`	Scaling factor.
`idx, idy`	Dilation factor.

Description

These methods provides routines for global deformations of a contour so that user can observe the effects of rigid motion on the *shape matrix*.

B.6.15 Computing the centre of gravity of contour

Synopsis

```
void computeCG()
```

Description

`computeCG` calculates the centre of gravity of a contour, $CG(U)$, as below :

$$CG(U) = \frac{1}{n}\sum_{i=1}^{n} v_i \tag{B.13}$$

B.6.16 Duplicating a contour

Synopsis

```
CONTOUR *duplicate(CONTOUR *target=NULL, short snx=0)
```

Arguments

target	Targeted contour.
snx	Flag (0:off, 1:on) to indicate copying of snaxels only.

Returns

Pointer to a new created contour if `target` is NULL or pointer to the `target` contour if otherwise. Returns NULL if memory allocation error

Description

`duplicate` creates an exact duplication of its contour if `target` is NULL. Otherwise, it will copy its content or snaxels information only to the `target` contour, depending on the `snx` flag. Copying of snaxels will not be allowed if both contours have different number of snaxels.

B.6.17 Retrieving and accessing of a contour content

Synopsis

```
SNAXEL *getHead(void)
SNAXEL *getTail(void)
SNAKEMODE getMode(void)
void putMode(SNAKEMODE _mode)
short getNumSnaxel(void)
double getCgRow(void)
double getCgCol(void)
double getZ(void)
```

```
void putCgRow(double _row)
void putCgCol(double _col)
double getSigNuSqr(void)
void putSigNuSqr(double _sig_nu_sqr)
void putDirection(short _direction)
void putZ(double _Z)
```

Arguments

`_mode`	Snake mode of a contour.
`_row, _col`	Center of gravity.
`_sig_nu_sqr`	White noise variance.
`_direction`	Snake direction.

Returns

Head or tail of contour, snake mode, center of gravity, white noise variance (σ_η^2), snake direction, or normalizing constant.

Description

These methods facilitate the retrieval and accessing of contour details.

B.6.18 Displaying contour shape

Synopsis

```
void display(short mag = 1);
```

Arguments

`mag`	Image magnification factor.

Description

`display` shows the shape of a contour, which will always be in the center of an X window.

B.6.19 Showing a contour on another image

Synopsis

```
void show(IMAGE *backgnd, unsigned char blowup=1,
          int pt_Xoffset=0, int pt_Yoffset=0, short expand=1);
```

Arguments

`backgnd`	Background image.
`blowup`	Image magnification factor.
`pt_Xoffset, pt_Yoffset`	Offset of a contour against a window origin.
`expand`	Expanding factor.

Description

`show` display an expanded contour on top of the background image by an offset of (pt_Xoffset, pt_Yoffset).

B.6.20 Example : Initialization of contours

This program performs the initialization of an opened and closed contour.

```
void testmain( SNAKEMODE smode,         /* snake mode */
               int Sx, int Sy,          /* starting point of a
                                           line */
               int Ex, int Ey,          /* ending point of a
                                           line */
               double Radius,           /* radius of circle */
               short num_points,        /* number of snaxels */
               int mag )                /* magnification factor */
{
        CONTOUR mycontour;              /* CONTOUR object */
        SNAXEL *sptr;                   /* SNAXEL objcet */
        register short i = 0;

        /* Initialize an arbitrary close snake */
        if (smode == _CLOSED)
                mycontour.init(Sy, Sx, Radius, num_points);
        else
                mycontour.init(Sx, Sy, Ex, Ey, num_points);

        // The average length of snaxel must be calculated first.
        mycontour.computeAvgLength();
        printf("\nContour information : \n");
        mycontour.print();
        printf("\nPress enter to display contour.");
        getchar();
        mycontour.display(mag);

        // Internal energy of snake without deformation should be 0
        printf("\nInternal energy.");
        for(sptr=mycontour.getHead(); sptr; sptr=sptr->getNext(),
                                                            i++)
                printf("\nEmodel of snaxel %d = %f",
                                i, mycontour.EInternal(sptr));

        printf("\nPress enter to continue");
        getchar();

        printf("\nEnd of test.\n");
}
```

getAvgLength must be called before the internal energy calculation so as to compute the average distant of snaxels. A contour should have internal energy of zero if no deformation happens.

B.6.21 Example : Affine transformations of contour

This program demonstrates the affine invariance of shape matrix. The internal energy before and after the transformation should be of the same.

```
void testmain( char *confile,           /* contour file */
               int mag,                 /* magnification factor */
               double angle,            /* rotation angle */
               int tx, int ty,          /* translation vector */
               double sx, double sy,    /* scaling factor */
               double dx, double dy,    /* dilution vector */
               char *outfile)           /* output file */
{
        CONTOUR mycontour;              /* CONTOUR object */
        SNAXEL *sptr;                   /* SNAXEL object */
        register short i;
        double temp;

        /* read a contour file */
        if (mycontour.read(confile)) exit(-1);
```

```
mycontour.display(mag);

// Calculate average length and internal energy of snaxel
temp = mycontour.computeAvgLength();
mycontour.computeShape();
mycontour.print();
for(i=0, sptr=mycontour.getHead();
                        sptr; sptr=sptr->getNext(), i++)
   printf("\nEmodel of snaxel %d = %f", i,
                        mycontour.EInternal(sptr));
printf("\nPress enter to continue");
getchar();

/* affine transformation should be done in contour centered
   formed */
mycontour.contourCentered();
mycontour.affineTransform( sx, sy, angle, tx, ty, dx, dy );
printf("\nShowing contour after transformations.");
mycontour.imageCentered();
mycontour.computeCG();
mycontour.display(mag);
printf("\nPress enter to continue");
getchar();

/* internal energy should be invariant to affine
   transforms */
printf("\nPerforming internal energy calculation after
       transforms.\n");
for(i=0, sptr=mycontour.getHead(); sptr;
                            sptr=sptr->getNext(), i++)
   printf("\nEmodel of snaxel %d = %f", i,
                            mycontour.EInternal(sptr));

/* Performing shape learning */
printf("\nContour and snaxel coefficients after learning.");
printf("\nAverage distance: %f\n",temp);
mycontour.computeShape();
mycontour.display();
mycontour.print();
printf("\nPress enter to continue");
getchar();

/* write contour to output file */
if (outfile) {
    printf("\nWriting contour to file %s", outfile);
```

```
            mycontour.write(outfile);
        }
}
```

B.6.22 Example : Coordinate conversion of contour

This program shows the manual initialization of a contour with mouse. Besides, `imageCentered` and `contourCentered` convert snaxels coordinates from contour centered form to image centered form and vice versa.

```
void testmain( char *imgfile,           /* image file */
               INITMODE imode,          /* manual initialization
                                           mode */
               SNAKEMODE smode,         /* snake mode */
               int numpts,              /* number of snaxels */
               int mag )                /* image magnification
                                           factor */
{
        CONTOUR mycontour;              /* CONTOUR object */
        SNAXEL *sptr;                   /* SNAXEL object */
        IMAGE myimage;                  /* IMAGE object */
        register short i;

        /* Read and show image */
        if (myimage.read(imgfile)) exit(-1);
        myimage.show(mag);

        // Generating line or circle based on image automatically
        if (imode == _LOADTEMPLATE) {

            int row, col;

            row = myimage.getRow();
            col = myimage.getCol();

            if ( smode == _CLOSED )
                mycontour.init( row/2, col/2,
                                (double)MIN(row,col)/4.0,numpts );
            else
                mycontour.init( (short)(col/4), (short)(row/4),
                         (short)(3*col/4), (short)(row/2), numpts );

        }

        else {
                /* Initialise contour with mouse*/
                mycontour.init(&myimage, mag, imode, smode);
        }

        mycontour.display(mag);
        printf("\nPress enter to continue.");
        getchar();

        /* first calculate Cg of contour */
        mycontour.computeAvgLength();
        mycontour.computeShape();
        mycontour.computeCG();
        printf("\nCG Row= %f CG col = %f",
                mycontour.getCgRow(), mycontour.getCgCol());

        /* Converting co-ordinate form */
        printf("\n * Getting contour centered co-ordinates *\n");

        mycontour.contourCentered();
        mycontour.print();
        for(sptr=mycontour.getHead(), i=0; sptr;
                                 sptr=sptr->getNext(), i++)
                printf("\nInternal energy of snaxel %d = %f",
                          i, mycontour.EInternal(sptr));

        printf("\nPress enter to continue.");
        getchar();

        printf("\nConverting back to image centered\n\n");
        mycontour.imageCentered();
        mycontour.print();
        for(sptr=mycontour.getHead(), i=0; sptr;
                                 sptr=sptr->getNext(), i++)
                printf("\nInternal energy of snaxel %d = %f",
                          i, mycontour.EInternal(sptr));
        printf("\nEnd of test.\n");
}
```

B.6.23 Example : Duplication of contour

This program shows the different modes of contour duplication.

```
void testmain(char *file1, char *file2)
{
        CONTOUR con1;
        CONTOUR *con3 = NULL;
        CONTOUR con2;

        printf("\nReading contour 1 from %s", file1);
        printf("\nReading contour 2 from %s",file2);
        if ( (con1.read(file1)) ||  (con2.read(file2)) ) exit(-1);


        printf("\n\n**** Contour 1 co-ordinates ****\n");
        con1.print();
        printf("\n\n**** Contour 2 co-ordinates ****\n");
        con2.print();
        printf("\nPress enter to continue");
        getchar();

        printf("\nCreating new contour from contour 1.");
        con3 = con1.duplicate(con3);
        con3->print();

        printf("\nCopying snaxel information only from contour 2");
        con3 = con2.duplicate(con3, 1);
        printf("\nPrinting contour information\n");
        con1.print();
        con3->print();
        getchar();
}
```

The output of the program should show that `con3` have properties such as `sigma_x` and `avglen` that of `con1` and have snaxel information that of `con2`.

B.7 GHOUGH : Generalised Hough Transform Object

Based on generalized Hough transform (GHT) [5], `GHOUGH` is a class for localizing objects of a particular shape from an input image or edge map. It is defined as follows :

```
class GHOUGH {

    protected :
        IMAGE **Planes;                 /* Planes of accumulator
                                           cells */
        CONTOUR **Template;            /* Templates under
                                          transformations */
        double **Angle;                // desired gradient angles
        int NumPlane;                  /* actual # of planes */
        int Qx, Qy;                    /* X, Y resolutions of
                                          planes */
        double QR, Qrxy;               /* scale resolutions */
        double Qt;                     /* theta resolutions */
        double Qdx, Qdy, Qc;           /* dilation resolutions */
        int NR, Nrrxy, Nt, Ndx, Ndy;   /* # of cells in each of
                                          the axis */
        double R;                      /* scaling factor */
        double THETA;                  /* initial angle for ref.
                                          line angle */
        int plane_max;                 // store index for blanking
        int row_max;
        int col_max;
};
```

Our implementation is capable of localizing contour which may have undergone affine transformation. We separate affine transformation, T, into scaling change, $T(R)$, and dilation matrix, $T(d)$, centered at 1, and rotation matrix, $T(t)$, indexed by `THETA` centered at 0, as follows :

$$T = T(R)T(t)T(d) \tag{B.14}$$

where

$$\mathbf{T}(r) = R\begin{bmatrix} rxy & 0 \\ 0 & rxy \end{bmatrix} \text{ and } \mathbf{T}(d) = \begin{bmatrix} 1 & dx \\ dy & 1 \end{bmatrix} \text{ and } \mathbf{T}(t) = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

The `Q` value control the resolution of transformation, while `N` values control the number of cells or allowable range. For a given reference contour, the total instances of transformation generated are $(NR)(Nrxy)(Nt)(Ndx)(Ndy)$.

B.7.1 GHOUGH constructor

Synopsis

```
GHOUGH(int _Qx=1, int _Qy=1,
            int _NR=1, double  _QR=0.25,
            int _Nrxy=1, double _Qrxy=0.25,
            int _Nt=1,  double _Qt=RADIAN(10),
            int _Ndx=1, double _Qdx=0.1,
            int _Ndy=1, double _Qdy=0.1,
            double _R=1.0, double _THETA=0.0)
```

Description

The constructor initializes the values of transformation parameters.

B.7.2 GHOUGH destructor

Synopsis

```
~GHOUGH()
```

Description

The destructor frees the memory allocated to `Template`, `Planes` and `Angle`, and resets the plane indexing values.

B.7.3 Resetting GHOUGH object

Synopsis

```
reset()
```

Description

The destructor frees the memory allocated to `Template`, `Planes` and `Angle`, and resets the plane indexing values.

B.7.4 Finding one contour

Synopsis

```
CONTOUR *localize(CONTOUR *reference, IMAGE *img)
CONTOUR *localize(CONTOUR *reference, EDGE *edgeMap)
```

Arguments

`reference`	Reference contour.
`*edgeMap`	Edge map.
`*img`	Gaussian image or intensity image.

Returns

Localized contour.

Description

`localize` performs general Hough transform to locate the desired image feature. It generates various instance of transformation contours and correlates them with the underlying image or edge map. The instance which consists of maximun count at center of gravity after correlation, is considered as the best fit of template.

B.7.5 Finding multiple contours

Synopsis

```
CONTOUR **localize(CONTOUR *reference, IMAGE *img, int numFind)
CONTOUR **localize(CONTOUR *reference, EDGE *edgeMap, int numFind)
```

Arguments

`reference`	Reference contour.
`*edgeMap`	Edge map.
`*img`	Gaussian image or intensity image.
`numFind`	Number of localized contours.

Returns

List of localized contours.

Description

`localize` performs general Hough transform to locate multiple desired image features. It generates various instance of transformation contours and correlates them with the underlying image or edge map. The instances which consist of first `numFind` maximun count at center of gravity after correlation, are considered as the best fit of templates and will be stored in decreasing order.

B.7.6 Example : Localization of a contour

The following example demonstrates the match of a rigid contour with the underlying image by generalized Hough transform.

```
void testmain( char *imgfile,           /* image file */
               char *cfile,             /* contour file */
               int mag,                 /* magnification factor */
               short level )            /* pyramid level */
{
        PYRAMID mypyramid;              /* PYRAMID object */
        EDGE    *edgemap;               /* EDGE object */
        CONTOUR mycontour;              /* CONTOUR object */
        CONTOUR *localised;             /* CONTOUR object */
        SNAXEL  *sptr;                  /* SNAXEL object */
        GHOUGH  GHT_OBJ;                /* GHOUGH object */
        int     row, col;               /* image of col x row */

        if ( mypyramid.putRawImg(imgfile) )
                exit (-1);

        if (mycontour.read(cfile))
                exit(-1);

        /* Generate pyramid in verbose mode to level 3 */
        mypyramid.generate(level, 1);

        /* Get highest level in pyramid */
        printf("\n Perform localization ... wait\n");
        edgemap = mypyramid.getEdge(level - 1);
        localised = GHT_OBJ.localize( &mycontour, edgemap );

        /* show image */
        mypyramid.rawImg->show(mag);
        for ( sptr=localised->getHead(); sptr; sptr=sptr->getNext() )
                sptr->show( mypyramid.rawImg,
                            ROUNDOFF(sptr->getRow())*mag,
                            ROUNDOFF(sptr->getCol())*mag);

        printf("\nPress enter to end.");
        getchar();
}
```

`generate` builds one level of edge map and Gaussian pyramid images. With edge map as an input image, `localize` places a contour on the image feature of interest.

B.7.7 Example : Localization of multiple contours

```
void testmain( char *imgfile,           /* image file */
               char *cfile,             /* contour file */
               short numgsnake,         // number of desired snakes
               int mag,                 /* magnification factor */
               short level )            /* pyramid level */
{
        PYRAMID mypyramid;              /* PYRAMID object */
        EDGE    *edgemap;               /* EDGE object */
        CONTOUR mycontour;              /* CONTOUR object */
        CONTOUR **localised;            /* CONTOUR object */
        SNAXEL  *sptr;                  /* SNAXEL object */
        GHOUGH  GHT_OBJ;                /* GHOUGH object */
        int     row, col;               // image of col x row
        int     ratio;

        if ( mypyramid.putRawImg(imgfile) )
                exit (-1);

        if (cfile) {

                if (mycontour.read(cfile))
                        exit(-1);
        }

        else {

        /* Initialise a closed contour in the centre of the test image
           Radius of circle = smaller of row and col divide by 5.
           The values are arbitrary */

                row = mypyramid.rawImg->getRow();
                col = mypyramid.rawImg->getCol();

                mycontour.init(row/2,col/2,(double) MIN(row,col)/5,8);
        }

        /* Generate pyramid in verbose mode to level 3 */

        mypyramid.generate(level, 1);
        mypyramid.show(mag,level);
        printf("\nPress enter to continue.");
        getchar();
        printf("\nCo-ordinates before transform\n ");
        mycontour.print();
        mypyramid.rawImg->show(mag);

        for (sptr=mycontour.getHead();sptr;sptr=sptr->getNext())
                sptr->show( mypyramid.rawImg,
                            ROUNDOFF(sptr->getRow())*mag,
                            ROUNDOFF(sptr->getCol())*mag );

        printf("\nPress enter to continue.");
        getchar();

        /* Use external energy input Edge.The edge object used is the
           highest level one because the search area  will be the
           smallest */
        ratio = mypyramid.getLevel()-1;

        /* Reduce contour to size of highest edge */
        mycontour.expand(LEVEL(-ratio));

        /* Get highest level in pyramid */
        edgemap = mypyramid.getEdge(ratio);
        localised = GHT_OBJ.localize( &mycontour, edgemap, numgsnake );

        register short i;

        for ( i=0; i< numgsnake; i++ ) {

            printf("\nLocalised contour [%d] on image.\n", i );

            /* Get the localised gsnake. */
            /* Contour will be in natural image size */
            localised[i]->expand(LEVEL(ratio));
            localised[i]->print();

            mypyramid.rawImg->show(mag);
            for ( sptr=localised[i]->getHead(); sptr;
                                                sptr=sptr->getNext() )
                    sptr->show( mypyramid.rawImg,
                            ROUNDOFF(sptr->getRow())*mag,
                            ROUNDOFF(sptr->getCol())*mag);
            printf("\n\nPress Enter to Continue\n");
            getchar();
        }
}
```

In this program, `generate` builds a pyramid of edge map and Gaussian images, which are treated as input to general Hough transform (GHT). Since GHT performs only at the highest level of pyramid so as to reduce computational time, `expand` reduces the size of contour by `LEVEL(ratio)`. `localize` then finds the `numgsnake` of image features of interest.

B.8 GSNAKE : Generalized Active Contour Model

`GSNAKE` is a class for modeling and extracting arbitrary deformable contours. It consists of `PYRAMID` for external energy calculation and `CONTOUR` class for internal energy calculation. `GSNAKE` has the following structure :

```
class GSNAKE : public PYRAMID,
               public CONTOUR {

     protected :
        double global_lambda;   /* global regularization
                                   parameters */
        double Eint;            /* Total internal energy of
                                   snake */
        double Eext;            /* Total external energy of
                                   snake */
        double Esnake;          /* Total snake energy */
        EEXTTYPE EextType;      /* external energy type */
        EDGE *EdgeMap;          /* edge map in current pyramid
                                   level */
        IMAGE *GaussImg;        /* gaussImg in current pyramid
                                   level */
};
```

We initialize GSNAKE using generalized Hough transform (section 3.3 of [1]) and minimize its energy using dynamic programming with stratified line search (section 3.4 of[1]). The internal energy, `Eint`, measures deviation in shape irregularities, while the external energy, `Eext`, adjust the contour model to match the underlying image feature. Based on parameter selection strategy (section 2.2 of [1]), `glabal_lambda`, we regularize the tradeoff between `Eint` and `Eext`, and compute the total energy, `Esnake`. Our implementation allows user to select minimax criterion (`_LOCAL_MINMAX`), local regularization parameter (`_LOCAL_LAMBDA`) or global lambda parameter selection strategy (within the range of 0.0 to 1.0). `ExtType` is the type of external energy used, which includes edge gradient (`_EDGE`), edge magnitude (`_EDGEMAG`) and image intensity (`_INTENSITY`).

B.8.1 GSNAKE constructor

Synopsis

```
GSNAKE(EextTYPE etype = _EDGE, double glambda = _LOCAL_MINMAX)
```

Arguments

`etype`	External energy type, includes intensity (_INTENSITY), edge magnitude only (_EDGEMAG) and edge gradient (_EDGE) image energy.
`glambda`	Regularization parameter selection strategy, includes minmax criterion (_LOCAL_MINMAX) and local regularization (_LOCAL_LAMBDA).

Description

The constructor selects *minimax* criterion as its parameter selection strategy, and uses edge gradient as its external energy type if user does not specify explicitly.

B.8.2 GSNAKE destructor

Synopsis

```
~GSNAKE(void)
```

Description

The destructor initializes `EdgeMap` and `GaussImg` to NULL.

B.8.3 GSNAKE destructor

Synopsis

```
void reset(void)
```

Description

`reset` initializes `EdgeMap` and `GaussImg` to NULL

B.8.4 Generating pyramid images

Synopsis

```
int genPyramid(short level, short verbose)
```

Arguments

`level`	Levels of pyramid to be generated.
`verbose`	Verbose operation mode.

Returns

`NOERROR`	Successful Operation.
`MEMORYERROR`	Memory allocation error.

Description

`genPyramid` generate `PYRAMID` object based on `EextType` specified. To increase robustness, we use histogram equalization parameters ($high\_pct = 1.0$, $low\_pct = 1.0$, $high\_val = 1.0$, $low\_val = 1.0$) for external energy of type _INTENSITY, and use default condition parameters ($high\_pct = 0.95$, $low\_pct = 0.9$, $high\_val = 0.9$, $low\_val = 0.2$) for external energy of type _EDGE and _EDGEMAG.

B.8.5 Localization of contour

Synopsis

```
int localize( int _Qx=1, int _Qy=1,
              int _NR=1, double  _QR=0.25,
              int _Nrxy=1, double _Qrxy=0.1,
              int _Nt=1,  double _Qt=RADIAN(10),
              int _Ndx=1, double _Qdx=0.1,
              int _Ndy=1, double _Qdy=0.1,
              double _R=1.0, double _THETA=0.0 )
```

Arguments

`_Qx, _Qx`	X and Y resolution of `GHOUGH` accumulator space.
`_NR`	Number of cells in scale change axis.
`_QR`	Resolution for scale change.
`_Nrxy`	Number of cells in diagonal stretching change.
`_Qrxy`	Resolution for diagonal stretching.
`_Nt`	Number of cells in rotation axis.
`_Qt`	Resolution of rotation (in radian).
`_Ndx, _Ndy`	Number of cell in X and Y of dilution axis.
`_Qdx, _Qdy`	X and Y resolution of dilution.
`_R`	Scaling factor.
`_THETA`	Initial angle for reference line gsnake.

Returns

`NOERROR`	Localisation performed successfully.
`MEMORYERROR`	Unable to allocate memory for GHOUGH objects.

Description

Based on resolution and allowable range of transformation specified by users, `localize` performs rigid match of a contour with the underlying image by generalized Hough transform (GHT). If Gaussian image and edge map do not exist, it will generate level 1 `PYRAMID`. Localization will only be performed at highest level of pyramid so as to reduce computational time. To increase accuracy, it activates `fineLocalize` if `_Qx` or `_Qy` is greater than 1.

B.8.6 Fine localization of GSNAKE template

Synopsis

```
void fineLocalize( int _qx = 1, int _qy = 1,
   int _cg_col = 0, int _cg_row = 0 );
```

Arguments

`_qx, _qy`	X and Y resolution of translation.
`_cg_col, _cg_row`	Coordinate of center of gravity.

Description

`fineLocalize` moves the localized contour in a small area of `_qx x _qy`. It calculates external energy during each move and then relocates the contour at place of lowest energy. To increase efficiency, user can also place contour at location with center (`_cg_col, _cg_row`) before moving. This is useful when the contour can be in a fixed and small locality.

B.8.7 Minimization of GSNAKE

Synopsis

```
int minimize( int segmentSpacing = 5, int numSearchSegment = 5,
              int snaxelSpacing  = 5, unsigned char blowup = 1,
              int verbose = 0, double theLambda = _DEFINED_LAMBDA,
              int showImg = 1, int img_Xoffset = 0,
                                             int img_Yoffset = 0 );
```

Arguments

`segmentSpacing`	Coarse search spacing.
`numSearchSegment`	Number of searched segments around snaxel.
`snaxelSpacing`	Snaxel spacing.
`blowup`	Image magnification factor.
`verbose`	Verbose Flag.
`theLambda`	Lambda value.
`showImg`	Indication (0:off 1:on) of whether image is to be shown.
`img_Xoffset, img_Yoffset`	Offset of a contour from an showing image.

Description

Based on dynamic programming with stratified line search algorithm, `minimize` performs coarse to fine energy minimization. As minimization progresses from the highest to lowest pyramid level, it inserts new snaxels at interval of `snaxelSpa cing` between two snaxels. The `showImg`, `img_Xoffset` and `img_Yoffset` allow greater flexibility when applying minimization over a small region of image. In this case, we can cut image into portions and only perform minimization on portion of interest.

B.8.8 Marginalizing gsnake to get probablity

```
double marginalize(int nhoodTangent, int nhoodNormal)
```

Arguments

`nhoodTangent`	Number of searched segments along the tangent axis of snaxel.
`nhoodNormal`	Number of searched segments along the normal axis of snaxel.

Returns

Probability of match.

Description

`marginalize` sums probablities and finds location that maximize probablity of matched contour operation. The summation is done in a small region of `nhoodTangent` x `nhoodNormal` around a contour.

B.8.9 Calculating total energy of a gsnake

Synopsis

```
double ESnake( short level, int verbose = 0 );
```

Arguments

`level`	Pyramid level of interest.
`verbose`	Flag (0:off 1:on) to operate verbose mode.

Returns

Total energy of `GSNAKE`.

Description

`ESnake` compute the total energy of gsnake by regularizing the internal and external energy of each snaxel.

B.8.10 Calculating total energy of a snaxel

Synopsis

```
double ESnaxel( SNAXEL *now,
BOOLEAN store = _FALSE, int verbose = 0 );
```

Arguments

`*sxptr`	Pointer to target snaxel.
`store`	Flag to indicate whether to store snaxel energy.
`verbose`	Flag (0:off 1:on) to operate verbose mode.

Returns

Snaxel energy. Returns 1.0 if the snaxel coordinate is invalid.

Description

`Esnaxel` calculates the total energy of a snaxel by regularize its internal and external energy.

B.8.11 Calculating internal energy of a snaxel

B.8.12 Caluclating Internal energy at snaxel co-ordinates

```
double EInternal(SNAXEL *now)
```

Arguments

`now`	Pointer to target snaxel.

Returns

Internal energy of a snaxel.

Description

`EInternal` computes the internal energy of a snaxel.

B.8.13 Calculating external energy of a snaxel

Synopsis

```
  double EExternal(SNAXEL *now);
```

Arguments

`*now`	Pointer to the target snaxel.

Returns

External energy of a snaxel.

Description

`Eexternal` calculates the external energy of snaxel. If `EextType` is `_INETNSITY`, it will return value $1 -$ intensity. If `_EDGEMAG`, it will return value $1 -$ edgeMag. Otherwise, it will consider both magnitude and direction of an edge point (Eqn 3.18 of [1]).

B.8.14 Showing GSNAKE

Synopsis

```
void showLine( unsigned char blowup=1, int Xoffset=0, int Yoffset= 0)
```

Arguments

`blowup`	Image Magnification factor.
`Xoffset, Yoffset`	X and Y offset for image display.

Description

`showLine` draws lines between snaxels so as to form a complete contour. The offset values help in showing a contour at various image position.

B.8.15 Displaying gsnake and image

Synopsis

```
void show( unsigned char blowup =1,short expand = 1,
           int showImg = 1,
           int img_Xoffset = 0, int img_Yoffset = 0,
           int pt_Xoffset = 0, int pt_Yoffset = 0 );
```

Arguments

`blowup`	Image magnification factor.
`expand`	Contour expansion factor.
`showImg`	Flag tp indicate whether the raw image is to be shown.
`img_Xoffset, img_Yoffset`	X and Y offset of image position.
`pt_Xoffset, py_Yoffset`	X and Y offset of snaxel position.

Description

`show` displays snaxel coordinate on the raw image. If `showImg` and `img_offset` are not specified, the raw image and snaxel coordinate will be shown on an X window. Otherwise, they will be shown on top of an existing image with offset (`img_Xoffset`, `img_Yoffset`). The `pt_Xoffset` and `pt_Yoffset` are necessary when showing an expanded contour.

B.8.16 Manual deformation of a contour

Synopsis

```
void deform(short blowup = 1, short expand = 1)
```

Arguments

`blowup`	Image magnification factor.
`expand`	Contour expansion factor.

Description

`deform` allows manual deformation of a contour shape. By using a mouse, the user can adjust and move individual snaxel around. The expansion factor is necessary when dealing with an expanded contour.

B.8.17 Duplicating a GSNAKE

Synopsis

```
GSNAKE *duplicate(GSNAKE *target= NULL)
```

Returns

- Pointer to new GSNAKE if target is NULL. Otherwise, target will be returned.
- NULL if memory allocation fails.

Description

duplicate creates an exact duplication of itself including CONTOUR and PYRAMID if target is NULL. Otherwise, only GSNAKE parameters and CONTOUR but not PYRAMID are copied. The rationale for this is that the external energy input should be that of the target GSNAKE.

B.8.18 Getting external energy input type

Synopsis

```
EextTYPE getEextType( void )
```

Returns

External energy type.

Description

getEextType returns the external energy input type. The current state of EextType will determine the input type in any function requiring external energy input.

B.8.19 Getting and writing the regularization parameter

Synopsis

```
double getGLambda( void )
void putGLambda(double _lambda)
```

Description

These methods facilitate the reading and writing of the regularization parameter. The regularization parameter can also be set at GSNAKE constructor.

B.8.20 Retrieving internal and external energy

Synopsis

```
double getEsnake(void)
double getEint(void)
double getEext(void)
```

Returns

Internal or external energy value.

Description

These methods provide facility for accessing the energy values of GSNAKE. Since these values are stored during energy calculation routines, they may not be the most updated values.

B.8.21 Example : Localization and minimization of GSNAKE

A previous example in the section on `GHOUGH` demonstrated the localization of a contour on an image. However, with `GSNAKE`, all these are performed simply by invoking the class methods.

```
void testmain( char *imgfile,   /* image file */
               char *confile,   /* contour file */
               int level,       /* pyramid level */
               short mag,       /* image magnifiaction factor */
               int sspacing,    /* snaxel spacing */
               int ispacing,    /* search segment spacing */
               int nhood,       /* number segment */
               double lambda,   /* regularization parameter */
               EEXTTYPE Etype)  /* external energy type */
{
        GSNAKE mysnake(Etype);

        /* If no image file or contour file, cannot continue */
        if ( (mysnake.putRawImg(imgfile)) ||
                              (mysnake.CONTOUR::read(confile)) )
                exit(-1);

        /* Displaying the operating parameters */
        printf("\n********   Operating Parameters    ************\n");
        printf("\nImage : %s   Contour : %s ", imgfile, confile);
        printf("\nSearch Spacing : %d  Insertion spacing : %d",
                                        sspacing, ispacing);
        printf("\nSearch Nhood : %d, Lambda : %f",nhood, lambda);
        printf("\nExternal Energy input : ");

        switch (Etype) {
                case _INTENSITY :
                        printf("Intensity");
                        break;

                case _EDGE :
                        printf("Edge energy");
                        break;

                case _EDGEMAG :
                        printf("Edge magnitude only");
                        break;
        }

        if (verbose) printf("\nVerbose mode on");
        printf("\n\n*****************************************\n\n");

        mysnake.generate( level, 1 );
        mysnake.PYRAMID::show( mag, level );
        printf("\nPress enter to continue");
        getchar();

        printf("\nPerforming localization and minimisation");
        mysnake.GSNAKE::localize();
        mysnake.GSNAKE::minimize(sspacing, nhood, ispacing, mag,
                                 verbose, lambda, 1);
        printf("\nEnd of test. Press enter. ");
        getchar();
}
```

This program first reads a contour file and filters an image file as its `rawImg`. Based on the operating parameters, a pyramid is generated and `localize` and `minimize` methods are invoked. The contour used for localization and minimization should preferably be generated from the image itself. This is due to the problem of mismatching coordinates. For example, two contours with the same shape but with coordinates differing by 100 points might not be localized on the same features, depending on the localization parameters. If the localization parameter is too large, the program may take extremely long time. Besides, if the contour is larger than the image, it will probably result in an error.

B.8.22 Example : Energy calculation I

This program performs internal, external and total energy calculation of a gsnake before and after manual deformation.

```
void testmain( char *imgfile,   /* image file */
               int level,       /* pyramid level */
               int mag,         /* magnification factor */
               char *outfile)   /* output file */
{
        int row, col;           /* image of size colxrow*/
        GSNAKE mysnake;         /* GSNAKE object */
        SNAXEL *sptr;           /* SNAXEL object */
        short register i;

        if (mysnake.putRawImg(imgfile) ) exit(-1);

        row = mysnake.PYRAMID::rawImg->getRow();
        col = mysnake.PYRAMID::rawImg->getCol();
        mysnake.generate(level,1);

        /* Autoinitialising of a closed snake */
        /* centre at cg of image, radius= smaller of row/col
           divide by 4 */
        printf("\nCreating template from image dimension ..");
        mysnake.CONTOUR::init(row/2, col/2,(double)MIN(row,col)/4);
        printf("\nPerforming localisation of contour.");
        mysnake.localize();
        printf("\nLocalized !");
        printf("\nPress enter to continue ..");
        getchar();

        printf("[col row]\tEint\tEext\tEtotal\tLambda\n");
        printf("---------\t----\t----\t------\t------\n");
        mysnake.ESnake( 0, 1 );
        printf("\nTotal energy of snake: %f\n",mysnake.getEsnake() );

        /* Manual deformation of snake */
        mysnake.deform(mag);

        printf("[col row]\tEint\tEext\tEtotal\tLambda\n");
        printf("---------\t----\t----\t------\t------\n");
        mysnake.ESnake( 0, 1 );
        printf("\nTotal energy after deformation : %f\n",
                mysnake.getEsnake() );
        printf("\nPress enter to continue..");
        getchar();

        if (outfile) {
                printf("\nWriting to file %s\n\n",outfile);
                mysnake.CONTOUR::write(outfile);
        }
}
```

The program reads in an image file and automatically generates a _CLOSED contour. deform allows the user to adjust manually the contour shape. ESnake will calculate and compare energy values before and after deformation.

Manual deformation is useful in cases where the contour is more or less standard and it is only necessary to slightly modify the template to fit the image better. It is also useful for shape learning purposes.

B.8.23 Example : Energy calculation II

This program performs internal, external and total energy calculation of a gsnake before and after fine localization.

```
void testmain( char *imgfile,   /* image file */
               int level,       /* pyramid level */
               int mag,         /* magnification factor */
               int correlateX,  /* X resolution for correlation */
               int correlateY,  /* Y resolution for correlation */
               char *outfile)   /* output file */
{

        int row, col, i;
        GSNAKE mysnake;
        SNAXEL *sptr;

        if (mysnake.putRawImg(imgfile) ) exit(-1);

        row = mysnake.PYRAMID::rawImg->getRow();
        col = mysnake.PYRAMID::rawImg->getCol();
        mysnake.generate(level,1);

        /* Autoinitialising of a closed snake */
        /* centre at cg of image, radius= smaller of row/col
           divide by 4 */

        printf("\nCreating template from image dimension ..");
        mysnake.CONTOUR::init(row/2, col/2,(double)MIN(row,col)/4);
        printf("\nPerforming localisation of contour.");
        mysnake.localize();
        mysnake.computeCG();
        mysnake.computeAvgLength();

        printf("\nEnergy values before fine localisation ");
        printf("\n   Snake energy calculation");
        printf("\n********************************");
        printf("\n[col row]\tEint\tEext\tEtotal\tLambda\n");
        printf("---------\t----\t----\t------\t------\n");
        mysnake.ESnake( 0, 1 );
        printf("\nTotal energy : %f\n",
                mysnake.getEsnake() );
        mysnake.show(mag);
        printf("\nPress enter to continue ..");
        getchar();

        mysnake.fineLocalize( correlateX, correlateY );

        printf("\nTotal energy after fine localisation ");
        printf("\n   Snake energy calculation");
        printf("\n********************************");
        printf("\n[col row]\tEint\tEext\tEtotal\tLambda\n");
        printf("---------\t----\t----\t------\t------\n");
        mysnake.computeCG();
        mysnake.computeAvgLength();
        mysnake.ESnake( 0, 1 );
        printf("\nTotal energy : %f\n", mysnake.getEsnake() );
        mysnake.show(mag);
        printf("\nPress enter to continue ..");
        getchar();
}
```

In this program, `fineLocalize` reallocates the contour at a finer resolution. `ESnake` will calculate and compare energy values before and after fine localization.

B.9 MODEL : Shape Learning Class

`MODEL` provides shape matrix and local regularization parameters learning routines. It has the following structure:

```
class MODEL : public GSNAKE {
    protected :
        short deformSample;      /* number of deformed samples */
        short shapeSample;       /* number of learned samples */
};
```

MODEL inherits extra functions from GSNAKE. With sufficient training samples, we can generate a robust contour model with specific prior knowledge.

B.9.1 MODEL constructor

Synopsis

`MODEL(void)`

Description

The constructor sets `deformSample` and `shapeSample` to 0.

B.9.2 Learning shape matrix

Synopsis

`int LearnShape(GSNAKE *sample)`

Arguments

`sample`	Gsnake sample.

Returns

`NOERROR`	Successfully operation.
`MEMORYERROR`	Memory allocation error.

Description

`LearnShape` performs learning of shape matrix from different samples. This is done by taking an initial estimate of shape matrix from the first sample. Using this shape matrix and *minmax* regularization, we minimize the second samples and average the shape matrix. By repeating this for the rest samples, we can derive a learned *model* to regenerate a new contour shape.

B.9.3 Learning local deformation variances

Synopsis

```
int LearnDeform(GSNAKE *sample)
```

Arguments

`*sample`	GSNAKE sample.

Returns

`NOERROR`	Successful operation.
`MEMORYERROR`	Memory allocation error.

Description

`LearnDeform` learns the local regularization parameters λ_i which control the `GSNAKE` deformation. By computing deformation variance (σ_i^2) and noise varaince (σ_η^2), we have λ_i as following,

$$\lambda_i = \frac{\sigma_\eta^2}{\sigma_\eta^2 + \sigma_i^2} \tag{B.15}$$

B.9.4 Accessing the trained model

Synopsis

```
GSNAKE *getModel(void)
```

Returns

Learned contour model.

Description

`getModel` facilitates the retrieval of a learned contour model.

B.9.5 Example : Learning of shape matrix from different samples

```
void testmain( char **argv,
               unsigned char mag,
               short level,
               int magPos,
               int levelPos )
{
        MODEL model;            /* to store results of learning */
        GSNAKE sample(_EDGE);   /* sample will use _EDGE as external
                                   energy */
        char **imgsamples;      /* image samples */
        register short i;

        imgsamples = &argv[1] ;

        for( i=1; *imgsamples ; i++, imgsamples++) {

            if ( ( i == magPos ) || ( i == levelPos) )
                break;

            printf("Using Sample %s to Learn SHAPE\n\n", *imgsamples);

            sample.putRawImg(*imgsamples);

            if( i==1 ) {

                /* use manually selected feature points to
                   estimate the shape matrix */

                sample.CONTOUR::init( sample.rawImg, mag );
                model.LearnShape( &sample );
            }

            /* Using the initial shape matrix and minimiax
               regularization, the total energy of gsnake is
       minimized and then the shape matrix is updated */

            model.duplicate(&sample);
            sample.generate(level, 1);          /* generate pyramid */

            sample.localize(5, 5, 1, 0.25, 3);  /* localize the
                                                   contour */
            sample.minimize(5, 5, 0, mag);      /* minimize energy */
            sample.deform( mag );                // manually adjust

            model.LearnShape( &sample );         /* average out the
                                                    shape coef*/

            /* Using the shape matrix and the last two snaxels,
               a contour is regenerated to show invariance of shape
               matrix */

            if( i != 1 ) {

                printf("Regenerate the shape...\n");
                model.regenerate();     /* regenerate the shape
                                           based on mtx */
                model.CONTOUR::display( mag );
                printf("Press Enter to continue...\n");
                getchar();
             }
        }

        printf("\nResulting contour :\n");
        model.CONTOUR::print() ;
}
```

This program reads in one sample at each interaction. An initial estimate of the shape matrix are computed from the first sample. `localize` and `minimize` will localize the second sample and minimize its energy, and then `LearnShape` updates the new shape matrix. Since the shape matrix is regenerative, `regenerate` will generate a new contour. By repeating this precedure for many samples, we can obtain a learned model. If we take several square images which undergo affine transformation to train the shape matrix, the program will show that the regenerative shape is still a square. This verify the invariance of a shape matrix.

B.10 CLASSIFY : Contour Classification

`CLASSIFY` provides advance routines for detecting and classifying deformable contours directly from noisy image (Chapter 4 of [1]). It calculates the score of each competitive templates based on marginalization of the distribution (`_MARGIN_PROB`), MAP probability (`_DEFORM_PROB`), match of deformable template (`_DEFORM_MATCH`) and match of rigid template (`_RIGID_MATCH`). It has the following structure :

```
class CLASSIFY : public REGION {
 protected :
     int numClass ;                /* number of class */
     EEXTTYPE Eexttype ;            // type of external energy
     CONTOUR **templates ;          /* reference templates */
     char **labels ;                /* label of templates */
     double *Margin_Prob;           // marginalized probability
     double *Rigid_Match ;          /* rigid match score */
     double *Deform_Match ;         /* deform match score */
     double *Deform_Prob ;          // deform match probability

/* these localization and minimization parameters are made
   public so that modifying them can be easy */
public :
     int Qx, Qy ;                   /* GHT Image Cell
                                              Resolutions */
     int NR, Nrxy, Nt, Ndx, Ndy ;    /* GHT ranges */
     double QR, Qrxy, Qt, Qdx, Qdy;  /* GHT Resolutions */
     double R, THETA ;              /* GHT Constants */
     int numSearchSegment ;         // number of search segments
     int segmentSpacing ;           // spacing between segments
     int numLevel ;                 // number of pyramid levels
     int verbose ;                  /* verbose mode */
     int nhoodTangent, nhoodNormal;  /* neighborhood used in
                                       marginaliz-n */
```

It stores a list of competitive templates and their corresponding score. In addition, extra parameters such as GHT range and resolution are used for localizaing and minimizing these templates.

B.10.1 CLASSIFY constructor

Synopsis

```
CLASSIFY(EEXTTYPE _Eexttype = _EDGE);
```

Arguments

`_Eexttype`	External energy type.

Description

The constructor sets the default parameters and initializes all pointer members to NULL.

B.10.2 CLASSIFY destructor

Synopsis

```
~CLASSIFY(void);
```

Description

The destructor frees memory allocated to the pointer members.

B.10.3 Loading a template into CLASSIFY

Synopsis

```
int read( CONTOUR *contour );
int read( char *filename );
```

Arguments

`contour`	Source `CONTOUR`.
`filename`	Contour file.

Returns

`NOERROR`	Template loaded successfully.
`MEMORYERROR`	Memory allocation failure or template initialization error

Description

These methods read in contour used for classification purpose. Once a contour is read, `numContour` is incremented by one, reflecting the total number of contours read.

B.10.4 Classifying templates

Synopsis

```
int classify(IMAGE *testimg, CLASSTYPE ClassType = _MARGIN_PROB) ;
```

Arguments

`testimg`	Testing image.
`ClassType`	Type of classification : `_MARGIN_PROB`, `_DEFORM_PROB`, `_DEFORM_MATCH` or `_RIGID_MATCH`.

Returns

Classification score.

Description

`classify` classify rigid and deformable templates directly from the testing image. For each template, it performs localization, minimization and marginalization to compute the classify score.

B.10.5 Selecting the best matching contour

Synopsis

```
int selectMax( ClassTYPE ClassType = _MARGIN_PROB )
```

Arguments

`ClassType`	Type of classification : `_MARGIN_PROB`, `_DEFORM_PROB`, `_DEFORM_MATCH` or `_RIGID_MATCH`.

Returns

Index to the template which consists of highest score.

Description

`selectMax` compares the classify score of templates of type interest and searches for the highest classify score.

B.10.6 Getting the number of templates read

Synopsis

```
int getNumClass( void )
```

Returns

Number of templates read.

Description

`getNumClass` retrieves the number of templates read.

B.10.7 Getting the matching score

Synopsis

```
double getScore( int template_id, CLASSTYPE ClassType = _MARGIN_PROB)
```

Arguments

`template_id`	Target template index.
`ClassType`	Type of classification : `_MARGIN_PROB`, `_DEFORM_PROB`, `_DEFORM_MATCH` or `_RIGID_MATCH`.

Returns

Classify score of target template.

Description

`getScore` retrieves the score of template indexed by `template_id`.

B.10.8 Printing templates score

Synopsis

```
void dump(char *imgName, FILE *stream = stdout)
```

Description

dump prints onto screen or file the score of each templates.

B.10.9 Getting label of template

Synopsis

```
char *getLabel(short class_id)
```

Arguments

ClassType	Type of classification : _MARGIN_PROB, _DEFORM_PROB, _DEFORM_MATCH or _RIGID_MATCH.

Returns

Label of template of interest.

Description

getlabel facilitates the retrieval of template label.

B.10.10 Example : Classifying various templates

```
void testmain( char **imgsamples ) /* image samples */
{
        register short i ;
        CLASSIFY shape(_EDGE) ;

        shape.Nrxy = 3 ;    /* allow some stretching in
                               diagonal direction */
        if( shape.read("ellip.con") != NOERROR ||
            shape.read("rect.con") != NOERROR  ) {
                printf("cannot find contour files\n") ;
                exit(-1) ;
        }


        for(i=0; *imgsamples; i++, imgsamples++) {

                IMAGE myImage ;
        if( myImage.read(*imgsamples) != NOERROR )
                break ;

        int id = shape.classify(&myImage, _MARGIN_PROB) ;

        printf("%s : %s\n",*imgsamples,shape.getLabel(id));

        shape.dump(*imgsamples) ;
}
exit(0) ;
```

This program read in two templates, namely `ellip.con` and `rect.con`. `classify` maginalizes the distribution and returns the matched template. `dump` will print on screen the classfy scores of the template.

Appendix C

GSNAKE API Command Line Utilities

GSNAKE is the Open Sourc segmentation tool we have used to build the lumen segmentation module. This section illustrates the modules that we are using.

C.1 Image Processing utilities

C.1.1 Image generation: `imggen`

Synopsis

```
imggen -[option]<value>
```

Optional switches

H	Print help screen.
V	Verbose mode.

Contour specifications :

C	Contour type switch Circle : Use a circle contour. Default. Rect : Use a rectangular contour. Manual : Use mouse to generate contour shape. Filename : Use contour file.
Mx	Magnification factor in X dimension. Default is 1.
My	magnification factor in Y dimension. Default is 1.
dx	Displacement in X dimension. Default is 0.
dy	Displacement in Y dimension. Default is 0.
R	Rotation angle in degrees. Default is 0.
Dx	Dilation in X dimension. Default is 0.
Dy	Dilation in Y dimension. Default is 0.
N	Number of snaxels (for circles and rectangles). Default is 8.
X	Standard deviation of local deformation. Default is 0.

Image specifications :

F	Output image name. Default is `test`.
r	Number of image rows. Default is 129.
c	Number of image columns. Default is 129.
B	Black pixel value. Default is 50.
W	White pixel value. Default is 150.
S	Standard deviation of image gaussian noise. Default is 0.

Function

`imggen` generates images based on a `_CLOSED` contour. The contour can be read from a given file, generated manually by mouse or automatically by the program itself. The image file generated will be of SUN raster file type.

Example

`imggen -S10 -X0.1` generates an image, `test.rs`, consists of a circle with the standard variation of the image noise set at `10`, and the standard deviation deformation set at `0.1`.

C.1.2 Image viewing: `imgshow`

Synopsis

```
 imgshow <image file1> <image file2> .. -[option]<value>
```

Optional switches

M	Magnification factor. Default value is 1.

Function

`imgshow` views images at a specified magnification factor. The file can be of either _bin or SUN raster type.

Example

`imgshow rect.rs.*` shows the sequence of images with root name `rect.rs`.

Error messages

File error	Cannot read or open an image file.
Memory error	Memory allocation failure due to incorrect file type.

C.1.3 Gaussian pyramid images generation: `imgpyramid`

Synopsis

```
imgpyramid <image file> -[option]<value>
```

Switches

L	Number of levels of pyramid to be generated. Default 2.
M	Magnification factor. Default 1.
Pl,Ph	Percentage range for histogram specification. Default 0.9-0.95.
Vl,Vh	Intensity range for histogram specification. Default 0.2-0.9.
E	Exponential value for transformation function. Default 1.

Function

`imgpyramid` displays a pyramid of Gaussian images and edge maps. To increase robustness, we condition edge maps using default histogram specification parameters. Histogram equalization can also be done by setting the percentage range from 0 to 1, and the intensity range from 0 to 1.

Example

`imgpyramid rect.rs.1` generates image pyramid for the image `rect.rs.1` up to level 2. The gaussian images, edge magnitudes and directions are shown on the screen.

Error messages

Memory error	Memory allocation failure.
File error	Cannot read or open an image file.

C.1.4 Image conditioning : `imgcond`

Synopsis

```
imgcond <file> -[option]<value>
```

Switches

M	Magnification factor. Default 1.
Pl,Ph	Percentage range for histogram conditioning. Default 0.9-0.95.
Vl,Vh	Intensity range for histogram conditioning. Default 0.2-0.9.
E	Exponential value for transformation function. Default 1.
O	Output filename, writes image file of SUN raster type if option '/' is available. Otherwise, write file of _bin type.

Function

`imgcond` performs histogram conditioning of image.

Example

`imgcond akina.rs` performs histogram equalization on the image `akina.rs`.

Error messages

Memory error	Memory allocation failure.
File error	Cannot read or open an image file.

C.1.5 Image learning : `imglearn`

Synopsis

```
imglearn <contour> <image samples>
```

Function

With sufficient training samples, `imglearn` generate a template with specific prior knowledge of shape matrix and deformation varaince.

Example

`imglearn test.con rect.rs.*` uses the series of images with root `rect.rs` to learn an contour named `test.con`. For the first image, the user will first use the mouse to initialise the snaxel positions. The program will then deform the snaxels to fit the image using minimax regularisation. The users will be asked to confirm or move the final positions of the snaxels using the mouse.

Error messages

Memory error	Memory allocation failure.
File error	Cannot read or open an image file.

C.2 GSNAKE utilities

C.2.1 Template generation : `gsinit`

Synopsis

```
gsinit <file> -[option]<value>
```

Switches

-I		Initialization mode. Default mode is automatic.
		Submodes :
	d	Drag. Use mouse to drag a contour of desired shape.
	c	Click. Use mouse to click a number of points over the image. Snaxels will be joined in sequential order.
	a	Automatic. Generate a template based on the image dimension.
-S		Contour mode.
	o	Opened contour.
	c	Closed contour.
-M		Magnification factor. Default 1.
-N		Number of snaxels. Default 16.
-O		Output file. Default filename ¡contour¿

Function

`gsinit` generates a template of desired shape.

Error messages

Unable to read file	File specified is non existent.
Wrong file format	File specified is not an image file.

C.2.2 Contour viewing: `gsshow`

Synopsis

```
gsshow <contour file1> <contour file2> .. -[option]<value>
```

Switches

M	Magnification factor. Default value is 1.

Function

`gsshow` views contours at a specified magnification factor. Since the contour shown will be at the center of a window, it is difficult to visualize translation of a contour.

Error messages

Unable to read file	File specified is non existent.
Wrong file format	File specified is not a contour file.

C.2.3 Contour matching: `gsfit`

Synopsis

```
gsfit <image file> <contour file> -[option]<value>
```

Switches

L	Level of pyramid to build to, default 2.
M	Magnification factor, default 1.
E	External energy input type, default e.
	i - Intensity e - Edge Data.
	m - Edge Magnitude only.
S	Stratified Search spacing.
	s - snaxel spacing, default 5.
	d - search segment spacing, default 2.
	n - number of search segment, default 2.
R	Regularization parameter, defaut _LOCAL_MINMAX.
Pl,Ph	Percentage range for histogram specification, default 0.9-0.95.
Vl,Vh	Intensity range for histogram specification, default 0.2-0.9.
e	Exponential Factor, default 1.
/	Activate verbose, default 0.

Function

`gsfit` performs the match of a deformable contour with the underlying image.

Example

`gsfit ellip.rs.1 ellip.con`.

Error messages

Memory error	Memory allocation failure.
File error	Cannot read or open an image file.

Appendix D

STL

D.1 STL Introduction

A standard format for rapid prototyping is the Stereo Lithographic (STL) format. A set of functions in the C programming language has been developed for reading and writing STL data in either binary or ascii format. In addition, functions have been developed to write data in Geomview "OFF" format and Open Data Explorer format. The basic reading and writing routines uses the algorithms use the ADMesh routines copyrighted by Anthony D. Martin.

One important attribute of this package is that the internal representation is that of triangles with vertices labeled by integers. The integers are indices to an array of points that have floating point positions. The internal format is more suitable for mesh calculations than is the STL format in which the vertices are floating point positions. The STL format does not indicate when vertices of different faces are actually the same point, whereas, the internal format gives the same index to all vertices that are the same point in space.

An example of usage is shown below.

```
#include <stdio.h>
#include <string.h>
#include "stereolith.h"

int main(int argc, char* argv[]) {
  StlData data;
  int status, ifdebug;
  char filename[MAX_STEREOLITH_FILENAME_LEN];
  char filename_off[MAX_STEREOLITH_FILENAME_LEN];
  char filename_dx[MAX_STEREOLITH_FILENAME_LEN];
  double delta, epsilon;

  delta = 0.000002;
  epsilon = delta*0.25;
  strcpy(filename, "samples/Yoke.stl");
  status = read_stl(filename, &data, delta, epsilon);
  if(status != 0) {
    fprintf(stderr," Error, read_stl returned %d\n",
    status);
    fprintf(stderr,"line %d file %s\n", __LINE__, __FILE__);
  }
  ifdebug = 1;
  status = fix_normals(&data, ifdebug);
  strcpy(filename_off, "testout.off");
  write_stl_to_off(filename_off, &data);
  strcpy(filename_dx, "testout.dx");
  write_stl_to_dx(filename_dx, &data);

  return 0;
}
```

D.2 Overall Description

This set includes functions to read and write STL data in either binary or ascii format. In addition, there are functions to write data in Geomview "OFF" format and Open Data Explorer format.

One important attribute of this package is that the internal representation is that of triangles with vertices labeled by integers. The integers are indices to an array of points that have floating point positions. The internal format is more suitable for mesh calculations than is the STL format in which the vertices are floating point positions. The STL format does not indicate when vertices of different faces are actually the same point, whereas, the internal format gives the same index to all vertices that are the same point in space.

The function that reads the data is given a double precision number "delta" which should be smaller than the distance between any two distinct vertices. Vertices within a distance "delta" are given the same index and assigned to the same spatial point. The read-in function is also given a double precision number "epsilon" that is smaller than "delta". Vertices within a distance epsilon are considered to be the same point, without a doubt. Vertices separated by more than epsilon but less than delta are assigned to the same point but a count of such cases is made and written to standard output. One can think of epsilon as being a little larger than the uncertainty imposed by the binary representation and one can think of delta as being a little smaller than the minimum resolution of the machine that will do the prototyping. Testing this program on many files showed that it was convenient to automatically scale epsilon and delta by the largest width, length or height of the piece. The read-in program automatically does a rescaling of delta and epsilon.

This packages requires the hash table functions found in the package HASH.

An example of usage is shown below.

```
#include <stdio.h>
#include <string.h>
#include "stereolith.h"

int main(int argc, char* argv[]) {
  StlData data;
  int status, ifdebug;
  char filename[MAX_STEREOLITH_FILENAME_LEN];
  char filename_off[MAX_STEREOLITH_FILENAME_LEN];
  char filename_dx[MAX_STEREOLITH_FILENAME_LEN];
  double delta, epsilon;

  delta = 0.000002;
  epsilon = delta*0.25;
  strcpy(filename, "samples/Yoke.stl");
  status = read_stl(filename, &data, delta, epsilon);
  if(status != 0) {
    fprintf(stderr," Error, read_stl returned %d\n",
    status);
    fprintf(stderr,"line %d file %s\n", __LINE__, __FILE__);
  }
  ifdebug = 1;
  status = fix_normals(&data, ifdebug);
  strcpy(filename_off, "testout.off");
  write_stl_to_off(filename_off, &data);
  strcpy(filename_dx, "testout.dx");
  write_stl_to_dx(filename_dx, &data);

  return 0;
}
```

*/

D.3 Description of the functions

Constants for the user

MAX_STEREOLITH_FILENAME_LEN This cpp macro is the maximum length (including end-of-string NULL) of a filename passed to the functions.

Constants used internally

MAX_STEREOLITH_HEADER_LINE This cpp macro is the size of the string that holds each line read.

STL_LABEL_SIZE This cpp macro is the size of a binary label.

STL_HEADER_SIZE This cpp macro is the size of a binary header: label plus number of facets.

SIZEOF_STL_FACET This cpp macro is the size of a block that describes one facet in a binary file.

Input and output data structures The input and output of binary files has been taken from the set of utilities called ADMesh.

The structures used include

stl_vertex,

stl_normal,

stl_facet,

stl_extra array,

stl_type enumeration type

ADMesh I/O functions Reading and writing integers and floating point numbers use routines taken from ADMesh, these are

stl_get_little_int,

stl_get_little_float,

stl_put_little_int and

stl_put_little_float.

Vector3 Any vector of three components, whether a normal vector or a position in space. **Vector3** is a typedef for **struct space_vector**, The components are **x**, **y** and **z**.

Facet A triangular face. **Facet** is a typedef for struct **one_facet**. The components, all of type **Vector3**, are positions **vertices[3]** and normal **normal**. This data structure is used for reading-in the data. The data is then converted to **StlData**, which is more useful for describing a mesh.

Triangle This type of face has a normal vector, called **Vector3 normal** and an *index* for each vertex, in an integer array **int vertices[3]**. The index points into the array **points** of the struct **stl_data**.

Triangle is a typedef for struct **one_triangle**.

StlData The entire surface geometry description is contained in **StlData**. **StlData** is a typedef for struct **stl_data**. The components are **int num_points**, **Vector3 *points**, **int num_triangles** and **Triangle *triangles**. The component **points** is an array of vertex positions. The component **triangles** is an array of faces. The structure **Triangle** (→ **??**) for each triangle uses indices that point into the array **points**.

dot The function **dot** returns the inner product of two Vector3 parameters.

magnitude The function **magnitude** returns the magnitude of a Vector3 parameter.

distance The function **distance** returns the distance between two Vector3 parameters.

add The function **add** returns a Vector3 that is the sum of two Vector3 parameters.

subtract The function **subtract** returns a Vector3 that is the difference of two Vector3 parameters, the first minus the second.

cross The function **cross** returns a Vector3 that is the cross product of two Vector3 parameters.

area The function **area** returns the area between three Vector3 points.

read_stl The function **read_stl** reads triangles of a surface in STL (stereolithograph) format. Note that when this function returns, space will be allocated inside a structure StlData given as a parameter. The function **free_stl_data** can be used to free that space.

All points with a distance delta of each other are treated as a single point. The distance epsilon should be smaller than delta and all points within a distance epsilon are surely the same point. If the distance between two points is larger than epsilon and smaller than delta, the position data should be considered ambiguous.

write_ascii_stl The function **write_ascii_stl** writes data in STL ascii format.

write_binary_stl The function **write_binary_stl** writes data in STL binary format. The binary format can have an 80-character ascii label written before the binary data.

free_stl_data As well as helping the user avoid dealing with the details of the stl_data struct, the function **free_stl_data** assists in the integration with C++. What is created with “new” should be returned to the heap with “delete” and what is created with “malloc” should be returned to the heap with “free”.

write_stl_to_off The function **write_stl_to_off** writes the surface mesh in Geomview 'off' format.

write_stl_to_dx The function **write_stl_to_dx** writes the data in Open Data Explorer format.

fix_normals The function **fix_normals** changes the normal vectors of the faces so that all normals point outward. The function may fail if the surface has holes or if the inside and outside surfaces are not distinct (a Klein bottle).