

# **A SIP User Manual**

for SIP version 0.3 (rev. 2)

Jocelyn DRUEL  
*jocelyn.druel1@libertysurf.fr*

May 25, 2004



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                                     | <b>5</b>  |
| <b>2</b> | <b>Starting with SIP</b>                                | <b>7</b>  |
| 2.1      | Installing SIP . . . . .                                | 7         |
| 2.2      | Looking at a demo . . . . .                             | 7         |
| 2.3      | A problem with memory . . . . .                         | 7         |
| 2.4      | Reading an image . . . . .                              | 8         |
| 2.5      | Viewing an image . . . . .                              | 8         |
| 2.6      | Writing a matrix as an image file . . . . .             | 8         |
| 2.7      | Normalising . . . . .                                   | 9         |
| 2.8      | About coordinates . . . . .                             | 9         |
| <b>3</b> | <b>Basic manipulations</b>                              | <b>11</b> |
| 3.1      | Mean of 2 images . . . . .                              | 11        |
| 3.2      | Multiplication of 2 images . . . . .                    | 11        |
| 3.3      | Addition or subtraction modulo 256 . . . . .            | 11        |
| <b>4</b> | <b>Grayscales, colors, pseudocolors and conversions</b> | <b>13</b> |
| 4.1      | Grayscale images . . . . .                              | 13        |
| 4.2      | Truecolor images . . . . .                              | 13        |
| 4.3      | Pseudocolor images . . . . .                            | 14        |
| 4.3.1    | Introduction . . . . .                                  | 14        |
| 4.3.2    | Viewing colormaps . . . . .                             | 14        |
| 4.3.3    | Pseudocolor images . . . . .                            | 15        |
| 4.3.4    | I want to see more . . . . .                            | 15        |
| 4.4      | Converting a colored image to grayscale . . . . .       | 16        |
| 4.5      | Thresholding . . . . .                                  | 16        |
| <b>5</b> | <b>Image filtering</b>                                  | <b>17</b> |
| 5.1      | Image convolution . . . . .                             | 17        |
| 5.1.1    | General usage . . . . .                                 | 17        |
| 5.1.2    | Pre-defined filters . . . . .                           | 18        |
| 5.1.3    | Examples . . . . .                                      | 19        |
| 5.2      | Other filters . . . . .                                 | 19        |

|           |  |           |
|-----------|--|-----------|
| 5.2.1     | Median filter . . . . .                                | 19        |
| 5.2.2     | Histogram equalization . . . . .                       | 19        |
| <b>6</b>  | <b>Using fft in images</b>                             | <b>21</b> |
| 6.1       | Observing the spectrum . . . . .                       | 21        |
| 6.2       | Modifying an image by acting on the spectrum . . . . . | 22        |
| <b>7</b>  | <b>Interfacing C with SIP</b>                          | <b>25</b> |
| 7.1       | Why ? . . . . .  | 25        |
| 7.2       | Warnings . . . . .                                     | 25        |
| 7.3       | How ? . . . . .  | 25        |
| 7.3.1     | Adding a real to a matrix . . . . .                    | 26        |
| 7.3.2     | Loading the code in Scilab . . . . .                   | 28        |
| 7.3.3     | A bit more difficult . . . . .                         | 29        |
| 7.3.4     | Misc. . . . .  | 31        |
| <b>8</b>  | <b>Thanks</b>  | <b>33</b> |
| <b>9</b>  | <b>References</b>                                      | <b>35</b> |
| <b>10</b> | <b>Index</b>   | <b>37</b> |

# Chapter 1

## Introduction

SIP stands for Scilab Image Processing. Its homepage is located at:

<http://siptoolbox.sourceforge.net>

This software was designed by Ricardo Fabbri <[rfabbri@if.sc.usp.br](mailto:rfabbri@if.sc.usp.br)> in order to be able to use the power (and tools) of Scilab to treat images.

Let's look at what he says about his goals:

" I have a dream that one day SIP/SciLab will be the great free prototyping environment for image processing, used and developed by people all over the world. There are, of course, many obstacles to overcome before this dream come true."

SIP is *free software* (GPL license), and as such external contributions are welcome. Several people have already contributed with other functions, documentations or simply tests and bug reports.

I was already a Scilab user when I discovered SIP. I had long searched for an easy way to treat images in various format (jpg, bmp,...). As SIP uses ImageMagick, every file format recognized by ImageMagick will be available in SIP.

This little book tries to help people start with SIP and develops some basic notions about image treatment.

*Caution:* it does *not* describe all possibilities of SIP !

It applies mainly to version 0.3 so expect some modifications if you use another one.

I'm developping with the Linux OS which I find much better than the M\$ OS (I love freedom). The file path with Linux is noted with a slash whether Windows uses backslash, so take care when copying examples from this book.

Feel free to send me comments and critics about it (you could also use the development list to do so, which is a better way to improve free software). It's always pleasant to have feedback from other people...

About the license of this document: well..., I not a license expert, so I'd like to say I wrote this manual on my spare time.

If you use some part of it, at least, don't claim it's your own work !

Author:  
Jocelyn Druel  
Lycee Gustave Eiffel  
Labo photonique  
96 rue Jules Lebleu  
59280 ARMENTIERES  
FRANCE  
mail: <jocelyn.druel1@libertysurf.fr>

## Chapter 2

# Starting with SIP

SIP uses ImageMagick to transform an image file in a matrix usable by Scilab. The file formats usable are all those of ImageMagick: JPEG, TIFF, BMP, PNG, ..., and many more.

### 2.1 Installing SIP

Two versions of SIP are available:

- a binary one which is self-containing, which means you don't have to download other packages (such as ImageMagick): it's certainly the easiest way to start !
- a source distribution that you need to compile.

To load the SIP toolbox in Scilab, please refer to the file "INSTALL.txt" which comes with your SIP version.

Shortly, once installed correctly, launch Scilab and type `exec loader.sce`.

You should have a message like "SIP - Scilab Image Processing loaded" in your Scilab window.

Ready ?

### 2.2 Looking at a demo

Launch `exec(SIPDEMO)` to have a little idea of SIP possibilities.

### 2.3 A problem with memory

As images are very big in memory occupation, you should increase the stacksize in Scilab by using

--> `stacksize(3e7)` (the number after stacksize depends on the memory available on your pc).

If you don't want to write this instruction each time you start Scilab, you can create (or edit if it exists) a file `/home/my_user_name/.scilab` which contains:

```
stacksize(3e7);
```

This file is executed at each start of Scilab.

Alternatively, to make this change system wide, you can modify (if you have sufficient permissions) the `scilab.star` file (usually located in `/usr/lib/scilab`) so that every user has the same settings.

## 2.4 Reading an image

If you don't have images available, you can use SIP ones by

```
--> chdir(SIPDIR+'images')
```

To read a grayscale or truecolor image:

```
--> mat=imread('ararauna.jpg');
```

(don't forget the `;` if you don't want to see all the content of the matrix displayed on the screen).

The matrix called "mat" now contains the image "ararauna.jpg". All tools availables in Scilab for matrix are instantly availables to treat images.

Wonderful !

If you have a pseudocolor image, you must also get the colormap:

```
--> [mat,map]=imread('ararauna.png');
```

If you don't know what are pseudocolor images, the chapter 4 "Grayscale, colors, pseudocolors and conversions" will give you more infos.

*Note:* For Grayscale and Truecolor images, `mat` values will be in the range 0-1.

For indexed (or paletted) images, `mat` values will be in the range 0-NC (NC: Number of Colors).

## 2.5 Viewing an image

```
--> xbascc();imshow(mat);
```

(`xbascc()` is to clear the current graphic window) for grayscale or truecolor images.

For a pseudocolor image:

```
--> xbascc();imshow(mat,map);
```

## 2.6 Writing a matrix as an image file

```
--> imwrite(mat,'/home/druel/test.jpg')
```

(be sure to have writing permissions on the destination repertory).

For pseudocolor images, you have to save the colormap too:

```
--> imwrite(mat,map,'/home/druel/test.jpg')
```



## 2.7 Normalising

To darken an image:

```
--> dmat=mat*0.6;  
--> xset('window',2);xasc();imshow(dmat);  
--> max(dmat) returns 0.6
```

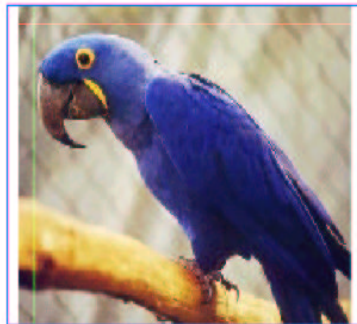
Now, if you want the image to reoccupy the full range from 0 to 1, you can use the instruction `normal`:

```
--> nmat=normal(dmat);
```

## 2.8 About coordinates

The coordinates of an image start at the left upper corner. Let's see it with an example:

```
--> image=imread(SIPDIR+'images/ararauna.jpg');  
--> [r c]=size(image); number of rows and columns  
--> image(10,:,1)=1; draws a red line on row number 10.  
--> image(:,10,2)=1; draws a green line on column number 10.
```





## Chapter 3

# Basic manipulations

### 3.1 Mean of 2 images

```
--> mat1=imread('image1.jpg');mat2=imread('image2.jpg');  
--> mat_mean=(mat1+mat2)/2;  
--> xbascc(); imshow(mat_mean);
```

### 3.2 Multiplication of 2 images

Do not forget that multiplication of 2 matrix does *not* mean multiplying each element of the 1st matrix at the place  $i, j$  by the element  $i, j$  of the 2nd matrix!

To do so, use the dot (.):

```
--> mat=imread('image.jpg');  
--> pmat=mat.*mat; calculates mat2.
```

Compare with

```
--> pmat=mat*mat;
```

### 3.3 Addition or subtraction modulo 256

When using the function `imread`, the result matrix is coded with doubles between 0 and 1.

If you have 8bit images and want to perform an addition or subtraction modulo 256 (ie  $10-250=-240=16$  modulo 256), simply do:

```
--> mat1=imread('image1.jpg');mat2=imread('image2.jpg');  
--> mat_modulo=uint8(mat1*255)-uint8(mat2*255);  
--> mat_modulo=double(mat_modulo)/255; in order to avoid future problems when trying to display mat_modulo.
```



## Chapter 4

# Grayscales, colors, pseudocolors and conversions

*Advice: pass this chapter if you're just discovering SIP and come back later to improve your knowledge (and maybe solve your problems).*

There are 3 types of images: grayscale, truecolor and pseudocolor.

### 4.1 Grayscale images

Grayscale images are 2D arrays of pixel values.

To read an image as a grayscale one, you should use the `gray_imread` function (instead of `imread`).

```
--> mat=gray\_imread('onca.gif');  
--> xbasec();imshow(mat);  
--> size(mat)
```

Usually, you can have 16 bits (values from 0 to  $2^{16} - 1 = 65535$ ) or 8 bits (values from 0 to  $2^8 - 1 = 255$ ) images.

SIP now stores these images in a 0-1 range. If you prefer to work with 0-255 or 0-65535 ranges, you just have to multiply `mat` by this factor.

### 4.2 Truecolor images

Truecolor images are made of 3 grayscale images, one for each channel: red, green and blue:

```
--> mat=imread('ararauna.jpg');  
--> xbasec();imshow(mat);
```

```
--> size(mat)
mat(:, :, 1) is the red component, mat(:, :, 2) is the green component and
mat(:, :, 3) is the blue component.
```

## 4.3 Pseudocolor images

### 4.3.1 Introduction

Again, let's do:

```
--> image=imread('onca.gif');
--> min(mat); is 1
--> max(mat); is 256
```

This time, the each pixel value points to a color defined in a colormap.

If you want to display this picture in gray, do

```
--> xbas(); imshow(mat, graycolormap(256));
```

to use a gray colormap with 256 values.

You may find that a gray display is a bit sad: a colorful life is joyfuler ! So let's decide that:

- the low values (near 1) will be displayed in dark red;
- the high values (near 256) will be displayed in light yellow;

To perform it:

```
--> map=hotcolormap(256);
--> xset('window', 1); xbas(); imshow(image, map);
```

Now, we look in details at `map`: it is a matrix with 3 columns and 256 rows. The 1st row describes the color associated with the pixel value 1, the 2nd row is for the pixel value 2, ...

The 3 columns represent the quantity of (respectively): red, green and blue associated with each pixel value (range 0-1).

We can reverse the order of the map matrix, ie we associate yellow to the pixel value 0 and red to the pixel value 255.

```
--> invmap(256:-1:1, 1:3)=map(1:1:256, 1:3); //reverse order
--> xbas(); imshow(image, invmap);
```

If you prefer blue colors, you can do:

```
--> xbas(); imshow(image, 1-hotcolormap(256));
```

Standard Scilab (version 2.7) has only 2 pre-defined colormaps: grayscale and hotcolormap. But, if you want more, install the add-on toolbox called "Enrico" (very easy: exec 'builder.sce' followed by exec 'loader.sce'). Then display an image and launch `es_demos()`. Choose different colormaps and have fun.

### 4.3.2 Viewing colormaps

Just a practical example to view these beautiful colormaps:

```
--> pseudo=ones(1:256) *(1:256);
```

```
--> xbase();imshow(pseudo,hotcolormap(256));
```

### 4.3.3 Pseudocolor images

Some images are coded as "pseudocolor images", with a complex colormap. The following instructions

```
--> image=imread('ararauna.png');
--> xbase();imshow(image,graycolormap(256));
```

does not show the expected bird in grayscale. This is because the computer associates each pixel value with a gray level, instead of associating it with a color described in the colormap.

The correct way is to get the 2D array and the colormap at the same time:

```
--> [image,map]=imread('ararauna.png');
--> xbase();imshow(image,map);
```

Display the colormap by:

```
--> pseudo=ones(1:256)';*(1:256);
--> xbase();imshow(pseudo,map);
```

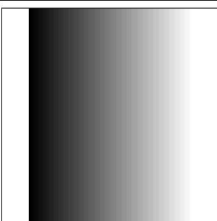
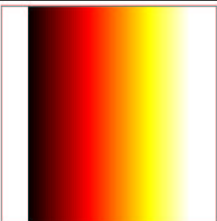
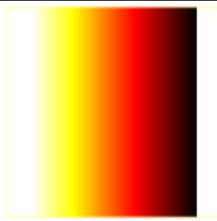
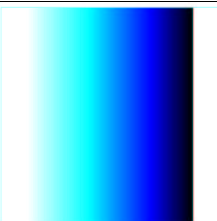
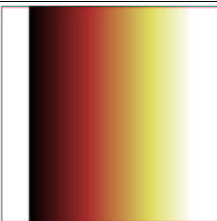

When writing this kind of picture, don't forget to save the map by:

```
--> imwrite(image,map,filename);
```

### 4.3.4 I want to see more ...

The introductory demo from Ricardo Fabbri is a visual explanation for colormaps. Launch it by `exec(SIPDEMO)`.

As colormaps are matrix, you can perform operations on them.

| Graycolormap  | Hotcolormap   | Hotcolormap (reverse)  |
|---|---|--|
|  |  |  |
| (1-hotcolormap)   | (hotcolormap(256)+graycolormap(256))/2  | colormap for ararauna.png  |
|  |  |  |

## 4.4 Converting a colored image to grayscale

```
--> mat=imread('ararauna.jpg');
--> gmat=im2gray(mat);
--> xset('window',1);xbase();imshow(gmat);
```

Or directly:

```
--> gmat=gray_imread('ararauna.jpg');
```

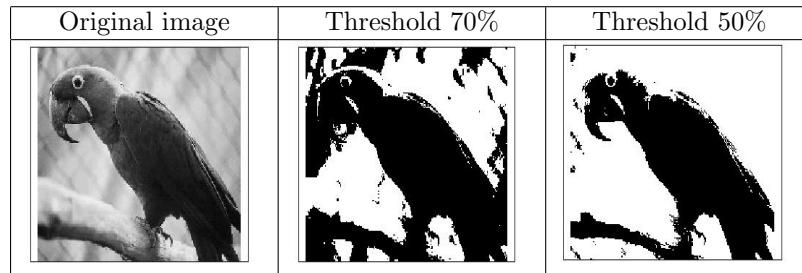
*Note:* `typeof(mat)` is 'hypermat' (color) and `typeof(gmat)` is constant (grayscale).

## 4.5 Thresholding

To threshold an image at the 70% level (the result will be rather dark):

```
--> bmat=im2bw(gmat,0.7);
--> imshow(bmat);
```

The `bmat` has only 2 values: 0 or 1.





# Chapter 5

## Image filtering

### 5.1 Image convolution

Convolution of an image by a matrix is an easy operation: the value of each pixel is modified depending of the values of its neighbours and of the coefficients of the convolution matrix (or "kernel").

Example: Consider the following pixels:

|   |   |   |
|---|---|---|
| 1 | 5 | 1 |
| 2 | 4 | 3 |
| 8 | 2 | 1 |

If you want to smooth the image (ie to have less variations), you can choose the following kernel:

|   |   |   |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

The central pixel (initial value = 4) will now have the value  $(1 \times 1) + (1 \times 5) + (1 \times 1) + (1 \times 2) + (1 \times 4) + (1 \times 3) + (1 \times 8) + (1 \times 2) + (1 \times 1) = 27$ . Of course, to make sense, you should divide by 9 (sum of the coefficients of the kernel).

The central pixel takes then the value = 3.

The operation is repeated for each pixel of the image.

Several kernels exist, depending on wheter you want to smooth the image, or show quick variations of luminosity, ...

#### 5.1.1 General usage

You've got a matrix named "mat". Let's do a mean filtering:

```
--> kern=[1 1 1;1 1 1;1 1 1]
--> mat_conv=(1/9)*imconv(mat,kern);
--> xbascc(); imshow(mat_conv);
```

### 5.1.2 Pre-defined filters

Do not reinvent the wheel all the time. There are several kernels which come with SIP.

```
--> mkfilter('mean')
--> mat_conv=imconv(mat,mkfilter('mean'));
--> xbasec(); imshow(mat_conv);
```

PS: if you've got an interesting kernel, please consider sending it to the SIP development list.

#### Low-pass filters

Low-pass filters reduce the variations of the luminosity in the image.

2 examples:

- "mean"

|     |     |     |
|-----|-----|-----|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

- "low-pass"

|      |      |      |
|------|------|------|
| 1/12 | 1/12 | 1/12 |
| 1/12 | 4/12 | 1/12 |
| 1/12 | 1/12 | 1/12 |

This one produces less correction than the "mean" kernel.

#### Laplaciens

Using this kernel will show points of the image where intensity is varying quickly (2nd derivative).

Because images are discrete (and not continuous), the second derivative can only be approximated. here are 3 common approximative kernels:

- "laplace1"

|    |    |    |
|----|----|----|
| 0  | -1 | 0  |
| -1 | 4  | -1 |
| 0  | -1 | 0  |

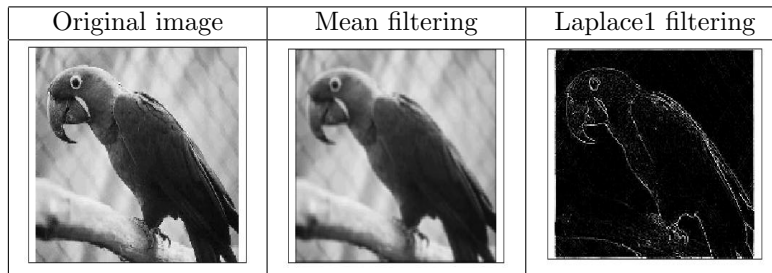
- "laplace2"

|    |    |    |
|----|----|----|
| -1 | -1 | -1 |
| -1 | 8  | -1 |
| -1 | -1 | -1 |

- "laplace3"

|    |    |    |
|----|----|----|
| 1  | -2 | 1  |
| -2 | 4  | -2 |
| 1  | -2 | 1  |

### 5.1.3 Examples



## 5.2 Other filters

These are non-linear filters.

### 5.2.1 Median filter

Consider the following pixels:

|   |   |   |
|---|---|---|
| 1 | 5 | 1 |
| 2 | 4 | 3 |
| 8 | 2 | 1 |

If you order these values, you have:

1 1 1 2 2 3 4 5 8

The median value is 2 (the 5th position), so the value 2 is affected to the central pixel.

This is a low-pass filter used to despeckle an image (it suppresses the high-frequency noise without blurring too much the image). Very usefull with images obtained with lasers (because of speckle).

--> `mmat=mogrify(mat,'-median 1')`; performs a median filter. The radius of the matrix used is 1, ie it's a 3x3 matrix.

### 5.2.2 Histogram equalization

This operation is often used to enhance the contrast of an image:

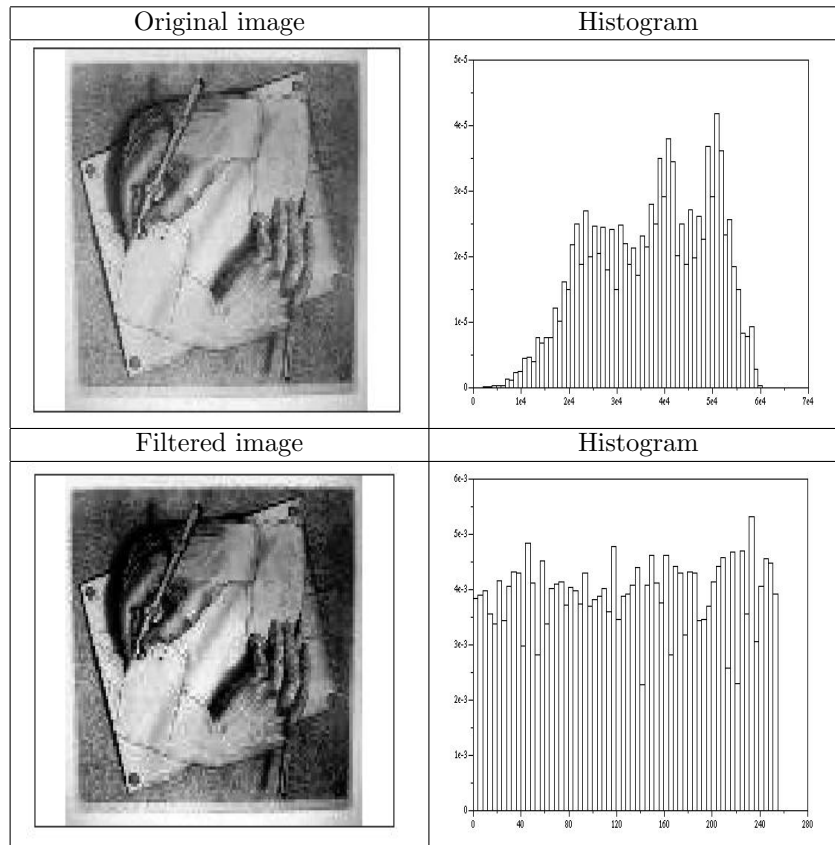
```
--> image=gray_imread('gra.jpg');
--> xset('window',0);imshow(image);
--> fimage=mogrify(image,['-equalize']);
```

```
--> xset('window',1);imshow(fimage);
```

If you want to see histogram modifications, you can plot them with:

```
--> xset('window',2);histplot(64,image);
```

```
--> xset('window',3);histplot(64,fimage);
```



# Chapter 6

## Using fft in images

### 6.1 Observing the spectrum

The Fourier Transform gives informations about which frequencies are present in a signal (=spectrum).

In images, small details correspond to high frequencies.

A great interest of the spectrum is that the original image can be re-obtained, using the spectrum. Of course, adding or suppressing frequencies in the spectrum will result in a modified image.

In optics, the diffraction figure represents the spatial frequencies of the aperture or Fourier spectrum. Small details (a slim slit) will create high frequencies, localised far from the center.

Usually you modify the spectrum and then recreate an image of the initial object (strioscopy).

In this first section, we will just visualize the influence of several filters on the spectrum.

```
--> image=zeros(400,300);
--> image(180:220,145:155)=1;
--> xset('window',0);xbase();imshow(image);
--> IM=fft(image,-1);
--> spectrum=real((IM).*conj(IM)); calculates the image power spectrum.
```

```
--> xset('window',1);xbase();imshow(spectrum,[]); shows a spectrum with low frequencies located at each corner. If you want to have low frequencies in the middle, just:
```

```
--> spectrum2=sip_fftshift(spectrum);
--> xset('window',2);xbase();imshow(spectrum2,[]);
```

Of course it's difficult to see all the frequencies. The figure shows a very high central peak. A better visualisation can be obtained by two methods:

- use the log properties:  
--> spectrum3=log(spectrum2+1);

```
--> xset('window',3);xasc();
--> plot3d1(1:4:400,1:4:300,spectrum3(1:4:400,1:4:300));
```

- or you can "saturate" the image:

```
--> spectrum3=spectrum2;threshold=1e3;
--> spectrum3(find(spectrum3>threshold))=threshold;
--> xset('window',3);xasc();
--> plot3d1(1:4:400,1:4:300,spectrum3(1:4:400,1:4:300));
```

See the effects of a low-pass and of a sharpener filter on the spectrum by

```
-->image=imconv(image,mkfilter('mean')); or
-->image=imconv(image,mkfilter('sh2')); before calculating the fft.
```

## 6.2 Modifying an image by acting on the spectrum

The original image is restituted by

```
--> im2=real(fft(IM,1));
```

Before restitution, we will apply a filter on the spectrum. The simplest filter is a binary one. Usually these filters are in cylindrical coordinates.

Let's see that on a real example:

```
-->image=gray_imread(SIPDIR+'images/ararauna.jpg');
-->[r,c]=size(image);
-->xset("window",0);xasc();imshow(image);
-->IM=fft(image,-1);
-->spectrum=real((IM).*conj(IM));
-->xset("window",1);xasc();imshow(spectrum,[]);
-->sp2=fftshift(spectrum);to center the spectrum
-->xset("window",2);xasc();imshow(sp2,[]);
-->sp3=log(sp2+1);
-->xset("window",3);xasc();imshow(sp3,[]);
-->//design the binary filter
-->z=zeros(image);
-->x0=round(r/2);radius=10;y0=round(c/2);
-->for x=x0-radius:x0+radius
-->  y=round(sqrt(radius^2-(x-x0)^2));
-->  z(x,y0-y:y0+y)=1;
-->end;
-->//if you want to inverse the filter
-->//z=abs(1-z);//complementary filter
-->xset("window",4);xasc();imshow(z,[]);
-->IM2=IM.*fftshift(z);//spectrum modification
-->//reverse transform
-->im2=real(fft(IM2,1));
```

```
-->xset("window",5);xasc();imshow(im2,[]);
```


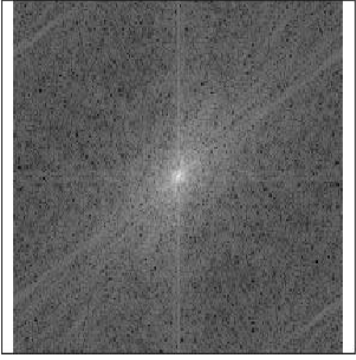
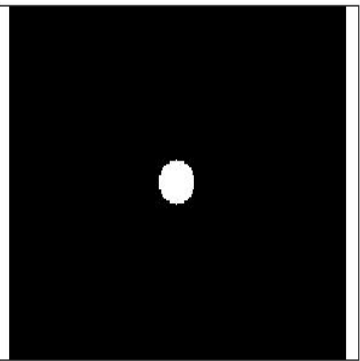

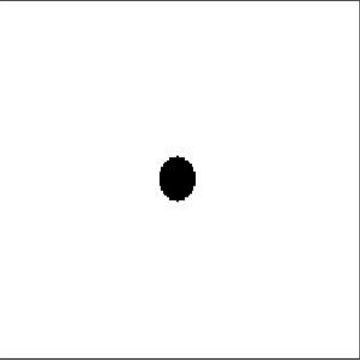
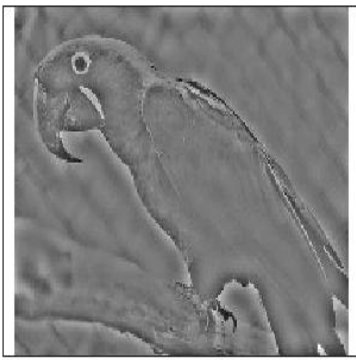
Be careful, the following is not equivalent:

```
-->IM3=fftshift(IM).*z;//spectrum modification
```

```
-->im3=real(fft(IM3,1));
```

High frequencies must be in the center to allow a correct behavior of the reverse Fourier Transform.

You can find already built filters with the function `mkfftfilter`.

|   |  |
|---|--|
| <p data-bbox="592 331 767 360">Original image</p>            | <p data-bbox="922 331 1279 360">Spectrum (log representation)</p>  |
| <p data-bbox="644 754 715 784">Filter</p>                   | <p data-bbox="1007 754 1193 784">Resulting image</p>              |
| <p data-bbox="555 1178 804 1207">Complementary filter</p>  | <p data-bbox="1007 1178 1193 1207">Resulting image</p>           |



## Chapter 7

# Interfacing C with SIP

### 7.1 Why ?

Scilab is a great language but it's an interpreted one. So, it's easy to use, but unfortunately slow. In particular, always try to avoid loops to maximize performance (use operations on matrix instead).

Well, a time will come when you'll want speed in Scilab - SIP, with your huge images and complex treatments.

There are two solutions: buy a newer PC or learn to interface C code to Scilab.

The benefit can be very high: I had a scilab function which took 14 seconds to complete. When I coded it in C, it took 0.1 second for the same result. So consider this ...

### 7.2 Warnings

1. I'm a poor C coder. Only necessity (a bit of curiosity also ?) made me do this.
2. I'm open to improvements you suggest...
3. this section is very inspired from "Introduction a Scilab" by the Scilab Group.

I just tried to make simpler things than they did.

### 7.3 How ?

Once you have your nice C function, you need to make it communicate with Scilab.

This is done by writing an interface. You can

- write it by hand: it's what we'll do
- generate it with intersci. Refer to the documentation of this tool if you want to do so.

There are several examples of interfacing in the directory `examples` of Scilab. You could have a look at `interface-tutorial` to begin.

### 7.3.1 Adding a real to a matrix

This is the first example, the easiest I found useful: the aim is to create a new Scilab function with this syntax:

```
C=imadd(A);
```

The function will simply add a real to the matrix.

I chose to add 0.4 to each element of the input matrix.

Here is the code:

```

1  /*
   * -----
   * imadd_int.c:
   * an example for learning interfacing
5  * -----
   */
   #include <string.h>
   #include "sip_common.h"

10  static void
   imadd(double a[], double c[],int m, int n);

   SipExport int
   imadd_int(char *fname)
15  {
   static int l1, m1, n1,/* a*/
   l2;/* c*/
   static int minlhs=1, maxlhs=1, minrhs=1, maxrhs=1;

20  /* Check number of inputs and outputs */
   CheckRhs(minrhs,maxrhs);
   CheckLhs(minlhs,maxlhs);

   /*Input parameters: */
25  /*GetRhsVar(number, type, matrix_size1, matrix_size2, value);*/
   GetRhsVar(1, "d", &m1, &n1, &l1);

   /* Ouput creation:*/
   /* CreateVar(number, type, matrix_size1, matrix_size2, &l3); */
30  CreateVar(2, "d", &m1, &n1, &l2);

```

```

    /* Call function */
    imadd(stk(l1), stk(l2), m1, n1);

35  /* The return variable is number (2) */
    LhsVar(1) = 2;
    return 0;
}

40 #define A(i,j) a[i + j*m]
    #define C(i,j) c[i + j*m]

    /* m= number of rows, n=columns*/
    /* i and j are row and column*/
45 void
    imadd(double a[], double c[],int m, int n)
    {
        int i,j;
        double s;

50     s=0.4;

        for( i=0 ; i < m; i++){
            for( j=0; j < n; j++){
55         C(i,j)=A(i,j)+s;
            }
        }
    }
}

```

I put the function `imadd` and its interface `imadd_int` in the same file, but it could be in separate ones.

It seems difficult, but once you've got an interface, you can easily duplicate it for other functions.

Explanations:

- ligne 10-11: the input matrix is  $a$ , the output one is  $c$ ,  $m$  and  $n$  are the sizes (rows and columns).
- line 20: check the number of arguments. rhs are for input arguments, lhs for output.

The easiest is to have  $\text{maxrhs}=\text{minrhs}$ .

- line 26: we get the input parameter:  $a$  is in the 1st place of `imadd`, its type is "double" ("d"), `&m1` and `&n1` are the sizes and `&l1` is an address to access datas.

- line 30: we create the output parameter: *c* is in the 2nd place of *imadd*, its type is "double" ("d"), its sizes are  $&m1$  and  $&n1$  (same as *a*) and  $&l2$  is an address to access *datas*.
- line 33: we call the *imadd* function.
- line 36: the 1st output parameter is in 2nd place in *imadd*.

### 7.3.2 Loading the code in Scilab

Now, we have a really great new function but Scilab does not recognize it yet.

To load the code in Scilab, you need a *builder.sce* file in the same directory as *imadd\_int.c*.

Here is an example:

```

1 // This is the builder.sce
  // must be run from this directory

  ilib_name = 'libtutorial'           // interface library name
5 files = ['intview.o',..
          'intmatmul.o',..
          'unwrapl_c_int.o',..
          'imvariance_int.o',..
          'imadd_int.o'] // objects files
10
  libs = []                          //
  table = [ 'view', 'intview';        // table of (scilab_name,interface-name)
           'matmul','intmatmul';
           'unwrapl_c','unwrapl_c_int';
15           'imvariance','imvariance_int';
           'imadd','imadd_int']; // for fortran coded interface use 'C2F(name)'

  // do not modify below
  // -----
20 ilib_build(ilib_name,table,files,libs)

```

This file will add 5 new functions to the library named 'libtutorial'.

You have to adapt to your own needs but it's very easy.

Then you launch Scilab, and execute `exec('builder.sce')`;

If everything is allright, many files were created. Just do: `exec('loader.sce')`;  
and test in Scilab: `a=[5 8];b=imadd(a)`;

*Note:* *builder.sce* is needed to compile a new library. After that only *loader.sce* is needed.

### 7.3.3 A bit more difficult

Now, we want to be able to pass the value to add in the function: `b=imadd2(a,0.3)`;  
to add 0.3 to each element of *a*.

To complicate, we will store result in a temporary matrix (named *temp1*).

```

1  /*
   * -----
   * imadd2_int.c:
   * another example for learning interfacing (with memory allocation)
5  * -----
   */
   #include <string.h>
   #include "sip_common.h"

10 static void
   imadd2(double a[], double *val, double c[],int m, int n);

   SipExport int
   imadd2_int(char *fname)
15 {
   static int l1, m1, n1,/* a */
       r_val,c_val,l_val, /* val */
       l2;/* c */
   static int minlhs=1, maxlhs=1, minrhs=2, maxrhs=2;

20   /* Check number of inputs and outputs */
   CheckRhs(minrhs,maxrhs);
   CheckLhs(minlhs,maxlhs);

25   /*Input parameters: */
   /*GetRhsVar(number, type, matrix_size1, matrix_size2, value);*/
   GetRhsVar(1, "d", &m1, &n1, &l1);
   GetRhsVar(2, "d", &r_val, &c_val, &l_val);

30   /* Duput creation:*/
   /* CreateVar(number, type, matrix_size1, matrix_size2, &l3); */
   CreateVar(3, "d", &m1, &n1, &l2);

   /* Call function */
35   imadd2(stk(l1), stk(l_val), stk(l2), m1, n1);

   /* The return variable is number (3) */
   LhsVar(1) = 3;
   return 0;
40 }

```

```

#define A(i,j) a[i + j*m]
#define C(i,j) c[i + j*m]

45  /* m= number of rows, n=columns*/
    /* i and j are row and column*/
    void
    imadd2(double a[], double *val, double c[],int m, int n)
    {
50     int i,j;
        double *temp1;

        /*memory allocation*/
        /*Essential !! */
55     temp1=(double*) calloc(n*m,sizeof(double));

        for( i=0 ; i < m; i++){
            for( j=0; j < n; j++){
60             temp1[i+j*m]=A(i,j)+(*val);
            }
        }
        printf("Temporary matrix completed\n");

65     for( i=0 ; i < m; i++){
            for( j=0; j < n; j++){
                C(i,j)=temp1[i+j*m];
            }
70     }
    }

```

Explanations:

- I hope most of the code is clear
- line 55 is very important: if you forget it, the function might work with little matrix, but Scilab will crash with huge ones (images for example).

The functions `GetRhs` and `CreateVar` reserve some memory space, so you don't have to bother about memory management. But, if you use temporary matrix in your function, you have to care about memory allocation (or crash).

Now, if you want to test this example, modify your `builder.sce` file and go !

### 7.3.4 Misc.

Several precisions:

- for integers, use `GetRhsVar(number,"i",&m,&n,&l)` then `istk(l)`
- for characters, use `GetRhsVar(number,"c",&m,&n,&l)` then `cstk(l)`





## Chapter 8

# Thanks

Many thanks to Ricardo Fabbri for creating SIP, for his encouragements and for his support.

I would also like to thank the following persons:

- J.-P Chancelier and the Scilab group who always took the time to answer my questions on Scilab.
- Daniel Droz for helping me start in photonics.



## Chapter 9

# References

Here is a list of the books I used:

- J-P PEREZ - Optique, fondements et applications - 6e edition, DUNOD
- Chancelier, Delebecque, Gomez, Goursat, Nikoukhah, Steer - Introduction a SCILAB - editions SPRINGER
- C. Rolland - Latex par la pratique - O'REILLY



# Chapter 10

# Index

See next page

# Index

colormap, 13  
convolution, 17  
coordinates, 9  
Druel, 6  
Fabbri, 5  
fft, 21  
fftshift, 21  
gray\_imread, 16  
grayscale, 13  
histogram, 19  
im2bw, 16  
im2gray, 16  
imconv, 17  
imread, 8  
imshow, 8  
license, 5  
median, 19  
mkfftfilter, 23  
mkfilter, 18  
modulo, 11  
moglify, 19  
multiplication, 11  
normal, 9  
power, 11  
pseudocolor, 13  
RGB, 16  
spectrum, 21  
stacksize, 7  
strioscopy, 21  
threshold, 16  
truecolor, 13  
typeof, 16