DIP*lib* Programmers Guide

dr. ir. Geert M. P. van Kempen
dr. ir. Michael van Ginkel
dr. ir. Cris L. Luengo Hendriks
prof. dr. ir. Lucas J. van Vliet

Quantitative Imaging Group,
Department of Applied Sciences,
Delft University of Technology

Delft
April 2, 2012

Contents

Chapter 1

Introduction

DIP*lib* is a scientific image processing library written in C. It consists of a large number of functions for processing and analysing multi-dimensional image data. The library provides functions for performing transforms, filter operations, object generation, local structure analysis, object measurements and statistical analysis of images.

The current release contains over five hundred (500) documented functions, and more than three hundred (300) of these functions provide image processing functionality. The remaining functions provide access to DIP*lib*'s data structures and other support functionality.

In the current release, only scalar images of standard C data types, binary data types and complex (floating point only) data types are supported. The data type is a property of an image. Our philosophy is that an image processing algorithm should not be tied too closely to the data type or dimensionality. In practice this means that a single function accepts images of various data types and dimensionality. We have attempted to deal sensibly with this. Two examples: the Gaussian filter accepts both integer and floating point images, but always returns a floating point image. A grey value dilation does not introduce new grey-values. The dilation function also accepts both integer and floating point images, but the output data type is the same as the input data type in this case. Both filter functions accept input images of arbitrary dimensionality. Some functions *are* tied to a particular dimensionality, for instance most skeletonisation algorithms, and these only operate on an image with the correct dimensionality.

The library does not contain any I/O or display functionality. A separate library, dipIO, is available for I/O functionality, with support for ICS, TIFF, JPEG, GIF and a variety of other file types.

We use MATLAB as our image processing environment and as frontend to DIP*lib*. The frontend, DIP*image*, is available as a MATLAB toolbox. Together MATLAB and DIP*image* yield a powerful workbench for working with scalar and vector images in any number of dimensions.

More information about DIP*lib* and DIP*image* can be found at their web page: http://www.diplib.org/.

Chapter 2

Conventions

This chapter describes the file and C statements conventions used in the DIP*lib* and dipIO source libraries. We will use the term *public* to specify that something (like a function) is documented, supported, and available on the DIP*lib* library level. Something is called *private* when it is undocumented and unsupported.

2.1 Include files

The include file `diplib.h` should always be included. This include file includes some other include files that are (almost) always necessary. The DIP*lib* library is internally organised as a set of smaller libraries, each with its own include file(s). The reference guide and the examples show which include file(s) should be used.

All DIP*lib* include files start with the prefix `dip_`.

The include file `dipio.h` should always be included if functions from dipIO are used. Other include files might be necessary; again, the reference guide shows which files to include.

All dipIO include files start with the prefix `dipio_`.

2.2 ANSI-C source code conventions

In general, names of variables, functions or structures which are composed out of two or more words are catenated by capitalising the first character of each word.

- Variables start with a lowercase character.
- Defines and macros are written capitals. Public macro's, defines and enumeration constants start with `DIP_`, and private ones with `DIP___`. Names composed out of two or more words, are catenated with an underscore.
- Public function and structure names start with the prefix `dip_`, followed by a function name starting with a capital. There is one exception to this rule. DIP*lib*'s simple numeric data types are entirely written in lower caps.
- Private function names start with the prefix `dip___`.
- Conditional statements are surrounded by braces, even single line ones.
- Functions which can only handle a specific data type have the suffix of this data type appended to the function name. For example, if the function `dip_MyOwnFunction` can only handle `dip_uint8`, its name will be `dip_MyOwnFunction_u8`.

Example:

```
#include "diplib.h"
/* dip_error.h is included through diplib.h */

dip_Error dip_MyFunction
(
   dip_Image in
)
{
   DIP_FN_DECLARE("dip_MyFunction");
   dip_int ii;

   DIPTS((param < DIP_MINIMUM_PARAMETER),
         dip_errorParameterOutOfRange );

   for( ii = 0; ii < param; ii++)
   {
      DIPXJ(dip__MyLowLevelFunction_sfl( in, ii ));
   }

dip_error:
   DIP_FN_EXIT;
}
```

Chapter 3

Initialisation

DIP*lib* and dipIO have to be initialised before use and also require some clean up operations after use.

3.1 Introduction

Before DIP*lib* functions can be used, the library needs to be initialised. This is achieved by calling the following function:

```
dip_Initialise();
```

It is safe to call `dip_Initialise` more than one time, only the first call will be effective.

The equivalent function for dipIO is:

```
dipio_Initialise();
```

3.2 Clean up

Before exiting a program linked with DIP*lib*, the following function should be called, allowing some clean up operations to be performed (preventing memory leaks):

```
dip_Exit();
```

The equivalent function for dipIO is:

```
dipio_Exit();
```

Chapter 4

Error handling

In order to use or write DIP*lib* functions it is necessary to know how errors are dealt with. DIP*lib*'s error mechanism has the following features:

- consistent error handling
- errors are handled using functions' return values
- a function call tree is maintained by the error functions and macros to facilitate the debugging of DIP*lib* code.
- each function in the call tree can have an arbitrary message associated with it.

Consider the following schematic piece of code:

```
main()
{
   DoSomethingClever();
   DoSomethingDumb();
}

DoSomethingClever()
{
   DoSomethingStupid();
}
```

Both `DoSomethingDumb()` and `DoSomethingStupid()` are rigged to always return an error. If `main()` is executed, a call tree as shown in figure 4.1 will be returned by `main()`.

In order to use the error mechanism it is necessary that a function is written using the proper initialisation and exit macros. These are described in section 4.1. Furthermore, functions that return DIP*lib* errors should not be called directly, but only through the macros described in section 4.2.

4.1 Writing a DIP*lib* function

Our error scheme requires that some local variables are defined and initialised. Therefore a DIP*lib* function should start with the `DIP_FN_DECLARE` macro. This macro takes the function name as an argument (this is needed for constructing a call tree). The function should be exited only through using the `DIP_FN_EXIT` macro. Both macros should be followed by a semicolon.

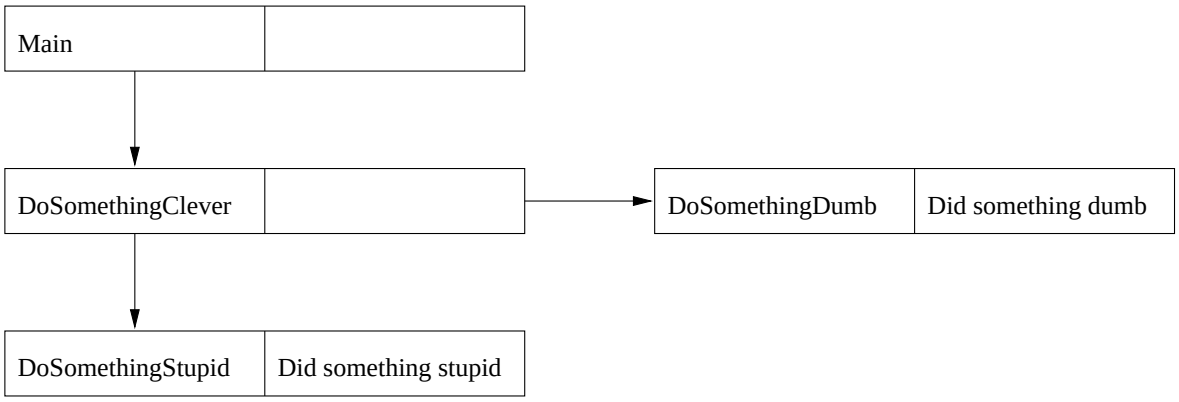

Figure 4.1: A DIP*lib* call tree. Each entry shows the function name and (optionally) a message.

A line containing the label `dip_error` should precede the `DIP_FN_EXIT` macro. This label is used by the error macros to jump to the end of the function if an error condition occurs. Only if your routine does not use one of these macros, can this label be omitted. Any cleanup operations, such as freeing memory, should be done after this label, but before the `DIP_FN_EXIT` macro.

The return type of the function should be `dip_Error`, which is a typedef for a pointer to the actual error structure `dip__Error`. If a function executes without generating an error, the zero pointer is returned (use the supplied macros to deal with errors, rather than manipulating the error structure yourself).

There is also a set of macros starting with `DIP_FNR`, which are similar to the `DIP_FN` macros. The introduction of these is deferred to chapter 5.

Summarising, a DIP*lib* function should:

- have a return type of `dip_Error`
- precede its declarations with the macro `DIP_FN_DECLARE`
- end with the label `dip_error`, followed by cleanup actions, and finally the `DIP_FN_EXIT` macro.

The template for a DIP*lib* function is as follows:

```
dip_Error
dip_MyFunction
(
   /* Your arguments */
)
{
   DIP_FN_DECLARE("dip_MyFunction");
   /* Your declarations here */

   /* Your code here */

   /*
    * Example of how to generate an error condition and set the
    * error message.
    */
   DIPSJ( "Your error message" );

dip_error:
   /* Your clean up code */
   DIP_FN_EXIT;
}
```

4.2 Error macros

In this section we describe the various macros for dealing with DIP*lib*'s error mechanism. The first and most common error related action is calling a function, checking if it returned an error and if so propagate the error to the caller of the current function. This task is performed by the `DIPXJ` macro. It has one argument: the function to be called. If this function returns an error, the macro jumps to the `dip_error` label. `DIP_FN_EXIT` takes care of adding the current function's name to the call tree and return the extended call tree to the caller. All macros that take care of error handling consist of the three letters DIP followed by two indicating their function. In this case "XJ", which is short for "e*X*ecute and *J*ump". Note that if only `DIPXJ` is used, the call tree will be reduced to a simple linked list.

Sometimes however, you do not wish to immediately terminate the current function, but do some further processing despite the fact that the function that was just called returned an error. This is possible using the `DIPXC` macro, which is short for "e*X*ecute and *C*ontinue". Using this macro may results in a true call tree with several branches. This macro should also be used when calling DIP*lib* functions after the `dip_error` label, otherwise we may get stuck in an infinite loop.

The macro `DIPSJ`, short for "*S*et and *J*ump", takes a string (`char` $*$) as an argument. The macro jumps to the end of the current function and passes the string to the `DIP_FN_EXIT` macro, which in turn adds the current function to the call tree and sets the message belonging to the current function to the string passed by `DIPSJ`. The message is copied, so the string does not need to remain valid after `DIP_FN_EXIT` is through with it. There are a number of

predefined static error messages, which you can use. A complete list is given in the reference manual.

It is also possible to pass dynamically allocated (using `dip_MemoryNew`) strings to `DIP_FN_EXIT`. These need to be explicitly freed by `dip_MemoryFree` after `DIP_FN_EXIT` has finished its job. To achieve this the string should be passed by the `DIPJF` macro, short for "set, *J*ump and *F*ree". `DIP_FN_EXIT` will now take care of freeing the string.

`DIPTS`, short for "*T*est, *S*et and jump", takes two arguments. The first is a test, the second the error message that should be passed to `DIP_FN_EXIT` if the test result is true. It is in fact just short hand for:

```
DIPTS( test, message ) is equivalent to
if ( test )
{
   DIPSJ( message );
}
```

Table 4.1 gives an overview of all the macros described in this chapter. It includes the `DIP_FNR` macros, which are discussed in chapter 5.

`DIP_FN_DECLARE( functionName )`	Declare local error variables
`DIP_FN_EXIT`	Take care of call tree and exit (return from) the current function
`DIP_FNR_DECLARE( functionName )`	Same as `DIP_FN_DECLARE` but also declares a `dip_Resources` structure named `rg`
`DIP_FNR_INITIALISE`	Initialise the resources structure
`DIP_FNR_EXIT`	Free resources, take care of call tree and exit (return from) the current function
`DIPXJ( function )`	Execute function and if an error occurs pass it to `DIP_FN(R)_EXIT`
`DIPXC( function )`	Execute function and if an error occurs add it to the call tree. Does *not* jump to the end of the routine
`DIPSJ( message )`	Pass error message to `DIP_FN(R)_EXIT`
`DIPJF( message )`	Pass error message to `DIP_FN(R)_EXIT` and free the message using `dip_MemoryFree`
`DIPTS( test, message )`	Perform the test, and if the result is true pass the error message to `DIP_FN(R)_EXIT`

Table 4.1: DIP*lib* error macros.

Chapter 5

Resource management

5.1 The scheme

One of the most error-prone tasks while writing a program is keeping track of the allocations that are done. All this bookkeeping also has consequences for the readability of the program. In DIP*lib* all allocation routines support a scheme that registers each allocation. Because all allocations are registered, they may be freed using a single call. This registration scheme will be referred to as resource management.

Resource management is very easy to use. First a `dip_Resources` structure is allocated using the `dip_ResourcesNew` function. Other allocation functions, such as `dip_ImageNew` (see chapter 9) accept a `dip_Resources` structure as a parameter. The `dip_ResourcesFree` function can be used to free a `dip_Resources` structure and all resources that were registered in it. A simple example:

```
dip_Resources resources;
dip_Image image, anotherImage;
dip_int *iptr;
void *vptr;

dip_ResourcesNew( &resources, 0 );
dip_ImageNew( &image, resources );
dip_ImageNew( &anotherImage, resources );
dip_MemoryNew( &vptr, 5000 * sizeof( dip_int ), resources );
iptr = vptr;

/* more code here */

/* Frees the dip_Resources structure, the images and the memory
   pointed to by iptr */
dip_ResourcesFree( &resources, 0 );
```

The code above is equivalent to:

```
dip_Image image, anotherImage;
dip_int *iptr;
void *vptr;

image = 0;
anotherImage = 0;
iptr = 0;

dip_ImageNew( &image, 0 );
dip_ImageNew( &anotherImage, 0 );
dip_MemoryNew( &vptr, 5000 * sizeof( dip_int ), 0 );
iptr = vptr;

/* more code here */

/* Frees the dip_Resources structure, the images and the memory
   pointed to by iptr */
dip_ImageFree( &image );
dip_ImageFree( &anotherImage );
dip_MemoryFree( iptr );
```

The order in which resources are freed using `dip_ResourcesFree` is unspecified. The order may even change between different releases of DIP*lib*.

The majority of the DIP*lib* functions starts by allocating a resources structure at the start and freeing it at the end of the routine. We have therefore made variants of the `DIP_FN` error macros that automate this process. `DIP_FNR_DECLARE` is identical to `DIP_FN_DECLARE`, but also declares a resources structure. The name of the resources structure is "rg". It should be followed by your own declarations. The resources structure is initialised by invoking the `DIP_FNR_INITIALISE` macro at the start of your code. Finally, the routine should exit using the `DIP_FNR_EXIT` macro. This macro first frees the resources associated with "rg", and subsequently performs the same tasks as `DIP_FN_EXIT`. The layout of a typical DIP*lib* routine using the resource management scheme is given below:

```
dip_Error
dip_MyFunction( void )
{
   DIP_FNR_DECLARE("dip_MyFunction");
   dip_Image image;

   DIP_FNR_INITIALISE;

   /* Allocate image and register it on "rg" */
   DIPXJ( dip_ImageNew( &image, rg ));

dip_error:
   /* Frees the dip_Resources structure and the image */
   DIP_FNR_EXIT;
}
```

5.2 Cleanup operations

The resource management scheme is not only used to keep track of allocations. As of version 2.0.0 of DIP*lib* it has also become the standard scheme used to invoke cleanup operations. Consider for example the `dip_ImagesSeparate` (see chapter 9) function that takes care of situations where an image is used as both input and output. `dip_ImagesSeparate` sets up things in such a way that consequent code can act as if the input and output images are distinct images. After the algorithm is finished, `dip_ImagesSeparate` needs to perform some cleanup operations. Before version 2.0.0 this would be done by calling a function `dip_CleanupImagesSeparate`. In newer versions `dip_ImagesSeparate` takes an `dip_Resources` parameter. It registers itself in the resources structure, causing its corresponding cleanup function to be called as soon as `dip_ResourcesFree` is used on the resources structure. The following is a schematic example of the scheme:

```
dip_ImagesSeparate( ..., resources );

/* Other code */

/* Clean up code for dip_ImagesSeparate is invoked through
   dip_ResourcesFree */

dip_ResourcesFree( &resources, 0 );
```

5.3 Tracking your own resources

It is possible to use the resource management scheme to keep track of your own resources. This is achieved through the `dip_ResourceSubscribe` function. With this function you can

register two things in a `dip_Resources` structure: a `void` pointer and a handler that will be called by `dip_ResourcesFree` when your resource is to be freed. `dip_ResourceUnsubscribe` can be used to stop tracking a particular resource. The resource itself will not be freed by `dip_ResourceUnsubscribe`. It should only be used on resources that have been registered directly by `dip_ResourceSubscribe`, not on indirectly registered resources such as images allocated by `dip_ImageNew`.

When writing a function that allocates a structure that requires the allocation of many substructures, `dip_ResourcesMerge` may come in handy. It is best demonstrated by the example given in figure 5.1:

```
dip_AllocateMyThing
(
   myThing *thing,
   dip_Resources resources
)

{
   DIP_FNR_DECLARE("dip_AllocateMyThing");

   DIP_FNR_INITIALISE;

   /*
    * Allocate your stuff and register it in "rg", the local
    * resources structure allocated and initialised by
    * the DIP_FNR macros.
    */

   /*
    * If none of the allocations failed, the following will merge
    * the local resources into the resources. It will then free the
    * local resources structure (not the resources it was tracking).
    */

   DIPXJ( dip_ResourcesMerge( resources, &rg ));
   /* Now rg == 0 again */

dip_error:
   /*
    * If anything went wrong before the call to dip_ResourcesMerge,
    * DIP_FNR_EXIT will free all resources allocated by our
    * function.
    */
   DIP_FNR_EXIT;
}
```

Figure 5.1: Using `dip_ResourcesMerge`.

Chapter 6

Memory allocation

The DIP*lib* library offers its own memory allocation functions. Use them instead of the standard malloc functions to allocate memory. In the future, the use of DIP*lib*'s own memory allocation functions could improve memory use and could simplify the portability of the library. Most importantly, the DIP*lib* allocation routines support the DIP*lib* error mechanism and resource management.

Memory can be allocated using the `dip_MemoryNew` function. It takes a pointer to a void pointer as its first argument. If the allocation is succesful the void pointer will point to the allocated memory. The second argument is the amount of bytes to allocate. The memory can be registered in a resources structure by passing it as the third parameter.

Memory can be freed using the `dip_MemoryFree` function. It takes a pointer to the memory to be freed. Usually memory is tracked using the resource management scheme, so `dip_MemoryFree` is not often used explicitly. It is allowed to pass a zero pointer to `dip_MemoryFree`, in which case it does nothing.

Finally, memory chunks can be resized using `dip_MemoryReallocate`. The arguments are the same as those for `dip_MemoryNew`, except that the pointer to a void pointer must be a valid address previously returned by `dip_MemoryNew`. If the reallocation fails, the pointer will not be overwritten.

If the developent environment provides its own memory management functions, `dip_MemoryFunctionsSet` can be used to tell DIP*lib* to use those functions instead of the default `malloc`, `realloc`, and `free`.

The following example shows how to allocate ten integers, please read the comment in the code carefully:

```
dip_int *mypointer;
void *voidpointer;

/* Allocate some memory */
DIPXJ( dip_MemoryNew( &voidpointer, 10 * sizeof( dip_int ), 0 ));
mypointer = voidpointer;

/*
 * One might be tempted to write:
 * dip_MemoryNew( &mypointer, 10 * sizeof( dip_int ), 0 ));
 * but "C" does not guarantee that different pointers have the
 * same representation. A void pointer can represent any other
 * pointer though, although a conversion may be required. In the
 * code above this conversion is done by the assignment of the
 * void pointer to the integer pointer (implicit cast).
 */

/* Free the memory */
DIPXJ( dip_MemoryFree( mypointer ));
```

Chapter 7

Arrays, structure hiding and pointers

7.1 Arrays

In many places an array is required to store some data. For example, the Gaussian filter in DIP*lib* may have a different sigma in every dimension. To pass these sigma values to the Gaussian filter an array is employed. Array's in `C` do not have an explicit size field. For example, it cannot be verified that the number of elements and the dimensionality of the image that is to be filtered match. In many other places it would be useful to know the size of the array as well.

To solve these problems, DIP*lib* defines a set of array types. One of these is the `dip_IntegerArray` type. It is defined as follows:

```
typedef struct
{
   dip_int size;
   dip_int *array;
} dip__IntegerArray, *dip_IntegerArray;
```

The basic set of these array types consists of: `dip_IntegerArray`, `dip_FloatArray`, `dip_ComplexArray`, `dip_BooleanArray` and `dip_VoidPointerArray`. We'll refer to these kind of array types as `Arrays` as opposed to normal arrays.

A `dip_IntegerArray` can be allocated using `dip_NewIntegerArray`. There are corresponding allocation routines for the other `Arrays` as well.

There are two typedefs pertaining to each `Array`. One for the actual structure itself and one for a pointer to the structure. The reason for this is explained in section 7.2. Usually `Arrays` are handles through the pointer types, such as `dip_IntegerArray`.

The `Array` type definitions are public. This means that their definition is fixed. Although using `dip_ResourcesFree` is usually much more convenient, it is also "legal" to free an `Array` by hand as follows:

```
dip_IntegerArray myArray;

/* Allocate an integer array with two elements initialised to 0 */
dip_IntegerArrayNew( &myArray, 2, 0, 0 );
/* NOT RECOMMENDED - NOT RECOMMENDED - NOT RECOMMMENDED */
dip_MemoryFree( myArray->array );
dip_MemoryFree( myArray );
```

The preferred way is of course:

```
dip_Resources resources;
dip_IntegerArray myArray;

dip_IntegerArrayNew( &myArray, 2, 0, resources ));
/* Do it this way, please? pretty please? */
dip_ResourcesFree( &resources, 0 );
```

An `Array` may have size zero. It is also allowed to set the size field to a smaller value without reallocating the array pointed to by the array element.

Using an `Array` is almost as simple as using a normal `C` array. The following code shows the content of an `Array`:

```
dip_IntegerArray myArray;
dip_int ii;

for ( ii = 0; ii < myArray->size; ii++ )
{
   printf("myArray[ %d ] = %d\n", ii, myArray->array[ ii ] );
}
```

7.2 Structure hiding and pointers

In this section we discuss some design principles that we have used in DIP*lib*. The first is that the actual definition of a complex structure, such as an image, should be hidden to the user. This allows us to do two things; first we are able to check every action that a user performs on the structure, because all access must be through access functions. Secondly it allows us to redesign the structure while its interface remains the same.

We will explain this with a simple example. We will start out with a simple structure to represent an image:

```
typedef struct
{
   dip_DataType dataType;
   dip_IntegerArray dimensions;
   void *data;
} dip__TheImage;
```

The `dip_DataType` type is explained in chapter 8. It simply states whether the pixels are represented by integers, floating point numbers etc. The dimensions `Array` contains the dimensionality and the size of each dimension. The prototype for a copy function looks like this:

```
dip_Error dip_Copy( dip__TheImage *, dip__TheImage * );
```

Our first step is to change the typedef to:

```
typedef struct
{
   dip_DataType dataType;
   dip_IntegerArray dimensions;
   void *data;
} dip__TheImage, *dip_TheImage;
```

And the prototype becomes:

```
dip_Error dip_Copy( dip_TheImage, dip_TheImage );
```

Here we have employed a design policy to hide pointer definitions in a typedef. Source code becomes a little more pleasant to read, although the programmer must remain aware of the pointer nature of these types.

Now our second objective of hiding the definition is achieved by putting the definition of `dip__TheImage` in a private include file, that is not included in the distribution and defining `dip_TheImage` as a void pointer. The result looks like this:

```
/* In a private include file */
typedef struct
{
   dip_DataType dataType;
   dip_IntegerArray dimensions;
   void *data;
} dip__TheImageInternal;

/* In the public include file */
typedef void *dip_TheImage;
dip_Error dip_Copy( dip_TheImage, dip_TheImage );
```

The definition of `dip_TheImage` as a void pointer has some undesirable consequences. When a pointer of the wrong type is passed to a function expecting a `dip_TheImage`, it will be silently converted to a void pointer by the compiler. To make sure that the compiler can check `dip_TheImage` parameters, we have to add an extra indirection to our definition:

```
/* In a private include file */
typedef struct
{
   dip_DataType dataType;
   dip_IntegerArray dimensions;
   void *data;
} dip__TheImageInternal;

/* In the public include file */
typedef struct
{
   void *internalImage;
} dip__TheImage, *dip_TheImage;

dip_Error dip_Copy( dip_TheImage, dip_TheImage );
```

This is the approach we have used with both the `dip_Resources` and `dip_Image` structure. The latter is introduced in chapter 9.

Because `dip_IntegerArray`, `dip_Resources` and `dip_Image` are in fact pointers, we can assign zero to a variable of one of these types. It is also possible to pass zero to a routine expecting a variable of one of these types. Many routines do accept zero as a valid argument, usually meaning that this argument must be ignored by the routine.

Chapter 8

Data types

8.1 Introduction

Pixel values can be represented by different types. Such types will be referred to as data types. DIP*lib* uses five sets of data types: unsigned integers, signed integers, floating point numbers, complex numbers and binary numbers. All of these come in different sizes, each able to represent a different range of values. The complete set of data types is given in table 8.1.

There are also generic data types: `dip_binary`, `dip_int`, `dip_float` and `dip_complex`. These can be used when the exact range doesn't matter (non pixel data).

The complex and binary data types are discussed in the following two sections.

8.1.1 The complex data types

The complex data types are defined by the following two structures:

```
typedef struct                          typedef
{                                       {
   dip_sfloat re;                          dip_dfloat re;
   dip_sfloat im;                          dip_dfloat im;
} dip_scomplex;                         } dip_dcomplex;
```

There are no integer based complex data types.

8.1.2 The binary data types

Binary data is represented by a single bit in one of the unsigned integer data types. One such integer may be used to store multiple binary values. The data types `dip_bin8`, `dip_bin16` and `dip_bin32` are equivalent to `dip_uint8`, `dip_uint16` and `dip_uint32`. Using explicit typedef's for the binary data types has the advantage that it is immediately clear in which fashion an integer is used: as a container for binary values or simply as an integer. Because multiple binary values may be stored in a single integer, care must be taken not to change any of the other bits. The following example shows how to clear and set bit 2 in an integer (the rightmost bit is bit 0) as well as how to read it:

integer			float	complex
binary	unsigned	signed		
dip_bin8	dip_uint8	dip_sint8	dip_sfloat	dip_scomplex
dip_bin16	dip_uint16	dip_sint16	dip_dfloat	dip_dcomplex
dip_bin32	dip_uint32	dip_sint32		

Figure 8.1: DIP*lib* data types.

```
/*
 * dip_bin8 is identical to dip_uint8, but shows our intent to use it
 * as a container for binary values.
 */

dip_bin8 binaryData;
dip_int  bit2;

/* Set bit 2, leaving the others as they are */
binaryData |= 1 << 2;

/* Clear bit 2, leaving the others as they are */
binaryData &= ~( 1 << 2 );

/*
 * read bit 2 from binaryData and set the variable bit2 accordingly
 */
bit2 = ( binaryData & ( 1 << 2 )) >> 2;
```

8.2 Dynamic versus static data types

Images, as discussed in chapter 9, store image data in the data type indicated by a field in the image structure. The data type is not fixed and can be changed by applying various operations to the image. This dynamic use of types is alien to the `C` language, and must therefore be simulated. To the user an image seems an entity with a dynamic type, while in reality each image processing routine will call a different low-level function for each specific data type it supports. Data types used in a dynamic fashion are represented by a `dip_DataType` flag. Table 8.2 lists all data type flags, as well as the corresponding data types.

The remaining sections of this chapter will deal with the following three topics:

- Getting information about a data type from a `dip_DataType` flag.
- Calling different low-level type specific routines based on a `dip_DataType` flag.
- Compiling code for several data types.

data type	dip_DataType	bitwise flag	suffix
dip_bin8	DIP_DT_BIN8	DIP_DTID_BIN8	_b8
dip_bin16	DIP_DT_BIN16	DIP_DTID_BIN16	_b16
dip_bin32	DIP_DT_BIN32	DIP_DTID_BIN32	_b32
dip_uint8	DIP_DT_UINT8	DIP_DTID_UINT8	_u8
dip_uint16	DIP_DT_UINT16	DIP_DTID_UINT16	_u16
dip_uint32	DIP_DT_UINT32	DIP_DTID_UINT32	_u32
dip_sint8	DIP_DT_SINT8	DIP_DTID_SINT8	_s8
dip_sint16	DIP_DT_SINT16	DIP_DTID_SINT16	_s16
dip_sint32	DIP_DT_SINT32	DIP_DTID_SINT32	_s32
dip_sfloat	DIP_DT_SFLOAT	DIP_DTID_SFLOAT	_sfl
dip_dfloat	DIP_DT_DFLOAT	DIP_DTID_DFLOAT	_dfl
dip_scomplex	DIP_DT_SCOMPLEX	DIP_DTID_SCOMPLEX	_scx
dip_dcomplex	DIP_DT_DCOMPLEX	DIP_DTID_DCOMPLEX	_dcx

Figure 8.2: Data types, their corresponding dip_DataType flags, bitwise flags, and suffixes.

8.2.1 Data type information from dip_DataType

The dip_DataTypeGetInfo can be used to obtain information about a data type from a dip_DataType flag. Some examples of the functionality provided by this function are: sizeof(dip_DataType), finding the floating point type corresponding to a complex type (i.e. dip_sfloat corresponds to dip_scomplex) and determining whether the data type is an unsigned integer.

8.2.2 Overloading

The term overloading is used to describe the scheme that is used by DIP*lib* to call type specific routines from a type independent routine. Which routine is called is determined by a dip_DataType. There are actually two overloading schemes. One more closely resembles a function call, while the other is more flexible. For both schemes the user has to provide a base name for the function to be called. The overload scheme attaches a type dependent suffix to this base name, as given by table 8.2, and calls the corresponding function.

Overloading scheme #1

The first scheme defines a macro called DIP_OVERLOAD_FUNC which allows you to invoke a type-dependent function almost like an ordinary function. The following steps must be undertaken to use the macro:

- At the top of your code, do the following:
  `#define DIP_OVL_ALLOW` <list of allowed data types>
  `#include "dip_overload.h"`
- At the spot where the type specific code should be called:
  `DIP_OVERLOAD_FUNC( <base name of the function> ( <argument list> ), <data type> )`

The data types for which a function is available are specified by defining DIP_OVL_ALLOW. This define must be followed by a bitwise OR of the flags specified in table 8.2. Besides these flags,

flag	data types
DIP_DTGID_UINT	uint
DIP_DTGID_UNSIGNED	uint
DIP_DTGID_SINT	sint
DIP_DTGID_INT(EGER)	uint, sint
DIP_DTGID_FLOAT	float
DIP_DTGID_REAL	uint, sint, float
DIP_DTGID_COMPLEX	complex
DIP_DTGID_SIGNED	sint, float, complex
DIP_DTGID_BINARY	binary
DIP_DTGID_ALL	all

Figure 8.3: Data type group flags.

which each specify a single data type, there is also a set of flags that specify entire groups of data types. These flags are given in table 8.3. If DIP_OVL_ALLOW is not defined, all data types will be overloaded.

Since DIP_OVL_ALLOW is defined at the start of the source file, DIP_OVERLOAD_FUNC can not be used for calling type specific functions that are available for different sets of data types. This is possible with the second scheme though.

Overloading scheme #2

The second scheme also uses the C preprocessor to do the overloading. To perform the function call to the type specific routine, the following recipe must be inserted into the source code at the place the function call is supposed to be executed:

- `#define DIP_OVL_FUNC` <base name of the function>
- `#define DIP_OVL_ARGS` <argument list>
- `#define DIP_OVL_ALLOW` <list of allowed data types>
- `#include "dip_ovl.h"`

The data type is assumed to be in a variable called `ovlDataType`. If DIP_OVL_ALLOW is not defined, all data types are overloaded. The following example shows how a function `dip_Filter` calls the appropriate low-level routine for the two data types it supports, DIP_DT_SFLOAT and DIP_DT_DFLOAT.

```
dip_Filter
(
   dip_Image in,
   dip_Image out
)

{
   dip_DataType ovlDataType;

   /*
    * Some code, which we assume initialises ovlDataType, as well
    * as the two data pointers inData and outData.
    */

   #define DIP_OVL_FUNC dip_Filter
   #define DIP_OVL_ARGS ( inData, outData )
   #define DIP_OVL_ALLOW DIP_DTGID_FLOAT
   #include "dip_ovl.h"

   /*
    * if ovlDataType = DIP_DT_SFLOAT the code above will execute:
    * DIPXJ( dip_Filter_sfl( inData, outData ));
    * if ovlDataType = DIP_DT_DFLOAT the code above will execute:
    * DIPXJ( dip_Filter_dfl( inData, outData ));
    * for all other data types :
    * DIPSJ( dip_errorDataTypeNotSupported );
    */
}
```

In addition to `DIP_OVL_ARGS` it is also possible to set `DIP_OVL_BINARY_ARGS`. If defined, it will be used instead of `DIP_OVL_ARGS` for the binary data types. Often the parameter list for binary data types includes extra parameters, such as plane numbers.

The default action undertaken by `dip_ovl.h` is to call a type specific routine using `DIPXJ()`. Sometimes it is necessary to set a function pointer to a type specific routine. This can be accomplished by defining `DIP_OVL_ASSIGN` as "`<filter pointer> =`". Again the binary data types can be treated separately using `DIP_OVL_BINARY_ASSIGN`.

8.2.3 Type iterators

In the example of the previous section, two functions (`dip_Filter_sfl` and `dip_Filter_dfl`) perform the same task, but for different data types. The code for the two functions is very likely equivalent, except for the data type. Writing the same code for different data types is tedious and error prone. Maintaining the code is also very difficult, because the different copies of the code have to be kept up to date with respect to each other.

In cases such as these DIP*lib* uses an `#include` file which is used to iterate over a selected set of data types and includes a user specified file which contains code that must be compiled

for the set of data types. The following recipe shows how to use the scheme:

- `#define DIP_TPI_FILE` <name of file containing type specific code>
- `#define DIP_TPI_ALLOW` <list of allowed data types>
- `#include "dip_tpi.h"`

The include file iterates over all speficied data types and during each iteration assigns the current data type to `DIP_TPI`. In the type specific code `DIP_TPI` must be used to refer to the current data type. Besides `DIP_TPI` a number of other symbols are also defined by `dip_tpi.h`. For example `DIP_TPI_DATA_TYPE` is the `dip_DataType` corresponding to `DIP_TPI`. The other symbols are explained in the reference manual.

The most elegant way to use this scheme is by putting the type independent code and the type specific code in one file and letting this file `#include` itself using `dip_tpi.h`. This works as demonstrated by the piece of pseudo code shown in figure 8.4.

The scheme can also be used in include files to generate prototypes for type specific functions.

```
filter.c:
/* When the compiler starts processing this program, DIP_TPI will be
 * undefined and process the code directly following the next
 * #ifndef... */
#ifndef DIP_TPI

/* Type independent code */
#include "diplib.h"

/* Start of type specific code */
#define DIP_TPI_ALLOW DIP_DTGID_FLOAT
#define DIP_TPI_FILE "filter.c"
#include "dip_tpi.h"
*/ End of type specific code */

dip_Filter ( dip_Image in, dip_Image out )
{
   /* Call the appropriate function dip_Filter_sfl or dip_Filter_dfl
    * using one of the overload schemes. */
}

/* This is where the type specific code is stored. The compiler will
 * reach this code only through including "dip_tpi.h". It will be
 * included for each realization of DIP_TPI, in this case both
 * dip_sfloat and dip_dfloat. */
#else

/* DIP_TPI_DEFINE attaches the proper suffix to dip_Filter depending
 * on the current contents of DIP_TPI. */
DIP_TPI_DEFINE(dip_Filter) ( void *inData, void *outData )
{
   DIP_TPI *in, *out;

   in  = inData;
   out = outData;
   /* Execute algorithm */
}

/* End of storage place for type specific code */
#endif
```

Figure 8.4: Example of self including code for multiple data types.

Chapter 9

Image infrastructure

9.1 The dip_Image structure

The most important structure in the DIP*lib* library is the `dip_Image` structure. This structure is used to store all the necessary information to represent an image. In this chapter we describe the `dip_Image` structure and the functions used to manipulate it.

There are images of different types, such as scalar and color images. The kind of information that is stored in a `dip_Image` will vary with the image type. There are a few fields that are always present in the `dip_Image` structure. These fields fully describe the only currently supported image type: scalar images. The image type is represented by a field in the `dip_Image` structure of the `dip_ImageType` type. Depending on the contents of this field, the other fields in an `dip_Image` may or may not have a meaning. The possible `dip_ImageType`'s are given in table 9.1.

The most important fields of a `dip_Image` are given in table 9.2. There are more fields, but these are either for internal use only or for very specific uses. These will be discussed on a "need to know" basis in the appropriate sections. `dip_Image` fields may only be accessed using a set of access functions.

The pixel values are stored in the data type indicated by the data type field. See table 8.2 for a list of the possible values and the corresponding types.

The dimensions of the image are stored in an Array. The dimensionality of an image is zero or higher. Scalar images with dimensionality zero are used to represent scalar values in DIP*lib*.

The data field is used to store a pointer to a block of memory where the pixel data is stored. The pointer points to the pixel at the origin of the image. The address of an arbitrary pixel in an D-dimensional image at the coordinate specified by the array `cor`[] with D elements, is given by:

$$address = origin + \sum_{i=0}^{D-1} \mathrm{cor}[i] * \mathrm{stride}{\rightarrow}\mathrm{array}[i]$$

Where origin is the address of the pixel at the origin of the image and `stride`[] is an Array stored in the `dip_Image` structure. For each dimension it holds the interleave between two neighbouring pixels in memory.

For binary images the plane field holds the number of the bit in which the binary data is stored.

The data pointer, plane and stride fields are all volatile. They can be changed by most

DIP_IMTP_SCALAR	Scalar images

Figure 9.1: The dip_ImageType's.

field type	short description	access function
dip_ImageType	The image type	dip_ImageGetType
dip_ImageState	The image state	-
dip_DataType	Data type used to store pixel values	dip_ImageGetDataType
dip_IntegerArray	Dimensions of the image	dip_ImageGetDimensions
void *	Pointer to the pixel data	dip_ImageGetData
dip_int	Plane number, for binary images	dip_ImageGetPlane
dip_IntegerArray	Stride array. See text	dip_ImageGetStride

Figure 9.2: The dip_Image fields and their access functions.

functions. It is only safe to use the information in these fields during the time you access the pixel data of the image. See section 9.3.5 on how to safely access the pixel data.

9.2 Allocating and de-allocating images

A new image can be allocated using the dip_ImageNew function. The fields of the newly allocated dip_Image are initialised to some default values. No image data is allocated. The fields of this image must now be set to their desired values using the following set of functions: dip_ImageSetType, dip_ImageSetDataType, and dip_ImageSetDimensions. Another useful way of initialising these fields is by using the dip_ImageCopyProperties function. This function copies all the fields from an existing image to the target image. The functions described above can then be used to override some of the fields.

When the fields are properly initialised, a data block to store the image data may be allocated by using the dip_ImageForge function. The following piece of code shows how to allocate a two-dimensional scalar image with dimensions (156, 111) and data type DIP_DT_SFLOAT:

```
dip_Image image;
dip_IntegerArray dimensions;

dip_IntegerArrayNew( &dimensions, 2, 0, 0 );
dimensions->array[ 0 ] = 156;
dimensions->array[ 1 ] = 111;
dip_ImageNew( &image, 0 );
dip_ImageSetType( image, DIP_IMTP_SCALAR );
dip_ImageSetDataType( image, DIP_DT_SFLOAT );
dip_ImageSetDimensions( image, dimensions );
dip_ImageForge( image );
```

An image is said to be *raw* before the call to dip_ImageForge and *forged* afterwards. The

`dip_ImageSet` functions can only be used on a raw image. While this scheme may seem complex, it is very flexible. It will also simplify the integration of any future image types. There are also simpler ways to allocate an image, see section 9.4.

The `dip_ImageStrip` function deallocates the image data if present and resets all image fields to their initial value, thus returning the image to its raw state. `dip_ImageFree` first calls `dip_ImageStrip` and then frees the `dip_Image` structure itself. It is almost never necessary to call `dip_ImageFree` directly because of the resource tracking scheme.

9.3 Writing image processing operations

9.3.1 The general structure

All DIP*lib* image processing routines have the same general structure. If you write your own routine using DIP*lib*, it will have to obey the same structure. This structure is as follows:

- Check to see if the input images have the type and size (and any other properties) that you support. For instance, you may only support floating point scalar images. Return an error if the input images do not have the properties you require. Raw input images make no sense, and this should also be detected.
- Users are allowed to call your functions as if they are able to operate in-place. This means that some of the input images may also be specified as output images. Most low-level code will overwrite its own input in such a case, so it must be explicitly dealt with.
- The output images must be adjusted so that they will be of the proper type and size as required by the routine. For output images raw images do make sense and should be supported.
- Get pointers to the pixel data and execute your algorithm.
- Clean up. Resource tracking (`dip_ResourcesFree`) will probably take care of this.

Not all image processing operations will access the pixel data directly. Such functions, that merely call existing image processing routines, can delegate much of the work described above to these existing routines. This is often not true for the in-place problem, so make sure that you explicitly deal with this.

The following sections will deal with each of the items on the list above in some more detail.

9.3.2 Checking the input

This is an easy task, although it can be tedious. There are a number of functions that simplify this task however:

```
dip_ImagesCompare(), dip_ImagesCompareTwo(),
dip_ImageCheck(),
dip_ImagesCheck(), dip_ImagesCheckTwo(),
dip_IsScalar(),
dip_DataTypeAllowed(),
```

Operation	step 1	step 2	step 3
Gauss(A → A)	New(TMP)	Gauss(A → TMP)	Replace(TMP → A)

Figure 9.3: The operations performed by `dip_ImagesSeparate`.

As an example we show how to check whether two images have the same size (dimensionality and individual dimensions):

```
/*
 * The last parameter is 0. This will cause dip_ImagesCompareTwo
 * to return an error when the images do not have the same size.
 */
dip_ImagesCompareTwo( image1, image2, DIP_CPIM_SIZE_MATCH, 0 );
```

We refer to the reference manual for the description of these functions.

9.3.3 Dealing with in-place operations

The scheme that deals with in-place operations is quite simple and elegant. The function `dip_ImagesSeparate` accepts an array of input images and another one of output images. It returns an array with output images that you should use instead of the original output images. `dip_ImagesSeparate` operates in four steps:

- First it examines the arrays of input and output images to see if any of the input images are also used as output images. Any output image that is also an input image is marked. If an image is specified as an output more than once, an error is returned.
- Now it returns the array containing the output images that the user must use from now on. For unmarked output images `dip_ImagesSeparate` simply returns the existing output image. In the case of a marked output image a new raw image is created with `dip_ImageNew` followed by a `dip_ImageCopyProperties` from the old to the newly allocated output image. An entry is inserted into the resource tracking structure that indicates that the output image has been replaced by a new one. The new (raw) image is returned to the user. When this step is finished, `dip_ImagesSeparate` returns.
- This step is not performed by `dip_ImagesSeparate`, but by the user. At this point you perform the adjustment of the output images and execute your algorithm. See sections 9.3.4 and 9.3.5.
- This step is invoked by `dip_ResourcesFree`. The output images that were allocated in step 2 now replace the original output images.

Figure 9.3 shows the data flow for a simple one input one output operation.

Some functions convert their input image to a temporary image, to support image types that are not directly supported. Consider a Gaussian filter that operates only on floating point images. It is very annoying that this filter does not accept an integer image for its input. The solution is to convert the input image to a temporary floating point image. If the input image has been copied to a temporary image, it is no longer necessary to create a temporary output image, since the input data has already been safely stored. Therefore `dip_ImagesSeparate`

accepts an array of flags (one for each input image) with which you can indicate that the input data was copied to a safe place. This prevents unnecessary allocation of temporary output images.

9.3.4 Preparing the output images

Usually an image processing operation will return some results in an output image. The output image often must be of some predescribed type and size, which may depend on other factors. Functions should accept both raw and forged output images. The basic sequence of steps that should be undertaken is as follows:

```
   if "out is forged"
   {
      dip_ImageStrip( out );
   }
   Set properties of out to what they are supposed to be;
   dip_ImageForge( out );
}
```

This sequence is often performed by the `dip_ImageAssimilate` function. It performs the sequence described above, and sets the properties of the output image by using `dip_ImageCopyProperties` and a second image used as an example. The way to use `dip_ImageAssimilate` is thus: set up a dummy image with the desired properties; call `dip_ImageAssimilate` with the dummy image as the example and the output image as its output.

Consider an image processing operation that requires its output to be of the same type and size as its input image. The following code shows how to achieve this:

```
dip_MyFunction( in, out )
{
   dip_ImageAssimilate( in, out );   /* Now out has inherited all
                                        properties from in... */
   /* the rest of your code */
}
```

If the output should have the same size as the input and should always have data type `DIP_DT_SFLOAT`, this can be achieved by the following code:

```
dip_MyFunction( in, out )
{
   dip_Image dummy;

   dip_ImageNew( &dummy, 0 );
   dip_ImageCopyProperties( in, dummy );
   dip_ImageSetDataType( dummy, DIP_DT_SFLOAT );
   dip_ImageAssimilate( dummy, out );

   /* the rest of your code */
}
```

It is also possible to use `dip_ImageAssimilate` to allocate a temporary image that has the same properties as some existing image:

```
/* image is an existing image... */
dip_Image tmpImage;

dip_ImageNew( &tmpImage, 0 );
dip_ImageAssimilate( image, tmpImage );
```

The function `dip_ImageClone` is merely short hand for these two calls. Note that the image data is not copied. If this is desired, substitute `dip_Copy` for `dip_ImageAssimilate` (`dip_Copy` will call `dip_ImageAssimilate` and then copy the data).

9.3.5 Accessing pixel data

When all preparations for your algorithm have been completed, the function `dip_ImageGetData` can be used to obtain pointers to the pixel data of each image. No other image processing functions should be called after the pointers have been obtained, because these can possibly alter the pointers. Only after you have finished using the pointers, it is safe to use other operations again. The plane and stride fields of an image should be requested after the call to `dip_ImageGetData` to ensure that they are up to date when the pixel data is accessed.

`dip_ImageGetData` has a number of parameters that are currently not used, but that are reserved for future extension. It makes a distinction between input and output images for the same reason. For both types of images (the pointers obtained from input images may only be used for reading data) an array of images is given as input, and an Array of data pointers is returned. Associated with each array of images is an array of flags that is currently not used. There is also a global flag parameter that is also unused. Finally there is a resource tracking parameter that can be used for any clean up operations that a future extension may require. The `dip_ResourcesFree` call associated with these resources should come right after you have finished using the data pointers.

9.4 Convenience functions

The previous sections described the basic tools necessary to build an image processing routine using the DIP*lib* library. Certain sequences of function calls will occur frequently and it is useful to have a set of convenience functions that take care of these recurring tasks.

The first of these is `dip_ScalarImageNew` which allocates a `DIP_IMTP_SCALAR` image and accepts data type and dimensions parameters.

Another common operation is that of changing the data type of an image. We have already shown how this can be achieved using `dip_ImageAssimilate`, but it is easier to use `dip_ChangeDataType` instead. The output inherits all properties from the input images, except the data type, which is explicitly specified using a parameter. `dip_ChangeTo0d` is a variant of `dip_ChangeDataType`, which performs the same tasks and sets the dimensionality of the output image to zero.

Chapter 10

Boundary conditions

One of the design features of the DIP*lib* library is to standardise the way filters deal with the borders of an image. This is done by defining a set of boundary conditions describing how an image should be extended beyond the borders of that image.

Most of the filters that are supplied by the DIP*lib* library accept an array of boundary conditions. This array specifies what the boundary condition is for each dimension of the image that has to be filtered. These functions also accept `NULL` for this parameter, causing the default boundary condition to be used. This default is set to `DIP_BC_SYM_MIRROR` by `dip_Initialise()`, and can be changed through `dip_GlobalBoundaryConditionSet()`. It is possible to specify a different boundary condition for each image dimension.

The boundary conditions supported by the DIP*lib* library are specified in the `dip_Boundary` enumeration type (defined in `dip_support.h`). The current inplemetation of the library supports the boundary conditions specified in table 10.1 (see 10.2)

Please note that using mirroring as implemented by DIP*lib*, the border pixels are duplicated. Thus, if an image `123` is extended by mirroring, it will become `123321` and not `12321`.

If a filter can not support a certain boundary condition, it should return the `DIP_E_BOUNDARY_CONDITION_NOT_SUPPORTED` error code.

The DIP*lib* library supplies two functions to facilitate the processing of the boundary conditions: The function `dip_FillBoundary()` extends a dip_Image according to the boundary condition. The function `dip_FillBoundaryArray()` extends a one dimensional array.

The FrameWork functions (see chapter 11) process the boundary conditions for the filters that use one of these FrameWork functions. Therefore these filters do not have to handle the boundary conditions themselves, but only have to pass on the boundary conditions to one of the frameworks.

Name	Description
DIP_BC_SYM_MIRROR	Symmetric mirroring
DIP_BC_ASYM_MIRROR	Asymmetric mirroring
DIP_BC_PERIODIC	Periodic copying
DIP_BC_ASYM_PERIODIC	Asymmetric periodic copying
DIP_BC_ADD_ZEROS	Extending the image with zeros
DIP_BC_ADD_MAX_VALUE	Extending the image with + infinity
DIP_BC_ADD_MIN_VALUE	Extending the image with − infinity

Figure 10.1: Supported boundary conditions.

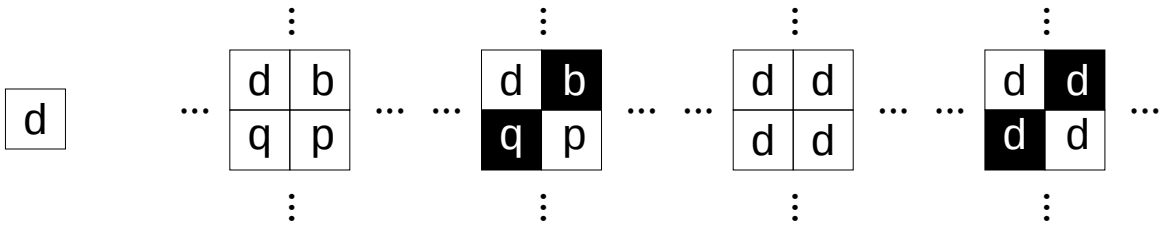

Figure 10.2: Illustration of the main boundary extensions. From left to right: the original image, symmetric mirror, asymmetric mirror, periodic, and asymmetric periodic. The images with a black background represent an image multiplied by -1.

Chapter 11

Frameworks

11.1 Introduction

To quicken the development and the execution speed of image processing filters, the DIP*lib* library supplies several framework functions. The following frameworks are available in the current release,

- `dip_SeparableFrameWork`
- `dip_MonadicFrameWork`
- `dip_SingleOutputFrameWork`
- `dip_PixelTableFrameWork`
- `dip_ScanFrameWork`

The following sections describe the funtionality and use of each of these frameworks. For a complete explaination of their function arguments and behaviour, we refer to the reference manual.

11.2 The separable filter framework

11.2.1 Introduction

The `dip_SeparableFrameWork` function provides a framework for separable filters. Separable filters are filters whose n-D filter operation can be separated in n consecutive one dimensional filter operations. This results in a considerable speed up.

Normally one has to write, besides the actual one filter operation, also code for processing the image in all its dimensions and for handling the boundary condition of each dimension. The `dip_SeparableFrameWork` framework takes care of these last two things leaving the user with the responsability to provide the one dimensional filter function only. This easens the creation of such a filter dramatically. Furthermore, the `dip_SeparableFrameWork` will process the image in a way that is optimized for cache performance, speeding up the processing time considerably.

As mentioned above, the `dip_SeparableFrameWork` takes care of a lot of "householding", i.e. it checks the validity of the input image and adjusts the size, dimensionality or datatype of the output image to the desired type. After this initialisation, it starts filtering the input. This done by copying, in a sequential order, all corresponding lines from the input and output images to 1-D arrays. Then it calls a user-supplied function that filters the input array to the output array such that the desired 1-D filter operation of the n-D separable filter is performed.

After processing all lines in one dimensions, `dip_SeparableFrameWork` will repeat this for all the dimensions in the image.

Besides copying the images lines to 1-D arrays, the `dip_SeparableFrameWork` also extends the arrays by adding extra pixels to both sides of the line. This done to facilitate the processing of the borders of the image. By enlarging the arrays on both sides with an user specified number of pixels, the user-supplied filter function can savely processing all the image line pixels without the need to worry about the edges. The values of the pixels added to the arrays is determined by a user-supplied array of boundary conditions.

11.2.2 Usage

The definition of the `dip_SeparableFrameWork` function is:

```
dip_Error dip_SeparableFrameWork ( dip_Image, dip_Image,
   dip_BoundaryArray, dip_IntegerArray, dip_FrameWorkProcessArray );
```

The first argument is the input image, the second the output image. The third argument is an array (its size equal to dimensionality of the input image) specifying the boundary condition of the image in each dimension. This argument can be set to zero, in which can the global default boundary conditions are used. The fourth argument specifies the border extension, i.e. how many pixels `dip_SeparableFrameWork` should add to the input and output array. (again this array has a siz equal to the dimensionality of the input image). If the input and output arrays do not require extension, this argument can be set to zero. The final argument is an array of `dip_FrameWorkProcess` structures.

The definition of the user-supplied 1-D filter function is:

```
dip_Error (*dip_SeparableFilter) (void *, void *, dip_int,
   dip_SeparableFilterParameters );
```

The first two arguments are pointers to the input and output arrays. These arguments are followed by the size (in number of pixels) of these arrays. The final argument is a structure containing additional information about the input and output arrays. This structure will be discussed later. We name this filter function the SeparableFilterFunction from now on.

`dip_SeparableFrameWork` creates this array of pixels by copying it from the input image and extendeding it with a border. The size of this border is specified by the function that calls `dip_SeparableFrameWork`. `dip_SeparableFrameWork` will give the filterfunction a pointer to the first pixel of the line of the input to be processed, allowing it to access pixels on both sides of the line. Therefore no special border processing code needs to be written for the filterfunction, reducing coding time and code complexity.

Having created a filterfunction, or different filterfunctions for processing some dimensions of the image in a different manner, one needs to create an array of `dip_FrameWorkProcess` structures. with the number of structures determines the number of times `dip_SeparableFrameWork` has to process the image (this number can be less, equal or larger than the dimensionality of the image). If just one process structure is provided and

the number of structures is set to zero, that structure is used to process all dimensions of the image.

11.3 The monadic and single output frameworks

These two frameworks are very similar to the separable framework. The monadic framework is merely a frontend to the `dip_SeparableFrameWork` function to provide a simplified function interface for operations that only need to scan the image. (the dimension in which the image is scanned can be specified or left to the `dip_MonadicFrameWork` function). This framework is primarily intended for creating point operations (like `dip_Clip`). The `dip_SingleOutputFrameWork` is very similar to the monadic framework. However, it only scans an output image. This framework is intended for functions that create or generate images (like `dip_FTEllipsoid`).

11.4 The scan framework

This framework is an extension of the monadic framework in the sense that it provides the possiblity to scan (in one dimension) n input and m output images (with n and $m >= 0$).

11.5 The pixel table framework

This framework is intended for image processing filters that filter the image with an arbitrary filtershape. By the coding the shape with a pixel table (runlength encoding), this framework will provide the filterfunction a line of pixels from the image it has to filter. The filterfunction is allowed to access pixels within a box around each pixel. The size of this box is specified by the function that calls the framework. The dimensionality of the box is equal to the image dimensionality. For efficiency reasons, the framework will convert the pixel table to an array of pixel position offsets and an array of runlength which are provided to the filterfuntion.

11.6 Framework convenience functions

Although the frameworks remove much of the burden of writing an image processing filter, some convenience functions are provided by the DIP*lib* library that make this creation almost enjoyable. The `dip_SeparableConvolution` function provides a high level interface for separable convolution filters. It only needs an array of filter elements and some flags for quidance. The `dip_MonadicPoint, dip_MonadicPointData and dip_SingleOutputPoint` functions only need a function that converts a single input function to a single output value. Several functions from the ALU library were created using these convenience functions (like `dip_Sin and dip_BesselJ0`).

Example:

```
#include "diplib.h"
#include "dip_framework.h"

dip_Error dip_Uniform3x3
(
   dip_Image     in,
   dip_Image     out,
   dip_BoundaryArray  boundary
)
{
   DIP_FNR_DECLARE("dip_Uniform3x3");
   dip_int              ii, dim;
   dip_IntegerArray     border;
   dip_FrameWorkProcess process;

   DIP_FNR_INITIALISE;

   /* allocate the border array */
   DIPXJ( dip_ImageGetDimensionality( in, &dim ));
   DIPXJ( dip_IntegerArrayNew( &border, dim, 1, rg ));

   /* fill the process array */
   process.process               = DIP_PROCESS_DO;
   process.frameWorkMethod       = DIP_FRAMEWORK_DEFAULT_METHOD;
   process.frameWorkOperation    = DIP_FRAMEWORK_DEFAULT_OPERATION |
                                   DIP_FRAMEWORK_USE_BUFFER_TYPES |
                                   DIP_FRAMEWORK_NO_BUFFER_STRIDE;
   process.inputBufferType       = DIP_DT_FLOAT;
   process.outputBufferType      = DIP_DT_FLOAT;
   process.frameWorkFunctionType = DIP_FRAMEWORK_SEPARABLE_FILTER;
   process.functionParameters    = 0;
   process.frameWorkFilter.separableFilter = dip__Uniform3x3;

   DIPXJ( dip_SeparableFrameWork(in, out, boundary, border,
          &process, 0 ));

dip_error:
   DIP_FNR_EXIT;
}
```

```
#include "diplib.h"
#include "dip_framework.h"

dip_Error dip__Uniform3x3
(
   void *input,
   void *output,
   dip_int size,
   dip_SeparableFilterParameters params
)
{
   DIP_FN_DECLARE("dip__Uniform3x3");
   dip_int ii;
   dip_float *in, *out;

   in  = input;
   out = output;

   for( ii = 0; ii < size; ii++ )
   {
      out[ ii ] = (in[ ii - 1 ] + in[ ii ] + in[ ii + 1 ])/ 3.0;
   }

   DIP_FN_EXIT;
}
```