APPLIED COMPUTER GRAPHICS

-Part-1 Field Theory of Computer Graphics-
First draft version September 21th 1999

For *Mathematica* Version 3.01

Yoshifuru SAITO

Department of Electronics and Electrical Engineering
College of Engineering
Hosei University

Kajio Koganei, Tokyo 184-8584, Japan
TEL/FAX:+81-42-387-6200
E-Mail: ysaitoh@ysaitoh.k.hosei.ac.jp

Current e-mail adress: ysaito@hosei.ac.jp

PREFACE

Here is another completely new research area I like to draw you attention to get your kind involvement to promote and advance it: Field Theory of Computer Graphics. It was started by Professor Yoshifuru Saito with rich application results; I am finding it remarkable.

The basic idea is as follows:

When we visualize objects, we cognized them through some incoming information media as changes of the field we are observing. The reverse is true; when display, we represent the light change of the display field. In neither case, we cannot directly handle objects unless we use mechanics to touch them or process them mechanically. That why field theory is very important. The field theory has not been well developed in case of digital and finite fields. Mainly analogue and infinite cases are well known in classical electromagnetism.

Professor Yoshifuru Saito. He is among the most novel and practical academics on this earth. He has been very little outside of Japan, and I appreciate your kindness to grasp what he presents in contents, not through texts. He formulate beautifully too, so your time will be rewarded.

Sincerely yours,

T. L. Kunii

August 18^{th} 1999

Selected sentences from the e-mail of Professor Tosiyasu L. Kunii

Contents

Chapter 1. Basic tools

1.1 Introduction 1

1.2 Preparation of Mathematica 1

1.3 Image data input 2

1.4 Monochrome image 4

1.5 Color image 6

1.6 Window operation 7

1.7 Summary 8

Chapter 2. Monochrome image processing

2.1 Introduction 10

2.2 Preparation of Mathematica 10

2.3 Sample image synthesis 12

2.4 Characteristic vector distribution 13

2.5 Sketch generation 16

2.6 Three-dimensional image generation 17

2.7 Monochrome static image governing equation 21

2.8 Image resolution 25

2.9 Illusive image generation 26

2.10 Summary 28

Chapter 3. Color Image Processing

3.1 Introduction 29

3.2 Preparation of Mathematica 30

3.3 Sample color image 33

3.4 Color characteristic vectors 35

3.5 Sketch and painted image generation 38

3.6 High-resolution image generation 44

3.7 Three-dimensional color image generation 47

3.8 Illusive color imaging 49

3.9 Summary 52

Chapter 4. Wavelet Image Processing

4.1 Introduction 53

4.2 Preparation of Mathematica 54

4.3 Wavelet image compression and recovery 59

4.4 Dynamic image processing 71

4.5 Summary 77

Chapter 5. Eigen Pattern Image Processing

5.1 Introduction 78

5.2 Preparation of Mathematica 79

5.3 The nature of eigen patterns 82

5.4 Summary 94

Chapter 6. Image identifications

6.1 Introduction 95

6.2 Preparation of Mathematica 96

6.3 Graphics image system of equations 97

6.4 Modeling 101

6.5 Image identification in real domain 105

6.6 Image identification in Fourier spectrum domain 118

6.7 Image identification in wavelet spectrum domain 130

6.8 Image identification in eigen pattern domain 143

6.9 Summary 153

Chapter 1. Basic Image Handling Tools

1.1 Introduction

This chapter introduces the basic image handling tools for image processing by *Mathematica*.

The programming language *Mathematica* is composed of the two major frame parts [1]. One is the front-end processor, which is an interface between the computer and user. The front-end is not only used for the text and code writing for the *Mathematica* notebook but also it displays the computed results. Enormous capability is included in the functions of front-end. For further details, click the help menu of the *Mathematica* front-end. The other frame part is the kernel, which carries out the practical computations including a large number of functions for simple computation as well as plotting the figures. The kernel is a computational engine of the *Mathematica* when regarding the front-end as a driving cockpit. Both of the front-end and kernel require a large memory to use the *Mathematica* with comfortable environment.

1.2 Preparation of *Mathematica*

We never have the machine installed infinitely large number of memories, so that it is preferable and essential to use the memory conserve command of *Mathematica*. This command works to conserve the memory by interchanging the contents of variable when using the same name. Also, we install the warning messages suppressing command for the similar variable name. Warning messages are sometimes very important and useful for debugging the codes, but in most cases, that give a negative impression to user. Further, the default package of the *Mathematica* includes various functions, but it is required to install the other standard packages for handling and processing the image data. Thereby, in this chapter we install a “Linear algebra package”. These packages can be installed as follows. We write the package input commands and click the right-hand cell by a mouse cursor. After that, we push the enter plus shift keys simultaneously.

Thus, we have installed the required *Mathematica* packages for starting a session of this chapter. It must be noted here that never push the shift plus enter keys to the cell including the just installed packages. When the shift pus enter operation to the installed packages is again carried out, you will get an enormous error messages punch from the *Mathematica*.

```
<<Utilities`MemoryConserve`
$MemoryIncrement=100000;
Off[General::spell1,MemoryConserve::start,MemoryConserve::end];
<< LinearAlgebra`MatrixManipulation`;
```

1.3 Image data input

One of the major aims of this book is to describe a computer graphics methodology, so that we have to insert the image data. Currently, we are available a large number of image data formats for the digital computers, one of the most popular and primitive image data formats is a 24-bitmap form. The 24-bitmap image data consist of the red, green and blue color components. Each of the color components has 8 bits dynamic range, and takes the numerical values between 0 and 1 The 24-bitmap image data is read in the *Mathematica* notebook by means of "MathLink" utility, which connects the *Mathematica* kernel to the external object package. In this book, we install a package called "RGBsplit.exe" in order to input the 24-bitmap image data. This package was developed by one of my students. In order to install the "RGBsplit.exe", we have to check the existence of "RGBsplit.exe" in the current directly. Typing a following command carries this out and hitting the shift plus enter keys simultaneously.

```
FileNames[]
```

```
{AF038-100.bmp, AF038-128.bmp, AF038-256.bmp,
 AF038-64.bmp, BF001.bmp, BF001M.bmp, Chapter 1 Basic Tools.nb,
 Chapter 2 Monochrome Image Processing.nb,
 Chapter 3 Color Image Processing.nb,
 Chapter 4 Wavelet Image Processing.nb,
 Chapter 5 Eigen Pattern Image Processing.nb,
 Chapter 6 Image Identifications.nb, imageDB63.m, imageTST10.m,
 Preface.nb, RGBsplit.exe, Sor04.exe, Wall-A-128.bmp}
```

As you can see, there is the "RGBsplit.exe", so we install this object by a following command.

```
link=Install["RGBsplit.exe"]
```

```
LinkObject[.\RGBsplit.exe, 2, 2]
```

Using this "RGBsplit.exe", we read in an image file "BF001.bmp" having 24-bitmap format and check up its array size in the *Mathematica* notebook by "Dimensions" command.

```
sample=RGBsplit["BF001.bmp"];
Dimensions[sample]
```

```
{128, 128, 3}
```

The 24-bitmap-image file "BF001.bmp" has been read in the *Mathematica* notebook as a list named "sample", which is a three dimensional array of 128 by 128 by 3 elements. The output of the command "Dimensions" reveals that the first, second and third figures in the wavy parentheses are the number of pixels used for the vertical, horizontal axes and color components, respectively. The 1,2 and 3 of the color components correspond to the red, green and blue, respectively.

Let us draw the red, green and blue components images, independently. The following steps carry this out. At first, the numerical values representing the red, green and blue components are respectively substituted into independent lists "red", "green" and "blue". After that, "ListDensityPlot" command is used for drawing the monochrome images.

```
red=Table[sample[[i,j,1]],{i,128},{j,128}];
green=Table[sample[[i,j,2]],{i,128},{j,128}];
blue=Table[sample[[i,j,3]],{i,128},{j,128}];
redG=ListDensityPlot[red,
    Mesh->False,Frame->False,PlotLabel->"Red",
    DisplayFunction->Identity];
greenG=ListDensityPlot[green,
    Mesh->False,Frame->False,PlotLabel->"Green",
    DisplayFunction->Identity];
blueG=ListDensityPlot[blue,
    Mesh->False,Frame->False,PlotLabel->"blue",
    DisplayFunction->Identity];
Show[GraphicsArray[{redG,greenG,blueG}],
    PlotLabel->"Fig.1. Color components",
    ImageSize->{450,150}];
```

Fig.1. Color components

The meanings of the option commands such as “Mesh”, “Frame” and “DisplayFunction” can be obtained by the command “Options[ListDensityPlot]” or clicking the help menu of the *Mathematica* front-end, e.g.

```
Options[ListDensityPlot]
```

```
{AspectRatio → 1, Axes → False, AxesLabel → None,
 AxesOrigin → Automatic, AxesStyle → Automatic, Background → Automatic,
 ColorFunction → Automatic, ColorOutput → Automatic,
 DefaultColor → Automatic, Epilog → {}, Frame → True, FrameLabel → None,
 FrameStyle → Automatic, FrameTicks → Automatic, ImageSize → Automatic,
 Mesh → True, MeshRange → Automatic, MeshStyle → Automatic,
 PlotLabel → None, PlotRange → Automatic, PlotRegion → Automatic,
 Prolog → {}, RotateLabel → True, Ticks → Automatic,
 DefaultFont :→ $DefaultFont, DisplayFunction :→ $DisplayFunction,
 FormatType :→ $FormatType, TextStyle :→ $TextStyle}
```

The readers are very surprised by so many options even though a relatively simple command “ListDensityPlot”, but never mind such a too many options. The readers are essentially led to use a limited number of options and commands in *Mathematica* depending on demands of their affectivities.

Thus, we have succeeded in reading the image file into the *Mathematica* notebook. To conserve the memory used for the “RGBsplit.exe” object from an entire computer memory, we remove this object from the memory by “Uninstall” command.

```
Uninstall[link];
Off[General::spell1];
```

1.4 Monochrome image

Many of the digital cameras can take a color image, but still the monochrome images are used for the industrial use for sake of its cost performance. Two types of monochrome images are considered in this book. One is the bitmap image file having only the monochrome information. When we read such a monochrome image in the *Mathematica* notebook by the “RGBsplit.exe” command,

```
link=Install["RGBsplit"];
monoSample=RGBsplit["BF001M.bmp"];
Uninstall[link];
Off[General::spell1];
```

then after substituting the numerical values included in the list “monoSample” into a list “monoData”, we check up the numerical values included in the list “monoData” by a command “==“. The command “==“ gives the “True” if the objects are the same else “False”.

```
dim=Dimensions[monoSample];
monoData=
    Table[monoSample[[i,j,k]],
        {k,dim[[3]]},{i,dim[[1]]},{j,dim[[2]]}];
```

The first, we compare the first and second components.

```
monoData[[1]]==monoData[[2]]
```

```
True
```

This result means that the first and second image components in the list “monoData” are the same numerical values. Similarly, we check up equivalence between the first and third components in the list “monoData”.

```
monoData[[1]]==monoData[[3]]
```

```
True
```

Thereby, the monochrome image data in the list “monoData” by installing the command “RGBsplit.exe” have the same color components, which represent a monochrome image as shown in Fig.2.

```
ListDensityPlot[monoData[[1]],
    Mesh->False,Frame->False,PlotLabel->"Fig.2. Monochrome"];
```

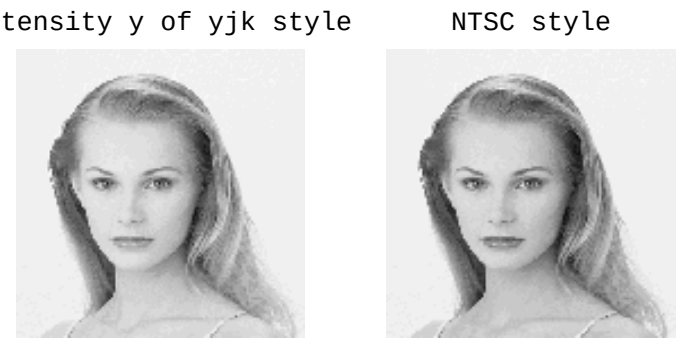
Fig.2. Monochrome

Thus, we can handle the monochrome image data by installing the command “RGBsplit.exe”. However, it must be noted here that the monochrome image format will greatly depend on a processing tool. In this textbook, we worked out a sample monochrome image “BF001M.bmp” form the color image data “BF001.bmp” by a popular commercial/shareware base image drawing software “PaintShopPro” for the Widows 95, 98 and NT versions.

The other way to obtain a monochrome image is to compose the monochrome images from the red, green and blue color components shown in Fig.1. We take up here the two types of monochrome images. One is the y component intensity of the yjk style, and the other is the NTSC style used in the televisions. Figure 3 shows both of the synthesized yjk and NTSC types monochrome images from the color components shown in Fig.1.

```
yjk = ( .5 * red + .25 * green + .125 * blue);
tvM = (.3 * red + .59 * green + .11 * blue);
yjkG = ListDensityPlot[yjk, Mesh -> False, Frame -> False,
            PlotLabel -> "Intensity y of yjk style",
            DisplayFunction -> Identity];
tvMG = ListDensityPlot[tvM, Mesh -> False, Frame -> False,
        PlotLabel -> "NTSC style",
        DisplayFunction -> Identity];
Show[GraphicsArray[{yjkG, tvMG}],
        PlotLabel -> "Fig.3. Two types of monochrome images",
        ImageSize -> {300, 150}];
```

Fig.3. Two types of monochrome images



1.5 Color image

In order to represent a color image in a list “sample”, we have to use the two *Mathematica* commands; one is the “RGBColor” function, which converts the numerical data representing the red, green and blue components into RGB color form. The other is the “RasterArray” function, which displays a color image from the RGB color data.

At first, we convert the numerical data in the list “red”, “green” and “blue” into a RGB color form by means of the function “RGBColor”. Secondly, we draw a color graphics image by combining a set of display commands “Show”, “Graphics”, and “RasterArray”.

```
rgbSample=Table[RGBColor[red[[i,j]],green[[i,j]],blue[[i,j]]],
    {i,128},{j,128}];
Show[Graphics[RasterArray[rgbSample],
    AspectRatio->Automatic],PlotLabel->"Fig.4. Color graphics image"];
```

Fig.4. Color graphics image

1.6 Window operation

Sometimes, we are required to focus on a particular region on an image. In such case, it is convenient to use a window operation extracting a region of interest [2]. Window operation is one of the convolution operations, and performs the element-to-element multiplications between the two lists containing the window and image data. In this textbook, we work out a simple round shape of window constructed by a following function.

```
window[xl_, yl_, radius_] :=
    Module[{i = 0, j = 0, k = 0, p = 0, q = 0, win = {{0.}}},
                win = ZeroMatrix[yl, xl];
                p = Round[0.5 yl]; q = Round[0.5 xl];
                Do[j = Round[Sqrt[radius^2 - i^2]];
                    Do[win[[p + i, k]] = 1., {k, q - j, q + j}],
            {i, -radius, radius}];
        win];
```

In the function “window”, the parameters “xl”, “yl” and “radius” are the number of pixels in the direction of x, y axes and a radius of circular window, so that the a number of pixels for the parameter “radius” should be always

smaller than those of the half of “xl” as well as “yl”. Using this function, we work out a 128 by 128 pixels list “win128” having a window's radius 60.

```
win128=window[128,128,60];
```

By means of the window “win128”, we carry out the convolution operations to the monochrome as well as each of the color components in Fig.1.

```
wMonoG=ListDensityPlot[win128*monoData[[1]],
    Mesh->False,Frame->False,PlotLabel->"Monochrome",
    DisplayFunction->Identity];
redW=win128*red;
greenW=win128*green;
blueW=win128*blue;
```

After converting the color image data into a RGB form, we draw the images having circular shape windows. Figure 5 shows the circular window operated images.

```
rgbSampleW=
    Table[RGBColor[redW[[i,j]],greenW[[i,j]],blueW[[i,j]],
    {i,128},{j,128}];
wColorG=Show[Graphics[RasterArray[rgbSampleW],
    AspectRatio->1.0],PlotLabel->"Color",
    DisplayFunction->Identity];
Show[GraphicsArray[{wMonoG,wColorG}],
    PlotLabel->"Fig. 5. Examples of window operation",
    ImageSize->{300,150}];
```

Fig. 5. Examples of window operation

1.7 Summary

This chapter has described about the most basic and fundamental part of this book. In spite of no knowledge about the programming language “*Mathematica*”, every one may be understood how to input and draw the images in the *Mathematica* notebook. Also, every one may be revealed that an image processing is not the simple try and error processes using the image handling software, but are the *Mathematica*l list and matrix operations. One of the best merits using the *Mathematica* is that every one can use the codes described in this textbook only setting up their original bitmap image data.

■ REFERENCES

[1] Stephen Wolfram, *The Mathematica Book*, 3rd ed. (Wolfram Media/Cambridge University Press, 1996).

[2] Yoshifuru Saito, *Introduction to image processing by Mathematica* in Japanese (Asakura Publishing Co.LTD., Tokyo 1998).

Chapter 2. Monochrome Image Processing

2.1 Introduction

At the beginning of this chapter, we introduce a concept of the gradient and curl operations of the classical field theory. Most of the conventional computer graphics, space derivatives are often carried out in order to extract the edges of a target object in a screen. An introduction of the vector operations instead of simple spatial derivatives renders a physical meaning to the operations. As an application of the gradient and curl operations, we draw the sketch images, even though the sketch drawing is one of the art works based on the human emotion. Regarding the numerical values representing a monochrome image as a scalar or one component of vector potentials, gradient operation to the scalar potentials or curl operations to the vector potentials yields one of the vector fields. Depending on the magnitude of potentials, a vector distribution takes different form. A vector magnitude distribution leads to a sketch image. Further, inner product operation among the vectors yields the other type images.

Second stage of the field theory in this chapter is to apply the Laplacian operator to the image data when regarding the numerical values representing an image as the scalar potentials. The Laplacian operation leads to a static image governing equation. A Poisson type partial differential equation is the static image governing equation. It is demonstrated that a size or resolution of an image can be changed freely.

The image data in computer graphics are essentially discretized quantities so that we have to carry out the vector calculus in terms of the discrete *Mathematica*l means. In this textbook, the gradient and curl operations are carried out by the central finite differences, and the Laplacian operation is carried out by the 9 points finite difference formula.

2.2 Preparation of *Mathematica*

Before to move on the practical image processing, we have to install the memory conserve utilities and the warning messages suppressing command for the similar variable name. In addition, the "LinearAlgebra`MatrixManipulation" and "Graphic`PlotField" packages have to be installed. The former is used for the list and matrix operations, and the latter is used for plotting the vector fields [1].

Mathematica packages

```
<<Utilities`MemoryConserve`
$MemoryIncrement=100000;
Off[General::spell1,MemoryConserve::start,MemoryConserve::end];
<< LinearAlgebra`MatrixManipulation`;
<<Graphics`PlotField`
```

Mathematica function rgbBMP

Further, we define several functions for image processing and data handling. The functions “rgbBMP”, “convertRGB” and “monoNTSC” are defined as follows. The function “rgbBMP” works to read in the 24-bitmap color image data. A parameter “colorFile” refers to a file name.

```
rgbBMP[colorFile_] :=
 Module[{i = 0, j = 0, k = 0, dim = {0}, input = {{{0.}}}},
  link = Install["RGBsplit.exe"]; Pause[0.01];
  input = RGBsplit[colorFile];
  Uninstall[link];
  Off[General::spell1];
  dim = Dimensions[input];
  Table[input[[i, j, k]], {k, 3},
   {i, dim[[1]]}, {j, dim[[2]]}]];
```

Mathematica function convertRGB

The function “convertRGB” converts the numerical data representing a color image into a RGB form in order to visualize the image on the *Mathematica* notebook. The parameter “colorData” refers to a list including the numerical values.

```
convertRGB[colorData_] :=
 Module[{i = 0, j = 0, dim = {0}, out},
  dim = Dimensions[colorData];
  Graphics[
   RasterArray[
    Table[RGBColor[colorData[[1, i, j]],
     colorData[[2, i, j]], colorData[[3, i, j]]],
   {i, dim[[2]]}, {j, dim[[3]]}]]]];
```

Mathematica function monoNTSC

The function “monoNTSC” construct a NTSC style monochrome image data from the color image data. The parameter “colorData” refers to a list including the numerical values.

```
monoNTSC[colorData_] :=
 Module[{i = 0, j = 0, dim = {0}},
  dim = Dimensions[colorData];
  Table[0.3 * colorData[[1, i, j]] + 0.59 * colorData[[2, i, j]] +
   0.11 * colorData[[3, i, j]],
 {i, dim[[2]]}, {j, dim[[3]]}]];
```

2.3 Sample image synthesis

In this section, we construct a sample monochrome image from a color image. At first, we read in a color image file “BF001.bmp” by means of the function “rgbBMP”.

```
colorSample=rgbBMP["BF001.bmp"];
```

Second, after converting the color image data “colorSample” into the RGB form by the function “convertRGB”, the original color image data is composed.

```
colorSampleG=Show[convertRGB[colorSample],
    PlotLabel->"Color image",AspectRatio->1,
    DisplayFunction->Identity];
```

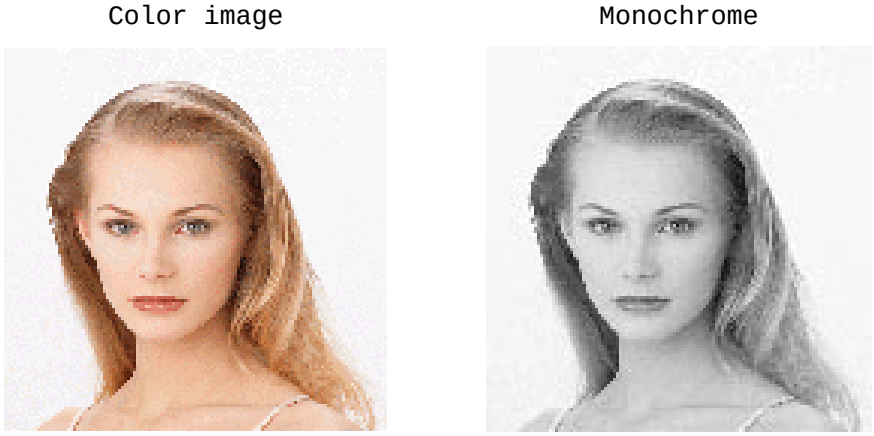
Third, after composing a monochrome image data “monoSample” by the function “monoNTSC”, we compute a monochrome sample image.

```
monoSampleG=
    ListDensityPlot[monoSample=monoNTSC[colorSample],
        Mesh->False,Frame->False,PlotLabel->"Monochrome",
        DisplayFunction->Identity];
```

Finally, we draw the original and monochrome sample images, which are shown in Fig.1. A reason why we use the color image sample “BF001.bmp” not the monochrome image sample “BF001M.bmp” is to introduce the color handling functions “rgbBMP” and “convertRGB” for the later chapters.

```
Show[GraphicsArray[{colorSampleG,monoSampleG}],
    PlotLabel->"Fig.1. Sample images",ImageSize->{400,200}];
```

Fig.1. Sample images



2.4 Characteristic vector distribution

The list “monoSample” is a simple two-dimensional array that contains the numerical values representing the NTSC style monochrome image. When we regard the numerical values in the list “monoSample” as one of the scalar potentials, then it is possible to obtain a set of divergent fields by gradient operation. On the other side, when we regard the numerical values in the list “monoSample” as one of the vector potential components, a set of rotational vector fields can be computed by curl operation. In order to derive the divergent as well as rotational vector of the sample monochrome image shown in Fig.1, we define the “grad2D” and “curl2D” functions. The former and latter functions derive the divergent and rotational vectors, respectively. The “grad2D” function is defined by a following function, where a parameter “darta2D” refers to a list containing two-dimensional scalar potentials.

Mathematica function grad2D

Gradient operation:

$$\boldsymbol{V}_{\mathrm{div}} = -\nabla U = -\frac{\partial U}{\partial x}\boldsymbol{i} - \frac{\partial U}{\partial y}\boldsymbol{j}, \tag{1}$$

Where $\mathbf{V}_{\mathrm{div}}$, U, **i** and **j** are the divergent vector to be evaluated, input scalar potential and unit directional vectors in the direction of x- and y-axes, respectively.

```
grad2D=Compile[{{data2D,_Real,2}},
    Module[{i=0,j=0,dim=Dimensions[data2D],
        c1={Table[data2D[[1,j]],{j,dim[[2]]}]},
        c2=Transpose[{Join[{data2D[[1,1]]},
            Table[data2D[[i,1]],{i,dim[[1]]}]]}],
        e2D=AppendRows[c2,AppendColumns[c1,data2D]]},
    Table[{-0.5(e2D[[i,j]]+e2D[[i+1,j]])+
            0.5(e2D[[i,j+1]]+e2D[[i+1,j+1]]),
            -0.5(e2D[[i,j]]+e2D[[i,j+1]])+
            0.5(e2D[[i+1,j]]+e2D[[i+1,j+1]])},
            {i,dim[[1]],1,-1},{j,dim[[2]]}]]];
```

Mathematica function curl2D

Similarly, the “curl2D” function is defined by a following way, where a parameter “darta2D” refers to a list containing one of the vector potential components arraigned in a two-dimensional form. Both of the gradient and curl operations have been carried out by the central finite differences.

Curl operation:

$$\boldsymbol{V}_{\mathrm{rot}} = -\nabla \times U_z = \frac{\partial U_z}{\partial y}\boldsymbol{i} - \frac{\partial U_z}{\partial x}\boldsymbol{j}, \tag{2}$$

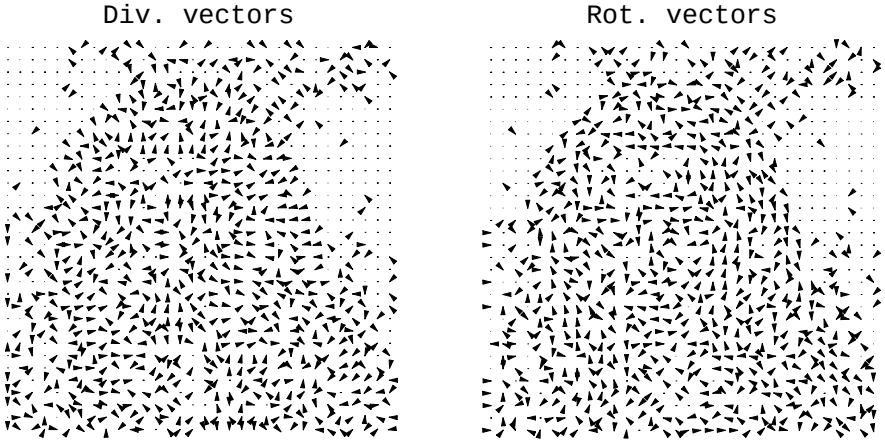
Where $\mathbf{V}_{\mathrm{rot}}$, U_z, **i** and **j** are the rotational vector to be evaluated, input z-component of vector potential and unit directional vectors in the direction of x- and y-axes, respectively.

```
curl2D=Compile[{{data2D,_Real,2}},
    Module[{i=0,j=0,dim=Dimensions[data2D],
        c1={Table[data2D[[1,j]],{j,dim[[2]]}]},
        c2=Transpose[{Join[{data2D[[1,1]]},
            Table[data2D[[i,1]],{i,dim[[1]]}]]}],
        e2D=AppendRows[c2,AppendColumns[c1,data2D]]},
    Table[{-0.5(e2D[[i,j]]+e2D[[i,j+1]])+
        0.5(e2D[[i+1,j]]+e2D[[i+1,j+1]]),
        0.5(e2D[[i,j]]+e2D[[i+1,j]])-
        0.5(e2D[[i,j+1]]+e2D[[i+1,j+1]])},
    {i,dim[[1]],1,-1},{j,dim[[2]]}]];
```

Let us compute the divergent as well as rotational vectors from the sample monochrome image. We apply the functions “grad2D” and “ curl2D” to the list “monoSample”. After computing the divergent and rotational vectors, we plot the vector distributions. Figure 2 shows both of the divergent and rotational vector distributions, where the vectors are periodically sampled with 4 pixels.

```
divV=-grad2D[monoSample];
rotV=curl2D[monoSample];
dim=Dimensions[divV];
divVG=ListPlotVectorField[
    Table[divV[[i,j,k]],{i,1,dim[[1]],4},
        {j,1,dim[[1]],4},{k,dim[[3]]}],
        Frame->False,PlotLabel->"Div. vectors",
        DisplayFunction->Identity];
rotVG=ListPlotVectorField[
    Table[rotV[[i,j,k]],{i,1,dim[[1]],4},
        {j,1,dim[[1]],4},{k,dim[[3]]}],
        Frame->False,PlotLabel->"Rot. vectors",
        DisplayFunction->Identity];
Show[GraphicsArray[{divVG,rotVG}],ImageSize->{400,200},
    PlotLabel->"Fig.2. Vector distributions"];
```

Fig.2. Vector distributions



The divergent and rotational vectors are the orthogonal each other. To check up this, we compute the inner products between them located at the same position, and then we print out the maximum as well as minimum values of the inner products.

```
innerPro=Table[divV[[i,j]].rotV[[i,j]],
    {i,dim[[1]]},{j,dim[[2]]}];
Print["Max=",Max[innerPro]," Min=",Min[innerPro]];
```

```
Max=0. Min=0.
```

Thus, we have confirmed that the divergent and rotational vectors are orthogonal. This means that both of the divergent and rotational vectors of a monochrome image independently exist each other. In the other words, according to the Helmholtz's theorem, taking a sum of the divergence and rotational field vectors can represent any vector fields, so that taking a sum of the divergent and rotational vectors represents the characteristic vector distribution of a monochrome image [2]. Figure 3 shows a characteristic vector distribution of the monochrome image in Fig.1, where the vectors are periodically sampled with 2 pixels.

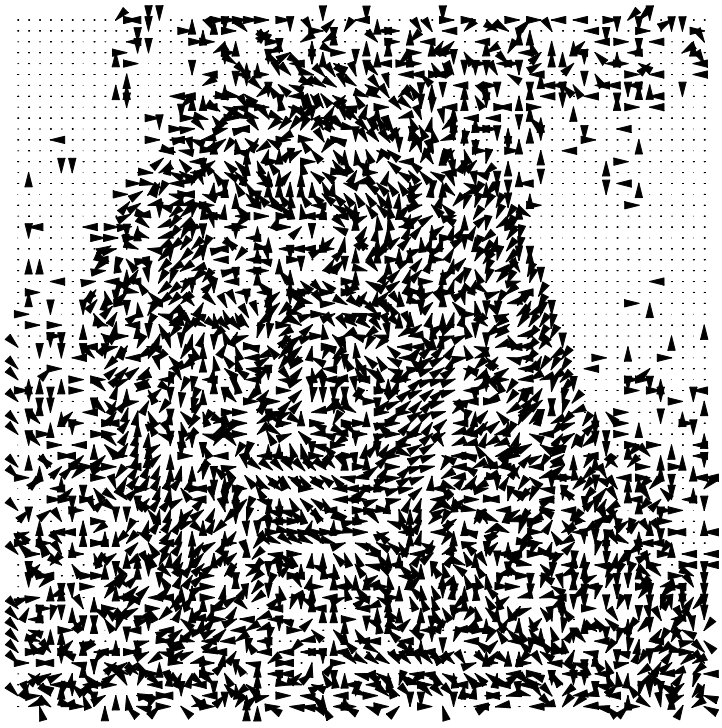
Helmholtz's theorem:

$$\mathbf{V} = \mathbf{V}_{\mathrm{rot}} + \mathbf{V}_{\mathrm{div}}\,, \tag{3}$$

Where $\mathbf{V}$, $\mathbf{V}_{\mathrm{div}}$ and $\mathbf{V}_{\mathrm{rot}}$ are the arbitrary, divergent and rotational vectors, respectively.

```
chractV=divV+rotV;
ListPlotVectorField[
    Table[chractV[[i,j,k]],{i,1,dim[[1]],2},
        {j,1,dim[[1]],2},{k,dim[[3]]}],
    Frame->False,
    PlotLabel->"Fig.3.Characteristic vector distribution"];
```

Fig.3.Characteristic vector distribution



2.5 Sketch generation

Sketch is one of the art works extracting the characteristics of a target object and then depicting by a set of monochrome lines. From such a viewpoint, it is difficult to work out a sketch art by means of computers. However, we have succeeded in extracting the characteristic vector distribution of the monochrome image. If this characteristic vector distribution exactly representing the distinct characteristics of the image, then it is possible to sketch the image. One of the simplest ways for sketching the monochrome image is to plot the magnitudes of the characteristic vectors distribution. In order to do this, we define a function "vectMag2D", which compute the vector magnitudes distribution at each position. A parameter "vector2D" is a three-dimensional array containing the two-dimensional vector components.

Mathematica function vectorMag2D

```
vectorMag2D=Compile[{{vector2D,_Real,3}},
    Module[{i=0,j=0,dim=Dimensions[vector2D]},
        Sqrt[Table[vector2D[[i,j,1]]^2+vector2D[[i,j,2]]^2,
        {i,dim[[1]],1,-1},{j,dim[[2]]}]]];
```

After computing the vector magnitude distribution of Fig. 3, we plot this in reversing the black and white mode. Thus, an obtained sketch is shown in Fig. 4. Obviously, this figure extracts the characteristics of the monochrome image in Fig. 1, and also represents them by a set of monochrome lines.

```
sketch=vectorMag2D[chractV];
ListDensityPlot[Max[sketch]-sketch,
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->"Fig.4. Sketch"];
```

Fig.4. Sketch

2.6 Three-dimensional image generation

As shown in Fig.4, the sketch captures the characteristics of the sample monochrome image. Thereby, it may be possible to work out the filters extracting the various characters from the vectors distribution in Fig.3.

Mathematica function imageNormalize

Before to continue the discussions, we define a *Mathematica* function "imageNormalize", which converts the 2-dimensional image data between the values 0 and 1.

```
imageNormalize = Compile[{{data2D, _Real, 2}},
  Module[{minimum = Min[data2D]},
   (data2D - minimum) / Max[data2D - minimum]]];
```

In this textbook, we work out the different angled lighting images based on the nature of vector fields. Let us consider the six different directed unit vectors "viewV" given by

```
rotation=6;
viewV=Table[{Cos[2.Pi i/rotation],Sin[2.Pi i/rotation]},
    {i,0,rotation-1}];
```

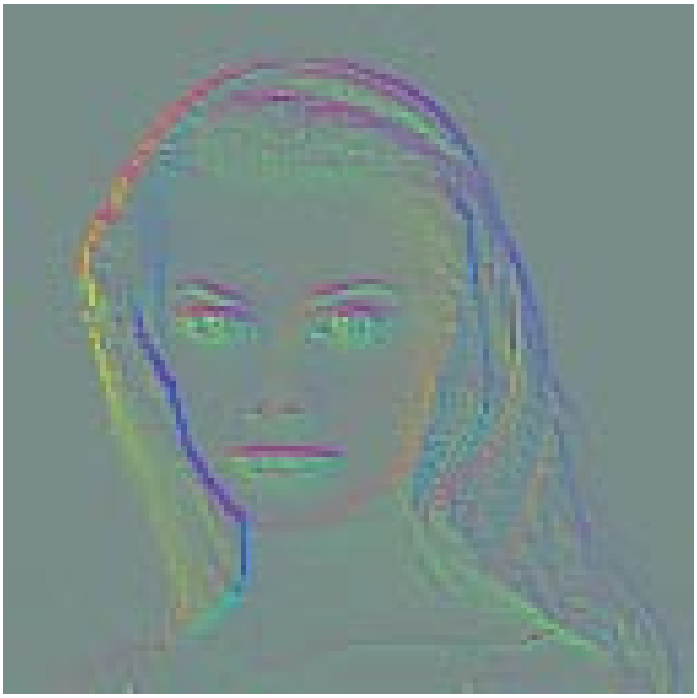
then after computing the inner product “angle” between the vectors in Fig.3 and in the list “viewV”, we normalize the inner products to the values between 0 and 1 by the function “imageNormalize”.

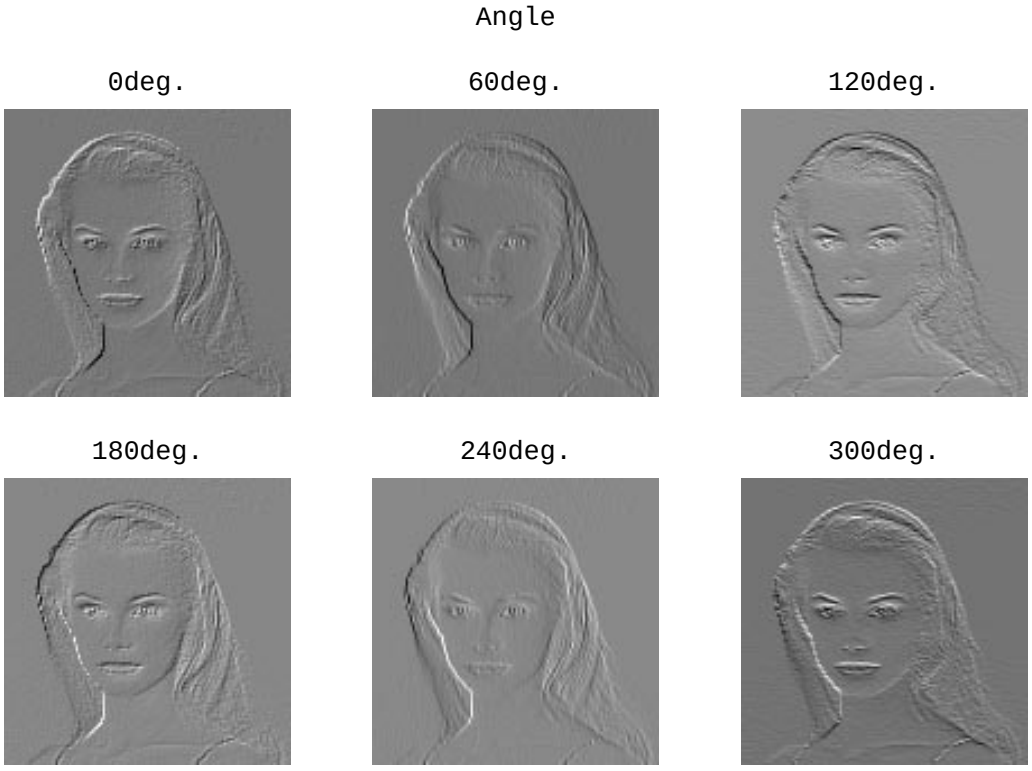
```
angle=Table[chractV[[i,j]].viewV[[k]],
    {k,rotation},{i,dim[[1]],1,-1},{j,dim[[2]]}];
```

Consequently obtained results are shown in Fig. 5, where the first image is represented in terms of the red, green and blue colors which are corresponding to the lighting angles 0, 120 and 240 degrees, respectively. The remaining images in Fig.5 are the shadowed three-dimensional relief images at each of the lighting angles.

```
angleN=Table[imageNormalize[angle[[i]]],{i,rotation}];
angleNG=Table[Table[ListDensityPlot[angleN[[i]],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["`1`deg.",(i-1)360/rotation],
    DisplayFunction->Identity]],
    {i,rotation}];

Show[convertRGB[Table[angleN[[i]],{i,1,rotation,rotation/3}]],
    AspectRatio->1,PlotLabel->"Fig.5.Lighting effect"];
Show[GraphicsArray[
    Table[Table[angleNG[[i+j]],{j,0,2}],{i,1,rotation,3}]],
    ImageSize->{450,300},PlotLabel->"Angle"];
```

Fig.5.Lighting effect





Further, we compute the more realistic three-dimensional image by convoluting the relief images in Fig.5 with the original monochrome image in Fig.1. The obtained images are shown in Fig.6.

```
cAngle=Table[monoSample*(1-imageNormalize[angle[[i]]]),{i,rotation}];

cAngleG=Table[Table[ListDensityPlot[cAngle[[i]],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["`1`deg.",(i-1)360/rotation],
    DisplayFunction->Identity]],
    {i,rotation}];

Show[convertRGB[Table[cAngle[[i]],{i,1,rotation,rotation/3}]],
    AspectRatio->1,PlotLabel->"Fig.6.Convoluted images"];
Show[GraphicsArray[
    Table[Table[cAngleG[[i+j]],{j,0,2}],{i,1,rotation,3}]],
    ImageSize->{450,300},PlotLabel->"Angle"];
```

Fig.6.Convoluted images

Angle

0deg.

60deg.

120deg.

180deg.

240deg.

300deg.

After setting a mouse cursor on a following image, twice clicking the left side button of mouse animates the facial changes depending on the lighting angles. This is only effective on the *Mathematica* notebook.

```
Do[Show[GraphicsArray[{cAngleG[[i]]}]],{i,rotation}];
```

0deg.

2.7 Monochrome static image governing equation

As shown above, the vectored imaging makes it possible to process the image in various ways. In this section, we establish a monochrome image governing equation.

Regarding the numerical values representing a monochrome image as one of he scalar potentials derives this equation. As is well known in the classical field theory, application of the second order partial derivatives to a scalar potential yields a field source density. Namely, when we apply the Laplacian operator to the monochrome image data shown in Fig. 1, then it is possible to obtain a source density distribution of the image. A function "laplace2D" is given by a following *Mathematica* code. This function is based on a 9 points finite difference formula, where the potentials along the edges of a target area are assumed to zero. Also, a parameter "data2D" refers to a two-dimensional array including a monochrome image data.

Mathematica function laplace2D

Laplacian operation:

$$\sigma = \nabla^2 U = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}, \tag{4}$$

where σ and U are the field source density to be evaluated and input scalar potential, respectively.

```
laplace2D=Compile[{{data2D,_Real,2}},
    Module[{e2D={{0.}},out={{0.}},dim={0},
        max=0.,i=0,j=0,nx=0,ny=0},
            dim=Dimensions[data2D];
            ny=dim[[1]]+1;nx=dim[[2]]+1;
            e2D=Table[0.,{ny+1},{nx+1}];
            Do[e2D[[i,j]]=data2D[[i-1,j-1]],
                {i,2,ny},{j,2,nx}];
            Table[e2D[[i+1,j-1]]+4.*e2D[[i+1,j]]+
                e2D[[i+1,j+1]]+4.*e2D[[i,j-1]]-
                20.*e2D[[i,j]]+4.*e2D[[i,j+1]]+
                e2D[[i-1,j-1]]+4.*e2D[[i-1,j]]+
                e2D[[i-1,j+1]],
            {i,2,ny},{j,2,nx}]/6.]];
```

Using this function, we compute the source density of the monochrome image in Fig. 1. Figure 7 shows the source density distribution.

```
source=laplace2D[monoSample];
ListDensityPlot[source,PlotRange->All,
    Mesh->False,Frame->False,
    PlotLabel->"Fig.7. Source density"];
```

Fig.7. Source density

According to the classical field theory, when we are given a source density distribution, then it is possible to establish a field governing equation. A field governing equation having time independent source density is one of the Poisson type equations. Obviously, our given image source density distribution in Fig. 7 is independent to the time, so that the static monochrome image governing equation becomes a Poisson type partial differential equation. Further, a solution of this governing equation should recover the original monochrome image.

In order to solve the image governing equation, we employ a 9 points finite difference formula assuming the zero

Dirichlet boundary condition at the image edges. A function “poissonSOR” gives an iterative solution by a simple successive over relaxation method. A parameter “source” of this function refers to the image source density.

Mathematica function poissonSOR

Poisson's equation:

$$\nabla^2 U = \sigma, \tag{5}$$

where σ and U are the input field source density and scalar potential to be evaluated, respectively.

```
poissonSOR=Compile[{{source,_Real,2}},
    Module[{i=0,j=0,omg=1.8,error=1.,solutionI=0.,diff=0.,
        max=1.*10^-5,dim={0},solution={{0.}},dummy={{0.}}},
    dim=Dimensions[source];
    solution=Table[0.,{dim[[1]]+2},{dim[[2]]+2}];
    dummy=solution;
    Do[dummy[[i+1,j+1]]=source[[i,j]],
        {i,dim[[1]]},{j,dim[[2]]}];
    While[error>max,
        error=0.;
        Do[solutionI=0.05*
            (solution[[i+1,j+1]]+4.*solution[[i+1,j]]+
            solution[[i+1,j-1]]+4.*solution[[i,j+1]]+
            4.*solution[[i,j-1]]+solution[[i-1,j+1]]+
            4.*solution[[i-1,j]]+solution[[i-1,j-1]]+
            6.*dummy[[i,j]]);
            diff=solutionI-solution[[i,j]];
            solution[[i,j]]=solution[[i,j]]+omg*diff;
            error=error+Abs[diff],
        {i,2,dim[[1]]+1},{j,2,dim[[2]]+1}]];
    Table[solution[[i+1,j+1]],
        {i,dim[[1]]},{j,dim[[2]]}]]];
```

By means of this solution routine, we can solve an image governing equation whose source density has been obtained by the Laplacian operation to the monochrome image dada “monoSample”. This function is useful code to show a practical over-relaxation technique using 9 points finite differences, but requires an extremely long CPU time. To overcome this difficulty, we employ the MathLink utility connecting an externally exploited object file to the *Mathematica* kernel.

Mathematica function poissonSOR2D

A function “poissonSOR2D” is the faster solving routine utilizing the MathLink utility of *Mathematica*. The function “poissonSOR2D has two parameters. One is the source density. Also, the other is a one-dimensional list including the desired number of pixels; the first and second integers of this list refer to the number of pixels in the directions of x- and y-axes, respectively.

```
poissonSOR2D[source2D_,size_]:=
    Module[{i,j,relax = 1.8,link,in,out,dim,
        dx,dy,interP},
    dim= Dimensions[source2D];
    dx=(1.*(dim[[2]]-1))/(size[[2]]-1);
    dy=(1.*(dim[[1]]-1))/(size[[1]]-1);
    interP=ListInterpolation[source2D];
    in=Table[interP[i,j],
        {i,1,dim[[1]],dy},{j,1,dim[[2]],dx}];
    link = Install["Sor04.exe"];Pause[0.01];
    out=Sor[in,relax];
    Uninstall[link];
    out];
```

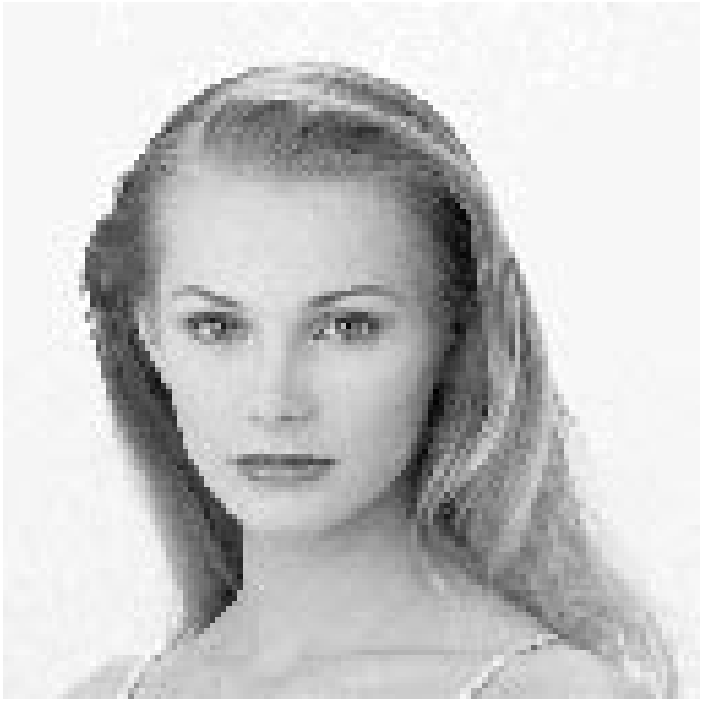
Figure 8 shows a recovered monochrome image whose numerical values are contained in a list “recoverMono”. A “Timing” command of *Mathematica* gives a CPU time used in the computations.

```
recoverMono=
    poissonSOR2D[-source,Dimensions[source]];//Timing
```

```
{27.94 Second, Null}
```

```
ListDensityPlot[recoverMono,PlotRange->All,Mesh->False,
    Frame->False,PlotLabel->"Fig.8. Recovered image"];
```

Fig.8. Recovered image



To check up the validity of solution, we print out the maximum difference between the original and recovered image data. Consequently, it is revealed that our solution has a good accuracy.

```
Max[Abs[monoSample-recoverMono]]
```

4.10691×10^{-7}

2.8 Image resolution

The image governing equation has been successfully solved with good accuracy. This mean when we change a number of pixels representing a recovered image then it is possible to change a resolution of the image.

Using this function “poissonSOR2D”, we produce the two images whose resolutions are lower (100 by 100) and higher (150 by 150) than those of the original 128 by 128 image.

```
low=poissonSOR2D[-source,{100,100}];//Timing
```

```
{10.75 Second, Null}
```

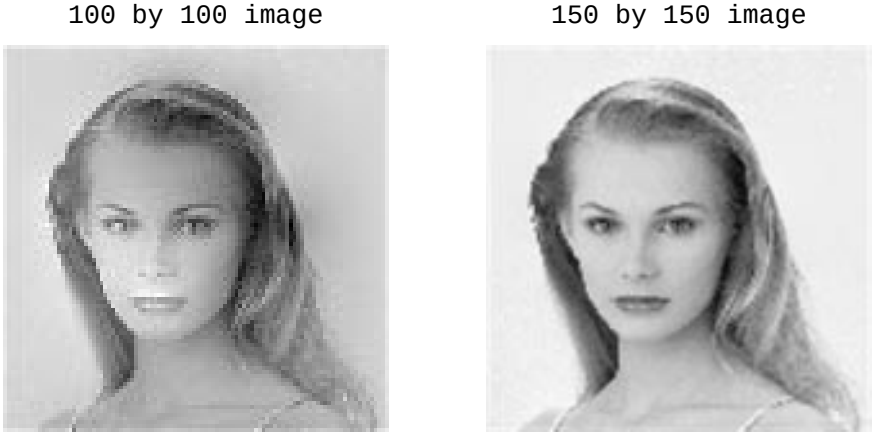
```
high=poissonSOR2D[-source,{150,150}];//Timing
```

```
{45.29 Second, Null}
```

The left and right in Fig. 9 show the images having 100 by 100 and 150 by 150 resolutions, respectively.

```
lowG=ListDensityPlot[low,PlotRange->All,Mesh->False,
    Frame->False,PlotLabel->"100 by 100 image",
    DisplayFunction->Identity];
highG=ListDensityPlot[high,PlotRange->All,Mesh->False,
    Frame->False,PlotLabel->"150 by 150 image",
    DisplayFunction->Identity];
Show[GraphicsArray[{lowG,highG}],ImageSize->{400,200},
    PlotLabel->"Fig.9. Low and high resolution images"];
```

Fig.9. Low and high resolution images



Because of the discretization error, the low-resolution image on the left in Fig. 8 includes a little bits of noise. However, observation of the images in Fig.8 suggests that the image governing equation makes it possible to reproduce the images having any resolutions.

2.9 Illusive image generation

Comparison the source density image in Fig.7 with the shadowed relief images in Fig.5 reveals that the angled lighting images are very similar to those of source density. This is because an inner product operation between the vectors changes the vectors into the scalar quantities.

Let us try to solve a Poisson's equation for the inner product images in Fig.5 as the source densities, and then we draw the images from the obtained solutions.

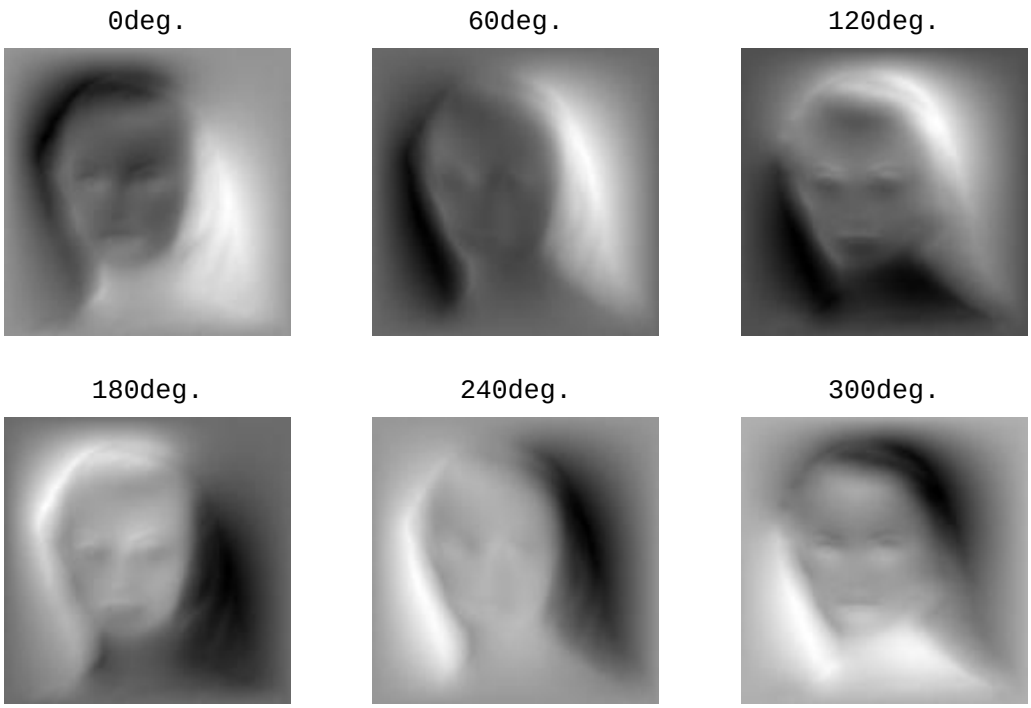
```
illusion=Table[poissonSOR2D[-angle[[i]],{128,128}],
    {i,rotation}];
```

Figure 10 shows the images derived from the inner product image data “angle”. Thus, when we regard the inner product image data as the source density, then the solutions of the Poisson's equation for these input source densities produce the illusive images emphasizing the lighting effects. Further, the illusive images in Fig. 9 are smooth and beautiful ones compared with these in Fig. 5. Solving the Poisson's equation is one of the surface integrations so that small noisy components are smoothed.

```
illusionG=Table[ListDensityPlot[illusion[[i]],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["`1`deg.",(i-1)360/rotation],
    DisplayFunction->Identity],
    {i,rotation}];
```

```
Show[GraphicsArray[
    Table[Table[illusionG[[i+j]],{j,0,2}],{i,1,rotation,3}]],
    ImageSize->{450,300},PlotLabel->"Fig.10. Illusive images"];
```

Fig.10. Illusive images



After setting a mouse cursor on a following image, twice clicking the left side button of mouse animates the facial changes depending on the lighting angles. This is only effective on the *Mathematica* notebook.

```
Do[Show[GraphicsArray[{illusionG[[i]]}]],{i,rotation}];
```

```
0deg.
```

2.10 Summary

This chapter has described about the image processing techniques based on the classical field theory. As a result, it has been shown that the vectored approach not the simple spatial derivatives leads to the fruitful results. For example, the characteristic vector distribution has been derived by means of the Helmholtz's theorem. Simple magnitude computation of the characteristic vectors has yielded the sketch image. Further, an application of vector field nature to the characteristic vectors of an image has worked out one of the angled lighting images. Namely, we have worked out the unit directional vectors, and then a set of inner products between the characteristic and unit vectors has yielded the images as if lighted up from the different directions.

The other important result of this chapter is that we have derived the static monochrome image governing equation. This governing equation is one of the Poisson's equations, which is able to produce the different resolution images, freely.

■ REFERENCES

[1] Stephen Wolfram, *The Mathematica Book*, 3rd ed. (Wolfram Media/Cambridge University Press, 1996).

[2] J.D.Jackson, "Classical Electrodynamics 3rd Edition," John Wiley & Sons, New York (1998).

Chapter 3. Color Image Processing

3.1 Introduction

In chapter 2, we have described the basic principle and practical examples of the field theory of computer graphics. This chapter extends the basic principles of the field theory to the color graphics images. In the monochrome images, we have only a set of numerical values representing an monochrome image. Thereby, in order to derive the image vectors, it is essential to use the partial derivative operators, such as gradient and curl. However, in the color graphics image, we have three independent components, i.e., red, green and blue color components. Thereby, it is possible to derive the image vectors without any spatial derivations. The first part of this chapter devotes to describe the nature and applications of these color image vectors.

In the second part, we describe the image processing techniques to generate the sketch and painted images. Two methodologies are considered for a monochrome image generation. One is to utilize a magnitude distribution of the color image vectors and the other is based on a distribution of the inner products between the unit directional and color image vectors. Combining the sketches of red, green and blue components generates a sketch image drawn by the color pencils. Each of the sketch components is generated by means of the characteristic vectors described in chapter 2. Combination of the painted red, green and blue component also generates a painted color image.

In third part, the high-resolution color images are generated from a low-resolution color image. The image governing equations of the red, green and blue components in a color graphic image are independently solved with higher resolution. Consequently, a composition of the solutions leads to the high-resolution color images. A correlation analysis between the generated and exact image data reveals that good high-resolution image can be generated by means of the differential equations.

Finally, similar to that of monochrome image, we generate the two-types of three-dimensional images. One is a shadowed lighting color image, and the other is an illusive image. Both images are based on the two-dimensional red, green and blue color characteristic vectors. The inner products between the unit directional and the color component characteristic vectors generate the shadowed lighting images of the red, green and blue color components. The convolutions between the shadowed and original color component images generate the shadowed lighting color images. The solutions of image governing equation regarding each of the shadowed lighting color component images as the source densities generate the illusive images. Because of the color effects, the impressive three-dimensional images can be obtained.

3.2 Preparation of *Mathematica*

■ 3.2.1 *Mathematica* utilities and packages

Before to move on the practical image processing, we have to install the memory conserve utilities and the warning message suppress command. In addition, the "LinearAlgebra`MatrixManipulation", "Graphics`PlotField" and "Graphics`PlotField3D" packages have to be installed. The last one is added in this chapter in order to plot the three-dimensional vectors [1].

Mathematica utilities and add-on packages

```
<<Utilities`MemoryConserve`
$MemoryIncrement=100000;
Off[General::spell1,MemoryConserve::start,MemoryConserve::end];
<< LinearAlgebra`MatrixManipulation`;
<<Graphics`PlotField3D`;
```

■ 3.2.2 *Mathematica* functions

Here, we define several functions that have been described and used in the previous chapters.

Function rgbBMP

The function "convertRGB" converts the numerical data representing color image into a RGB form. This makes it possible to visualize the color images on the *Mathematica* notebook. The parameter "colorData" refers to a list including the numerical values.

```
rgbBMP[colorFile_] :=
 Module[{i = 0, j = 0, k = 0, dim = {0}, input = {{{0.}}}},
  link = Install["RGBsplit.exe"]; Pause[0.01];
  input = RGBsplit[colorFile];
  Uninstall[link];
  Off[General::spell1];
  dim = Dimensions[input];
  Table[input[[i, j, k]], {k, 3},
   {i, dim[[1]]}, {j, dim[[2]]}]];
```

Function convertRGB

The function "convertRGB" converts the numerical data representing color image into a RGB form. This is used to visualize the image on the *Mathematica* notebook. The parameter "colorData" refers to a list including the numerical values.

```
convertRGB[colorData_] :=
 Module[{i = 0, j = 0, dim = {0}, out},
  dim = Dimensions[colorData];
  Graphics[
   RasterArray[
    Table[RGBColor[colorData[[1, i, j]],
     colorData[[2, i, j]], colorData[[3, i, j]]],
    {i, dim[[2]]}, {j, dim[[3]]}]]]];
```

Function grad2D

The function “grad2D” computes the divergent vectors of a monochrome image. The parameter “data2D” refers to a two-dimensional list including the numerical values regarded as the scalar potentials.

```
grad2D=Compile[{{data2D,_Real,2}},
    Module[{i=0,j=0,dim=Dimensions[data2D],
        c1={Table[data2D[[1,j]],{j,dim[[2]]}]},
        c2=Transpose[{Join[{data2D[[1,1]]},
            Table[data2D[[i,1]],{i,dim[[1]]}]]}],
        e2D=AppendRows[c2,AppendColumns[c1,data2D]]},
    Table[{-0.5(e2D[[i,j]]+e2D[[i+1,j]])+
            0.5(e2D[[i,j+1]]+e2D[[i+1,j+1]]),
            -0.5(e2D[[i,j]]+e2D[[i,j+1]])+
            0.5(e2D[[i+1,j]]+e2D[[i+1,j+1]])},
            {i,dim[[1]],1,-1},{j,dim[[2]]}]];
```

Function curl2D

The “curl2D” function generates the rotational vectors. A parameter “darta2D” refers to a two-dimensional list containing one of the vector potential components Both of the functions “grad2D” and “curl2D” are based on the central finite differences.

```
curl2D=Compile[{{data2D,_Real,2}},
    Module[{i=0,j=0,dim=Dimensions[data2D],
        c1={Table[data2D[[1,j]],{j,dim[[2]]}]},
        c2=Transpose[{Join[{data2D[[1,1]]},
            Table[data2D[[i,1]],{i,dim[[1]]}]]}],
        e2D=AppendRows[c2,AppendColumns[c1,data2D]]},
    Table[{-0.5(e2D[[i,j]]+e2D[[i,j+1]])+
        0.5(e2D[[i+1,j]]+e2D[[i+1,j+1]]),
        0.5(e2D[[i,j]]+e2D[[i+1,j]])-
        0.5(e2D[[i,j+1]]+e2D[[i+1,j+1]])},
    {i,dim[[1]],1,-1},{j,dim[[2]]}]];
```

Function eigen2DVect

The “eigen2DVect” function is given by a summation of divergent and rotational vectors. The parameter “monoSample” is a two-dimensional list regarded as either scalar or one component of vector potentials.

```
eigen2DVect[monoSample_]:=Module[{out},
    out=-grad2D[monoSample]+curl2D[monoSample];out]
```

Function vectorMag2D

The “vectorMag2D” function computes the magnitudes of the two-dimensional vectors including a list “vector2D”
.

```
vectorMag2D=Compile[{{vector2D,_Real,3}},
    Module[{i=0,j=0,dim=Dimensions[vector2D]},
        Sqrt[Table[vector2D[[i,j,1]]^2+vector2D[[i,j,2]]^2,
        {i,dim[[1]],1,-1},{j,dim[[2]]}]]];
```

Function laplace2D

The “laplace2D” function carries out the Laplacian operation to the input data listed in “data2D”. This operation is based on the 9 points finite difference formula assuming the zero Dirichlet boundary condition along with an image enclosing line.

```
laplace2D=Compile[{{data2D,_Real,2}},
    Module[{e2D={{0.}},out={{0.}},dim={0},
        max=0.,i=0,j=0,nx=0,ny=0},
            dim=Dimensions[data2D];
            ny=dim[[1]]+1;nx=dim[[2]]+1;
            e2D=Table[0.,{ny+1},{nx+1}];
            Do[e2D[[i,j]]=data2D[[i-1,j-1]],
                {i,2,ny},{j,2,nx}];
            Table[e2D[[i+1,j-1]]+4.*e2D[[i+1,j]]+
                e2D[[i+1,j+1]]+4.*e2D[[i,j-1]]-
                20.*e2D[[i,j]]+4.*e2D[[i,j+1]]+
                e2D[[i-1,j-1]]+4.*e2D[[i-1,j]]+
                e2D[[i-1,j+1]],
            {i,2,ny},{j,2,nx}]/6.]];
```

Function poissonSOR2D

The “poissonSOR2D” function gives an iterative solution of the Poisson's equation. A discretizing strategy used in this code is a 9 point finite difference formula, and an over relaxation method solves a system of linear equations. In order to get a solution as soon as possible, a MathLink utility is employed. A command “Pause” is essential for establishing a reliable linkage between the *Mathematica* kernel and external object.

The function “poissonSOR2D has two parameters. One is the source density. Also, the other is a one-dimensional list including the desired number of pixels; the first and second integers of this list refer to the number of pixels in the directions of x- and y-axes, respectively.

```
poissonSOR2D[source2D_,size_]:=
    Module[{i,j,relax = 1.8,link,in,out,dim,
        dx,dy,interP},
    dim= Dimensions[source2D];
    dx=(1.*(dim[[2]]-1))/(size[[2]]-1);
    dy=(1.*(dim[[1]]-1))/(size[[1]]-1);
    interP=ListInterpolation[source2D];
    in=Table[interP[i,j],
        {i,1,dim[[1]],dy},{j,1,dim[[2]],dx}];
    link = Install["Sor04.exe"];Pause[0.01];
    out=Sor[in,relax];
    Uninstall[link];
    out];
```

Function imageNormalize

This function “imageNormalize” converts the two-dimensional image data between the values 0 and 1.

```
imageNormalize = Compile[{{data2D, _Real, 2}},
  Module[{minimum = Min[data2D]},
   (data2D - minimum) / Max[data2D - minimum]]];
```

Function hueColor

The “hueColor” function provides a customized hue color function. This function gives a color pattern only depending on an absolute value of “z”. The graphics objects “colorBar10” and “colorBar5” are the color tables.

```
hueColor[z_]:=Hue[Abs[z],Abs[z],1];
colorBar10=ListDensityPlot[Table[i,{2},{i,0.,1,0.1}],
    Frame->False,Mesh->False,AspectRatio->0.1,
    ColorFunction->hueColor,DisplayFunction->Identity];
colorBar5=ListDensityPlot[Table[i,{2},{i,0.,1,0.2}],
    Frame->False,Mesh->False,AspectRatio->0.1,
    ColorFunction->hueColor,DisplayFunction->Identity];
```

3.3 Sample color image

In this section, we have to input a sample color image. To demonstrate a distribution of color characteristic vectors and its applicability to a nondestructive testing of the concrete cracks, we read a color image file “Wall-A-128.bmp” by the function rgbBMP.

```
sample=rgbBMP["Wall-A-128.bmp"];
```

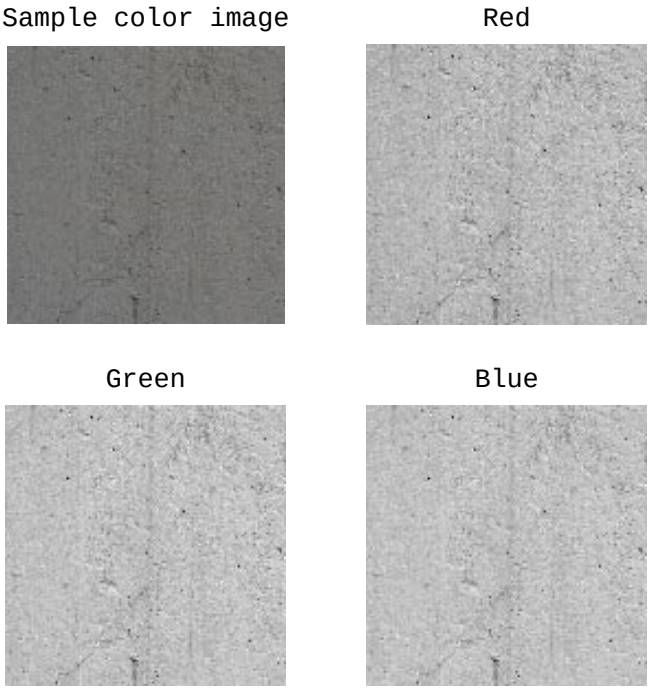
After computing a resolution of the input image, we construct its color image.

```
dim=Dimensions[sample];
sampleG=Show[convertRGB[sample],
    PlotLabel->"Sample color image",
    AspectRatio->dim[[2]]/dim[[3]],
    DisplayFunction->Identity];
```

Further we construct the monochrome images of red, green and blue components. Figure 1 shows the input color image and its components.

```
color={"Red","Green","Blue"};
colorCompG=Table[ListDensityPlot[sample[[i]],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["`1`",color[[i]]],
    AspectRatio->dim[[2]]/dim[[3]],
    DisplayFunction->Identity],{i,3}];
Show[
    GraphicsArray[
        {{sampleG,colorCompG[[1]]},
        {colorCompG[[2]],colorCompG[[3]]}}],
    PlotLabel->"Fig.1. Sample and its color components"];
```

Fig.1. Sample and its color components

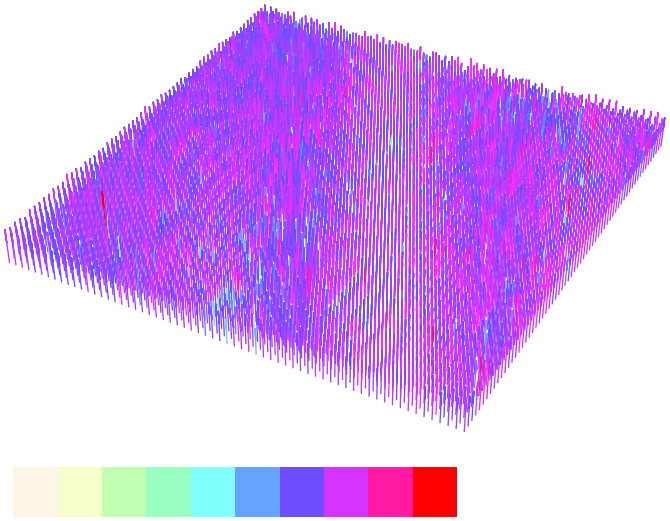


3.4 Color characteristic vectors

Color image is always composed of the three-color components depending on a wavelength of light [2]. The longest, middle and shortest wavelengths are corresponding to the red, green and blue color components, respectively. Since a classification of the color components is based on this fact, then project the red, green and blue respectively to the x-, y- and z-axes components of a Cartesian coordinate system yields a set of three-dimensional vectors. This defines the three-dimensional color image characteristic vectors. Figure 2 shows the color characteristic vectors of the sample image in Fig. 1.

```
vectored3D=Flatten[Table[{{j,i,k},(0.1/Sqrt[3.])*
        {sample[[1,i,j]],sample[[2,i,j]],sample[[3,i,j]]}},
        {i,1,dim[[2]],2},{j,1,dim[[3]],2},{k,1}],2];

ListPlotVectorField3D[vectored3D,
    VectorHeads -> True,ColorFunction->hueColor,
    PlotLabel->"Fig.2. Color vectors",
    BoxRatios -> {1, 1, 0.1},Boxed->False];
Show[GraphicsArray[{colorBar10}],ImageSize->{200,20}];
```

Fig.2. Color vectors



Mathematica function VectorMag3D

Since each of the vector lengths corresponds to light reflection strength, a comparison between the vector distribution and color bar reveals that a light reflection strength of the wall is not uniform but somewhat random nature. Reader may suppose that this sample concrete wall has several cracks. In order to confirm this, we compute a distribution of vector magnitudes in Fig. 2 by a following *Mathematica* function “vectorMag3D”. A parameter “vectorData3D” refers to the three-dimensional vector data.

```
vectorMag3D=Compile[{{vectorData3D,_Real,3}},
    Module[{mag={{0.}},max=0.,min=0.,i=0,j=0,dim={0}},
        dim=Dimensions[vectorData3D];
        mag=Sqrt[Table[vectorData3D[[1,i,j]]^2+
            vectorData3D[[2,i,j]]^2+
            vectorData3D[[3,i,j]]^2,
            {i,dim[[2]]},{j,dim[[3]]}]];
        min=Min[mag];max=Max[mag-min];
        If[max=!=0.,Return[(mag-min)/max],
            Print["Zero vectors !"];
        Return[mag]]]];
```

Using this function, we compute the magnitude of the characteristic vectors in Fig. 1.

```
vectMag=vectorMag3D[sample];
```

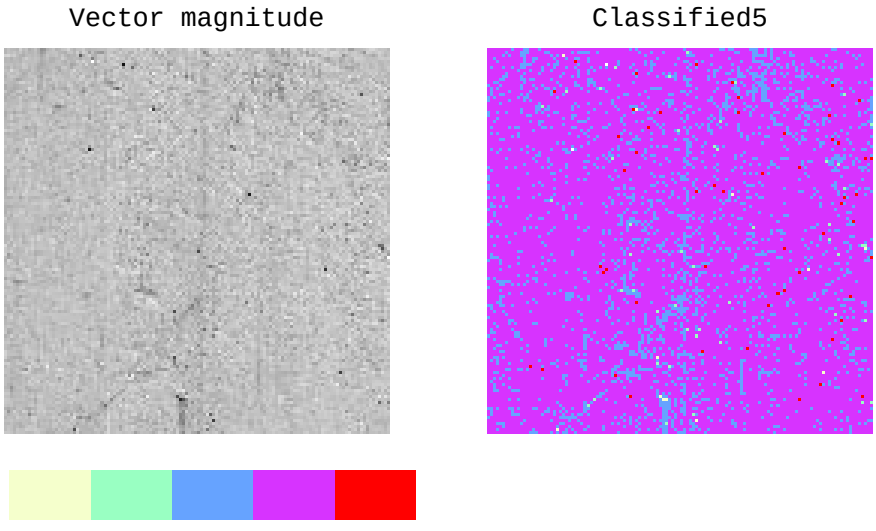
After that, we construct an image data of the vector magnitude distribution.

```
vectMagG=ListDensityPlot[vectMag,
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->"Vector magnitude",
    DisplayFunction->Identity];
```

Before simply plotting the vector magnitude image, we classify the vector magnitude distribution into 5 levels by their magnitude, and then we construct the classified image data. Figure 3 shows the classified vector magnitude distributions.

```
nx=5;
vectMagCG=ListDensityPlot[1. Round[nx vectMag]/nx,
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["Classified``",nx],
    ColorFunction->hueColor,DisplayFunction->Identity];
Show[GraphicsArray[{vectMagG,vectMagCG}],
    PlotLabel->"Fig.3. Vector magnitude distribution",
    ImageSize->{400,200}];
Show[GraphicsArray[{colorBar5}],ImageSize->{400,20}];
```

Fig.3. Vector magnitude distribution



A simple vector magnitude image on the left in Fig. 3 does not extract the cracks clearly, but the blue lines shows the cracks in the classified image on the right. To extract the cracks more clearly, we use the other vector property "inner product" between the unit directional and characteristic vectors. Defining a following *Mathematica* function "innerPro3D" carries out a three-dimensional inner product computation.

Mathematica function innerPro3D

```
innerPro3D=Compile[{{vector3D,_Real,3}},
    Module[{ref={0.},vec={0.},dim={0},i=0,j=0},
        ref={1.,1.,1.}/Sqrt[3.];
        dim=Dimensions[vector3D];
        Table[
            vec={vector3D[[1,i,j]],vector3D[[2,i,j]],
                vector3D[[3,i,j]]};ref.vec,
        {i,dim[[2]]},{j,dim[[3]]}]
    ]];
```

We compute an inner product distribution between the unit directional and characteristic vectors in Fig. 2 by means of this function.

```
innerPro=innerPro3D[sample];
```

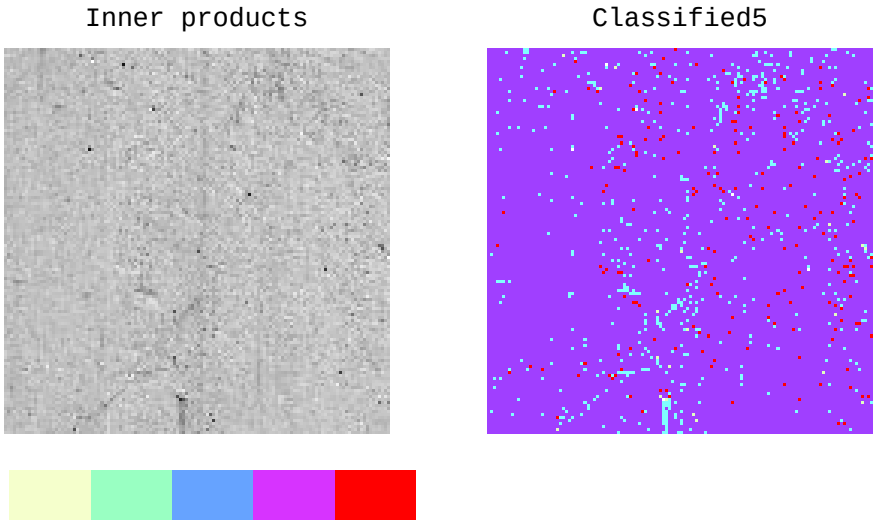
We construct an image data of the inner products, and then we classify the inner products into 5 levels by the magnitude.

```
innerProG=ListDensityPlot[innerPro,
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->"Inner products",DisplayFunction->Identity];
```

Figure 4 shows the inner product distributions. A simple inner product distribution on the left in Fig.4 does not extract the cracks but the classified distribution on the right clearly extracts the cracks as the light blue dotted lines.

```
innerProCG=ListDensityPlot[1. Round[nx innerPro]/nx,
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["Classified``",nx],
    ColorFunction->hueColor,DisplayFunction->Identity];
Show[GraphicsArray[{innerProG,innerProCG}],
    PlotLabel->"Fig.4. Inner product distribution",
    ImageSize->{400,200}];
Show[GraphicsArray[{colorBar5}],ImageSize->{400,20}];
```

Fig.4. Inner product distribution



Thus, it is obvious that the vectored representation of color image is useful methodology for the nondestructive inspections.

3.5 Sketch and painted image generation

As described in chapter 2, the sketch is one of the human art works. However, we have succeeded in work out a plausible monochrome sketch by computer graphics with the aid of field theory.

In this section, we propose the two methodologies to work out the sketch from a color graphics image. Both methods are based on the nature of color characteristic vectors. The distributions of the vector magnitude and of the inner products gives the monochrome images. Observing the left side images in Figs 3 and 4 easily recognizes this. Applying the method developed to a monochrome image in chapter 2 to these images generates the sketch images.

Before to move on the next computations, we remove the needless memories from the *Mathematica* front end processor and check up the used memories.

```
Remove["sampleG","colorCompG","vectored3D","vectMag",
    "vectMagG","nx","sample","dim"];
Unprotect[In,Out];Clear[In,Out];Protect[In,Out];
Print["Memory: ",Round[N[MemoryInUse[]/1000]],"K Bytes used"];
```

```
Memory: 2158K Bytes used
```

We read in the image file “AF038-256.bmp” by means of the function rgbBMP, and compute the size of a list “sample”.

```
sample=rgbBMP["AF038-256.bmp"];
dim=Dimensions[sample];
```

Figure 5 shows a sample image.

```
Show[convertRGB[sample],PlotLabel->"Fig.5. Color image",
    AspectRatio->dim[[2]]/dim[[3]]];
```

Fig.5. Color image

■ 3.5.1 Monochrome sketch

We compute a magnitude distribution of the color characteristic vectors as well as an inner product distribution between the unit directional and characteristic vectors of Fig. 5. Further we compute both of the image data.

```
vectMagG=ListDensityPlot[vectMag=vectorMag3D[sample],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->"By vector magnitude",
    DisplayFunction->Identity];
innerProG=ListDensityPlot[innerPro=innerPro3D[sample],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->"By inner product",DisplayFunction->Identity];
```

Figure 6 shows the monochrome images.

```
Show[GraphicsArray[{vectMagG,innerProG}],
    ImageSize->{400,200},
    PlotLabel->"Fig.6. Monochrome images"];
```

Fig.6. Monochrome images

By vector magnitude

By inner product

At first, we compute the monochrome characteristic vectors to the both images in Fig.6 by the function "eigen2Dvect". Second, we compute the magnitude distributions of the monochrome characteristic vectors by the function "vectorMag2D". Consequently, we compute the image data reversing the black and white mode.

```
monoMG=ListDensityPlot[1-vectorMag2D[eigen2DVect[vectMag]],
    Mesh->False,Frame->False,PlotRange->All,
    PlotLabel->"By vector magnitude",
    DisplayFunction->Identity];
monoIG=ListDensityPlot[1-vectorMag2D[eigen2DVect[innerPro]],
    Mesh->False,Frame->False,PlotRange->All,
    PlotLabel->"By inner product",
    DisplayFunction->Identity];
```

Figure 7 shows the monochrome images.

```
Show[GraphicsArray[{monoMG,monoIG}],ImageSize->{400,200},
    PlotLabel->"Fig.7. Monochrome sketches"];
```

Fig.7. Monochrome sketches

By vector magnitude

By inner product

Thus, we have succeeded in drawing the monochrome sketch images. Both images on the right and left in Fig.7 are based on the three-dimensional color as well as two-dimensional monochrome characteristic vector natures.

■ 3.5.2 Color sketch

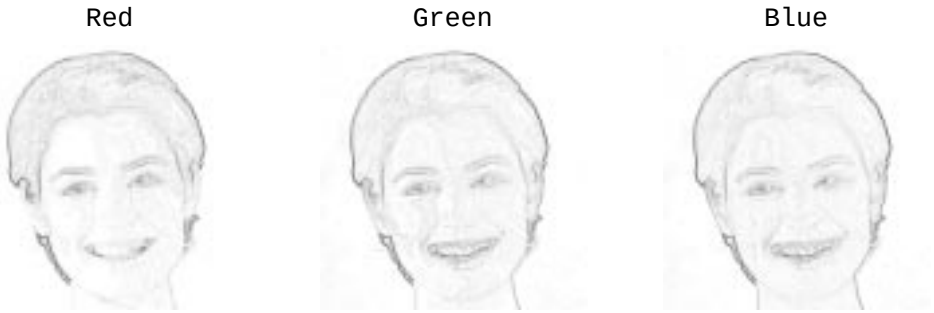
In order to draw a sketch by color pencils, it is essential to use the red, green and blue color components of the image shown in Fig. 5. In addition, the monochrome sketches to the red, green and green color components are required. These processes are carried out by a following computation.

```
colorSketchData=
    Table[1-vectorMag2D[eigen2DVect[sample[[i]]],{i,dim[[1]]}];
```

Figure 8 shows the sketches of the red, green and blue components in Fig. 5.

```
Show[GraphicsArray[
    Table[ListDensityPlot[colorSketchData[[i]],
        PlotRange->All,Mesh->False,Frame->False,
        PlotLabel->StringForm["``",color[[i]]],
        DisplayFunction->Identity],{i,dim[[1]]}]],
    PlotLabel->"Fig.8. Components of color sketch",
    ImageSize->{450,150}];
```

Fig.8. Components of color sketch



After normalizing the image data in Fig. 8 between the values of 0 and 1, we convert the red, green and blue color sketch data into a color image data by means of the function “convertRGB”. Consequently, we can obtain a color sketch as shown in Fig. 9.

```
normalizedCSD=Table[
    imageNormalize[colorSketchData[[i]]],{i,dim[[1]]}];
Show[convertRGB[normalizedCSD],
    PlotLabel->"Fig.9. Color sketch",
    AspectRatio->dim[[2]]/dim[[3]]];
```

Fig.9. Color sketch

Thus, it is obvious that the field theory makes it possible to draw the monochrome as well as color sketch images.

■ 3.5.3 Painted image generation

Sometimes, it is required a painted image as if painted by artist. In order to generate such an image, this section describes a simple methodology. A key idea is to emulate a paint touch imaging by classifying the numerical data into a limited number of gropes depending on their magnitudes. Computer graphics is able to draw any high-resolution images, but the painted image by artist is essentially composed of a limited number of color components. Thereby, classification of the numerical data into a limited number of gropes depending on their magnitudes makes it possible to emulate a paint touch.

At first, we classify the numerical data into 5 gropes depending on their magnitudes.

```
level=5.;
pSample=Table[Round[level*sample[[i]]/level,{i,dim[[1]]}];
```

Figure 10 shows the monochrome images classified into 5 groups.

```
Show[GraphicsArray[
    Table[ListDensityPlot[pSample[[i]],
        PlotRange->All,Mesh->False,Frame->False,
        PlotLabel->StringForm["``",color[[i]]],
        DisplayFunction->Identity],{i,dim[[1]]}]],
    PlotLabel->"Fig.10. Components of color painted image",
    ImageSize->{450,150}];
```

Fig.10. Components of color painted image

Red Green Blue

Combination of the red, green and blue components in Fig. 10 yields a painted image as shown in Fig. 11.

```
Show[convertRGB[pSample],
    PlotLabel->"Fig.11. Painted color image",
    AspectRatio->dim[[2]]/dim[[3]]];
```

Fig.11. Painted color image

3.6 High-resolution image generation

In this section, we generate the high-resolution color images by means of the image governing equation, i.e. a Poisson type partial differential equation.

At first, we read in a low-resolution color image. Second, we derive the source densities of the red, green and blue components composing the low-resolution color image. Third, we solve each of the partial differential equations having the red, green and blue component source density inputs, independently. Setting the high-resolution conditions generates the high-resolution red, green and blue component images. Finally, composition of the generated high-resolution color components gives a high-resolution color image. We generate two high-resolution images from the one low-resolution image. Recoverability is checked up by the correlation coefficients between the generated and original color image data.

Before to continue the computation, we remove the needless memories by a following command and check up the used memories.

```
Remove["vectMagG","innerProG","monoMG","monoIG","colorSketchData",
    "normalizedCSD","level","pSample"];
Unprotect[In,Out];Clear[In,Out];Protect[In,Out];
Print["Memory: ",Round[N[MemoryInUse[]/1000]],"K Bytes used"];
```

```
Memory: 4578K Bytes used
```

More memories than the previous session were used. This means that the response speed of *Mathematica* front-end processor becomes slow.

In order to obtain the low-resolution source densities, we have to read in a low-resolution color image data “AF038-64.bmp” whose image is shown in Fig. 12.

```
sampleL=rgbBMP["AF038-64.bmp"];
dim=Dimensions[sampleL];
Show[convertRGB[sampleL],PlotLabel->StringForm[
    "Fig.12.Original`1` by `2`pixles image",dim[[2]],dim[[3]]],
    AspectRatio->dim[[2]]/dim[[3]]];
```

```
Fig.12.Original64 by 64pixles image
```

We compute the source densities regarding the red, green and blue components as the scalar potentials. This is carried out by means of the function “laplace2D”.

```
source=-Table[laplace2D[sampleL[[i]]],{i,dim[[1]]}];
```

We solve the Poison's type partial differential equations to generate the 128 by 128 and 256 by 256 resolution images. It must be noted that a large CPU time is required for generating a higher-resolution image. Further, because of the numerical errors, the solutions are essentially containing a little bit of errors. Normalize the solutions reduces the effects of numerical errors.

The first 128 by 128 resolution image data are computed by a following simple command.

```
sampleH1=Table[imageNormalize[poissonSOR2D[source[[i]],
    {128,128}]],{i,dim[[1]]}];
```

Higher-resolution image generation requires longer CPU time, because higher resolution image generation is reduced into solving a larger linear system of equations. must be noted that a high-resolution color image generation by differential equations essentially requires a relatively large CPU time. A *Mathematica* command “Timing” used bellow gives a required CPU time in second.

```
sampleH2=Table[imageNormalize[poissonSOR2D[source[[i]],
    {256,256}]],{i,dim[[1]]}];//Timing
```

```
{1072.03 Second, Null}
```

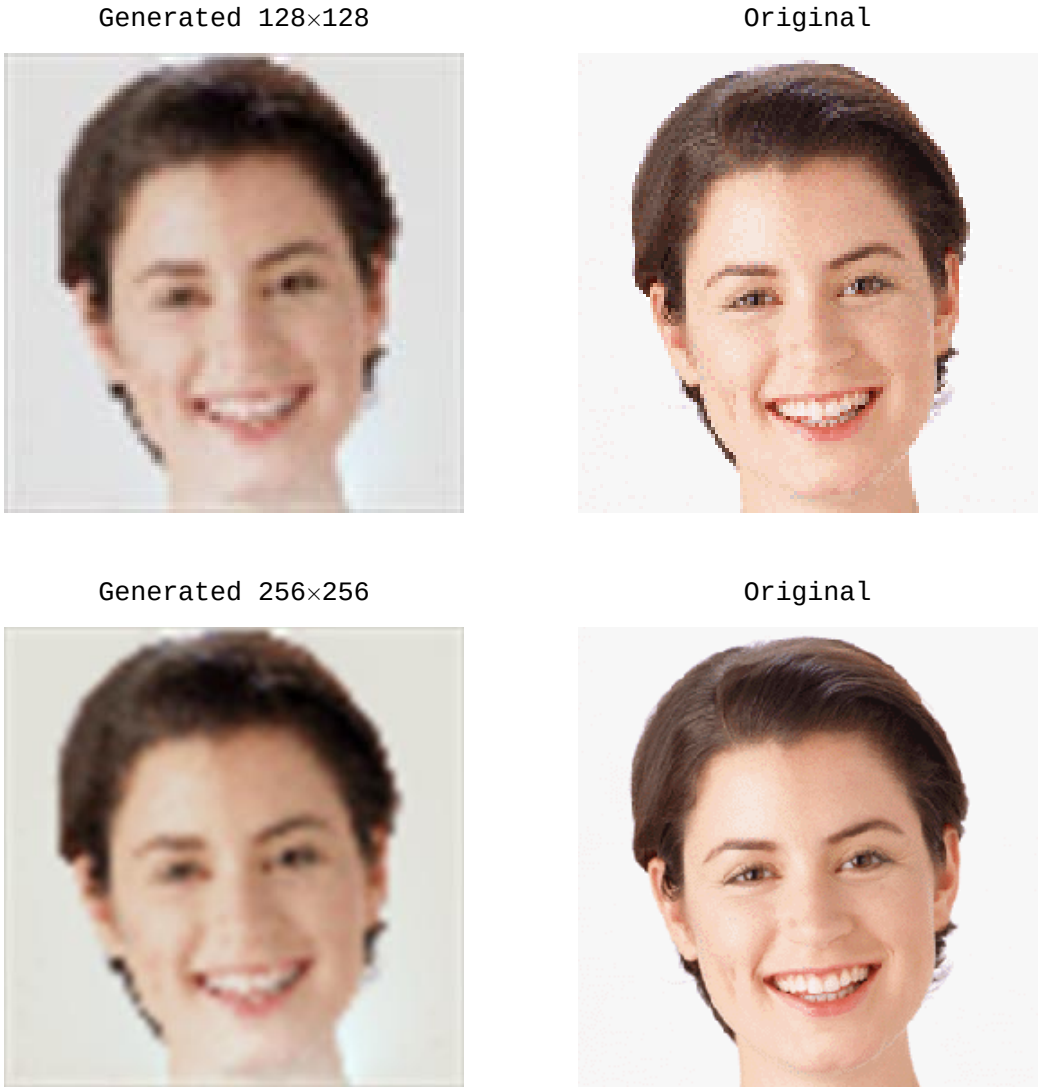
The normalized solutions are converted into the RGB color image data, and an original 128 by 128 resolution image data “AF-038-128.bmp” is read in a list “sampleH1C”.

```
sample1G=Show[convertRGB[sampleH1],PlotLabel->"Generated 128×128",
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity];
sample1CG=Show[convertRGB[sampleH1C=rgbBMP["AF038-128.bmp"]],
    PlotLabel->"Original",AspectRatio->dim[[2]]/dim[[3]],
    DisplayFunction->Identity];
sample2G=Show[convertRGB[sampleH2],PlotLabel->"Generated 256×256",
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity];
sample2CG=Show[convertRGB[sample],PlotLabel->"Original",
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity];
```

Figure 13 shows the obtained high-resolution images arranging with their original ones.

```
Show[GraphicsArray[{{sample1G,sample1CG},{sample2G,sample2CG}}],
    PlotLabel->"Fig.13. Original and generated images",
    ImageSize->{450,450}];
```

Fig.13. Original and generated images



Obviously, the generated high-resolution images are not so clear than these of original ones. However, their correlation coefficients between the generated and original image data are very good values.

Mathematica function corRelation

A *Mathematica* function “corRelation” for computing a correlation coefficient is given as follows. The lists “a” and “b” have to contain the real numerical values and be the same order one-dimensional array.

```
corRelation=Compile[{{a,_Real,1},{b,_Real,1}},
    Module[{aa={0.},bb={0.},av=0.,bv=0.},
    av=Apply[Plus,a]/Length[a]; bv=Apply[Plus,b]/Length[b];
    aa=a-av; bb=b-bv; aa.bb/(Sqrt[aa.aa]*Sqrt[bb.bb])]];
```

A correlation coefficient between the generated and original 128 by 128 images is given by

```
corRelation[Flatten[sampleH1C],Flatten[sampleH1]]
```

```
0.968677
```

Similarly, a correlation coefficient between the generated and original 256 by 256 images is given by

```
corRelation[Flatten[sample],Flatten[sampleH2]]
```

```
0.955257
```

Only the 6.25 percent data quantity has yielded over 95 percent recoverability. Thus, the differential equation method is a quite effective tool for generating the high-resolution images.

3.7 Three-dimensional color image generation

In this section, we generate two types of three-dimensional color images. One is the shadowed lighting three-dimensional image, and the other is the color illusive image. The principle is similar to that of monochrome ones in chapter 2. One difference between the monochrome and color images is that a monochrome three-dimensional image generation process should be repetitively applied to all of the color components red, green and blue.

Before to continue the computations, we remove the needless memories and check up them.

```
Remove["source","sampleH1","sampleH2","sample1G",
    "sampleH1C","sample2G","sample2CG"];
Unprotect[In,Out];Clear[In,Out];Protect[In,Out];
Print["Memory: ",Round[N[MemoryInUse[]/1000]],"K Bytes used"];
```

```
Memory: 5096K Bytes used
```

We define a *Mathematica* function “colorImage3D”. This function requires the function "eigen"2Dvect" and "imageNormalize". The former generates the two-dimensional characteristic vectors in each of the color components, and the latter converts the two-dimensional image data into the values between 0 and 1. The parameters "color" and "rotation" are the color image data and the number of lighting angles, respectively. An output of this function is a four-dimensional array. The first, second, third and fourth indices in this output array refer to the number of lighting angles, color components, y-axis pixels and x-axis pixels, respectively.

Mathematica function colorImage3D

The parameter “color” and “rotation” of “colorImage3D” are the list containing color image data and a integer giving a number of lighting angle subdivisions.

```
colorImage3D[color_,rotation_]:=
    Module[{view,dim,vector,dummy,sol,i,j,k,p},
        view=Table[{Cos[2.Pi i/rotation],Sin[2.Pi i/rotation]},
            {i,0,rotation-1}];
        dim=Dimensions[color];
        vector=Table[eigen2DVect[color[[i]]],{i,dim[[1]]}];
        dummy=Table[vector[[p,i,j]].view[[k]],
            {k,rotation},{p,dim[[1]]},
            {i,dim[[2]],1,-1},{j,dim[[3]]}];
        Table[(1-imageNormalize[dummy[[i,j]]])*color[[j]],
        {i,rotation},{j,dim[[1]]}]];
```

Using this function, we compute the shadowed lighting color image data.

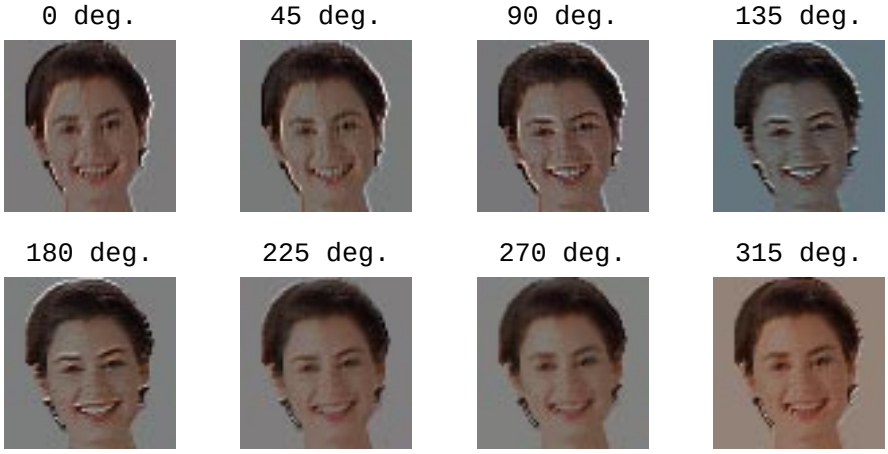
```
rotation=8;
color3D=colorImage3D[sampleL,rotation];//Timing
```

```
{5.93 Second, Null}
```

Figure 14 show a set of the shadowed lighting images. From the left to right images on the upper in Fig. 14 are corresponding to the 0, 45, 90 and 135-degree lighting angles, respectively. Similarly, from the left to right images on the lower are corresponding to the 180, 225, 270 and 315-degree lighting angles.

```
color3DG=Table[Show[convertRGB[color3D[[i]]],
    PlotLabel->StringForm["`1` deg.",
    (i-1) (360/rotation)],AspectRatio->dim[[2]]/dim[[3]],
    DisplayFunction->Identity],{i,rotation}];

Show[GraphicsArray[
    Table[Table[color3DG[[i+j]],{j,0,3}],{i,1,rotation,4}]],
    ImageSize->{400,200},PlotLabel->"Fig.14. 3D color images"];
```

Fig.14. 3D color images



After setting a mouse cursor on a following image, twice clicking the left side button of mouse animates the facial changes depending on the lighting angles. This is only effective on the *Mathematica* notebook.

```
Do[Show[GraphicsArray[{color3DG[[i]]}]],{i,rotation}];
```

```
0 deg.
```

3.8 Illusive color imaging

In this section, we carry out a color illusive imaging. The principle of the method is similar to those of monochrome illusive imaging in chapter 2. One difference between the monochrome and color images is that a monochrome illusive image generation process should be repetitively applied to all of the color components red, green and blue.

To improve a *Mathematica* front-end response speed, we remove the needless memories and check up them.

```
Remove["color3D","color3DG"];
Unprotect[In,Out];Clear[In,Out];Protect[In,Out];
Print["Memory: ",Round[N[MemoryInUse[]/1000]],"K Bytes used"];
```

```
Memory: 5138K Bytes used
```

We define a *Mathematica* function “illusion3D”. This function is based on a knowledge obtained in the monochrome illusive image generation processes, and requires using the functions “eigen2Dvect” and “poissonSOR2D”.

Mathematica function illusion3D

The parameter “color” and “rotation” of “illusive3D” are the list containing color image data and a integer giving a number of lighting angle subdivisions.

```
illusion3D[color_,rotation_]:=
    Module[{view,dim,vector,dummy,sol,i,j,k,p},
        view=Table[{Cos[2.Pi i/rotation],Sin[2.Pi i/rotation]},
            {i,0,rotation-1}];
        dim=Dimensions[color];
        vector=Table[eigen2DVect[color[[i]]],{i,dim[[1]]}];
        dummy=Table[vector[[p,i,j]].view[[k]],
            {k,rotation},{p,dim[[1]]},
            {i,dim[[2]],1,-1},{j,dim[[3]]}];
        sol=Table[poissonSOR2D[-dummy[[i,j]],
            {dim[[2]],dim[[3]]}],{i,rotation},{j,dim[[1]]}];
        Table[imageNormalize[sol[[i,j]]],{i,rotation},{j,dim[[1]]}]];
```

Using this function, we compute the illusive image data.

```
illusive=illusion3D[sampleL,rotation];//Timing
```

```
{47.65 Second, Null}
```

Figure 15 show a set of illusive images. From the left to right images on the upper in Fig. 15 are corresponding to the 0, 45, 90 and 135-degree lighting angles, respectively. Similarly, from the left to right images on the lower are corresponding to the 180, 225, 270 and 315-degree lighting angles.

```
illusiveG=Table[Show[convertRGB[illusive[[i]]],
    PlotLabel->StringForm["`1` deg.",(i-1) (360/rotation)],
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity],
    {i,rotation}];

Show[GraphicsArray[
    Table[Table[illusiveG[[i+j]],{j,0,3}],{i,1,rotation,4}]],
    ImageSize->{400,200},PlotLabel->"Fig.15. Illusive color images"];
```

Fig.15. Illusive color images

After setting a mouse cursor on a following image, twice clicking the left side button of mouse animates the facial changes depending on the lighting angles. This is only effective on the *Mathematica* notebook.

```
Do[Show[GraphicsArray[{illusiveG[[i]]}]],{i,rotation}];
```

```
0 deg.
```

3.9 Summary

This chapter has described about the color image processing techniques based on the linear vector space as well as classical field theory.

Color image data are always composed of the three independent color components, i.e. red, green and blue. This is a clear distinction to the monochrome image data. The color image characteristic vectors have been obtained by combining the color image components without any spatial derivatives. Even though an original image could be exactly recovered from its second spatial derivatives, i.e. source density, the characteristic vectors of monochrome image essentially had to be deduced by means of the spatial derivatives. A deduction of the characteristic vectors without any vector operations is an extremely significant. The color image characteristic vectors are corresponding to the reflected light intensity vectors. This means that the characteristic vectors of color image have a firm physical background, so that it may become a useful tool for inspection, identification and cognition. In this chapter, we have demonstrated as a tool of nondestructive testing.

In chapter 2, we described the basic principle and practical examples of the field theory of computer graphics. This chapter has extended the basic principles of the field theory to the color graphics images.

The first part has devoted to describe about the nature and application of this color image vector field.

In the second part, we have described the image processing techniques to generate the sketch and painted images. Two methodologies have been taken into account for the monochrome image generation. One has utilized a magnitude distribution of the color characteristic image vectors, and the other has been based on a distribution of the inner products between the unit directional and color image vectors. Combination of the sketch data of red, green and blue components has generated a colored sketch image. Each of the sketch components has been generated by means of the two-dimensional monochrome image characteristic vectors described in chapter 2. The color painted image has been obtained by combining the painted image data of the red, green and blue components.

In the third part, the high-resolution color images have been generated from a low-resolution color image. Each of the governing equations concerning with the red, green and blue components has been independently solved with higher resolution, and then a combination of their solutions has led to the high-resolution color images. A correlation analysis between the generated and exact image data has suggested that good high-resolution color images can be generated by the method of differential equations.

Finally, we have generated the two types of three-dimensional color images. One is the shadowed lighting color image, and the other is the illusive color image. Because of color, both three-dimensional color images have been more impressive to that of three-dimensional monochrome images.

■ REFERENCES

[1] Stephen Wolfram, *The Mathematica Book*, 3rd ed. (Wolfram Media/Cambridge University Press, 1996).

[2] J.D.Jackson, "Classical Electrodynamics 3rd Edition," John Wiley & Sons, New York (1998).

Chapter 4. Wavelet Image Processing

4.1 Introduction

This chapter introduces the applications of the wavelet transform to the image data. As described in the previous chapters, we have three types of image data. The first represents the static monochrome image, the second represents the color image and the third is the three-dimensional shadowed lighting image. The monochrome image data are housed in the two-dimensional arrays. Also, the monochrome three-dimensional shadowed lighting image data are housed in the two-dimensional arrays. The color image data are housed in the three two-dimensional arrays, which house the red, green and blue color components. Similarly, the color three-dimensional shadowed lighting image data are housed in the three two-dimensional arrays. Thus, it looks like to use the two-dimensional wavelet transform for the image data processing. However, consideration of the time depended images, i.e. animation image data, leads to use the three-dimensional wavelet transform. The monochrome animation image data are essentially housed in the three-dimensional arrays. Further, the color animation image data have to be housed by the three three-dimensional arrays.

The first section of this chapter describes about the discrete orthogonal wavelet transform, which employ the Daubechies, Coifman and Baylkin's base functions. Details of the discrete orthogonal wavelet are not described but the n^{th} dimensional wavelet transform *Mathematica* code is described.

The second section describes the monochrome image data compression and expansion by the wavelet transform. This section is an introduction of the wavelet image processing. Also, this section demonstrates that Mathematically remarkable image compression rate is possible by the wavelet transform.

The third section concerns with the color image compression and recovery. It is revealed that the compression rate and recoverability depend on the order of base function.

The fourth section proposes one of the orthogonal color image decomposition. In chapter 3, we have projected the color image components red, green and blue, to the x-, y-, and z-axes in the Cartesian coordinate, respectively. In this section, the color image data are represented in the spherical coordinate. The magnitude of color image characteristic vector corresponds to a radius. The attitude and longitude are represented in terms of their directional co-sinusoidal components. In continuation to this section, the fifth section describes about the wavelet compression and recovery to the color image data represented in terms of the spherical coordinate quantities. The sixth section reveals that the color image data compression rate by the wavelet transform depends on the way of image data representations. Namely, higher recoverability can be achieved by the spherical coordinate representation.

The seventh and final sections are concerning with the expansion of a small number of color animation data to a large number of ones by the wavelet transform. This demonstrates that a three-dimensional wavelet transform makes it possible to increase the number of animation data.

4.2 Preparation of *Mathematica*

■ 4.2.1 *Mathematica* utilities and packages

Before to move on the practical image processing, we have to install the memory conserve utilities and the warning messages suppressing command for the similar variable name. In addition, the "LinearAlgebra`MatrixManipulation" and "Graphics`ContourPlot3D" packages have to be installed.

Mathematica utilities and add-on packages

```
<<Utilities`MemoryConserve`
$MemoryIncrement=100000;
Off[General::spell1,MemoryConserve::start,MemoryConserve::end];
<< LinearAlgebra`MatrixManipulation`;
<<Graphics`ContourPlot3D`;
```

■ 4.2.2 *Mathematica* functions

Here, we define several functions that have been described and used in the previous chapters. The functions "rgbBMP", "convertRGB", "grad2D", "curl2D", "eigen2DVect", "vectorMag2D", "laplace2D", "poissonSOR2D", "imageNormalize" and "hueColor" have been defined in the previous chapters, so that the comments of such functions are not described.

However, the functions "vectorMag3D", "corRelation", "colorImage3D" "illusion3D" and "momoryUsed" are newly defined in this chapter, so that their comments will be described.

Function rgbBMP

Function convertRGB

Function grad2D

Function curl2D

Function eigen2DVect

Function vectorMag2D

Function laplace2D

Function poissonSOR2D

Function imageNormalize

Function hueColor

Function VectorMag3D

Function corRelation

Function colorImage3D

Function illusion3D

Function memoryUsed

■ 4.2.3 *Mathematica* functions for the discrete wavelet transform

In this *Mathematica* notebook includes the discrete orthogonal base function of "Daubechies 2-20th", "Coifman 6-30th" and "Baylkin 6th-18th". All of the coefficients are included in the *Mathematica* notebook, but are not explicitly described in the text. If you desire to see such coefficients, open the *Mathematica notebook.*

Wavelets base functions

Wavelets transform matrix

This function gives a wavelet transform matrix with order "n" by "n" using a base function "baseType". The parameter "n" must be the integer and the power of 2. For example, a 4 by 4 wavelet transform matrix using the Daubechies 2nd order base function is obtained by "waveletMatrix[4,daub4]".

```
waveletMatrix[n_,baseType_]:=
    Apply[Dot,Flatten[Table[{pPrime[j,n,baseType],
        cPrime[j,n,baseType]},{j,Log[2,n]-
        Ceiling[Log[2.,Length[baseType[[2]]]]],1,-1}],
        1]].p[n].c[n,baseType];
```

Wavelets transform function

This function carries out the wavelet and inverse wavelet transforms to a “n”-th dimensional “data” using a transform matrix “wMat”. For example, if we wish to carry out a wavelet transform to a three-dimensional 32 by 32 by 32 array “data” using the base functions “daub2”, “daub4” and “daub8”, then a waveletpectrum “wSpect” is obtained by the following steps.

1) Create the transform matrices by

```
wMat={waveletMatrix[32,daub2],waveletMatrix[32,daub4],waveletMatrix[32,daub8]};
```

2) Carry out a wavelet transform by

```
wSpect=waveletND[data,wMat,3];
```

If we wish to carry out an inverse wavelet transform to the waveletpectrum “wSpect”, then it is carried out by the following steps.

1) Transpose the transform matrices by

```
wMatTrans={Transpose[waveletMatrix[32,daub2]],
        Transpose[waveletMatrix[32,daub4]],Transpose[waveletMatrix[32,daub8]]};
```

2) Carry out an inverse transform by

```
recover=waveletND[data,wMatTrans,3];
```

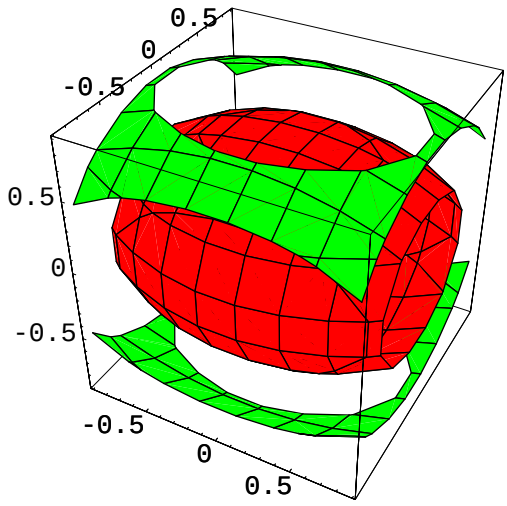
```
waveletND[data_, wMat_, n_] :=
 Block[{s, ww}, s = Join[{n}, Table[i, {i, n - 1}]];
   ww = Transpose[wMat[[1]].data, s];
   Do[ww = Transpose[wMat[[i]].ww, s], {i, 2, n}]; ww];
```

A practical example of the wavelet transform is given bellow. Figure 1 shows an example three-dimensional data "data3D".

```
data3D = Table[x^2+2*y^2+3*z^2,{z,-.875,.875,.25},
    {y,-.875,.875,.25},{x,-.875,.875,.25}];
```

```
ListContourPlot3D[data3D,PlotLabel->"Fig.1. Example of 3D data",
    MeshRange -> {{-.875,.875}, {-.875,.875}, {-.875,.875}},
    Contours -> {1.5, 3.},Lighting -> False, Axes -> True,
    ContourStyle -> {{RGBColor[0,1,0]},{RGBColor[1,0,0]}},
    ImageSize->{200,200}];
```

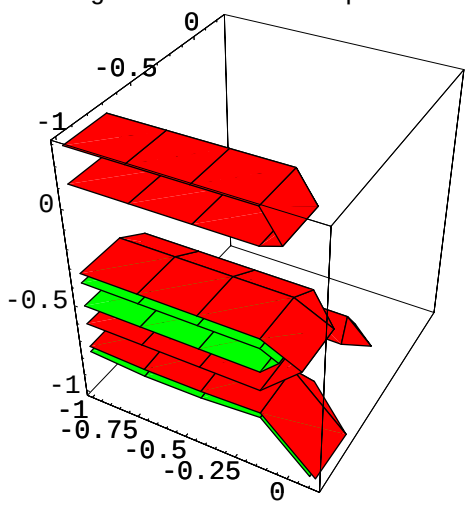
Fig.1. Example of 3D data



Using the Daubechies 2nd, 4th and 6th order base functions, we construct the wavelet transform matrix "wMat", and then we compute a wavelet spectrum "wSpect". Figure 2 shows the wavelet spectrum.

```
base={daub2,daub4,daub6};
wMat=Table[waveletMatrix[8,base[[i]]],{i,3}];
wSpect=waveletND[data3D,wMat,3];
ListContourPlot3D[wSpect,PlotLabel->"Fig.2.3D wavelet spectrum",
    MeshRange -> {{-1,1}, {-1,1}, {-1,1}},
    MeshRange -> {{-1,1}, {-1,1}, {-1,1}},
    Contours -> {1.5, 3.},Lighting -> False, Axes -> True,
    ContourStyle -> {{RGBColor[0,1,0]},{RGBColor[1,0,0]}},
    ImageSize->{200,200}];
```

Fig.2.3D wavelet spectrum



To recover the original data, we transpose the wavelet transform matrices, and then carry out an inverse wavelet transform. In addition, we compute the maximum absolute error between the original "data3D" and recovered

"recover3D" data. Because of the numerical errors, the recovered data "recover3D" are not exactly equivalent to the original ones, but the errors are negligible small values.

Final commands remove the needless memories and check the used memories. Thereby, it is possible to know that the *Mathematica* front-end uses about 1.7-mega bytes memories

```
wMatTrans=Table[Transpose[wMat[[i]]],{i,3}];
recover3D=waveletND[wSpect,wMatTrans,3];
Max[Abs[data3D-recover3D]]
```

2.64464×10^{-11}

```
Remove["data3D","base","wMat","wSpect",
    "wMatTrans","recover3D"];memoryUsed
```

```
1735K Bytes used
```

4.3 Wavelet image compression and recovery

■ 4.3.1 Sample images

In this section, we have to input a sample color image. The 24-bitmap image data file "AF038-128.bmp" is read in a list "sample by the function "rgbBMP". After that we construct its monochrome image data by computing a magnitude distribution of the color image characteristic vectors.

```
sample=rgbBMP["AF038-128.bmp"];
monoSample=vectorMag3D[sample];
```

After computing a resolution of the input image by the *Mathematica* command "Dimensions", we construct its color and monochrome image data. Figure 3 shows the 128 by 128 resolution color and monochrome images.

```
dim=Dimensions[sample];
sampleG=Show[convertRGB[sample],PlotLabel->"Color image",
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity];
monoSampleG=ListDensityPlot[monoSample,PlotRange->All,
    Mesh->False,Frame->False,PlotLabel->"Monochrome image",
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity];
Show[GraphicsArray[{sampleG,monoSampleG}],
    PlotLabel->"Fig.3. Sample images",ImageSize->{300,150}];
```

Fig.3. Sample images

Color image Monochrome image

■ 4.3.2 Principle of wavelet image compression and recovery

In order to show the wavelet image compression and recovery, we apply a two-dimensional wavelet transform to the monochrome image shown on the right in Fig. 3.

At first, we construct the wavelet transform matrices employing the Daubechies 2^{nd} and 4^{th} order base functions. Second, we carry out a wavelet transform. Figure 4 shows the obtained wavelet spectrum. Surprisingly, major parts of this wavelet spectrum are zero, where the zero value is painted in black. Only a small number of elements at the bottom on the left side take the non-zero values. This means that the small number of non-zero values in Fig. 4 could represent the major monochrome image data in a wavelet spectrum domain.

```
base={daub2,daub4};
wMat=Table[waveletMatrix[dim[[2]],base[[i]]],{i,2}];
spect=waveletND[monoSample,wMat,2];
```

```
ListDensityPlot[spect,PlotRange->All,
    Mesh->False,Frame->False,AspectRatio->dim[[2]]/dim[[3]],
    PlotLabel->"Fig.4. Wavelet spectram"];
```

Fig.4. Wavelet spectram

According to the result shown in Fig. 4, we take only 25 percent wavelet spectrum into account. Namely, the monochrome image data is compressed into the 25 percent data. Apply an inverse wavelet transform to this 25 percent spectrum recovers the monochrome image approximately. Figure 5 shows a recovered image. It is revealed that the approximately recovered image in Fig. 5 is a poor image. This is because the original 128 by 128 resolution image has been reduced into a 32 by 32 resolution image.

To evaluate a correlation coefficient between the original and recovered image data, we construct an approximate 128 by 128 resolution wavelet spectrum. The following steps carry this out.

```
partSpect=TakeMatrix[spect,{1,1},{dim[[2]]/4,dim[[3]]/4}];
wMatPrimeTrans=Table[Transpose[waveletMatrix[dim[[2]]/4,
    base[[i]]]],{i,2}];
recoverPrime=waveletND[partSpect,wMatPrimeTrans,2];
```

```
ListDensityPlot[recoverPrime,PlotRange->All,
    Mesh->False,Frame->False,AspectRatio->dim[[2]]/dim[[3]],
    PlotLabel->"Fig.5.Compressed image",ImageSize->{200,200}];
```

Fig.5.Compressed image

At first, we construct a 128 by 128 zero-matrix. Second, the 25 percent wavelet spectrum is embedded into this zero-matrix. Thus, we have the approximate 128 by 128 wavelet spectrum. Apply an inverse wavelet transform recovers a 128 by 128 resolution image data. Figure 6 shows an approximately recovered 128 by 128 resolution image.

```
spectComp=ZeroMatrix[dim[[2]]];
Do[spectComp[[i,j]]=partSpect[[i,j]],{i,dim[[2]]/4},{j,dim[[3]]/4}];
wMatTrans=Table[Transpose[wMat[[i]]],{i,2}];
recover=waveletND[spectComp,wMatTrans,2];

ListDensityPlot[recover,PlotRange->All,
    Mesh->False,Frame->False,AspectRatio->dim[[2]]/dim[[3]],
    PlotLabel->"Fig.6. Recovered image",ImageSize->{200,200}];
```

Fig.6. Recovered image

We compute a correlation coefficient between the image data in Fig. 6 and on the right in Fig. 3. Surprisingly, the correlation coefficient is about 0.97.

Thus, from a *Mathematica*l viewpoint, the wavelet transform is capable of compressing the image data.

```
corRelation[Flatten[recover],Flatten[monoSample]]
```

```
0.968937
```

At the finishing of this section, we remove the used variables and check the used memories.

```
Remove["base","wMat","spect","partSpect","wMatPrimeTrans",
    "recover"];memoryUsed
```

```
2574K Bytes used
```

■ 4.3.3 RGB color image compression and recoverability

The purpose of this section is to examine a nature of wavelet image compression as well as recoverability. The nature of wavelet transform may be classified into two major categories. One is depended on a selection of the base functions, and the other is the order of the selected base function. In this section, we employ the Coifman's 6^{th}, 12^{th}, 18^{th}, 24^{th} and 30^{th} order base functions. A target image to be compressed is the color sample shown on the left in Fig. 3.

In order to extract the 25 percent major wavelet spectrum, we construct a filter matrix by the following codes.

```
filter=ZeroMatrix[dim[[2]]];
Do[filter[[i,j]]=1.,{i,dim[[2]]/4},{j,dim[[3]]}];
```

The color image data are rearranged to a one-dimensional form for a convenience of a correlation coefficient computation. After selecting the base functions, we compute the correlation coefficients and approximately recovered color image data by the following *Mathematica* codes.

```
sampleFl=Flatten[sample];
base={coif6,coif12,coif18,coif24,coif30};

rgbR=Table[ZeroMatrix[dim[[2]],dim[[3]]],{Length[base]},
    {dim[[1]]}];
corC=Table[
    wMat=waveletMatrix[dim[[2]],base[[i]]];
    wMatTrans=Transpose[wMat];
    pSpect=Table[filter*waveletND[sample[[j]],{wMat,wMat},2],
        {j,dim[[1]]}];
    rgbR[[i]]=Table[waveletND[pSpect[[j]],{wMatTrans,wMatTrans},2],
        {j,dim[[1]]}];
    corRelation[Flatten[rgbR[[i]]],sampleFl],
    {i,Length[base]}];
```

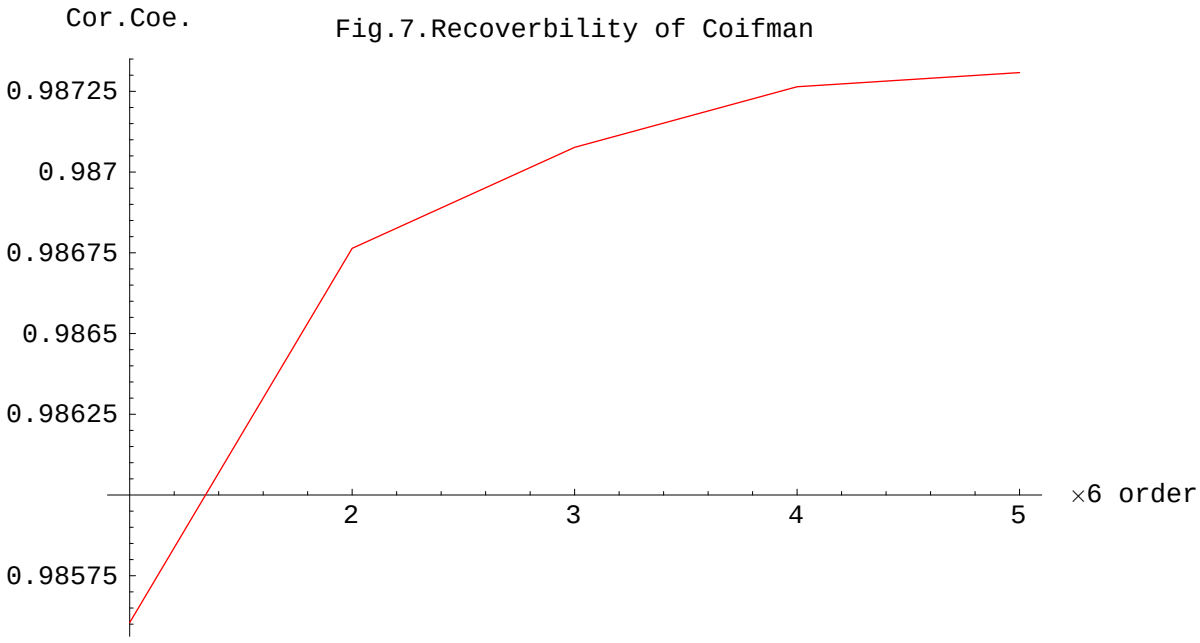
The approximately recovered color image data are normalized to the values between 0 and 1, because the wavelet compressed color image data do not always take the values between the 0 and 1.

```
dummy=rgbR;
rgbR=Table[imageNormalize[dummy[[i,j]]],{i,Length[base]},{j,dim[[1]]}];
```

Figure 7 shows a relationship between the correlation coefficients and the order of base function. Obviously, employment of the higher order base functions yields the higher correlation coefficients.

Before to finishing this section, we remove the needless variables and check the used memories. As a result, it is found that a relatively large amount of memories are required to keep the approximately recovered color images.

```
rgbG=ListPlot[corC,PlotStyle->RGBColor[1,0,0],PlotRange->All,
    PlotJoined->True,PlotLabel->"Fig.7.Recoverbility of Coifman",
    AxesLabel->{"×6 order","Cor.Coe."},ImageSize->{450,300}];
Remove["corC","wMat","wMatTrans","pSpect","dummy"];
memoryUsed
```




```
7589K Bytes used
```

■ 4.3.4 Orthogonal image data decomposing and composing

This section proposes a methodology, which converts a color image data into the spherical coordinate components.

Mathematica function colorDecode

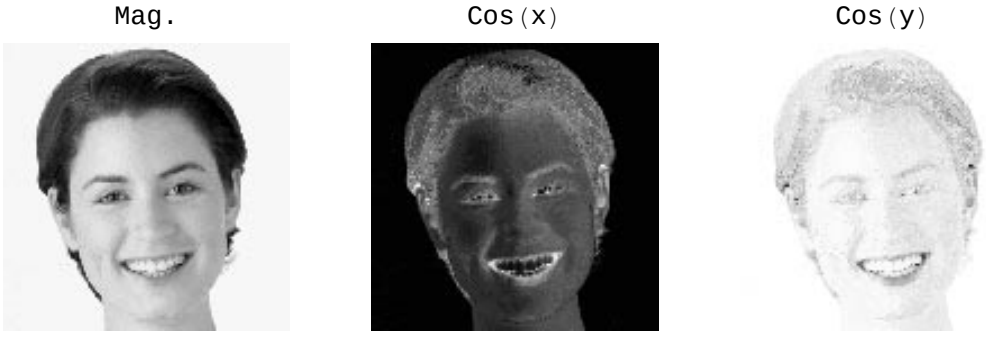
The magnitude of color image characteristic vector corresponds to a radius. The attitude and longitude are represented in terms of their directional co-sinusoidal components. A following function "colorDecomp" decomposes a color image data "vector3D" into the spherical coordinate components.

```
colorDecomp=Compile[{{vector3D,_Real,3}},
    Module[{vec={0.},out={{{0.}}},dim={0},
        vn=0.,i=0,j=0,k=0},
        dim=Dimensions[vector3D];
        out=Table[
            vec=Table[vector3D[[k,i,j]],{k,dim[[1]]}];
            vn=Sqrt[vec.vec];
            If[vn=!=0.,vec={vn,vec[[1]]/vn,vec[[2]]/vn},
                vec={0.,0.,0.}];vec,
        {i,dim[[2]]},{j,dim[[3]]}];
        Table[out[[i,j,k]],{k,dim[[1]]},{i,dim[[2]]},{j,dim[[3]]}]
    ]];
```

Apply this function to the sample color image data yields a set of decomposed results as shown in Fig. 8. It is revealed that the color sample image data can be decomposed into a set of quite different quantities instead of the red, green and blue components. One of the merits of this spherical coordinate representation is that the radius component is corresponding to the magnitude distribution of color image characteristic vectors, and is a monochrome image.

```
colorD=colorDecomp[sample];

label={"Mag.","Cos(x)","Cos(y)"};
colorDG=Table[ListDensityPlot[colorD[[i]],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["``",label[[i]]],
    DisplayFunction->Identity],{i,3}];
Show[GraphicsArray[colorDG],ImageSize->{450,150},
    PlotLabel->"Fig.8.Decomposed images"];
```

Fig.8.Decomposed images



Mathematica function colorDecode

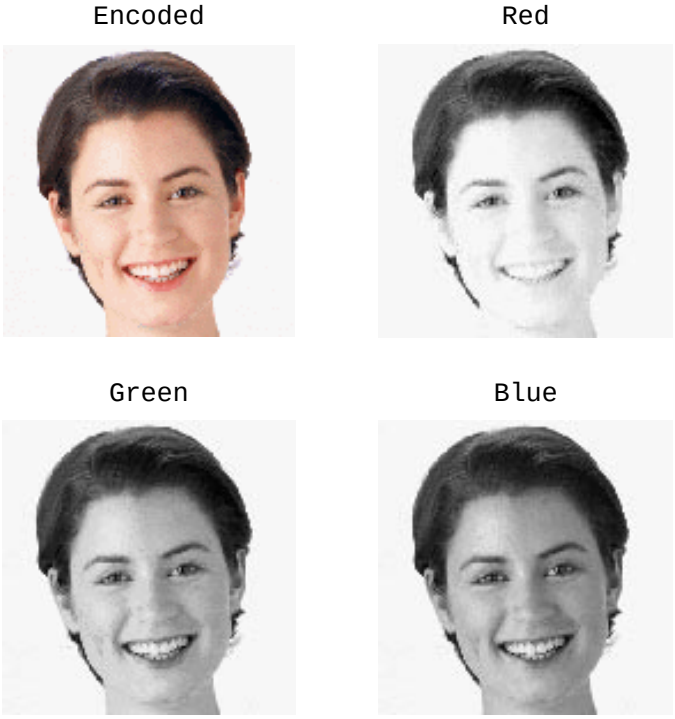
To draw a color image, we have to convert the decomposed components in Fig. 8 into the original RGB components. This is carried out by a following *Mathematica* function "colorComp". A parameter "colorD" of this function are a three dimensional array housing the color image components in term of the spherical coordinate system.

```
colorComp=Compile[{{colorD,_Real,3}},
    Module[{r={{0.}},g={{0.}},p={{0.}},out={{{0.}}},
        i=0,min=0.,max=0.},
        r=colorD[[1]] colorD[[2]];
        g=colorD[[1]] colorD[[3]];
        p=r^2+g^2;
        out={r,g,Sqrt[Abs[colorD[[1]]^2-p]]};
        Table[min=Min[out[[i]]];
            (out[[i]]-min)/Max[out[[i]]-min],{i,3}]
        ]];
```

Using this function, we compose the color image red, green and blue components. Figure 9 shows the recovered color, red, green and blue components images. Because of the orthogonal decomposition and composition, the original color image as well as its components is exactly recovered in Fig. 9.

```
colorC=colorComp[colorD];

label={"Red","Green","Blue"};
colorCG=Table[ListDensityPlot[colorC[[i]],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["``",label[[i]]],
    DisplayFunction->Identity],{i,3}];
composedG=Show[convertRGB[colorC],PlotLabel->"Encoded",
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity];
Show[GraphicsArray[{{composedG,colorCG[[1]]},
    {colorCG[[2]],colorCG[[3]]}}],ImageSize->{300,300},
    PlotLabel->"Fig.9. Composed and componets images"];
```

Fig.9. Composed and componets images



Before to move on a next section, we remove the needless variables, and check the used memories. In addition to the memories required to store the approximately recovered color image data, the decomposed color components in the spherical coordinate require about 1.1 mega bytes memories.

```
Remove["colorDG","colorC","label","colorCG","composedG"];
memoryUsed
```

```
8617K Bytes used
```

■ 4.3.5 Decomposed Image compression

This section examines a nature of wavelet image compression as well as recoverability to the decomposed color image components in the spherical coordinate. Similar to that of the red, green and blue components image data compression, we employ the Coifman's 6^{th}, 12^{th}, 18^{th}, 24^{th} and 30^{th} order base functions for the wavelet transform. A target image to be compressed is the image data shown in Fig. 8.

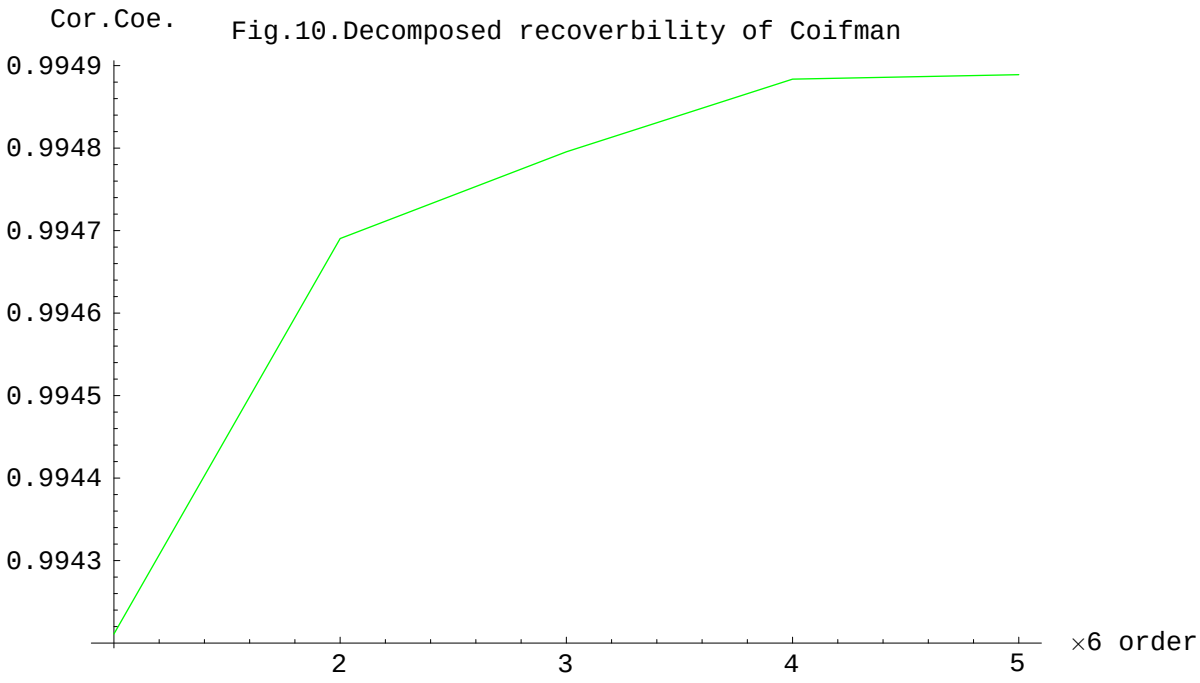
The decomposed color image data are rearranged to a one-dimensional form for a convenience of a correlation coefficient computation. After that, we compute the correlation coefficients and approximately recovered color image data by the following *Mathematica* codes.

```
dSampleFl=Flatten[colorD];
deCompR=Table[ZeroMatrix[dim[[2]],dim[[3]]],
    {Length[base]},{dim[[1]]}];
corC=Table[
    wMat=waveletMatrix[dim[[2]],base[[i]]];
    wMatTrans=Transpose[wMat];
    pSpect=Table[filter*waveletND[colorD[[j]],{wMat,wMat},2],
        {j,dim[[1]]}];
    deCompR[[i]]=Table[waveletND[pSpect[[j]],
        {wMatTrans,wMatTrans},2],{j,dim[[1]]}];
    corRelation[Flatten[deCompR[[i]]],dSampleFl],
    {i,Length[base]}];
```

Figure 10 shows a relationship between the correlation coefficients and the order of base function. Similar to those of the red, green and blue components image data compression, employment of the higher order base functions yields the higher correlation coefficients.

Finally, we remove the needless variables and check the used memories.

```
deCompG=ListPlot[corC,PlotStyle->RGBColor[0,1,0],
    PlotRange->All,PlotJoined->True,ImageSize->{450,300},
    PlotLabel->"Fig.10.Decomposed recoverbility of Coifman",
    AxesLabel->{"×6 order","Cor.Coe."}];
Remove["corC","wMat","wMatTrans","pSpect"];memoryUsed
```



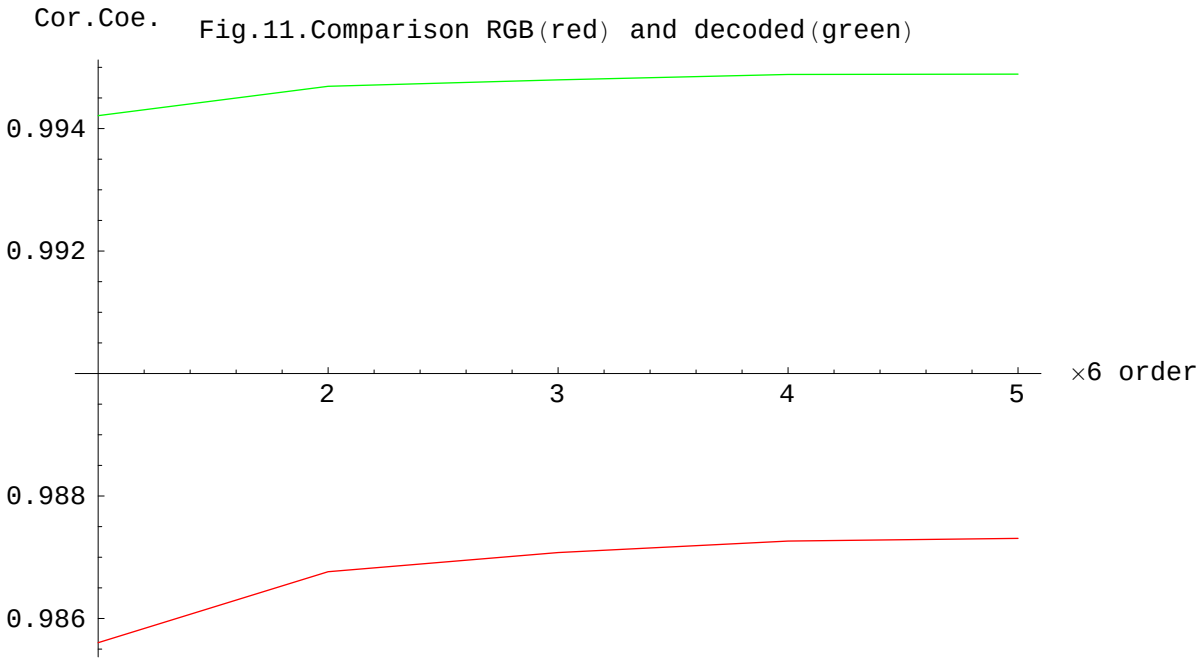

```
12981K Bytes used
```

Figure 11 shows a comparison between the RGB image and the decomposed image data. The red and green lines are corresponding to the RGB and decomposed image data, respectively. From a result shown in Fig. 11, it is clarified that the color image data compression by the wavelet transform greatly depends on the data arrangement.

Thus, it is preferable to use the decomposed image data in the spherical coordinate in order to attain a high compression rate by the discrete orthogonal wavelet transform.

Before to move on the next section, we remove the variables used for the correlation coefficients, and check the used memories.

```
Show[rgbG,deCompG,
    PlotLabel->"Fig.11.Comparison RGB(red) and decoded(green)",
    AxesLabel->{"×6 order","Cor.Coe."},ImageSize->{450,300}];
    Remove["rgbG","deCompG"];memoryUsed
```




```
12982K Bytes used
```

■ 4.3.6 Recovered image comparison

This section shows the approximately recovered color images. We have two types of the recovered images. One is recovered from the conventional red, green and blue compressed components. The other is recovered from the compressed radius, attitude and longitude components.

Figure 12 shows the recovered color images from the 25 percent compressed image data. *Mathematica*lly, the right side images have higher recoverability than those of right side ones, but our human eyes could not find out a big difference between them. Hence, a small different recoverability between them is no meaning to us, but sometimes, it becomes a significant difference for the image identification and visualizations.

Because of a great memory requirement of a next section, we remove the used variables and check up the used memories.

```
compR=Table[colorComp[deCompR[[i]]],{i,Length[base]}];
```

```
compeG=Table[{
    Show[convertRGB[rgbR[[i]]],
        PlotLabel->StringForm["RGB ``order",6*i],
        AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity],
    Show[convertRGB[compR[[i]]],
        PlotLabel->StringForm["Decomposed ``oder",6*i],
        AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity]},
{i,1,Length[base],2}];

Show[GraphicsArray[compeG],ImageSize->{300,400},
    PlotLabel->"Fig.12. Compressed image by Coifman"];
Remove["compR","deCompR","compeG","rgbR","base"];
memoryUsed
```

Fig.12. Compressed image by Coifman

RGB 6order

Decomposed 6oder

RGB 18order

Decomposed 18oder

RGB 30order

Decomposed 30oder

3097K Bytes used

4.4 Dynamic image processing

■ 4.4.1 Three-dimensional image sample

Previous sections have carried out the two-dimensional wavelet image compressions. In this section, we carry out the three-dimensional wavelet image expansions.

At the beginning, we have to set up the sample images. A 64 by 64 resolution color image is employed as a sample image. This sample image data is read in the *Mathematica* front-end by a following code. After reading in the sample image data, we compute its resolution.

```
sample=rgbBMP["AF038-64.bmp"];
dim=Dimensions[sample];
```

By means of the inner product in vector fields, we wok out the 8 shadowed lighting images. Figure 13 shows the shadowed lighting images. After removing the graphics image data, it is revealed that about 3.4 mega bytes are required to store the 8 shadowed lighting images.

```
rotation=8;
color3D=colorImage3D[sample,rotation];

color3DG=Table[Show[convertRGB[color3D[[i]]],
    PlotLabel->StringForm["``deg.",360*(i-1)/rotation],
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity],
    {i,rotation}];

Show[GraphicsArray[
    Table[Table[color3DG[[i+j]],{j,0,3}],{i,1,rotation,4}]],
    ImageSize->{400,200},PlotLabel->"Fig.13. 3D color images"];
Remove["color3DG"];memoryUsed
```

Fig.13. 3D color images


```
6432K Bytes used
```

■ 4.4.2 Three-dimensional wavelet transform

The sample image shown in Fig. 13 consists of the shadowed images having 8 different lighting directions. Each of the samples can be represented by the three two-dimensional image data, so that each of the red, green and blue color component data is represented in a three-dimensional image data. Thereby, we apply a three-dimensional wavelet transform to each of the color component data. The Daubechies 4^{th}, 16^{th} order base functions are employed.

```
zcolor3D=Table[color3D[[j,i,k,l]],
    {i,dim[[1]]},{j,rotation},{k,dim[[2]]},{l,dim[[3]]}];
dimZ=Dimensions[zcolor3D];
base={daub4,daub16,daub16};
wMat=Table[waveletMatrix[dimZ[[i+1]],base[[i]]],{i,Length[base]}];
zSpect=Table[waveletND[zcolor3D[[i]],wMat,3],{i,dim[[1]]}];
```

After computing the three-dimensional wavelet spectrum, we work out an approximate wavelet spectrum of the 16 shadowed lighting image data. An inverse wavelet transform generates the expanded 16 shadowed lighting image data from the 8 ones. These processes are applied to each of the red, green and blue components.

```
angle=16;
aSpect=Table[0.,{dim[[1]]},{angle},{dim[[2]]},{dim[[3]]}];
Do[aSpect[[i,j]]=zSpect[[i,j]],{i,dim[[1]]},{j,rotation}];
wMat[[1]]=waveletMatrix[angle,base[[1]]];
wMatTrans=Table[Transpose[wMat[[i]]],{i,Length[base]}];
angleInc=Table[waveletND[aSpect[[i]],wMatTrans,3],{i,dim[[1]]}];
incColor3D=Table[imageNormalize[angleInc[[i,j]]],
    {j,angle},{i,dim[[1]]}];
```

The expanded color image data are converted into the color graphics image ones. Figure 14 shows the expanded images. Depending on the lighting angles, the 8 shadowed lighting color images in Fig. 13 are successfully expanded to the 16 ones by the three-dimensional wavelet transform.

```
incColor3DG=Table[Show[convertRGB[incColor3D[[i]]],
    PlotLabel->StringForm["``deg.",Round[360*(i-1.)/angle]],
    AspectRatio->dim[[2]]/dim[[3]],
    DisplayFunction->Identity],{i,angle}];
```

```
Show[GraphicsArray[
    Table[Table[incColor3DG[[i+j]],{j,0,3}],{i,1,angle,4}]],
    ImageSize->{400,400},PlotLabel->"Fig.14. Increased 3D color images"];
```

Fig.14. Increased 3D color images

After setting a mouse cursor on a following image, twice clicking the left side button of mouse animates the facial changes depending on the lighting angles. This is only effective on the *Mathematica* notebook.

```
Do[Show[GraphicsArray[{incColor3DG[[i]]}]],{i,angle}];
```

0deg.

After removing the needless variables, checking the used memories reveals that a relatively large amount of memories are used.

```
Remove["color3D","zcolor3D","zSpect","incColor3D","incColor3DG"];
memoryUsed
```

```
9975K Bytes used
```

■ 4.4.3 Illusive image

Similar to those of the previous section, we work out the 8 illusive color images by means of the function "illusion3D". Figure 15 shows the 8 illusive color images.

```
illusion=illusion3D[sample,rotation];
```

```
illusionG=Table[Show[convertRGB[illusion[[i]]],
    PlotLabel->StringForm["``deg.",360*(i-1)/rotation],
    AspectRatio->dim[[2]]/dim[[3]],DisplayFunction->Identity],
    {i,rotation}];
Show[GraphicsArray[
    Table[Table[illusionG[[i+j]],{j,0,3}],{i,1,rotation,4}]],
    ImageSize->{400,200},PlotLabel->"Fig.15. Original illusive images"];
```

Fig.15. Original illusive images

Similar to that of the shadowed color images, we generate the 16 illusive color image data by the three-dimensional wavelet transform.

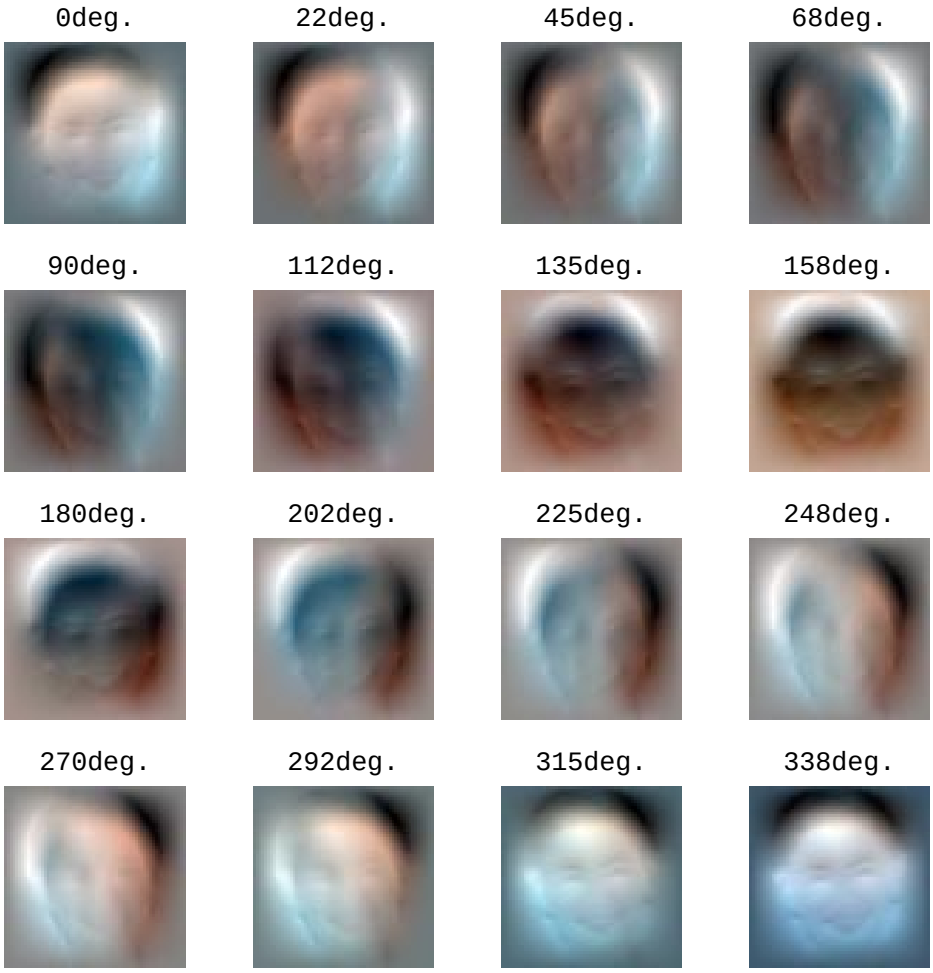
```
zillusion=Table[illusion[[j,i,k,l]],
    {i,dim[[1]]},{j,rotation},{k,dim[[2]]},{l,dim[[3]]}];
wMat[[1]]=waveletMatrix[rotation,base[[1]]];
zSpect=Table[waveletND[zillusion[[i]],wMat,3],{i,dim[[1]]}];
aSpect=Table[0.,{dim[[1]]},{angle},{dim[[2]]},{dim[[3]]}];
Do[aSpect[[i,j]]=zSpect[[i,j]],{i,dim[[1]]},{j,rotation}];
wMat[[1]]=waveletMatrix[angle,base[[1]]];
wMatTrans=Table[Transpose[wMat[[i]]],{i,Length[base]}];
angleInc=Table[waveletND[aSpect[[i]],wMatTrans,3],{i,dim[[1]]}];
incIllusion=Table[imageNormalize[angleInc[[i,j]]],
    {j,angle},{i,dim[[1]]}];
```

After converting the image data into color graphics one, we can get the 16 illusive color images as shown in Fig. 16.

```
incIllusionG=Table[Show[convertRGB[incIllusion[[i]]],
    PlotLabel->StringForm["``deg.",Round[360*(i-1.)/angle]],
    AspectRatio->dim[[2]]/dim[[3]],
    DisplayFunction->Identity],{i,angle}];
```

```
Show[GraphicsArray[
    Table[Table[incIllusionG[[i+j]],{j,0,3}],{i,1,angle,4}]],
    ImageSize->{400,400},PlotLabel->"Fig.16. Increased illusive images"];
```

Fig.16. Increased illusive images



Thus, we have succeeded in expanding the three-dimensional color image by means of the wavelet transform.

After setting a mouse cursor on a following image, twice clicking the left side button of mouse animates the facial changes depending on the lighting angles. This is only effective on the *Mathematica* notebook.

```
Do[Show[GraphicsArray[{incIllusionG[[i]]}]],{i,angle}];
```

0deg.

4.5 Summary

As shown above, this chapter have clarified that the discrete orthogonal wavelet transform is a quite useful tool for image data compression as well as expansion.

This chapter has introduced the applications of the wavelet transform to the image data. The first section of this chapter has described about the discrete orthogonal wavelet transform, which employ the Daubechies, Coifman and Baylkin's base functions. The second section has described the monochrome image data compression and expansion by the wavelet transform. This section has been an introduction of the wavelet image processing. Also, this section has demonstrated that *Mathematica*lly remarkable image compression rate is possible by the wavelet transform. The third section has concerned with the color image compression and recovery. It has been revealed that the compression rate and recoverability depend on the order of base function. The fourth section has proposed one of the orthogonal color image decomposition. In this section, the color image data have been represented in terms of the spherical coordinate quantities. The magnitude of color image characteristic vector has corresponded to a radius. The attitude and longitude have been represented in terms of their directional co-sinusoidal components. In continuation to this section, the fifth section has described about the wavelet compression and recovery to the color image data represented in terms of the spherical coordinate quantities. The sixth section has revealed that the color image data compression rate by the wavelet transform depends on the way of image data representations. Namely, higher recoverability could be achieved by the spherical coordinate representation. The seventh and final sections have concerned with the expansion of a small number of color animation data to a large number of ones by the wavelet transform. This has demonstrated that the three-dimensional wavelet transform makes it possible to increase the number of animation data.

Thus, we have confirmed that the wavelet transform along with the vector fields provides not only the simple image data compression and expansion tool, but also suggests a higher compression rate possibility, i.e., the image data represented in terms of the spherical coordinate components could be compressed with higher recoverability by the wavelet transform.

■ REFERENCES

[1] Stephen Wolfram, *The Mathematica Book*, 3rd ed. (Wolfram Media/Cambridge University Press, 1996).

[2] J.D.Jackson, "Classical Electrodynamics 3rd Edition," John Wiley & Sons, New York (1998).

Chapter 5. Eigen Pattern Image Processing

5.1 Introduction

In any physical vector fields, we can find the eigen value and vectors. The eigen value represents the distinct physical system parameter. For example, the time constants of the electrical resistance, inductance and capacitance circuits are the inverse of eigen value. In the mechanical mass and spring systems, we have the eigen value, which corresponds to a natural resonant frequency of the system.

The target of this chapter is to find the parameter representing the distinct image. Since we have regarded the image data as the scalar or one component of the vector potentials, then we have found and established the new methodologies for the image processing. In physical system, we have found the eigen value and vectors representing the intrinsic characteristics of the system. This means that the eigen value or vectors should be found in the computer graphics.

In classical vector fields, the eigen value or its equivalent has been deduced as a result of the continuous field governing equations. Fundamental difference between the classical field theory and computer graphics is that the physical field theory has been established in a continuous space but the computer graphics can be established only in a discretized space. In the other words, the computer graphics has been established in an artificial space. Thereby, it is difficult to find and define the eigen value of a digital image, exactly. Modern discrete mathematics has revealed that the approximate eigen values could be computed from the discretized system of equations. In such meaning, it is possible to find the image eigen values from an image governing system of equations. Discretization of the Poisson type partial differential equations can derive the image governing system of equations. When we evaluate the eigen values from such an image governing system of equations, the smaller and larger eigen values may provide the eigen vectors representing the smoother and spikier lines as well as surfaces, respectively. The investigations about this are significant, but we do not discuss about these eigen values.

Principal purpose of this chapter is to derive an eigen pattern not the eigen value. In chapter 4, we have derived one of the eigen patterns by the discrete orthogonal wavelet transform. A wavelet spectrum is one of the eigen patterns derived by a simple linear transform using a square wavelet transform matrix. In the present chapter, we generalize this linear transform to a nonlinear transform using a rectangular transform matrix. A meaning of the term “nonlinear” is that an inverse transform using the rectangular transform matrix never recover an exact original data but gives the best approximate data.

In the first section of this chapter, we describe a key idea deriving an eigen pattern. In the second section, we describe to the practical *Mathematica* codes deriving the eigen pattern. Also, we examine the nature of eigen pattern employing several one-dimensional image examples, which are the time domain sinusoidal waves. The third section derives an eigen pattern of the monochrome images, where we are demonstrated the angle and resolution independencies of the image eigen pattern. In addition to this section, the fourth section derives the eigen pattern of color image.

As a result, it is revealed that the image eigen pattern makes it possible to generate any angled and resolution images.

5.2 Preparation of *Mathematica*

■ 5.2.1 *Mathematica* utilities and packages

Before to move on the practical image processing, we have to install the memory conserve utilities and the warning messages suppressing. In addition *Mathematica* utilities, the " LinearAlgebra ' MatrixManipulation" package has to be installed..

```
<<Utilities`MemoryConserve`
$MemoryIncrement=100000;
Off[General::spell1,MemoryConserve::start,MemoryConserve::end];
<< LinearAlgebra`MatrixManipulation`;
```

■ 5.2.2 *Mathematica* functions

Here, we define several functions that are described and used in the previous chapters. The functions "rgbBMP", "convertRGB", "vectorMag3D", "corRelation" and "memoryUsed" have been defined in the previous chapters, so that the comments of such functions are not described.

Function rgbBMP

Function convertRGB

Function vectorMag3D

Function corRelation

Function memoryUsed

Function window

Function convert2D

■ 5.2.3 *Mathematica* functions for eigen pattern generation

Principle

Let us consider a vector given by

```
x={0,1,2,3,3,3,2,1,0};
```

This vector "x" is composed of the two 0, two 1, two 2 and three 3 elements. Our purpose is to extract the number of the elements taking the same absolute value not taking into account the zero elements. In order to achieve this, we set a resolution to m=3 because we have to count the three kinds of numerical values. After that, we normalize the vector "x" by

```
m = 3;
xn = m * x / Max[Abs[x]];
```

In order to extract the three kinds of numerical values from the vector 9^{th} order vector "x", we construct a 3 by 9 rectangular transform matrix "c" by

```
n=9;
c=Table[
        If[Sign[xn[[j]]] xn[[j]]==i,p=1./x[[j]],p=0];p,
        {i,m},{j,n}];
```

This matrix "c" can be rewritten in matrix form by

```
c//MatrixForm
```

$$\begin{pmatrix} 0 & 1. & 0 & 0 & 0 & 0 & 0 & 1. & 0 \\ 0 & 0 & 0.5 & 0 & 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.333333 & 0.333333 & 0.333333 & 0 & 0 & 0 \end{pmatrix}$$

Using this matrix transform "c", we can extract the number of non-zero elements from the vector "x" by

```
b=c.x
```

```
{2., 2., 3.}
```

The first, second and third elements in a vector "b" are corresponding to the number of the elements taking the numeric values 1, 2 and 3, respectively.

Thus, we have succeeded in extracting the number of non-zero elements from the vector "x". Here, we define the vector "b" as an **eigen pattern** vector. In order to recover the original vector "x" from the eigen pattern vector "b", we construct an inverse matrix by means of a least norm solution sense for an ill posed linear system.

```
c.Transpose[c]//MatrixForm
```

$$\begin{pmatrix} 2. & 0. & 0. \\ 0. & 0.5 & 0. \\ 0. & 0. & 0.333333 \end{pmatrix}$$

Since all of the elements excepting the diagonal elements are zero, then the row vectors of the transform matrix "c" are the independent orthogonal vectors. Thereby, similar to the least norm method, an inverse transform matrix "d" is formally obtained by

```
d=Transpose[c].Inverse[c.Transpose[c]];
%//MatrixForm
```

$$\begin{pmatrix} 0. & 0. & 0. \\ 0.5 & 0. & 0. \\ 0. & 1. & 0. \\ 0. & 0. & 1. \\ 0. & 0. & 1. \\ 0. & 0. & 1. \\ 0. & 1. & 0. \\ 0.5 & 0. & 0. \\ 0. & 0. & 0. \end{pmatrix}$$

Using this inverse transform matrix "d", we can recover the original vector "x" by

```
d . b
```

```
{0., 1., 2., 3., 3., 3., 2., 1., 0.}
```

Thus, the original vector “x” is successfully obtained by means of the inverse transform matrix “d”. In this case, the target vector “x” has a 3 level resolution and we have set the 3 level resolution. This leads to a successful recovery result. However, it is difficult to set up the resolution in accordance with exact one. In such case, we confront to the zero row vector of the transform matrix, even though these vectors are the independent orthogonal ones. In such a case, it is difficult to derive an exact inverse transform matrix “d”.

Function eigenMatrix

This function “eigenMatrix” is derived from an input vector “base”. The resulting transform matrix becomes a rectangular matrix with the “resolution”th rows and length of the vector “base”th columns.

```
eigenMatrix = Compile[{{base, _Real, 1}, {resolution, _Integer}},
 Module[{p = 0., dataPrime = {0},
    n = Length[base], i = 0, j = 0},
    dataPrime = Round[resolution * base / Max[Abs[base]]];
     Table[
       If[Sign[base[[j]]] * dataPrime[[j]] == i,
        p = 1. / base[[j]], p = 0.]; p,
     {i, resolution}, {j, n}]
  ]];
```

Function eigenPattern

This function “eigenPattern” derives an eigen pattern directly from an input vector “base”. This functional type routine looses a structured process for the eigen pattern extraction, but it is extremely useful routine to handle a large size base function.

```
eigenPattern =
 Compile[{{base, _Real, 1}, {resolution, _Integer}},
  Module[{dummy = Abs[base], out = {0}, i = 0},
   out = Round[resolution * dummy / Max[dummy]];
   Table[Count[out, i], {i, resolution}]
  ]];
```

Function inverseEigenMatrix

The inverse transform matrix “inverseEigenMatrix” is based on the fact that each of the row vectors in the transform matrix “eigenMat” is an orthogonal vector. The output matrix of the *Mathematica* function “inverseEigenMatrix” has a transposed form of the transform matrix “eigenMat”.

```
inverseEigenMatrix = Compile[{{eigenMat, _Real, 2}},
  Module[{v = {0.}, d = {{0.}}, dim = {0}},
   dim = Dimensions[eigenMat];
   v = Table[eigenMat[[i]] . eigenMat[[i]], {i, dim[[1]]}];
   d = ZeroMatrix[dim[[1]]];
   Do[If[v[[i]] =!= 0., d[[i, i]] = 1. / v[[i]]], {i, dim[[1]]}];
   Transpose[eigenMat] . d
 ]];
```

Function inverseEigenPattern

This “inverseEigenPattern” routine recovers an original image directly from an eigen pattern along with a base function “base”. Because of the different numerical processes, this function gives a recovered vector with a small difference compared with those of the function “inverse EigenMatrix”.

```
inverseEigenPattern =
 Compile[{{eigen, _Real, 1}, {base, _Real, 1}, {resolution, _Integer}},
  Module[{out = {{0.}}, d = {0.}},
   d = 1. * Round[resolution * base] / resolution;
   out = Transpose[Table[d, {resolution}]];
   out . eigen / Apply[Plus, eigen]
 ]];
```

5.3 The nature of eigen patterns

■ 5.3.1 One-dimensional image

Sample images

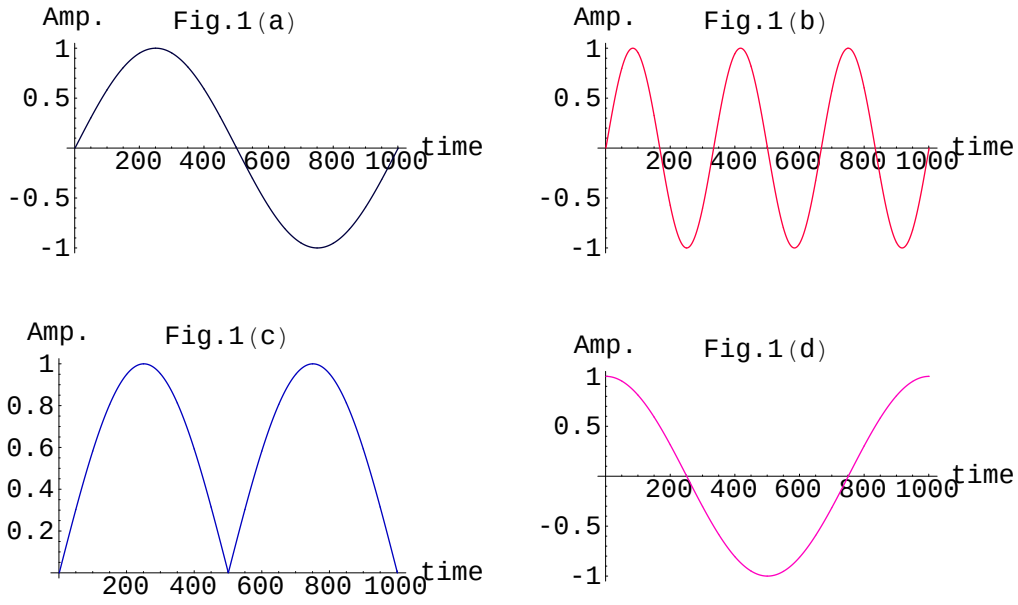
Let us consider the four one-dimensional images in time domain having the different frequencies, time phase and waveforms.

```
resolution=100;
sample1D={Table[Sin[2Pi i/resolution],{i,0,resolution,0.1}],
    Table[Sin[6Pi i/resolution],{i,0,resolution,0.1}],
    Table[Abs[Sin[2Pi i/resolution]],{i,0,resolution,0.1}],
    Table[Cos[2Pi i/resolution],{i,0,resolution,0.1}]};
```

Figure 1 shows the one-dimensional sample images.

```
label={"Fig.1(a)","Fig.1(b)","Fig.1(c)","Fig.1(d)"};
sample1DG=Table[ListPlot[sample1D[[i+j]],
    PlotRange->All,PlotJoined->True,AxesLabel->{"time","Amp."},
    PlotStyle->RGBColor[i,0,j/4],
    PlotLabel->StringForm["`1`",label[[i+j]]],
    DisplayFunction->Identity],{j,1,4,2},{i,0,1}];
```

```
Show[GraphicsArray[sample1DG],ImageSize->{400,250}];
```



Eigen patterns

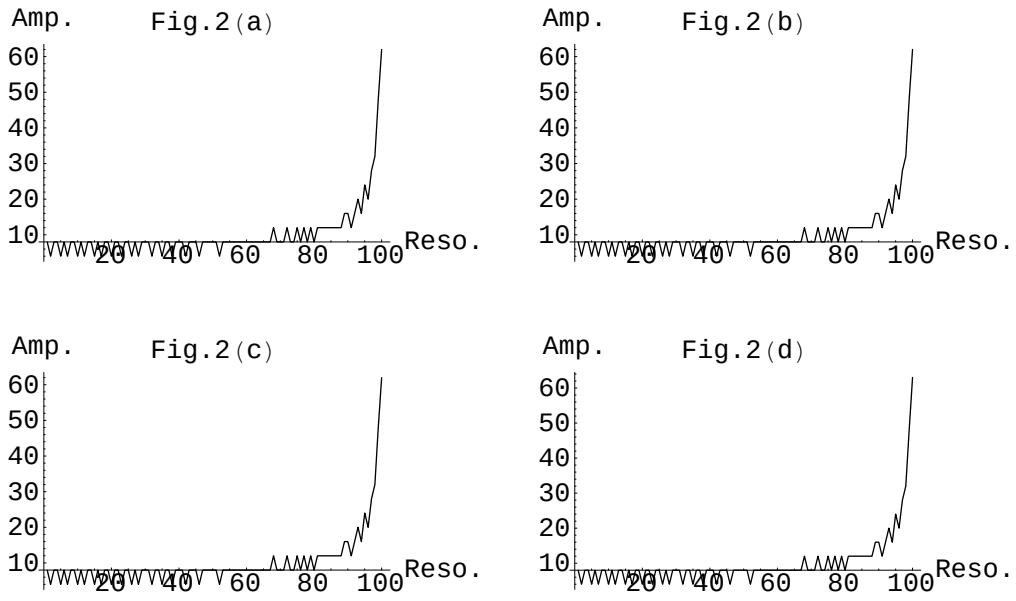
Let us compute the eigen pattern transform matrices by

```
eigenMat=Table[eigenMatrix[sample1D[[i]],resolution],{i,4}];
```

then we evaluate the eigen patterns, which are shown in Fig.2.

```
eigenPatternM=Table[eigenMat[[i]].sample1D[[i]],{i,4}];
```

```
label={"Fig.2(a)","Fig.2(b)","Fig.2(c)","Fig.2(d)"};
eigenM1DG=Table[ListPlot[eigenPatternM[[i+j]],
    PlotRange->All,PlotJoined->True,
    AxesLabel->{"Reso.","Amp."},
    PlotLabel->StringForm["`1`",label[[i+j]]],
    DisplayFunction->Identity],{j,1,4,2},{i,0,1}];
Show[GraphicsArray[eigenM1DG],ImageSize->{400,250}];
```



Comparison the original one-dimensional images in Fig.1 and their eigen patterns reveals that all of the eigen patterns are the same even though their frequencies, time phase and waveforms are different. To check this, we compute a correlation coefficient between them.

```
Table[
    corRelation[eigenPatternM[[1]],eigenPatternM[[i]]],
{i,2,4,1}]
```

```
{1., 1., 0.999956}
```

Thus, our method has extracted the unique eigen pattern that has a common nature of the sinusoidal waveform.

Inverse transform

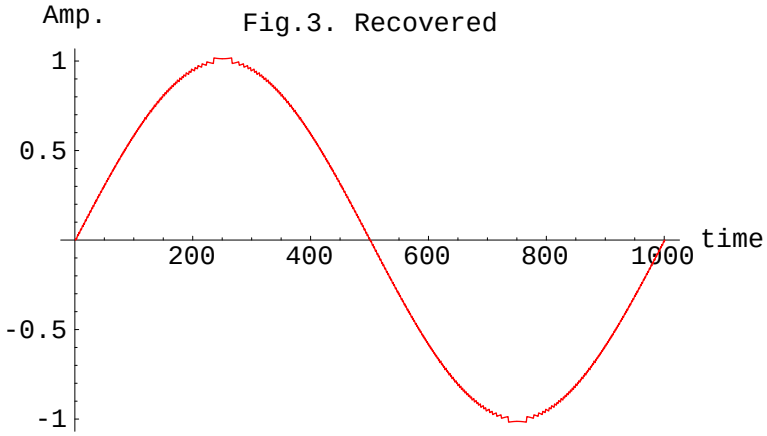
The recovering to the original time domain one-dimensional images from the eigen pattern depends on an inverse transform matrix. Because of the same eigen pattern, the time domain image is determined by which transform matrix to be used for constructing the inverse transform matrix. In this textbook, we recover the sinusoidal waveform from the eigen pattern of the co-sinusoidal waveform shown in Fig.1 (d).

At first, we construct the inverse transform matrix based on the transform matrix of the sinusoidal waveform in Fig. 1(a).

```
inverseMat=inverseEigenMatrix[eigenMat[[1]]];
```

Second, we recover the sinusoidal waveform from the eigen pattern of co-sinusoidal waveform in Fig 2(d). Figure 3 shows a recovered sinusoidal waveform.

```
sinR=inverseMat.eigenPatternM[[4]];
sinRG=ListPlot[sinR,PlotRange->All,PlotJoined->True,
    PlotStyle->RGBColor[1,0,0],AxesLabel->{"time","Amp."},
    PlotLabel->"Fig.3. Recovered"];
```



To check up recoverability, we compute a maximum absolute difference between the original and recovered one-dimensional image data.

```
Max[Abs[sample1D[[1]]-sinR]]
```

```
0.0218712
```

As you can see, about 2 percent error has been occurred by the finite resolution=100 limit. Finally, we remove the needless memories and check the used memories. Thereby, it is possible to know that the *Mathematica* front-end uses about 1.7 mega bytes memories.

```
Remove["sample1D","sample1DG","label","eigenMat","eigenPatternM",
    "eigenM1DG","inverseMat","sinR","sinRG"];
memoryUsed
```

```
1706K Bytes used
```

■ 5.3.2 Eigen pattern of a monochrome image

Sample images

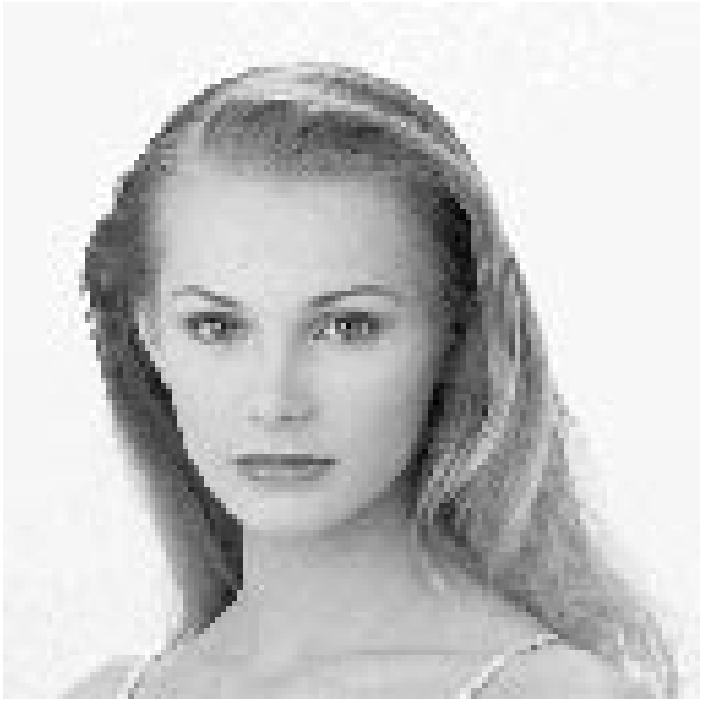
This section extracts the eigen patterns of the monochrome images. At first, we have to read in a sample image data by

```
sample=rgbBMP["BF001.bmp"];
```

After computing an array size and three-dimensional vector magnitude of the sample image, Fig. 4 shows the original monochrome sample image.

```
dim=Dimensions[sample];
monoSample=vectorMag3D[sample];
ListDensityPlot[monoSample,PlotRange->All,Mesh->False,Frame->False,
      PlotLabel->"Fig.4. Original monochrome sample"];
```

Fig.4. Original monochrome sample



In order to remove an effect of background in Fig. 4, we apply a round shape window to the sample image.

At first, a window function, we work out a 128 by 128 pixels list “win128” having a window's radius 50.

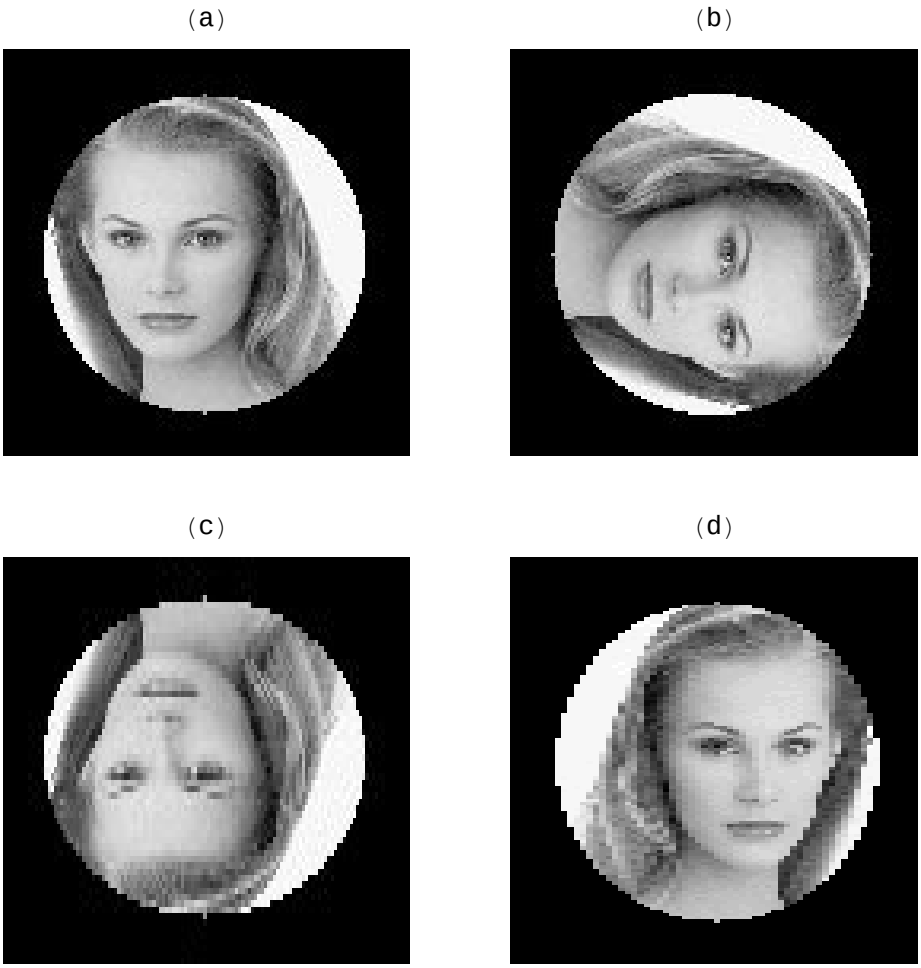
```
win128=window[128,128,50];
```

Convolution between the window and sample image data yields a window operated image. Further, we work out the other sample images based on the original sample in Fig. 4. Figure 5 shows the monochrome sample images. The differences among them are the angle and resolutions.

```
monoSampleW=win128*monoSample;
monoSampleWD={monoSampleW,Transpose[monoSampleW],
    Table[monoSampleW[[dim[[2]]-i+1,j]],{i,1,dim[[2]],2},{j,dim[[3]]}],
    Table[monoSampleW[[i,dim[[3]]-j+1]],{i,dim[[2]]},{j,1,dim[[3]],2}]};

label={"(a)","(b)","(c)","(d)"};
mSamplewDG=Table[ListDensityPlot[monoSampleWD[[i]],PlotRange->All,
    Mesh->False,Frame->False,AspectRatio->dim[[2]]/dim[[3]],
    PlotLabel->StringForm["`1`",label[[i]]],
    DisplayFunction->Identity],{i,4}];
```

```
Show[GraphicsArray[{{mSamplewDG[[1]],mSamplewDG[[2]]},
    {mSamplewDG[[3]],mSamplewDG[[4]]}}],
    PlotLabel->"Fig.5. Monochrome images",
    ImageSize->{400,400}];
```

Fig.5. Monochrome images



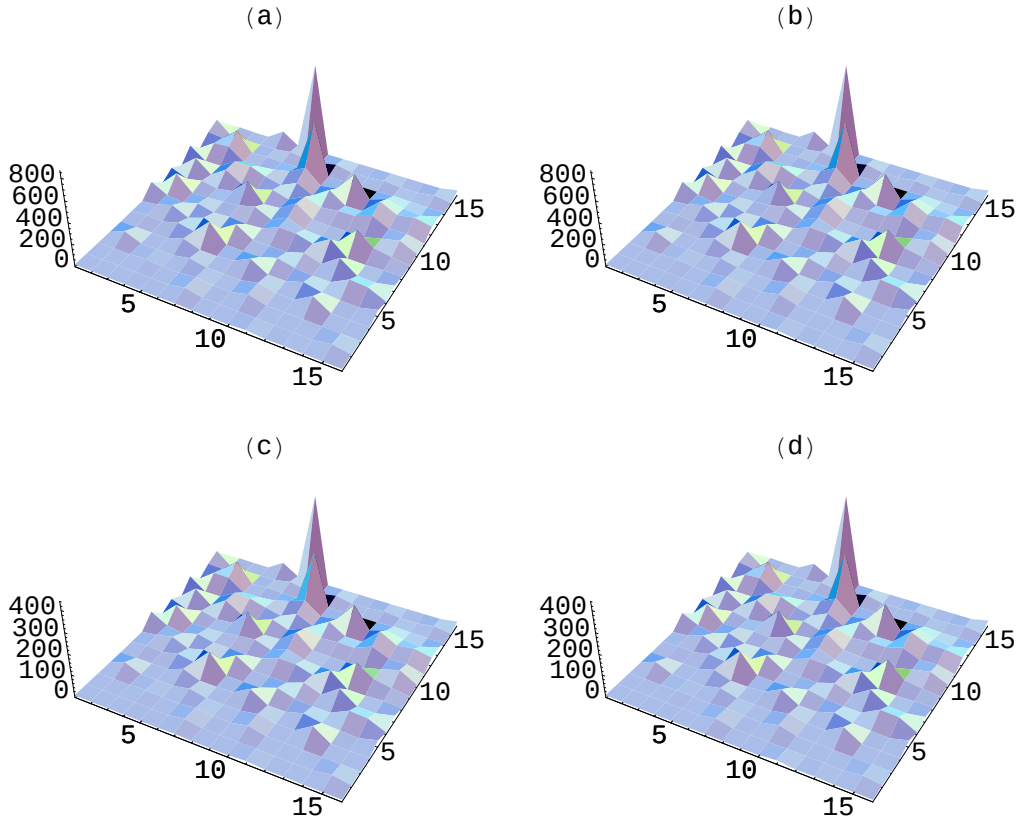
Eigen patterns

In order to extract the eigen patterns of the images in Fig. 5, we set the resolutions of the x- and y-axes to the values of 16 and 16, respectively. After rearranging the sample image data into one-dimensional form, we compute their eigen patterns. Consequently obtained eigen patterns take a one-dimensional form, so that we rearrange the eigen patterns into a two-dimensional form. Figure 6 shows the eigen patterns of the monochrome image samples in Fig. 5.

```
xReso=16;
yReso=16;
resolution=xReso*yReso;
eigen2D=Table[eigenPattern[Flatten[monoSampleWD[[i]]],
    resolution],{i,4}];

eigen2DG=Table[ListPlot3D[convert2D[eigen2D[[i]],xReso,yReso],
    PlotRange->All,Mesh->False,Boxed->False,
    PlotLabel->StringForm["`1`",label[[i]]],
    DisplayFunction->Identity],{i,4}];
```

```
Show[GraphicsArray[{{eigen2DG[[1]],eigen2DG[[2]]},{eigen2DG[[3]],
    eigen2DG[[4]]}}],PlotLabel->"Fig.6. Eigen patterns",
    ImageSize->{400,400}];
```

Fig.6. Eigen patterns



Even though the sample images in Fig. 5 are the different angled and resolutions, their eigen patterns are similar ones. The sample images having the same resolution take the same eigen pattern even if they are angled. The lower resolution sample images take the eigen patterns having smaller maximum amplitude.

The correlation coefficient computations between them suggest that all of the sample images in Fig. 5 are the same ones.

Thus, the meaning of the eigen pattern may be understood by every one. Before to continue the next section, we remove the needless variables and check the memories used.

```
Table[corRelation[eigen2D[[1]],eigen2D[[i]]],{i,2,4,1}]
```

```
{1., 0.998162, 0.998309}
```

```
Remove["monoSample","monoSampleW","mSamplewDG","eigen2DG"];
memoryUsed
```

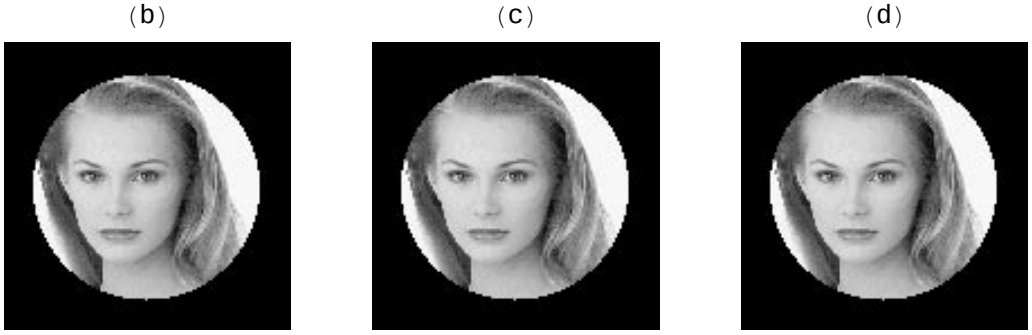
```
3414K Bytes used
```

Inverse transform

In this section, we generate or recover the monochrome images from the monochrome image eigen patterns in Fig.6. Similar to the one-dimensional image recovery, the recovered images are depended on the base image data. The base image data work as a guideline for which type image will be recovered from the same eigen pattern. We select the data representing a sample image of Fig. 5(a) as the base image data, after that we recover the monochrome images from the eigen patterns in Fig. 6(b)-(d). Figure 7 shows the recovered monochrome images formatted by the based image of Fig. 5(a). As you can see, we have just recovered the same images to the windowed one in Fig. 5(a).

```
recover2D=Table[inverseEigenPattern[eigen2D[[i]],
    Flatten[monoSampleWD[[1]]],resolution],{i,2,4,1}];

recover2DG=Table[ListDensityPlot[
    convert2D[recover2D[[i]],dim[[2]],dim[[3]]],
    PlotRange->All,Mesh->False,Frame->False,
    PlotLabel->StringForm["`1`",label[[i+1]]],
    DisplayFunction->Identity],{i,3}];

Show[GraphicsArray[recover2DG],PlotLabel->"Fig.7. Recovered images",
    ImageSize->{450,150}];
```

Fig.7. Recovered images



Similar to that of the one-dimensional images, the correlation coefficient computations between the images in Fig. 5(a) and in Fig. 7 suggest that all of the sample images in Fig. 5 are the same ones. Finally, we remove the needless variables and check the used memories.

```
Table[corRelation[Flatten[monoSampleWD[[1]]],recover2D[[i]]],{i,3}]
```

```
{0.999997, 0.999997, 0.999997}
```

```
Remove["monoSampleWD","recover2D","recover2DG","eigen2D"];
memoryUsed
```

```
2238K Bytes used
```

■ 5.3.3 Eigen pattern of a color image

Sample images

The sample data read in the *Mathematica* front-end are converted into the color image data format. After that, we have a color sample image as shown in Fig. 8.

```
Show[convertRGB[sample],AspectRatio->dim[[2]]/dim[[3]],
    PlotLabel->"Fig.8. Original color sample"];
```

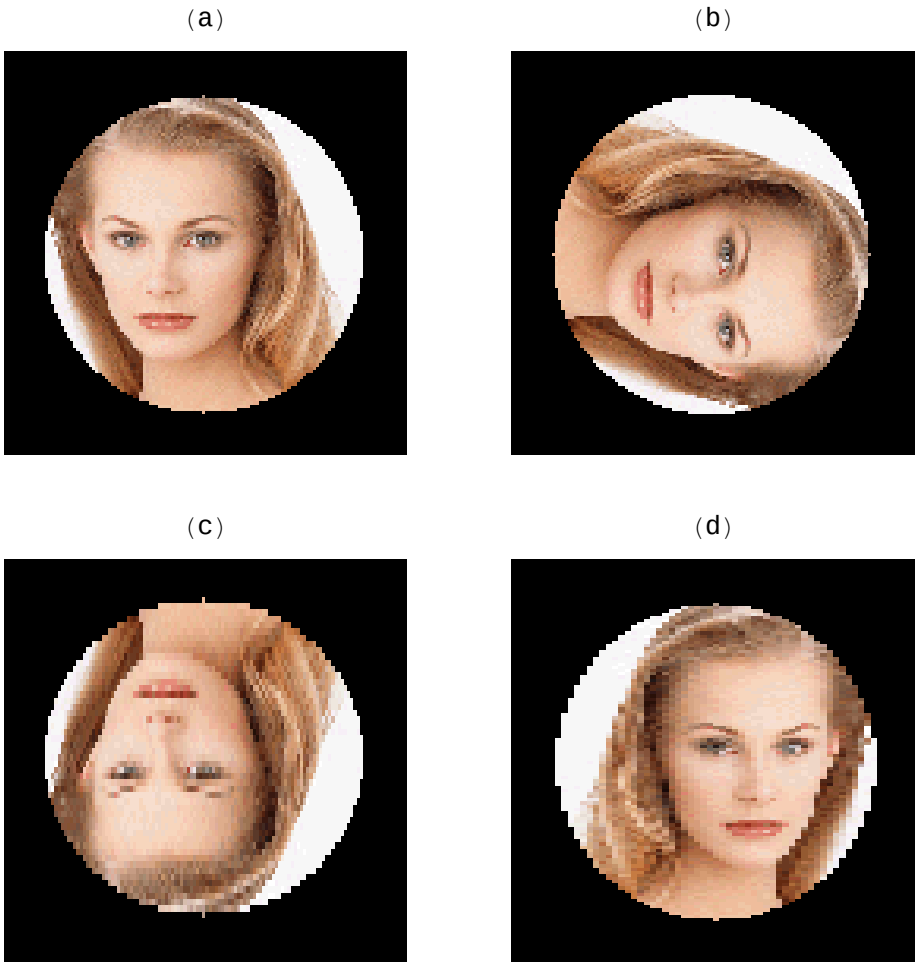
Fig.8. Original color sample

To reduce a background effect of the sample image, we employ a simple round shape of window. After window operation, we work out the angled and low resolution color image data. Figure 9 shows the color image samples.

```
sampleW=Table[win128*sample[[i]],{i,dim[[1]]}];
sampleWD={sampleW,Table[Transpose[sampleW[[i]]],{i,dim[[1]]}],
    Table[sampleW[[k,dim[[2]]-i+1,j]],
        {k,dim[[1]]},{i,1,dim[[2]],2},{j,dim[[3]]}],
    Table[sampleW[[k,i,dim[[3]]-j+1]],
        {k,dim[[1]]},{i,dim[[2]]},{j,1,dim[[3]],2}]};

samplewDG=Table[Show[convertRGB[sampleWD[[i]]],
    AspectRatio->dim[[2]]/dim[[3]],
    PlotLabel->StringForm["`1`",label[[i]]],
    DisplayFunction->Identity],{i,4}];
```

```
Show[GraphicsArray[{{samplewDG[[1]],samplewDG[[2]]},{samplewDG[[3]],
    samplewDG[[4]]}}],PlotLabel->"Fig.9. Color images",
    ImageSize->{400,400}];
```

Fig.9. Color images



We remove the needless variables, because the computations of color image eigen pattern require an enormous memories. Checking the used memories reveals that our *Mathematica* front-end is now using about 2.5 mega bytes memories. Even though a relatively small memory is used, next section may require a virtual memory use.

```
Remove["sample","sampleW","samplewDG"];
memoryUsed
```

```
2462K Bytes used
```

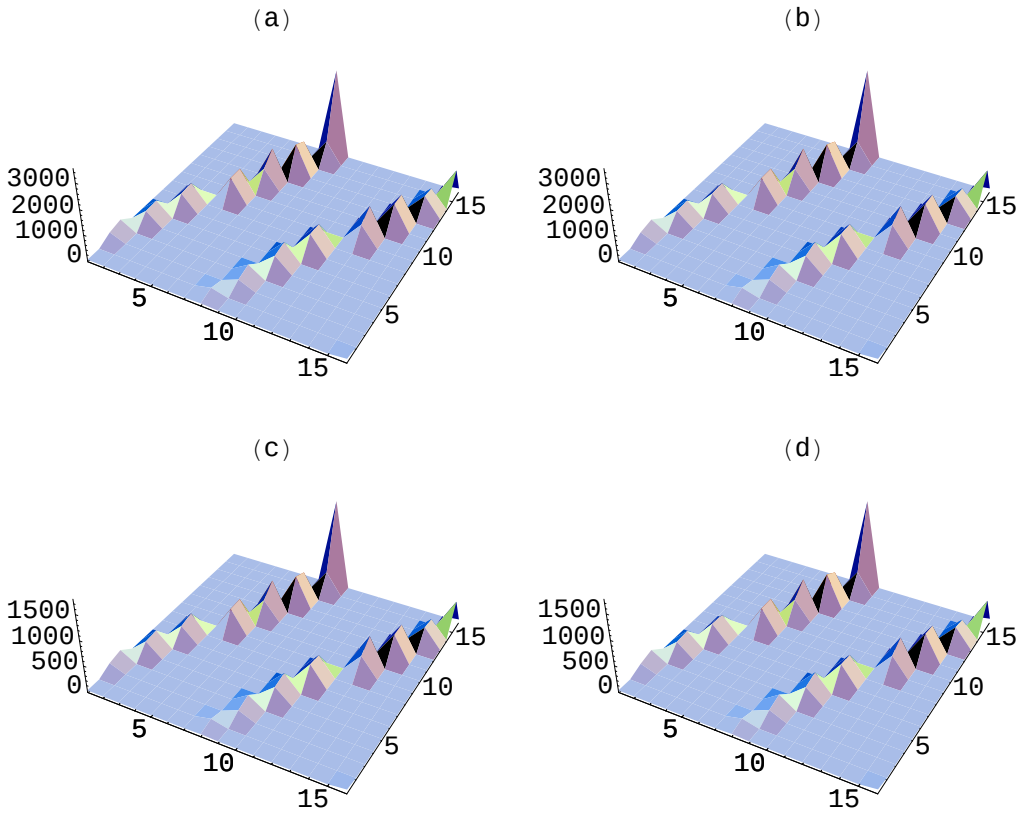
Eigen patterns

We compute the color image eigen patterns. The red, green and blue color components are not independently computed but simultaneously computed. This leads to a enormous memory. Figure 10 shows the color image eigen patterns. Surprisingly, all of the color images in Fig. 9 take the similar eigen patterns, even if their images are the angled and low-resolution ones.

```
eigen3D=Table[eigenPattern[Flatten[sampleWD[[i]]],
    resolution],{i,4}];
```

```
eigen3DG=Table[ListPlot3D[convert2D[eigen3D[[i]],xReso,yReso],
    PlotRange->All,Mesh->False,Boxed->False,
    PlotLabel->StringForm["`1`",label[[i]]],
    DisplayFunction->Identity],{i,4}];

Show[GraphicsArray[{{eigen3DG[[1]],eigen3DG[[2]]},{eigen3DG[[3]],
    eigen3DG[[4]]}}],PlotLabel->"Fig.10. Eigen patterns",
    ImageSize->{400,400}];
```

Fig.10. Eigen patterns



Similar to that of monochrome images, the sample images having the same resolution take the same eigen pattern even if they are angled. The lower resolution sample images take the eigen patterns having smaller maximum amplitude. However, computation of the correlation coefficients between them reveals that all of the sample images in Fig. 9 are the same ones.

Before to move on the next computations, we remove the needless variables and check the used memories.

```
Table[corRelation[eigen3D[[1]],eigen3D[[i]]],{i,2,4,1}]
```

```
{1., 0.999644, 0.999782}
```

```
Remove["eigen3DG","win128"];
memoryUsed
```

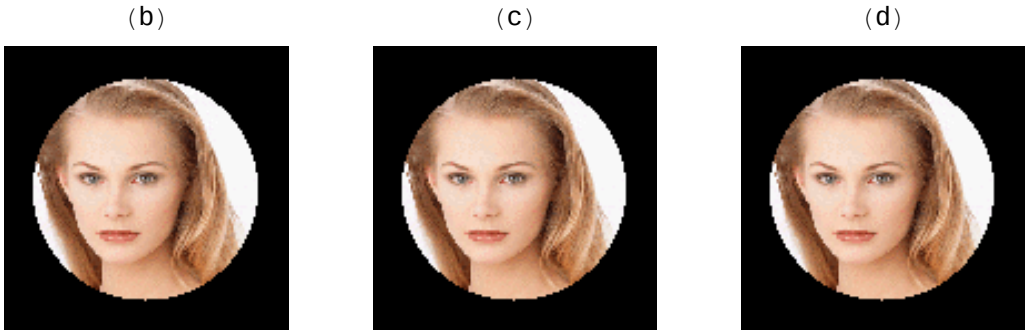
```
2462K Bytes used
```

Inverse transform

We generate or recover the color images from the color image eigen patterns in Fig.10. Similar to the one-dimensional and monochrome image recovery, the recovered images are depended on the base image data. Also described in the monochrome image recovery from their eigen patterns, the base image data work as a guideline for which type image will be recovered from the same eigen pattern. We select the data representing a sample image of Fig. 9(a) as the base image data. And then we recover the color images from the eigen patterns in Fig. 10(b)-(d). Figure 11 shows the recovered color images formatted by the base image in Fig. 9(a).

```
recover3D=Table[inverseEigenPattern[eigen3D[[i]],
    Flatten[sampleWD[[1]]],resolution],{i,2,4,1}];

k=dim[[2]]*dim[[3]];
recoverC=Table[Take[recover3D[[i]],{k*(j-1)+1,k*j}],
    {i,3},{j,dim[[1]]}];
color2D=Table[convert2D[recoverC[[i,j]],dim[[2]],dim[[3]]],
    {i,3},{j,dim[[1]]}];

recover3DG=Table[Show[convertRGB[color2D[[i]]],
    PlotLabel->StringForm["`1`",label[[i+1]]],
    AspectRatio->dim[[2]]/dim[[3]],
    DisplayFunction->Identity],{i,3}];

Show[GraphicsArray[recover3DG],PlotLabel->"Fig.11. Recovered images",
    ImageSize->{450,150}];
```

Fig.11. Recovered images



Computation of the correlation coefficients between the images in Fig. 9(a) and in Fig. 11 suggests that all of the sample images in Fig. 9 are the same ones. Finally, we remove the needless variables and check the used memories.

```
Table[corRelation[Flatten[sampleWD[[1]]],recover3D[[i]]],{i,3}]
```

```
{0.999998, 0.999998, 0.999998}
```

```
memoryUsed
```

```
4028K Bytes used
```

5.4 Summary

In this chapter, we have discussed about the image eigen pattern not the eigen values. Derivation of the eigen patterns has been based on the nonlinear transformation employing a rectangular transform matrix. Depending on the image as well as resolution, even if the transform matrix is a rectangular, we have successfully recovered the images from their eigen patterns. The wavelet transform has extracted one of the eigen patterns. The wavelet spectrum of an image has extracted the characteristics of the image but not extracted the common characteristics of among the images. On the other side, the method described in this chapter has been able to extract the common characteristics among the images. We have defined the eigen pattern representing the common image characteristics. According to this definition, it has been demonstrated that any angled and low-resolution images have the same eigen pattern as long as the angled and low-resolution images visualize the same object. In the other words, when keeping the distinct eigen pattern of an image, we can identify the image from the others. This means that the image eigen pattern defined above may have an enormous applications for the image identification, cognition and visualization

In the first section of this chapter, we have described a key idea deriving an eigen pattern. In the second section, we have described to the practical *Mathematica* codes deriving the eigen pattern. Also, we have examined the nature of eigen pattern employing several one-dimensional image examples, which are the time domain sinusoidal waves. The third section has derived an eigen pattern of the monochrome images. This has verified the angle and resolution independencies of the image eigen pattern. In addition to this section, the fourth section has derived the eigen pattern of color images, and recovered the original images from their eigen patterns.

■ REFERENCES

[1] Stephen Wolfram, *The Mathematica Book*, 3rd ed. (Wolfram Media/Cambridge University Press, 1996).

[2] J.D.Jackson, "Classical Electrodynamics 3rd Edition," John Wiley & Sons, New York (1998).

Chapter 6. Image Identification in Real Domain

6.1 Introduction

In the previous chapters, we described to the basic tools, sketch generation, three-dimensional angled lighting image generation, image governing equation, illusive image generation, wavelet image processing and eigen pattern of the computer graphics images based on the classical field theory. The image identification is one of the most important applications of the computer graphics, because it leads to work out the artificial human eyes. When we can realize the capability of human eyes information processing by computers, the most of the works requiring the information processing of the human eyes can be replaced by the computers equipped CCD. As is well known, our human society is composed of the cooperated works to maintain the social system, consuming products and peacekeeping. Many of them are relayed on the information processing ability of human eyes. Thus, one of the final targets of this book is aimed to construct a frame part of the artificial human eyes.

In this chapter, we try to identify the particular images in a large number of database images by means of the three different approaches. The first is a conventional correlation analysis. The other approaches are based on the inverse analysis methodologies. The inverse analysis is essentially reduced into solving for the ill posed linear system of equations. An inverse solution method yields a solution vector. In this solution vector, it is assumed that the most dominant or positive maximum element in the solution vector identifies a particular image. Also, a weighted sum of entire images generates a synthesized image, while the weights are the elements of solution vector. We employ the two inverse solution methodologies. One is a conventional least squares and the other is an iterative method so called vector generalized sampled pattern matching (vector GSPM in short). A difference between them is that the former always requires an inverse matrix but latter never require the inverse matrix. This means that the least squares can be applied only the limited cases, where a least squares projective operator is successfully evaluated, but the vector GSPM can be universally applicable to any types of ill posed system matrix.

The image identifications are carried out in the four different domains. The first is a real domain, the second is a Fourier spectrum domain, the third is a wavelet spectrum domain, and the fourth is an eigen pattern domain. In the Fourier spectrum domain, we carry out the image identifications using the absolute values of Fourier spectra in order to remove the spatial phase differences among the test and database images.

The solutions of ill posed system of equations are generally depending on a system structure and not uniquely evaluated. In the other words, a typical ill posed system of equations is that a number of equations is less or larger than those of the unknowns. This means that the former and latter have the infinitely large number of solutions and no solution exactly satisfying the entire equations, respectively. In most of the image identification problems, the number of equations and unknowns are corresponding to the number of equations and the database images. Thereby, the image identification problem is reduced into solving for an ill posed problem, which is composed of the larger number of equations than those of the unknowns. In such an ill posed system, it is difficult to evaluate the solution exactly satisfying the all equations. Thus, least square is one of the well-established methodologies. Even though, the least squares are well-established methodology, it requires, in essence, computing an inverse matrix. Because of too ill posed system matrix, we sometimes confront a singular matrix. In order to overcome this difficulty, we introduce an iterative solution technique so called "vector GSPM". This iterative solution strategy is described in detail in this chapter.

6.2 Preparation of *Mathematica*

■ 6.2.1 *Mathematica* utilities and packages

Before to move on the practical computations, we have to install the memory conserve utilities and the warning messages suppressing. In addition *Mathematica* utilities, the " LinearAlgebra ' MatrixManipulation" package has to be installed [1].

```
<<Utilities`MemoryConserve`
$MemoryIncrement=100000;
Off[General::spell1,MemoryConserve::start,MemoryConserve::end];
<< LinearAlgebra`MatrixManipulation`;
```

■ 6.2.2 *Mathematica* functions

Here, we define several functions that are described and used in the previous chapters. The functions "convertRGB", "corRelation", "imageNormalize", "memoryUsed", "wavelet base functions", "waveletMatrix", waveletND" and "eigenPattern" have been defined in the previous chapters, so that the comments of such functions are not described.

Function convertRGB

Function window

Function corRelation

Function imageNormalize

Function memoryUsed

Wavelet base functions

Function waveletMatrix

Function waveletND

Function eigenPattern

6.3 Graphics image system of equations

■ 6.3.1 Input vector

Let us consider a test color image $I_{n\times n}$ with n by n resolution:

$$I_{n\times n} \in f_r(x_i, y_j),\ f_g(x_i, y_j),\ f_b(x_i, y_j)$$

$$i=1,2,.,n, \qquad j=1,2,.,n \tag{1}$$

where the functions f_r, f_g, f_b refer to the red, green and blue components; x_i, y_j denote the on x-axis and on y-axis locations of a pixel, respectively.

Arranging the pixels of image $I_{n\times n}$ into a column-wise form gives an input vector $\mathbf{Y}$ with *-th* order as

$$\begin{aligned}\mathbf{Y} = [&f_r(x_1, y_1),\ f_r(x_2, y_2),\ .,\ f_r(x_n, y_1),\ f_r(x_1, y_2),\ f_r(x_2, y_2),\ .,\\ &f_r(x_n, y_2),\ .,\ f_r(x_{n-1}, y_n),\ f_r(x_n, y_n),\\ &f_g(x_1, y_1),\ f_g(x_2, y_2),\ .,\ f_g(x_n, y_1),\ f_g(x_1, y_2),\ f_g(x_2, y_2),\ .,\\ &f_g(x_n, y_2),\ .,\ f_g(x_{n-1}, y_n),\ f_g(x_n, y_n),\\ &f_b(x_1, y_1),\ f_b(x_2, y_2),\ .,\ f_b(x_n, y_1),\ f_b(x_1, y_2),\ f_b(x_2, y_2),\ .,\\ &f_b(x_n, y_2),\ .,\ f_g(x_{n-1}, y_n),\ f_g(x_n, y_n)]^T\end{aligned} \tag{2}$$

■ 6.3.2 . System matrix

Let us assume the p-th m by m database images:

$$C^{(k)}{}_{m\times m} \in g^{(k)}{}_r(x_i, y_j),\ g^{(k)}{}_g(x_i, y_j),\ g^{(k)}{}_b(x_i, y_j)$$

$$i=1,2,.,m, \qquad j=1,2,.,m, \qquad k=1,2,.,p, \tag{3}$$

where the functions $g^{(k)}{}_r$, $g^{(k)}{}_g$, $g^{(k)}{}_b$ refer to the red, green and blue components of the database image, respectively.

By means of the wavelet transform, the database images with m by m resolution are reduced into the images with n by n resolution as same as the test image one. Denoting the database images with n by n resolution by

$$C^{(k)}{}_{n\times n} \in g^{(k)}{}_r(x_i, y_j),\ g^{(k)}{}_g(x_i, y_j),\ g^{(k)}{}_b(x_i, y_j)$$

$$i=1,2,.,n, \qquad j=1,2,.,n, \qquad k=1,2,.,p, \tag{3}$$

k-th vector of a system matrix is given by

$$\begin{aligned}\mathbf{d}^{(k)} = [&g_r(x_1, y_1),\ g_r(x_2, y_2),\ .,\ g_r(x_n, y_1),\ g_r(x_1, y_2),\ g_r(x_2, y_2),\ .,\\ &g_r(x_n, y_2),\ .,\ g_r(x_{n-1}, y_n),\ g_r(x_n, y_n),\\ &g_g(x_1, y_1),\ g_g(x_2, y_2),\ .,\ g_g(x_n, y_1),\ g_g(x_1, y_2),\ g_g(x_2, y_2),\ .,\\ &g_g(x_n, y_2),\ .,\ g_g(x_{n-1}, y_n),\ g_g(x_n, y_n),\\ &g_b(x_1, y_1),\ g_b(x_2, y_2),\ .,\ g_b(x_n, y_1),\ g_b(x_1, y_2),\ g_b(x_2, y_2),\ .,\\ &g_b(x_n, y_2),\ .,\ g_g(x_{n-1}, y_n),\ g_g(x_n, y_n)]^T\end{aligned} \tag{4}$$

Thus, a system matrix with $3\times n\times n$ *-th* rows and *p-th* columns is given by

$$D = [\mathbf{d}^{(1)}, \mathbf{d}^{(2)}, .,\ \mathbf{d}^{(p)}]. \tag{5}$$

■ 6.3.3 System of equations

Denoting a solution vector **X** with order *p*, a system of graphics image equations is formally written by

$$\mathbf{Y}=D\mathbf{X}. \quad (6)$$

■ 6.3.4 Least squares

In most case, the number of equations is much greater than those of unknowns *p*, so that it is possible to apply a conventional least squares mean to Eq. (6) [2]:

$$\mathbf{X} = [D^T D]^{-1} D^T \mathbf{Y} \quad (7)$$

By considering the input vector **Y** in Eq. (6) and the column vector **d** in Eq. (5), it is revealed that the elements in the solution vector **X** are the weights $w_i (i=1,2, . ,p)$ to the database images. This means a sythersized image *S* is composed of

$$S=\sum_{i=1}^{p} w_i C_i, \quad (8)$$

where C_i is the database image .

When w_1 = 1and the other weights are zero in Eq.(8), then it is obvious that the test image is the same to the first database image . In the other words, the test image is identified as the first database image. A *Mathematica* function "leastSQ" gives a least squares solution.

Mathematica function leastSQ

This function "leastSQ" gives a least squares solution of the ill posed system. In order to use this function, it is essental that a number of equations is larger than those of unknows, and a product between the transpose of system and original system matrices should be a positive definite square matrix. The parameters "systemMat" and "vector" are the system matrix and input vector, respectively.

```
leastSQ = Compile[{{systemMat, _Real, 2}, {vector, _Real, 1}},
  Module[{tMat = {{0.}}, matP = {{0.}}},
   tMat = Transpose[systemMat];
   matP = Inverse[tMat . systemMat];
   matP . tMat . vector]];
```

■ 6.3.5 Vector generalized sampled pattern matching method

The product between the transpose of system D^T and original system *D* matrices sometimes not becomes to be a positive definite square matrix. In such a case, it is difficult to apply the least squares to Eq. (6). To overcome this difficulity, we describe here an iterative solution method called "vector GSPM".

Normalized system of equations

Eq. (6) can be rewritten by

$$\mathbf{Y}=\sum_{i=1}^{p} x_i \mathbf{d}_i,$$

$$\mathbf{X} = [x_1, x_2, .(, x)_p]^T. \quad (9)$$

Further modification to Eq. (9) becomes

$$\frac{\boldsymbol{Y}}{|\boldsymbol{Y}|} = \sum_{i=1}^{p} \frac{|\mathbf{d}_i|}{|\boldsymbol{Y}|} \frac{\mathbf{d}_i}{|\mathbf{d}_i|}, \qquad \text{or} \qquad \mathbf{Y}'=D'\mathbf{X}' \tag{10}$$

Eq. (3) means that the normalized input vector Y' is always given by a linear combination of the weighted solutions with normalized column vectors $\frac{\mathbf{d}_1}{|\mathbf{d}_1|}, \frac{\mathbf{d}_2}{|\mathbf{d}_2|}, .., \frac{\mathbf{d}_p}{|\mathbf{d}_p|}$.

Objective function

Eq. (2) means that the input vector **Y** is always given by means of a linear combination of the column vector $\mathbf{C}_i$ (i=1,2,.,m). Therefore, when an angle between the input vectors of **Y** and of $C\mathbf{X}^{(k)}$ given in terms of the *k-th* iterative solution $\mathbf{X}^{(k)}$ is defined by

$$h(\boldsymbol{X}^{(k)}) = \frac{\boldsymbol{Y}}{|\boldsymbol{Y}|} \frac{D\mathbf{X}^{(k)}}{|D\mathbf{X}^{(k)}|}$$

$$= \frac{\boldsymbol{Y}}{|\boldsymbol{Y}|} \frac{|\boldsymbol{Y}|}{|\boldsymbol{Y}|} \frac{D\mathbf{X}^{(k)}}{|D\mathbf{X}^{(k)}|} = \frac{\boldsymbol{Y}}{|\boldsymbol{Y}|} \frac{\sum_{i=1}^{p} x_i^{(k)} \frac{|\mathbf{d}_i|\mathbf{d}_i}{|Y||\mathbf{d}_i|}}{\left|\sum_{i=1}^{p} x_i^{(k)} \frac{|\mathbf{d}_i|\mathbf{d}_i}{|Y||\mathbf{d}_i|}\right|}$$

$$=\mathbf{Y}' \frac{D'\,\boldsymbol{X}'^{(k)}}{|D'\,\boldsymbol{X}'^{(k)}|}, \tag{11}$$

then

$$h(\boldsymbol{X}^{(k)}) \to 1, \tag{12}$$

means that the solution vector $X'^{(k)}$ satisfies the Eq. (10), *i.e.*,

$$\boldsymbol{Y}' = D'\,\mathbf{X}'^{(k)}. \tag{13}$$

When an initial solution vector $\mathbf{X}'^{(0)}$ is given by

$$\boldsymbol{X}'^{(0)} = D'^{T}\,\mathbf{Y}', \tag{14}$$

then the first deviation to the normalized input vector **Y'** becomes

$$\Delta\mathbf{Y}'^{(1)}=\mathbf{Y}'- \frac{D'\,\mathbf{X}'^{(0)}}{|D'\,\mathbf{X}'^{(0)}|}, \tag{15}$$

By means of Eqs.(13) and (14), the *k-th* iterative solution vector $\mathbf{X}'^{(k)}$ is given by

$$\mathbf{X}'^{(k)} = \mathbf{X}'^{(k-1)} + D'^{T}\Delta\mathbf{Y}'^{(k-1)}$$

$$= \mathbf{X'}^{(k-1)} + \boldsymbol{D'^{T}} \left[\mathbf{Y'} - \frac{D'\,\mathbf{X'}^{(k-1)}}{|D'\,\mathbf{X'}^{(k-1)}|} \right]$$

$$= D'^{T}\,\mathbf{Y'} + \left[I_p - \frac{D'\,\mathbf{X'}^{(k-1)}}{|D'\,\mathbf{X'}^{(k-1)}|} \right] \mathbf{X'}^{(k-1)} \qquad (16)$$

Convergence condition

Convergence of the iterative scheme Eq. (16) should be examined by considering a state transition matrix *S* from the solution vectors $\mathbf{X}^{(k-1)}$ to $\mathbf{X}^{(k)}$ in Eq. (16):

$$T = I_p - \frac{D'\,\mathbf{X'}^{(k-1)}}{|D'\,\mathbf{X'}^{(k-1)}|} \quad \bullet \qquad (17)$$

When the maximum eigen value of *T* is less than 1, then the solution is always converged to an exact solution vector. However, the state transition matrix *T* in Eq. (10) is not a constant but function of the solution vector . This means that the convergence depends on the solution vector $\mathbf{X}^{(k)}$. As is well known the eigen values of a unit square matrix are the multiple roots of 1. The convergence condition of our problem is described by

$$|I_p| \geq |T|,$$

$$\left| I_p \right| \geq I_p - \frac{D'\,\mathbf{X'}^{(k-1)}}{|D'\,\mathbf{X'}^{(k-1)}|}$$

or

$$|D'\,\mathbf{X'}^{(k-1)}|\,|I_p| \geq |\boldsymbol{D'}\,\mathbf{X'}^{(\boldsymbol{k}-1)}|\,|I_{\boldsymbol{p}} - \boldsymbol{D'^{T}}\,\boldsymbol{D'}| \,. \qquad (18)$$

In Eq . (18), all of the diagonal elements in the matrix $D'^{T}\,D'$ are 1, and the other off-diagonal elements of this matrix are always less than 1. Thereby, the convergence condition is always held. This means that Eq. (16) gives an absolutely stable iterative solution. A following *Mathematica* function "vectorGSPM" gives this iterative solution [3,4].

***Mathematica* function vectorGSPM**

The parameters of this functions are as follows:

systemMatrix: arbitraly rectangular *n* by *m* system matrix,

inputVector: input vector with order *n*,

iteration: maximum number of iterations,

if this is zero or negative integer, the maximum number of iterations is set to *m.*

The output of this function is given in terms of one-dimensional array, which is composed of two parts. The first header part is the solution vector and remaining last part is the pattern matching figures.

```
vectorGSPM=
    Compile[{{systemMatrix,_Real,2},{inputVector,_Real,1},
        {iteration,_Integer}},
    Module[{m=Length[systemMatrix[[1]]],defaultIteration=10 m,
        innerProduct=N[{}],outputVector={0.},dummy={0.},
        systemMatrixPrime={{0.}},normalizedInput={0.},
        transposedSystemPrime={{0.}},columnNorm={0.}},

        normalizedInput=
            inputVector/Sqrt[inputVector.inputVector];
        transposedSystemPrime=Transpose[systemMatrix];
        columnNorm=Table[Sqrt[transposedSystemPrime[[i]].
            transposedSystemPrime[[i]]],{i,m}];
        If[iteration>0, defaultIteration=iteration];
        transposedSystemPrime=
            Table[transposedSystemPrime[[i]]/columnNorm[[i]],{i,m}];
        systemMatrixPrime=Transpose[transposedSystemPrime];
        outputVector=transposedSystemPrime.normalizedInput;

        Do[dummy=((dummy=systemMatrixPrime.outputVector)/
                Sqrt[dummy.dummy]);
            innerProduct=Join[innerProduct,{normalizedInput.dummy}];
            outputVector=outputVector+transposedSystemPrime.
                (normalizedInput-dummy),
        {defaultIteration}];

        dummy=systemMatrixPrime.outputVector;

        If[Abs[Max[dummy]-Min[dummy]]<2.22044604925031308 10^-16,
            Print["Solution has no pattern !"]];

        Join[(Sqrt[inputVector.inputVector]/Sqrt[dummy.dummy])*
            outputVector/columnNorm,innerProduct]
    ]];
```

6.4 Modeling

■ 6.4.1 Test image

We read in the test image data from a data file "testRGB10.m".

```
test=<<"imageTST10.m";//Timing
```

```
{6.16 Second, Null}
```

After checking a size of read image data “testRGB10.m”, Fig 1 shows the test images, which are identified from the 63 database images in Fig. 2 below.

```
dimT=Dimensions[test];
testG=Table[Show[convertRGB[test[[i]]],
    AspectRatio->dimT[[3]]/dimT[[4]],
    PlotLabel->StringForm["T`1`",i],
    DisplayFunction->Identity],{i,dimT[[1]]}];

Show[GraphicsArray[Table[Table[testG[[i+j]],{j,0,4}],{i,1,dimT[[1]]-4,5}]],
    PlotLabel->"Fig.1. Test images",
    ImageSize->{85*5,90*2}];
```

Fig.1. Test images

■ 6.4.2 Database images

Original database image

In this section, we read in the database images. Each of the database images is composed of the 128 by 128 pixels, and is not a distinct image but overlapped images.

```
dataBase = << "imageDB63.m"; // Timing
```

```
{126.71 Second, Null}
```

```
dimDB=Dimensions[dataBase]
```

```
{63, 3, 128, 128}
```

Resolution adjustment of the database image

The reolution of the test images in Fig.1 is 64 by 64 pixels, so that the resolution of the database is higher than those of test ones. In order to set up the same resolution database images to the test ones, we construct the low-resolution database images by a simple 2 by 2 pixels averaging.

```
dbColor= Table[0.25*(dataBase[[i,j,k,l]]+dataBase[[i,j,k+1,l]]+
    dataBase[[i,j,k,l+1]]+dataBase[[i,j,k+1,l+1]]),
    {i,dimDB[[1]]},{j,dimDB[[2]]},{k,1,dimDB[[3]],2},{l,1,dimDB[[4]],2}];
```

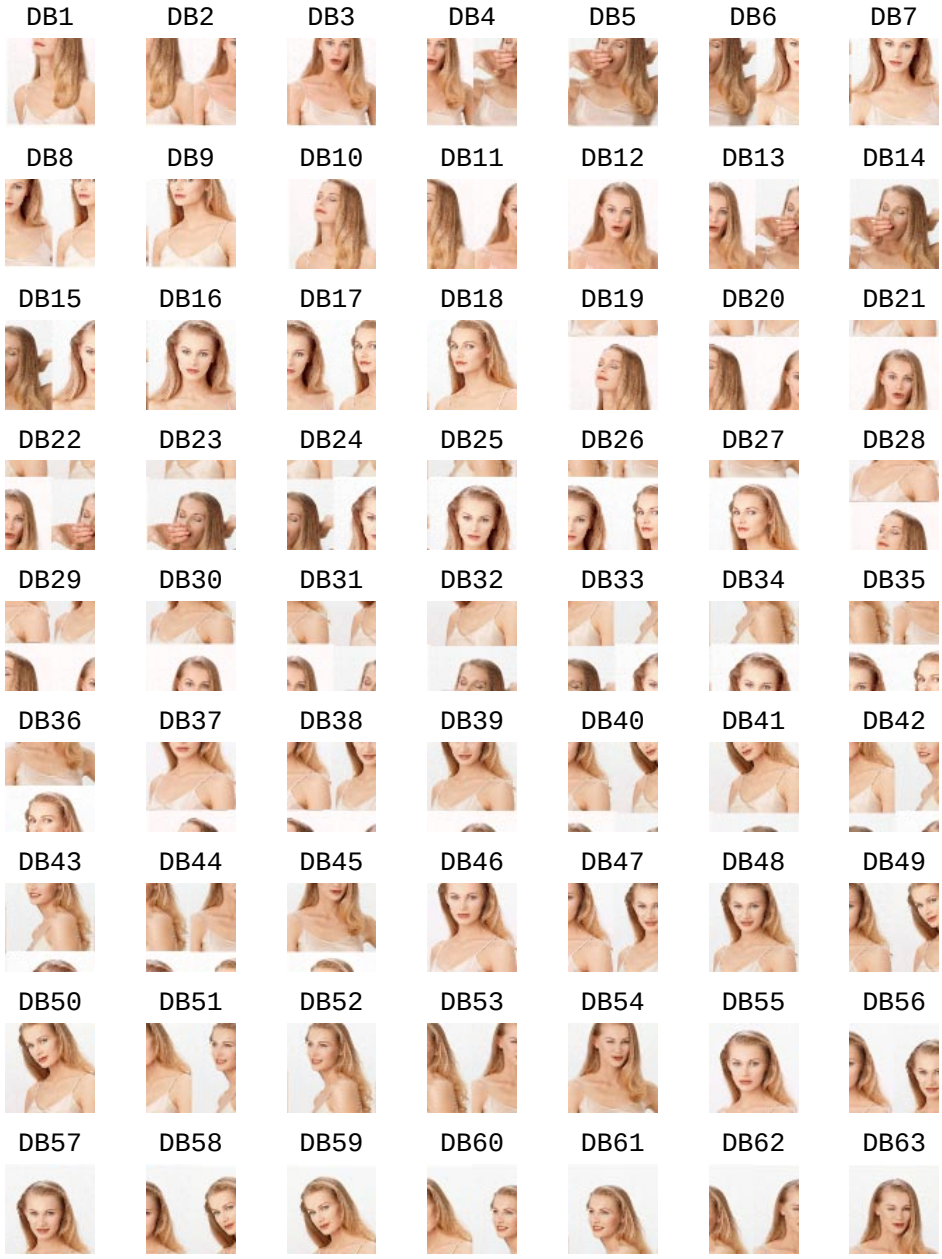
We compute the image data of the modified database.

```
dbColorG = Table[Show[convertRGB[dbColor[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["DB`1`",i]],{i,dimDB[[1]]}];
```

Figure 2 shows the modified 63 database images. Our problem is to find an each of the test images in Fig. 1 from these database images.

```
Show[GraphicsArray[Table[Table[dbColorG[[i + j]], {j, 0, 6}],
  {i, 1, dimDB[[1]] - 6, 7}]], ImageSize -> {64 * 7, 64 * 8},
  PlotLabel -> "Fig.2. Modified database images"];
memoryUsed
```

Fig.2. Modified database images




```
14682K Bytes used
```

6.5 Image identification in real domain

In this section, we search for the image containing the same as those of test from the database images in Fig. 2. At first, we compute the correlation coefficients between the test and database images. The maximum correlation coefficient reveals the identified image. Second and third approaches are the inverse solution strategies. This means that we set up the image system of equations, and then we solve them by the least squares and vector GSPM means. Both approaches provide the solution vectors. Taking the maximum element in the solution vector gives an identified image. Further, combination of the solution vector and database images generates the synthesized images.

■ 6.5.1 Correlation analysys

Data arrangement

In order to compute the correlation coefficients between the test and database images, we rearrange the image data in column-wise form.

```
baseMat=Table[Flatten[dbColor[[i]]],{i,dimDB[[1]]}];
testV=Table[Flatten[test[[i]]],{i,dimT[[1]]}];
```

Correlation coefficients

Compute the correlation coefficients between the test and database image data.

```
corCoe=Table[corRelation[baseMat[[i]],testV[[j]]],
    {j,dimT[[1]]},{i,dimDB[[1]]}];
```

Taking the maximum correlation coefficient to each of the test images gives the identified images by the correlation analysis.

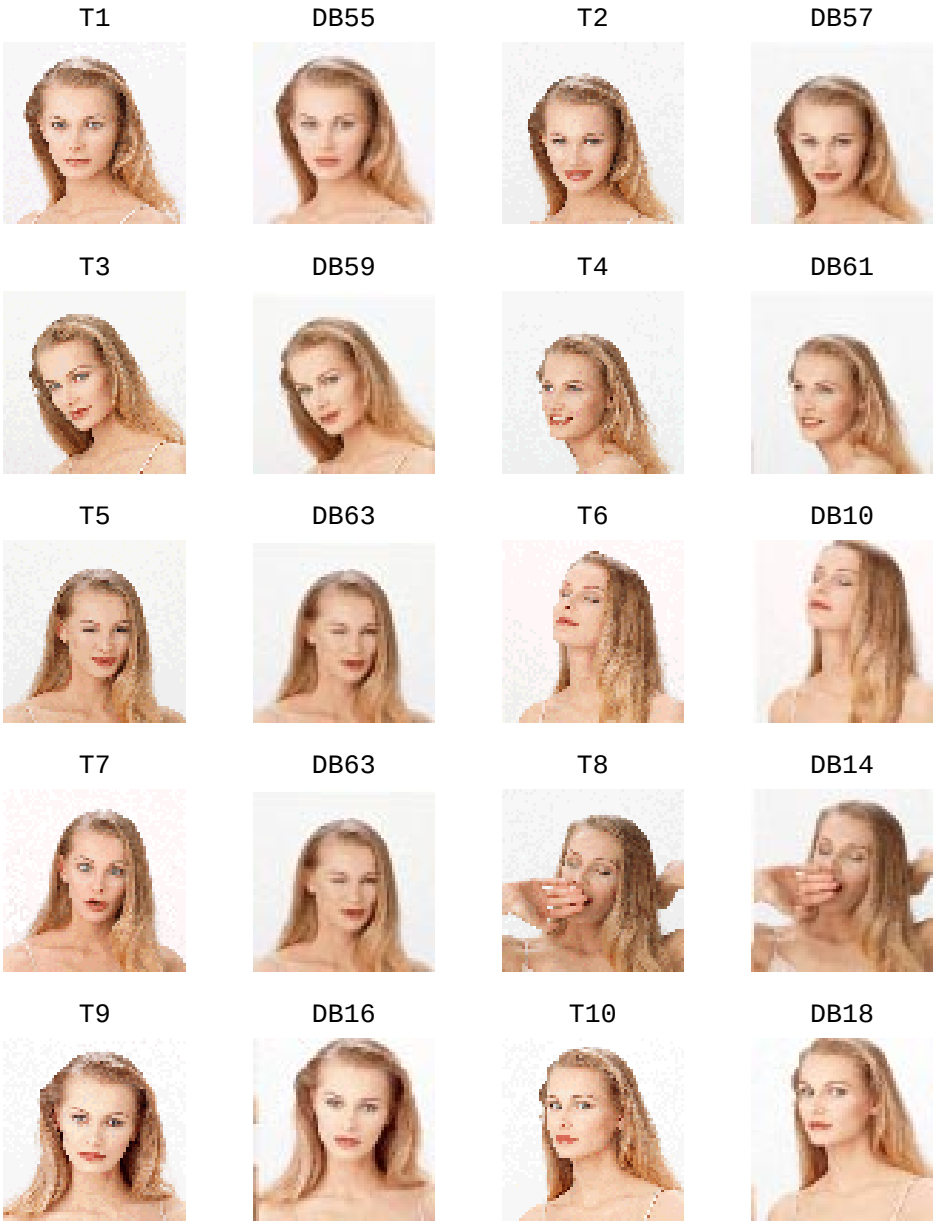
```
identified=Table[Position[corCoe[[i]],Max[corCoe[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 10, 63, 14, 16, 18}
```

Figure 3 shows the identified images along with test ones. As you can see, the fairly good results have obtained. Only one test image “T7” was not exactly identified.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.3. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
Remove["corCoe","identified"];memoryUsed
```

Fig.3. Test(T) and identified(DB) images



18305K Bytes used

■ 6.5.2 Least squares

System matrix

We construct a system matrix by transposing the “baseMat” used for the correlation analysis.

```
systemMat=Transpose[baseMat];
```

Least squares solution

Compute the least squares solutions.

```
solution=Table[leastSQ[systemMat,testV[[i]]],{i,dimT[[1]]}];//Timing
```

```
{149.07 Second, Null}
```

Compute the image data of the least squares solutions.

```
solutionG=Table[ListPlot[solution[[i]],
    PlotRange->All,PlotJoined->True,
    PlotLabel->StringForm["LS`1`",i],
    DisplayFunction->Identity],{i,dimT[[1]]}];
```

Figure 4 shows the solution vectors. Obviously, the test images “T1”-“T5” were exactly identified, but remaining test images “T6”-”T10” were doubtful results. This means that it is difficult to represent the test images “T6-T10” by the simple linear combination of the database images in Fig. 2.

```
Show[GraphicsArray[
    Table[{solutionG[[i]],solutionG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.4. Least squares solutions"];
```

Fig.4. Least squares solutions

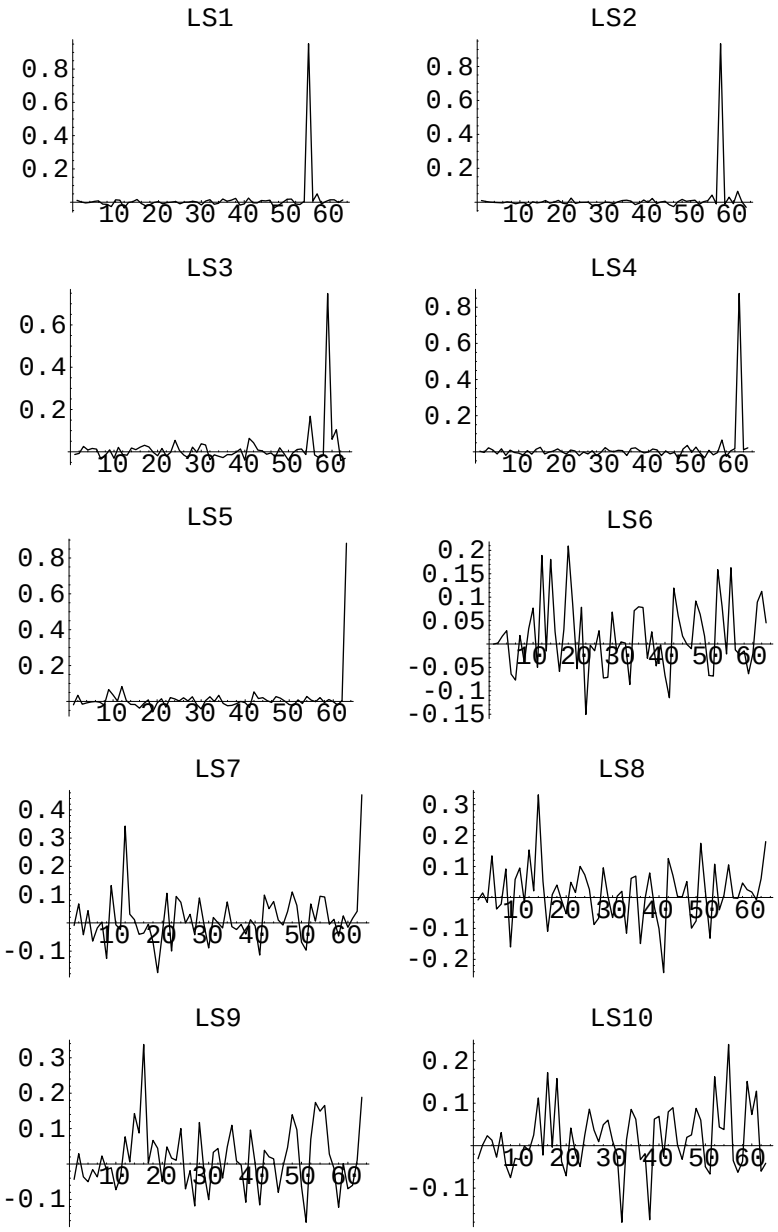


Image identification

Taking the maximum elements in the solution vectors, we can obtain the identified images by least squares.

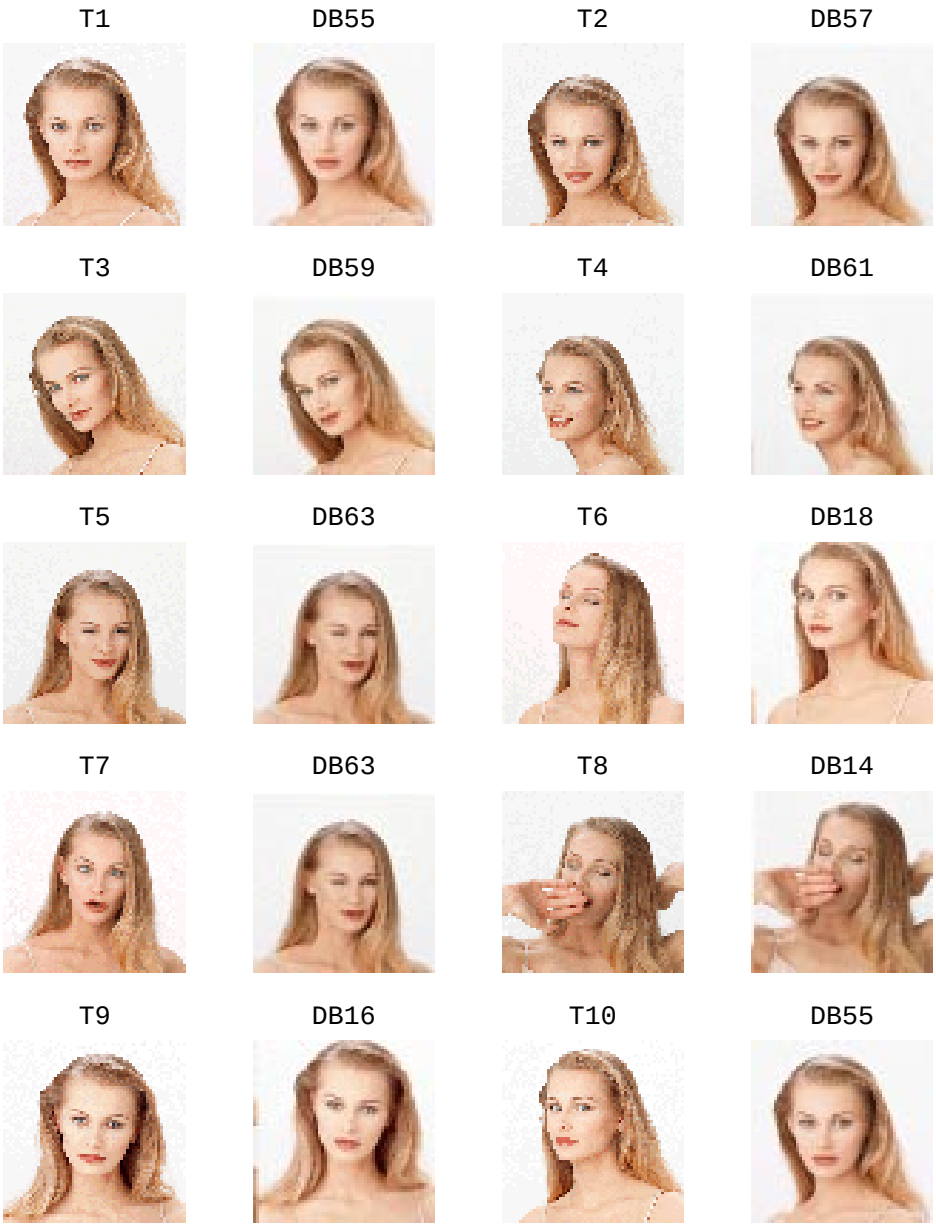
```
identified=Table[Position[solution[[i]],Max[solution[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 18, 63, 14, 16, 55}
```

Figure 5 shows the identified images by means of the least squares. Even though, the test images “T6-T10” were not represented by the linear combination of the database images, over 70% is successfully identified.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.5. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
memoryUsed
```

Fig.5. Test(T) and identified(DB) images




```
21892K Bytes used
```

Image synthesize

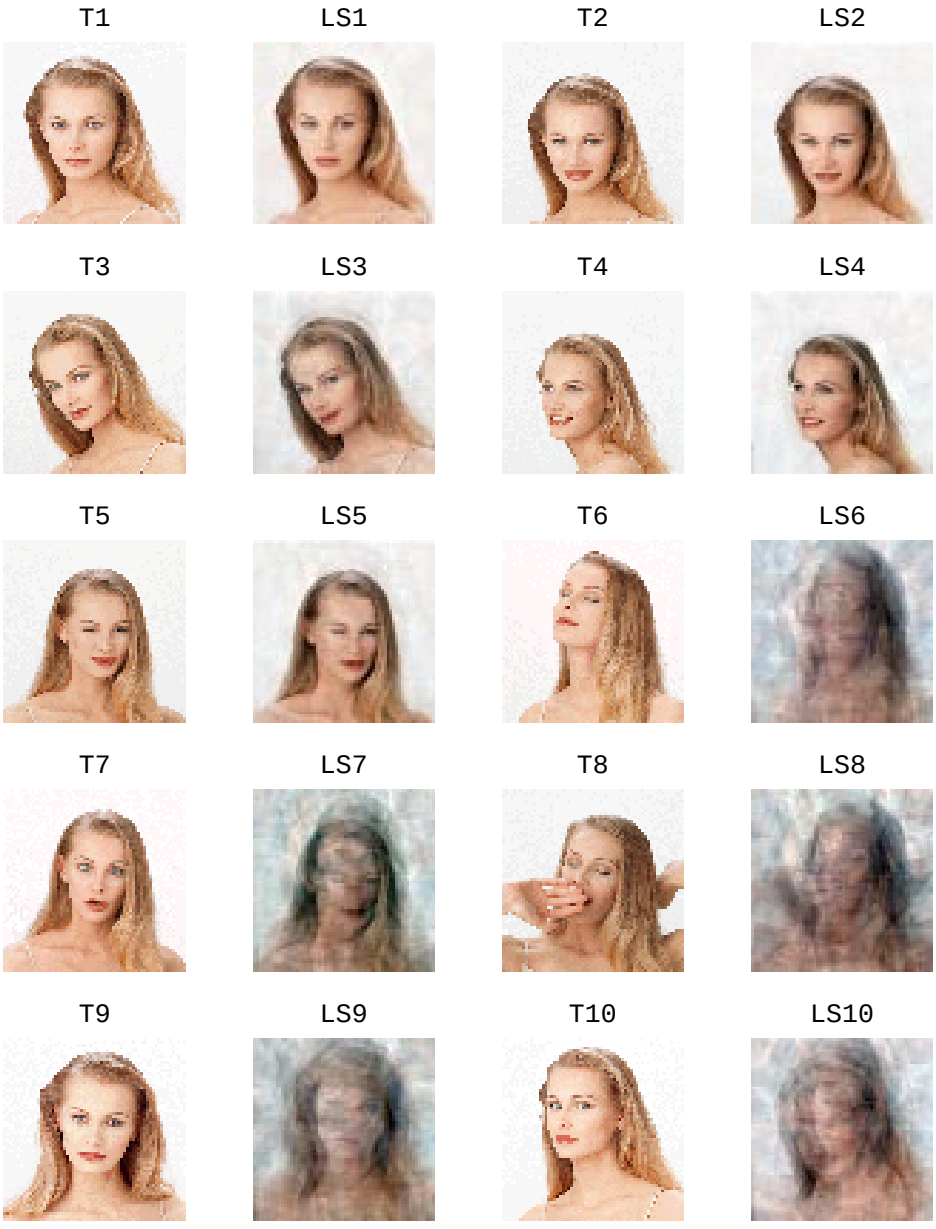
By means of Eq. (8), we synthesize the image satisfying the image system of equations in a least square sense.

```
comLS=Table[Sum[solution[[i,j]]*dbColor[[j]],
    {j,dimDB[[1]]}],{i,dimT[[1]]}];

comLSN=Table[imageNormalize[comLS[[i,j]]],{i,dimT[[1]]},{j,dimT[[2]]}];

comLSG=Table[Show[convertRGB[comLSN[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["LS`1`",i]],{i,dimT[[1]]}];
```

Figure 6 shows the synthesized images along with the input test images. In accordance with the solution vectors shown in Fig. 4, the images “T1-T5” were well synthesized but the others were poor results.

```
Show[GraphicsArray[
    Table[{testG[[i]],comLSG[[i]],testG[[i+1]],comLSG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.6. Test(T) and sythesized(LS) images",
    ImageSize->{4*100,5*100}];
Remove["solution","solutionG","identified","comLS","comLSN","comLSG"];
memoryUsed
```

Fig.6. Test(T) and sythesized(LS) images




```
21889K Bytes used
```

■ 6.5.3 Vector GSPM

Vector GSPM solutions

The image system of equations has a larger number of equations than those of unknowns so that it is an ill posed system. In the other words, there are no solutions satisfying all of the equations, simultaneously. This means that the solution vector of the image system of equation depends on the solution strategy. To clarify the differences between the solution methods, we employ the vector GSPM method, which has promising results to the physical ill posed system of equations[3,4].

```
solution=Table[vectorGSPM[systemMat,testV[[i]],500],{i,dimT[[1]]}];//Timing
```

```
{652.24 Second, Null}
```

After a relatively long computation time, we classify the solutions into the solution and pattern matching figure parts.

```
sol = Table[Take[solution[[i]], dimDB[[1]]], {i, dimT[[1]]}];
matF = Table[Take[solution[[i]], -500], {i, dimT[[1]]}];
```

Figure 7 shows the convergence processes to the test images. As described in Eq. (18), any solution vectors by the vector GSPM method have been converged to the fixed vectors. The pattern-matching figure “Ganma” in Fig. 7 corresponds to the value of objective function of Eq. (12), so that the value near to 1 means a goodness of the solutions.

```
convG=Table[ListPlot[matF[[i]],
    PlotRange->All,PlotJoined->True,AxesLabel->{"Itas.","Ganma"},
    PlotLabel->StringForm["SPM`1`",i],DisplayFunction->Identity],
    {i,dimT[[1]]}];
```

```
Show[GraphicsArray[
    Table[{convG[[i]],convG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.7. Convergence processes"];
```

Fig.7. Convergence processes

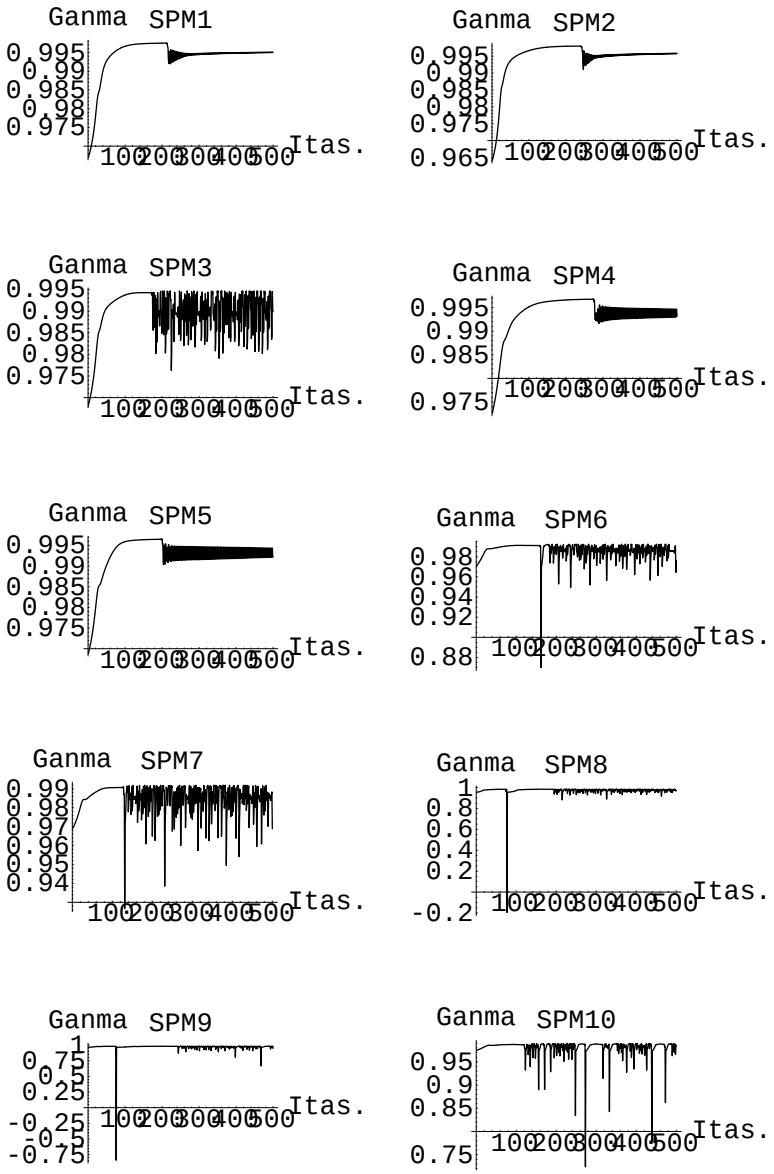


Figure 8 shows the solution vectors by the vector GSPM method. Similar to these of least squares, the test images "T1-T5" were successfully identified but the other images were not represented by the linear combination of the database images in Fig. 2.

```
solutionG=Table[ListPlot[sol[[i]],
    PlotRange->All,PlotJoined->True,
    PlotLabel->StringForm["SPM`1`",i],DisplayFunction->Identity],
    {i,dimT[[1]]}];
```

```
Show[GraphicsArray[
    Table[{solutionG[[i]],solutionG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.8. Vector GSPM solutions"];
```

Fig.8. Vector GSPM solutions

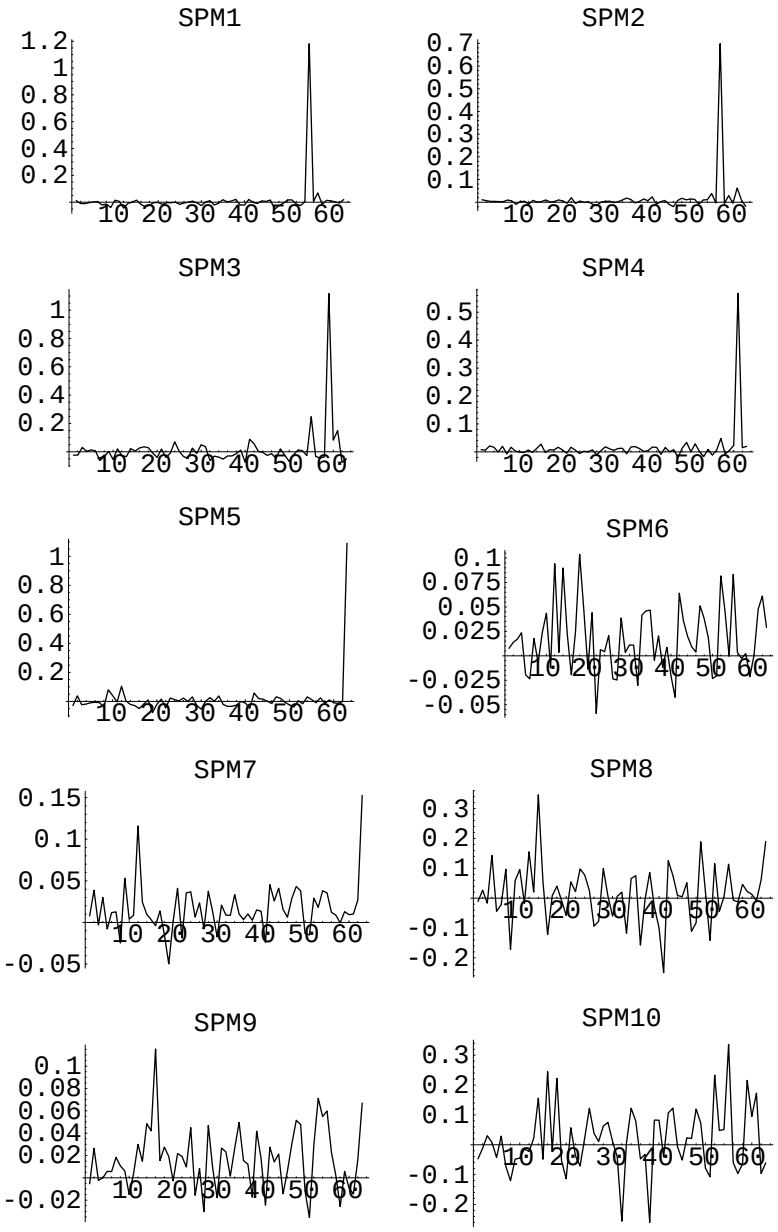


Image identification

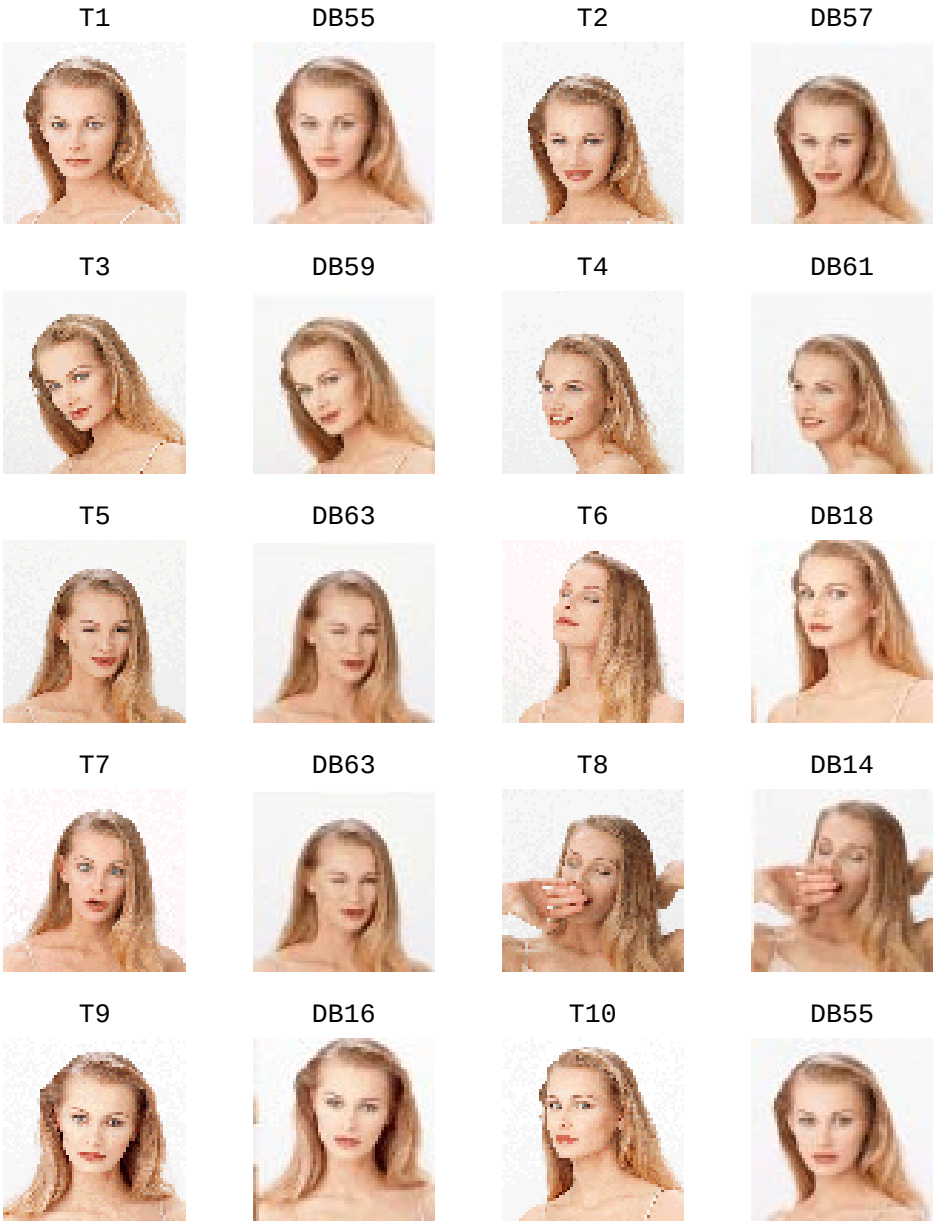
Taking the maximum elements in the solution vectors in Fig. 8 gives the identified images by the iterative means. Figure 9 shows the identified images. The seven test images were exactly identified.

```
identified=Table[Position[sol[[i]],Max[sol[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 18, 63, 14, 16, 55}
```

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.9. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
memoryUsed
```

Fig.9. Test(T) and identified(DB) images




```
22333K Bytes used
```

Image synthesize

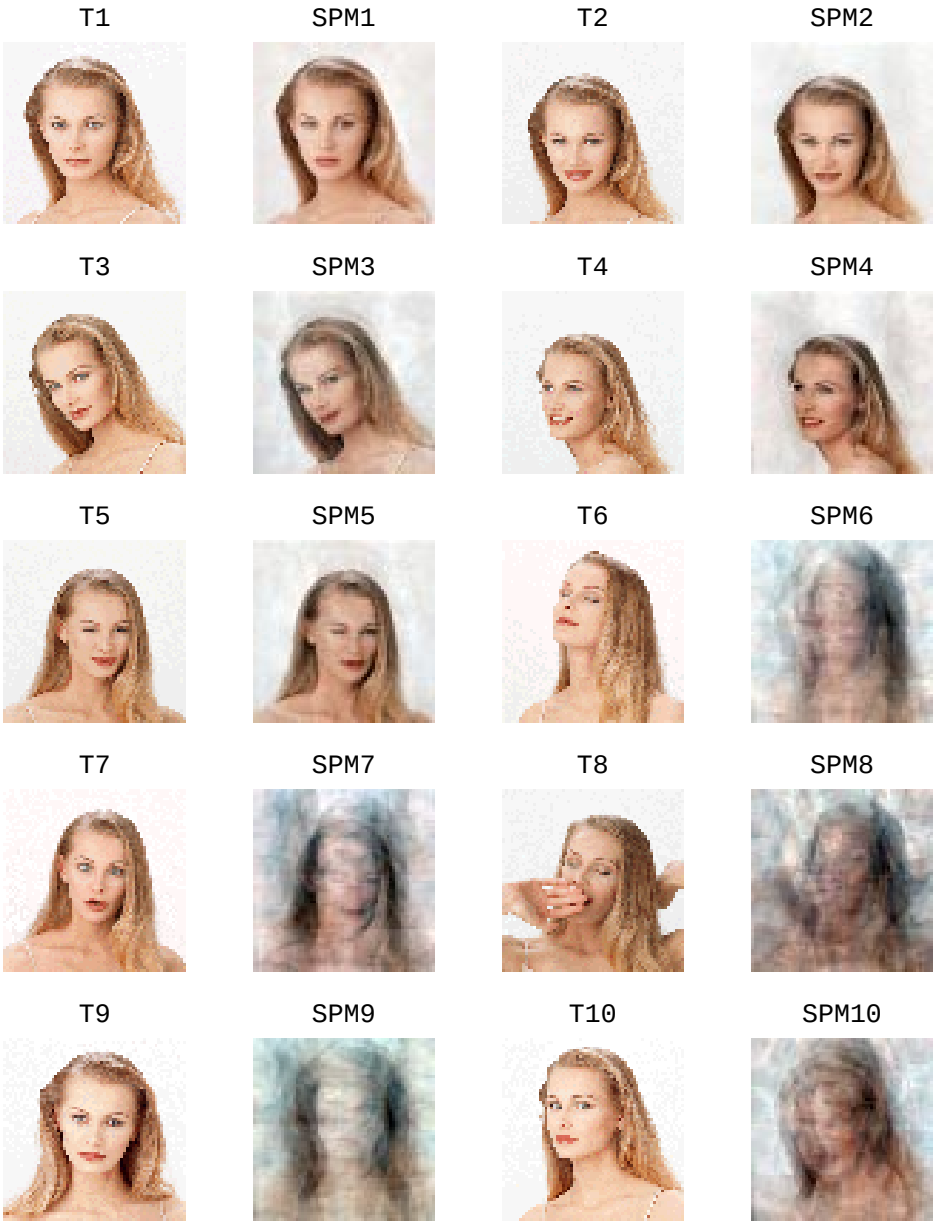
Similar to those of the least squares, it is possible to synthesize the images, which suggest the solvability of the ill posed system of equations. When we can synthesize a clear image, it means that the system could be solved with satisfactory accuracy.

```
comLS=Table[Sum[sol[[i,j]]*dbColor[[j]],
    {j,dimDB[[1]]}],{i,dimT[[1]]}];

comLSN=Table[imageNormalize[comLS[[i,j]]],{i,dimT[[1]]},{j,dimT[[2]]}];

comLSG=Table[Show[convertRGB[comLSN[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["SPM`1`",i]],{i,dimT[[1]]}];
```

Figure 10 shows the synthesized images by means of the iterative solutions. As expected from the solution vectors in Fig. 8, the test images “T1-T5” were clearly synthesized similar to the test images in Fig. 1.

```
Show[GraphicsArray[
    Table[{testG[[i]],comLSG[[i]],testG[[i+1]],comLSG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.10. Test(T) and sythesized(GSPM) images",
    ImageSize->{4*100,5*100}];
Remove["systemMat","solution","solutionG",
    "identified","comLS","comLSN","comLSG"];memoryUsed
```

Fig.10. Test(T) and sythesized(GSPM) images



22216K Bytes used

6.6 Image identification in Fourier spectrum domain

In the previous section, we carried out the image identifications by means of the correlation as well as inverse approaches in the real domain. The correlation analysis provided a fairly good result but the others did not so good results. One of the reasons why the inverse approaches could not provide the good results is that the target position in each of the database images is not always coincided with those of the test images. On of the methods to remove this difficulty is to employ the Fourier transform. The image data are represented in terms of the complex spatial frequencies. A combination of the real and imaginary parts having the same spatial frequency represents a position of the image having such the spatial frequency. In the other words, when we take the absolute values of the complex frequencies, it is possible to remove the spatial phase difference.

Thus, we have to try the image identification in the Fourier spectrum or spatial frequency domain.

■ 6.6.1 Correlation analysys

Fourier transform

At first, we compute the Fourier spectra of the test and database images.

```
baseMat=Table[Flatten[Table[TakeMatrix[
    Abs[Fourier[dataBase[[i,j]]]],{1,1},{dimDB[[3]]/4,dimDB[[4]]/4}],
    {j,dimDB[[2]]}]],{i,dimDB[[1]]}];

testV=Table[Flatten[Table[TakeMatrix[
    Abs[Fourier[test[[i,j]]]],{1,1},{dimT[[3]]/2,dimT[[4]]/2}],
    {j,dimT[[2]]}]],{i,dimT[[1]]}];
```

Correlation coefficients

Compute the correlation coefficients between the test and database image data.

```
corCoe=Table[corRelation[baseMat[[i]],testV[[j]]],
    {j,dimT[[1]]},{i,dimDB[[1]]}];
```

Taking the maximum correlation coefficient to each of the test images gives the identified images by the correlation analysis.

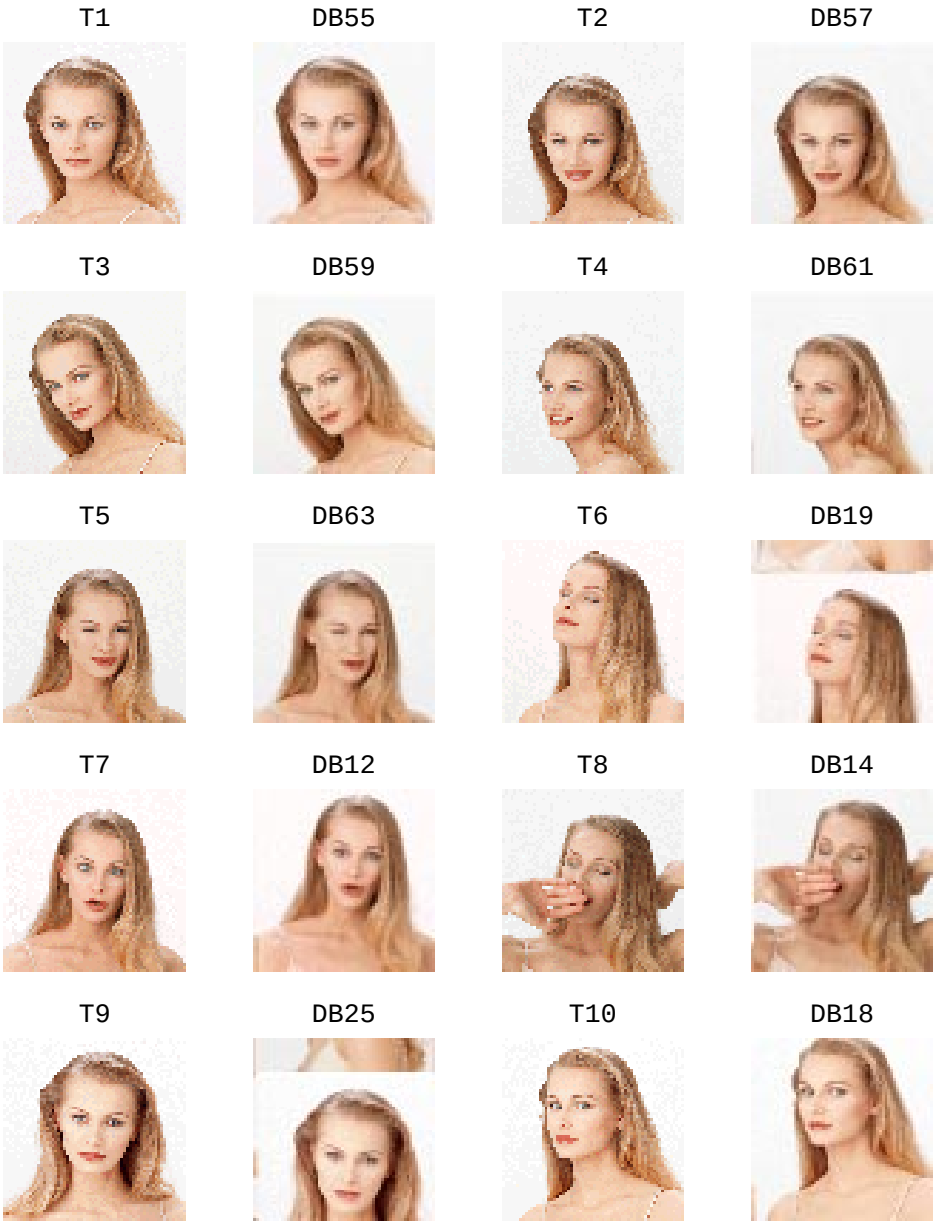
```
identified=Table[Position[corCoe[[i]],Max[corCoe[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 19, 12, 14, 25, 18}
```

Figure 11 shows the identified images along with test ones. In the Fourier spectrum domain, the correlation analysis gives an improved result even though somewhat spatial phase differences are observed.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.11. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
Remove["corCoe","identified"];memoryUsed
```

Fig.11. Test(T) and identified(DB) images



19686K Bytes used

■ 6.6.2 Least squares

System matrix

Transposing the “baseMat” used for the correlation analysis yields a system matrix.

```
systemMat=Transpose[baseMat];
```

Least squares solution

Compute the least squares solutions.

```
solution=Table[leastSQ[systemMat,testV[[i]]],
    {i,dimT[[1]]}];//Timing
```

```
{41.3 Second, Null}
```

Compute the image data of the least squares solutions.

```
solutionG=Table[ListPlot[solution[[i]],
    PlotRange->All,PlotJoined->True,
    PlotLabel->StringForm["LS`1`",i],
    DisplayFunction->Identity],{i,dimT[[1]]}];
```

Figure 12 shows the solution vectors. Obviously, the test images “T1”-“T5” were exactly identified, but remaining test images “T6”-”T10” were doubtful results. This means that the it is difficult to represent the test images “T6-T10” by the simple linear combination of the database images in Fig. 2, even if the Fourier spectrum domain.

```
Show[GraphicsArray[
    Table[{solutionG[[i]],solutionG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.12. Least squares solutions"];
```

Fig.12. Least squares solutions

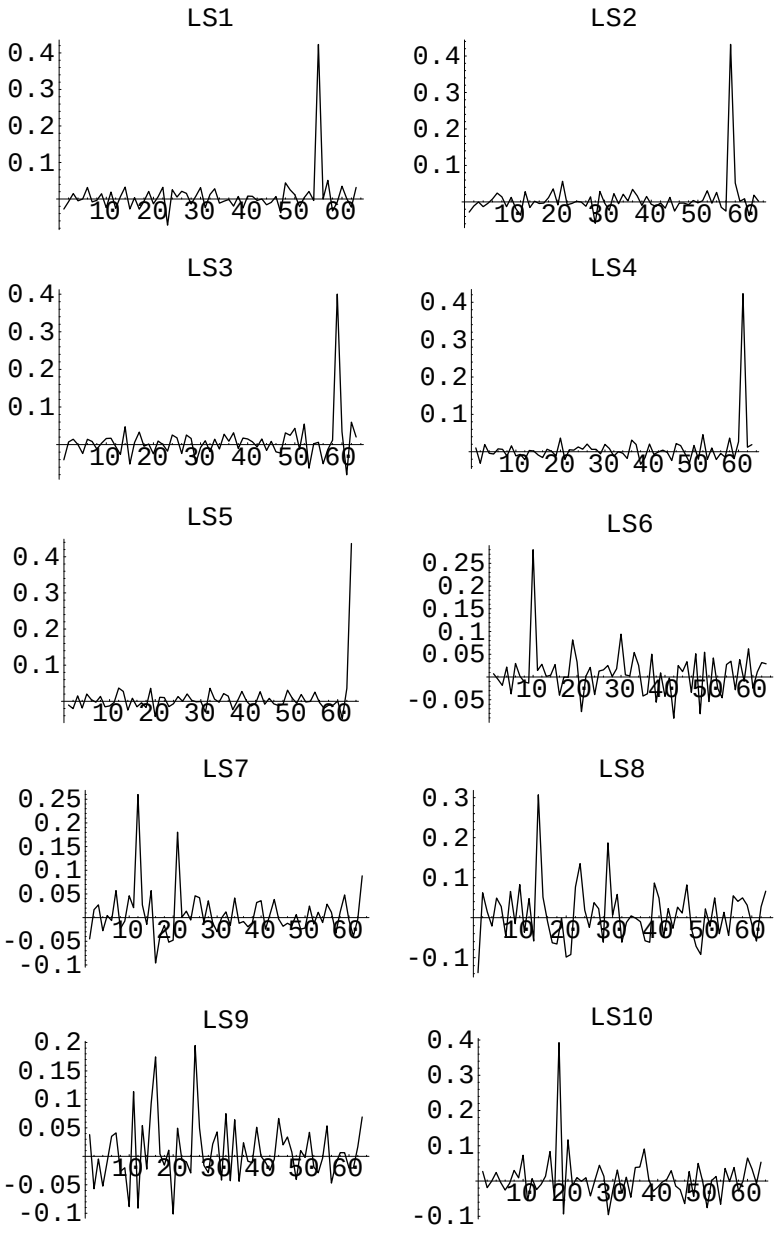


Image identification

Taking the maximum elements in the solution vectors, we can obtain the identified images by least squares.

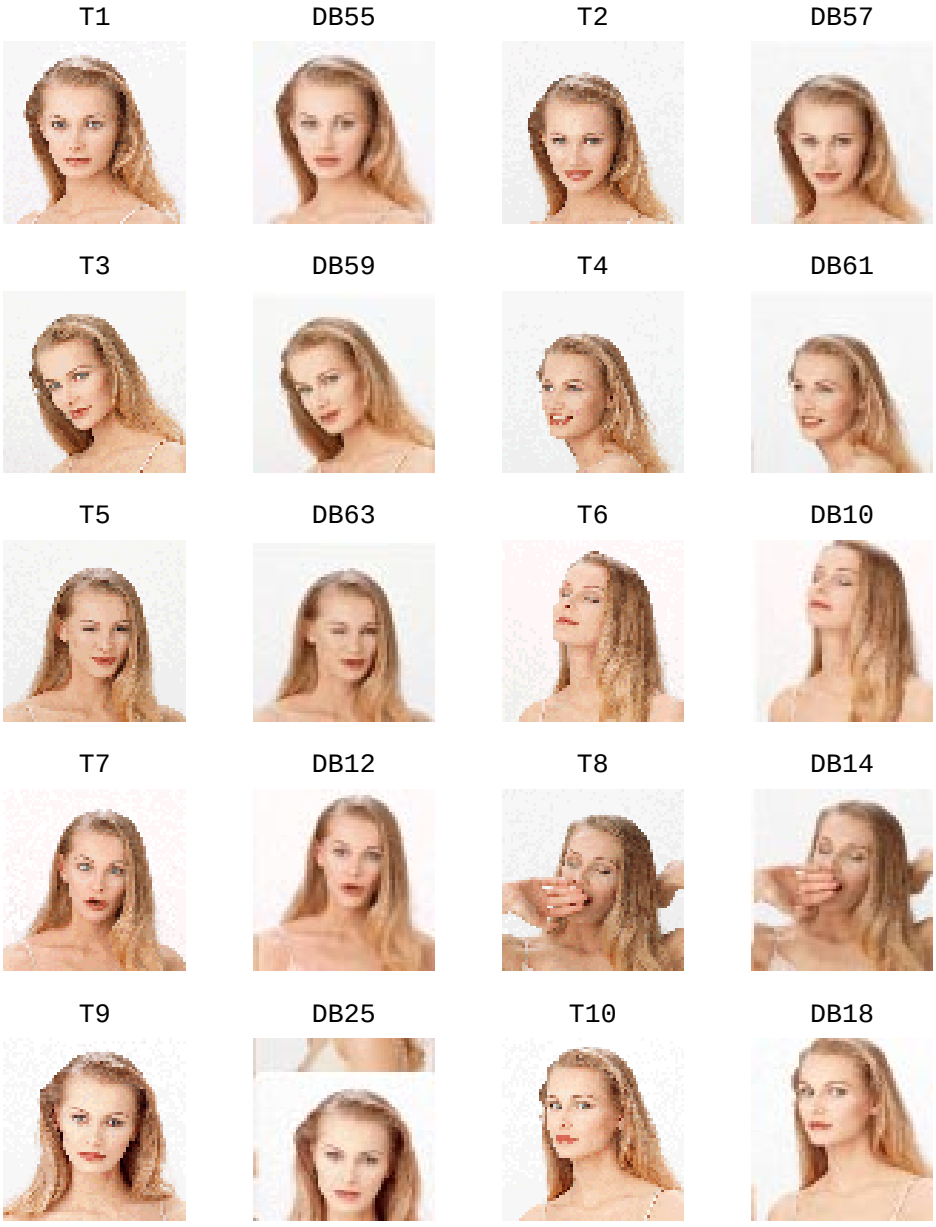
```
identified=Table[Position[solution[[i]],Max[solution[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 10, 12, 14, 25, 18}
```

Figure 13 shows the identified images by means of the least squares. Surprisingly, all of the test images were successfully identified. Further, the identified images are better than that of the correlation analysis.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.13. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
memoryUsed
```

Fig.13. Test(T) and identified(DB) images




```
20658K Bytes used
```

Image synthesize

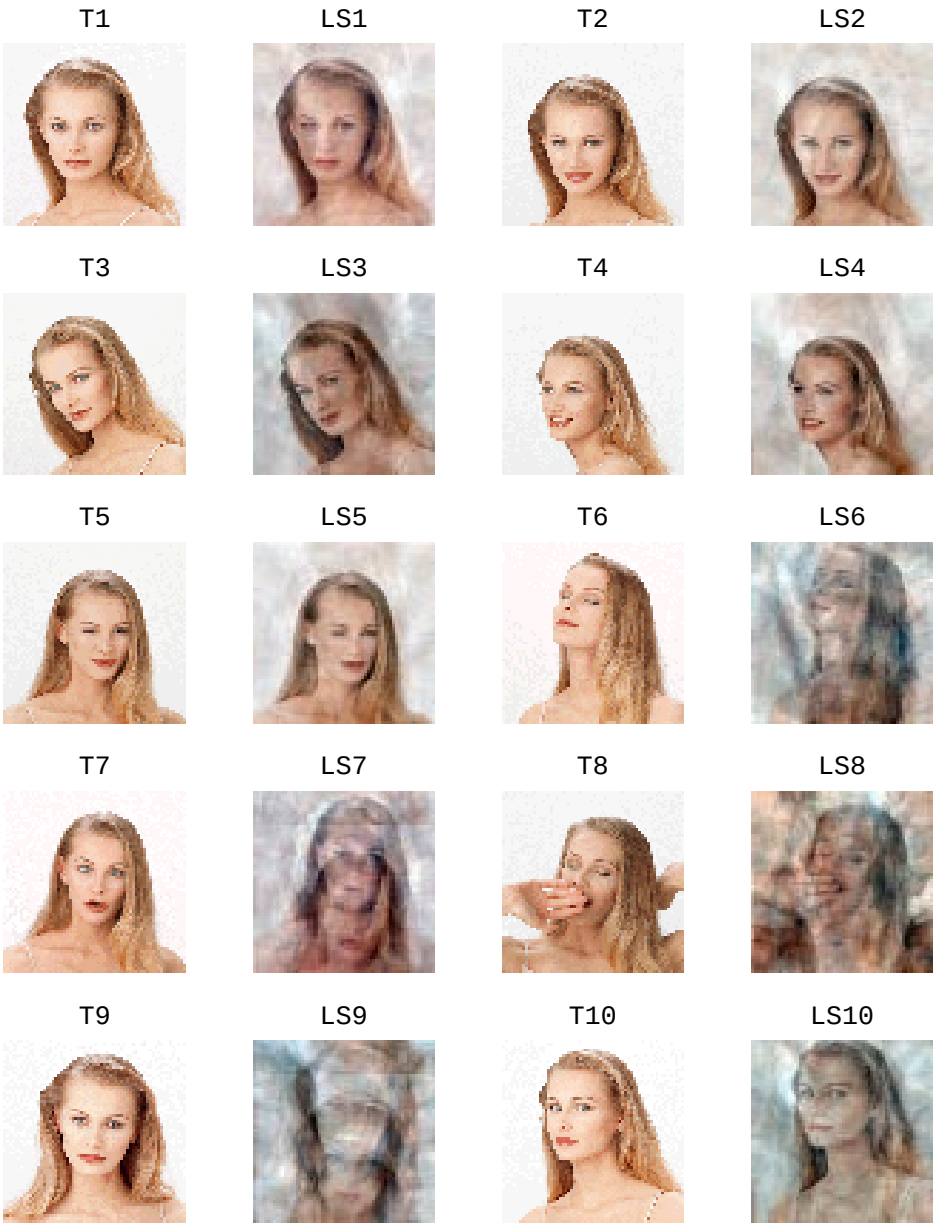
By means of Eq. (8), we synthesize the image satisfying the image system of equations in a least square sense.

```
comLS=Table[Sum[solution[[i,j]]*dbColor[[j]],
    {j,dimDB[[1]]}],{i,dimT[[1]]}];

comLSN=Table[imageNormalize[comLS[[i,j]]],{i,dimT[[1]]},{j,dimT[[2]]}];

comLSG=Table[Show[convertRGB[comLSN[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["LS`1`",i]],{i,dimT[[1]]}];
```

Figure 14 shows the synthesized images along with the input test images. In accordance with the solution vectors shown in Fig. 12, the images “T1-T5” were well synthesized also the others were improved comparing with that of the real domain.

```
Show[GraphicsArray[
    Table[{testG[[i]],comLSG[[i]],testG[[i+1]],comLSG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.14. Test(T) and sythesized(LS) images",
    ImageSize->{4*100,5*100}];
Remove["solution","solutionG","identified","comLS",
    "comLSN","comLSG"];
memoryUsed
```

Fig.14. Test(T) and sythesized(LS) images




```
20638K Bytes used
```

■ 6.6.3 Vector GSPM

Vector GSPM solution

To clarify the differences between the solution methods of the ill posed system, we employ the vector GSPM method, which has a similar result that of the least squares in the real domain.

```
solution=Table[vectorGSPM[systemMat,testV[[i]],500],{i,dimT[[1]]}];//Timing
```

```
{117.16 Second, Null}
```

The pattern matching figures of the iterative solutions are as follows. These figures suggest that all of the solutions have been obtained over 99% pattern matching accuracy.

```
Table[Take[solution[[i]], -1], {i, dimT[[1]]}] // Flatten
```

```
{0.999335, 0.999449, 0.999252, 0.99946, 0.999164, 0.999426, 0.999229,
 0.999288, 0.999294, 0.999106}
```

Figure 15 shows the solution vectors by the vector GSPM method. Similar to these of least squares, the test images “T1-T5” were successfully identified but the other images were not yet represented by the linear combination of the database images in Fig. 2.

```
sol = Table[Take[solution[[i]], dimDB[[1]]], {i, dimT[[1]]}];

solutionG=Table[ListPlot[sol[[i]],
    PlotRange->All,PlotJoined->True,
    PlotLabel->StringForm["SPM`1`",i],
    DisplayFunction->Identity],
    {i,dimT[[1]]}];
```

```
Show[GraphicsArray[
    Table[{solutionG[[i]],solutionG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.15. Vector GSPM solutions"];
```

Fig.15. Vector GSPM solutions

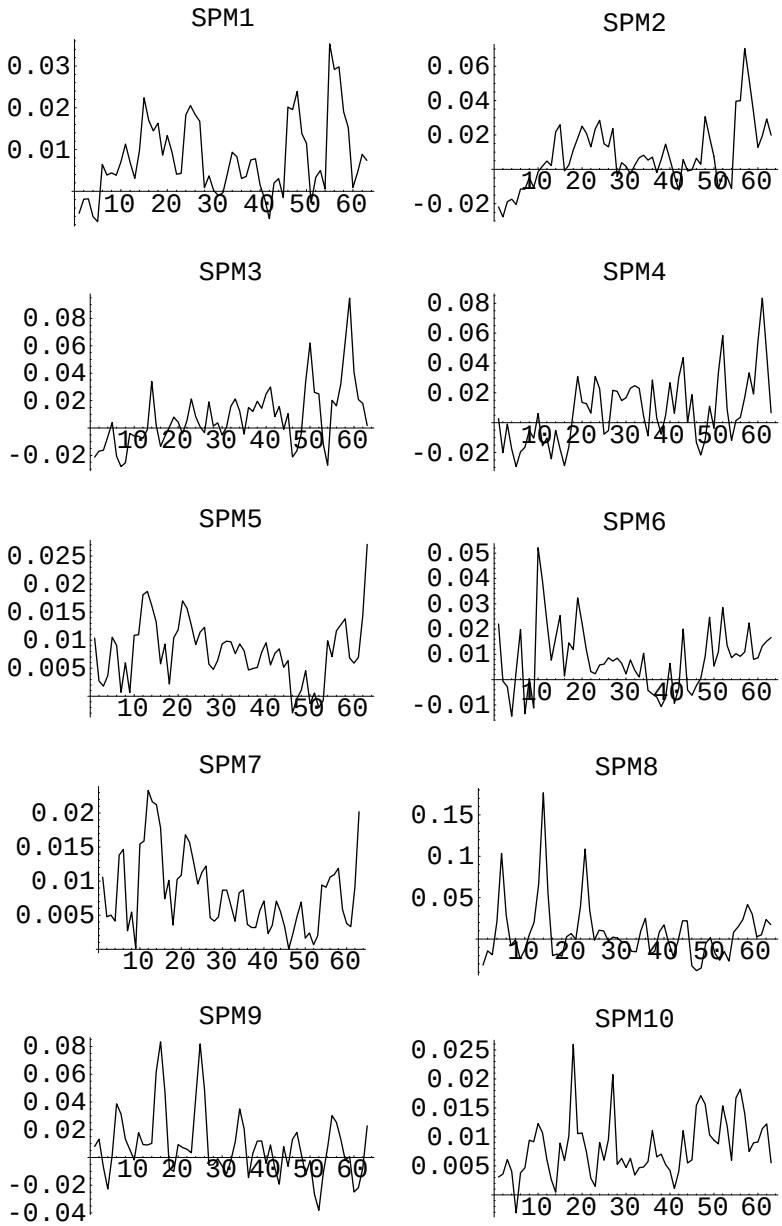


Image identification

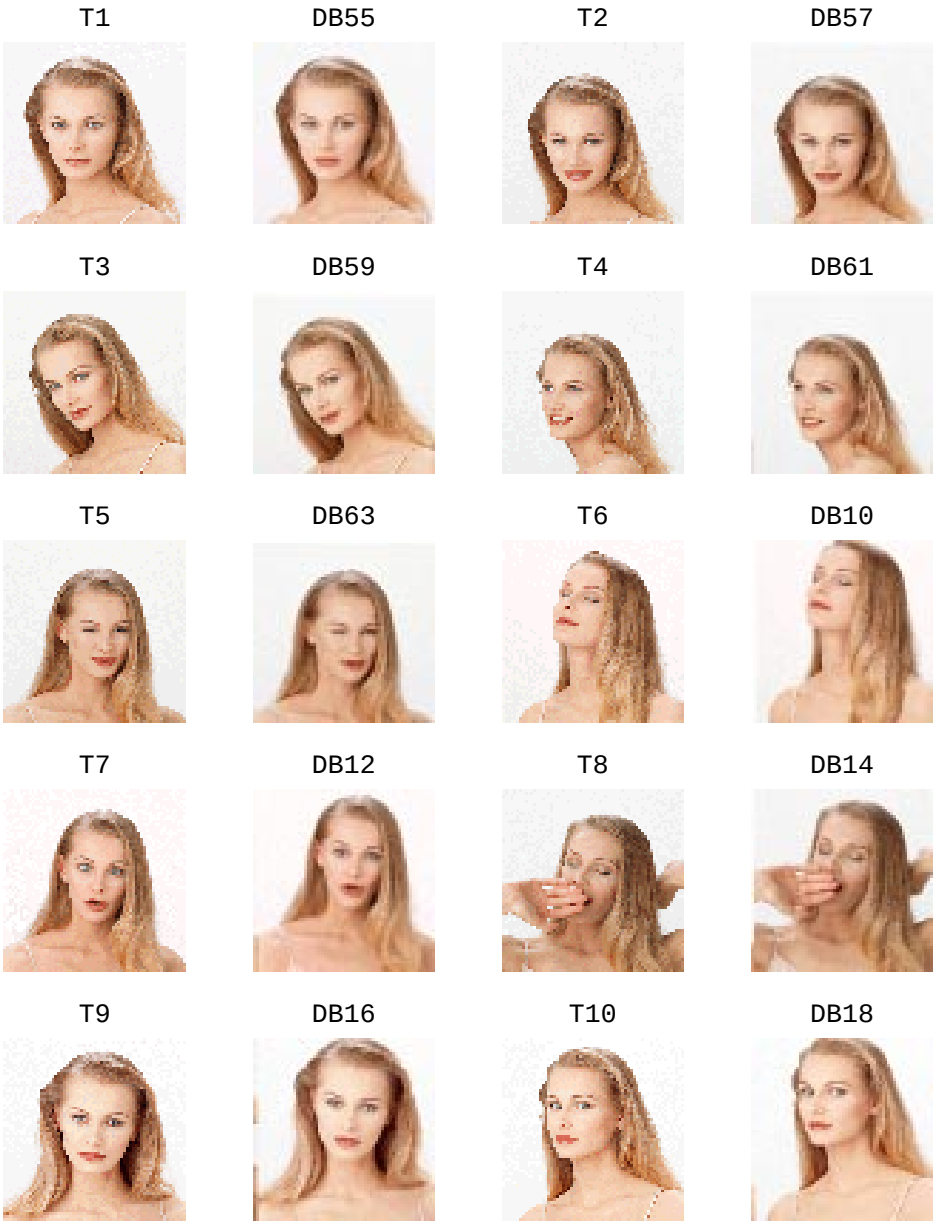
Taking the maximum elements in the solution vectors in Fig. 16 gives the identified images by the iterative means. Figure 17 shows the identified images. All of the test images have been perfectly identified.

```
identified=Table[Position[sol[[i]],Max[sol[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 10, 12, 14, 16, 18}
```

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.16. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
memoryUsed
```

Fig.16. Test(T) and identified(DB) images




```
20831K Bytes used
```

Image synthesize

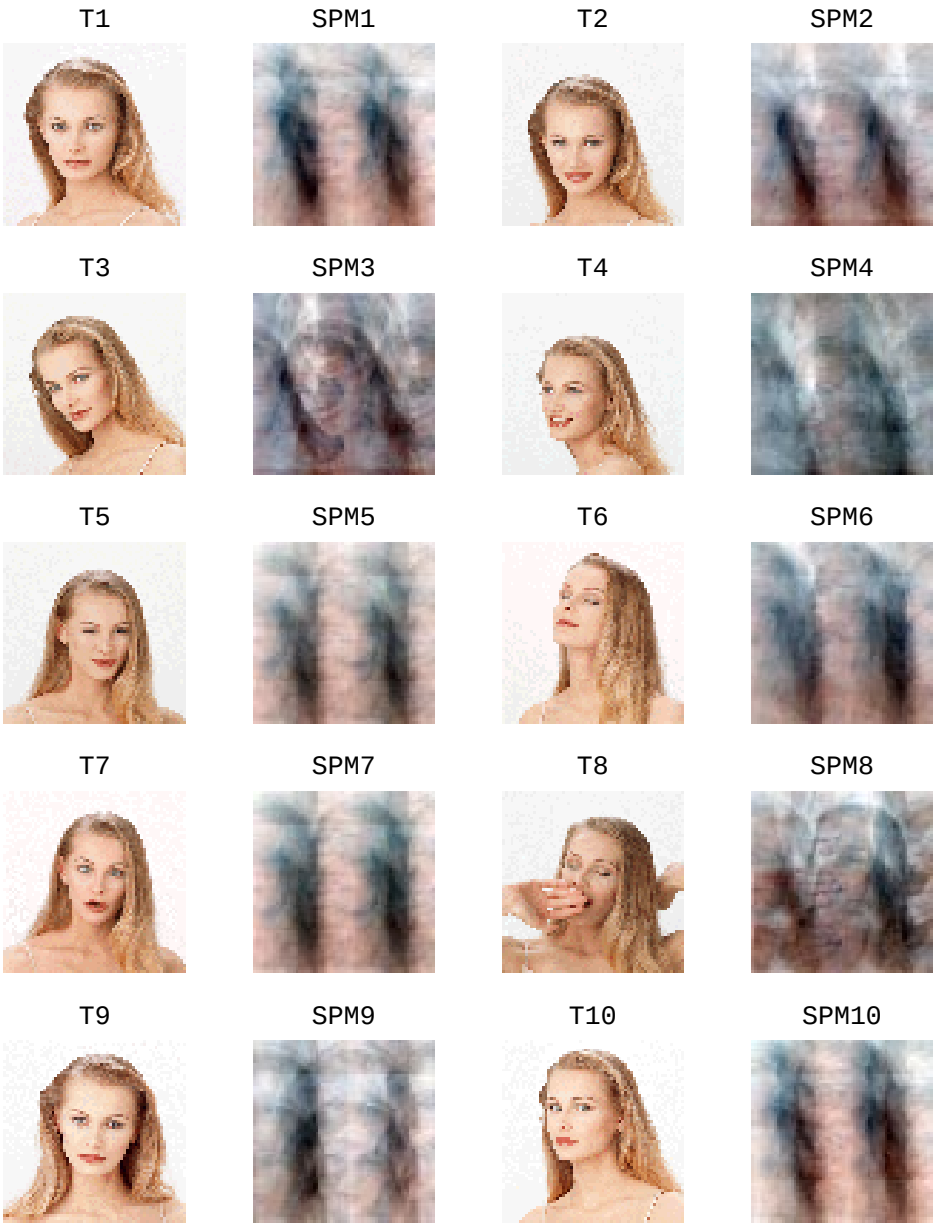
Similar to those of the least squares, it is possible to synthesize the images, which suggest the solvability of the ill posed system of equations. When we can synthesize a clear image, it means that the system could be solved, vice versa.

```
comLS=Table[Sum[sol[[i,j]]*dbColor[[j]],
    {j,dimDB[[1]]}],{i,dimT[[1]]}];

comLSN=Table[imageNormalize[comLS[[i,j]]],{i,dimT[[1]]},{j,dimT[[2]]}];

comLSG=Table[Show[convertRGB[comLSN[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["SPM`1`",i]],{i,dimT[[1]]}];
```

Figure 17 shows the synthesized images by means of the iterative solutions. As expected from the solution vectors in Fig. 15, all of the test images were not clearly synthesized.

```
Show[GraphicsArray[
    Table[{testG[[i]],comLSG[[i]],testG[[i+1]],comLSG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.17. Test(T) and sythesized(GSPM) images",
    ImageSize->{4*100,5*100}];
Remove["systemMat","solution","solutionG","identified",
    "comLS","comLSN","comLSG"];
memoryUsed
```

Fig.17. Test(T) and sythesized(GSPM) images



19848K Bytes used

6.7 Image identification in wavelet spectrum domain

■ 6.7.1 Correlation analysis

In the previous section, we carried out the image identifications in the real and spatial frequency domains. As a result, it ha been clarified that the usefulness of the methodologies depends greatly on the domain. Namely, in the real domain, the correlation analysis was superior methodology than the inverse approaches. But in the frequency domain, the inverse approaches are far superior to the correlation analysis. The difference between the real and frequency domains is that one is a practical real domain and the other is a completely abstract domain. In addition to these domains, we have a neutral domain between the real and frequency domain. This is a wavelet spectrum domain, which holds both of the real and frequency domain information. Wavelet transform is one of the linear transforms and is one of the data sorting methodologies. Depending on the wavelet base functions, wavelet spectrum holds the nature of Fourier spectrum and also includes the real domain information.

Thus, the image identification in the wavelet spectrum domain is significant research theme.

Wavelet transform

Employing the Daubechies 8th order base functions, we compute the wavelet transform matrices. After that, we compute the wavelet spectra of the test as well as database images.

```
wMat={waveletMatrix[dimDB[[3]],daub8],
    waveletMatrix[dimDB[[4]],daub8]};

baseMat=Table[Flatten[Table[TakeMatrix[
    waveletND[dataBase[[i,j]],wMat,2],{1,1},{dimT[[3]],dimT[[4]]}],
    {j,dimDB[[2]]}]],{i,dimDB[[1]]}];

wMat={waveletMatrix[dimT[[3]],daub8],
    waveletMatrix[dimT[[3]],daub8]};

testV=Table[Flatten[Table[waveletND[test[[i,j]],wMat,2],
    {j,dimDB[[2]]}]],{i,dimT[[1]]}];
```

Correlation coefficients

Compute the correlation coefficients between the test and database wavelet spectra.

```
corCoe=Table[corRelation[baseMat[[i]],testV[[j]]],
    {j,dimT[[1]]},{i,dimDB[[1]]}];
```

Taking the highest correlation coefficient to each of the test images gives the identified images by the correlation analysis.

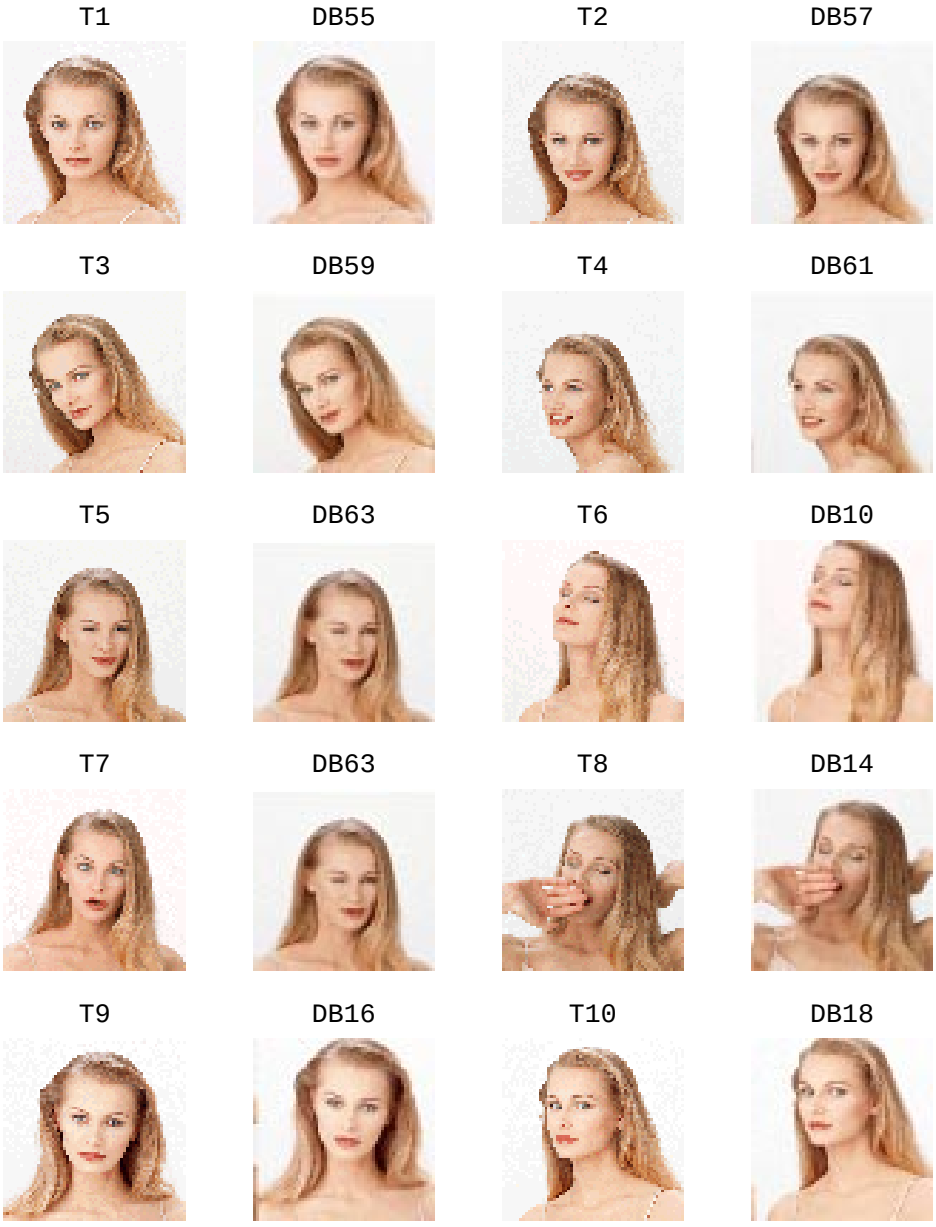
```
identified=Table[Position[corCoe[[i]],Max[corCoe[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 10, 63, 14, 16, 18}
```

Figure 18 shows the identified images along with test ones. In the wavelet spectrum domain, the correlation analysis gives the same result in the real domain, i.e., only one test image has not been identified exactly.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.18. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
Remove["corCoe","identified"];memoryUsed
```

Fig.18. Test(T) and identified(DB) images




```
30901K Bytes used
```

■ 6.7.2 Least squares

System matrix

Transposing the “baseMat” used for the correlation analysis yields a system matrix.

```
systemMat=Transpose[baseMat];
```

Least squares solution

Compute the least squares solutions.

```
solution=Table[leastSQ[systemMat,testV[[i]]],
    {i,dimT[[1]]}];//Timing
```

```
{152.85 Second, Null}
```

Compute the image data of the least squares solutions.

```
solutionG=Table[ListPlot[solution[[i]],
    PlotRange->All,PlotJoined->True,
    PlotLabel->StringForm["LS`1`",i],DisplayFunction->Identity],
    {i,dimT[[1]]}];
```

Figure 19 shows the least squares solution vectors. Similar to the results of previous least squares, the test images “T1”-“T5” were exactly identified, but remaining test images “T6”-”T10” were doubtful results.

```
Show[GraphicsArray[
    Table[{solutionG[[i]],solutionG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.19. Least squares solutions"];
```

Fig.19. Least squares solutions

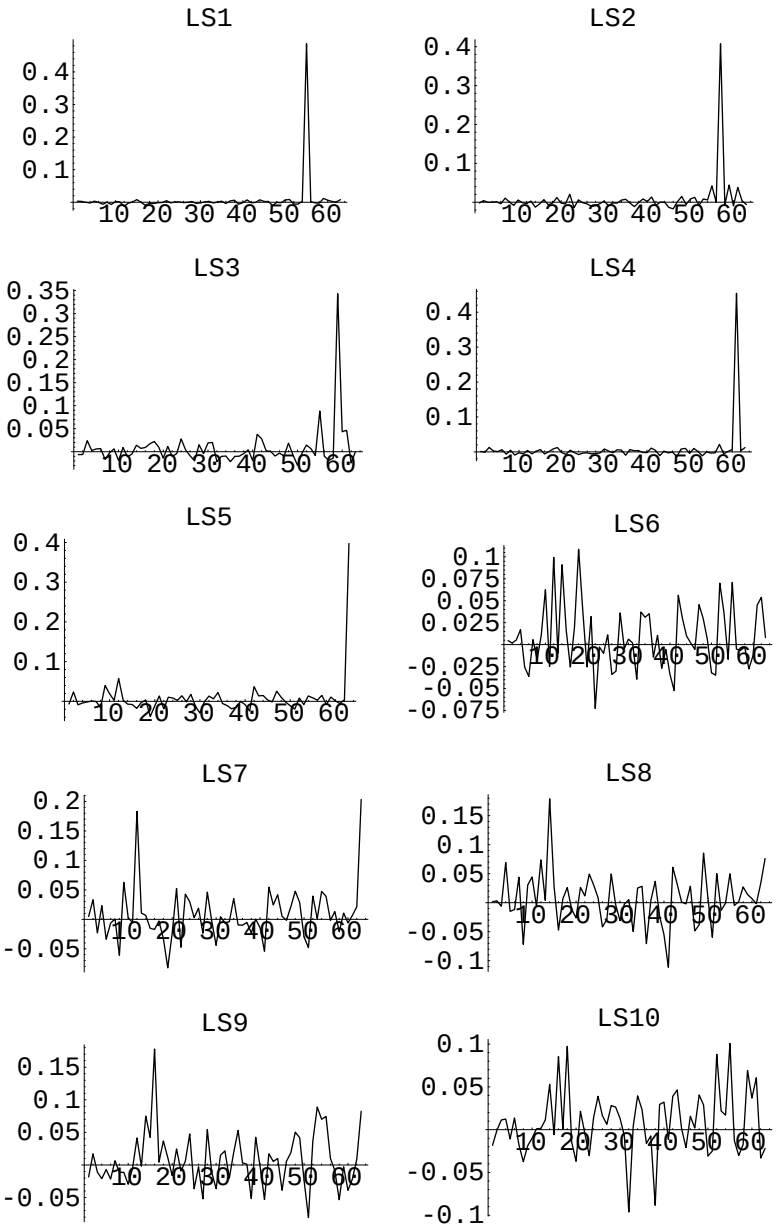


Image identification

Taking the maximum elements in the solution vectors, we can obtain the identified images by least squares.

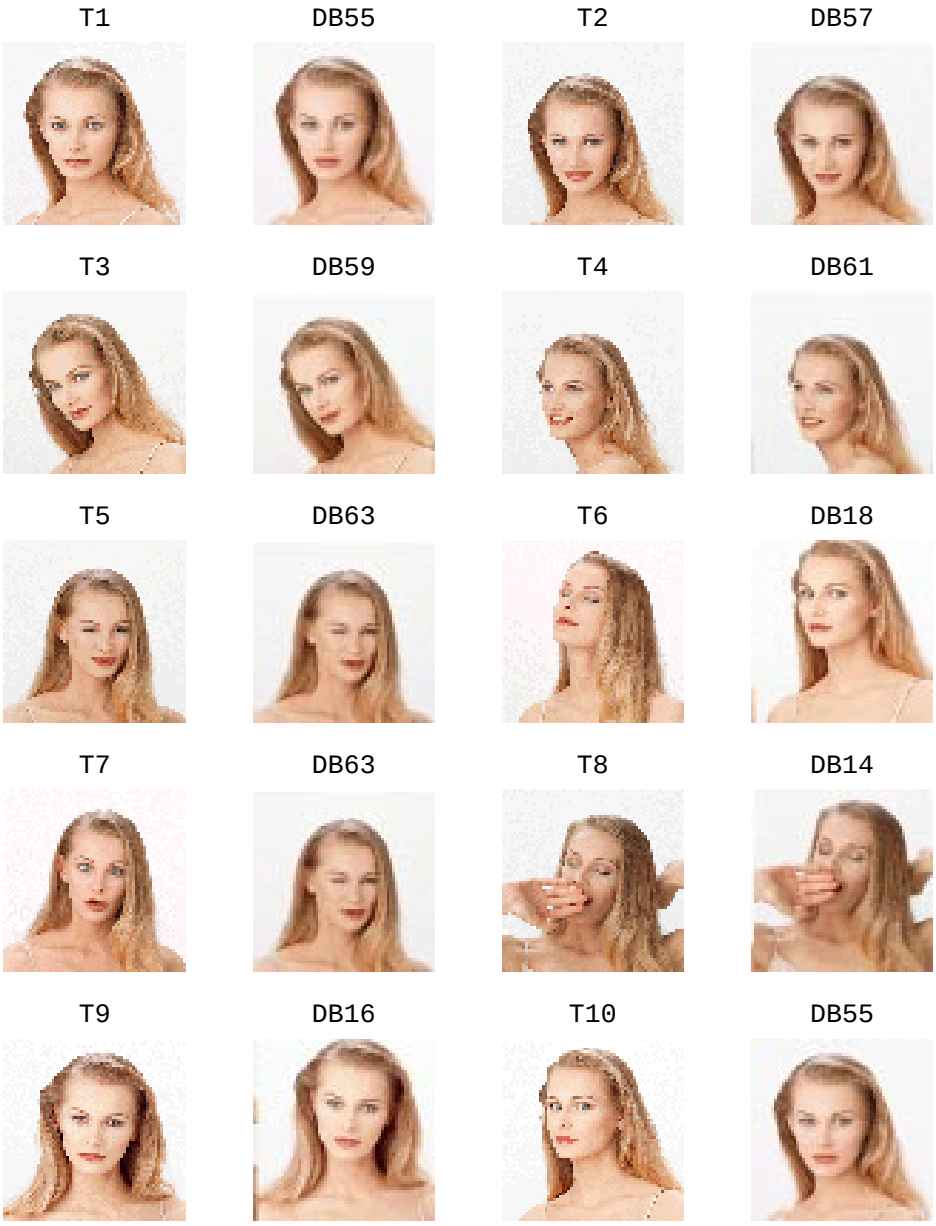
```
identified=Table[Position[solution[[i]],Max[solution[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 18, 63, 14, 16, 55}
```

Figure 20 shows the identified images by means of the least squares. The seven images were successfully identified but remaining four images were not identified by the least squares. This result is just same as those of the real domain. Thereby, the nature of wavelet spectrum domain is similar to those of real domain.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.20. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
memoryUsed
```

Fig.20. Test(T) and identified(DB) images




```
34472K Bytes used
```

Image synthesize

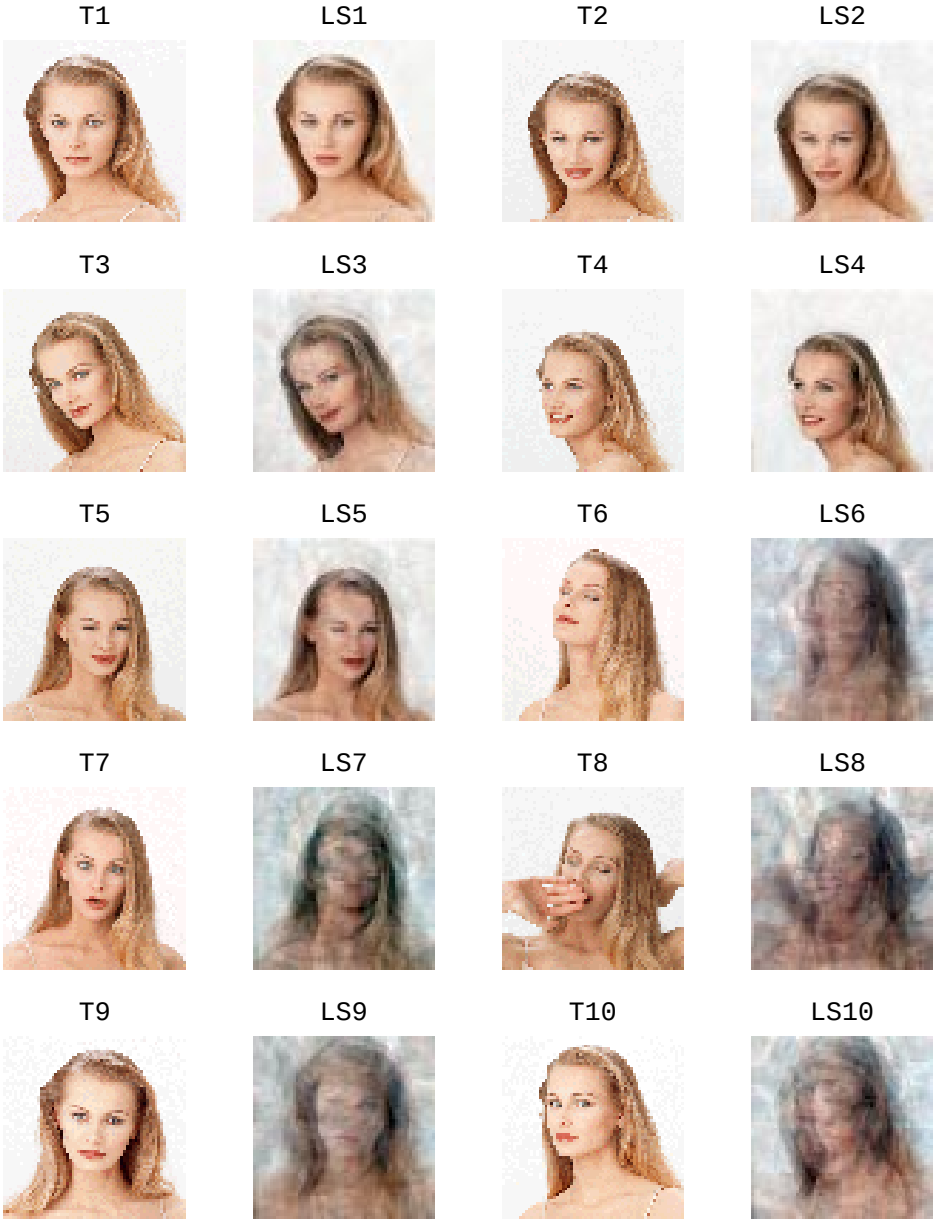
By means of Eq. (8), we synthesize the image satisfying the image system of equations in a least square sense.

```
comLS=Table[Sum[solution[[i,j]]*dbColor[[j]],
    {j,dimDB[[1]]}],{i,dimT[[1]]}];

comLSN=Table[imageNormalize[comLS[[i,j]]],{i,dimT[[1]]},{j,dimT[[2]]}];

comLSG=Table[Show[convertRGB[comLSN[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["LS`1`",i]],{i,dimT[[1]]}];
```

Figure 21 shows the synthesized images along with the input test images. In accordance with the solution vectors shown in Figs. 19 and 4, the images “T1-T5” were well synthesized but the others were still poor results. This result is also similar to those of the real domain.

```
Show[GraphicsArray[
    Table[{testG[[i]],comLSG[[i]],testG[[i+1]],comLSG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.21. Test(T) and sythesized(LS) images",
    ImageSize->{4*100,5*100}];
Remove["solution","solutionG","identified","comLS","comLSN","comLSG"];
memoryUsed
```

Fig.21. Test(T) and sythesized(LS) images




```
34468K Bytes used
```

■ 6.7.3 Vector GSPM

Vector GSPM solution

Compute the solution vectors by the vector GSPM solutions.

```
solution=Table[vectorGSPM[systemMat,testV[[i]],500],{i,dimT[[1]]}];//Timing
```

```
{645.71 Second, Null}
```

The pattern matching figures of the iterative solutions are as follows.

```
Table[Take[solution[[i]], -1], {i, dimT[[1]]}] // Flatten
```

```
{0.996572, 0.993255, 0.992069, 0.995135, 0.993046, 0.992186,
 0.98957, 0.963951, 0.978248, 0.989402}
```

Compute the image data of the vector GSP solutions.

```
sol = Table[Take[solution[[i]], dimDB[[1]]], {i, dimT[[1]]}];
solutionG = Table[ListPlot[sol[[i]],
 PlotRange -> All, PlotJoined -> True,
 PlotLabel -> StringForm["SPM`1`", i],
 DisplayFunction -> Identity],
 {i, dimT[[1]]}];
```

Figure 22 shows the solution vectors by the vector GSPM method. Similar to these of least squares, the test images “T1-T5” were successfully identified but the other images were not yet represented by the linear combination of the database images in Fig. 2.

```
Show[GraphicsArray[
    Table[{solutionG[[i]],solutionG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.22. Vector GSPM solutions"];
```

Fig.22. Vector GSPM solutions

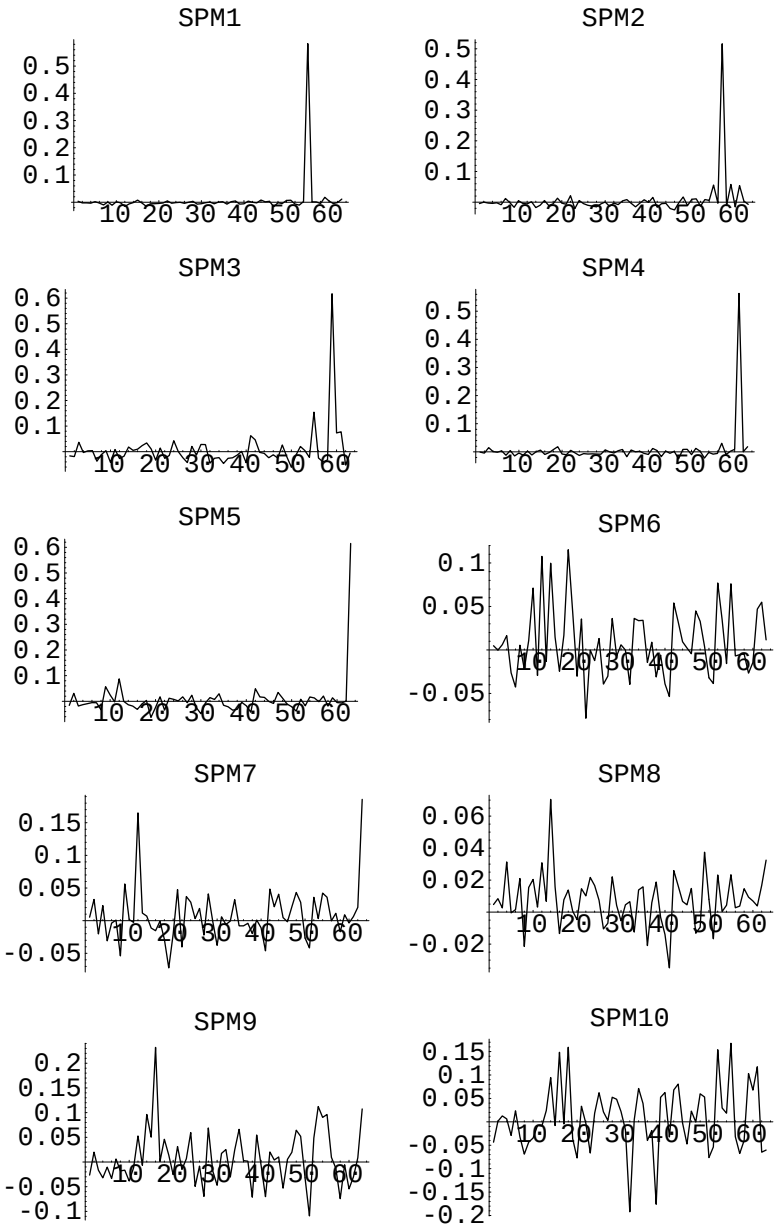


Image identification

Taking the maximum elements in the solution vectors in Fig. 22 gives the identified images by the iterative means. Figure 23 shows the identified images. The result is just same as those of the real domain. Three test images were not exactly identified and the vector GSPM method successfully identified remaining seven images.

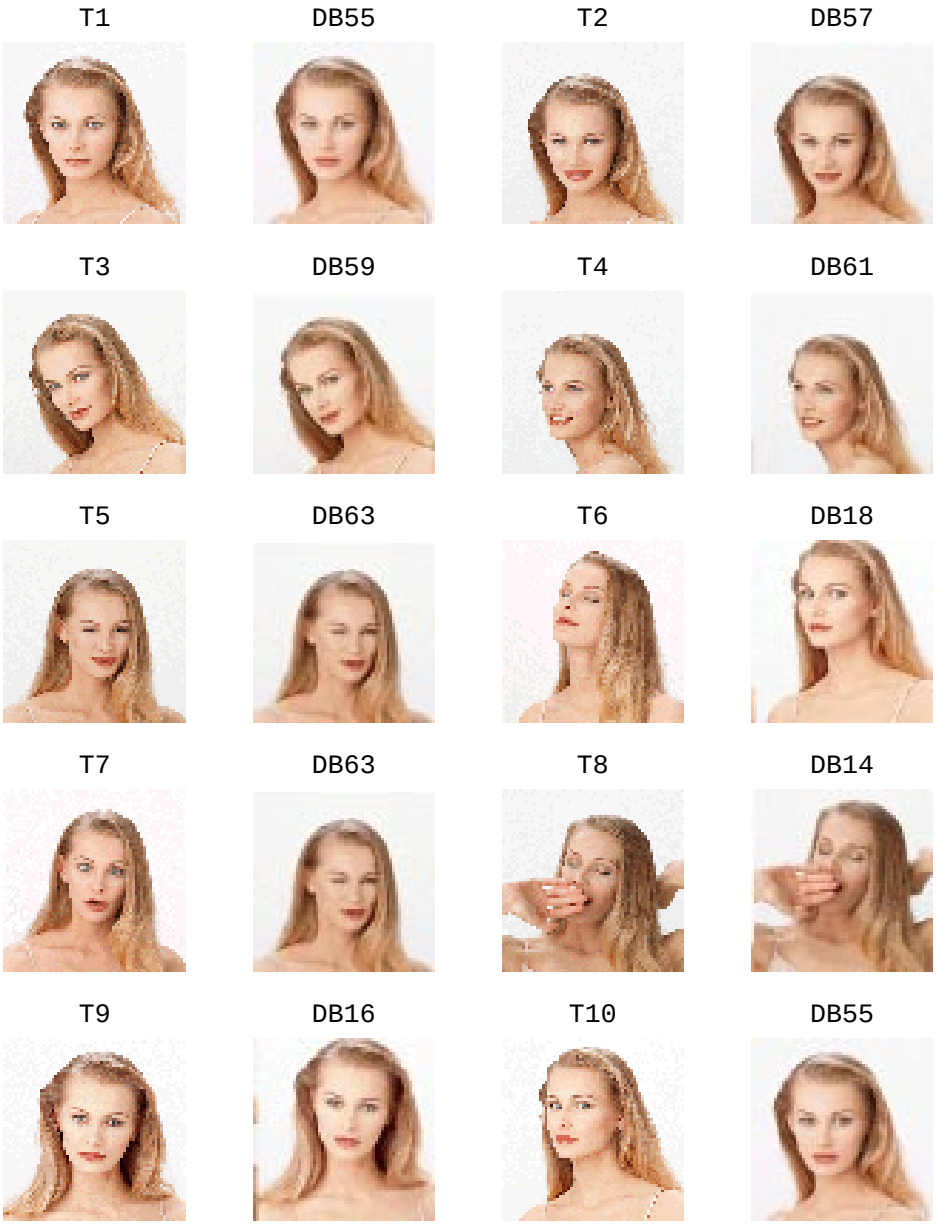
Here we describe the reason why the nature of wavelet spectrum and real domains are the same. The reason of this is very simple, we have not take the particular wavelet spectrum including the mother wavelet into account but all of the wavelet spectrum have been used to the image identification. In order to emphasis the nature of wavelet spectrum domain, we have to take the particular wavelet spectrum into account. However, we did not carry out this because we did not establish a firm methodology which wavelet spectrum should be taken in to account the image identifications. Further, the nature of wavelet spectrum is greatly depending on the employed base functions. But, we did not have enough knowledge which base function should be employed. When we employ the wavelet base function and take the wavelet spectrum into account for the case by case, we may be possible to get the good result for the inverse approaches. But this has no general meaning. Thus, we talked all of the wavelet spectra into account for the image identification. As a result, it has been clarified that a simple sorting of the image data does not change the results.

```
identified=Table[Position[sol[[i]],Max[sol[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 63, 18, 63, 14, 16, 55}
```

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.23. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
memoryUsed
```

Fig.23. Test(T) and identified(DB) images




```
34664K Bytes used
```

Image synthesize

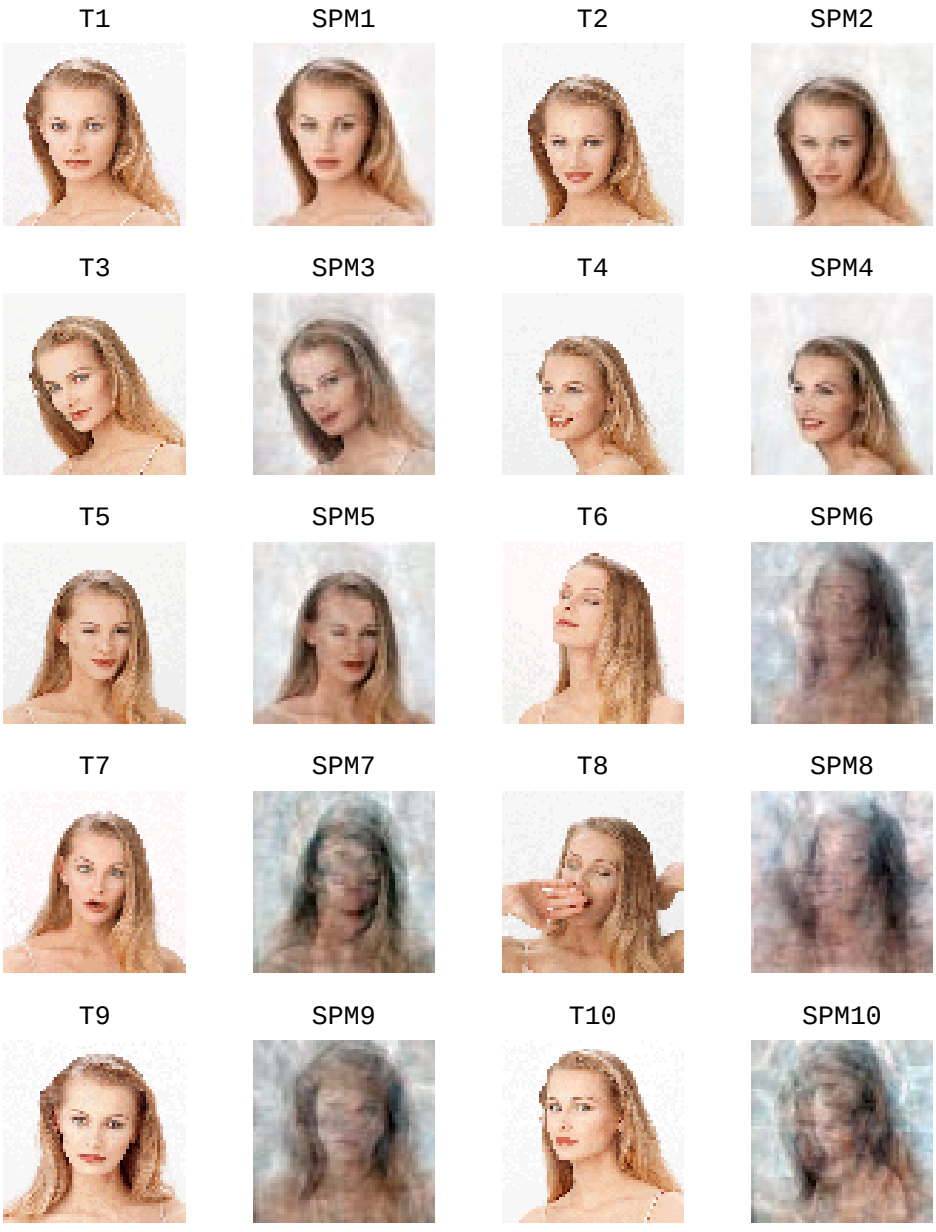
Similar to those of the least squares, it is possible to synthesize the images, which suggest the solvability of the ill posed system of equations.

```
comLS=Table[Sum[sol[[i,j]]*dbColor[[j]],
    {j,dimDB[[1]]}],{i,dimT[[1]]}];

comLSN=Table[imageNormalize[comLS[[i,j]]],{i,dimT[[1]]},{j,dimT[[2]]}];

comLSG=Table[Show[convertRGB[comLSN[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["SPM`1`",i]],{i,dimT[[1]]}];
```

Figure 24 shows the synthesized images by means of the iterative solutions. As expected from the solution vectors in Figs. 22 and 8, only the test images “T1-T5” were clearly synthesized similar to the test images in Fig. 1.

```
Show[GraphicsArray[
    Table[{testG[[i]],comLSG[[i]],testG[[i+1]],comLSG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.24. Test(T) and sythesize(GSPM) images",
    ImageSize->{4*100,5*100}];
Remove["wMat","systemMat","solution","solutionG","identified",
    "comLS","comLSN","comLSG"];
memoryUsed
```

Fig.24. Test(T) and sythesize(GSPM) images



31035K Bytes used

6.8 Image identification in eigen pattern domain

In chapter 5, we have defined the eigen pattern of the image. The eigen pattern can be derived by means of a nonlinear transformation. Any linear transformations utilize the square transform matrices such as the Fourier and wavelet transforms, and are capable of recovering the original data exactly. But our eigen pattern is derived by means of the rectangular transform matrices, so that an inverse transform of the eigen pattern approximately recovers the original data. As shown in chapter 5, even though the nonlinear transform, the eigen pattern represents a distinct characteristic of the target image not depending on the image resolution and position.

Thus, in this section, we carry out the image identifications in the eigen pattern domain.

■ 6.8.1 Image eigen pattern

The eigen pattern can be derived by means of the rectangular transform matrices. This makes it possible to derive the same resolution eigen pattern from the different resolution images.

Consideration of 8-bits resolution in each of the red, green and blue color components leads to set the 256 resolution of the color image. Further, the test images in Fig. 2 were windowed in order to reduce the effects of background of image. Thereby, we apply a window operation to the 256 by 256 resolution database images.

```
resolution=256;
win128=window[128,128,64-8];
dbW=Table[win128*dataBase[[i,j]],{i,dimDB[[1]]},{j,dimDB[[2]]}];
```

■ 6.8.2 Correlation analysys

Data arrangement

In order to implement the correlation analysis in the eigen pattern domain, we compute the eigen patterns of the windowed test and 256 by 256 resolution database images.

```
baseMat=Table[eigenPattern[Flatten[dbW[[i]]],resolution],
    {i,dimDB[[1]]}];

win64=window[64,64,32-4];
testW=Table[win64*test[[i,j]],{i,dimT[[1]]},{j,dimT[[2]]}];
testV=Table[eigenPattern[Flatten[testW[[i]]],resolution],
    {i,dimT[[1]]}];
```

Correlation coefficients

We compute the correlation coefficients between the eigen patterns of test and database images.

```
corCoe=Table[corRelation[baseMat[[i]],testV[[j]]],
    {j,dimT[[1]]},{i,dimDB[[1]]}];
```

Taking the maximum correlation coefficient to each of the test images, we can obtain the identified database image corresponding to the tested one.

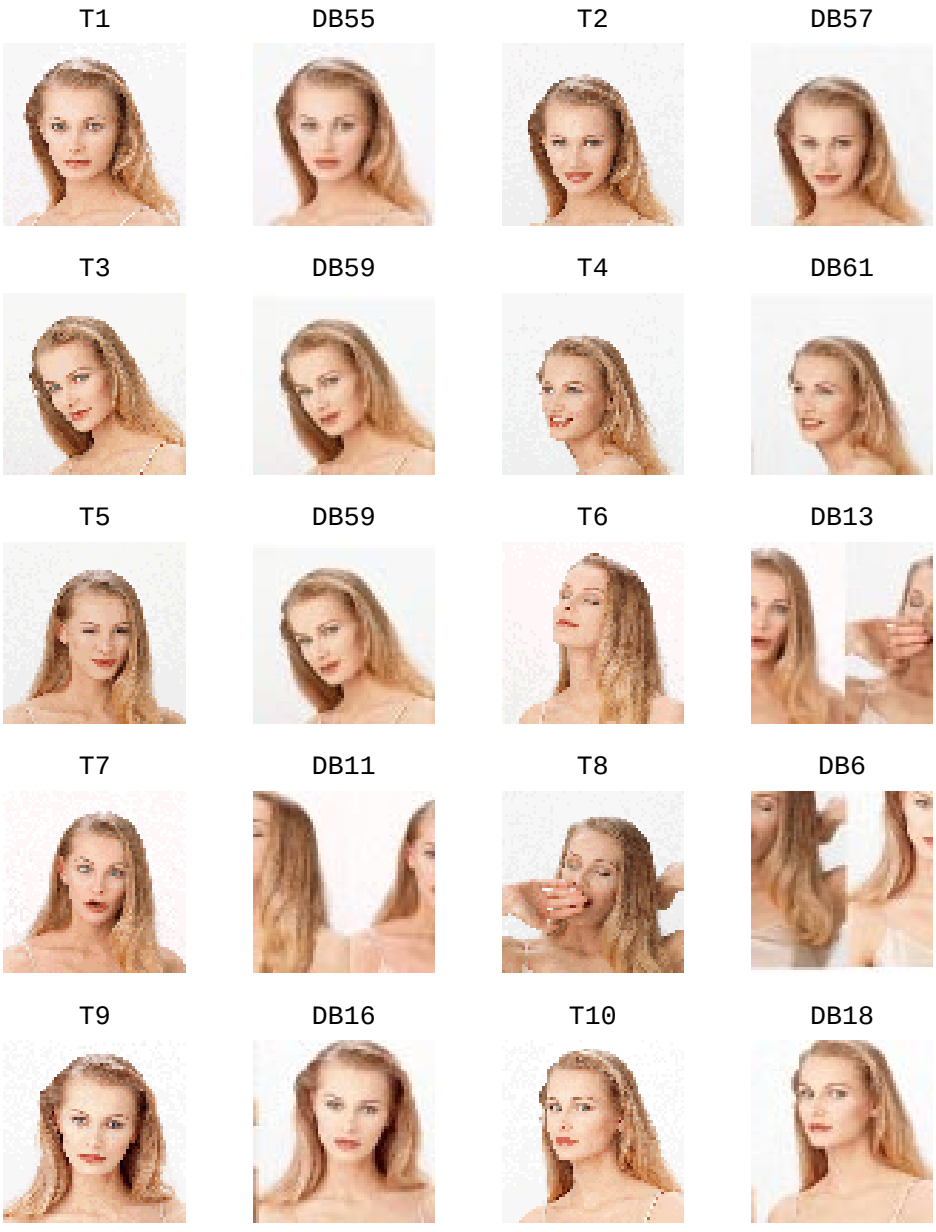
```
identified=Table[Position[corCoe[[i]],Max[corCoe[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 59, 13, 11, 6, 16, 18}
```

Figure 25 shows the identified images together with tested ones. Only the six test images were correctly identified. This result suggests that the correlation analysis is only effective methodology in the real domain. In the other words, the correlation analysis is useful methodology to identify the images taking into account the resolution and position of the test images.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.25. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
Remove["corCoe","identified","win128","win64","dbW"];
memoryUsed
```

Fig.25. Test(T) and identified(DB) images




```
17453K Bytes used
```

■ 6.8.2 Least squares

System matrix

Transpose of the “baseMat” used for correlation analysis yields a system matrix.

```
systemMat=Transpose[baseMat];
```

Inverse check

In order to get the least squares solution, a product between the transpose of system and original system matrices should be a positive definite square matrix. However, in the eigen pattern domain, this condition is not held.

```
Inverse[Transpose[systemMat].systemMat];
```

```
Inverse::sing : Matrix «1» is singular.
```

■ 6.8.3 Vector GSPM

Vector GSPM solution

According to the previous inverse matrix check for the least squares, the system of eigen patterns is badly ill posed, so that we set a number of iterations 1000 to the vector GSPM method.

```
solution=Table[vectorGSPM[systemMat,testV[[i]],1000],{i,dimT[[1]]}];//Timin
g
```

```
{32.68 Second, Null}
```

After a few computation times, we classify the solutions into the solution and pattern matching figure parts.

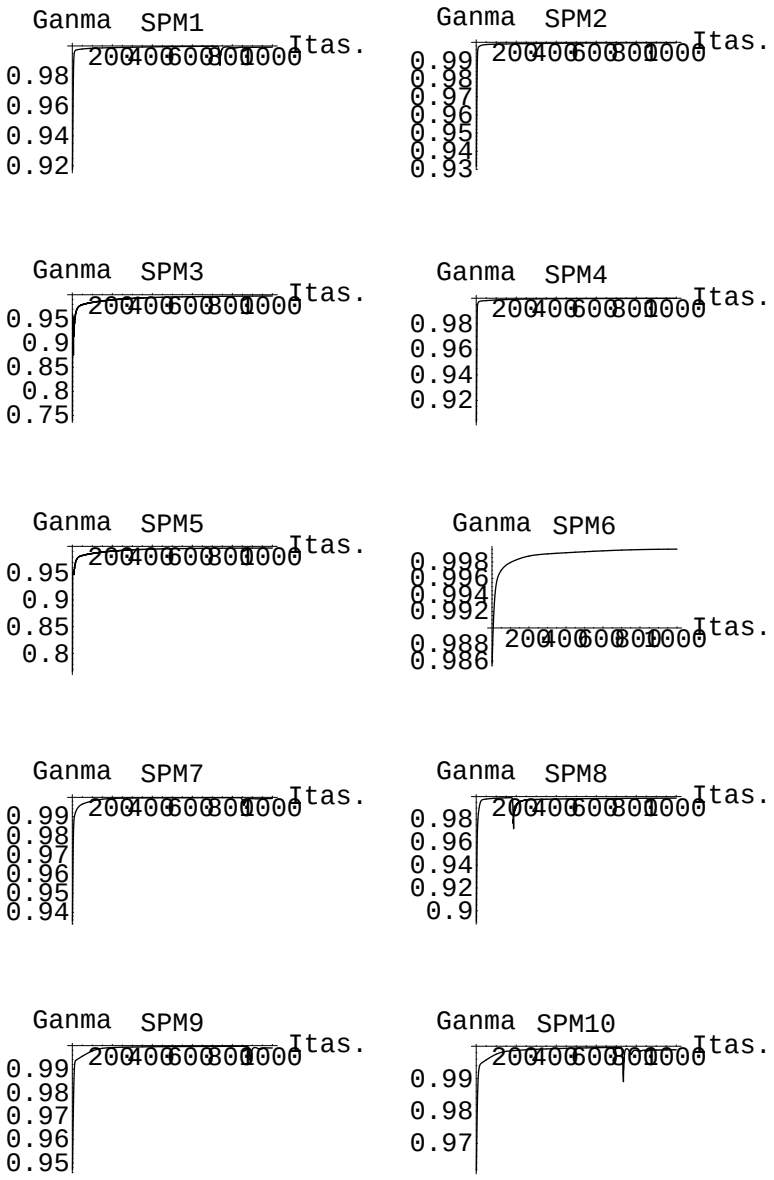
```
sol = Table[Take[solution[[i]], dimDB[[1]]], {i, dimT[[1]]}];
matF = Table[Take[solution[[i]], -1000], {i, dimT[[1]]}];
```

Figure 26 shows the convergence processes to the test images. As described in Eq. (18), any solution vectors by the vector GSPM method have been converged to the fixed vectors. The pattern-matching figure “Ganma” in Fig. 26 corresponds to the value of objective function of Eq. (12), so that the value near to 1 means a goodness of the solutions.

```
convG=Table[ListPlot[matF[[i]],
    PlotRange->All,PlotJoined->True,AxesLabel->{"Itas.","Ganma"},
    PlotLabel->StringForm["SPM`1`",i],DisplayFunction->Identity],
    {i,dimT[[1]]}];
```

```
Show[GraphicsArray[
    Table[{convG[[i]],convG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.26. Convergence processes"];
```

Fig.26. Convergence processes



Compute the image data of the vector GSP solutions.

```
solutionG=Table[ListPlot[sol[[i]],
    PlotRange->All,PlotJoined->True,
    PlotLabel->StringForm["SPM`1`",i],
    DisplayFunction->Identity],
    {i,dimT[[1]]}];
```

Figure 27 shows the solution vectors by the vector GSPM method. The solution vectors are quite different compared with all of the previous solutions. The solutions to the test images "T1-T5" are not the one-peak solutions shown in Figs. 5,9,13,16,20 and 23, but every solution vector is composed of the several peak elements. This means that the eigen pattern system of equations has been established under the same conditions to the tested and database images. Namely, the eigen pattern in each of the images represents the resolution and position independent characters.

```
Show[GraphicsArray[
    Table[{solutionG[[i]],solutionG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],ImageSize->{400,500},
    PlotLabel->"Fig.27. Vector GSPM solutions"];
```

Fig.27. Vector GSPM solutions

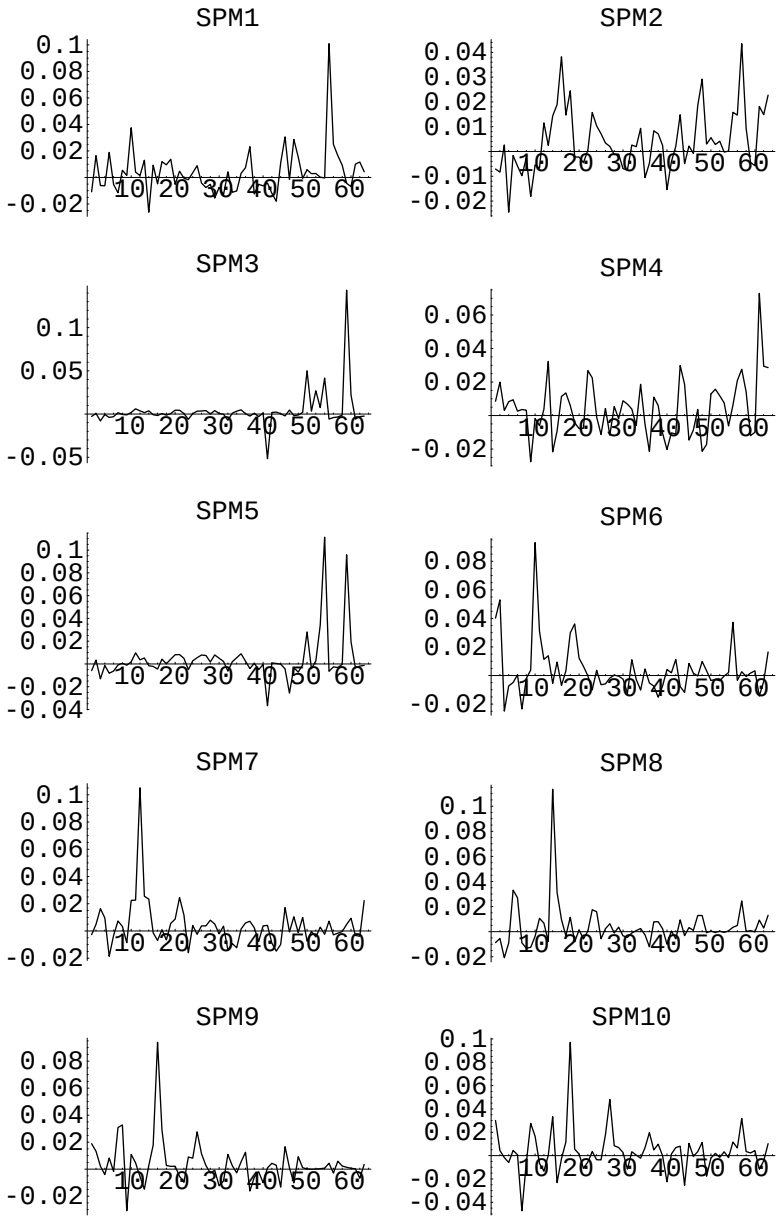


Image identification

Taking the maximum elements in the solution vectors in Fig. 27 gives the identified images by the iterative means. Figure 28 shows the identified images.

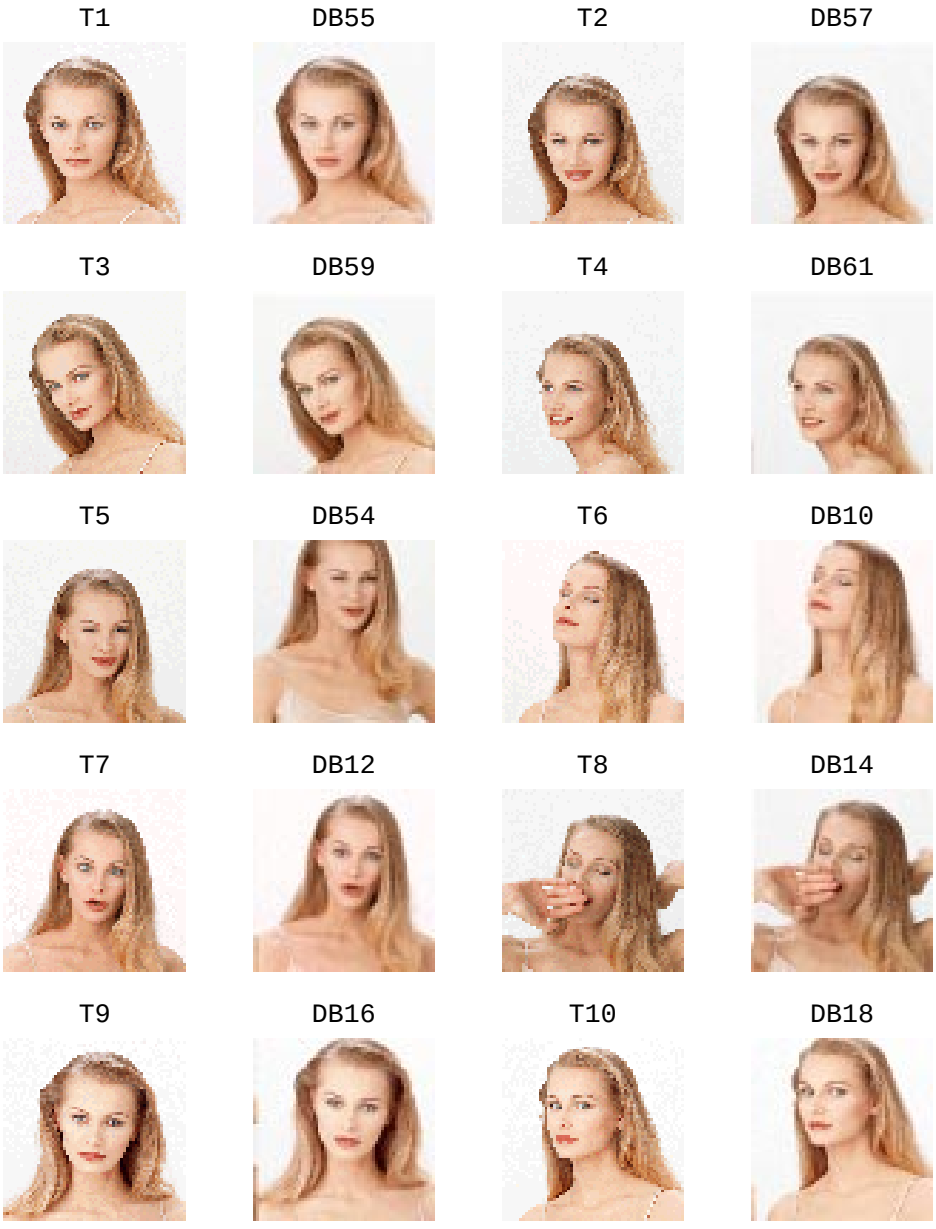
```
identified=Table[Position[sol[[i]],Max[sol[[i]]]],
    {i,dimT[[1]]}]//Flatten
```

```
{55, 57, 59, 61, 54, 10, 12, 14, 16, 18}
```

Surprisingly, all of the test images in Fig.1 were identified from the 63 database images in Fig. 2. Thus, we have confirmed that the concept of eigen pattern makes it possible to remove the effects of image resolution and position, and leads to a sophisticate image identification methodology.

```
Show[GraphicsArray[
    Table[{testG[[i]],dbColorG[[identified[[i]]]],
        testG[[i+1]],dbColorG[[identified[[i+1]]]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.28. Test(T) and identified(DB) images",
    ImageSize->{4*100,5*100}];
memoryUsed
```

Fig.28. Test(T) and identified(DB) images




```
18051K Bytes used
```

Image synthesize

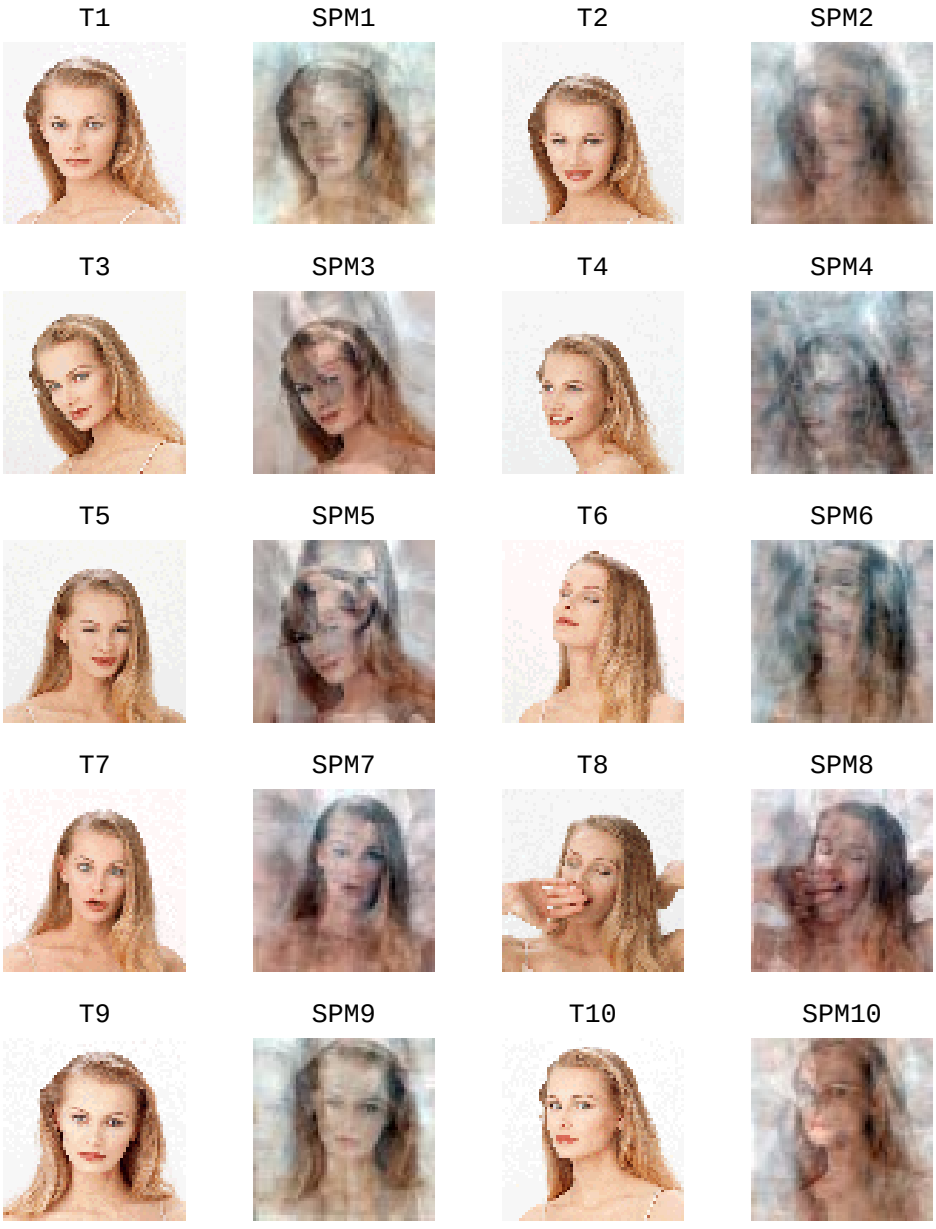
According to Eq. (8), we synthesize the images by combining the solution vectors in Fig. 27 and the database image in Fig.2.

```
comLS=Table[Sum[sol[[i,j]]*dbColor[[j]],
    {j,dimDB[[1]]}],{i,dimT[[1]]}];

comLSN=Table[imageNormalize[comLS[[i,j]]],{i,dimT[[1]]},{j,dimT[[2]]}];

comLSG=Table[Show[convertRGB[comLSN[[i]]],
    AspectRatio->Automatic,DisplayFunction->Identity,
    PlotLabel->StringForm["SPM`1`",i]],{i,dimT[[1]]}];
```

Figure 29 shows the synthesized images. Fairly well images reflecting on their test images were obtained excepting the images “T2” and “T5”. This is as a matter of course fact because all of the test images have been exactly identified.

```
Show[GraphicsArray[
    Table[{testG[[i]],comLSG[[i]],testG[[i+1]],comLSG[[i+1]]},
    {i,1,dimT[[1]]-1,2}]],
    PlotLabel->"Fig.29. Test(T) and sythesize(GSPM) images",
    ImageSize->{4*100,5*100}];
Remove["systemMat","solution","solutionG",
    "identified","comLS","comLSN","comLSG"];memoryUsed
```

Fig.29. Test(T) and sythesize(GSPM) images



21389K Bytes used

6.9 Summary

In this chapter, we have tried to identify the particular images in a large number of database images by means of the three different approaches. The first was a conventional correlation analysis. The other approaches were based on the inverse analysis methodologies. The inverse analysis is essentially reduced into solving for the ill posed linear system of equations. Hence, we have derived the image system of equations. To obtain the solution of the image system of equations, we have employed two methodologies. On is the well known least squares, and the other is the vector GSPM method. The former requires using a matrix inversion. This limits the application of the least squares. The latter is an iterative scheme to solve the ill posed system of equations. Iterative solution strategy can be applied to any ill posed system, but always gives a converged solution. To overcome this difficulty, we have introduced the vector GSPM method, which guarantees the converged solutions.

These identification methodologies have been implemented in the real, Fourier, wavelet and eigen pattern domains. As the results, it has been clarified that the correlation approach is the distinguished methodology in the real domain. This means that the correlation analysis has to take into account the resolution, position and background of a target image. On the other side, the inverse approaches are superior methodologies in the frequency and eigen pattern domains. Particularly, iterative solution strategy is a powerful tool to identify the images.

Summarizing the implemented results is as follows:

Correlation analysis:	90% in real domain,
	100% in Fourier spectrum domain,
	90% in wavelet spectrum domain
	60% in eigen pattern domain.
Least squares:	70% in real domain,
	100% in Fourier spectrum domain,
	70% in wavelet spectrum domain
	Not available in eigen pattern domain.
vector GSPM	70% in real domain,
	100% in Fourier spectrum domain,
	70% in wavelet spectrum domain
	100% in eigen pattern domain.

■ REFERENCES

[1] Stephen Wolfram, *The Mathematica Book*, 3rd ed. (Wolfram Media/Cambridge University Press, 1996).

[2] G.Strang, " Linear Algebra and its Applications ", 1976, Academic Press, Inc.

[3] Y.Midorikawa, J.Ogawa, T.Doi, S.Hayano and Y.Saito, " Inverse analysis for magnetic field source searching in thin film conductor ", IEEE Transaction on Magnetics, Vol.MAG-33, No.5, Sep.,1997, pp.4008-4010.

[4] K.Yoda, Y.Saito and H.Sakamoto, " Dose optimization of proton and heavy ion therapy using generalized sampled pattern matching," Phys.Med.Biol., IOP Publishing,1997,pp.2411-2420.