

**TS. PHẠM HỒNG LIÊN
ĐẶNG NGỌC KHOA - TRẦN THANH PHƯƠNG**

MATLAB VÀ ỨNG DỤNG TRONG VIỄN THÔNG



Tháng 11/2005

LỜI NÓI ĐẦU

MỤC LỤC

LỜI NÓI ĐẦU	i
MỤC LỤC.....	iii
1. MỞ ĐẦU	3
1.1. NHẬP MỘT DÒNG LỆNH.....	3
1.2. CÔNG CỤ GIÚP ĐỠ.....	5
1.3. DỪNG MỘT LỆNH HAY CHƯƠNG TRÌNH.....	6
1.4. ĐƯỜNG DẪN.....	6
1.5. KHÔNG GIAN LÀM VIỆC (WORKSPACE).....	6
1.6. SAVE VÀ LOAD DỮ LIỆU.....	7
2. CƠ BẢN VỀ CÚ PHÁP VÀ BIẾN	9
2.1. MATLAB NHƯ LÀ MỘT CÔNG CỤ TÍNH TOÁN	9
2.2. PHÉP GÁN VÀ BIẾN	10
3. PHÉP TOÁN VỚI VECTOR VÀ MA TRẬN	14
3.1. VECTOR	14
3.1.1. DẤU ‘:’ VÀ PHẦN TRÍCH RA TỪ VECTOR	15
3.1.2. VECTOR CỘT VÀ PHÉP CHUYỂN VỊ.....	16
3.1.3. NHÂN, CHIA VÀ SỐ MŨ CỦA VECTOR.....	17
3.2. MA TRẬN	19
3.2.1. NHỮNG MA TRẬN ĐẶC BIỆT	21
3.2.2. XÂY DỰNG MA TRẬN VÀ TRÍCH RA MỘT MA TRẬN CON TỪ MỘT MA TRẬN LỚN HƠN.....	23
3.2.3. TÍNH TOÁN VỚI MA TRẬN.....	27
4. ĐỒ THỊ 2D VÀ 3D	32
4.1. NHỮNG ĐỒ THỊ ĐƠN GIẢN.....	32
4.2. MỘT SỐ HÀM ĐƯỢC SỬ DỤNG TRONG VẼ ĐỒ THỊ	34
4.3. CÁC THUỘC TÍNH KHÁC CỦA ĐƯỜNG CONG 2D.....	37
4.4. IN ẤN.....	38
4.5. ĐỒ THỊ 3D.....	39

4.6. BỀ MẶT ĐỒ THỊ	40
4.7. HÌNH ĐỘNG	46
5. BIỂU THỨC RỄ NHÁNH	49
5.1. CÁC TOÁN TỬ LOGIC VÀ BIỂU THỨC QUAN HỆ	49
5.2. BIỂU THỨC ĐIỀU KIỆN	53
5.3. VÒNG LẶP	56
6. TẬP LỆNH VÀ HÀM	60
6.1. TẬP LỆNH M-FILE	60
6.2. HÀM M-FILE	61
6.2.1. NHỮNG BIẾN ĐẶC BIỆT TRONG HÀM	63
6.2.2. BIẾN TOÀN CỤC VÀ BIẾN CỤC BỘ	64
6.2.3. CÁCH GỌI HÀM GIÁN TIẾP	65
6.3. TẬP TIN VÀ HÀM	66
7. VĂN BẢN	69
7.1. CHUỖI KÝ TỰ	69
7.2. XUẤT VÀ NHẬP VĂN BẢN	71
8. GIAO DIỆN NGƯỜI SỬ DỤNG (GUI)	77
8.1. CÁCH LÀM VIỆC CỦA MỘT GUI	77
8.2. TẠO VÀ HIỂN THỊ MỘT GUI	79
8.3. THUỘC TÍNH CỦA CÁC ĐỐI TƯỢNG	85
8.4. CÁC THÀNH PHẦN TẠO NÊN GUI	86
8.4.1. Text Fields	87
8.4.2. Edit Boxes	87
8.4.3. Frames	88
8.4.4. Pushbuttons	89
8.4.5. Toggle Buttons	89
8.4.6. Checkboxes và Radio Buttons	90
8.4.7. Popup Menus	91
8.4.8. List Boxes	93
8.4.9. Sliders	95
9. TÍN HIỆU VÀ HỆ THỐNG	91

9.1. BIỂU DIỄN MỘT TÍN HIỆU TRONG MATLAB	91
9.2. TẠO TÍN HIỆU: VECTOR THỜI GIAN	91
9.3. LÀM VIỆC VỚI CÁC FILE DỮ LIỆU	94
9.4. PHÂN TÍCH VÀ THIẾT KẾ CÁC BỘ LỌC	94
9.5. CÁC HÀM KHÁC ĐỂ THỰC HIỆN LỌC	97
9.5.1. THỰC HIỆN BĂNG LỌC ĐA TỐC ĐỘ (MULTIRATE FILTER BANK)	97
9.5.2. KHỬ MÉO PHA CHO BỘ LỌC IIR.....	98
9.5.3. THỰC HIỆN BỘ LỌC TRONG MIỀN TẦN SỐ.....	99
9.6. ĐÁP ỨNG XUNG	100
9.7. ĐÁP ỨNG TẦN SỐ	100
9.7.1. TRONG MIỀN SỐ	100
9.7.2. TRONG MIỀN ANALOG.....	101
9.7.3. ĐÁP ỨNG BIÊN ĐỘ VÀ ĐÁP ỨNG PHA	101
9.7.4. THỜI GIAN TRỄ	102
9.8. GIẢN ĐỒ CỰC – ZERO	103
9.9. CÁC MÔ HÌNH HỆ THỐNG TUYẾN TÍNH	104
9.9.1. CÁC MÔ HÌNH HỆ THỐNG RỜI RẠC THEO THỜI GIAN.....	104
9.9.2. CÁC MÔ HÌNH HỆ THỐNG LIÊN TỤC THEO THỜI GIAN	108
9.9.3. CÁC PHÉP BIẾN ĐỔI HỆ THỐNG TUYẾN TÍNH	108
9.10. BIẾN ĐỔI FOURIER RỜI RẠC	109
10. THIẾT KẾ CÁC BỘ LỌC	117
10.1. CÁC CHỈ TIÊU THIẾT KẾ BỘ LỌC	117
10.2. THIẾT KẾ BỘ LỌC IIR	118
10.2.1. THIẾT KẾ CÁC BỘ LỌC IIR CỔ ĐIỂN DỰA TRÊN CÁC NGUYÊN MẪU ANALOG	119
10.2.2. THIẾT KẾ TRỰC TIẾP CÁC BỘ LỌC IIR TRONG MIỀN SỐ.....	126
10.2.3. THIẾT KẾ BỘ LỌC BUTTERWORTH TỔNG QUÁT	127
10.2.4. PHƯƠNG PHÁP MÔ HÌNH THÔNG SỐ	128
10.3. THIẾT KẾ BỘ LỌC FIR.....	129
10.3.1. CÁC BỘ LỌC CÓ PHA TUYẾN TÍNH	129
10.3.2. PHƯƠNG PHÁP CỬA SỔ (WINDOWING).....	130

10.3.3. THIẾT KẾ BỘ LỌC FIR NHIỀU DẢI TẦN VỚI CÁC DẢI CHUYỂN TIẾP	133
10.3.4. THIẾT KẾ BỘ LỌC FIR VỚI GIẢI THUẬT BÌNH PHƯƠNG CỰC TIỂU CÓ GIỚI HẠN (CLS – CONSTRAINED LEAST SQUARE)	136
10.3.5. THIẾT KẾ BỘ LỌC FIR CÓ ĐÁP ỨNG TẦN SỐ TÙY CHỌN	139
10.4. THỰC HIỆN BỘ LỌC	141
11. CƠ BẢN VỀ XỬ LÝ ẢNH SỐ	147
11.1. BIỂU DIỄN ẢNH VÀ XUẤT NHẬP ẢNH	147
11.1.1. CÁC KIỂU HÌNH ẢNH TRONG MATLAB	147
11.1.2. ĐỌC VÀ GHI CÁC DỮ LIỆU ẢNH	150
11.1.3. CHUYỂN ĐỔI GIỮA CÁC KIỂU DỮ LIỆU	151
11.1.4. CÁC PHÉP TOÁN SỐ HỌC CƠ BẢN ĐỐI VỚI DỮ LIỆU ẢNH	152
11.1.5. CÁC HÀM HIỂN THỊ HÌNH ẢNH TRONG MATLAB	155
11.2. CÁC PHÉP BIẾN ĐỔI HÌNH HỌC	156
11.2.1. PHÉP NỘI SUY ẢNH	156
11.2.2. THAY ĐỔI KÍCH THƯỚC ẢNH	156
11.2.3. PHÉP QUAY ẢNH	157
11.2.4. TRÍCH XUẤT ẢNH	158
11.2.5. THỰC HIỆN PHÉP BIẾN ĐỔI HÌNH HỌC TỔNG QUÁT	158
11.3. CÁC PHÉP BIẾN ĐỔI ẢNH	160
11.3.1. BIẾN ĐỔI FOURIER	160
11.3.2. BIẾN ĐỔI COSINE RỜI RẠC	163
11.3.3. BIẾN ĐỔI RADON	165
11.3.4. PHÉP BIẾN ĐỔI FAN-BEAM	168
12. NÂNG CAO CHẤT LƯỢNG ẢNH	176
12.1. PHƯƠNG PHÁP BIẾN ĐỔI MỨC XÁM	176
12.2. CÂN BẰNG HISTOGRAM	180
12.2.1. TẠO VÀ VẼ BIỂU ĐỒ HISTOGRAM	180
12.2.2. CÂN BẰNG HISTOGRAM	181
12.3. LỌC ẢNH	186
12.3.1. LỌC TUYẾN TÍNH	187
12.3.2. LỌC PHI TUYẾN	191

12.3.3. LỌC THÍCH NGHI	194
13.NÉN ẢNH SỐ	199
13.1. PHƯƠNG PHÁP MÃ HOÁ XỬ LÝ KHỐI BTC (BLOCK TRUNCATING CODING)	199
13.2. NÉN TỔN HAO DỰA VÀO DCT	201
13.3. NÉN ẢNH BẰNG GIẢI THUẬT PHÂN TÍCH TRỊ RIÊNG (SVD).....	205
13.3.1. GIỚI THIỆU PHƯƠNG PHÁP SVD	205
13.3.2. ỨNG DỤNG SVD ĐỂ NÉN ẢNH SỐ	206
14.MÃ HÓA NGUỒN	203
14.1. TẠO MỘT NGUỒN TÍN HIỆU.....	203
14.2. LƯỢNG TỬ HÓA TÍN HIỆU	206
14.3. TỐI ƯU HÓA CÁC THÔNG SỐ CỦA QUÁ TRÌNH LƯỢNG TỬ	208
14.4. ĐIỀU CHẾ MÃ XUNG VI SAI DPCM (DIFFERENTIAL PULSE CODE MODULATION)	209
14.5. TỐI ƯU HOÁ CÁC THÔNG SỐ CỦA QUÁ TRÌNH MÃ HÓA DPCM	210
14.6. NÉN VÀ GIẢN TÍN HIỆU.....	211
14.7. MÃ HÓA HUFFMAN	213
14.8. MÃ HÓA SỐ HỌC (ARITHMETIC CODING).....	215
15.TRUYỀN DẪN BASEBAND VÀ PASSBAND	219
15.1. ĐIỀU CHẾ TƯƠNG TỰ	219
15.2. ĐIỀU CHẾ SỐ.....	221
16.KÊNH TRUYỀN VÀ ĐÁNH GIÁ CHẤT LƯỢNG KÊNH TRUYỀN.....	231
16.1. KÊNH TRUYỀN AWGN (ADDITIVE WHITE GAUSSIAN NOISE).....	231
16.2. KÊNH TRUYỀN FADING	235
16.3. KÊNH TRUYỀN ĐẢO BIT NHỊ PHÂN	239
16.4. ĐÁNH GIÁ CHẤT LƯỢNG THÔNG QUA MÔ PHỎNG (PHƯƠNG PHÁP MONTE CARLO)	240
16.5. TÍNH XÁC SUẤT LỖI TRÊN LÝ THUYẾT.....	243
16.6. MỘT SỐ CÔNG CỤ HỖ TRỢ ĐỂ VẼ ĐỒ THỊ BER	245
16.7. GIẢN ĐỒ MẮT (EYE DIAGRAM)	247
16.8. ĐỒ THỊ PHÂN BỐ (SCATTER PLOT).....	249

16.9. ĐÁNH GIÁ CHẤT LƯỢNG DÙNG KỸ THUẬT SEMIANALYTIC (BÁN PHÂN TÍCH)	250
17.MÃ HÓA KÊNH TRUYỀN	256
17.1. MÃ KHỐI.....	256
17.1.1. BIỂU DIỄN MỘT PHẦN TỬ TRONG TRƯỜNG GALOIS.....	257
17.1.2. MÃ REED-SOLOMON	258
17.1.3. MÃ BCH	262
17.1.4. MÃ KHỐI TUYẾN TÍNH	264
17.2. MÃ CHẬP	269
17.2.1. DẠNG ĐA THỨC CỦA BỘ MÃ HOÁ CHẬP	270
17.2.2. DẠNG CẤU TRÚC TRELIS CỦA BỘ MÃ HÓA CHẬP	271
17.2.3. MÃ HÓA VÀ GIẢI MÃ MÃ CHẬP	273
18.CÁC BỘ CÂN BẰNG	281
18.1. CÁC BỘ CÂN BẰNG THÍCH NGHI	281
18.1.1. BỘ CÂN BẰNG KHOẢNG CÁCH KÝ HIỆU	281
18.1.2. BỘ CÂN BẰNG ĐỊNH KHOẢNG TỶ LỆ	282
18.1.3. BỘ CÂN BẰNG HỒI TIẾP QUYẾT ĐỊNH	283
18.2. CÁC GIẢI THUẬT CÂN BẰNG THÍCH NGHI.....	284
18.2.1. GIẢI THUẬT BÌNH PHƯƠNG TRUNG BÌNH CỤC TIỂU (LMS – LEAST MEAN SQUARE).....	284
18.2.2. GIẢI THUẬT LMS CÓ DẤU (SIGN LMS)	285
18.2.3. GIẢI THUẬT LMS CHUẨN HÓA (NORMALIZED LMS)	285
18.2.4. GIẢI THUẬT LMS CÓ BƯỚC NHẢY THAY ĐỔI (VARIABLE-STEP-SIZE LMS)	285
18.2.5. GIẢI THUẬT BÌNH PHƯƠNG CỤC TIỂU HỒI QUY (RLS – RECURSIVE LEAST SQUARE)	285
18.2.6. GIẢI THUẬT MODULUS HẰNG SỐ (CONSTANT MODULUS ALGORITHM)	286
18.3. SỬ DỤNG CÁC BỘ CÂN BẰNG THÍCH NGHI TRONG MATLAB	286
18.3.1. XÁC ĐỊNH GIẢI THUẬT THÍCH NGHI	286
18.3.2. XÂY DỰNG ĐỐI TƯỢNG MÔ TẢ BỘ CÂN BẰNG THÍCH NGHI	288
18.3.3. TRUY XUẤT VÀ HIỆU CHỈNH CÁC ĐẶC TÍNH CỦA BỘ CÂN BẰNG THÍCH NGHI	289

18.3.4. SỬ DỤNG BỘ CÂN BẰNG THÍCH NGHI	289
18.4. CÁC BỘ CÂN BẰNG MLSE	295
PHỤ LỤC	303
TÀI LIỆU THAM KHẢO	324



PHẦN I

CƠ BẢN VỀ MATLAB VÀ LẬP TRÌNH TRÊN MATLAB

Chương 1

MỞ ĐẦU

MATLAB là một công cụ tính toán toán học. MATLAB có thể được sử dụng để tính toán, nó cũng cho phép chúng ta vẽ các biểu đồ, đồ thị theo nhiều cách khác nhau. Giống như một chương trình phần mềm, chúng ta có thể tạo, thực thi và lưu một dãy các lệnh để máy tính có thể chạy tự động. Cuối cùng, MATLAB cũng có thể được coi như là một ngôn ngữ lập trình. Tóm lại, như là một môi trường dùng để lập trình hay tính toán, MATLAB được thiết kế để làm việc với những tập dữ liệu đặc biệt chẳng hạn như ma trận, vector, hình ảnh.

Trong môi trường Windows, sau khi cài MATLAB biểu tượng của nó sẽ xuất hiện trên màn hình của máy tính, chúng ta có thể khởi động MATLAB bằng cách double click vào biểu tượng của nó. Trong khi chạy, tùy theo yêu cầu của người sử dụng, MATLAB sẽ tạo ra một hoặc nhiều cửa sổ trên màn hình. Cửa sổ quan trọng nhất là cửa sổ lệnh (*Command Window*), đây là nơi chúng ta giao tiếp (tương tác) với MATLAB và cũng là nơi chúng ta nhập vào các lệnh và MATLAB sẽ cho ra các kết quả. Chuỗi `>>` là dấu nhắc của chương trình MATLAB. Khi MATLAB hoạt động, con trỏ chuột sẽ xuất hiện sau dấu nhắc, lúc này MATLAB đang chờ người sử dụng nhập lệnh vào. Sau khi nhập lệnh và nhấn enter, MATLAB đáp ứng lại bằng cách in ra các dòng kết quả trong cửa sổ lệnh hay tạo ra một cửa sổ hình (*Figure Window*). Để thoát khỏi chương trình MATLAB chúng ta sử dụng lệnh **exit** hoặc **quit**.

1.1. NHẬP MỘT DÒNG LỆNH

Bảng 1.1: Tương quan giữa các phép toán và lệnh.

Phép toán	Lệnh MATLAB
$a + b$	<code>a + b</code>
$a - b$	<code>a - b</code>
ab	<code>a*b</code>
a/b	<code>a/b</code> hay <code>b\ a</code>
x^b	<code>x^b</code>
$\sqrt{x}$	<code>sqrt(x)</code> hay <code>x^0.5</code>
$ x $	<code>abs(x)</code>
π	<code>pi</code>
$4 \cdot 10^3$	<code>4e3</code> hay <code>4*10^3</code>
i	<code>i</code> hay <code>j</code>
$3 - 4i$	<code>3 - 4*i</code> hay <code>3 - 4*j</code>
e, e^x	<code>exp(1), exp(x)</code>
$\ln x, \log x$	<code>log(x), log10(x)</code>
$\sin x, \arctan x, \dots$	<code>sin(x), atan(x), \dots</code>

MATLAB là một hệ thống tương tác, lệnh sẽ được thực thi ngay lập tức khi nhấn Enter. Những kết quả của mỗi lệnh, nếu được yêu cầu, sẽ được xuất hiện trên màn hình. Tuy nhiên, một lệnh chỉ được thực thi nếu lệnh được nhập vào đúng cú pháp. **Bảng 1.1** là danh sách các

phép toán cơ bản và lệnh tương ứng của chúng được sử dụng trong chương trình MATLAB để giải những phương trình toán học (a, b và x là những số).

Sau đây là một số lưu ý để nhập vào một dòng lệnh đúng:

- Những lệnh trong MATLAB được thực thi ngay lập tức khi nhấn Enter. Kết quả của mỗi lệnh sẽ được hiển thị trên màn hình ngay lập tức. Thử thi hành với các lệnh sau đây:

```
>> 3 + 7.5
```

```
>> 18/4
```

```
>> 3 * 7
```

Lưu ý rằng khoảng trống trong MATLAB là không quan trọng.

- Kết quả của phép tính cuối cùng sẽ được gán cho biến **ans**.

```
>> 14/4
```

```
ans =
```

```
3.5000
```

```
>> ans^(-6)
```

```
ans =
```

```
5.4399e-04
```

5.4399e-04 là một cách thể hiện của 5.4399×10^{-4} .

*Lưu ý rằng **ans** luôn được cập nhật giá trị bởi kết quả của phép tính cuối cùng.*

- Chúng ta cũng có thể định nghĩa những biến mới. Theo dõi giá trị được lưu trong biến a và b:

```
>> a = 14/4
```

```
a =
```

```
3.5000
```

```
>> b = a^(-6)
```

```
b =
```

```
5.4399e-04
```

- Khi một lệnh được kết thúc bởi dấu ‘;’ thì kết quả của nó sẽ không được xuất hiện trên màn hình. Kiểm nghiệm sự khác biệt giữa hai biểu thức sau:

```
>> 3 + 7.5
```

```
>> 3 + 7.5;
```

- Để có thể thực thi nhiều lệnh cùng một lúc, các lệnh cần được cách nhau bởi dấu “,” (hiển thị kết quả) hay cách nhau bởi dấu “;” (không hiển thị kết quả)

```
>> sin(pi/4), cos(pi); sin(0)
```

```
ans =
```

```
0.7071
```

```
ans =
```

```
0
```

Lưu ý rằng trong các kết quả trên giá trị của cos(pi) không được hiển thị.

- Với mỗi giá trị MATLAB mặc định sẽ hiển thị ở dạng có 5 chữ số. Lệnh **format long** sẽ tăng số chữ số hiển thị lên 15 và lệnh **format short** sẽ giảm trở về 5.

```
>> 312/56
```

```
ans =
```

```
5.5714
```

```
>> format long
```

```
>> 312/56
```

```
ans =
```

```
5.57142857142857
```

- Kết quả của mỗi lệnh có thể chứa vài dòng trống, điều này có thể được khắc phục bởi lệnh **format compact**. Ngược lại lệnh **format loose** sẽ thêm vào những dòng trống.

- Để nhập vào một biểu thức quá dài ta sử dụng dấu ‘...’ để xuống hàng

```
>> sin(1) + sin(2) - sin(3) + sin(4) - sin(5) + sin(6) - ...
    sin(8) + sin(9) - sin(10) + sin(11) - sin(12)
```

- MATLAB phân biệt chữ thường và chữ hoa.

- Tất cả các ký tự từ sau dấu ‘%’ đến cuối dòng chỉ có tác dụng ghi chú.

```
>> sin(3.14159)           % gần bằng sin(pi)
```

- Nội dung của lệnh đã thực thi cũng có thể được lấy lại bằng phím ↑. Để thay đổi nội dung của lệnh ta sử dụng các phím mũi tên → và ← để di chuyển con trỏ đến vị trí mong muốn và sửa lệnh. Trong trường hợp lệnh quá dài, **Ctrl-a** và **Ctrl-e** được sử dụng để di chuyển nhanh con trỏ đến vị trí đầu và cuối của lệnh.

- Để gọi lại lệnh đã thực thi bắt đầu bằng ký tự, ví dụ ‘c’, ta nhấn phím ↑ sau khi nhấn phím ‘c’. Điều này cũng đúng với một cụm từ, ví dụ, **cos** theo sau bởi phím ↑ sẽ tìm những lệnh đã thực thi bắt đầu bởi **cos**.

Lưu ý: nên kết thúc mỗi lệnh bằng dấu ‘;’ để tránh trường hợp xuất ra màn hình một kết quả quá lớn, ví dụ xuất ra màn hình một ma trận 1000x1000.

1.2. CÔNG CỤ GIÚP ĐỠ

MATLAB cung cấp một công cụ giúp đỡ trực tiếp. Lệnh **help** là cách đơn giản nhất để được giúp đỡ. Để biết chi tiết hơn về lệnh **help**:

```
>> help help
```

Nếu đã biết tên đề mục hay tên một lệnh cụ thể nào đó, ta có thể sử dụng lệnh **help** một cách cụ thể hơn, ví dụ:

```
>> help ops
```

cho ta biết thông tin về các toán tử và các ký tự đặc biệt trong MATLAB. Khi sử dụng lệnh **help** tên đề mục mà bạn muốn giúp đỡ phải chính xác và đúng. Lệnh **lookfor** hữu dụng hơn trong trường hợp bạn không biết chính xác tên của lệnh hay đề mục. Ví dụ:

```
>> lookfor inverse
```

thể hiện danh sách các lệnh và một mô tả ngắn gọn của các lệnh mà trong phần giúp đỡ có từ **inverse**. Bạn cũng có thể sử dụng một tên không hoàn chỉnh, ví dụ **lookfor inv**. Bên cạnh lệnh **help** và lệnh **lookfor** còn có lệnh **helpwin**, lệnh **helpwin** mở ra một cửa sổ mới thể hiện thư mục các đề mục giúp đỡ.

✎ Bài tập 1-1.

Sử dụng lệnh **help** hoặc **lookfor** để tìm kiếm thông tin cho các câu hỏi sau:

- Hãy tìm lệnh thể hiện phép toán hàm cosin đảo hay $\cos^{-1}$.
- Có phải MATLAB có một hàm toán học dùng để tính ước số chung lớn nhất (the greatest common divisor)?
- Tìm thông tin về hàm logarithms.

1.3. DỪNG MỘT LỆNH HAY CHƯƠNG TRÌNH

Thình thoảng chúng sẽ gặp một lỗi bên trong lệnh hay chương trình của mình, lỗi này có thể làm cho lệnh hay chương trình không thể dừng lại. Để dừng lệnh hay chương trình này lại ta nhấn tổ hợp phím **Ctrl-C** hoặc **Ctrl-Break**. Đôi khi để chương trình dừng lại ta phải làm động tác này vài lần và phải chờ trong vài phút.

1.4. ĐƯỜNG DẪN

Trong MATLAB, lệnh hay chương trình có thể chứa **m-file**, các file này chỉ là các file text và có phần mở rộng là **.m**. Các file này phải được đặt trong các thư mục mà MATLAB thấy được. Danh sách các thư mục này có thể được liệt kê bởi lệnh **path**. Một trong các thư mục mà MATLAB luôn nhìn thấy là thư mục làm việc hiện tại, thư mục này có thể được xác định bởi lệnh **pwd**. Sử dụng hàm **path**, **addpath** và **rmpath** để thêm hay xóa các thư mục đường dẫn. Công việc này cũng có thể được thực hiện từ thanh công cụ: **File – Set path ...**

✎ Bài tập 1-2.

Gõ lệnh **path** để kiểm tra các thư mục có trong đường dẫn. Cộng một thư mục bất kỳ vào trong đường dẫn.

1.5. KHÔNG GIAN LÀM VIỆC (WORKSPACE)

Khi làm việc trong cửa sổ lệnh (*Command Window*), MATLAB sẽ nhớ tất cả các lệnh và tất cả các biến mà chúng ta đã tạo ra. Các lệnh và biến này được hiện thị trong *workspace*. Chúng ta có thể dễ dàng gọi lại các lệnh này khi cần, ví dụ để gọi lệnh trước ta sử dụng phím **↑**. Các giá trị biến có thể được kiểm tra lại bởi lệnh **who**, lệnh **who** sẽ cho danh sách các biến có trong *workspace*. Và lệnh **whos** thể hiện cả tên, kích thước và lớp của biến. Ví dụ, giả sử rằng bạn đã thực thi tất cả các lệnh trong phần 1, khi thực thi lệnh **who** bạn sẽ có được các thông tin như sau:

```
>> who
```

```
Your variables are:
```

```
a      ans      b      x
```

Lệnh **clear <tên biến>** sẽ xóa biến này khỏi *workspace*, **clear** hay **clear all** sẽ xóa tất cả các giá trị biến. Việc xóa tất cả các giá trị biến là cần thiết khi ta bắt đầu một chương trình hay một bài tập mới

```
>> clear a x
```

```
>> who
```

```
Your variables are:
```

```
ans      b
```

1.6. SAVE VÀ LOAD DỮ LIỆU

Cách dễ nhất để save hay load các biến là sử dụng thanh công cụ, chọn **File** và sau đó chọn **Save Workspace as...** hay **Load Workspace...**. MATLAB cũng có lệnh để save dữ liệu vào file hoặc load dữ liệu ra từ file.

Lệnh **Save** sẽ lưu các biến trong *workspace* một file nhị phân hoặc file ASCII, file nhị phân tự động có phần mở rộng **‘.mat’**.

🔗 Bài tập 1-3.

Học cách thực thi lệnh **save**.

```
>> s1 = sin(pi/4);
>> c1 = cos(pi/4); c2 = cos(pi/2);
>> str = 'hello word';           % đây là một chuỗi ký tự.
>> save                          % lưu các biến ở dạng nhị phân vào
                                % file matlab.mat.
>> save numdata s1, c1          % lưu hai biến s1 và c1 vào file
                                % numdata.mat
>> save strdata str             % lưu chuỗi str vào file strdata
>> save allcos.dat c* -ascii    % lưu 2 biến c1 và c2 dưới dạng ascii
                                % vào file allcos.dat
```

Lệnh **load** cho phép chép các biến và giá trị của chúng từ file vào *workspace*. Cú pháp giống như lệnh **save**. Lệnh **load** được sử dụng khi chúng ta cần nạp các giá trị đã được khởi tạo trước vào trong chương trình.

🔗 Bài tập 1-4.

Giả sử rằng chúng ta đã làm bài tập trước, thực thi các lệnh **load** sau đây. Trước mỗi lệnh **load**, sử dụng lệnh **clear all** để xóa *workspace* và sau đó sử dụng lệnh **who** để kiểm tra giá trị các biến có trong *workspace*.

```
>> load                        % load tất cả các biến trong file matlab.mat
>> load data s1 c1            % chỉ load các biến được chỉ định trong file
                                % data.mat
>> load strdata %              load tất cả các biến trong file strdata.mat
Ta cũng có thể đọc được file ASCII, là file chứa một dãy các biến riêng rẽ. Chẳng hạn như file chứa các chú thích được bắt đầu từ ký tự ‘%’. Kết quả được đặt vào biến có cùng tên với file ASCII (ngoại trừ phần mở rộng).
```

```
>> load allcos.dat           % load dữ liệu từ file allcos vào biến allcos
>> who                      % liệt kê tất cả các biến có trong workspace
```

Danh sách các lệnh và hàm được giới thiệu trong chương 1

addpath	Thêm đường dẫn vào danh sách các đường dẫn của MATLAB
clear	Xoá không gian làm việc (workspace)
exit	Thoát khỏi MATLAB
format	Định dạng dữ liệu hiển thị
help	Xem thông tin giúp đỡ về một hàm nào đó
helpwin	Mở một cửa sổ hướng dẫn mới
load	Tải dữ liệu từ file .mat vào không gian làm việc hiện tại
lookfor	Tìm kiếm các hàm MATLAB nhờ từ khoá cho trước
path	Liệt kê các đường dẫn mà MATLAB có thể thấy được
pwd	Xác định thư mục hiện hành của MATLAB
quit	Thoát khỏi MATLAB
rmpath	Xoá một đường dẫn khỏi danh sách các đường dẫn của MATLAB
save	Lưu các biến dữ liệu vào file .mat
who	Liệt kê danh sách các biến có trong workspace
whos	Liệt kê các biến có trong workspace: gồm tên, kích thước và lớp của biến

Chương 2

CƠ BẢN VỀ CÚ PHÁP VÀ BIẾN

2.1. MATLAB NHƯ LÀ MỘT CÔNG CỤ TÍNH TOÁN

Các kiểu số cơ bản được sử dụng trong MATLAB là số nguyên, số thực và số phức. MATLAB cũng có thể biểu diễn các số non-number. Có hai dạng số non-number trong MATLAB: **Inf**, là số vô cực dương được tạo ra bởi phép chia $1/0$ và **NaN**, là số được tạo ra từ các phép toán chẳng hạn như $0/0$ hay $\infty - \infty$.

Như chúng ta đã biết, MATLAB là một công cụ thực sự hữu dụng đối với các phép tính. Chẳng hạn khi nhập vào lệnh:

```
>> (23*17)/7
```

Kết quả sẽ là

```
ans =  
55.8571
```

MATLAB có sáu phép toán số học cơ bản: +, -, *, / hoặc \ và ^ (số mũ).

Lưu ý rằng hai phép toán chia trái và chia phải là khác nhau

```
>> 19/3           % 19/3
```

```
ans =  
6.3333
```

```
>> 19\3, 3/19     % 3/19
```

```
ans =  
0.1579
```

```
ans =  
0.1579
```

Các hàm lượng giác và các hàm mũ logarith cũng được sử dụng trong MATLAB. Sử dụng **help elfun** để liệt kê danh sách các hàm cơ bản có trong MATLAB.

✎ Bài tập 2-1.

Thử tính toán các biểu thức sau đây bằng tay và sau đó sử dụng MATLAB để kiểm tra lại kết quả. Lưu ý sự khác nhau giữa phép chia trái và phải. Sử dụng lệnh **help** để có hướng dẫn về cách sử dụng các lệnh mới, chẳng hạn như các lệnh làm tròn số: **round**, **floor**, **ceil**, ...

- $2/2*3$
- $8*5\backslash 4$
- $8*(5\backslash 4)$
- $7 - 5*4\backslash 9$
- $6 - 2/5 + 7^2 - 1$
- $10/2\backslash 5 - 3 + 2*4$
- $3^2/4$
- $3^2\wedge 3$
- $2 + \text{round}(6/9 + 3*2)/2$
- $2 + \text{floor}(6/9 + 3*2)/2$
- $2 + \text{ceil}(6/9 + 3*2)/2$
- $x = \pi/3, x = x - 1, x = x + 5, x = \text{abs}(x)/x$

🔗 Bài tập 2-2.

Sử dụng các lệnh để định dạng MATLAB không xuất hiện dòng trống trong kết quả và kết quả được xuất ra ở dạng số có 15 chữ số. Thực hiện các lệnh:

```
>> pi
>> sin(pi)
```

sau đó sử dụng lệnh các lệnh **format** để khôi phục lại định dạng chuẩn.

2.2. PHÉP GÁN VÀ BIẾN

Các phép toán liên quan đến số phức được thực thi một cách dễ dàng bởi MATLAB.

🔗 Bài tập 2-3.

- Cho hai số phức bất kỳ, ví dụ $-3 + 2i$ và $5 - 7i$. Hãy thực hiện các phép toán để cộng, trừ, nhân và chia hai số phức này với nhau.

Trong bài tập này với 4 phép tính thì các số phức phải được nhập 4 lần, để đơn giản việc này ta gán mỗi số phức cho một biến. Kết quả của bài tập này sẽ là:

```
>> z = -3 + 2*i;
>> w = 5 - 7*i;
>> y1 = z + w;
>> y2 = z - w;
>> y3 = z*w;
>> y4 = z/w; y5 = w\z;
```

Không giống như các ngôn ngữ lập trình thông thường, trong MATLAB ta không cần phải khai báo biến. Một biến sẽ được tự động tạo ra trong quá trình gán dữ liệu cho biến đó. Mỗi giá trị khi mới tạo ra thì được mặc định có kiểu số là double, kiểu số 32 bit. Chúng ta có thể sử dụng lệnh **single** để chuyển kiểu số từ dạng double sang dạng single, là kiểu số 16 bit.

```
>> a = single(a);
```

Lệnh **single** nên được sử dụng trong trường hợp cần xử lý những ma trận có kích thước lớn. Tuy nhiên trong trường hợp chỉ có vài giá trị được sử dụng thì ta nên chuyển qua dạng double để có được sự chính xác hơn. Sử dụng lệnh **double** để thực hiện phép biến đổi này.

```
>> a = double(a);
```

Lưu ý rằng một biến chưa được định nghĩa thì không được sử dụng để gán cho một biến khác

```
>> clear x;
>> f = x^2 + 4*sin(x);
```

Đoạn lệnh ở trên sẽ không cho ra một kết quả đúng bởi vì giá trị của x chưa được khởi tạo. Biểu thức trên có thể sửa lại bằng cách gán một giá trị bất kỳ cho biến x.

```
>> x = pi;
>> f = x^2 + 4*sin(x);
```

Trong MATLAB, tên của một biến phải được bắt đầu bởi một ký tự chữ, có thể là chữ thường hay chữ in hoa, và theo sau bởi các ký tự chữ, các ký tự số hoặc dấu gạch chân. MATLAB chỉ có thể phân biệt được các biến với nhau bởi tối đa 31 ký tự đầu tiên của tên biến.

✎ Bài tập 2-4.

Sau đây là ví dụ về một vài kiểu biến số khác nhau của MATLAB. Chúng ta sẽ được học kỹ hơn về các lệnh này ở phần sau.

```
>> this_is_my_very_simple_variable_today = 5    % điều gì sẽ xảy ra?
>> 2t = 8;                                     % điều gì sẽ xảy ra?
>> M = [1 2; 3 4; 5 6];                       % một ma trận
>> c = 'E'                                     % một ký tự
>> str = 'Hello word';                         % một chuỗi
>> m = ['J','o','h','n']                      % m là cái gì?
```

Sử dụng lệnh **who** để kiểm tra thông tin về các biến. Sử dụng lệnh **clear** <tên biến> để xóa các biến khỏi *workspace*.

Trong MATLAB có những số được mặc định tạo ra và được xem như là các hằng số, ví dụ như **pi**, **eps**, hay **i**, một số các giá trị khác được cho trong **bảng 2.1**.

Bảng 2.1: Một số biến mặc định trong MATLAB

Tên biến	Giá trị / Ý nghĩa
ans	Tên biến mặc định dùng để lưu kết quả của phép tính cuối cùng.
pi	$\pi = 3.14159 \dots$
eps	Số dương nhỏ nhất
inf	Mô tả số dương ∞
nan hay NaN	Mô tả một not-a-number , ví dụ 0/0
<i>i</i> hay <i>j</i>	$i = j = \sqrt{-1}$
nargin/nargout	Số đối số input/output của hàm
realmin/realmax	Số thực dương nhỏ nhất/lớn nhất có thể

- Các biến được tạo ra bằng cách gán giá trị cho chúng. Một cách khác là gán giá trị của biến này cho biến khác.

```
>> b = 10.5
```

```
>> a = b
```

Theo cách này biến *a* tự động được tạo ra, nếu biến *a* đã tồn tại thì giá trị cũ của nó sẽ bị chồng lên bởi một giá trị mới.

Một biến cũng có thể được tạo ra bởi kết quả của một phép toán:

```
>> a = 10.5;
```

```
>> c = a^2 + sin(pi*a)/4;
```

Kết quả trả về của một hàm có thể được sử dụng để gán và tạo ra các biến mới. Ví dụ, nếu **min** là tên của một hàm (tìm hiểu thêm chức năng và cách sử dụng của hàm **min** bởi lệnh **help min**) thì:

```
>> b = 5; c = 7;
```

```
>> a = min(b, c);
```

 % giá trị nhỏ nhất của b và c
sẽ gọi một hàm với hai biến b và c là hai đối số. Kết quả của hàm này (giá trị trả về của hàm) sẽ được gán cho biến a.

Lưu ý: ta không được sử dụng tên biến trùng với tên hàm. Để kiểm tra một tên nào đó có phải tên hàm hay không ta sử dụng lệnh **help <tên>** để xác định. Nếu kết quả là các hướng dẫn liên quan đến hàm thì tên đó đã được sử dụng để làm tên hàm.

Ví dụ, trong trường hợp thực hiện các vòng lặp liên quan đến số phức, ta không sử dụng biến *i* hoặc *j* để làm biến đếm bởi vì các giá trị này được sử dụng để làm số phức.

Danh sách các lệnh và hàm được giới thiệu trong chương 2

ceil	Làm tròn lên
double	Chuyển sang kiểu số chiều dài 32 bit
floor	Làm tròn xuống
format	Định dạng các dữ liệu số
min	Trả về giá trị nhỏ nhất của hai hay nhiều số
round	Làm tròn về số nguyên gần nhất
single	Chuyển sang kiểu số chiều dài 16 bit
who	Liệt kê các biến có trong workspace

Chương 3

PHÉP TOÁN VỚI VECTOR VÀ MA TRẬN

Trong Matlab, tất cả các đối tượng đều được xem như là một ma trận hay còn được gọi là mảng. Một chữ số được xem như là một ma trận 1×1 và ma trận chỉ có một hàng hay một cột được gọi là vector.

3.1. VECTOR

Trong quá trình khởi tạo, các thành phần của một vector được phân biệt với nhau bởi khoảng trống hoặc dấu ','. Chiều dài của một vector là số thành phần tồn tại trong nó (lệnh **length** được sử dụng để xác định chiều dài của vector). Tất cả các thành phần của một vector phải được đặt trong dấu []:

```
>> v = [-1      sin(3)      7]
v =
    -1.0000    0.1411    7.0000
>> length(v)
ans =
     3
```

Ta có thể áp dụng nhiều phép tính cơ bản khác nhau đối với vector. Một vector có thể nhân với một hệ số hoặc cộng/trừ với một vector khác có cùng chiều dài. Trong phép cộng/trừ, từng thành phần của hai vector cộng/ trừ với nhau và cho ra một vector có cùng chiều dài. Ta cũng có thể sử dụng phép gán đối với vector.

```
>> v = [-1      2      7];
>> w = [2      3      4];
>> z = v + w           % cộng từng thành phần với nhau
z =
     1     5    11
>> vv = v + 2          % 2 được cộng vào từng thành phần của vector v
vv =
     1     4     9
>> t = [2*v, -w]
ans =
    -2     4    14    -2    -3    -4
```

Mỗi thành phần của vector cũng có thể được sử dụng một cách riêng biệt:

```
>> v(2) = -1           % thay đổi giá trị thành phần thứ 2 của v
v =
    -1    -1     7
>> w(2)                % hiển thị giá trị thành phần thứ 2 của w
ans = 3
```

3.1.1. DẤU ‘:’ VÀ PHẦN TRÍCH RA TỪ VECTOR

Dấu ‘:’ là một toán tử quan trọng, nó được sử dụng để xử lý với các vector hàng (xem thêm ở **bảng 3.1** hoặc sử dụng lệnh **help colon** để có nhiều thông tin hơn về toán tử này):

Bảng 3.1: Những thành phần con của ma trận

Lệnh	Kết quả
$A(i,j)$	A_{ij}
$A(:,j)$	Cột thứ j của A
$A(i,:)$	Hàng thứ i của A
$A(k:l,m:n)$	Ma trận con của ma trận A
$v(i:j)$	Một phần của vector v

```
>> 2:5
```

```
ans =
```

```
2    3    4    5
```

```
>> -2:3
```

```
ans =
```

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

Một cách tổng quát, lệnh có cấu trúc **first:step:last** sẽ tạo ra một vector có thành phần đầu tiên bằng **first**, giá trị của các thành phần tiếp theo được tăng bởi **step** và thành phần cuối cùng có giá trị $\leq$ **last**:

```
>> 0.2:0.5:2.4
```

```
ans =
```

```
0.2000    0.7000    1.2000    1.7000    2.2000
```

```
>> -3:3:10
```

```
ans =
```

```
-3    0    3    6    9
```

```
>> 1.5:-0.5:-0.5 % step cũng có thể là số âm
```

```
ans =
```

```
1.5000    1.0000    0.5000    0    -0.5000
```

Toán tử ‘:’ cũng được sử dụng để trích ra một số thành phần từ một vector.

```
>> r = [-1:2:6, 2, 3, -2] % -1:2:6  $\equiv$  -1    1    3    5
```

```
r =
```

```
-1    1    3    5    2    3    -2
```

```
>> r(3:6) % các giá trị của r từ 3 đến 6
```

```
ans =
```

```
3    5    2    3
```

```
>> r(1:2:5) % lấy các vị trí 1, 3, 5
```

```
ans =
```

```
-1    3    2
>> r(5:-1:2)    % điều gì sẽ xảy ra?
```

3.1.2. VECTOR CỘT VÀ PHÉP CHUYỂN VỊ

Đối với một vector cột, các thành phần của nó phải được phân biệt với nhau bởi dấu ‘;’ hoặc xuống dòng:

```
>> z = [1
        7
        7];
z =
     1
     7
     7
>> u = [-1;    3;    5]
u =
    -1
     3
     5
```

Những phép toán áp dụng với vector hàng cũng có thể được sử dụng ở vector cột. Tuy nhiên, chúng ta không thể cộng một vector hàng với một vector cột. Để thực hiện được phép tính này, ta cần sử dụng toán tử chuyển vị, toán tử này sẽ chuyển một vector hàng thành một vector cột và ngược lại:

```
>> u'                % u là vector cột, u' sẽ là vector hàng
ans =
    -1     3     5
>> v = [-1    2    7]; % v là một vector hàng
>> u + v              % không thể cộng một vector cột và vector hàng
??? Error using ==> +
Matrix dimensions must agree.
>> u' + v
ans =
    -2     5    12
>> u + v'
ans =
    -2
     5
    12
```

Nếu z là một vector phức thì z' sẽ cho ra một chuyển vị liên hợp của z , nghĩa là từng thành phần của z' là liên hợp phức với các thành phần trong z . Trong trường hợp cần chuyển vị theo kiểu thông thường (các số phức được giữ nguyên giá trị) ta phải sử dụng toán tử ‘.’

```
>> z = [1+2i, -1+i]
z =
    1.0000 + 2.0000i    -1.0000 + 1.0000i
>> z'                                % chuyển vị liên hợp
ans =
    1.0000 - 2.0000i
   -1.0000 - 1.0000i
>> z.'                                % phép chuyển vị thông thường
ans =
    1.0000 + 2.0000i
   -1.0000 + 1.0000i
```

3.1.3. NHÂN, CHIA VÀ SỐ MŨ CỦA VECTOR

Chúng ta có thể nhân hai vector có cùng chiều dài, $x^T y = \sum_i x_i y_i$ theo cách đơn giản:

```
>> u = [-1;    3;    5]    % một vector cột
>> v = [-1;    2;    7]    % một vector cột
>> u * v                    % không thể nhân 2 vector cột với nhau
??? Error using ==> *
Inner matrix dimensions must agree.
>> u' * v                    % kết quả nhân 2 vector
ans =
    42
```

Một cách khác để nhân hai vector là sử dụng toán tử `.*`. Với toán tử này các thành phần tương ứng của hai vector sẽ được nhân với nhau. Cho hai vector x và y có cùng chiều dài, tích `.*` của hai vector là $[x_1 y_1, x_2 y_2, \dots, x_n y_n]$:

```
>> u .* v                    % nhân từng thành phần tương ứng
    1
    6
   35
>> sum(u.*v)                 % tương đương phép nhân 2 vector
ans =
    42
>> z = [4    3    1];      % z là vector hàng
>> sum(u' .* z)              % phép nhân 2 vector
ans =
    10
>> u' * z'                  % tích 2 vector
ans =
```

10

Ví dụ 3-1:

```
>> x = 1:0.5:4;
>> y = sqrt(x) .* cos(x)
y =
    0.5403    0.0866   -0.5885   -1.2667   -1.7147   -1.7520   -1.3073
```

Trong toán học không tồn tại phép chia hai ma trận cho nhau. Tuy nhiên, trong MATLAB toán tử ‘./’ được định nghĩa như là phép chia từng thành phần tương ứng của hai ma trận với nhau. Kết quả cũng là một ma trận có cùng kích thước:

```
>> x = 2:2:10
x =
     2     4     6     8    10
>> y = 6:10
y =
     6     7     8     9    10
>> x./y
ans =
    0.3333    0.5714    0.7500    0.8889    1.0000
>> z = -1:3
z =
    -1     0     1     2     3
>> x./z
% 4/0, kết quả là Inf
Warning: Divide by zero.
ans =
   -2.0000    Inf    6.0000    4.0000    3.3333
>> z./z
% 0/0, kết quả là NaN
Warning: Divide by zero.
ans =
     1    NaN     1     1     1
```

Toán tử ‘./’ cũng có thể được sử dụng để chia một số cho một vector:

```
>> x=1:5; 2/x
% chương trình báo lỗi
??? Error using ==> /
Matrix dimensions must agree.
>> 2./x
% phép tính đúng
ans =
    2.0000    1.0000    0.6667    0.5000    0.4000
```

Bài tập 3-1.

một vector. Các thành phần của hàng được phân biệt với nhau bởi dấu ‘,’ hoặc khoảng trống, còn các hàng được phân biệt bởi dấu ‘;’. Ví dụ ma trận $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ được định nghĩa như

sau:

```
>> A = [1    2    3;    4    5    6;    7    8    9]
A =
     1     2     3
     4     5     6
     7     8     9
```

Một số ví dụ khác:

```
>> A2 = [1:4; -1:2:5]
A2 =
     1     2     3     4
    -1     1     3     5
>> A3 = [1     3
        -4     7]
A3 =
     1     3
    -4     7
```

Từ những ví dụ ở trên ta nhận thấy rằng một vector hàng là một ma trận $1 \times k$ và một vector cột là một ma trận $n \times 1$. Phép chuyển vị sẽ chuyển một vector hàng thành một vector cột và ngược lại. Điều này có thể mở rộng cho một ma trận, phép chuyển vị sẽ biến các hàng của ma trận thành các cột và ngược lại.

```
>> A2
A2 =
     1     2     3     4
    -1     1     3     5
>> A2'                                % chuyển vị của ma trận A2
ans =
     1    -1
     2     1
     3     3
     4     5
>> size(A2)                           % kích thước của ma trận A2
ans =
     2     4
>> size(A2')
```

ans =

4 2

3.2.1. NHỮNG MA TRẬN ĐẶC BIỆT

Trong MATLAB có một số hàm được sử dụng để tạo ra các ma trận đặc biệt, tham khảo thêm ở bảng 3.2.

Bảng 3.2: Một số hàm và phép toán thường sử dụng với ma trận.

Lệnh	Kết quả
$n = \text{rank}(A)$	Số chiều của ma trận A
$x = \text{det}(A)$	Định thức của ma trận A
$x = \text{size}(A)$	Kích thước của A
$x = \text{trace}(A)$	Tổng các thành phần trên đường chéo của A
$x = \text{norm}(v)$	Chiều dài Euclide của vector v
$C = A + B$	Tổng hai ma trận
$C = A - B$	Hiệu hai ma trận
$C = A * B$	Tích hai ma trận
$C = A .* B$	Tích từng thành phần tương ứng của hai ma trận
$C = A^k$	Lũy thừa của ma trận
$C = A.^k$	Lũy thừa từng thành phần của ma trận
$C = A'$	Ma trận chuyển vị A^T
$C = A ./ B$	Chia từng thành phần tương ứng của hai ma trận
$C = \text{inv}(A)$	Nghịch đảo của ma trận A
$X = A \setminus B$	Giải phương trình $AX = B$
$X = B \setminus A$	Giải phương trình $XA = B$
$x = \text{linspace}(a,b,n)$	Vector x có n thành phần phân bố đều trong khoảng [a,b]
$x = \text{logspace}(a,b,n)$	Vector x có n thành phần bắt đầu 10^a và kết thúc với 10^b
$A = \text{eye}(n)$	Ma trận đồng nhất
$A = \text{zeros}(n,m)$	Ma trận all-0
$A = \text{ones}(n,m)$	Ma trận all-1
$A = \text{diag}(v)$	Ma trận zero với đường chéo là các thành phần của vector v
$X = \text{tril}(A)$	Trích ra ma trận tam giác dưới
$X = \text{triu}(A)$	Trích ra ma trận tam giác trên
$A = \text{rand}(n,m)$	Ma trận A với các thành phần là phân bố đồng nhất giữa (0,1)
$A = \text{randn}(n,m)$	Giống như trên với các thành phần phân bố chuẩn.
$v = \text{max}(A)$	Nếu A là một vector thì v là giá trị lớn nhất của A Nếu A là ma trận thì v là một vector với các thành phần là giá trị lớn nhất trên mỗi cột của A
$v = \text{min}(A)$	Như trên với giá trị nhỏ nhất
$v = \text{sum}(A)$	Như trên với tổng

```
>> E = [] % một ma trận rỗng 0 hàng 0 cột
E =
[]
>> size(E)
ans =
0 0
>> I = eye(3); % ma trận đồng nhất 3x3
I =
1 0 0
0 1 0
0 0 1
>> x = [2; -1; 7]; I*x % I*x = x
ans =
2
-1
7
>> r = [1 3 -2];
>> R = diag(r) % tạo một ma trận đường chéo
R =
1 0 0
0 3 0
0 0 -2
>> A = [1 2 3; 4 5 6; 7 8 9];
>> diag(A) % trích ra đường chéo của ma trận A
ans =
1
5
9
>> B = ones(3,2)
B =
1 1
1 1
1 1
>> C = zeros (size(C')) % ma trận all zero với kích thước của C'
C =
0 0 0
0 0 0
```

```
>> D = rand(2,3)           % ma trận các giá trị ngẫu nhiên trong
                             khoảng (0, 1)

D =
    0.0227    0.9101    0.9222
    0.0299    0.0640    0.3309

>> v = linspace(1, 2, 4)   % v là một vector có 4 giá trị cách đều
                             nhau trong khoảng [1, 2]

v =
    1.0000    1.3333    1.6667    2.0000
```

3.2.2. XÂY DỰNG MA TRẬN VÀ TRÍCH RA MỘT MA TRẬN CON TỪ MỘT MA TRẬN LỚN HƠN

Chúng ta có thể tạo ra một ma trận lớn từ các ma trận nhỏ hơn:

```
>> x = [4; -1], y = [-1 3]

x =
     4
    -1

y =
    -1     3

>> X = [x y']               % X bao gồm các cột của x và y'

X =
     4     -1
    -1     3

>> T = [ -1 3 4; 4 5 6]; t = 1:3;

>> T = [T; t]               % cộng thêm vào T một dòng mới, t

T =
    -1     3     4
     4     5     6
     1     2     3

>> G = [1 5; 4 5; 0 2];     % G là ma trận 3X2

>> T2 = [T G]              % kết nối 2 ma trận

T2 =
    -1     3     4     1     5
     4     5     6     4     5
     1     2     3     0     2

>> T3 = [T; G ones(3,1)]    % G là ma trận 3X2
                             % T là ma trận 3X3

T3 =
    -1     3     4
     4     5     6
```

$$A =$$

1	2	3
4	5	8
7	8	9
0	0	0
0	5	0

Các thành phần zero của ma trận A ở trên cũng có thể được thay đổi:

```
>> A(4,:) = [2, 1, 2];      % gán vector [2, 1, 2] vào hàng thứ 4 của A
```

```
>> A(5,[1,3]) = [4, 4];    % gán A(5,1) = 4 và A(5,3) = 4
```

```
>> A % kiểm tra sự thay đổi của ma trận A?
```

Những phần khác nhau của ma trận A được trích ra theo cách sau đây:

```
>> A(3,:)                  % trích ra hàng thứ 3 của A
```

```
ans =
```

7	8	9
---	---	---

```
>> A(:,2)                  % trích ra cột thứ 2 của A
```

```
ans =
```

2
5
8
1
5

```
>> A(1:2,:)                % trích ra hàng thứ 1 và 2 của A
```

```
ans =
```

1	2	3
4	5	8

```
>> A([2,5],1:2)           % trích ra một phần của A
```

```
ans =
```

4	5
4	5

Các lệnh ở những ví dụ trên được giải thích ngắn gọn trong bảng 3.1.

Lưu ý khái niệm ma trận rỗng [], chẳng hạn các hàng hay cột của ma trận có thể được xóa bỏ bằng cách gán giá trị của nó cho ma trận rỗng [].

```
>> C = [1 2 3 4; 5 6 7 8; 1 1 1 1];
```

```
>> D = C; D(:,2) = []      % xóa cột thứ 2 của D
```

```
>> C([1,3],:) = []        % xóa cột thứ 1 và 2 của C
```

Bài tập 3-3.

Hãy xóa tất cả các biến (sử dụng lệnh **clear**). Định nghĩa ma trận $A = [1:4; 5:8; 1 \ 1 \ 1 \ 1]$. Hãy thực thi và kiểm tra kết quả của các phép tính sau:

- $x = A(:, 3)$
- $B = A(1 : 3, 2 : 2)$
- $A(1, 1) = 9 + A(2, 3)$
- $A(2 : 3, 1 : 3) = [0 \ 0 \ 0; 0 \ 0 \ 0]$
- $A(2 : 3, 1 : 2) = [1 \ 1; 3 \ 3]$
- $y = A(3 : 3, 1 : 4)$
- $A = [A; 2 \ 1 \ 7 \ 7; 7 \ 7 \ 4 \ 5]$
- $C = A([1, 3], 2)$
- $D = A([2, 3, 5], [1, 3, 4])$
- $D(2, :) = []$

✎ Bài tập 3-4.

Cho $A = [1, 5, 6; 3, 0, 8]$, $B = [7, 3, 5; 2, 8, 1]$, $C = 10$ và $D = 2$. Hãy thực hiện các phép tính sau đây:

- $E = A - B$
- $F = D * B$
- $G = A .* B$
- $H = A'$
- $J = B/D$
- Gán cột đầu tiên của A cho M
- Gán cột thứ hai của của G cho N
- Chỉ nhân cột thứ 3 của A cho 5
- Cộng M và N
- Tìm tổng tất cả các giá trị của ma trận A

✎ Bài tập 3-5.

Định nghĩa ma trận $T = [3 \ 4; 1 \ 8; -4 \ 3]$; $A = [\text{diag}(-1:2:3) \ T; -4 \ 4 \ 1 \ 2 \ 1]$. Thực hiện các phép biến đổi sau đây đối với ma trận A :

- Trích ra một vector bao gồm thành phần thứ 2 và 4 của hàng thứ 3.
- Tìm giá trị nhỏ nhất của cột thứ 3
- Tìm giá trị lớn nhất của hàng thứ 2
- Tính tổng các thành phần của cột thứ 2
- Tính giá trị trung bình của hàng thứ 1 và thứ 4.
- Trích ma trận con bao gồm hàng thứ 1 và thứ 3.
- Trích ma trận con bao gồm hàng thứ 1 và 2 của cột 3, 4, 5.
- Tính tổng các thành phần của hai hàng 1 và 2.
- Cộng các thành phần của cột thứ 2 và thứ 3 với 3.

✎ Bài tập 3-6.

Cho ma trận $A = [2 \ 4 \ 1; 6 \ 7 \ 2; 3 \ 5 \ 9]$. Thực thi các phép toán sau đối với ma trận A :

- Gán hàng thứ 1 cho vector x
- Gán 2 hàng cuối của A cho y .
- Cộng các thành phần trong từng hàng của A
- Cộng các thành phần trong từng cột của A

✎ Bài tập 3-7.

Cho $A = [2 \ 7 \ 9 \ 7; 3 \ 1 \ 5 \ 6; 8 \ 1 \ 2 \ 5]$. Giải thích kết quả của các lệnh sau:

- A'
- $\text{sum}(A)$

- $A(1, :)$
- $A(:, [1 \ 4])$
- $A([2 \ 3], [3 \ 1])$
- $\text{reshape}(A, 2, 6)$
- $A(:)$
- $\text{flipud}(A)$
- $\text{fliplr}(A)$
- $[A; A(\text{end}, :)]$
- $[A; A(1 : 2, :)]$
- $\max(\min(A))$
- $[[A; \text{sum}(A)] [\text{sum}(A, 2); \text{sum}(A(:))]]$
- Gán các cột chẵn của A cho B
- Gán các hàng lẻ của A cho C
- Biến A thành ma trận 4×3
- Tính $1/x$ các thành phần của A
- Tính bình phương các thành phần A
- Cộng một hàng all-1 vào đầu và cuối A
- Hoán đổi hai hàng 2 và 3
- $\text{sum}(A')$
- $\text{mean}(A)$
- $\text{mean}(A')$
- $\text{sum}(A, 2)$
- $\text{mean}(A, 2)$
- $\min(A)$
- $\max(A')$
- $\min(A(:, 4))$
- $[\min(A)' \max(A)']$
- Xóa cột thứ 2 của A

Lưu ý: sử dụng lệnh **help** để tìm hiểu ý nghĩa của các lệnh mới.

3.2.3. TÍNH TOÁN VỚI MA TRẬN

Các hàm và phép toán thường sử dụng với ma trận được cho trong bảng 3.2. Lưu ý toán tử '.' trong phép nhân ma trận với ma trận và phép nhân ma trận với vector. Toán tử '.' xuất hiện trong phép nhân, phép chia và số mũ. Khi có toán tử này, phép toán sẽ được thực hiện với từng thành phần của ma trận. Cụ thể trong phép nhân/chia, từng thành phần tương ứng của 2 ma trận sẽ nhân/chia với nhau và kết quả sẽ là một ma trận có cùng kích thước với 2 ma trận ban đầu. Như vậy trong trường hợp sử dụng toán tử này 2 ma trận phải có cùng kích thước với nhau. Xem xét các ví dụ sau:

```
>> B = [1    -1    3; 4    0    7]
B =
     1     -1     3
     4      0     7

>> B2 = [1 2; 5 1; 5 6];
>> B = B + B2' % cộng 2 ma trận
B =
     2      4      8
     6      1     13
```

```
>> B-2                                % trừ các thành phần của B cho 2
ans =
     0     2     6
     4    -1    11

>> ans = B./4                          % chia các thành phần của B cho 4
ans =
     0.5000     1.0000     2.0000
     1.5000     0.2500     3.2500

>> 4/B                                % sai cú pháp
??? Error using ==> /
Matrix dimensions must agree.

>> 4./B                                % tương đương với 4.*ones(size(B))./B
ans =
     2.0000     1.0000     0.5000
     0.6667     4.0000     0.3077

>> C = [1 -1 4; 7 0 -1];
>> B .* C                              % nhân từng vị trí với nhau
ans =
     2     -4     32
    42     0    -13

>> ans.^3 - 2                          % mũ 3 các thành phần sau đó trừ cho 2
ans =
         6     -66    32766
    74086     -2    -2199

>> ans ./ B.^2                         % từng vị trí chia cho nhau
ans =
     0.7500    -1.0312    63.9961
    342.9907    -2.0000    -1.0009

>> r = [1 3 -2]; r * B2
ans =
     6     -7
```

Lưu ý các phép nhân ma trận-ma trận và phép nhân ma trận-vector.

```
>> b = [1 3 -2];
>> B = [1 -1 3; 4 0 7]
B =
     1     -1     3
     4      0     7
```

```
>> b * B           % không thể thực hiện được
??? Error using ==> *
Inner matrix dimensions must agree.
>> b * B'           % thực hiện được
ans =
    -8    -10
>> B' * ones(2,1)
ans =
     5
    -1
    10
>> C = [3 1; 1 -3];
>> C * B
ans =
     7     -3    16
    -11     -1   -18
>> C.^3             % mũ 3 từng thành phần
ans =
    27     1
     1   -27
>> C^3             % tương đương với C*C*C
ans =
    30    10
    10   -30
>> ones(3,4)./4 * diag(1:4)
ans =
    0.2500    0.5000    0.7500    1.0000
    0.2500    0.5000    0.7500    1.0000
    0.2500    0.5000    0.7500    1.0000
```

✎ Bài tập 3-8.

Hãy thực thi tất cả phép toán trong bảng 3.2, tự chọn các giá trị cho ma trận A, B vector v và các hệ số k, a, b, n và m.

✎ Bài tập 3-9.

Cho A là một ma trận vuông, tạo ma trận B giống ma trận A nhưng tất cả các thành phần trên đường chéo chính đều bằng 1.

✎ Bài tập 3-10.

Cho vector $x = [1 \ 3 \ 7]$, $y = [2 \ 4 \ 2]$ và ma trận $A = [3 \ 1 \ 6; 5 \ 2 \ 7]$, $B = [1 \ 4; 7 \ 8; 2 \ 2]$. Phép toán nào sau đây là không đúng? Kết quả của mỗi phép tính?

- $x + y$
- $x + A$
- $x' + y$
- $A - [x' \ y']$
- $[x; y] + A$
- $B * A$
- $A .* B$
- $A' .* B$
- $2 * B$
- $2 .* B$
- $[x; y']$
- $[x; y]$
- $A - 3$
- $A + B$
- $B' + A$
- $B ./ x'$
- $B ./ [x' \ x']$
- $2 / A$
- $\text{ones}(1, 3) * A$
- $\text{ones}(1, \text{ê5})$

🔗 **Bài tập 3-11.**

Cho A là một ma trận ngẫu nhiên 5×5 , b là một vector ngẫu nhiên 5×1 . Tìm x thỏa mãn biểu thức $Ax = b$ (tham khảo thêm ở bảng 3.2). Giải thích sự khác nhau giữa toán tử '\', '/' và lệnh **inv**. Sau khi có x , hãy kiểm tra $Ax - b$ có phải là một vector all-zero hay không?

🔗 **Bài tập 3-12.**

Hãy tìm hai ma trận 2×2 A và B thỏa mãn $A .* B \neq A * B$. Sau đó tìm tất cả các ma trận A và B sao cho $A .* B = A * B$ (gợi ý: sử dụng các toán tử '/', '\', ' và lệnh **inv**).

Danh sách các lệnh và hàm được giới thiệu trong chương 3

det	Định thức của ma trận
diag	Tạo ma trận đường chéo
eye	Ma trận đơn vị
inv	Nghịch đảo của 1 ma trận
length	Chiều dài của vector
linspace	Chia một đoạn thành các khoảng chia tuyến tính
logspace	Chia một đoạn thành các khoảng chia logarithm
max	Hàm giá trị lớn nhất
min	Hàm giá trị nhỏ nhất
norm	Chiều dài Euclide của vector
ones	Ma trận toàn 1
rand	Ma trận ngẫu nhiên với các thành phần phân bố đều trên (0,1)
randn	Ma trận ngẫu nhiên với các thành phần phân bố chuẩn
rank	Hạng của ma trận
size	Kích thước của ma trận
sum	Hàm tính tổng
trace	Tổng các thành phần trên đường chéo của ma trận (vết của ma trận)
tril	Trích ra ma trận tam giác dưới
triu	Trích ra ma trận tam giác trên
zeros	Ma trận toàn zero

Chương 4

ĐỒ THỊ 2D VÀ 3D

MATLAB có thể được sử dụng để thể hiện các kết quả dưới dạng đồ thị, mỗi biến sẽ chứa tất cả các giá trị của một đối số trong lệnh vẽ đồ thị.

4.1. NHỮNG ĐỒ THỊ ĐƠN GIẢN

Với lệnh **plot** chúng ta dễ dàng vẽ được những đồ thị đơn giản. Cho vector **y**, lệnh **plot(y)** sẽ xác định những điểm $[1, y(1)]$, $[2, y(2)]$, ..., $[n, y(n)]$ và nối các điểm này lại bằng những đường thẳng. Lệnh **plot(x,y)** thực hiện một công việc tương tự như vậy với những điểm $[x(1), y(1)]$, $[x(2), y(2)]$, ..., $[x(n), y(n)]$.

Lưu ý rằng hai vector **x** và **y** phải cùng là vector hàng hoặc vector cột và có cùng chiều dài (số thành phần trong vector).

Những lệnh **loglog**, **semilogx** và **semilogy** có chức năng tương tự như lệnh **plot**, ngoại trừ một hoặc hai trục đồ thị của chúng được xác định theo logarith.

✎ Bài tập 4-1.

Dự đoán kết quả của đoạn chương trình sau đây, sau đó hãy kiểm chứng kết quả bằng MATLAB:

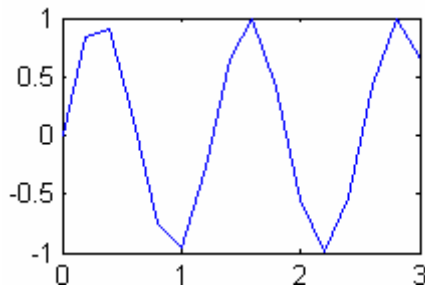
```
>> x = 0:10;
>> y = 2.^x;          % y = [1 2 4 8 16 32 64 128 256 512 1024]
>> plot(x,y)          % biểu diễn dưới dạng đồ thị
>> semilogy(x,y)      % vẽ đồ thị với trục y theo logarith
```

Sau khi thực thi xong đoạn chương trình trên ta nhận thấy, cả hai đồ thị đều được thể hiện trong cùng một cửa sổ **Figure No.1**. Đồ thị thứ nhất sẽ bị xóa bỏ ngay khi đồ thị thứ hai xuất hiện. Để vẽ hai đồ thị trên hai cửa sổ khác nhau, ta sử dụng lệnh **figure** để tạo ra một cửa sổ mới trước khi thực hiện lệnh vẽ đồ thị thứ hai. Bằng cách này chúng ta sẽ có hai cửa sổ riêng biệt để thể hiện 2 đồ thị. Chúng ta cũng có thể chuyển đến các cửa sổ khác nhau bằng lệnh **figure(n)**, lệnh này sẽ đưa cửa sổ **No.n** lên trên màn hình. Thực thi lại đoạn chương trình trên và quan sát sự thay đổi.

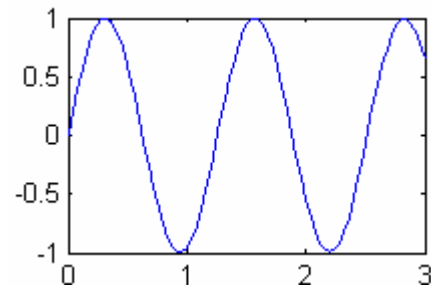
```
>> x = 0:10;
>> y = 2.^x;
>> plot(x,y)
>> figure
>> semilogy(x,y)
```

Để vẽ một đồ thị hàm tương đối chính xác và đẹp, điều quan trọng là phải lấy mẫu một cách thích hợp:

```
>> n = 5;
>> x = 0:1/n:3;      % lấy mẫu không tốt
>> y = sin(5*x);
>> plot(x,y)
```



Lấy mẫu không tốt

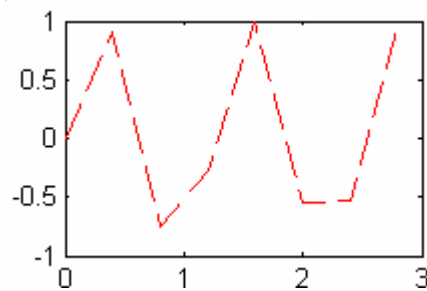


Lấy mẫu tốt

```
>> n = 25;
>> x = 0:1/n:3; % lấy mẫu tốt
>> y = sin(5*x);
>> plot(x,y)
```

Lệnh **plot** mặc định sẽ vẽ đồ thị bằng những đường nét liền màu đen. Ta có thể thay đổi kiểu cũng như màu sắc của nét vẽ, ví dụ:

```
>> x = 0:0.4:3; y = sin(5*x);
>> plot(x,y, 'r--')
```



Hình 4.1.

đồ thị trên được vẽ bởi đường nét đứt màu đỏ. Thông số thứ ba của lệnh **plot** chỉ định màu và kiểu đường của nét vẽ. Bảng 4.1 trình bày một số trường hợp có thể, sử dụng lệnh **help plot** để có những thông tin chi tiết hơn.

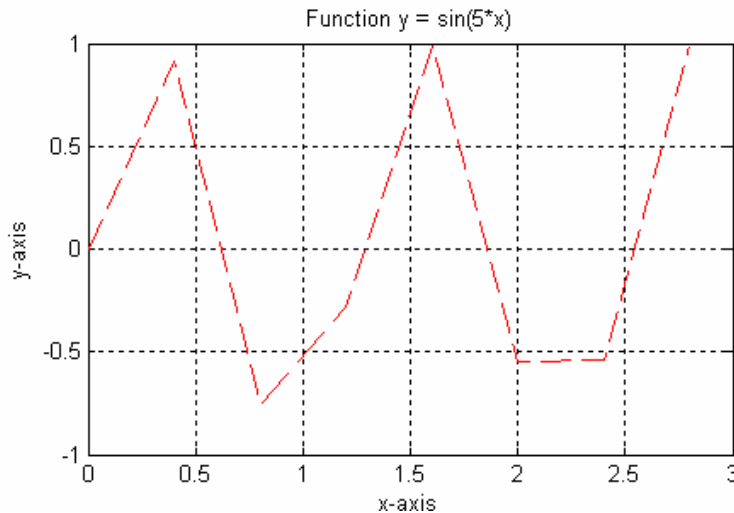
Bảng 4.1: Các ký hiệu màu và kiểu vẽ của đồ thị

Ký hiệu	Màu	Ký hiệu	Kiểu vẽ
r	Red	., o	Đánh dấu các điểm bằng dấu '.,', 'o'
g	Green	*	Đánh dấu các điểm bằng dấu '*'
b	Blue	x, +	Đánh dấu các điểm bằng dấu 'x', '+'
y	Yellow	-	Vẽ bằng đường nét liền
m	Magenta	--	Vẽ bằng đường nét đứt dài
c	Cyan	:	Vẽ bằng đường nét đứt ngắn
k	Black	-. ,	Vẽ bằng đường nét đứt chấm – gạch.

Tiêu đề của đồ thị, đường kẻ và nhãn cho các trục được xác định bởi các lệnh sau:

```
>> title('Function y = sin(5*x)');
```

```
>> xlabel('x-axis');
>> ylabel('y-axis');
>> grid % loại bỏ các đường kẻ bằng lệnh grid off
```



Hình 4.2.

✎ Bài tập 4-2.

Vẽ một đường bằng nét gạch ngắn màu đỏ nối các điểm sau lại với nhau: (2, 6), (2.5, 18), (5, 17.5), (4.2, 12.5) và (2, 12).

✎ Bài tập 4-3.

Vẽ đồ thị của hàm $y = \sin(x) + x - x\cos(x)$ trong hai cửa sổ riêng biệt với hai khoảng $0 < x < 30$ và $-100 < x < 100$. Cộng thêm tiêu đề và mô tả của các trục vào đồ thị.

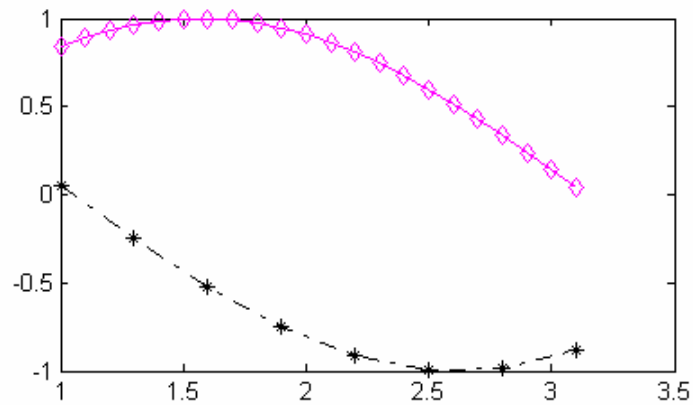
✎ Bài tập 4-4.

Vẽ đường tròn có bán kính bằng 2, biết rằng phương trình của đường tròn là $[x(t); y(t)] = [r \cos(t); r \sin(t)]$ với $t = [0; 2\pi]$.

4.2. MỘT SỐ HÀM ĐƯỢC SỬ DỤNG TRONG VẼ ĐỒ THỊ

Có vài cách để vẽ nhiều đồ thị trong cùng một cửa sổ hình. Cách đầu tiên là sử dụng lệnh **hold on**. Sau lệnh này, tất cả các đồ thị sẽ được vẽ trên cùng một cửa sổ hình cho đến khi có lệnh **hold off**. Khi nhiều hàm được vẽ trên cùng một cửa sổ, ta nên sử dụng màu sắc và hình dạng khác nhau cho mỗi đồ thị. Ví dụ:

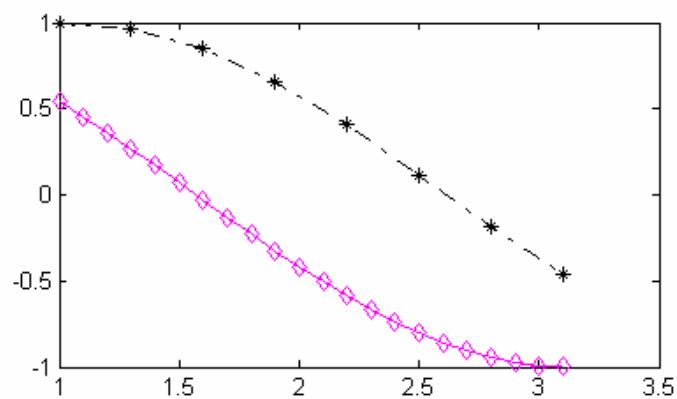
```
>> x1 = 1:.1:3.1; y1 = sin(x1);
>> plot(x1,y1,'md');
>> x2 = 1:.3:3.1; y2 = sin(-x2+pi/3);
>> hold on
>> plot(x2,y2,'k*-.')
>> plot(x1,y1,'m-')
>> hold off
```



Hình 4.3.

Cách thứ hai là vẽ nhiều hàm cùng một lúc. Với cách này, các hàm sẽ được vẽ cùng một lúc trên cùng một cửa sổ hình:

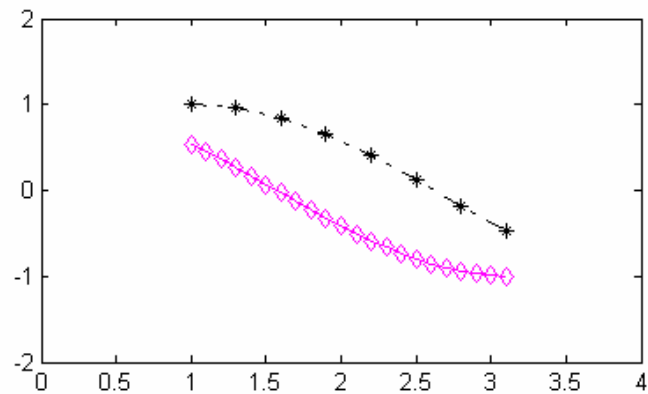
```
>> x1 = 1:.1:3.1; y1 = cos(x1);
>> x2 = 1:.3:3.1; y2 = cos(-x2+pi/3);
>> plot(x1, y1, 'md', x2, y2, 'k*-.', x1, y1, 'm-')
```



Hình 4.4.

Để xác định cụ thể độ dài của mỗi trục ta sử dụng lệnh **axis**. Để thấy sự thay đổi hãy thêm lệnh sau vào đoạn lệnh ở trên.

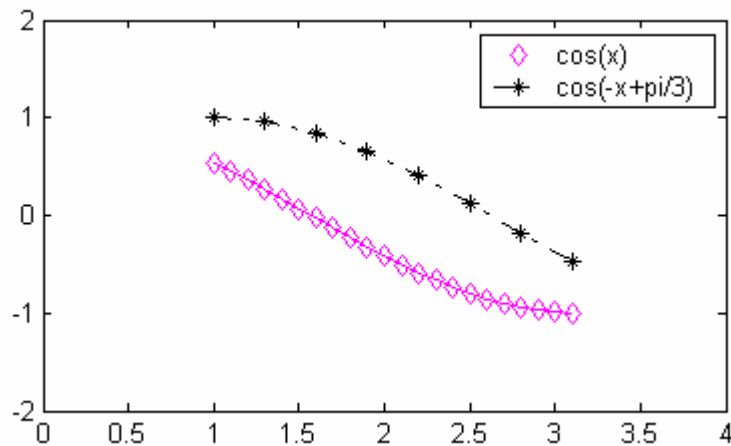
```
>> axis([0,4,-2,2])
```



Hình 4.5.

Lệnh **axis tight** cũng cho ra kết quả tương tự. Sử dụng lệnh **help** để tìm hiểu thêm về các lệnh **axis on/off**, **axis equal**, **axis image** và **axis normal**. Để tạo dòng ghi chú cho mỗi đồ thị ta sử dụng lệnh **legend**, thêm lệnh sau vào đoạn chương trình ở trên

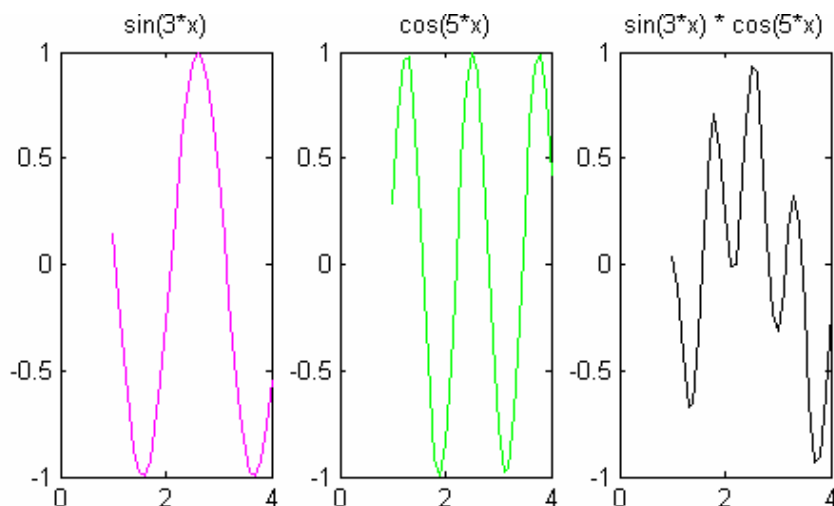
```
>> legend ('cos(x)', 'cos(-x+pi/3)');
```



Hình 4.6.

Để tạo ra nhiều cửa sổ con trong cùng một cửa sổ hình ta sử dụng lệnh **subplot**. Với lệnh **subplot**, cửa sổ hình được chia thành $p \times r$ cửa sổ con được đánh số từ 1 đến $p \cdot r$, bắt đầu từ cửa sổ trên cùng bên trái và đếm theo từng hàng. Các lệnh **plot**, **title**, **grid** chỉ làm việc với cửa sổ con hiện tại.

```
>> x = 1:.1:4;
>> y1 = sin(3*x);
>> y2 = cos(5*x);
>> y3 = sin(3*x).*cos(5*x);
>> subplot(1,3,1); plot(x,y1,'m-'); title('sin(3*x)')
>> subplot(1,3,2); plot(x,y2,'g-'); title('cos(5*x)')
>> subplot(1,3,3); plot(x,y3,'k-'); title('sin(3*x) * cos(5*x)')
```



Hình 4.7.

✎ Bài tập 4-5.

Vẽ đồ thị các hàm $f(x) = x$, $g(x) = x^3$, $h(x) = e^x$ và $z(x) = e^{x^2}$ với $x \in [0, 4]$ trên cùng một cửa sổ hình và với hệ trục có độ chia bình thường và hệ trục tọa độ có độ chia log-log. Lấy mẫu thích hợp để làm phẳng đồ thị. Mô tả mỗi đồ thị bằng các hàm: **xlabel**, **ylabel**, **title**, **legend**.

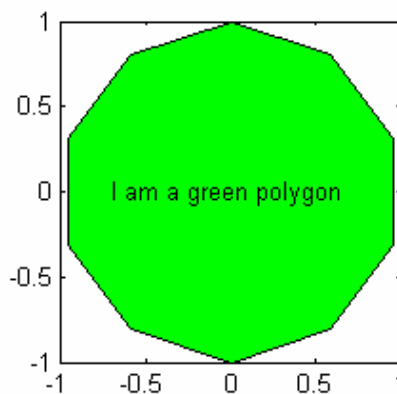
✎ Bài tập 4-6.

Vẽ đồ thị của các hàm $f(x) = \sin(1/x)$, $f(x) = \cos(1/x)$ trên cùng một cửa sổ hình, với $x \in [0.01, 0.1]$. Lấy mẫu thích hợp để các đường cong rõ ràng nhất.

4.3. CÁC THUỘC TÍNH KHÁC CỦA ĐƯỜNG CONG 2D

MATLAB có một số hàm được thiết kế đặc biệt để sử dụng với những hình 2D, ví dụ như các hàm: **fill**, **polar**, **bar**, **barh**, **pie**, **hist**, **errorbar** hay **stem**. Trong ví dụ sau, lệnh **fill** được sử dụng để tạo ra một hình đa giác:

```
>> N = 5; k = -N:N;
>> x = sin(k*pi/N);
>> y = cos(k*pi/N); % (x, y) là các đỉnh của đa giác
>> fill(x,y,'g')
>> axis square
>> text(-0.7,0,'I am a green polygon')
```



Hình 4.8.

✎ Bài tập 4-7.

Thực thi các lệnh sau và mô tả kết quả (*lưu ý rằng lệnh **figure** sẽ tạo ra một cửa sổ hình mới*):

```
>> figure % vẽ đồ thị cột của một đường cong hình
chuông.
>> x = -2.9:0.2:2.9;
>> bar(x,exp(-x.*x));
>> figure % vẽ sóng hình sin ở dạng bậc thang
>> x = 0:0.25:10;
>> stairs(x,sin(x));
>> figure % vẽ errorbar
>> x = -2:0.1:2;
>> y = erf(x); % dùng lệnh help để hiểu thêm
```

```
>> e = rand(size(x)) / 10;
>> errorbar (x,y,e);
>> figure
>> r = rand(5,3);
>> subplot(1,2,1); bar(r,'grouped')
>> subplot(1,2,2); bar(r,'stacked')
>> figure
>> x = randn(200,1); % số ngẫu nhiên của phân bố bình thường
>> hist(x,15) % biểu đồ với 15 cột
```

4.4. IN ẮN

Trước khi tiến hành in để miêu tả một cách rõ ràng hơn chúng ta có thể thêm một số thông tin vào đồ thị, chẳng hạn như tựa đề, hay thay đổi cách trình bày. **Bảng 4.2** trình bày một số lệnh được sử dụng để trình bày hình vẽ hay đồ thị.

Bảng 4.2: Một số lệnh thao tác với đồ thị

Lệnh	Kết quả
<code>grid on/off</code>	Cộng lưới vào đồ thị
<code>axis([xmin xmax ymin ymax])</code>	Xác định giá trị lớn nhất và nhỏ nhất của các trục
<code>box off/on</code>	Xóa/hiển thị đường viền khung của đồ thị.
<code>xlabel('text')</code>	Nhãn của trục x
<code>ylabel('text')</code>	Nhãn của trục y
<code>title('text')</code>	Tựa đề ở trên đồ thị.
<code>text(x,y,'text')</code>	Cộng dòng ký tự vào điểm (x,y)
<code>gtext('text')</code>	Cộng dòng ký tự vào vị trí xác định bởi chuột.
<code>legend('fun1','fun2')</code>	Cộng vào tên của các hàm.
<code>legend off</code>	Xóa tên của các hàm.
<code>clf</code>	Xóa cửa sổ hình hiện tại.
<code>subplot</code>	Tạo ra những cửa sổ hình con.

✎ Bài tập 4-8.

Vẽ đồ thị các hàm $y_1 = \sin(4x)$, $y_2 = x\cos(x)$, $y_3 = (x + 1)^{-1}x^{1/2}$ với $x = 1:0.25:10$, và một điểm $(x; y) = (4; 5)$ trên cùng một cửa sổ hình. Sử dụng các màu và kiểu khác nhau cho các đồ thị. Cộng thêm vào lời chú thích, nhãn cho cả hai trục và một tựa đề. Cộng một đoạn text ‘single point’ đến vị trí (4; 5). Thay đổi giá trị lớn nhất và nhỏ nhất của các trục sao cho có thể quan sát hàm y_3 một cách chi tiết nhất.

Cách đơn giản nhất để tiến hành in là chọn **File** trên thanh công cụ và sau đó chọn **Print**. Lệnh **print** được sử dụng để gửi một đồ thị đến máy in hoặc lưu vào một file. Do có nhiều thông số không được giải thích chi tiết ở đây, sử dụng lệnh **help** để hiểu rõ hơn. Cố gắng hiểu các ví dụ sau đây.

```
>> print -dwinc
      % sử dụng máy in hiện hành để in màu hình đang hiển thị
>> print -fl -deps myfile.eps
      % lưu hình Figure no.1 đến file myfile.eps ở dạng đen
      trắng.
>> print -fl -depssc myfilec.eps
      % lưu hình Figure no.1 đến file myfilec.eps ở dạng màu.
>> print -dtiff myfile1.tiff
      % lưu hình hiện hành đến file myfile1.tiff
>> print -dpssc myfile1c.ps
      % lưu hình hiện hành đến file myfile1.ps ở dạng màu
>> print -f2 -djpeg myfile2
      % lưu hình Figure no.2 đến file myfile2.jpg
```

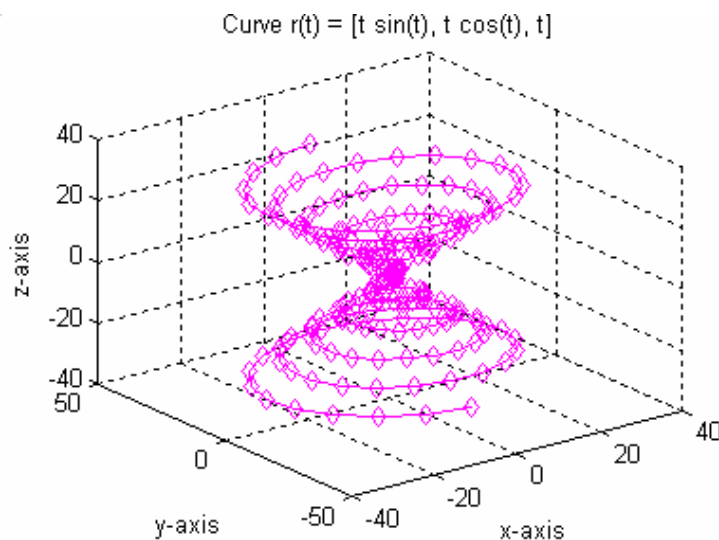
✎ Bài tập 4-9.

Trong những bài tập ở trước, thực hành lưu các hình đồ thị đến một file được chỉ định.

4.5. ĐỒ THỊ 3D

Lệnh **plot3** được sử dụng để vẽ đồ thị 3D, lệnh này tương đương với lệnh **plot** trong 2D. Về hình thức nó giống như lệnh **plot** tuy nhiên lệnh **plot3** được mở rộng thêm cho một trục tọa độ thứ 3. Ví dụ vẽ đường cong r được định nghĩa $r(t) = [t \sin(t); t \cos(t); t]$ với t nằm trong khoảng $[-10\pi; 10\pi]$.

```
>> t = linspace(-10*pi,10*pi,200);
>> plot3(t.*sin(t), t.*cos(t), t, 'md-'); % vẽ đường cong với màu
hồng
>> title('Curve r(t) = [t sin(t), t cos(t), t]');
>> xlabel('x-axis'); ylabel('y-axis'); zlabel('z-axis');
>> grid
```

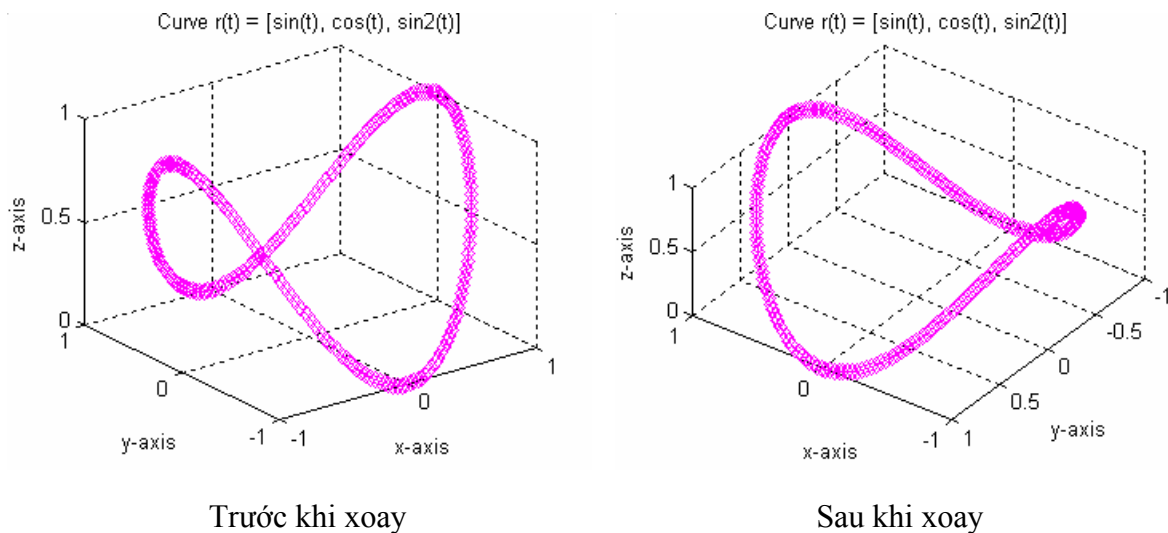


Hình 4.9.

✎ Bài tập 4-10.

Vẽ một đồ thị 3D được định nghĩa $[x(t); y(t); z(t)] = [\sin(t); \cos(t); \sin^2(t)]$ với $t = [0, 2\pi]$. Vẽ đường ở trên bằng màu xanh và các điểm đánh dấu là các vòng tròn. Cộng thêm tựa đề của đồ thị, mô tả các trục và vẽ các đường kẻ.

Chúng ta có thể xoay hình 3D vừa vẽ bằng cách chọn mục **Tools** trên thanh công cụ của cửa sổ hình và chọn mục **Rotate 3D** hay bằng cách nhập vào lệnh **rotate3D**. Sau đó nhấn chuột vào hình, giữ và bắt đầu xoay theo hướng mình mong muốn.



Hình 4.10.

4.6. BỀ MẶT ĐỒ THỊ

Một bề mặt được định nghĩa bởi một hàm $f(x, y)$, với mỗi cặp giá trị của (x, y) , độ cao z được tính $z = f(x, y)$. Để vẽ một bề mặt, vùng lấy mẫu (x, y) nên là một vùng hình chữ nhật. Hệ thống các giá trị x và y được tạo ra bởi lệnh **meshgrid** như sau:

```
>> [X, Y] = meshgrid (-1:.5:1, 0:.5:2)
```

X =

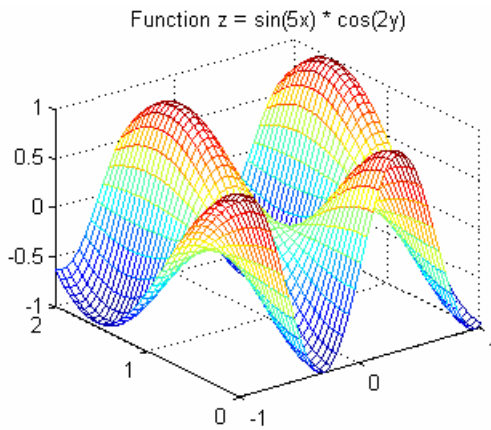
-1.0000	-0.5000	0	0.5000	1.0000
-1.0000	-0.5000	0	0.5000	1.0000
-1.0000	-0.5000	0	0.5000	1.0000
-1.0000	-0.5000	0	0.5000	1.0000
-1.0000	-0.5000	0	0.5000	1.0000

Y =

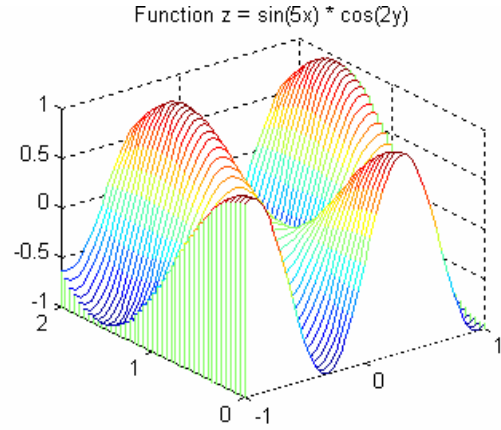
0	0	0	0	0
0.5000	0.5000	0.5000	0.5000	0.5000
1.0000	1.0000	1.0000	1.0000	1.0000
1.5000	1.5000	1.5000	1.5000	1.5000
2.0000	2.0000	2.0000	2.0000	2.0000

Vùng lấy mẫu trong trường hợp này là $[-1, 1] \times [0, 2]$ và khoảng lấy mẫu là 0.5. Để bề mặt đồ thị phẳng hơn ta nên lấy mẫu dày đặc hơn.

```
>> [X,Y] = meshgrid(-1:.05:1, 0:.05:2);
>> Z = sin(5*X) .* cos(2*Y);
>> mesh(X,Y,Z);
>> title('Function z = sin(5x) * cos(2y)')
```



Sử dụng lệnh **mesh**



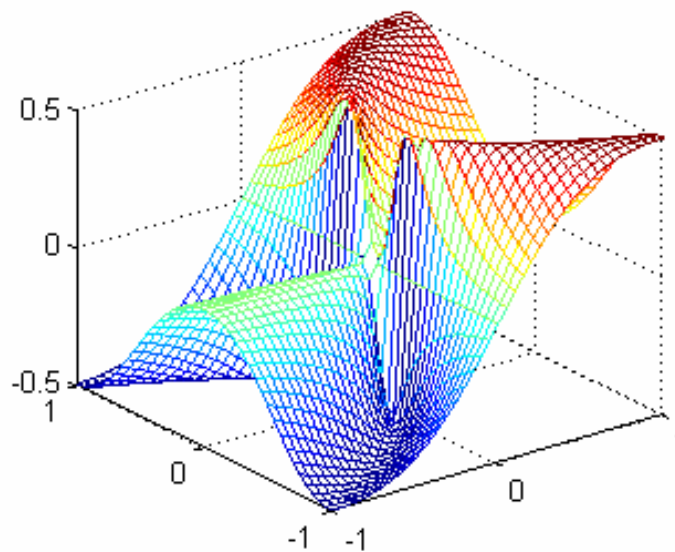
Sử dụng lệnh **waterfall**

Hình 4.11

Thử thay lệnh **mesh** bằng lệnh **waterfall** và nhận xét sự khác nhau giữa hai lệnh.

✎ Bài tập 4-11.

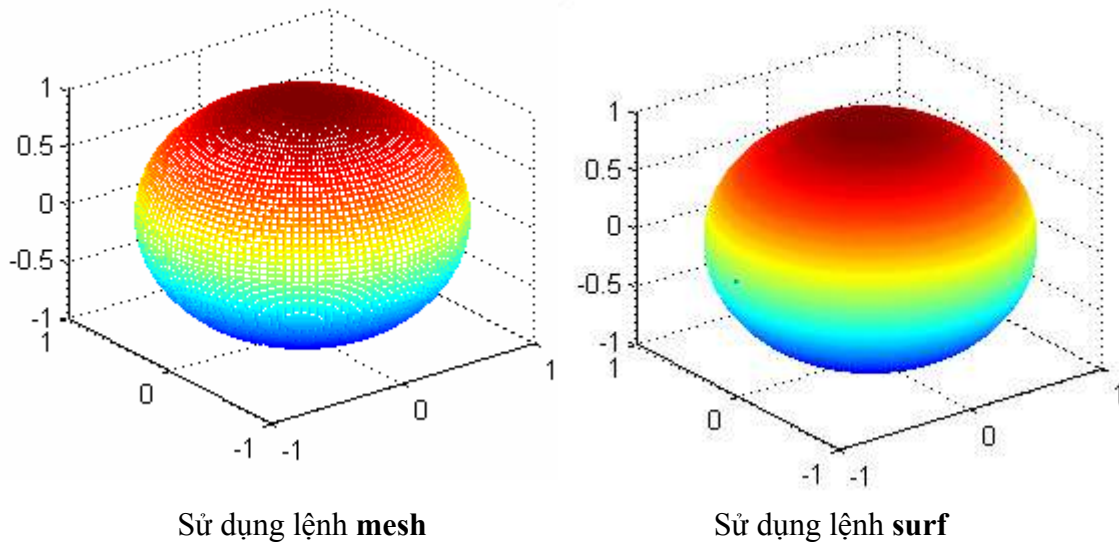
Vẽ đồ thị mặt của hàm $f(x,y) = \frac{xy^2}{x^2 + y^4}$ tại vùng gần điểm $(0, 0)$. Lưu ý nên sử dụng mật độ lấy mẫu dày đặc.



Hình 4.12.

✎ **Bài tập 4-12.**

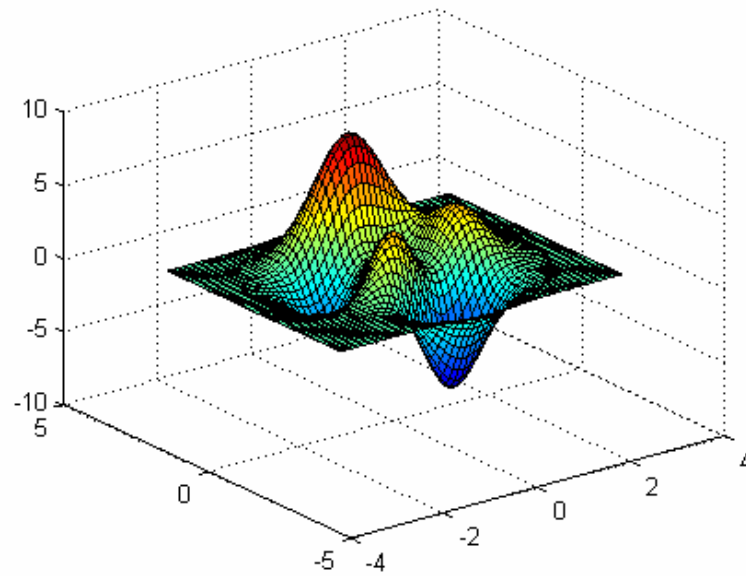
Vẽ một quả cầu với các thông số được định nghĩa $[x(t, s), y(t, s), z(t, s)] = [\cos(t)\cos(s), \cos(t)\sin(s), \sin(t)]$, với $t, s = [0, 2\pi]$ (sử dụng lệnh **surf**). Sử dụng lệnh **shading interp** để xóa các đường màu đen, sau đó sử dụng lệnh **shading faceted** để phục hồi lại hình nguyên thủy.

**Hình 4.13**✎ **Bài tập 4-13.**

Vẽ hàm theo r và θ : $[x(r, \theta); y(r, \theta); z(r, \theta)] = [r\cos(\theta); r\sin(\theta); \sin(6\cos(r)-n\theta)]$. Chọn n là một hằng số. Quan sát sự thay đổi của hình theo n .

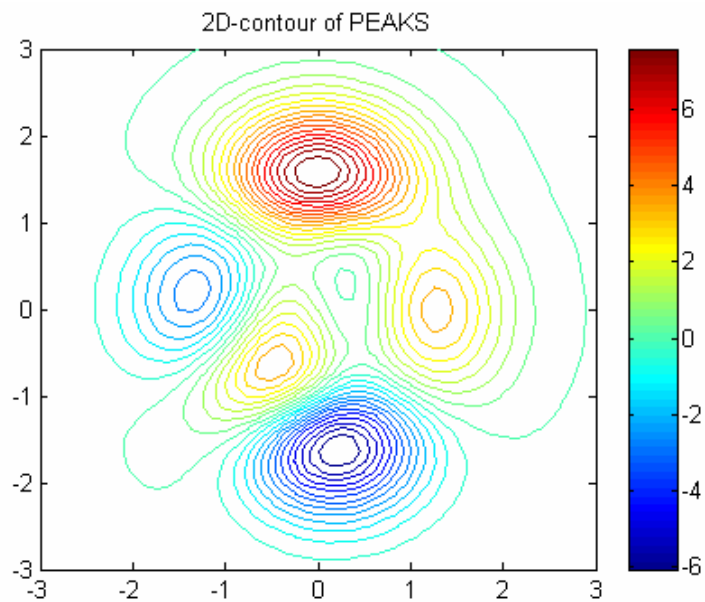
Trong MATLAB, hàm **peaks** là một hàm hai biến, có được từ phép biến đổi phân bố Gauss. Công dụng của hàm **peaks** là tạo ra các biến giá trị để sử dụng trong các hàm 3D như: **mesh**, **surf**, **pcolor**, **contour**, Ví dụ:

```
>> [X,Y,Z] = peaks;    % tạo các giá trị để vẽ,
                        % X, Y, Z là các ma trận 49x49
>> surf(X,Y,Z);       % vẽ bề mặt
```



Hình 4.14

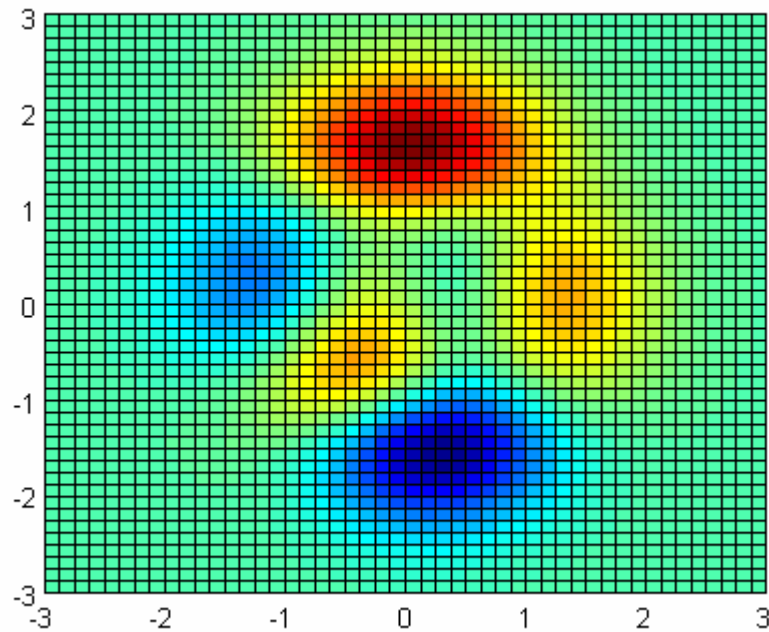
```
>> figure
>> contour (X,Y,Z,30); % vẽ đường viền trong 2D
>> colorbar          % thêm vào trục z một thanh màu tương ứng
với
                        % giá trị của Z.
>> title('2D-contour of PEAKS');
```



Hình 4.15

```
>> figure
>> contour3(X,Y,Z,30); % vẽ đường viền trong 3D
>> title('3D-contour of PEAKS');
```

```
>> pcolor(X,Y,Z);      % giá trị z được thể hiện với giá trị
                        % màu tương ứng, xem thêm lệnh colorbar
```

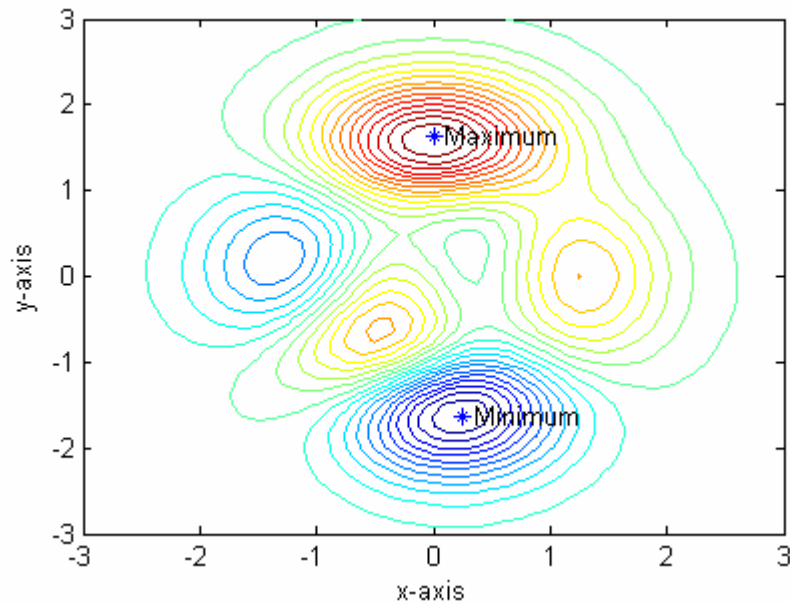


Hình 4.16

Lệnh **close all** được sử dụng để đóng tất cả các cửa sổ hình, để đóng từng cửa sổ ta có thể sử dụng lệnh **close** (**close 1** để đóng cửa sổ hình **Figure No.1**). Đoạn chương trình sau đây dùng để xác định vị trí có giá trị nhỏ nhất của đồ thị 3D.

```
>> [mm,I] = min(Z);      % mm là vector hàng chứa các giá trị nhỏ nhất.
                        % I chứa các vị trí có giá trị nhỏ nhất.
>> [Zmin, j] = min (mm); % Zmin là giá trị nhỏ nhất và j là vị trí
                        % tương ứng. Zmin mang giá trị của
Z(I(j),j)
>> xpos = X(I(j),j);    % giá trị tương ứng của X.
>> ypos = Y(I(j),j);    % giá trị tương ứng của Y.
>> contour (X,Y,Z,25);
>> xlabel('x-axis'); ylabel('y-axis');
>> hold on
>> plot(xpos,ypos,'*');
>> text(xpos+0.1,ypos,'Minimum');
```

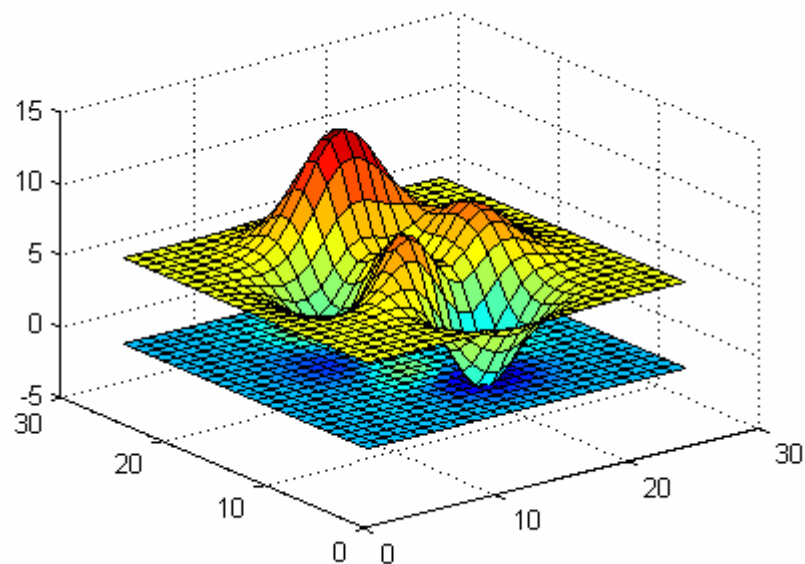
Thực hiện tương tự để xác định được vị trí có giá trị lớn nhất:



Hình 4.17

Chúng ta cũng có thể kết hợp hai hay nhiều đồ thị vào trong cùng một hình.

```
>> surf(peaks(25)+6);           % dịch đồ thị theo trục z +6 đơn vị.
>> hold on
>> pcolor(peaks(25));
>> hold off
```



Hình 4.18

Bài tập 4-14.

Vẽ bề mặt của hàm $f(x,y) = xye^{-x^2-y^2}$ với vùng lấy mẫu $[-2, 2] \times [-2, 2]$. Tìm vị trí, giá trị lớn nhất và nhỏ nhất của hàm.

4.7. HÌNH ĐỘNG

Trong MATLAB chúng ta có thể tạo ra hình chuyển động bằng cách tạo ra một chuỗi các hình. Để hiểu cách tạo ra một hình động, phân tích đoạn chương trình sau đây với đồ thị của hàm $f(x) = \sin(nx)$ trong đó $x = [0, 2\pi]$ và $n = 1, \dots, 5$.

```
N = 5;
M = moviein(N);
x = linspace(0, 2*pi);
for n=1:N
    plot(x, cos(n*x), 'r-');
    xlabel('x-axis')
    if n > 1,
        ss = strcat('cos(', num2str(n), 'x)');
    else
        ss = 'cos(x)';
    end
    ylabel(ss)
    title('Cosine functions cos(nx)', 'FontSize', 12)
    axis tight % giới hạn các trục theo giá trị.
    grid
    M(:,n) = getframe;
    pause(1.8)
end
movie(M) % chạy đoạn film.
```

Các lệnh được tô đậm ở trên là các thành phần chính để tạo nên một đoạn hình chuyển động trong MATLAB.

Trong đoạn chương trình ở trên, cấu trúc vòng lặp **for** được sử dụng để tạo ra các khung ảnh cho đoạn film. Chúng ta sẽ đề cập đến cấu trúc vòng lặp một cách chi tiết hơn ở phần sau. Lệnh **strcat** dùng để nối các chuỗi lại với nhau, sử dụng lệnh **help strcat** để hiểu rõ hơn.

Khi chạy đoạn film, đầu tiên 5 frame được hiển thị và sau đó 5 frame xuất hiện lại với tốc độ nhanh hơn. Lệnh **moviein(M)** báo với MATLAB rằng có một đoạn film gồm có M frame sẽ được tạo ra. Các frame liên tiếp nhau sẽ được tạo ra bên trong vòng lặp. Với lệnh **getframe**, các frame sẽ được lưu vào các cột của ma trận M. Lệnh **movie(M)** sẽ chạy đoạn film vừa tạo.

*Lưu ý rằng việc tạo ra đoạn film yêu cầu khá nhiều bộ nhớ, do vậy khi kết thúc công việc nên **clear M** ra khỏi workspace.*

✎ Bài tập 4-15.

Viết chương trình tạo một đoạn film gồm có 5 frame là bề mặt của hàm $f(x; y) = \sin(nx)\sin(ny)$ với $x, y = [0; 2\pi]$ và $n = 1 : 5$. Thêm vào đoạn film tựa đề, miêu tả của các trục.

Danh sách các lệnh và hàm được giới thiệu trong chương 4

Các hàm vẽ đồ thị 2D

bar	Vẽ biểu đồ cột
barh	Vẽ biểu đồ cột nằm ngang
errorbar	Vẽ đồ thị lỗi
fill	Tô màu một đa giác 2D
hist	Vẽ biểu đồ histogram
loglog	Vẽ đồ thị trên hệ trục logarithm
pie	Vẽ biểu đồ phần trăm
plot	Vẽ đồ thị trên hệ trục tuyến tính
polar	Vẽ đồ thị trên hệ tọa độ cực
semilogx	Vẽ đồ thị trên hệ trục có trục hoành chia theo thang logarithm
semilogy	Vẽ đồ thị trên hệ trục có trục tung chia theo thang logarithm
stem	Vẽ đồ thị xung

Các hàm phục vụ cho đồ thị 2D

axis	Điều chỉnh các giới hạn của các trục tọa độ
box off/on	Xóa/hiển thị đường viền khung của đồ thị.
clf	Xóa cửa sổ hình hiện tại
close	Đóng một hay nhiều figure đang mở
figure	Tạo một figure mới để vẽ đồ thị
grid	Hiển thị hoặc không hiển thị lưới ô vuông
gtext	Thêm dòng ký tự tại điểm xác định bởi con trỏ chuột
hold on	Cho phép vẽ đồ thị khác lên cùng hệ trục tọa độ hiện hành
hold off	Xóa các đồ thị cũ trước khi vẽ đồ thị mới
legend	Hiển thị các chú thích trên đồ thị
legend off	Xóa các chú thích trên đồ thị
subplot	Chia figure thành nhiều phần, mỗi phần vẽ một đồ thị
text	Thêm dòng ký tự tại một điểm xác định trên đồ thị
title	Hiển thị tiêu đề của đồ thị
xlabel	Hiển thị tên của trục X
ylabel	Hiển thị tên của trục Y

Các hàm vẽ đồ thị 3D

contour	Vẽ đồ thị đường đồng mức
mesh	Vẽ đồ thị 3D với bề mặt dạng lưới

meshgrid	Tạo lưới các điểm (miền xác định) để vẽ đồ thị
pcolor	Vẽ bảng màu từ một ma trận các tọa độ
peaks	Một hàm hai biến dùng để minh họa cho các đồ thị 3D
plot3	Vẽ đồ thị 3D
rotate3D on/off	Cho phép / không cho phép xoay đồ thị 3D bằng chuột
shading	Chọn chế độ phủ màu
surf	Vẽ đồ thị 3D bằng cách tô màu
waterfall	Vẽ đồ thị 3D dạng waterfall

Các hàm liên quan đến hình động

getframe	Lưu các frame hình động vào một ma trận
moviein	Khởi động bộ nhớ chứa các frame (khung) hình động
movie	Hiển thị đoạn ảnh động

Các lệnh và hàm khác

for ... end	Thực hiện vòng lặp có số lần lặp hữu hạn
strcat	Nối hai hay nhiều chuỗi với nhau

Chương 5

BIỂU THỨC RẼ NHÁNH

Cấu trúc của biểu thức rẽ nhánh bao gồm một nhóm lệnh cho phép thực thi một đoạn chương trình có điều kiện hay thực hiện một vòng lặp.

5.1. CÁC TOÁN TỬ LOGIC VÀ BIỂU THỨC QUAN HỆ

Để thực thi biểu thức rẽ nhánh, chương trình cần thực hiện những phép toán mà kết quả của nó là ĐÚNG hay SAI (TRUE hay FALSE). Trong MATLAB, kết quả trả về của một toán tử logic là 1 nếu nó đúng và kết quả là 0 nếu nó sai. Bảng 5.1 trình bày những toán tử logic và những biểu thức quan hệ trong MATLAB. Sử dụng lệnh **help relop** để hiểu rõ hơn về toán tử logic và các biểu thức quan hệ. Những toán tử quan hệ $<$, $<=$, $>$, $>=$, $==$ và $\sim$ có thể được sử dụng để so sánh hai ma trận có cùng kích thước (*lưu ý rằng một số thực được xem như là một ma trận 1x1*). Hơn nữa, các hàm **xor**, **any**, và **all** cũng được sử dụng như là các toán tử logic (sử dụng lệnh **help** để hiểu rõ hơn)

Bảng 5.1: Toán tử logic và biểu thức quan hệ.

Lệnh	Kết quả
$a = (b > c)$	a là 1 nếu b lớn hơn c. Tương tự với $<$, $>=$, $<=$
$a = (b == c)$	a là 1 nếu b bằng c
$a = (b \sim c)$	a là 1 nếu b không bằng c
$a = \sim b$	Logic bù: a bằng 1 nếu b bằng 0
$a = (b \& c)$	Logic AND: a là 1 nếu a = TRUE và b = TRUE.
$a = (b c)$	Logic OR: a là 1 nếu a = TRUE hoặc b = TRUE.

Lưu ý rằng trong MATLAB các toán tử AND và OR có mức ưu tiên bằng nhau, điều này có nghĩa là các toán tử này sẽ được thực thi từ trái sang phải. Ví dụ:

```
>> b = 10;
>> 1 | b > 0 & 0      % '|' là logic OR, '&' là logic AND.
ans =
    0
>> (1 | b > 0) & 0     % biểu thức này tương đương với biểu thức ở
trên.
ans =
    0
>> 1 | (b > 0 & 0)
ans =
    1
```

Lưu ý rằng để tránh các trường hợp nhầm lẫn ta nên sử dụng dấu ngoặc để phân biệt các biểu thức logic có cấp độ ưu tiên khác nhau.

✎ Bài tập 5-1.

- Dự đoán trước kết quả và sau đó kiểm tra lại bằng MATLAB các biểu thức quan hệ trong bảng 5.1 với $b = 0$ và $c = -1$.
- Dự đoán trước kết quả và sau đó kiểm tra lại bằng MATLAB các toán tử logic (and, or) với $b = [2, 31, -40, 0]$ và $c = 0$.
- Định nghĩa hai vector ngẫu nhiên (**randn(1,7)**) và thực thi tất cả các toán tử logic bao gồm cả **xor**, **any**, và **all**.

✎ Bài tập 5-2.

1. Cho $x = [1, 5, 2, 8, 9, 0, 1]$ và $y = [5, 2, 2, 6, 0, 0, 2]$. Thực thi và giải thích kết quả của các lệnh sau:

- | | | |
|------------|--------------|--------------------------|
| • $x > y$ | • $x \leq y$ | • $x \& (\sim y)$ |
| • $x < y$ | • $y \geq x$ | • $(x > y) \mid (y < x)$ |
| • $x == y$ | • $x \mid y$ | • $(x > y) \& (y < x)$ |

2. Cho $x = 1 : 10$ và $y = [3, 5, 6, 1, 8, 2, 9, 4, 0, 7]$. Thực thi và giải thích kết quả của các lệnh sau:

- | | |
|------------------------|---------------------------------|
| • $(x > 3) \& (x < 8)$ | • $x ((x < 2) \mid (x \geq 8))$ |
| • $x(x > 5)$ | • $y ((x < 2) \mid (x \geq 8))$ |
| • $y(x \leq 4)$ | • $x (y < 0)$ |

✎ Bài tập 5-3.

Cho $x = [3, 16, 9, 12, -1, 0, -12, 9, 6, 1]$. Thực thi các công việc sau:

- Gán các thành phần giá trị dương của x bằng zero.
- Nhân 2 các giá trị là bội số của 3.
- Nhân các giá trị lẻ của x với 5.
- Trích các giá trị của x lớn hơn 10 thành vector y .
- Gán các giá trị của x nhỏ hơn giá trị trung bình của nó thành zero.

✎ Bài tập 5-4.

Thực thi các lệnh sau đây và lưu ý giá trị của z .

```
>> hold on
>> x = -3:0.05:3;
>> y = sin(3*x);
>> subplot(1, 2, 1);
>> plot(x, y);
>> axis tight
>> z = (y < 0.5).*y;
>> subplot(1, 2, 2);
>> plot(x, y, 'r:');
>> plot(x, z, 'r');
```

```
>> axis tight
>> hold off
```

Trước khi tiếp tục, chắc chắn rằng bạn đã hiểu được ý nghĩa và hoạt động các biểu thức sau:

```
>> a = randperm(10);      % hoán vị ngẫu nhiên.
>> b = 1:10;
>> b - (a <= 7)           % (b - 1) nếu (a <= 7) và (b - 0) nếu (a > 7)
>> (a >= 2) & (a < 4)     % bằng 1 nếu 2 <= a < 4
>> ~(b > 4)              % bằng 1 nếu b <= 4
>> (a == b) | b == 3      % bằng 1 nếu a = b hoặc b = 3
>> any(a > 5)             % bằng 1 nếu bất kỳ thành phần nào
                           % của a lớn hơn 5.
>> any(b < 5 & a > 8)     % bằng 1 nếu tồn tại a và b sao cho
                           % (b < 5 và a > 8)
>> all(b > 2)            % bằng 1 nếu tất cả các giá trị
                           % của b lớn hơn 2.
```

Các thành phần của vector hay ma trận có thể được trích ra khi chúng thỏa mãn một điều kiện nào đó. Sử dụng lệnh **find** cũng cho ta một kết quả tương tự. Kết quả trả về của lệnh **find** là vị trí của các thành phần thỏa mãn điều kiện. Ví dụ:

```
>> x = [1 1 3 4 1];
>> i = (x == 1)
i =
     1     1     0     0     1
>> y = x(i)
y =
     1     1     1
>> j = find(x == 1)      % j là các vị trí của x thỏa mãn x(j) = 1.
j =
     1     2     5
>> z = x(j)
z =
     1     1     1
```

Một ví dụ khác là:

```
>> x = -1:0.05:1;
>> y = sin(x) .* sin(3*pi*x);
>> plot (x, y, '-');
>> hold on
>> k = find (y <= -0.1)
k =
```

```

          9      10      11      12      13      29      30      31      32      33
>> plot (x(k), y(k), 'ro');
>> r = find (x > 0.5 & y > 0)
r =
        35      36      37      38      39      40      41
>> plot (x(r), y(r), 'r*');

```

Với ma trận, lệnh **find** cũng được thực thi theo cách tương tự.

```

>> A = [1  3  -3  -5; -1  2  -1  0; 3  -7  2  7];
>> k = find (A >= 2.5)
k =
        3
        4
       12
>> A(k)
ans =
        3
        3
        7

```

Theo cách này, đầu tiên lệnh **find** sắp xếp ma trận A lại thành một vector cột theo cách cột sau được nối tiếp vào cột trước nó. Do vậy, k là vị trí các thành phần của vector vừa tạo ra có giá trị lớn hơn hoặc bằng 2,5 và A(k) là giá trị của các thành phần này. Các giá trị cột và dòng của các thành phần này cũng có thể được xác định:

```

>> [I, J] = find (A >= 2.5)
I =
        3
        1
        3
J =
        1
        2
        4
>> [A(I(1), J(1)), A(I(2), J(2)), A(I(3), J(3))]
ans =
        3        3        7

```

Bài tập 5-5.

Cho $A = \text{ceil}(5 * \text{randn}(6, 6))$

- Tìm các thành phần của A nhỏ hơn -3.
- Tìm các thành phần của A lớn hơn -1 và nhỏ hơn 5.

- Loại bỏ các cột có chứa thành phần 0.

5.2. BIỂU THỨC ĐIỀU KIỆN

Cấu trúc điều kiện **if** được sử dụng để quyết định nhóm lệnh nào sẽ được thực thi. Dưới đây là mô tả tổng quát của các lệnh điều kiện. Trong những ví dụ này, lệnh **disp** được sử dụng thường xuyên. Lệnh này in ra màn hình đoạn văn bản ở giữa hai dấu nháy.

- Cấu trúc **if ... end**

Cú pháp
if <i>biểu thức logic</i> lệnh thứ 1 <i>lệnh thứ 1</i> ... end

Ví dụ
if (a > 0) b = a; disp ('a is positive'); end

- Cấu trúc **if ... else ... end**

Cú pháp
if <i>biểu thức logic</i> <i>các lệnh được thực thi khi biểu thức logic TRUE</i> else <i>các lệnh được thực thi khi biểu thức logic FALSE</i> end

Ví dụ
if (temperature > 100) disp ('Above boiling.');
toohigh = 1; else disp ('Temperature is OK.');
toohigh = 0; end

- Cấu trúc **if ... elseif ... else ... end**

Cú pháp
if <i>biểu thức logic 1</i> <i>các lệnh được thực thi khi biểu thức logic 1 TRUE</i> elseif <i>biểu thức logic 2</i> <i>các lệnh được thực thi khi biểu thức logic 2 TRUE</i> else <i>các lệnh được thực thi khi không có biểu thức logic nào TRUE</i> end

Ví dụ
if (height > 190) disp ('very tall');
elseif (height > 170) disp ('tall');
elseif (height < 150) disp ('small');
else disp ('average');
end

Lưu ý: Chúng ta có thể sử dụng một m-file để thực thi các ví dụ ở trên. m-file là một tập tin chứa một chuỗi các lệnh trong MATLAB. Để tạo ra một m-file, đầu tiên chúng ta mở một chương trình soạn thảo sau đó nhập vào tất cả các lệnh cần thiết (không có dấu nhắc '>>' ở đầu mỗi lệnh). Cuối cùng lưu file này với phần mở rộng ".m", ví dụ mytask.m. Để thực thi m-file vừa tạo, từ cửa sổ lệnh của MATLAB chúng ta gõ vào lệnh **mytask**. Một cách khác để tạo ra m-file là sử dụng chương trình soạn thảo có sẵn của MATLAB, chọn **File** trên thanh công cụ, sau đó chọn **New** và cuối cùng chọn **m-file**. Một cửa sổ mới sẽ xuất hiện, trong cửa sổ này chúng ta có thể nhập vào đoạn chương trình cần thiết và lưu chúng vào trong ổ cứng (chương trình sẽ tự động lưu file này với phần mở rộng .m). Tất cả các lệnh trong tập tin này sẽ được thực thi một cách tuần tự trong MATLAB. Để m-file có thể được thực thi thì chúng phải được lưu vào những thư mục mà MATLAB có thể "thấy" được, đây chính là các thư mục nằm trong danh sách đường dẫn của MATLAB. Chúng ta sẽ đề cập một tiết hơn về m-file trong những phần sau.

 **Ví dụ 5-1.** Tạo một m-file short.m chứa hai dòng lệnh sau đây và thực thi chúng bằng lệnh **short**.

```
x = 0:0.5:4;
plot(x, sin(x), '*-');
```

Bài tập 5-6.

Hãy xác định các giá trị trả về của mỗi đoạn chương trình dưới đây, (lưu ý rằng không phải tất cả các lệnh đều đúng).

- | | | |
|----------------------------|------------|-------|
| 1. if n > 1 | a) n = 7 | m = ? |
| m = n + 2 | b) n = 0 | m = ? |
| else | c) n = -7 | m = ? |
| m = n - 2 | | |
| end | | |
| 2. if s <= 1 | a) s = 1 | t = ? |
| t = 2z | b) s = 7 | t = ? |
| elseif s < 10 | c) s = 57 | t = ? |
| t = 9 - z | d) s = 300 | t = ? |
| elseif s < 100 | | |
| t = sqrt(s) | | |
| else | | |
| t = s | | |
| end | | |
| 3. if t >= 24 | a) t = 50 | h = ? |
| z = 3t + 1 | b) t = 19 | h = ? |
| elseif t < 9 | c) t = -6 | h = ? |
| z = t ² /3 - 2t | d) t = 0 | h = ? |
| else | | |

```

z = -t
end
4. if 0 < x < 7
    y = 4x
elseif 7 < x < 55
    y = -10x
else
    y = 333
end

```

a) $x = -1$ $y = ?$
b) $x = 5$ $y = ?$
c) $x = 30$ $y = ?$
d) $x = 56$ $y = ?$

✎ Bài tập 5-7.

Tạo một đoạn chương trình nhập vào giá trị N và tính toán để trả về giá trị C (sử dụng cấu trúc **if ... elseif ...**). Sử dụng lệnh **input** để yêu cầu nhập vào một giá trị (sử dụng lệnh **help input** để hiểu rõ hơn cách sử dụng của lệnh **input**).

$$C = \begin{cases} 0, & N \leq 0 \\ 24 / N, & N \in (0, 0.1] \\ 24 / N(1 + 0.14N^{0.7}), & N \in (0.1, 1e3] \\ 0.43, & N \in (1e3, 5e5] \\ 0.19 - 8e4 / N, & N > 5e5 \end{cases}$$

Sử dụng đoạn chương trình trên để tính C với $N = -3e3, 0.01, 56, 1e3, 3e6$ (lưu ý rằng $3e3$ được thể hiện trong MATLAB là $3*10^3$).

✎ Bài tập 5-8.

Viết một đoạn chương trình nhập vào một số nguyên và kiểm tra nó có chia hết cho 2 hay 3 không. Thực hiện với tất cả các trường hợp có thể: chia hết cho cả 2 và 3, chia hết cho 2 nhưng không chia hết cho 3, ... (sử dụng lệnh **rem**).

Cấu trúc **switch** cũng thường được sử dụng trong biểu thức điều kiện, **switch** chuyển đổi giữa các trường hợp phụ thuộc vào giá trị của một biểu thức, biểu thức này có thể là số hay là chuỗi.

Cú pháp	Ví dụ
<pre> switch biểu thức case choice1 khối lệnh thứ 1 case choice2 khối lệnh thứ 2 ... otherwise khối lệnh end </pre>	<pre> method = 2; switch method case {1,2} disp('Method is linear. '); case 3 disp('Method is cubic. '); case 4 disp('Method is nearest. '); otherwise disp('Unknown method. '); end </pre>

Những phát biểu trong lệnh **case** đầu tiên có biểu thức đúng sẽ được chọn để thực thi. Biểu thức có thể là một số hay là một chuỗi ký tự. Trong trường hợp là số, nhóm lệnh được chọn nếu **biểu thức = choice**. Còn trong trường hợp là chuỗi biểu thức được chọn khi **strcmp(biểu thức, choice)** trả về giá trị 1 (đúng) (lệnh **strcmp** được sử dụng để so sánh hai chuỗi với nhau).

Lưu ý rằng cấu trúc **switch** chỉ cho phép thực thi duy nhất một nhóm lệnh.

✎ Bài tập 5-9.

Giả sử rằng biến **tháng** có giá trị từ 1 đến 12. Sử dụng cấu trúc **switch**, viết một đoạn chương trình nhập vào giá trị của biến **tháng**, kết quả trả về là số ngày trong tháng đó và tên tháng ('November', 'October', 'December', ...).

5.3. VÒNG LẶP

Cấu trúc vòng lặp sẽ lặp lại một khối các phát biểu cho đến khi một số điều kiện không còn thỏa mãn. Có hai kiểu cấu trúc lặp phổ biến là **for** và **while**.

- Vòng lặp **for** sẽ thực hiện một nhóm các phát biểu trong một số lần xác định.

Cú pháp
for <i>index</i> = <i>first</i> : <i>step</i> : <i>last</i> <i>nhóm các phát biểu</i> end

Ví dụ
<i>sumx</i> = 0; for <i>i</i> = 1:length(<i>x</i>) <i>sumx</i> = <i>sumx</i> + <i>x</i> (<i>i</i>); end

Trong cấu trúc vòng lặp **for**, **step** có thể là số âm, **index** có thể là một vector. Xem các ví dụ sau:

Ví dụ 1
for <i>i</i> = 1 : 2 : <i>n</i> end

Ví dụ 2
for <i>i</i> = <i>n</i> : -1 : 3 end

Ví dụ 3
for <i>i</i> = 0 : 0.5 : 4 disp(<i>x</i> ²); end

Ví dụ 4
for <i>i</i> = [25 9 8] disp(sqrt(<i>x</i>)); end

- Vòng lặp **while** thực hiện một nhóm các phát biểu cho đến khi biểu thức điều kiện là FALSE.

Cú pháp
while <i>biểu thức điều kiện</i> phát biểu 1; phát biểu 2; phát biểu 3; end

Ví dụ
<i>N</i> = 100; <i>iter</i> = 1; <i>msum</i> = 0; while <i>iter</i> <= <i>N</i> <i>msum</i> = <i>msum</i> + <i>iter</i> ; <i>iter</i> = <i>iter</i> + 1; end ;

Sau đây là một ví dụ đơn giản, sử dụng cấu trúc vòng lặp để vẽ đồ thị hàm $f(x) = \cos(nx)$ với $n = 1, \dots, 9$ trong các cửa sổ con (**subplots**) khác nhau.

```
figure
hold on
x = linspace(0, 2*pi);
for n=1:9
    subplot(3, 3, n);
    y = cos(n*x);
    plot(x, y);
    axis tight
end
```

Cho hai vector x và y , ví dụ sau đây sử dụng cấu trúc vòng lặp để tạo ra một ma trận có các thành phần được định nghĩa như sau: $A_{ij} = x_i y_j$.

```
n = length(x);
m = length(y);
for i=1:n
    for j=1:m
        A(i,j) = x(i) * y(j);
    end
end
```

Sử dụng cấu trúc vòng lặp **while** để làm lại ví dụ trên với $x = [1 \ 2 \ -1 \ 5 \ -7 \ 2 \ 4]$ và $y = [3 \ 1 \ -5 \ 7]$.

```
n = length(x);
m = length(y);
i = 1; j = 1; % gán giá trị ban đầu cho i và j
while i <= n
    while j <= m
        A(i,j) = x(i) * y(j);
        j = j+1; % tăng j lên 1
    end
    i = i+1; % tăng i lên 1
end
```

✎ Bài tập 5-10.

Sử dụng cấu trúc vòng lặp để tính tổng của bình phương 50 số đầu tiên.

✎ Bài tập 5-11.

Viết đoạn chương trình tìm số lớn nhất của n sao cho: $\sqrt{1^2} + \sqrt{2^2} + \dots + \sqrt{n^2} < 1000$

✎ Bài tập 5-12.

Sử dụng cấu trúc vòng lặp để viết các đoạn chương trình sau đây:

1. Cho vector $x = [1 \ 8 \ 3 \ 9 \ 0 \ 1]$, cộng tất cả các thành phần của vector (kiểm tra lại với lệnh **sum**)
2. Cho $x = [4 \ 1 \ 6 \ -1 \ -2 \ 2]$ và $y = [6 \ 2 \ -7 \ 1 \ 5 \ -1]$, xác định ma trận có các thành phần như sau:
 - $a_{ij} = \frac{y_i}{x_j}$
 - $b_i = x_i y_i$
 - $c_{ij} = \frac{x_i}{(2 + x_i + y_j)}$
 - $d_{ij} = \frac{1}{\max(x_i, y_j)}$
3. Cho một ma trận A bất kỳ, viết đoạn chương trình xác định ma trận B là ma trận nghịch đảo của ma trận A (kiểm tra lại với toán tử A^{-1}).
4. Tạo một ma trận ngẫu nhiên A bằng lệnh **rand**, xóa bỏ tất cả các thành phần nhỏ hơn 0.5 của A.

🔗 Bài tập 5-13.

Viết đoạn chương trình yêu cầu nhập vào giá trị nhiệt độ °C (biến **tc**) và tính toán giá trị nhiệt độ °F tương đương (biến **tf**), $tf = \frac{9}{5} * tc + 32$. Đoạn chương trình sẽ chạy cho đến khi không có số nào được nhập để biến đổi. Sử dụng lệnh **input** để nhập vào các giá trị và lệnh **isempty** để kiểm tra có giá trị nào được nhập hay không (sử dụng lệnh **help input** và lệnh **help isempty** để hiểu rõ thêm).

🔗 Bài tập 5-14.

Sử dụng cấu trúc **while** viết đoạn chương trình tìm giá trị dương nhỏ nhất của x (lấy 4 số lẻ) sao cho $0.91 < \sin(x) + \cos(x) < 0.92$.

🔗 Bài tập 5-15.

Sử dụng cấu trúc vòng lặp **for** viết một đoạn chương trình nhập vào hai số **a** và **b**, sau đó cho biết sự tương quan giữa chúng (lớn hơn, nhỏ hơn, bằng).

Danh sách các lệnh và hàm được giới thiệu trong chương 5**Các lệnh rẽ nhánh**

for ... end	Vòng lặp có số lần lặp hữu hạn
if ... end	
if ... else ... end	Rẽ nhánh theo điều kiện
if ... elseif ... else ... end	
switch ... case ... end	Rẽ nhánh theo điều kiện
while ... end	Lặp theo điều kiện

Các hàm khác

all	Trả về giá trị true nếu tất cả các phần tử của một vector đều khác 0
any	Trả về giá trị true nếu ít nhất một phần tử của vector khác 0
disp	Hiển thị một chuỗi lên màn hình
find	Tìm vị trí của các phần tử khác 0 trong một vector
input	Yêu cầu người sử dụng nhập một chuỗi và trả về chuỗi được nhập
isempty	Kiểm tra một ma trận có rỗng hay không
rand	Ma trận ngẫu nhiên với các thành phần phân bố đều trên (0,1)
rem	Tính phần dư của phép chia hai số
xor	Xor hai phần tử logic

Chương 6

TẬP LỆNH VÀ HÀM

6.1. TẬP LỆNH M-FILE

Để giải quyết những vấn đề đơn giản, số lệnh sử dụng trong chương trình ít, các lệnh của MATLAB có thể được nhập trực tiếp vào cửa sổ lệnh. Tuy nhiên trong trường hợp phức tạp, số lệnh cần được sử dụng là khá nhiều và do vậy việc nhập trực tiếp vào cửa sổ lệnh là không khả thi. Giải pháp trong trường hợp này là sử dụng tập lệnh m-file. Ngoài ưu điểm trong trường hợp chương trình cần sử dụng nhiều lệnh, tập lệnh m-file còn hữu ích khi chúng ta muốn thay đổi một vài giá trị của biến hay cho chương trình thực thi nhiều lần. Trong trường hợp này, m-file là một tập tin chứa một chuỗi các lệnh hay phát biểu trong MATLAB. Tuy nhiên do chúng không có các đối số vào và ra nên tập tin m-file không phải là hàm, nó đơn giản chỉ là tập hợp của nhiều lệnh được thực thi một cách trình tự.

🔗 Bài tập 6-1.

Mở chương trình soạn thảo của MATLAB (chọn **File** trên thanh công cụ, chọn **New** và sau đó chọn **m-file**) nhập vào cửa sổ chương trình các dòng lệnh sau đây và lưu tập tin với tên **sinplot.m**

```
x = 0:0.2:6;  
y = sin(x);  
plot(x,y);  
title('Plot of y = sin(x)');
```

Thực thi tập tin vừa tạo bằng cách: trong cửa sổ lệnh của chương trình MATLAB, nhập vào lệnh **sinplot**. Quan sát các kết quả thu được.

```
>> sinplot
```

🔗 Bài tập 6-2.

Đoạn chương trình trong bài tập 51 được sử dụng để vẽ hình sin. Tương tự như bài tập 51 hãy tạo các tập lệnh để vẽ những đồ thị hàm sau đây:

- Vẽ đồ thị hàm $y = 2^{\cos(x)}$ với x trong khoảng $[-10, 10]$.
- Vẽ hàm $y = x^2$ và $x = y^2$ trong cùng một cửa sổ hình. Cộng thêm vào tựa đề và các mô tả của đồ thị.

🔗 Bài tập 6-3.

Tạo một tập lệnh **cubic_roots** yêu cầu nhập vào ba biến a, b, c và kết quả trả về là nghiệm của phương trình bậc ba: $ax^2 + bx + c = 0$.

🔗 Bài tập 6-4.

Hãy viết một tập lệnh yêu cầu nhập vào hai số a và b . Xác định đỉnh còn lại của tam giác đều có hai đỉnh là $[a, a]$ và $[a, b]$. Vẽ tam giác đều vừa mới được xác định, tô hình tam giác đó với lệnh **fill** (sử dụng lệnh **help fill** để tìm hiểu cách sử dụng của lệnh này).

6.2. HÀM M-FILE

Hàm m-file là một chương trình con do chúng yêu cầu các đối số ngõ vào và có thể trả về đối số ngõ ra. Các biến được định nghĩa và sử dụng trong hàm thì chỉ có giá trị bên trong hàm đó. Cú pháp tổng quát của một hàm như sau:

function [outputArgs] = **function_name**(inputArgs)

outputArgs (đối số ngõ ra) được đặt trong dấu []:

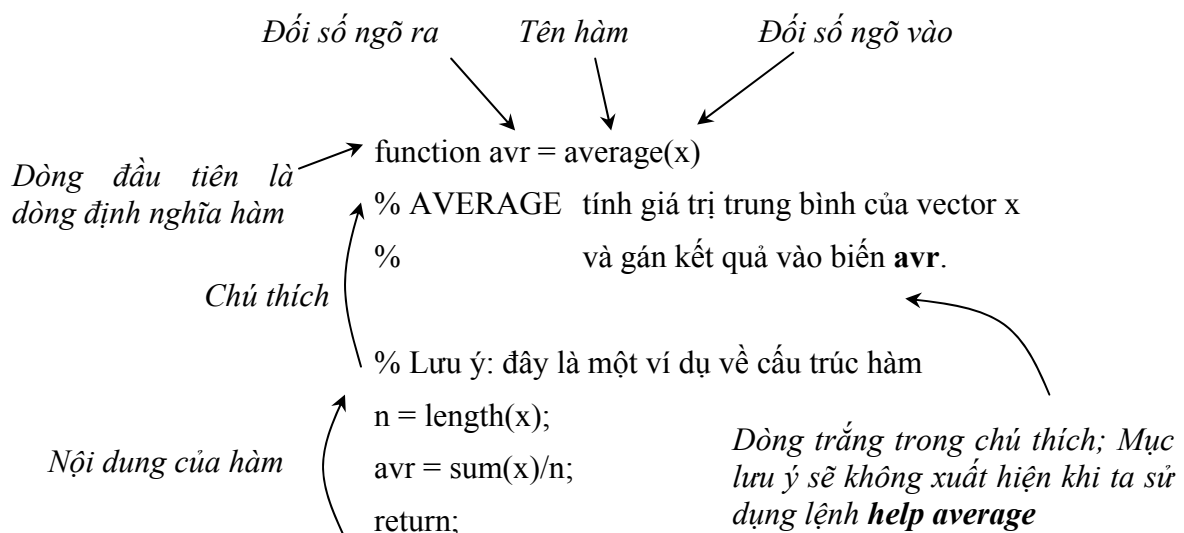
- tên các đối số ngõ ra được liệt kê và cách nhau bởi dấu ‘,’
- có thể không sử dụng dấu [] nếu chỉ có duy nhất một đối số.
- một hàm cũng có thể không có đối số ngõ ra¹.

outputArgs (đối số ngõ vào) được đặt trong dấu ():

- tên các đối số ngõ vào được liệt kê và cách nhau bởi dấu ‘,’
- một hàm cũng có thể không có đối số ngõ vào.

Cấu trúc chung của một hàm trong MATLAB như sau:

- Dòng đầu tiên là định nghĩa của hàm, dòng này bao gồm: từ khóa khai báo hàm, tên hàm và các đối số ngõ vào và ngõ ra.
- Tiếp theo là các dòng chú thích, các dòng này được bắt đầu bởi dấu ‘%’, nội dung của các dòng này là giải thích mục đích của hàm cũng như mô tả ý nghĩa các đối số ngõ vào và ngõ ra. Các dòng chú thích này sẽ xuất hiện khi ta sử dụng lệnh **help** để tìm hiểu về hàm chúng ta đang tạo ra.
- Cuối cùng là phần nội dung, đây chính là phần chính của hàm. Việc thực thi nội dung của hàm sẽ kết thúc khi chương trình gặp lệnh **return** hoặc thực thi xong lệnh cuối cùng. Khi quá trình thực thi kết thúc các đối số ngõ ra sẽ được trả về. Ví dụ hàm **average** được định nghĩa như sau:



Lưu ý rằng tên của hàm và tên của file lưu phải giống nhau. Trong trường hợp trên, hàm nên được lưu với tên **average.m**.

¹ Trong các ngôn ngữ lập trình khác, một hàm không có đối số ngõ ra được gọi là **thủ tục**.

✎ Bài tập 6-5.

Tạo hàm **average** (có phần chú thích) và lưu với tên `average.m`. Kiểm tra khả năng và công dụng của hàm bởi lệnh **help average**. Thực thi hàm `average.m` bằng lệnh `avr1 = average(1:10)`. Giải thích các kết quả đạt được.

Một ví dụ khác về hàm:

```
function d = determinant(A)
%DETERMINANT Tính định thức của ma trận A.
[m,n] = size(A);
if (m ~= n)
    disp('Lỗi. Ma trận không phải là ma trận vuông.');
```

`return;`

```
else
    d = det(A);           % det là hàm cơ bản của MATLAB.
end
return;
```

Hàm **error** có thể được sử dụng để kiểm tra các điều kiện của thông số. Hàm này sẽ hiển thị một thông báo lỗi, lờ đi tất cả các lệnh khác trong chương trình và quay về cửa sổ lệnh của chương trình MATLAB. Ví dụ hãy tạo một tập lệnh chứa đoạn chương trình sau đây, đoạn chương trình này kiểm tra điều kiện của thông số `a`.

```
a = input('Hãy nhập vào giá trị của a:');
if (a >= 1)
    error('a phải nhỏ hơn 1');
```

`end`

✎ Bài tập 6-6.

Viết hàm `[elems, mns] = nonzero(A)`, với đối số ngõ vào là ma trận `A` và giá trị trả về là vector `elems` chứa tất cả các thành phần nonzero của `A`, đối số `mns` chứa giá trị trung bình từng cột của `A`.

✎ Bài tập 6-7.

Viết hàm `[A,B] = sides(a,b,c)`, với đối số ngõ vào là 3 số dương. Nếu 3 số đó có thể là ba cạnh của một tam giác chương trình sẽ trả về giá trị diện tích và chu vi của nó. Trong trường hợp còn lại thì thông báo lỗi cho các trường hợp còn lại.

✎ Bài tập 6-8.

Viết hàm `[A] = matrix(n, m, min, max)`, các đối số ngõ vào là các số nguyên. Nếu `n` và `m` là các số dương, chương trình sẽ trả về một ma trận `nxm` có các thành phần là các số ngẫu nhiên trong khoảng `[min, max]`. Thông báo lỗi cho các trường hợp còn lại.

✎ Bài tập 6-9.

Viết hàm `[x1, x2] = bac_hai(a, b, c)`. Trong trường hợp `a, b, c` khác zero, chương trình trả về nghiệm của phương trình $ax^2 + bx + c = 0$. Thông báo lỗi cho các trường hợp còn lại.

Một hàm m-file có thể chứa nhiều hơn một hàm. Trong trường hợp một m-file chứa nhiều hàm thì hàm xuất hiện đầu tiên được gọi là hàm chính, hàm chính vẫn có các tính chất thông thường của một hàm: cách hoạt động, các đối số, tên hàm, Những hàm còn lại được gọi là

hàm con, chúng chỉ có thể được truy xuất từ hàm chính và không thể được truy xuất từ bất kỳ hàm nào khác, kể cả từ cửa sổ lệnh. Cấu trúc của một hàm con cũng gồm ba phần: khai báo hàm, chú thích và thân hàm. Các hàm con cần được sử dụng trong trường hợp một hàm quá dài hay quá phức tạp. Ví dụ hàm **average** bây giờ là một hàm con của hàm **stat.m**.

```
function [a,sd] = stat(x)
% STAT Simple statistics.
% Computes the average value and the standard deviation of a
vector x.
n = length(x);
a = average(x,n);
sd = sqrt(sum((x - avr).^2)/n);
return;

function a = average (x,n)
% AVERAGE subfunction
a = sum(x)/n;
return;
```

Trong đoạn chương trình trên, lệnh **a = average(x, n)** được sử dụng trong nội dung của hàm **stat** để gọi một hàm con. Trong trường hợp này **stat** là hàm chính và **average** là hàm con.

6.2.1. NHỮNG BIẾN ĐẶC BIỆT TRONG HÀM

Khi một hàm được sử dụng, chương trình sẽ tự động tạo ra hai biến nội là **nargin**, **nargout**. Biến **nargin** là số đối số ngõ vào được sử dụng khi gọi hàm, **nargout** là số đối số ngõ ra. Phân tích hàm sau đây:

```
function [out1,out2] = checkarg (in1,in2,in3)
%CHECKARG Một ví dụ sử dụng biến nargin và nargout.
if (nargin == 0)
    disp('no input arguments');
return;
elseif (nargin == 1)
    s = in1;
    p = in1;
    disp('1 input argument');
elseif (nargin == 2)
    s = in1+in2;
    p = in1*in2;
    disp('2 input arguments');
elseif (nargin == 3)
    s = in1+in2+in3;
    p = in1*in2*in3;
    disp('3 input arguments');
```

```

else
    error('Too many inputs.');
```

```

end
if (nargout == 0)
    return;
elseif (nargout == 1)
    out1 = s;
else
    out1 = s;
    out2 = p;
end
```

Trong ví dụ ở trên, chương trình sẽ cho ra các kết quả khác nhau tùy thuộc vào có bao nhiêu đối số ngõ vào được sử dụng và có bao nhiêu đối số ngõ ra. Trong các chương trình lớn, một cách tổng quát người ta thường hay sử dụng biến nargin và nargout để điều khiển linh hoạt nội dung của chương trình.

✎ Bài tập 6-10.

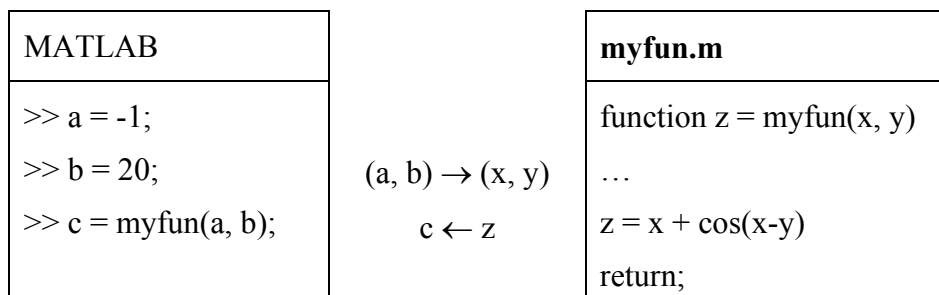
Xây dựng hàm **checkarg** với nội dung như trên, hãy gọi hàm **checkarg** với các đối số ngõ vào và ngõ ra khác nhau. Một số trường hợp gợi ý:

```

>> checkarg
>> s = checkarg(-6)
>> s = checkarg(23,7)
>> [s,p] = checkarg(3,4,5)
```

6.2.2. BIẾN TOÀN CỤC VÀ BIẾN CỤC BỘ

Mỗi hàm m-file truy cập đến một phần riêng biệt của bộ nhớ trong không gian làm việc của MATLAB. Điều này có nghĩa là mỗi hàm m-file có những biến cục bộ của riêng nó, những hàm khác không thể truy cập đến các biến này. Để hiểu rõ hơn chúng ta hãy phân tích sơ đồ sau đây:



Hình 6.1

Các biến a, b, c xuất hiện trong không gian làm việc của MATLAB, trong khi các biến x, y, z chỉ có giá trị bên trong hàm **myfun**. Tuy nhiên, nếu ta khai báo biến ở dạng toàn cục thì tất cả các hàm khác đều có thể sử dụng được biến này (tham khảo thêm bằng lệnh **help global**).

Lưu ý rằng chúng ta nên cẩn thận khi sử dụng biến toàn cục, nó dễ gây nên những xáo trộn và nhầm lẫn.

6.2.3. CÁCH GỌI HÀM GIÁN TIẾP

Để chương trình trở nên tổng quát hơn ta nên sử dụng cách gọi hàm gián tiếp, trong trường hợp này tên hàm được coi như là một đối số ngõ vào. Ta sử dụng lệnh **feval** (function evaluation) để gọi hàm gián tiếp. Cách sử dụng thông thường của lệnh **feval** như sau:

$$[y1, \dots, yn] = \text{feval}(F, x1, \dots, xn),$$

với F là tên của hàm được định nghĩa trong MATLAB, $x1, \dots, xn$ là các đối số ngõ vào, $y1, \dots, yn$ là các đối số ngõ ra. Ví dụ:

```
>> x = pi; y = cos(x);
```

```
>> z = feval('cos', x);
```

Lệnh sau thì tương đương với cả hai lệnh đầu tiên. Một cách sử dụng khác của hàm **feval**.

```
>> F = 'cos';
```

```
>> z = feval(F, x)
```

Cách gọi hàm gián tiếp là một công cụ cần thiết để xây dựng chương trình xem hàm như là một đối số.

✎ Bài tập 6-11.

Hãy miêu tả hoạt động của hàm **funplot** sau:

```
function funplot (F, xstart, xend, col);
%FUNPLOT makes a plot of the function F at the interval
%      [xstart, xend]. The plot should be made in one
%      of the standard Matlab colors, so 'col' is one
%      of the following value:
%      'b','k','m','g','w','y' or 'r'.
% default values:
% [xstart,xend] = [0,10]
% col = 'b'
% Note: illustrates the use of feval command

if (nargin == 0)
    error ('No function is provided.');
```

```
end
if (nargin < 2)
    xstart = 0;
    xend = 10;
end
if (nargin == 2)
    error ('Wrong number of arguments. You should provide xstart
and xend.');
```

```
end
if (nargin < 4)
```

```

        col = 'b';
    end

    if (xstart == xend),
        error ('The [xstart, xend] should be a non-zero range. ');
    elseif (xstart > xend),
        exchange = xend;
        xend = xstart;
        xstart = exchange;
    end

    switch col
    case {'b','k','m','g','w','y','r'}; % do nothing; the right color choice
    otherwise
        error ('Wrong col value provided. ')
    end

    x = linspace(xstart, xend);
    y = feval(F, x);
    plot (x, y, col);
    description = ['Plot of ', F];
    title (description);
    return;

```

Lưu ý cách ghi chú thích, biến **nargin** và cấu trúc **switch**.

Hãy sử dụng hàm **funplot** vừa tạo để vẽ các đồ thị hàm khác nhau, ví dụ: sin, cos, exp,

6.3. TẬP TIN VÀ HÀM

Điểm khác nhau cơ bản nhất giữa tập tin và hàm là tất cả các tham số và biến số trong tập tin đều có thể được sử dụng từ bên ngoài (các tham số và biến này nằm trong workspace), trong khi các biến của hàm chỉ có giá trị bên trong hàm đó và không được truy xuất hoặc sử dụng bởi các hàm khác. Do vậy trong trường hợp cần giải quyết một vấn đề với các thông số bất kỳ ta nên sử dụng hàm, còn trong trường hợp cần thử nghiệm nhiều lần một vấn đề ta nên sử dụng tập tin.

Bài tập 6-12.

Tạo ra một hàm **binom**, có header **function b = binom(n, k)**, để tính giá trị của nhị thức $\binom{n}{k}$.

Lưu ý rằng trong trường hợp này ta nên tạo ra một hàm con **factorial** để tính giá trị của biểu thức $n! = 1 * 2 * \dots * n$. Hàm **factorial** có thể là một hàm độc lập hay cũng có thể là một hàm

con trong file **binom.m**. Sau khi tạo xong hàm **binom**, hãy viết một tập tin tính toán các nhị thức $\binom{n}{k}$ với $n = 8$ và $k = 1, 2, \dots, 8$.

✎ Bài tập 6-13.

Viết một hàm có ba đối số là các số nguyên dương.

- Nếu ba số đó có thể là ba cạnh của một tam giác, kết quả trả về là diện tích của hình tam giác đó. Biết công thức tính diện tích của một tam giác:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \text{ với } s = \frac{a+b+c}{2}$$

- Ngược lại nếu ba số nhập vào không phải là ba cạnh của một tam giác thì thông báo lỗi.

✎ Bài tập 6-14.

Viết một hàm kiểm tra hai đối số nguyên dương có phải là hai số nguyên tố tương đối hay không? Hai số được gọi là nguyên tố tương đối khi chúng có ước số chung lớn nhất là 1.

✎ Bài tập 6-15.

Một công ty điện lực tính giá tiền điện theo tỷ giá sau đây:

- 1000đ/kwh cho 300 kwh đầu tiên.
- 800đ/kwh cho 300 kwh tiếp theo.
- 600đ/kwh cho 200 kwh tiếp theo.
- 500đ/kwh cho tất cả các mức sử dụng điện trên 800 kwh.

Hãy viết chương trình tính tiền điện theo số kwh.

✎ Bài tập 6-16.

Viết hàm có một đối số là vector x (vector của các số nguyên dương), kết quả trả về là vector y chứa các ước số nhỏ nhất lớn hơn 1 của các thành phần trong vector x.

✎ Bài tập 6-17.

Viết hàm sắp xếp các thành phần trong vector x theo thứ tự tăng dần hoặc giảm dần.

✎ Bài tập 6-18.

Viết chương trình phân loại sinh viên theo điểm trung bình:

- Yếu: $0 \leq đtb < 5$
- Trung bình: $5 \leq đtb < 6.5$
- Khá: $6.5 \leq đtb < 8$
- Giỏi: $8 \leq đtb < 9$
- Xuất sắc: $9 \leq đtb \leq 10$

Danh sách các lệnh và hàm được giới thiệu trong chương 6

error	Hiển thị một thông báo lỗi và dừng đoạn chương trình đang thực thi
feval	Thực thi một hàm được định nghĩa trước
fill	Tô màu một đa giác phẳng (2D)
global	Định nghĩa biến toàn cục
return	Kết thúc gọi hàm và trả về giá trị của hàm

Chương 7

VĂN BẢN

7.1. CHUỖI KÝ TỰ

Trong MATLAB văn bản được lưu ở dạng là chuỗi của các ký tự. Chuỗi là một vector mà các thành phần của nó lưu trữ giá trị ASCII của các ký tự trong chuỗi. Do văn bản là một vector của các ký tự nên ta cũng có thể xử lý văn bản giống như xử lý một vector bất kỳ. Ví dụ:

```
>> t = 'This is a character string'
t =
This is a character string
>> size(t)
ans =
     1     27
>> whos
      Name      Size      Bytes      Class
      t         1x27         54      char array
>> t(10 : 19)
ans =
character
>> t([2, 3, 10, 17])
ans =
hi t
```

Để miêu tả chuỗi dưới dạng mã ASCII ta sử dụng lệnh **double** hoặc **abs** để biến đổi:

```
>> double(t(1:12))
ans =
     84    104    105    115     32    105    115     32     97     32     99    104
```

Hàm **char** được sử dụng để biến đổi từ dạng mã ASCII sang ký tự chuỗi:

```
>> t([16:17])
ans =
ct
>> t([16:17])+3           % mã ASCII được sử dụng.
ans =
    102    119
>> t([16:17])-3          % thay đổi mã ASCII
ans =
     96    113
>> char(t([16:17])-2)     % biến đổi mã ASCII sang ký tự chuỗi.
```

```
ans =
ar
```

Bài tập 7-1.

Thực thi các lệnh được miêu tả ở trên, sử dụng chuỗi `t` để tạo ra chuỗi `u` chỉ chứa các ký tự *'character'*. Biến đổi chuỗi `u` thành chuỗi `u1 = 'retcarahc'`.

Sau đây là một số ví dụ về các hàm được sử dụng để biến đổi chuỗi. Ví dụ hàm **findstr** được sử dụng để tìm một ký tự hay một nhóm các ký tự xuất hiện trong chuỗi.

```
>> findstr(t, 'c')           % tìm vị trí của 'c' xuất hiện trong t
ans =
    11    16
>> findstr(t, 'racter')      % tìm vị trí của chuỗi 'racter' trong t
ans =
    14
>> findstr(t,u)              % tìm chuỗi u trong chuỗi t
ans =
    11
>> strcat(u,u1)              % nối hai chuỗi u và u1 lại với nhau
ans =
characterretcarahc
>> strcmp(u,u1)              % so sánh hai chuỗi
ans =
                                % trả về 1 nếu hai chuỗi giống nhau
                                % và ngược lại là 0
    0
>> q = num2str(34.35)         % biến đổi số thành chuỗi
q =
34.35
>> z = str2num('7.6')         % biến đổi chuỗi thành số.
z =
    7.6
>> whos    q    z              % q là chuỗi (ma trận các ký tự), z là số.

    Name  Size  Bytes    Class
    q      1x5   10    char array
    z      1x1    8    double array
>> t = str2num('1  -7  2') % biến đổi chuỗi thành vector của các số.
t =
    1    -7     2
>> t = str2num('1  - 7  2') % lưu ý các khoảng trắng xung quanh
```

```

% dấu '-' hoặc '+', trong trường hợp
% này là: [1-7, 2]

t =

    -6     2

```

Lưu ý rằng khi biến đổi từ chuỗi sang số hoặc từ số sang chuỗi, các khoảng trắng xung quanh dấu '-' hoặc '+' mang ý nghĩa rất quan trọng.

```

>> A = round(4*rand(3,3))+0.5;
>> ss = num2str(A)           % biến đổi ma trận A thành chuỗi các ký
tự.
ss =

    -3.5   -3.5    6.5
    -2.5   -1.5    0.5
     5.5   -1.5   -3.5

>> whos ss

      Name  Size  Bytes      Class
      ss    3x28  168      char array

>> ss(2,1), ss(3,15:28)      % ss là một ma trận các ký tự.
ans =

-
ans =

.5   -3.5

>> ss(1:2,1:3)
ans =

-3.
-2.

```

Bài tập 7-2.

Thực thi các lệnh trong ví dụ ở trên. Định nghĩa một chuỗi mới `s = 'Nothing wastes more energy than worrying'` và thực hiện các ví dụ với lệnh **findstr**.

7.2. XUẤT VÀ NHẬP VĂN BẢN

Lệnh **input** được sử dụng để yêu cầu người dùng nhập vào một số hay một chuỗi.

```

>> myname = input('Enter your name: ','s');
>> age = input('Enter your age: ');

```

Lưu ý ý nghĩa của hai lệnh trên, lệnh thứ nhất yêu cầu nhập vào một chuỗi và lệnh thứ hai yêu cầu nhập vào một số.

Có hai hàm được sử dụng để xuất văn bản là **disp** và **fprintf**. Hàm **disp** chỉ thể hiện giá trị của một đối số là một ma trận số hay ma trận chuỗi. Ví dụ:

```

>> disp('This is a statement.') % xuất ra một chuỗi.
This is a statement.

>> disp(rand(3))              % xuất ra một ma trận.

```

0.2221	0.0129	0.8519
0.4885	0.0538	0.5039
0.2290	0.3949	0.4239

Hàm **fprintf** (tương tự trong ngôn ngữ lập trình C) được sử dụng để ghi dữ liệu vào file hay xuất ra màn hình định dạng chính xác của đối số. Ví dụ:

```
>> x = 2;
>> fprintf('Square root of %g is %8.6f.\n', x, sqrt(x));
Square root of 2 is 1.414214.
>> str = 'beginning';
>> fprintf('Every %s is difficult.\n',str);
Every beginning is difficult.
```

Hàm **fprintf** có khả năng biến đổi, định dạng và ghi đối số của nó vào một file hay thể hiện chúng trên màn hình theo một định dạng đặc biệt. Cú pháp sau đây được sử dụng để xuất ra màn hình:

fprintf (format, a, ...)

Chuỗi **format** chứa các ký tự thông thường sẽ được đưa đến ngõ ra và các *chỉ định biến đổi*, mỗi *chỉ định biến đổi* sẽ chuyển đổi một đối số của hàm **fprintf** thành chuỗi ký tự có định dạng tương ứng và sau đó in ra màn hình. **a** là các đối số của hàm **fprintf**. Mỗi *chỉ định biến đổi* bắt đầu bằng ký tự '%' và kết thúc bởi một *ký tự chuyển đổi*. Giữa '%' và *ký tự chuyển đổi* có thể là:

- dấu '-': khi có ký tự này đối số sẽ được in ra với định dạng canh lề trái, trong trường hợp ngược lại, mặc định là canh lề phải.
- số xác định chiều dài tối thiểu: là phạm vi tối thiểu mà đối số sẽ được in ra, trong trường hợp chiều dài của đối số lớn hơn chiều dài tối thiểu thì phạm vi in ra sẽ bằng với chiều dài của đối số.
- số xác định số các chữ số thập phân
- dấu '.': là dấu dùng để phân biệt hai định dạng trên.

Bảng 7.1: Một số ký tự chuyển đổi và ý nghĩa của chúng

Ký tự	Ý nghĩa.
d	biến đổi đối số sang kiểu số thập phân
u	biến đổi đối số sang kiểu số thập phân không dấu
c	biến đổi đối số thành một ký tự
s	biến đổi đối số thành một chuỗi
e	biến đổi số single hay double thành số thập phân có dạng [±]m.nnnnnnE[±]xx . Xem thêm ví dụ bên dưới
f	biến đổi số single hay double thành số thập phân có dạng [±]mmm.nnnnnn . Xem thêm ví dụ bên dưới
g	định dạng giống trường hợp e và f nhưng được viết ở dạng ngắn hơn. Các số zero không có ý nghĩa sẽ không được in ra

Các định dạng `\n`, `\r`, `\t`, `\b` lần lượt được sử dụng để tạo ra một dòng mới, xuống dòng, tab và tab ngược (ngược với phím tab). Để in ra ký tự `\` ta sử dụng `\\`, tương tự, để in ra ký tự `%` ta sử dụng `%%`. Phân tích ví dụ sau đây:

```
>> fprintf('look at %20.6e!\n', 1000*sqrt(2))
look at                1.414214e+3!
>> fprintf('look at %-20.6f!', 1000*sqrt(2))
look at 1414.213562          !
```

Trong cả hai trường hợp, khoảng tối thiểu để in đối số là 20 và số chữ số thập phân là 6. Trong trường hợp đầu, giá trị $1000\sqrt{2}$ được định dạng canh lề phải và trong trường hợp thứ hai, bởi vì có dấu `-` nên giá trị $1000\sqrt{2}$ được định dạng canh lề trái. Sự khác nhau của hai kết quả còn do ý nghĩa của các ký tự biến đổi *e* và *f*.

🔗 Bài tập 7-3.

Sử dụng lệnh **input** để nhập vào các giá trị của một vector. Sau đó sử dụng lệnh **disp** hay lệnh **fprintf** để xuất vector mới vừa tạo ra màn hình.

🔗 Bài tập 7-4.

Xem xét các ví dụ sau đây về lệnh **fprintf** và sau đó tự làm các bài tập tương tự để hiểu rõ hơn về các *chỉ định* này:

```
>> str = 'life is beautiful';
>> fprintf('My sentence is: %s\n',str);      % lưu ý định dạng \n
My sentence is: life is beautiful
>> fprintf('My sentence is: %30s\n',str);
My sentence is:                life is beautiful
>> fprintf('My sentence is: %30.10s\n',str);
My sentence is:                life is be
>> fprintf('My sentence is: %-20.10s\n',str);
My sentence is: life is be
>>
>> name = 'John';
>> age = 30;
>> salary = 6130.50;
>> fprintf('My name is %4s. I am %2d. My salary is f %7.2f.\n',name,
age, salary);
My name is John. I am 30. My salary is f 6130.50.
>>
>> x = [0, 0.5, 1];
>> y = [x; exp(x)];
>> fprintf('%6.2f %12.8f\n',y);
0.00      1.00000000
0.50      1.64872127
1.00      2.71828183
```

```
>>
>> fprintf('%6.1e %12.4e\n',y);
0.0e+00    1.0000e+00
5.0e-01    1.6487e+00
1.0e+00    2.7183e+00
>>
>> x = 1:3:7;
>> y = [x; sin(x)];
>> fprintf('%2d %10.4g\n',y);
1      0.8415
4     -0.7568
7      0.657
```

Lưu ý rằng lệnh `fprintf` sử dụng các chỉ định biến đổi để khai báo có bao nhiêu đối số theo sau và kiểu của từng đối số. Nếu chúng ta không cung cấp đủ đối số hay cung cấp đối số không đúng kiểu thì chúng ta sẽ nhận được một kết quả không đúng. Do vậy, cần cẩn thận với việc khai báo đối số trong lệnh `fprintf`. Xem xét ví dụ sau:

```
>> fprintf('My name is %4s. I am %2d. My salary is f
%7.2f.\n',name,salary);
My name is John. I am 6.130500e+03. My salary is f
```

Chú ý rằng lệnh `sprintf` cũng tương tự như lệnh `fprintf` ngoại trừ lệnh `sprintf` được sử dụng để gán kết quả cho một chuỗi. Xem xét ví dụ sau đây:

```
>> str = sprintf('My name is %4s. I am %2d. My salary is f
%7.2f.\n',name,age,salary)
str =
My name is John. I am 30. My salary is f 6500.50.
```

🔗 Bài tập 7-5.

Định nghĩa chuỗi `s = 'How much wood could a wood-chuck chuck if a wood-chuck could chuck wood?'`, sử dụng lệnh `findstr` tìm các vị trí xuất hiện các chuỗi con 'wood', 'o', 'uc' hay 'could' trong chuỗi `s`.

🔗 Bài tập 7-6.

- Viết một tập tin hay hàm để biến đổi số La Mã sang giá trị thập phân tương ứng.

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Ví dụ: IV(4), IX(9), XL(40), XC(90), CD(400) và CM(900).

- Viết một hàm làm công việc ngược lại, biến đổi số thập phân sang số La Mã.

🔗 Bài tập 7-7.

Viết hàm mã hóa và giải mã theo luật sau đây: *tất cả các ký tự trong chuỗi sẽ có giá trị ASCII của nó dịch đi n, với n là một đối số. Ví dụ trong trường hợp n = 1, chuỗi 'abcde' sẽ được mã hóa thành 'bcdef' và 'alphabet' được mã hóa thành 'bmqibcfu'. Trong trường hợp này 'z' sẽ*

được mã hóa thành 'a'. Mã hóa với $n = 26$ sẽ cho ta ta kết quả là chuỗi đã nhập vào. Viết hai hàm **coder** và **decoder**, mỗi hàm có hai đối số, đối số thứ nhất là chuỗi cần mã hóa/giải mã và đối số thứ hai là số cần dịch n . Trong trường hợp mã hóa, chuỗi nhập vào nên là chuỗi đã được mã hóa.

✎ **Bài tập 7-8.**

Viết một hàm xóa bỏ tất cả các ký tự trắng trong chuỗi.

✎ **Bài tập 7-9.**

Viết một hàm nhập vào một danh từ và trả về dạng số nhiều của nó dựa trên những quy tắc sau:

- a) Nếu danh từ kết thúc bằng 'y', hãy loại bỏ 'y' rồi thêm vào 'ies'
- b) Nếu danh từ kết thúc bằng 's', 'ch' hay 'sh' hãy thêm vào 'es'
- c) Trong các trường hợp khác thêm 's'.

✎ **Bài tập 7-10.**

Viết chương trình đảo ngược các từ trong chuỗi. Ví dụ 'Nguyen Van An' thành 'An Van Nguyen'.

Danh sách các hàm được giới thiệu trong chương 7

abs	Chuyển kiểu ký tự thành mã ASCII
char	Chuyển mã ASCII thành ký tự
disp	Hiển thị một chuỗi ra cửa sổ lệnh của MATLAB
double	Chuyển kiểu ký tự thành mã ASCII
findstr	Tìm một chuỗi trong một chuỗi khác
fprintf	Biến đổi định dạng một chuỗi và ghi vào một file hoặc hiển thị ra màn hình
input	Yêu cầu người sử dụng nhập vào một chuỗi và trả về chuỗi được nhập
sprintf	Giống fprintf nhưng kết quả được gán cho một chuỗi

Chương 8

GIAO DIỆN NGƯỜI SỬ DỤNG (GUI)

Giao diện người sử dụng (Graphical User Interface – GUI) là giao diện bằng hình ảnh của chương trình. Một GUI tốt có thể làm cho chương trình trở nên dễ sử dụng bằng cách cung cấp những thông tin ban đầu cần thiết và với những công cụ điều khiển như: nút nhấn (pushbutton), hộp liệt kê (list box), thanh trượt (slider), trình đơn (menu), GUI nên được thiết kế một cách dễ hiểu và thân thiện để người sử dụng có thể hiểu và dự đoán được kết quả của một tác động.

8.1. CÁCH LÀM VIỆC CỦA MỘT GUI

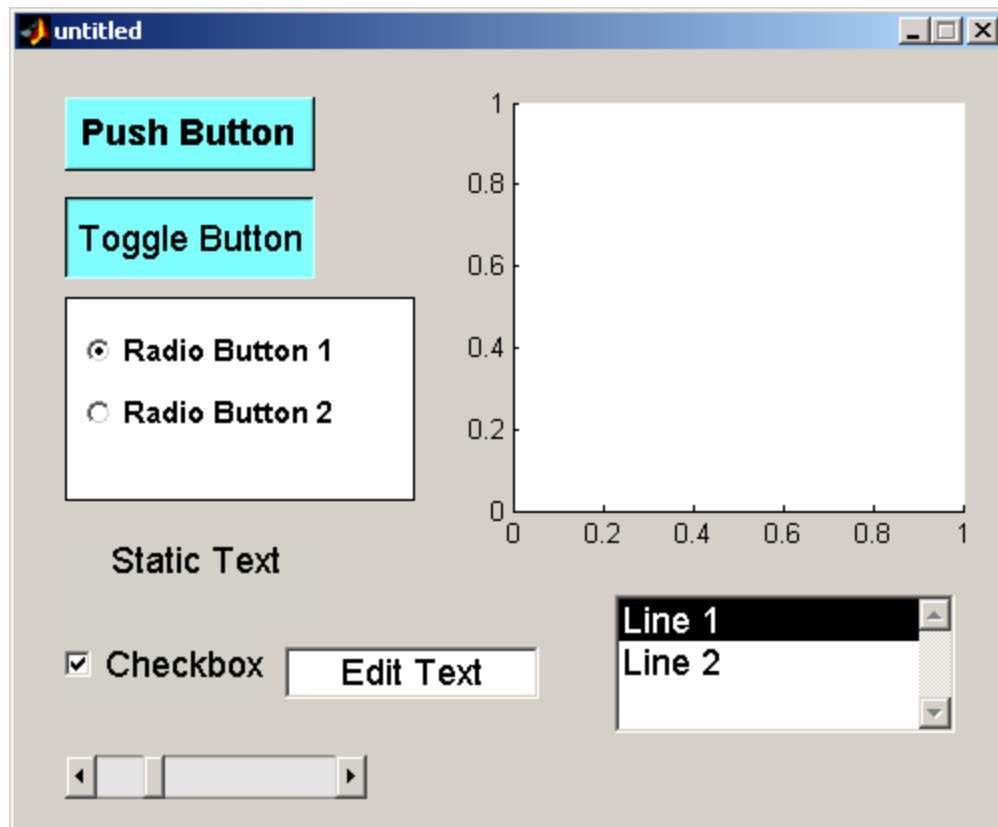
GUI bao gồm các nút nhấn, hộp liệt kê, thanh trượt, menu, ..., chúng cung cấp cho người sử dụng một môi trường làm việc thân thiện để họ tập trung vào các ứng dụng của chương trình hơn là đi tìm hiểu cách thức làm việc của chương trình. Tuy nhiên, tạo ra GUI là công việc khó khăn đối với người lập trình bởi vì chương trình phải được xử lý với các click chuột cho bất kỳ thành phần nào của GUI và vào bất kỳ thời điểm nào. Trong MATLAB, để tạo ra một GUI lưu ý ba yêu cầu chính sau đây:

- **Component** (các thành phần): mỗi đối tượng trong GUI (nút nhấn, nhãn, hộp soạn thảo, ...) là một thành phần. Các thành phần được phân loại thành: công cụ điều khiển (nút nhấn, hộp soạn thảo, thanh trượt, ...), các thành phần tĩnh (khung hình, chuỗi ký tự, ...), menu và axes (là các hệ trục dùng để hiển thị hình đồ họa). Các công cụ điều khiển và các thành phần tĩnh được tạo ra bởi hàm **uicontrol**, menu được tạo ra bởi các hàm **uimenu** và **uicontextmenu**, axes được tạo ra bởi hàm **axes**.
- **Figure**: các thành phần của GUI phải được sắp xếp vào trong một figure, là một cửa sổ được hiển thị trên màn hình máy vi tính. Trong các chương trước, một figure được tự động tạo ra khi vẽ đồ thị. Lệnh **figure** tạo ra một figure được sử dụng để chứa các thành phần của GUI.
- **Callback**: cuối cùng, khi người sử dụng tác động vào chương trình bằng cách nhấn chuột hay gõ bàn phím thì chương trình phải đáp ứng lại mỗi sự kiện này. Ví dụ, trong trường hợp người sử dụng tác động vào một nút nhấn thì MATLAB sẽ thực thi một hàm tương ứng với nút nhấn đó. Mỗi thành phần của GUI phải callback một hàm tương ứng của nó.

Các thành phần cơ bản của GUI được tóm tắt trong **bảng 8.1** và được thể hiện trong **hình 8.1**. Chúng ta sẽ tìm hiểu những thành phần này thông qua các ví dụ và sau đó sử dụng chúng để tạo ra các GUI.

Bảng 8.1 Một số thành phần cơ bản của GUI

Công cụ	Tạo bởi hàm	Miêu tả
Các công cụ điều khiển		
Pushbutton	uicontrol	Là một nút nhấn. Nó sẽ gọi hàm khi nhấn vào nó.
Toggle button	uicontrol	Là nút nhấn có hai trạng thái là “on” và “off”. Khi có tác động nó sẽ gọi hàm tương ứng và thay đổi trạng thái từ “on” sang “off” hoặc ngược lại.
Radio button	uicontrol	Cũng là một nút nhấn có hai trạng thái được thể hiện bởi một vòng tròn nhỏ, trạng thái “on” tương ứng với trường hợp có dấu chấm giữa vòng tròn và ngược lại là trạng thái “off”. Trong một nhóm Radio button ta chỉ có thể chọn được một thành phần. Khi có tác động vào mỗi thành phần sẽ có một hàm được gọi.
Check box	uicontrol	Cũng là một nút nhấn có hai trạng thái được thể hiện bởi một hình vuông nhỏ, trạng thái “on” tương ứng với trường hợp có đánh dấu giữa hình vuông và ngược lại là trạng thái “off”. Khi có tác động nó sẽ gọi hàm tương ứng và thay đổi trạng thái từ “on” sang “off” hoặc ngược lại.
List box	uicontrol	Là một danh sách các chuỗi. Người sử dụng có thể chọn một chuỗi bằng cách click hoặc double click vào nó. Chương trình sẽ gọi một hàm khi có một chuỗi được chọn.
Popup menus	uicontrol	Là công cụ cho phép chúng ta chọn một chuỗi trong một nhóm các chuỗi. Danh sách tất cả các chuỗi sẽ được hiển thị khi có click chuột. Khi không có click chuột công cụ chỉ thể hiện chuỗi hiện tại được chọn.
Slider	uicontrol	Là công cụ cho phép điều chỉnh một cách liên tục giá trị trong một thanh trượt. Mỗi khi giá trị của thanh trượt thay đổi sẽ có hàm được gọi.
Các thành phần tĩnh		
Frame	uicontrol	Được sử dụng để tạo ra một khung hình chữ nhật. Frame còn được sử dụng để nhóm các công cụ điều khiển lại với nhau. Frame không có khả năng gọi hàm.
Text field	uicontrol	Được sử dụng để tạo ra một nhãn bao gồm các ký tự. Text field không có khả năng gọi hàm.
Menu và trục đồ thị		
Menu items	uicontrol	Được sử dụng để tạo ra menu trên thanh công cụ. Chương trình sẽ gọi hàm khi một đối tượng trong menu được chọn.
Context menus	uicontextmenu	Được sử dụng để tạo ra menu xuất hiện khi right-click vào một hình trong giao diện.
Axes	axes	Được sử dụng để tạo một hệ trục đồ thị. Axes không có khả năng gọi hàm.

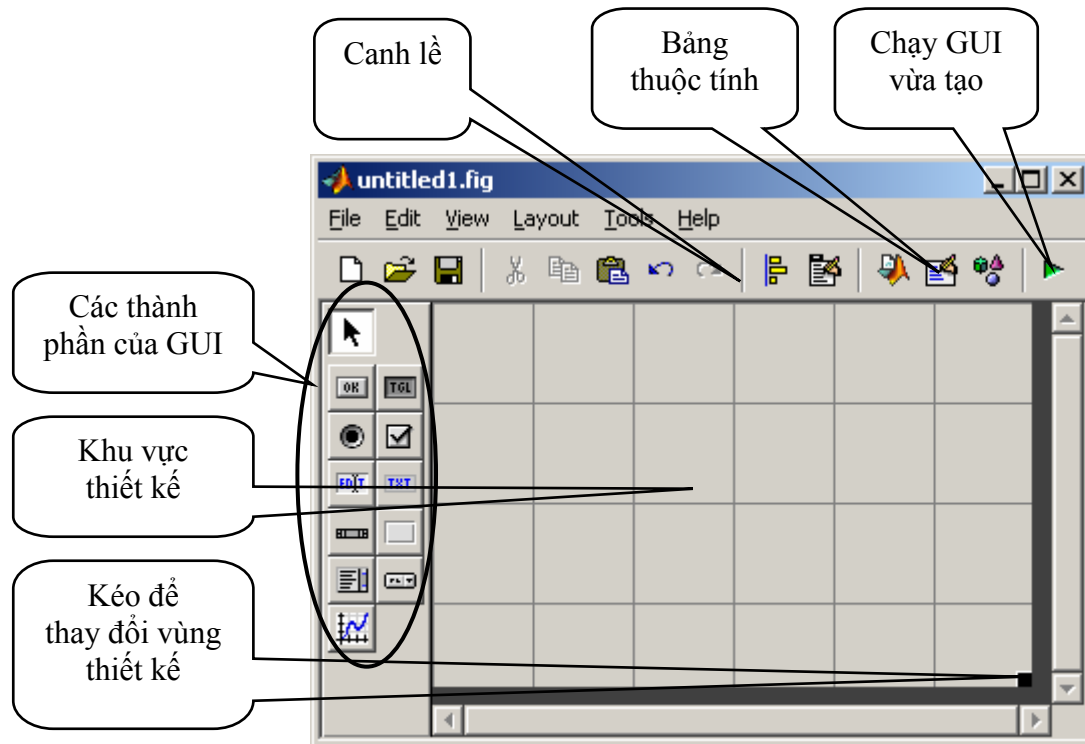


Hình 8.1 Một số thành phần tạo nên giao diện trong MATLAB.

8.2. TẠO VÀ HIỂN THỊ MỘT GUI

Trong MATLAB, công cụ **guide** được sử dụng để tạo ra các GUI, công cụ này cho phép bố trí, lựa chọn và sắp xếp các thành phần. Các thuộc tính của mỗi thành phần: tên, màu sắc, kích cỡ, font chữ, ... đều có thể được thay đổi. Công cụ **guide** được thực thi bằng cách chọn **File – New – GUI**, khi được gọi **guide** sẽ tạo ra một **Layout Editor**, như hình 8.2. Vùng màu xám có những đường kẻ là vùng làm việc, trong vùng này chúng ta có thể sắp xếp các thành phần để tạo nên giao diện. Ở bên trái vùng làm việc là các thành phần có sẵn trong GUI. Chúng ta có thể tạo ra bất kỳ thành phần nào bằng cách click vào biểu tượng của nó, sau đó kéo vào thả vào vùng làm việc. Bên trên vùng làm việc là thanh công cụ bao gồm các công cụ thường sử dụng. Sau đây là các bước cơ bản để tạo ra một giao diện trong MATLAB:

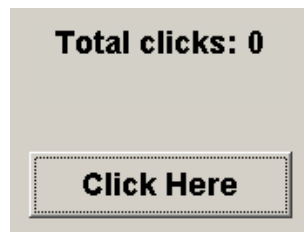
- Xác định các thành phần của giao diện và hàm bị tác động bởi mỗi thành phần. Phát họa trên giấy vị trí của các thành phần.
- Mở công cụ tạo GUI (**File – New – GUI**), sắp xếp các thành phần vào vùng làm việc. Nếu cần thiết, thay đổi kích thước của vùng làm việc cũng như của mỗi thành phần.
- Thiết lập thuộc tính của mỗi thành phần chẳng hạn như: tên, màu sắc, hiển thị, ...
- Lưu giao diện vừa tạo. Khi lưu một giao diện MATLAB sẽ tạo ra hai file có cùng tên nhưng khác phần mở rộng. File có phần mở rộng **.fig** chứa nội dung của giao diện, trong khi file có phần mở rộng **.m** chứa những đoạn mã liên quan đến giao diện.
- Viết hàm để thực thi lệnh gọi của mỗi thành phần trong giao diện.



Hình 8.2 Cửa sổ tạo giao diện

Để hiểu rõ hơn hãy xét ví dụ sau đây. Trong ví dụ này, giao diện đơn giản chỉ gồm một nút nhấn và một chuỗi ký tự. Mỗi khi nhấn vào nút nhấn dòng ký tự sẽ được cập nhật và hiển thị số lần nhấn vào nút nhấn.

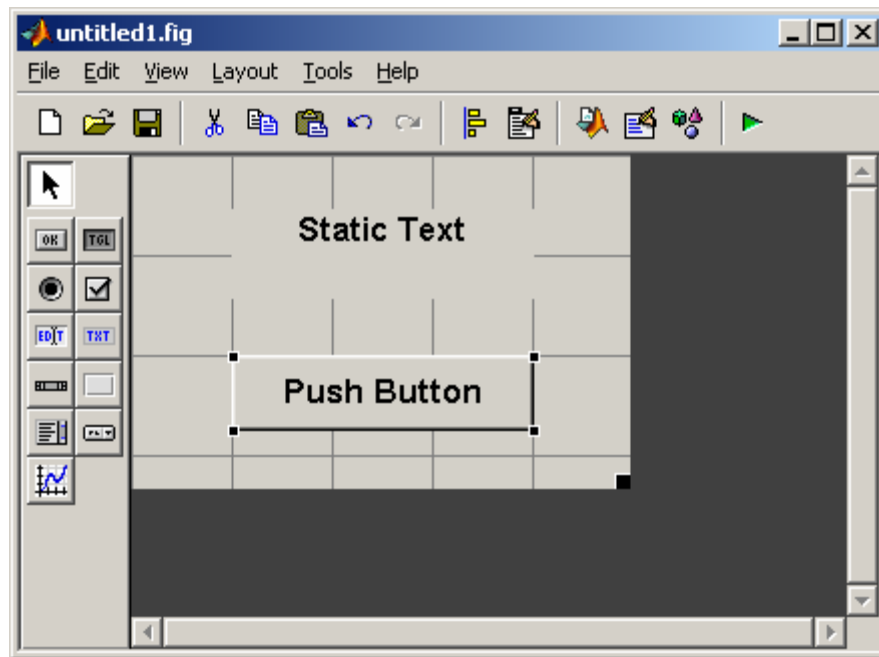
Bước 1: Trong ví dụ này, giao diện chỉ bao gồm một nút nhấn và một dòng ký tự. Hàm được gọi từ nút nhấn sẽ cập nhật chuỗi ký tự và thể hiện số lần nhấn vào nút nhấn. Phác họa giao diện như **hình 8.3**.



Hình 8.3 Phác họa của một giao diện đơn giản

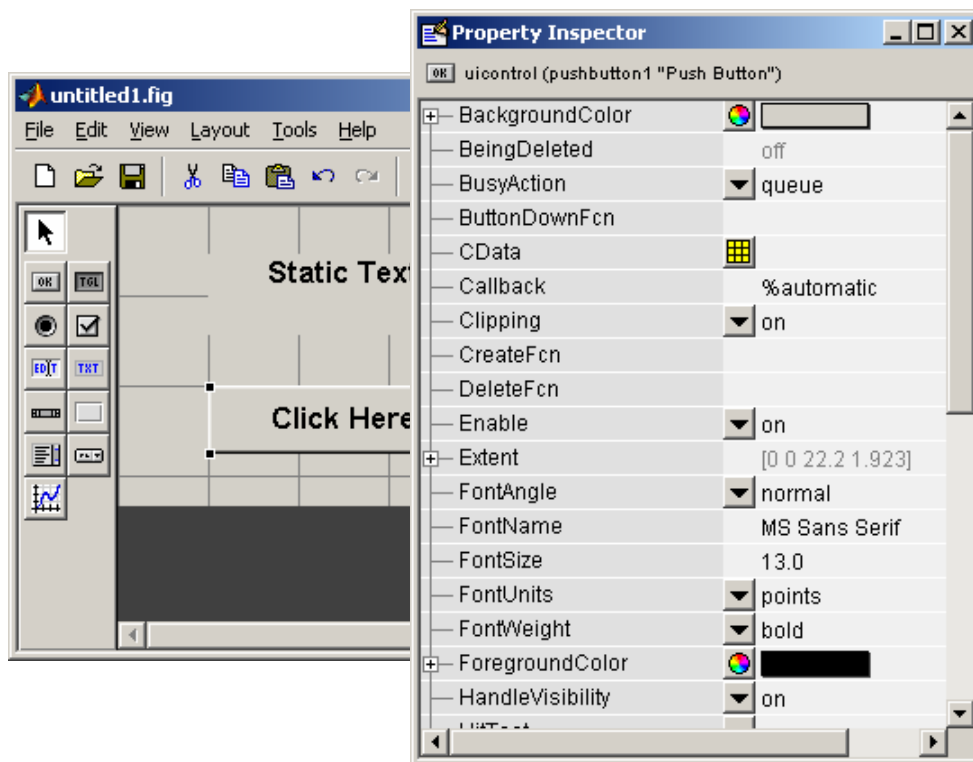
Bước 2: Mở chương trình **guide** để bắt đầu bố trí các thành phần vào cửa sổ làm việc. Chương trình *guide* sẽ mở ra một cửa sổ như **hình 8.2**.

Đầu tiên chúng ta phải thay đổi kích thước của vùng làm việc, đây chính là kích thước của toàn bộ giao diện. Kích thước của vùng làm việc được thay đổi bằng cách kéo hình vuông nhỏ ở góc dưới bên trái của cửa sổ làm việc. Sau đó chúng ta tạo ra một nút nhấn bằng cách nhấn vào biểu tượng nút nhấn, kéo và thả vào vùng làm việc. Làm tương tự để tạo ra một chuỗi ký tự. Sau khi thực hiện bước này ta có kết quả như sau:



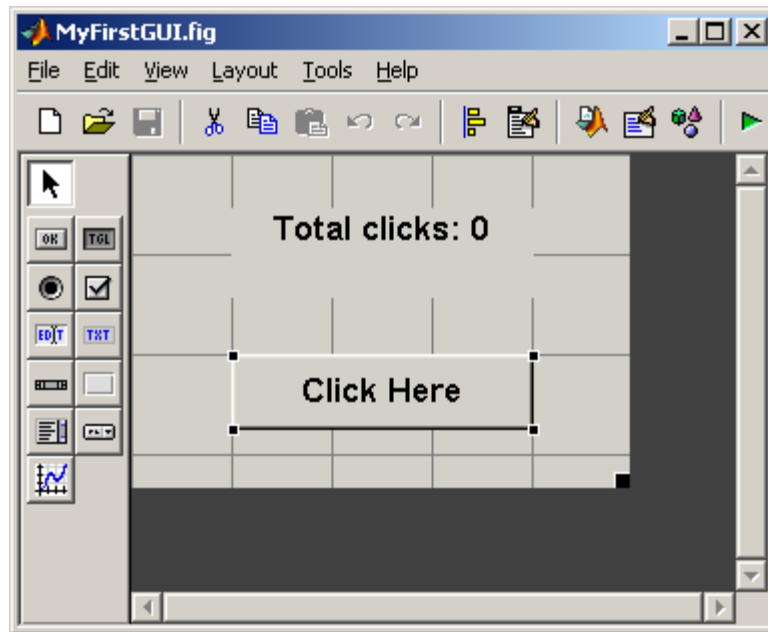
Hình 8.4 Cửa sổ thể hiện nội dung của giao diện

Bước 3: Để thiết lập thuộc tính cho nút nhấn, đầu tiên ta phải chọn nút nhấn cần thiết lập các thuộc tính, sau đó chọn mục **“Property Inspector”** trong thanh công cụ. Ta cũng có thể tiến hành như sau, right-click vào đối tượng và chọn mục **“Inspect Properties”**. Cửa sổ **“Property Inspector”** được thể hiện như hình 8.5. Cửa sổ này liệt kê tất cả các thuộc tính của nút nhấn và cho phép chúng ta thay đổi giá trị của các thuộc tính này.



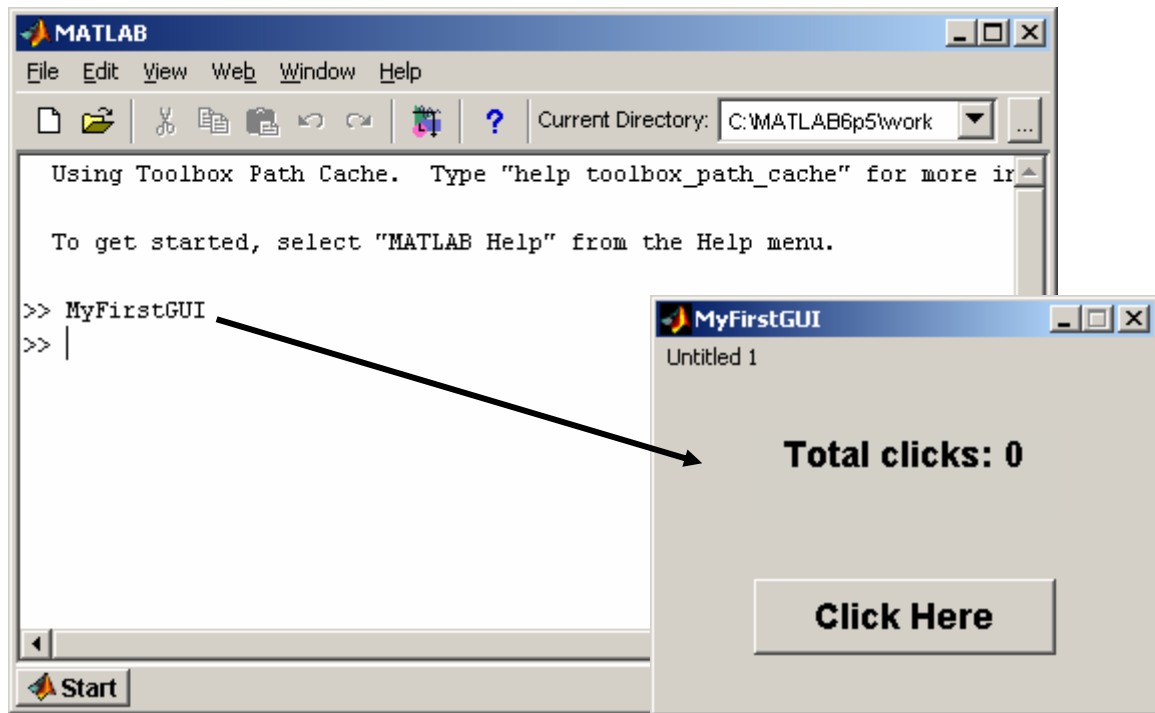
Hình 8.5 Cửa sổ thể hiện các thuộc tính của nút nhấn.

Với nút nhấn chúng ta có thể thay đổi nhiều thuộc tính chẳng hạn như màu sắc, kích cỡ, font chữ, Tuy nhiên chúng ta cần phải xác lập hai thuộc tính **String property**, là dòng ký tự xuất hiện trên nút nhấn và thuộc tính thứ hai cần xác lập là **Tag property**, là tên của nút nhấn. Trong trường hợp này **String property** được thiết lập là 'Click Here' và **Tag property** được thiết lập là 'MyFirstButton'. Đối với đối tượng chuỗi ký tự, chúng ta cũng thiết lập hai thuộc tính: **String property** là chuỗi ký tự xuất hiện trên giao diện và **Tag property** là tên của đối tượng chuỗi. Tên của đối tượng chuỗi là yêu cầu cần thiết trong quá trình gọi hàm để cập nhật nội dung của chuỗi. Trong trường hợp này **String property** được thiết lập là chuỗi 'Total clicks: 0' và **Tag property** được thiết lập là 'MyFirstText'. Sau khi thực hiện các bước ở trên ta có hình 8.6.

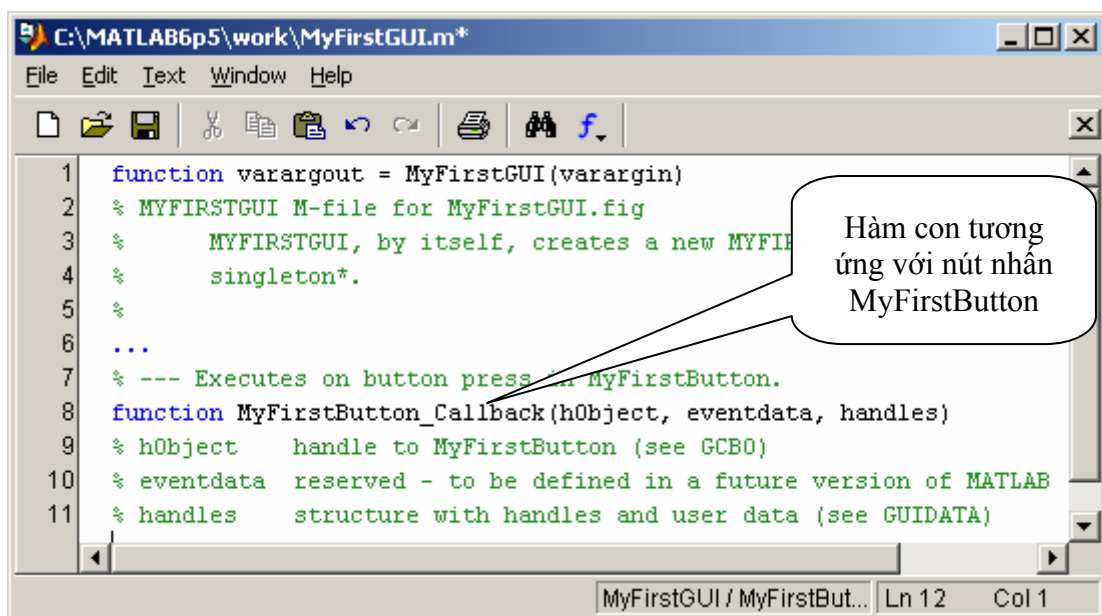


Hình 8.6 Vùng thiết kế sau khi cài đặt các thuộc tính.

Bước 4: Lưu giao diện vừa tạo với tên MyFirstGUI. Sau khi lưu chương trình sẽ tạo ra hai file **MyFirstGUI.fig** và **MyFirstGUI.m**. Đến đây chúng ta đã tạo xong giao diện nhưng chưa hoàn thành bài tập như ý tưởng ban đầu. Chúng ta có thể bắt đầu chương trình bằng cách gõ lệnh **MyFirstGUI** trong cửa sổ lệnh, kết quả như hình 8.7. Chương trình sẽ không có hoạt động gì khi ta nhấn vào nút nhấn bởi vì ta chưa lập trình cho những hàm được gọi. Một phần của file **MyFirstGUI.m** được trình bày trong hình 8.8. File này chứa hàm **MyFirstGUI** và một số hàm con tương ứng với tác động của mỗi thành phần trong giao diện. Nếu gọi hàm **MyFirstGUI** trong trường hợp không có đối số ngõ vào thì nội dung của file **MyFirstGUI.fig** được thể hiện. Trong trường hợp có đối số ngõ vào khi gọi hàm **MyFirstGUI** thì đối số đầu tiên là tên của hàm con và các đối số còn lại sẽ được đưa đến hàm con đó. Mỗi hàm con được gọi sẽ tương ứng với một đối tượng trong giao diện. Khi click chuột vào một đối tượng thì MATLAB sẽ gọi hàm con tương ứng với đối tượng đó. Tên của hàm được gọi sẽ là giá trị **Tag property** của đối tượng tương ứng. Tên của hàm chính là tên của đối tượng cộng với chuỗi ký tự **'_Callback'**. Như vậy hàm con tương ứng với nút nhấn **MyFirstButton** sẽ là **MyFirstButton_Callback**. File .m của chương trình sẽ tạo ra các hàm con tương ứng với tất cả các đối tượng.

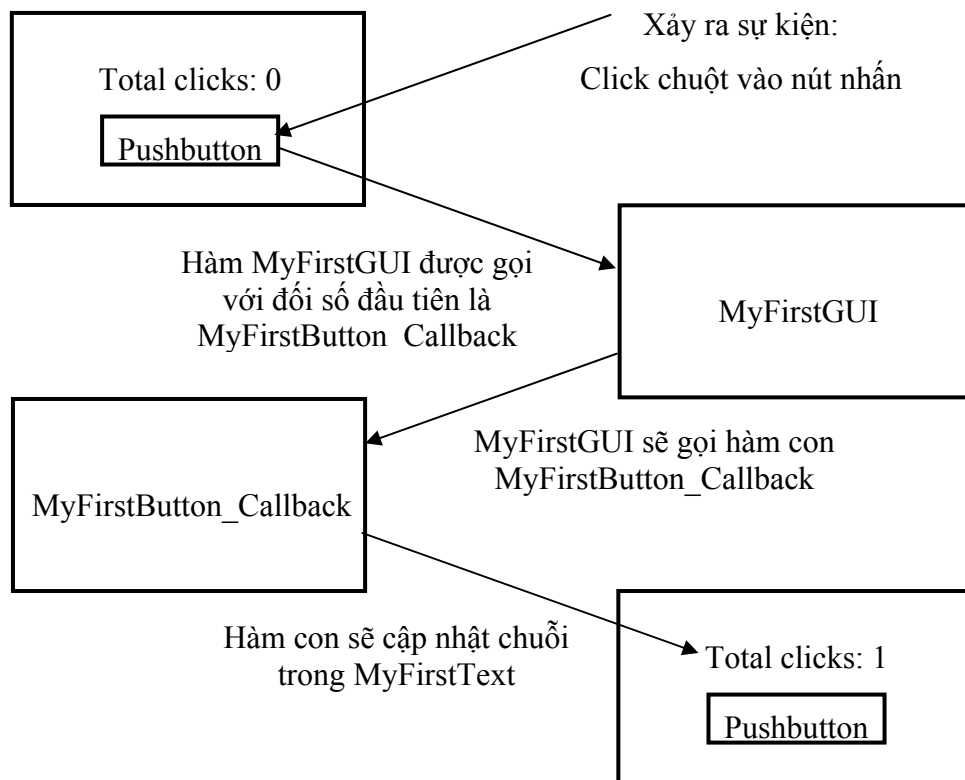


Hình 8.7 Gõ MyFirstGUI trong cửa sổ lệnh để bắt đầu chương trình



Hình 8.8 Một phần nội dung của file .m

Bước 5: Trong bước này, chúng ta sẽ lập trình cho hàm con tương ứng với nút nhấn. Trong hàm này chúng ta sẽ sử dụng một biến **persistent** (là biến mà giá trị của nó sẽ được nhớ giữa những lần gọi hàm) để đếm số click chuột vào nút nhấn. Khi có click chuột vào nút nhấn, MATLAB sẽ gọi hàm **MyFirstGUI** với đối số đầu tiên là **MyFirstButton_callback**. Sau đó hàm **MyFirstGUI** sẽ gọi hàm con **MyFirstButton_callback**, quá trình này được thể hiện trong **hình 8.9**. Trong hàm con, giá trị biến đếm sẽ tăng lên khi có click chuột, một chuỗi mới chứa giá trị biến đếm sẽ tạo ra và được lưu vào trong thuộc tính **String property** của đối tượng **MyFirstText**.



Hình 8.9 Quá trình gọi hàm và cập nhật giá trị đếm

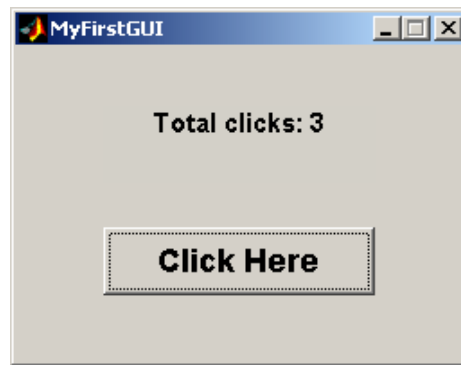
Nội dung của hàm con được lập trình như sau:

```
function MyFirstButton_Callback(hObject, eventdata, handles)

% Khai báo và thiết lập giá trị ban đầu cho biến đếm count
persistent count
if isempty(count)
    count = 0;
end
% Cập nhật giá trị biến đếm
count = count + 1;
% Tạo một chuỗi mới
str = sprintf('Total clicks: %d',count);
% Cập nhật giá trị của đối tượng chuỗi
set(handles.MyFirstText,'String',str);
```

Lưu ý rằng hàm con khai báo biến **count** có dạng biến “nhỏ” và gán giá trị ban đầu cho nó là zero.

Kết quả của giao diện sau khi click chuột ba lần vào nút nhấn như **hình 8.10**.



Hình 8.10 Kết quả sau khi nhấn nút ba lần

8.3. THUỘC TÍNH CỦA CÁC ĐỐI TƯỢNG

Mỗi đối tượng của GUI đều có một số thuộc tính mà người sử dụng có thể thay đổi tùy ý. Các thuộc tính có một số điểm khác nhau tùy vào từng đối tượng (figures, axes, uicontrol, ...). Đặc tính và cách ứng dụng của tất cả các thuộc tính này đều được đề cập đến một cách chi tiết trong phần **Help** của MATLAB. Bảng 8.2 và bảng 8.3 liệt kê một số đặc tính quan trọng nhất của **figure** và **uicontrol**.

Giá trị của các thuộc tính này có thể thay đổi bằng **Property Inspector** hoặc thay đổi trực tiếp bằng cách sử dụng lệnh **get** hay **set**. Trong trường hợp thiết kế một giao diện ta nên sử dụng **Property Inspector** để thay đổi thuộc tính của các đối tượng còn trong phạm vi của một chương trình thì chúng ta phải sử dụng lệnh **get** hay **set**.

Bảng 8.2 Một số thuộc tính cơ bản của figure

Tên thuộc tính	Miêu tả	Giá trị
Position	Vị trí và kích thước của figure	<i>Value</i> : là một vector gồm 4 thành phần [left, bottom, width, height] <i>Default</i> : phụ thuộc vào trạng thái hiện tại
Units	Đơn vị của các thành phần trong <i>Position</i> .	<i>Values</i> : inches, centimeters, normalized, points, pixels, characters <i>Default</i> : pixels
Color	Màu nền của figure.	<i>Values</i> : ⁽¹⁾ <i>Default</i> : phụ thuộc vào màu hiện tại của hệ thống.
MenuBar	Tắt hay hiện thanh công cụ	<i>Values</i> : none, figure <i>Default</i> : figure
Name	Tiêu đề của figure	<i>Values</i> : chuỗi <i>Default</i> : " (chuỗi rỗng)
NumberTitle	Hiển thị "Figure No. n", với n là số thứ tự figure	<i>Values</i> : on, off <i>Default</i> : on
Resize	Cửa sổ hình có thể thay đổi kích thước được hay không.	<i>Values</i> : on, off <i>Default</i> : on
Pointer	Chọn kiểu hình ảnh dấu nhấn chuột	<i>Values</i> : crosshair, arrow, watch, topl, topr, botl, botr, circle, cross, fleur, left, right, top, bottom, fullcrosshair, ibeam, custom <i>Default</i> : arrow

Bảng 8.3 Một số thuộc tính cơ bản của uicontrol

Tên thuộc tính	Miêu tả	Giá trị
BackgroundColor	Màu nền của đối tượng	<i>Value:</i> ⁽¹⁾ <i>Default:</i> phụ thuộc vào hệ thống
ForegroundColor	Màu chữ	<i>Value:</i> ⁽¹⁾ <i>Default:</i> [0 0 0]
String	Nhãn của Uicontrol	<i>Value:</i> string
Enable	Enable hay disable uicontrol	<i>Value:</i> on, inactive, off <i>Default:</i> on
Style	Kiểu của đối tượng	<i>Value:</i> pushbutton, togglebutton, radiobutton, checkbox, edit, text, slider, frame, listbox, popupmenu <i>Default:</i> pushbutton
Position	Kích thước và vị trí của đối tượng	<i>Value:</i> ma trận 1x4 <i>Default:</i> [20 20 60 20]
Units	Đơn vị trong vector vị trí	<i>Value:</i> pixels, normalized, inches, centimeters, points, characters <i>Default:</i> pixels
FontAngle	Trạng thái của ký tự	<i>Value:</i> normal, italic, oblique <i>Default:</i> normal
FontName	Font chữ	<i>Value:</i> chuỗi <i>Default:</i> phụ thuộc vào hệ thống
FontSize	Kích thước chữ	<i>Value:</i> kích thước trong FontUnits <i>Default:</i> phụ thuộc vào hệ thống
FontWeight	Độ đậm nhạt của ký tự	<i>Value:</i> light, normal, demi, bold <i>Default:</i> normal
HorizontalAlignment	Canh lề	<i>Value:</i> left, center, right <i>Default:</i> phụ thuộc vào đối tượng uicontrol
Callback	Hoạt động điều khiển	Value: <i>string</i>
Max	Giá trị lớn nhất	<i>Value:</i> vô hướng <i>Default:</i> phụ thuộc đối tượng
Min	Giá trị nhỏ nhất	<i>Value:</i> vô hướng <i>Default:</i> phụ thuộc đối tượng
Value	Giá trị hiện tại của đối tượng	<i>Value:</i> vô hướng hay vector <i>Default:</i> phụ thuộc đối tượng

8.4. CÁC THÀNH PHẦN TẠO NÊN GUI

Phần này miêu tả cách tạo lập và sử dụng các thành phần đối tượng trong GUI:

- Text Fields

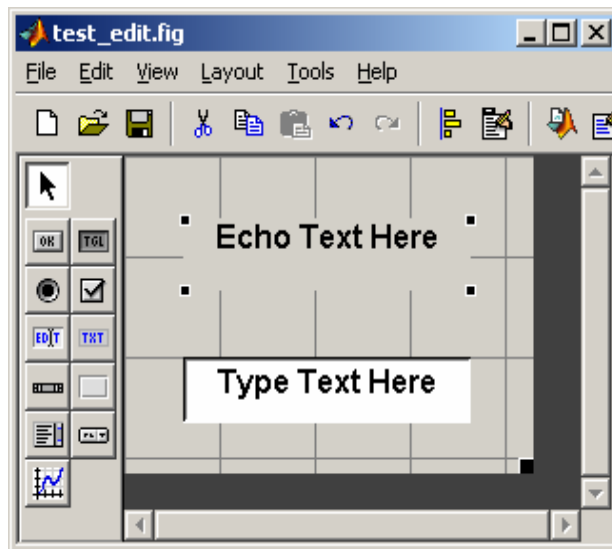
- Edit Boxes
- Frames
- Pushbuttons
- Toggle Buttons
- Checkboxes
- Radio Buttons
- Popup Menus
- List Boxes
- Slide

8.4.1. Text Fields

Text Field là một đối tượng được sử dụng để thể hiện chuỗi ký tự trên màn hình. Chúng ta có thể xác định thuộc tính canh lề của chuỗi ký tự. Mặc định, chuỗi ký tự sẽ được canh ở giữa. Text Field được tạo ra bằng cách sử dụng hàm **uicontrol** với thuộc tính **style** là 'text' hay cũng có thể tạo ra bởi công cụ **text** trong Layout Editor. Text Field không có khả năng gọi hàm nhưng giá trị của nó có thể được cập nhật trong một hàm bằng cách thay đổi thuộc tính **String** của Text Field, như trong phần 2.

8.4.2. Edit Boxes

Edit Box là một đối tượng cho phép người sử dụng nhập vào một chuỗi ký tự. Sau khi nhập xong chuỗi ký tự và nhấn Enter Edit Box sẽ gọi hàm tương ứng của nó. Đối tượng này được tạo ra bằng cách sử dụng hàm **uicontrol** với thuộc tính **style** là 'edit' hay cũng có thể tạo ra bởi công cụ **edit box** trong Layout Editor. **Hình 8.11a** thể hiện một giao diện đơn giản gồm một Edit Box có tên 'EditBox' và một Text Field có tên là 'TextBox'. Khi người sử dụng nhập vào khung Edit Box thì nó tự động gọi hàm **EditBox_Callback** như **hình 8.11b**. Chương trình này có nhiệm vụ xác định chuỗi ký tự được nhập vào, sau đó gán chuỗi này cho Text Field và hiển thị lên màn hình giao diện. **Hình 8.12** là giao diện của chương trình khi bắt đầu và sau khi nhập 'Hello' vào Edit Box.



(a)

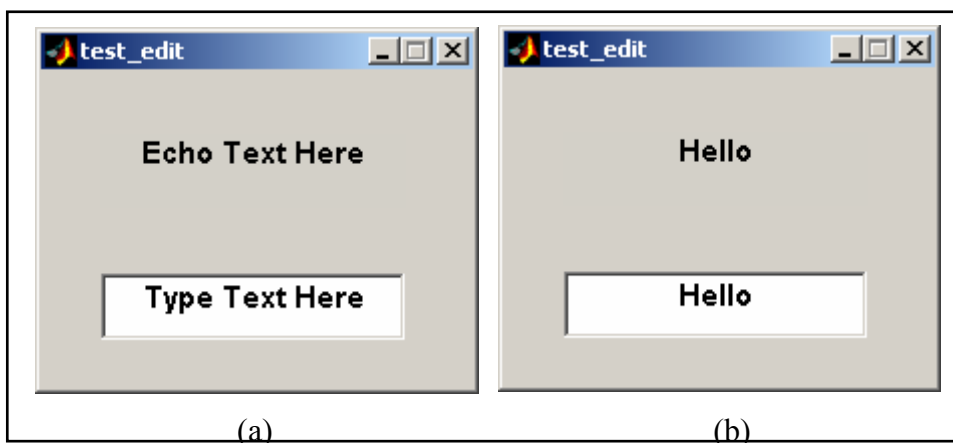
```
function EditBox_Callback(hObject, eventdata, handles)
```

```
% Xác định giá trị gõ vào Edit Box
```

```
str=get(handles.EditBox,'String');
```

```
% Gán giá trị vào Text Field
```

Hình 8.11 Ví dụ về Edit Box và nội dung của chương trình con được gọi



(a)

(b)

Hình 8.12 Giao diện của chương trình trước và sau khi gõ 'Hello' vào Edit Box

8.4.3. Frames

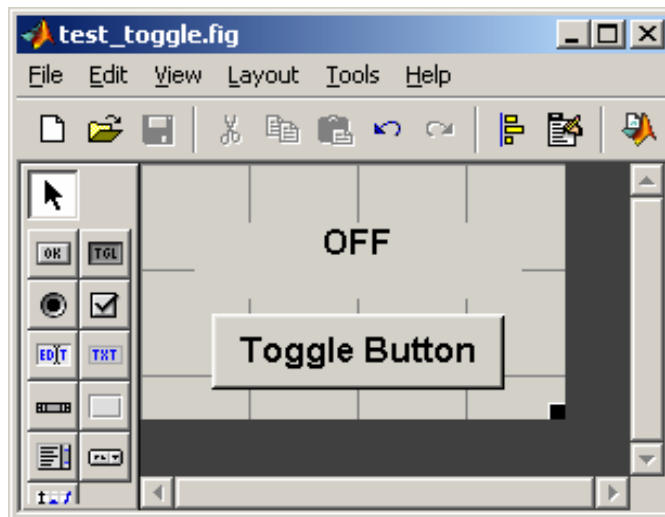
Frame là một đối tượng được sử dụng để vẽ hình chữ nhật, nó còn được sử dụng để nhóm các đối tượng có liên quan lại với nhau. Ví dụ một Frame được sử dụng để nhóm các Radio Button trong **hình 8.1** lại với nhau. Một Frame được tạo bằng cách sử dụng hàm **uicontrol** với đối tượng style là 'frame' hay cũng có thể tạo ra bởi công cụ **frame** trong Layout Editor.

8.4.4. Pushbuttons

PushButton là một đối tượng cho phép người sử dụng kích hoạt một tác vụ bằng cách click chuột vào nó. Đối tượng này được tạo ra bởi hàm **uicontrol** với thuộc tính style là 'pushbutton' hay cũng có thể tạo ra bởi công cụ **pushbutton** trong Layout Editor. Một ví dụ về hoạt động của PushButton được miêu tả bởi hàm MyFirstGUI trong **hình 8.10**.

8.4.5. Toggle Buttons

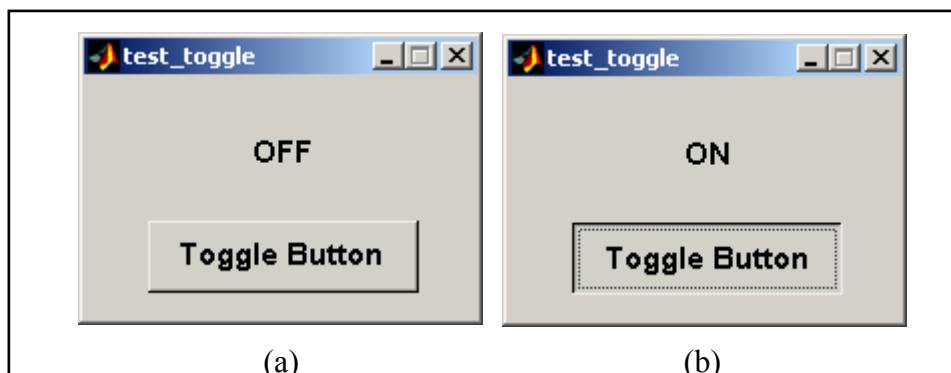
Toggle Button là một kiểu nút nhấn có hai trạng thái **on** và **off** tương ứng với trường hợp bị nhấn xuống hay không bị nhấn xuống. Mỗi khi có click chuột vào nó, Toggle Button sẽ thay đổi trạng thái và gọi hàm tương ứng của nó. Thuộc tính **Value** của Toggle Button được xác định bởi giá trị lớn nhất (thường là 1) khi nó ở trạng thái **on** và được xác định bởi giá trị nhỏ nhất (thường là 0) khi nó ở trạng thái **off**. Kiểu nút nhấn này được tạo ra bằng hàm **uicontrol** với thuộc tính style là 'toggle button' hay cũng có thể tạo ra bởi công cụ **toggle button** trong Layout Editor. **Hình 8.13a** trình bày một giao diện đơn giản bao gồm một Toggle Button tên 'ToggleButton' và một Text Field tên 'TextBox'. Khi người sử dụng click vào Toggle Button nó sẽ tự động gọi hàm ToggleButton_Callback như trong **hình 8.13b**. Hàm này xác định trạng thái của nút nhấn từ thuộc tính 'Value', sau đó chương trình sẽ thể hiện trạng thái của nút nhấn bởi Text Field. **Hình 8.14** thể hiện giao diện trước và sau khi có click chuột vào Toggle Button.



(a)

```
function ToggleButton_Callback(hObject, eventdata, handles)
% Xác định trạng thái của toggle button
state = get(handles.ToggleButton,'Value');
% Gán giá trị vào Text Field
if state == 0
    set(handles.TextBox,'String','OFF');
else
    set(handles.TextBox,'String','ON');
end
```

(b)

Hình 8.13 Ví dụ về Toggle Button và nội dung của chương trình con được gọi

(a)

(b)

Hình 8.14 Hai trạng thái ON và OFF của Toggle Button

8.4.6. Checkboxes và Radio Buttons

Về cơ bản Checkbox và Radio Button là tương tự với Toggle Button ngoại trừ hình dạng khác nhau của chúng. Giống như Toggle Button, Checkbox và Radio Button cũng có hai trạng thái ON và OFF. Trạng thái của chúng sẽ thay đổi mỗi khi có click chuột và một hàm tương ứng sẽ được gọi. Thuộc tính **Value** của Checkbox và Radio Button được xác định bởi giá trị lớn

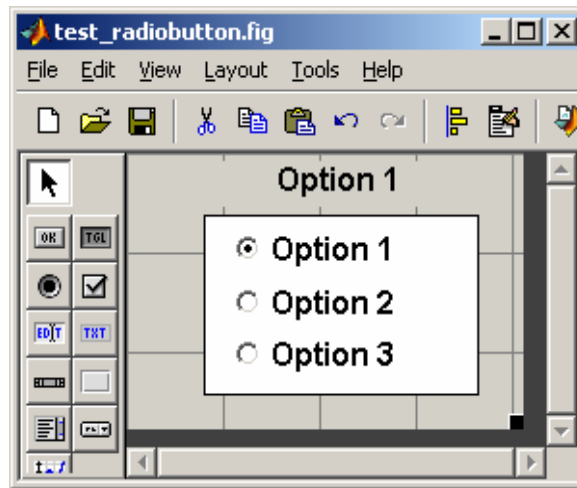
nhất (thường là 1) khi nó ở trạng thái **on** và được xác định bởi giá trị nhỏ nhất (thường là 0) khi nó ở trạng thái **off**. Hình dạng của Checkbox và Radio Button được mô tả trong **hình 8.1**.

Hai đối tượng này có thể được tạo ra từ Layout Editor hay sử dụng hàm **uicontrol** với thuộc tính style tương ứng là `'checkbox'` và `'radiobutton'`. Thông thường CheckBox được sử dụng trong các lựa chọn on/off và một nhóm Radio Button được sử dụng để xác định một trong số các lựa chọn. **Hình 8.15a** thể hiện một ví dụ về cách sử dụng Radio Button. Trong ví dụ này, chương trình sẽ tạo ra ba Radio Button có nhãn là `'Option 1'`, `'Option 2'` và `'Option 3'`. **Hình 8.15b** thể hiện các hàm con được gọi. Khi người sử dụng click vào một Radio Button thì hàm con tương ứng sẽ được thực thi, hàm này sẽ thể hiện đối tượng được lựa chọn, mở Radio Button hiện tại và tắt tất cả các Radio Button khác. Lưu ý rằng một Frame được sử dụng để nhóm các Radio Button này lại với nhau. **Hình 8.16** thể hiện giao diện khi đối tượng thứ hai được chọn.

8.4.7. Popup Menus

Popup Menu cho phép người sử dụng chọn một giá trị trong số các lựa chọn. Danh sách các lựa chọn được xác định bởi một mảng của các chuỗi. Giá trị của thuộc tính `'Value'` thể hiện kết quả lựa chọn hiện tại. Một Popup Menu có thể được tạo ra bởi công cụ **popup menu** trong Layout Editor.

Hình 8.17a thể hiện một ví dụ của popup menu. Trong ví dụ này, một popup menu được tạo ra với năm trường hợp được gắn nhãn `'Option 1'`, `'Option 2'`, **Hình 8.17b** thể hiện nội dung của hàm được gọi tương ứng, hàm này sẽ xác định giá trị được chọn trong đối số `'Value'` và hiển thị giá trị được chọn. **Hình 8.18** là trường hợp Option 5 được chọn.



(a)

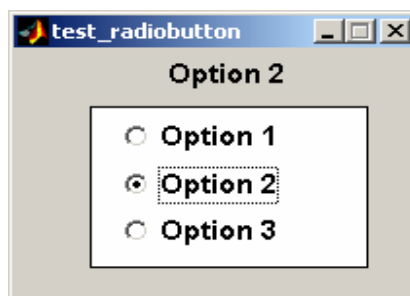
```
function radiobutton1_Callback(hObject, eventdata, handles)
set(handles.TextBox,'String','Option 1');
set(handles.radiobutton1,'Value',1);
set(handles.radiobutton2,'Value',0);
set(handles.radiobutton3,'Value',0);
```

```
function radiobutton2_Callback(hObject, eventdata, handles)
set(handles.TextBox,'String','Option 2');
set(handles.radiobutton1,'Value',0);
set(handles.radiobutton2,'Value',1);
set(handles.radiobutton3,'Value',0);
```

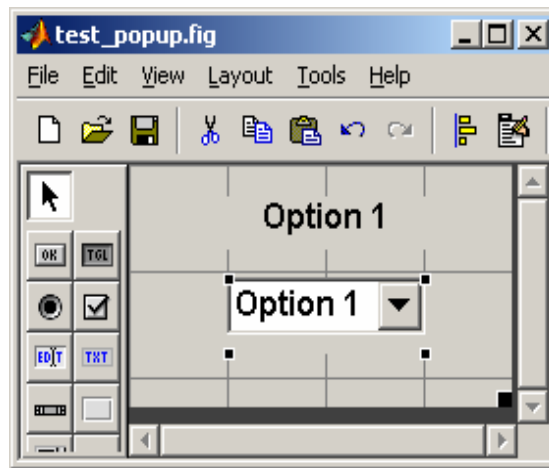
```
function radiobutton3_Callback(hObject, eventdata, handles)
set(handles.TextBox,'String','Option 3');
set(handles.radiobutton1,'Value',0);
set(handles.radiobutton2,'Value',0);
set(handles.radiobutton3,'Value',1);
```

(b)

Hình 8.15 Ví dụ về Radio Button và nội dung của chương trình con được gọi



Hình 8.16 Giao diện của chương trình **test_radiobutton**



(a)

```
function popupmenu1_Callback(hObject, eventdata, handles)
```

```
% Xác định giá trị của popup menu
value=get(handles.popupmenu1,'Value');
% Gán giá trị mới cho text box
str=['Option ',num2str(value)];
set(handles.TextBox,'String',str)
```

(b)

Hình 8.17 Ví dụ về Pupop Menu và nội dung của chương trình con được gọi



Hình 8.18 Giao diện của chương trình **test_popup**

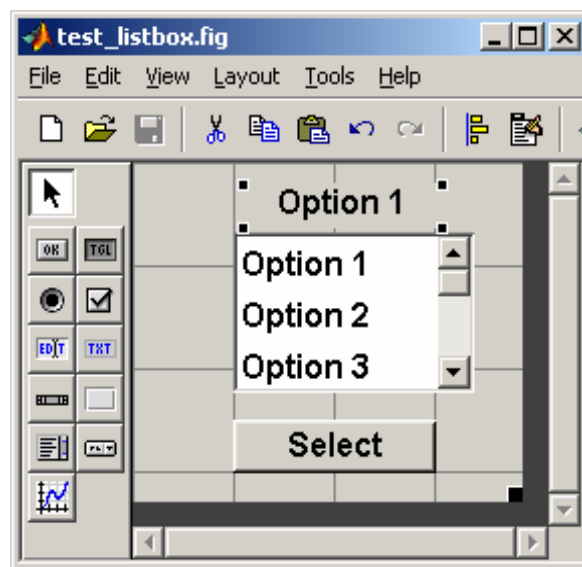
8.4.8. List Boxes

List Box tạo ra một đối tượng có nhiều dòng ký tự và cho phép người sử dụng chọn một hay nhiều dòng trong số các dòng này. Nếu số dòng ký tự của đối tượng lớn hơn phạm vi của List Box thì một thanh cuộn sẽ được tạo ra cho phép người sử dụng di chuyển lên xuống trong phạm vi của List Box. Những dòng ký tự mà người sử dụng có thể lựa chọn được xác định bởi một mảng các chuỗi và thuộc tính 'Value' sẽ xác định dòng nào được chọn. Một List Box được tạo ra bởi hàm **uicontrol** với thuộc tính style là 'listbox' hoặc được tạo ra từ công cụ listbox trong Layout Editor.

Trong trường hợp sử dụng một single-click để chọn một thành phần trong List Box thì chương trình sẽ không gây ra bất kỳ phản ứng nào cho đến khi có một tác động bên ngoài xảy ra, chẳng hạn như một nhấn nút. Tuy nhiên trong trường hợp sử dụng double-click thì có thể

gây ra một tác động ngay lập tức. Có thể sử dụng thuộc tính `SelectionType` của figure để phân biệt single-click hay double-click. Giá trị của thuộc tính `SelectionType` sẽ là 'normal' tương ứng với single-click và trong trường hợp double-click thì giá trị này sẽ là 'open'. Điều này cũng đúng trong trường hợp chọn nhiều thành phần cùng một lúc, là trường hợp khoảng cách giữa giá trị của hai thuộc tính **min** và **max** lớn hơn 1. Còn trong các trường hợp khác thì chỉ có duy nhất một thành phần được chọn.

Hình 8.19a thể hiện một ví dụ bao gồm một list box có tám thành phần 'Option 1', 'Option 2', ..., 'Option 8', một nút nhấn để thực thi chương trình và một text field để hiển thị kết quả.



(a)

```
function pushbutton1_Callback(hObject, eventdata, handles)
```

```
% Xác định giá trị của list box
value=get(handles.listbox1,'Value');
% Cập nhật và hiển thị kết quả
str=['Option ',num2str(value)];
set(handles.text1,'String',str);
```

```
function listbox1_Callback(hObject, eventdata, handles)
```

```
% Nếu là double-click thì cập nhật và hiển thị kết quả
selectiontype=get(gcf,'SelectionType');
if selectiontype(1)=='o'
    value=get(handles.listbox1,'Value');
    str=['Option ',num2str(value)];
    set(handles.text1,'String',str);
end
```

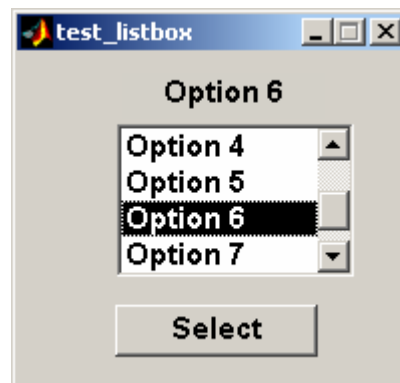
(b)

Hình 8.19 Ví dụ về List Box và nội dung của chương trình con được gọi

Hình 8.19b là những hàm được gọi tương ứng với nút nhấn và list box.

- Nếu nút nhấn được chọn thì hàm `button1_callback` sẽ xác định giá trị của List Box và thể hiện kết quả thông qua Text Field.
- Nếu list box được chọn, đầu tiên hàm `listbox_callback` sẽ xác định trạng thái click chuột là single hay double. Nếu là single-click thì chương trình sẽ chương trình không làm gì cả, còn trong trường hợp là double-click thì hàm `listbox_callback` sẽ xác định giá trị của List Box và thể hiện kết quả thông qua Text Field.

Giao diện của ví dụ này được thể hiện trong **hình 8.20**.



Hình 8.20 Giao diện của chương trình `test_listbox`

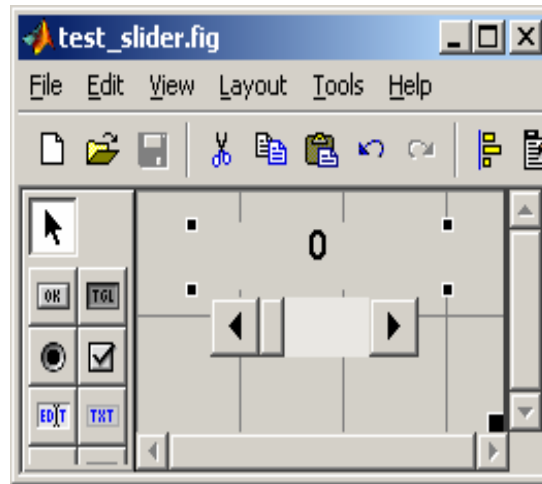
8.4.9. Sliders

Slider là một đối tượng cho phép người sử dụng lựa chọn giá trị trong một khoảng xác định bằng cách dùng chuột di chuyển thanh trượt. Thuộc tính `'Value'` sẽ xác định giá trị của Slider trong khoảng `[min max]`. Đối tượng này có thể được tạo ra bởi hàm `uicontrol` với thuộc tính style là `'slider'` hoặc được tạo ra bằng công cụ `slider` trong Layout Editor.

Hình 8.22a thể hiện một ví dụ đơn giản chỉ chứa một slider và một text field. Thiết lập giá trị của thuộc tính `min` là zero và giá trị của thuộc tính `max` là mười. Khi kéo thanh trượt, chương trình sẽ gọi hàm `slider1_callback`, hàm này sẽ xác định giá trị của slider từ thuộc tính `'value'` và thể hiện giá trị này thông qua text field.



Hình 8.21 Giao diện của chương trình `test_slider`



(a)

```
function slider1_Callback(hObject, eventdata, handles)
```

```
value=get(handles.slider1,'Value');
```

```
str=sprintf('%3.2f',value);
```

```
set(handles.text1,'String',str);
```

(b)

Hình 8.22 Ví dụ về Slider và nội dung của chương trình con được gọi

Danh sách các hàm được giới thiệu trong chương 8

axes	Tạo một hệ trục tọa độ
figure	Tạo một figure (cửa sổ giao diện) mới
get	Truy xuất giá trị của các thuộc tính của một đối tượng giao diện
guide	Mở chương trình thiết kế giao diện GUI
set	Thay đổi giá trị của các thuộc tính của một đối tượng giao diện
uicontextmenu	Tạo một đối tượng context menu
uicontrol	Tạo một đối tượng điều khiển trong giao diện với người sử dụng
uimenu	Tạo một đối tượng menu

(1)



PHẦN II

ỨNG DỤNG MATLAB TRONG XỬ LÝ TÍN HIỆU VÀ XỬ LÝ ẢNH

Chương 9

TÍN HIỆU VÀ HỆ THỐNG

Mô phỏng một hệ thống viễn thông là một quá trình làm việc với các tín hiệu: khởi tạo, biến đổi, thu phát, so sánh, xử lý,... Để việc mô phỏng được tiến hành thuận lợi và hiệu quả, cần phải có đầy đủ các công cụ xử lý tín hiệu cơ bản. Đối với MATLAB, các công cụ này được cung cấp trong MATLAB Signal Processing Toolbox. Đó là một tập hợp các hàm được xây dựng trên cơ sở các giải thuật toán học và hỗ trợ hầu hết các thao tác xử lý tín hiệu, bao gồm: khởi tạo tín hiệu, phân tích và thiết kế các bộ lọc tương tự và số, phân tích phổ, xử lý tín hiệu thống kê, phân tích các hệ thống tuyến tính,...

9.1. BIỂU DIỄN MỘT TÍN HIỆU TRONG MATLAB

▪ MATLAB là một môi trường tính toán số, do đó một tín hiệu phải được biểu diễn thông qua các mẫu dữ liệu rời rạc của nó (tín hiệu được lấy mẫu). Nói cách khác, một tín hiệu bất kỳ được biểu diễn bằng một vector hàng (kích thước $1 \times n$) hoặc vector cột (kích thước $n \times 1$). Ví dụ:

```
>> x = [1 1.2 1.5 1.2 1 0.8 0.5 0.8];
```

▪ Để biểu diễn một tín hiệu đa kênh, ta sử dụng dạng ma trận. Một tín hiệu m kênh có chiều dài ứng với mỗi kênh đều bằng n được biểu diễn bằng một ma trận kích thước $n \times m$, trong đó mỗi cột của ma trận tương ứng với một kênh, mỗi hàng của ma trận ứng với một thời điểm lấy mẫu. Ví dụ: sau đây là biểu diễn của một tín hiệu 3 kênh:

```
>> y = [x' 2*x' x'/pi]
```

y =

1.0000	2.0000	0.3183
1.2000	2.4000	0.3820
1.5000	3.0000	0.4775
1.2000	2.4000	0.3820
1.0000	2.0000	0.3183
0.8000	1.6000	0.2546
0.5000	1.0000	0.1592
0.8000	1.6000	0.2546

9.2. TẠO TÍN HIỆU: VECTOR THỜI GIAN

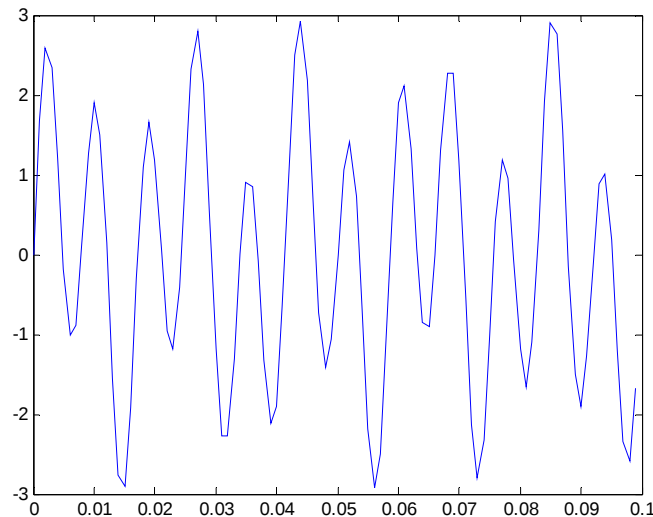
Để tạo một tín hiệu tương tự trong MATLAB, cần có một vector thời gian để xác định các thời điểm lấy mẫu tín hiệu. Tín hiệu tạo ra sẽ là một vector mà mỗi phần tử của nó chính là giá trị của mẫu tín hiệu được lấy ở thời điểm xác định bởi phần tử tương ứng của vector thời gian. Ví dụ, để tạo tín hiệu $y = \sin(100\pi t) + 2\sin(240\pi t)$, ta lấy mẫu tín hiệu tại các thời điểm cách nhau 0,001s và ta có các vector thời gian và vector biểu diễn tín hiệu y như sau:

```
>> t = (0:0.001:1)';
```

```
>> y = sin(2*pi*50*t) + 2*sin(2*pi*120*t);
```

Dùng hàm **plot** để vẽ 100 mẫu đầu tiên của y :

```
>> plot(t(1:100), y(1:100))
```



Hình 9.1

Với phương pháp trên, ta có thể khởi tạo bất kỳ tín hiệu nào ta muốn, chỉ cần xác định biểu thức thời gian của nó. Sau đây là một số tín hiệu đặc biệt:

- Các tín hiệu xung đơn vị, hàm nấc đơn vị và hàm dốc đơn vị:

```
>> t = (0:0.001:1)'; % vector thời gian
>> y = [1; zeros(99,1)]; % hàm xung đơn vị
y = ones(100,1); % hàm nấc đơn vị
y = t; % hàm dốc
```

- Các tín hiệu tuần hoàn:

```
>> fs = 10000; % tần số lấy mẫu
>> t = 0:1/fs:1.5; % vector thời gian
>> x = sin(2*pi*50*t); % Tín hiệu lượng giác (sine)
>> y = square(2*pi*50*t); % Tín hiệu sóng vuông
>> x = sawtooth(2*pi*50*t); % Tín hiệu sóng răng cưa
```

Hàm **sawtooth**(t , $width$) tạo tín hiệu sóng răng cưa hoặc sóng tam giác có các đỉnh ± 1 , chu kỳ 2π , $width$ là tỷ lệ thời gian lên trên tổng chu kỳ.

Hàm **square**(t , $width$) tạo tín hiệu sóng vuông có các mức là ± 1 , chu kỳ 2π , $width$ là tỷ lệ thời gian mức 1 trên tổng chu kỳ.

- Các tín hiệu aperiodic:

Hàm **gauspuls**(t , f_c , bw) tạo một xung Gaussian tần số RF có biên độ bằng 1, tần số trung tâm f_c và băng thông bw .

Hàm **chirp** tạo tín hiệu tần số quét, có nhiều phương pháp quét khác nhau: tuyến tính, bậc hai hay logarithm.

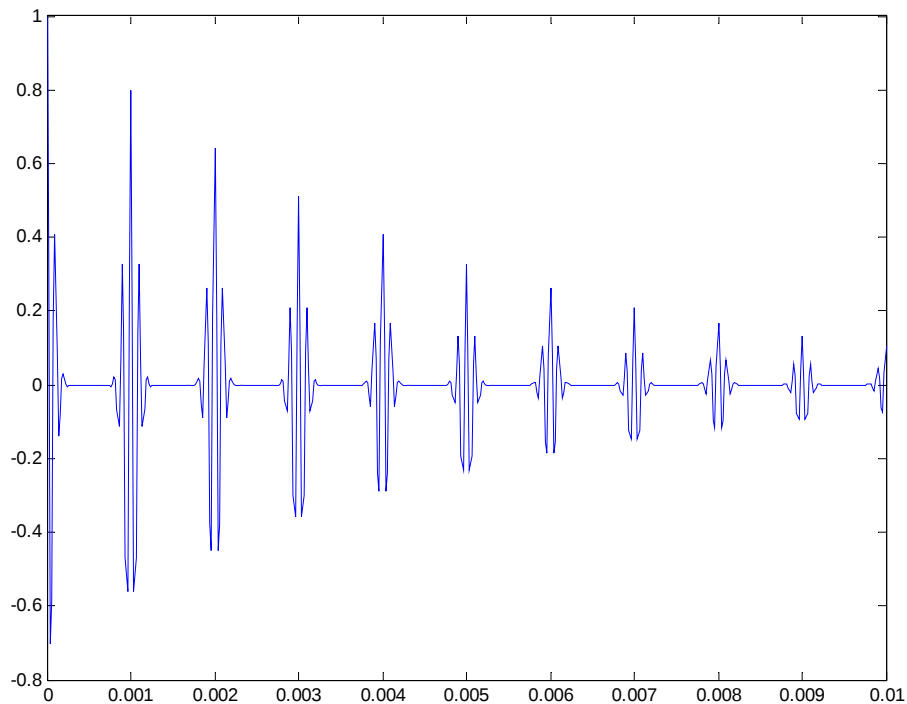
- Hàm **pulstran**: tạo một chuỗi các xung có cùng dạng với một xung gốc.

 **Ví dụ 9-1.** Tạo một chuỗi xung là sự lặp lại của các xung Gauss sau những khoảng thời gian bằng nhau. Các thông số cụ thể như sau: tốc độ lấy mẫu của chuỗi xung là 50kHz, chiều

dài chuỗi xung là 10ms, tốc độ lặp lại của chuỗi xung Gauss là 1kHz, suy hao theo hàm mũ cơ số 0.8. Xung Gauss có tần số trung tâm là 10kHz, băng thông 50%.

```
T = 0:1/50E3:10E-3; % vector thời gian của chuỗi xung (0 - 10ms)
D = [0:1/1E3:10E-3;0.8.^(0:10)]'; % cột 1 của D xác định các thời điểm lặp
% cột 2 của D xác định biên độ tương ứng của xung Gauss (bị suy hao)
Y = pulstran(T,D,'gauspuls',10E3,0.5); % gọi hàm pulstran, hai thông số cuối
% là tần số trung tâm và tỷ lệ băng thông của xung Gauss
plot(T,Y) % vẽ tín hiệu
```

Kết quả như sau:



Hình 9.2

- Hàm **sinc** (x): là biến đổi Fourier ngược của xung chữ nhật có chiều rộng bằng 2π và chiều cao bằng 1:

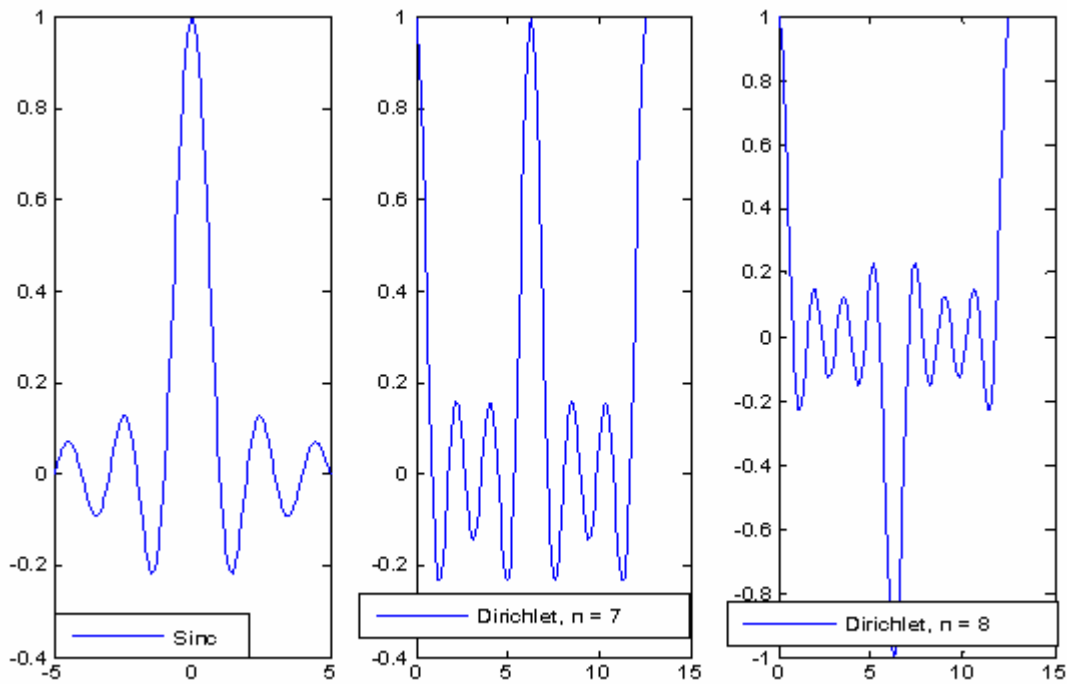
$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \quad (9.1)$$

x có thể là một vector hay một ma trận.

- Hàm Dirichlet: còn gọi là hàm sinc tuần hoàn hay hàm sinc chồng lấn (aliased sinc), được định nghĩa như sau:

$$D_n(x) = \begin{cases} (-1)^{k(n-1)} & x = 2\pi k, k \in \mathbb{Z} \\ \frac{\sin(nx/2)}{n \sin(x/2)} & \text{otherwise} \end{cases} \quad (9.2)$$

Trong MATLAB, để thực hiện hàm Dirichlet ta gọi hàm **diric** (x, n).



Hình 9.3

9.3. LÀM VIỆC VỚI CÁC FILE DỮ LIỆU

Trong các phần trên, các dữ liệu có thể được tạo ra chủ yếu bằng hai cách:

- Nhập trực tiếp từ bàn phím các giá trị dữ liệu
- Dùng các hàm có sẵn của MATLAB để tạo ra các mẫu dữ liệu.

Ngoài hai cách trên, dữ liệu còn có thể được tạo ra bằng một trong những cách sau:

- Dùng lệnh **load** của MATLAB để tải dữ liệu chứa trong các file ASCII hoặc file MAT vào không gian làm việc của MATLAB.
- Đọc dữ liệu vào MATLAB bằng cách dùng các lệnh truy xuất ngoại vi cấp thấp như **fopen**, **fread**, **fscanf**.
- Xây dựng file MEX để đọc dữ liệu.

9.4. PHÂN TÍCH VÀ THIẾT KẾ CÁC BỘ LỌC

Vấn đề phân tích và thiết kế các bộ lọc có một ý nghĩa rất quan trọng trong lý thuyết xử lý tín hiệu vì bất kỳ một hệ thống tuyến tính nào cũng có thể xem như là một bộ lọc với một đáp ứng xung hoặc một hàm truyền đạt nào đó.

- Cơ sở toán học của quá trình lọc một tín hiệu là phép lấy tích chập. Nếu $x(k)$ là tín hiệu ngõ vào và $y(k)$ là tín hiệu ngõ ra của một bộ lọc có đáp ứng xung là $h(k)$ thì $y(k)$ chính là tích chập của $x(k)$ và $h(k)$:

$$y(k) = h(k) * x(k) = \sum_{l=-\infty}^{+\infty} h(k-l)x(l) \quad (9.3)$$

Nếu $x(k)$ và $h(k)$ có chiều dài hữu hạn thì $y(k)$ cũng có chiều dài hữu hạn và phép lấy tích chập nói trên có thể thực hiện bằng cách gọi hàm **conv** trong MATLAB.

```
>> y = conv(h, x)
```

Chiều dài của vector y bằng $\text{length}(x) + \text{length}(h) - 1$.

Ngoài ra, ta cũng có thể lấy tích chập của hai ma trận bằng cách dùng hàm tích chập hai chiều **conv2**.

Ví dụ:

```
>> x = rand(5,1) % tín hiệu ngẫu nhiên chiều dài 5
>> h = [1 1 1 1]/4 % bộ lọc trung bình chiều dài bằng 4
>> y = conv(h,x)
y =
    0.2375
    0.2953
    0.4470
    0.5685
    0.5538
    0.4960
    0.3443
    0.2228
```

- Hàm truyền đạt của bộ lọc: nếu $X(z)$ là biến đổi $-z$ của tín hiệu vào $x(k)$, $Y(z)$ là biến đổi $-z$ của tín hiệu ra $y(k)$ và $H(z)$ là biến đổi $-z$ của $h(k)$ thì:

$$Y(z) = H(z).X(z) = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}} X(z) \quad (9.4)$$

$H(z)$ được gọi là hàm truyền đạt của bộ lọc. Các hằng số $a(i)$, $b(i)$ là các hệ số của bộ lọc và bậc của bộ lọc bằng $\max\{m,n\}$.

Để biểu diễn một bộ lọc, MATLAB sử dụng hai vector hàng: vector a biểu diễn các hệ số của tử số và vector b biểu diễn các hệ số của mẫu số.

Tùy theo các vector a và b mà mỗi bộ lọc có thể có các tên gọi khác nhau. Cụ thể là:

Nếu $n = 0$ (b là một vô hướng) thì bộ lọc trên gọi là bộ lọc đáp ứng xung vô hạn (IIR – Infinite Impulse Response), bộ lọc toàn cực, bộ lọc hồi quy hoặc bộ lọc AR (autoregressive).

Nếu $m = 0$ (a là một vô hướng) thì bộ lọc trên gọi là bộ lọc đáp ứng xung hữu hạn (FIR – Finite Impulse Response), bộ lọc toàn zero, bộ lọc không hồi quy hoặc bộ lọc trung bình thay đổi (MA – Moving Average).

Nếu cả m và n đều lớn hơn 0, bộ lọc trên gọi là bộ lọc đáp ứng xung vô hạn (IIR – Infinite Impulse Response), bộ lọc cực-zero, bộ lọc hồi quy hoặc bộ lọc ARMA (autoregressive moving-average)

- Từ phương trình (9.4) có thể xây dựng một quá trình để xác định các mẫu dữ liệu ra. Giả sử $a(1) = 1$. Chuyển mẫu số sang vế trái rồi lấy biến đổi $-z$ ngược cả hai vế, ta được phương trình sai phân:

$$y(k) + a(2)y(k-1) + \dots + a(m-1)y(k-m) = b(1)x(k) + b(2)x(k-1) + \dots + b(n+1)x(k-n)$$

Vậy:

$$y(k) = b(1)x(k) + b(2)x(k-1) + \dots + b(n+1)x(k-n) - a(2)y(k-1) - \dots - a(m-1)y(k-m) \quad (9.5)$$

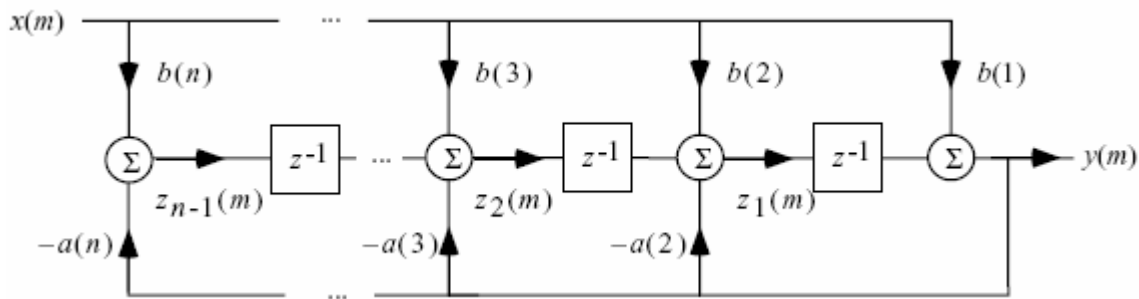
Đây là dạng chuẩn của biểu thức tín hiệu ra trong miền thời gian. Giả sử điều kiện đầu bằng 0, ta có quy trình tính toán như sau:

$$y(1) = b(1)x(1)$$

$$y(2) = b(1)x(2) + b(2)x(1) - a(2)y(1)$$

$$y(3) = b(1)x(3) + b(2)x(2) + b(3)x(1) - a(2)y(2) - a(3)y(1) \quad (9.6)$$

Trong MATLAB, quy trình này được thực hiện bằng hàm **filter**. Chỉ cần cung cấp các vector hệ số của bộ lọc (a và b) cùng với vector tín hiệu vào, hàm sẽ trả về vector tín hiệu ra y có cùng chiều dài với x. Nếu $a(1) \neq 1$, hàm này sẽ chia các hệ số a cho $a(1)$ trước khi thực hiện tính toán. Hàm **filter** thực hiện bộ lọc theo cấu trúc trực tiếp dạng II. Đây là cấu trúc chuẩn tắc có số khâu trễ là ít nhất.



Hình 9.4. Cấu trúc trực tiếp dạng II để thực hiện hàm filter

Ứng với mẫu thứ m của ngõ ra, hàm **filter** thực hiện các phép tính sau:

$$y(m) = b(1)x(m) + z_1(m-1)$$

$$z_1(m) = b(2)x(m) + z_2(m-1) - a(2)y(m)$$

$$z_{n-2}(m) = b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m)$$

$$z_{n-1}(m) = b(n)x(m) - a(n)y(m)$$

Dạng cú pháp cơ bản của hàm **filter**:

```
>> [y,zf] = filter(b,a,x,zi)
```

trong đó *zi* là vector xác định các giá trị đầu của ngõ ra các khối trễ, còn *zf* là vector các giá trị này sau khi thực hiện xong hàm **filter**.

Ví dụ 9-2. Bộ lọc số thông thấp tần số cắt $\omega = 0,4.\pi$ có hàm truyền đạt:

$$H(z) = \frac{0,1 + 0,3z^{-1} + 0,3z^{-2} + 0,1z^{-3}}{1 - 0,58z^{-1} + 0,42z^{-2} - 0,06z^{-3}}$$

được dùng để lọc bỏ thành phần tần số cao trong tín hiệu $x(n) = \sin(\pi n/5) + \cos(4\pi n/5)$, với $0 \leq n \leq 100$. Hãy xác định và vẽ tín hiệu ra $y(n)$.

```
b = [0.0985 0.2956 0.2956 0.0985]; % Các hệ số của đa thức tử
```

```
a = [1.0000 -0.5772 0.4218 -0.0563]; % Các hệ số của đa thức mẫu
```

```
k = (0:100); % Vector thời gian
```

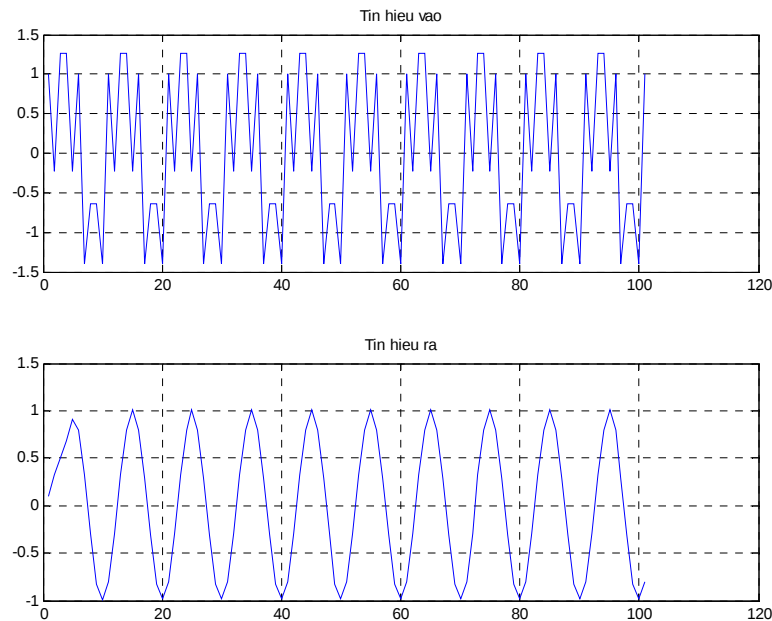
```
x = sin(k*pi/5) + cos(4*k*pi/5); % Tín hiệu x(n)
```

```

y = filter(b,a,x); % Tín hiệu ngõ ra bộ lọc y(n)
subplot(2,1,1);
plot(x);grid % Vẽ tín hiệu vào
title('Tin hieu vao');
subplot(2,1,2);
plot(y);grid % Vẽ tín hiệu ra
title('Tin hieu ra');

```

Kết quả thực thi chương trình:



Hình 9.5.

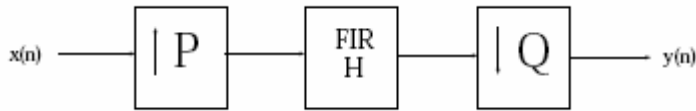
9.5. CÁC HÀM KHÁC ĐỂ THỰC HIỆN LỌC

Ngoài hàm **filter**, để thực hiện các quá trình lọc, trong Signal Processing Toolbox, MATLAB còn cung cấp thêm một số hàm khác, bao gồm hàm **upfirdn** thực hiện bộ lọc FIR được lấy mẫu lại, hàm **filtfilt** cho phép loại bỏ méo pha trong quá trình lọc, hàm **fftfilt** thực hiện quá trình lọc trong miền tần số, hàm **lactfilt** thực hiện bộ lọc theo cấu trúc mạng (lattice).

9.5.1. THỰC HIỆN BĂNG LỌC ĐA TỐC ĐỘ (MULTIRATE FILTER BANK)

Các băng lọc đa tốc độ có thể thực hiện bằng cách dùng hàm **upfirdn**. Hàm này cho phép thay đổi tốc độ lấy mẫu tín hiệu theo một tỷ lệ P/Q với P, Q là các số nguyên. Có thể xem hàm này là kết quả của sự ghép liên tiếp ba hệ thống:

- Bộ lấy mẫu lên (chèn thêm bit 0) với hệ số tỷ lệ P
- Bộ lọc FIR có đáp ứng xung h
- Bộ lấy mẫu xuống với hệ số tỷ lệ Q



Hình 9.6. Nguyên tắc thực hiện hàm **upfirdn**

Cấu trúc nói trên được thực hiện bằng kỹ thuật lọc nhiều pha (polyphase), đó là trọng tâm của lý thuyết băng lọc đa tốc độ.

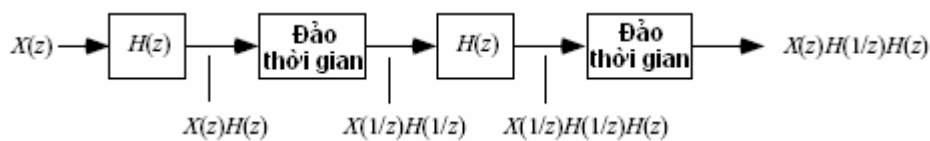
```
>> y = upfirdn(x, h, P, Q)
```

Để thực hiện một băng lọc gồm nhiều bộ lọc ta dùng hàm **upfirdn**, trong đó bộ lọc h là một ma trận mà mỗi cột biểu diễn một bộ lọc FIR. Các tín hiệu ngõ ra cũng biểu diễn dưới dạng ma trận.

9.5.2. KHỬ MÉO PHA CHO BỘ LỌC IIR

Trong trường hợp bộ lọc FIR, ta có thể thiết kế bộ lọc có pha tuyến tính, dữ liệu ra chỉ lệch so với dữ liệu vào một số lượng ký hiệu cố định. Nhưng với bộ lọc IIR, méo pha thường có tính chất phi tuyến cao. Thông thường, người ta sử dụng các thông tin về tín hiệu tại các thời điểm trước và sau thời điểm hiện tại để khắc phục hiện tượng méo pha này. MATLAB xây dựng hàm **filtfilt** để thực hiện giải thuật nói trên.

Đầu tiên chúng ta khảo sát mô hình dưới đây. Chú ý rằng nếu biến đổi z của một chuỗi $x(n)$ là $X(z)$ thì biến đổi z của chuỗi đảo ngược thời gian của x sẽ là $X(1/z)$.



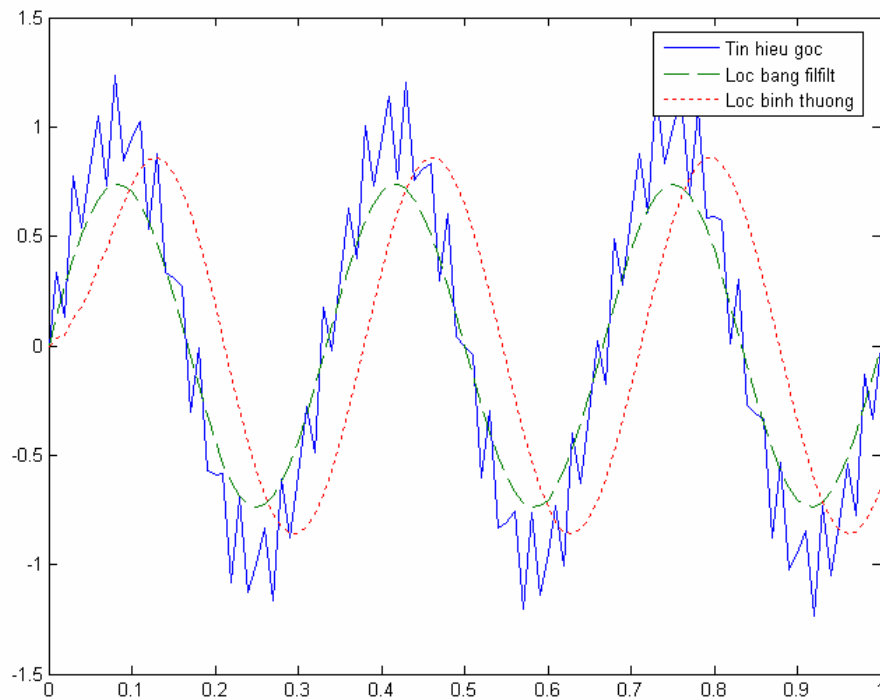
Hình 9.7.

Khi $|z| = 1$, tức là $z = e^{j\omega}$, tín hiệu ngõ ra trở thành $X(e^{j\omega})|H(e^{j\omega})|^2$. Vậy nếu biết tất cả các mẫu của tín hiệu $x(n)$ thì sau hai lần lọc liên tiếp, ta được một tín hiệu có độ lệch pha bằng 0 so với $x(n)$.

Ví dụ 9-3. So sánh hai phương pháp lọc dùng hàm **filter** và hàm **filtfilt** để thực hiện lọc một tín hiệu sin có hai thành phần tần số 3Hz và 40Hz bằng bộ lọc trung bình 10 điểm.

```
fs = 100;
t = 0:1/fs:1;
x = sin(2*pi*t*3) + .25*sin(2*pi*t*40);
b = ones(1,10)/10; % 10 point averaging filter
y = filtfilt(b,1,x); % Noncausal filtering
yy = filter(b,1,x); % Normal filtering
plot(t,x,t,y,'--',t,yy,':')
legend('Tín hiệu gốc','Lọc bằng filtfilt','Lọc bình thường')
```

Hình 9.8 là đồ thị thu được sau khi thực hiện chương trình.



Hình 9.8.

Từ đồ thị này ta nhận thấy cả hai bộ lọc đều loại bỏ được thành phần tần số 40Hz trong tín hiệu gốc. Tuy nhiên, trong khi tín hiệu ra của bộ lọc **filfilt** cùng pha với tín hiệu gốc thì tín hiệu ra của bộ lọc **filter** bị trễ đi khoảng 5 ký hiệu so với tín hiệu gốc. Chúng ta cũng thấy rằng biên độ ngõ ra của bộ lọc **filfilt** nhỏ hơn do kết quả của việc bình phương biên độ hàm truyền H.

Lưu ý: để kết quả lọc là tốt nhất, cần bảo đảm chiều dài của tín hiệu vào tối thiểu phải gấp ba lần bậc của bộ lọc **filfilt**, và tín hiệu vào có xu hướng giảm về 0 ở hai phía.

9.5.3. THỰC HIỆN BỘ LỌC TRONG MIỀN TẦN SỐ

Do tính đối ngẫu giữa hai miền thời gian và tần số, bất kỳ một thao tác nào thực hiện được trong miền này đều cũng có thể thực hiện được trong miền còn lại.

Trong miền tần số, bộ lọc IIR được thực hiện bằng cách nhân biến đổi Fourier rời rạc (DFT) của tín hiệu vào với thương số của các biến đổi Fourier của các hệ số bộ lọc. Ví dụ:

```
>> n = length(x);
>> y = ifft(fft(x).*fft(b,n)./fft(a,n));
```

Kết quả tính toán tương tự như hàm filter, tuy vẫn khác nhau ở đoạn quá độ lúc đầu (hiệu ứng biên). Khi chiều dài của chuỗi vào càng tăng, phương pháp này càng kém hiệu quả do phải thêm vào nhiều điểm zero cho các hệ số bộ lọc khi tiến hành tính FFT, đồng thời giải thuật FFT cũng giảm hiệu quả khi số điểm n tăng lên.

Ngược lại, đối với các bộ lọc FIR, ta có thể tách một chuỗi dài thành nhiều chuỗi ngắn hơn, sau đó dùng phương pháp chồng và cộng (overlap and add). Hàm **fftfilt** của MATLAB được xây dựng dựa trên giải thuật này.

```
>> y = fftfilt(b,x)
>> y = fftfilt(b,x,n)
```

x là chuỗi vào, y là chuỗi ra, b là vector các hệ số của bộ lọc, n là số điểm FFT tối thiểu. Hàm **fftfilt**(b, x) tương đương với hàm **filter**(b, 1, x).

9.6. ĐÁP ỨNG XUNG

Đáp ứng xung của bộ lọc là chuỗi tín hiệu ngõ ra của bộ lọc khi đưa vào bộ lọc tín hiệu xung đơn vị:

$$x(n) = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases} \quad (9.8)$$

Để tìm đáp ứng xung của một bộ lọc, ta có thể tạo một chuỗi xung đơn vị rồi dùng hàm **filter**:

```
>> imp = [1; zeros(49,1)];
>> h = filter(b,a,imp);
```

9.7. ĐÁP ỨNG TẦN SỐ

Signal Processing Toolbox cho phép thực hiện các phép phân tích trong miền tần số đối với cả bộ lọc tương tự lẫn bộ lọc số.

9.7.1. TRONG MIỀN SỐ

Trong miền số, đáp ứng tần số được tính toán bằng giải thuật FFT. Hàm **freqz** sẽ trả về đáp ứng tần số phức (FFT) p điểm của bộ lọc số có các vector hệ số là a và b.

```
>> [H,W] = freqz(b,a,p)
```

H là đáp ứng tần số của bộ lọc được tính tại p điểm tần số cho bởi vector W. Các điểm tần số được chọn cách đều nhau và nằm ở nửa trên của vòng tròn đơn vị.

$$H(e^{j\omega}) = \frac{b(1) + b(2)e^{-j\omega} + \dots + b(n+1)e^{-j\omega n}}{a(1) + a(2)e^{-j\omega} + \dots + a(m+1)e^{-j\omega m}} \quad (9.9)$$

Ngoài ra, còn có thể cung cấp cho hàm **freqz** các thông số khác. Sau đây là các cú pháp khác của hàm:

```
>> [h,w] = freqz(b,a,n,'whole') tính đáp ứng tần số tại n điểm phân bố đều trên toàn bộ vòng tròn đơn vị.
```

```
>> h = freqz(b,a,w) tính đáp ứng tần số tại các điểm xác định bởi vector w.
```

```
>> [h,f] = freqz(b,a,n,fs) hoặc [h,f] = freqz(b,a,n,'whole',fs) tính đáp ứng tần số n điểm của bộ lọc với tần số lấy mẫu là fs. n điểm tần số phân bố đều trên khoảng [0, fs/2] (hoặc [0,fs] nếu dùng 'whole').
```

```
>> h = freqz(b,a,f,fs) tính đáp ứng tần số tại các điểm xác định bởi vector f, trong đó fs là tần số lấy mẫu.
```

Nếu gọi hàm **freqz** mà không yêu cầu trả về các thông số ra, hàm **freqz** sẽ vẽ đáp ứng biên độ và đáp ứng pha của bộ lọc.

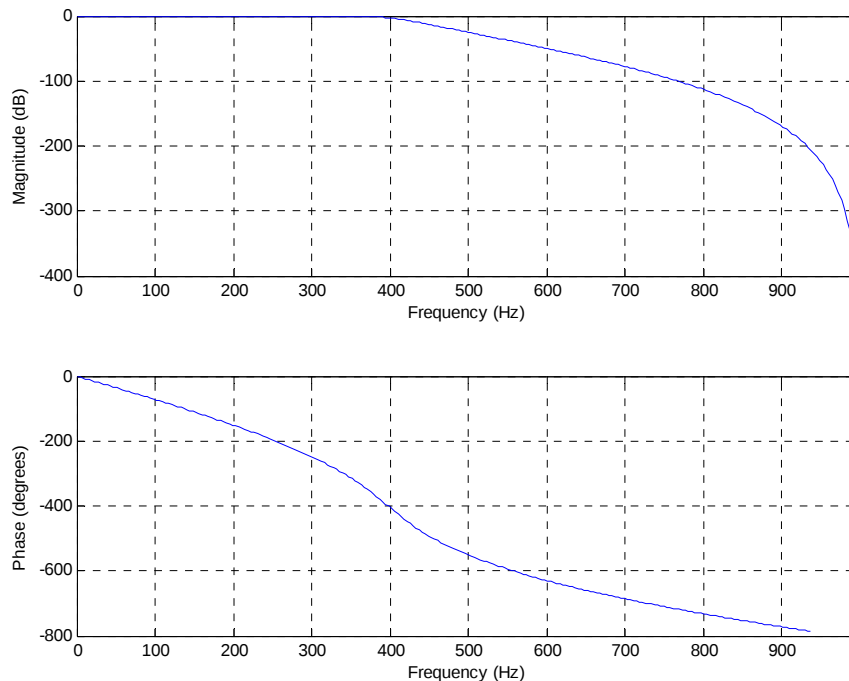
Lưu ý: trong MATLAB thường sử dụng các giá trị tần số chuẩn hoá, trong đó tần số đơn vị là tần số Nyquist, tức một nửa tần số lấy mẫu. Muốn chuyển từ tần số chuẩn hoá sang tần số góc quanh vòng tròn đơn vị, chỉ cần nhân với π , muốn chuyển từ tần số chuẩn hoá sang tần số Hertz, nhân với một nửa tần số lấy mẫu.

Ví dụ 9-4. Tính và vẽ đáp ứng tần số 256 điểm của bộ lọc Butterworth bậc 9, tần số cắt 400Hz. Biết tần số lấy mẫu là 2000Hz.

```
[b,a] = butter(9,400/1000); % Các hệ số của bộ lọc Butterworth
[h,f] = freqz(b,a,256,2000); % Đáp ứng tần số
```

```
freqz(b,a,256,2000)
```

Kết quả:



Hình 9.9.

9.7.2. TRONG MIỀN ANALOG

Hàm **freqs** có thể thực hiện các chức năng tương tự như hàm **freqz** nhưng đối với các bộ lọc analog.

```
>> [h,w] = freqs(b,a)
```

```
>> h = freqs(b,a,w)
```

9.7.3. ĐÁP ỨNG BIÊN ĐỘ VÀ ĐÁP ỨNG PHA

Muốn có đáp ứng biên độ và đáp ứng pha của một bộ lọc, ta chỉ cần xác định đáp ứng tần số của nó bằng cách dùng hàm **freqz** hoặc **freqs**, sau đó dùng hàm **abs** để lấy đáp ứng biên độ hoặc hàm **angle** để lấy đáp ứng pha.

MATLAB còn cung cấp hàm **unwrap** trả về đáp ứng pha liên tục tại các vị trí $\pm 360^\circ$ (thông thường góc pha chỉ được tính trong phạm vi -360° đến $+360^\circ$ nên tại các vị trí này đáp ứng pha sẽ có bước nhảy) bằng cách thêm vào các bội số của $\pm 360^\circ$ nếu cần thiết.

Ngoài ra, ta cũng có thể sử dụng hàm **phasez** để có đáp ứng pha liên tục như trên. Cú pháp của hàm **phasez** hoàn toàn tương tự như hàm **freqz**, chỉ khác là kết quả trả về là đáp ứng pha thay vì đáp ứng tần số.

 **Ví dụ 9-5.** Xác định và vẽ đáp ứng biên độ và đáp ứng pha của bộ lọc FIR bậc 25.

```
h = fir1(25,0.4);
```

```
[H,f] = freqz(h,1,512,2);
```

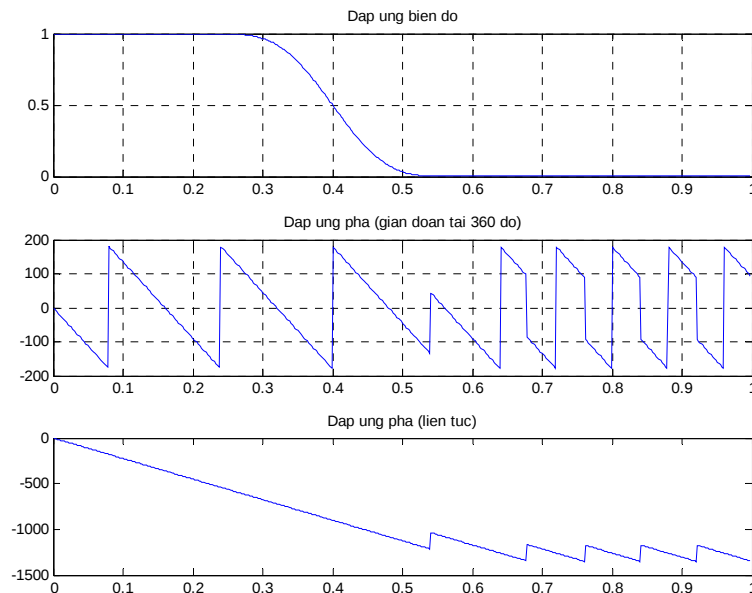
```
subplot(3,1,1);
```

```
plot(f,abs(H)); grid
```

```

title('Đáp ứng biên độ');
subplot(3,1,2);
plot(f,angle(H)*180/pi); grid
title('Đáp ứng pha (gian đoạn tại 360 độ)');
subplot(3,1,3);
plot(f,unwrap(angle(H))*180/pi);
title('Đáp ứng pha (liên tục)');

```



Hình 9.10.

Ở đồ thị đáp ứng pha đầu tiên, ta có thể phân biệt các điểm nhảy 360° (do góc pha vượt ra ngoài giới hạn hàm **angle**) với các điểm nhảy 180° (ứng với các điểm zero của đáp ứng tần số). Nếu dùng hàm **unwrap** ta có thể phân biệt được các vị trí này (như ở đồ thị thứ hai).

9.7.4. THỜI GIAN TRỄ

Độ trễ nhóm (group delay) của một bộ lọc là một thông số đánh giá thời gian trễ trung bình của bộ lọc (là một hàm của tần số). Nếu đáp ứng tần số phức của bộ lọc là $H(e^{j\omega})$ thì độ trễ nhóm được xác định bằng biểu thức:

$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega} \quad (9.10)$$

trong đó $\theta(\omega)$ là đáp ứng pha của bộ lọc.

Trong MATLAB, ta có thể xác định độ trễ nhóm bằng cách dùng hàm **grpdelay** với cú pháp hoàn toàn tương tự như hàm **freqz**.

Ví dụ:

```
>> [gd,w] = grpdelay(b,a,n)
```

trả về độ trễ nhóm $\tau_g(\omega)$ của bộ lọc số xác định bởi các vector hệ số a và b , được tính tại n điểm tần số xác định bởi vector v .

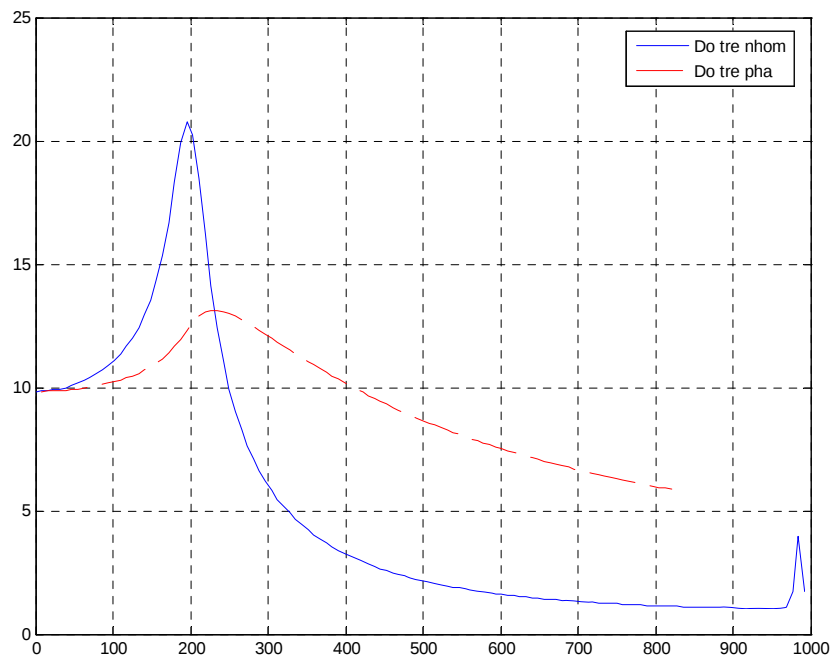
Độ trễ pha (phase delay) của bộ lọc được định nghĩa bởi:

$$\tau_p(\omega) = -\frac{\theta(\omega)}{\omega} \quad (9.11)$$

Tương tự với độ trễ nhóm, độ trễ pha cũng có thể được xác định bằng cách dùng hàm **phasedelay** (với cú pháp hoàn toàn tương tự **freqz**).

Ví dụ 9-6. Xác định và vẽ độ trễ nhóm và độ trễ pha của bộ lọc Butterworth bậc 10, tần số cắt 200Hz, với tần số lấy mẫu là 2000Hz.

```
fs = 2000;
n = 128;
[b,a] = butter(10,200/1000);
[gd,f] = grpdelay(b,a,n,fs);
[pd,f] = phasedelay(b,a,n,fs);
plot(f,gd,'b-',f,pd,'r--'); grid; hold on;
legend('Do tre nhom','Do tre pha');
```



Hình 9.11.

9.8. GIẢN ĐỒ CỰC – ZERO

Ngoài các cách biểu diễn thông qua đáp ứng xung hoặc hàm truyền đạt, một hệ thống tuyến tính nói chung hay một bộ lọc nói riêng cũng có thể biểu diễn dưới dạng giản đồ cực – zero, tức là sự phân bố các điểm cực và điểm zero của hàm truyền đạt trong mặt phẳng z.

Hàm **zplane** cho phép vẽ giản đồ cực – zero của một hệ thống tuyến tính nếu chúng ta cung cấp các điểm cực và zero của hàm truyền hoặc cung cấp các vector hệ số a và b của hệ thống.

>> **zplane(z,p)** vẽ giản đồ cực – zero dựa vào vector các zero z và vector các điểm cực p.

>> **zplane(b,a)** vẽ giản đồ cực – zero dựa vào vector các hệ số a và b.

Lưu ý: giữa mô hình hàm truyền và mô hình cực – zero có thể được chuyển đổi qua lại bằng cách dùng các hàm **tf2zp** và **zp2tf**.

```
>> [b, a] = zp2tf(z, p, k)
```

```
>> [z, p, k] = tf2zp(b, a)
```

trong đó k là hệ số khuếch đại khi $z \rightarrow \infty$.

9.9. CÁC MÔ HÌNH HỆ THỐNG TUYẾN TÍNH

Như ta đã biết, một hệ thống tuyến tính có thể được mô tả bằng nhiều mô hình khác nhau như mô hình hàm truyền, mô hình cực – zero,... Trong MATLAB, Signal Processing Toolbox cung cấp cho ta một cách đầy đủ các mô hình này. Người sử dụng có thể lựa chọn mô hình nào thích hợp để mô phỏng hệ thống một cách nhanh chóng và chính xác nhất.

9.9.1. CÁC MÔ HÌNH HỆ THỐNG RỜI RẠC THEO THỜI GIAN

Các mô hình hệ thống rời rạc được MATLAB hỗ trợ bao gồm:

- Mô hình hàm truyền đạt
- Mô hình độ lợi – cực – zero
- Mô hình không gian trạng thái
- Mô hình khai triển hữu tỷ (mô hình thặng dư)
- Mô hình các khâu bậc hai (SOS – Second Order Sections)
- Mô hình lattice
- Mô hình ma trận chấp

Mô hình hàm truyền đạt

Hệ thống được mô tả thông qua hàm truyền đạt $H(z)$ trong miền z của nó:

$$Y(z) = H(z).X(z) = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}} X(z)$$

Các hệ số $b(i)$ và $a(i)$ là các hệ số của bộ lọc, bậc của bộ lọc là $\max(m, n)$. Trong MATLAB, hàm truyền đạt được biểu diễn bằng hai vector a và b lưu các hệ số của bộ lọc.

Mô hình độ lợi – cực – zero

Hàm truyền đạt của hệ thống có thể viết lại dưới dạng nhân tử:

$$H(z) = \frac{q(z)}{p(z)} = k \frac{(z - q(1))(z - q(2)) \dots (z - q(n))}{(z - p(1))(z - p(2)) \dots (z - p(m))} \quad (9.12)$$

MATLAB biểu diễn mô hình bằng một vô hướng k chỉ độ lợi và hai vector z và p mà các phần tử của chúng lần lượt là các zero (nghiệm của $q(z)$) và các cực (các nghiệm của $p(z)$).

Có thể dùng các hàm **poly** và **roots** để chuyển đổi qua lại giữa các vector cực và zero với các vector hệ số của bộ lọc. Hàm **poly** trả về các hệ số của đa thức nếu biết các nghiệm của nó, còn hàm **roots** trả về các nghiệm của một đa thức chỉ cần cung cấp các hệ số của đa thức. Tuy nhiên, ở phần sau ta sẽ thấy rằng MATLAB cung cấp sẵn các hàm để chuyển trực tiếp từ mô hình hàm truyền sang mô hình cực – zero mà không cần phải chuyển từng đa thức như cách nêu trên.

Mô hình không gian trạng thái

Bất kỳ một bộ lọc số nào cũng có thể được biểu diễn dưới dạng một hệ thống các phương trình sai phân cấp một. Cụ thể là một hệ thống tuyến tính rời rạc có thể được mô tả bằng hệ sau:

$$\begin{cases} x(n+1) = Ax(n) + Bu(n) \\ y(n) = Cx(n) + Dy(n) \end{cases} \quad (9.13)$$

trong đó:

u là tín hiệu vào, x là vector trạng thái, y là tín hiệu ngõ ra.

A là một ma trận kích thước $m \times m$ với m là bậc của bộ lọc, B và C là các vector cột và D là một vô hướng.

Trong trường hợp hệ thống đa kênh thì ngõ vào u , ngõ ra y trở thành các vector, còn B , C và D trở thành các ma trận.

Xuất phát từ biểu thức $Y(z) = H(z)U(z)$, ta suy ra quan hệ giữa hàm truyền đạt $H(z)$ của bộ lọc với các ma trận B , C và D :

$$H(z) = C(zI - A)^{-1}B + D \quad (9.14)$$

Mô hình khai triển các phân thức (mô hình thẳng dư)

Bất kỳ hàm truyền đạt $H(z)$ nào cũng có thể khai triển thành tổng của các phân thức hữu tỷ theo dạng sau (còn gọi là dạng thẳng dư):

$$H(z) = \frac{b(z)}{a(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \dots + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \dots + k(m-n+1)z^{-(m-n)} \quad (9.15)$$

với điều kiện $H(z)$ không có cực nào lặp lại. Trong trường hợp $H(z)$ có một cực r nào đó được lặp lại s_r lần (nghĩa là r là cực bội s_r của $H(z)$) thì ứng với các cực $p(j) = p(j+1) = \dots = p(j+s_r-1)$ này, trong khai triển hữu tỷ của $H(z)$ sẽ có các số hạng sau:

$$\frac{r(j)}{1 - p(j)z^{-1}} + \frac{r(j+1)}{(1 - p(j)z^{-1})^2} + \dots + \frac{r(j+s_r-1)}{(1 - p(j)z^{-1})^{s_r}} \quad (9.16)$$

Mô hình này được biểu diễn bằng ba vector cột: vector p chứa các cực của $H(z)$, vector r chứa các thẳng dư tương ứng với các cực (tức các hệ số $r(j)$ trong các biểu thức (9.15), (9.16)) và vector k chứa các hệ số $k(i)$. Vậy $\text{length}(r) = \text{length}(p) = \text{length}(a) - 1$.

Hàm **residuez** cho phép ta chuyển đổi từ mô hình hàm truyền sang mô hình thẳng dư và ngược lại.

```
>> [r,p,k] = residuez(b,a)
```

```
>> [b,a] = residuez(r,p,k)
```

b , a là các vector hệ số của bộ lọc còn (r, p, k) biểu diễn mô hình thẳng dư của bộ lọc.

Khi xác định mô hình thẳng dư từ các vector a và b , MATLAB sẽ xem như hai cực của $H(z)$ là trùng nhau nếu chúng sai khác nhau không quá 0,1% biên độ của cả hai, và khi đó MATLAB sẽ sử dụng (9.16) để xác định các giá trị thẳng dư.

Mô hình các khâu bậc 2 (SOS – Second Order Sections)

Hàm truyền $H(z)$ cũng có thể biểu diễn dưới dạng:

$$H(z) = \prod_{k=1}^L H_k(z) = \prod_{k=1}^L \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}} \quad (9.17)$$

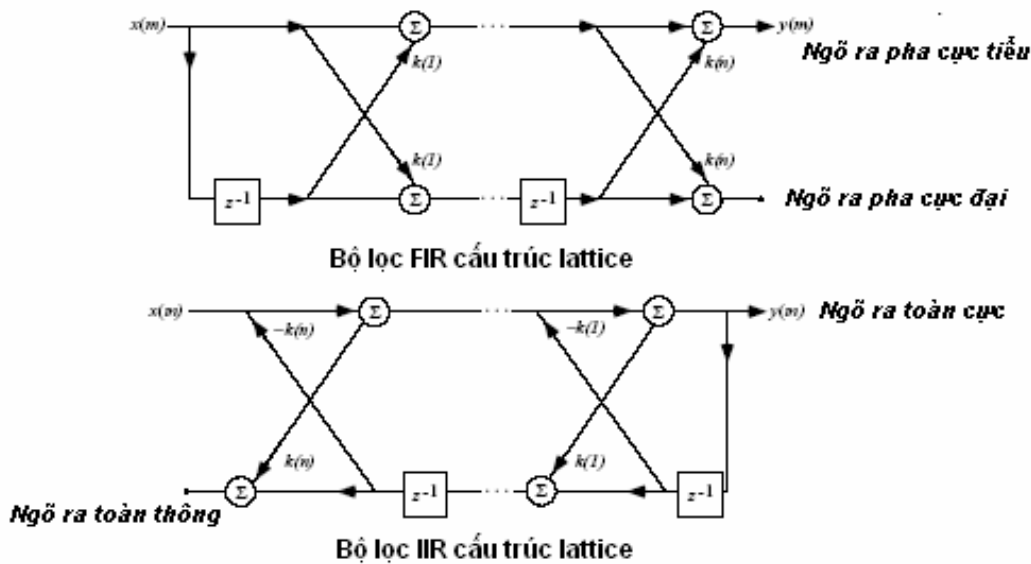
trong đó L là số khâu bậc hai trong hệ thống, mỗi hàm $H_k(z)$ biểu diễn một khâu bậc hai.

MATLAB biểu diễn mô hình SOS của một hệ thống bằng một ma trận kích thước $L \times 6$ với các phần tử được bố trí như sau:

$$SOS = \begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & a_{0L} & a_{1L} & a_{2L} \end{bmatrix} \quad (9.18)$$

Với một hàm truyền $H(z)$ cho trước, có nhiều cách để triển khai thành mô hình SOS. Thông qua việc chọn lựa các cặp cực để ghép với nhau, sắp thứ tự các khâu bậc hai và thay đổi các hệ số nhân, ta có thể giảm hệ số khuếch đại nhiều lượng tử và tránh hiện tượng tràn số khi thực hiện các bộ lọc fixed-point.

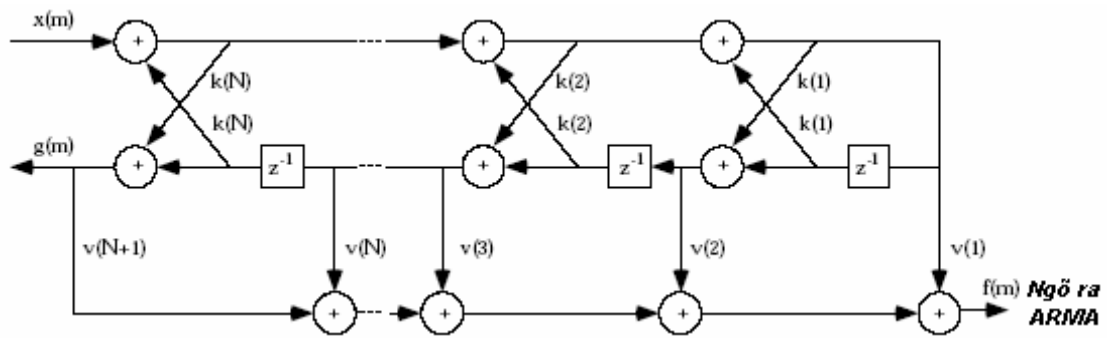
Mô hình lattice



Hình 9.12. Cấu trúc lattice của các bộ lọc FIR và IIR

Với các bộ lọc rời rạc bậc n toàn điểm cực hoặc toàn zero, được biểu diễn bởi đa thức với các hệ số $a(i)$, $i = 1, 2, \dots, n + 1$, ta có thể tìm được n hệ số tương ứng $k(i)$, $i = 1, 2, \dots, n$, các hệ số này là cơ sở để xây dựng cấu trúc lattice cho bộ lọc. Các hệ số $k(i)$ này còn gọi là các hệ số phân xạ của bộ lọc. Hình 9.12 mô tả cấu trúc lattice của các bộ lọc FIR và IIR với các hệ số phân xạ $k(i)$ cho trước.

Với bộ lọc IIR tổng quát (có cả cực lẫn zero) có các hệ số bộ lọc chứa trong các vector a và b , ngoài các hệ số $k(i)$ ứng với vector a còn có các hệ số $v(i)$, $i = 1, 2, \dots, N + 1$, gọi là các hệ số bậc thang. Cấu trúc lattice của bộ lọc xây dựng dựa trên các hệ số $k(i)$ và $v(i)$ được mô tả ở hình 9.13.



Hình 9.13. Cấu trúc lattice của bộ lọc ARMA

Để tìm các hệ số của cấu trúc lattice từ các hệ số a, b của bộ lọc, ta dùng hàm **tf2latc**:

>> **k = tf2latc(num)** hoặc **k = tf2latc(num, 'max')** hoặc **k = tf2latc(num, 'min')**
trả về cấu trúc lattice của bộ lọc FIR hoặc bộ lọc FIR pha cực đại hoặc cực tiểu.

>> **k = tf2latc(1, den)** trả về cấu trúc lattice của bộ lọc IIR toàn cực

>> **[k, v] = tf2latc(num, den)** trả về cấu trúc lattice của bộ lọc IIR tổng quát

num, den là các vector hệ số của bộ lọc còn k là vector các hệ số lattice, v là vector các hệ số bậc thang.

Ngược lại, từ cấu trúc lattice của bộ lọc có thể suy ra hàm truyền đạt của bộ lọc bằng cách dùng hàm **latc2tf**.

>> **[num, den] = latc2tf(k, v)**

>> **[num, den] = latc2tf(k, 'allpole')** (K: cấu trúc lattice của bộ lọc IIR toàn cực)

>> **[num, den] = latc2tf(k, 'allpass')** (K: cấu trúc lattice của bộ lọc IIR toàn thông)

>> **num = latc2tf(k)**

>> **num = latc2tf(k, 'max')** (K: cấu trúc lattice của bộ lọc FIR pha cực đại)

>> **num = latc2tf(k, 'min')** (K: cấu trúc lattice của bộ lọc FIR pha cực tiểu)

Mô hình ma trận chập

Trong lý thuyết xử lý tín hiệu, phép lấy tích chập hai vector hay hai ma trận tương đương với quá trình lọc một trong hai vector này bằng một bộ lọc có hệ số xác định bởi vector còn lại. Trên cơ sở đó, ta có thể mô tả một bộ lọc số bằng một ma trận chập.

Với một vector cho trước, hàm **convmtx** sẽ tạo ra một ma trận chập sao cho tích của ma trận này với một vector x nào đó khác sẽ bằng với tích chập của vector đã cho trước với vector x, với điều kiện vector x phải có chiều dài thích hợp với kích thước của ma trận chập để phép nhân ma trận có ý nghĩa.

Ví dụ: giả sử một bộ lọc FIR có các hệ số là **b = [1 2 3]**, ma trận chập của nó được xây dựng như sau:

```
>> b = [1 2 3]; x = rand(3,1);
```

```
>> C = convmtx(b',3)
```

C =

1	0	0
2	1	0
3	2	1

0	3	2
0	0	3

Như vậy, có hai cách tương đương để tính tích chập của b và x .

```
>> y1 = C*x;
```

```
>> y2 = conv(b,x);
```

9.9.2. CÁC MÔ HÌNH HỆ THỐNG LIÊN TỤC THEO THỜI GIAN

Các mô hình hệ thống liên tục theo thời gian dùng để mô tả các bộ lọc analog. Đa số các mô hình đã xây dựng cho các hệ thống rời rạc đã đề cập ở trên cũng có thể áp dụng cho các hệ thống liên tục theo thời gian. Cụ thể là các mô hình sau:

- Mô hình không gian trạng thái
- Mô hình khai triển phân thức hữu tỷ
- Mô hình hàm truyền đạt
- Mô hình độ lợi – cực – zero

Mô hình không gian trạng thái biểu diễn hệ thống bằng một hệ các phương trình vi phân bậc nhất:

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx + Du \end{cases} \quad (9.19)$$

u là vector các ngõ vào (chiều dài nu), y là vector các ngõ ra (chiều dài ny), x là vector các biến trạng thái (chiều dài nx). A , B , C , D là các ma trận và MATLAB dùng các ma trận này để biểu diễn mô hình không gian trạng thái của hệ thống.

Hệ thống liên tục cũng có thể được mô tả bằng mô hình hàm truyền đạt Laplace $H(s)$:

$$Y(s) = H(s)U(s) \quad (9.20)$$

Giữa hàm truyền đạt $H(s)$ với các ma trận A , B , C , D có mối liên hệ:

$$H(s) = C(sI - A)^{-1} B + D \quad (9.21)$$

Với hệ thống một ngõ vào, một ngõ ra, hàm $H(s)$ có dạng:

$$H(s) = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \dots + a(m+1)} \quad (9.22)$$

MATLAB biểu diễn mô hình hàm truyền đạt Laplace bằng hai vector a và b lưu các hệ số $a(i)$ và $b(i)$.

Hàm truyền đạt Laplace cũng có thể được biểu diễn dưới dạng nhân tử:

$$H(s) = \frac{z(s)}{p(s)} = k \frac{(s - z(1))(s - z(2)) \dots (s - z(n))}{(s - p(1))(s - p(2)) \dots (s - p(m))} \quad (9.23)$$

Đây chính là cơ sở cho việc xây dựng một mô hình khác cho hệ thống liên tục, đó là mô hình độ lợi – cực – zero. Tương tự như với hệ thống rời rạc, MATLAB biểu diễn mô hình này bằng một vô hướng k chỉ độ lợi và hai vector p và z chứa các cực và zero của $H(s)$.

9.9.3. CÁC PHÉP BIẾN ĐỔI HỆ THỐNG TUYẾN TÍNH

Bảng 9.1 tóm tắt các hàm MATLAB dùng để chuyển đổi giữa các mô hình hệ thống tuyến tính. Để chuyển từ một mô hình (nguồn) sang một mô hình khác (đích), ta xuất phát từ hàng

có chứa tên mô hình nguồn, sau đó tìm ô ứng với cột mang tên mô hình đích, nội dung trong ô này chính là tên hàm để thực hiện chuyển đổi.

Bảng 9.1. Các hàm MATLAB thực hiện các phép biến đổi hệ thống tuyến tính

	Hàm truyền đạt	Không gian trạng thái	Độ lợi – cực -	Phân thức hữu tỷ	Bộ lọc lattice	SOS	Ma trận chấp
Hàm truyền đạt		tf2ss	tf2zp roots	residuez	tf2latc	không có	convmtx
Không gian trạng thái	ss2tf		ss2zp	không có	không có	ss2sos	không có
Độ lợi – cực - zero	zp2tf poly	zp2ss		không có	không có	zp2sos	không có
Phân thức hữu tỷ	residuez	không có	không có		không có	không có	không có
Bộ lọc lattice	latc2tf	không có	không có	không có		không có	không có
SOS	sos2tf	sos2ss	sos2zp	không có	không có		không có

9.10. BIẾN ĐỔI FOURIER RỜI RẠC

Biến đổi Fourier rời rạc (DFT – Discrete Fourier Transform) là công cụ cơ bản nhất trong xử lý số tín hiệu. Trong Signal Processing Toolbox, hầu hết các hàm đều có sử dụng giải thuật FFT (một giải thuật tính DFT nhằm giảm thời gian thực thi).

Trong MATLAB có hai hàm **fft** và **ifft** dùng để tính toán biến đổi DFT thuận và nghịch bằng giải thuật FFT. Với một chuỗi tín hiệu vào x và biến đổi DFT của nó là X , giải thuật FFT được thực hiện trên cơ sở các biểu thức sau:

$$\begin{cases} X(k+1) = \sum_{n=0}^{N-1} x(n+1)W_N^{kn} \\ x(n+1) = \frac{1}{N} \sum_{k=0}^{N-1} X(k+1)W_N^{-kn} \end{cases} \quad (9.24)$$

trong đó $W_N = e^{-j(2\pi/N)}$.

Cú pháp của các hàm **fft** và **ifft** như sau:

```
>> fft(x)
>> fft(x,N)
```

x là chuỗi tín hiệu vào còn N là số điểm FFT. Nếu x có ít hơn N điểm, hàm **fft** sẽ tự động chèn thêm zero vào, nếu x có nhiều hơn N điểm, hàm sẽ tự động cắt bỏ các điểm thừa.

```
>> ifft(X)
>> ifft(X,N)
```

X là biến đổi DFT của tín hiệu vào, N là số điểm FFT.

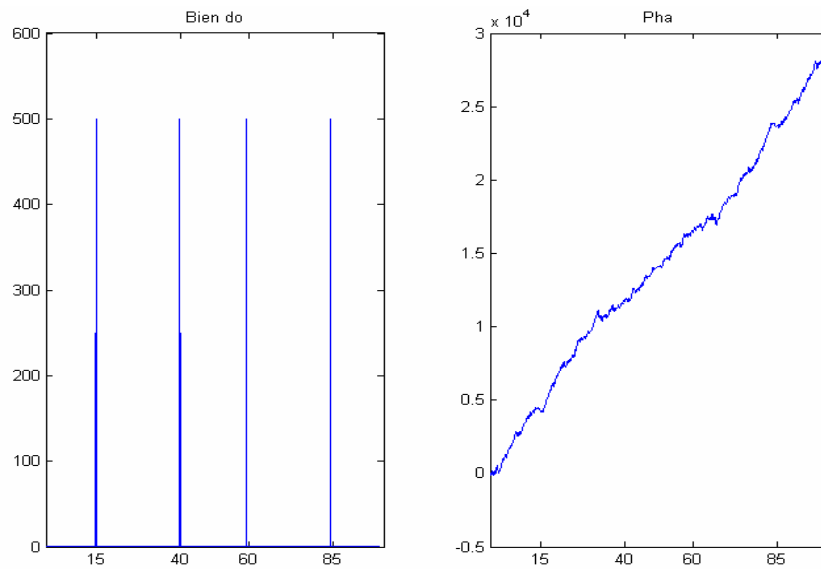
 **Ví dụ 9-7.** Hãy xác định và vẽ biên độ và pha của biến đổi DFT của tín hiệu $x(t) = \sin(30\pi t) + \sin(80\pi t)$, tần số lấy mẫu 100Hz. Dùng biến đổi DFT ngược để khôi phục lại tín hiệu ban đầu. So sánh với tín hiệu gốc.

```

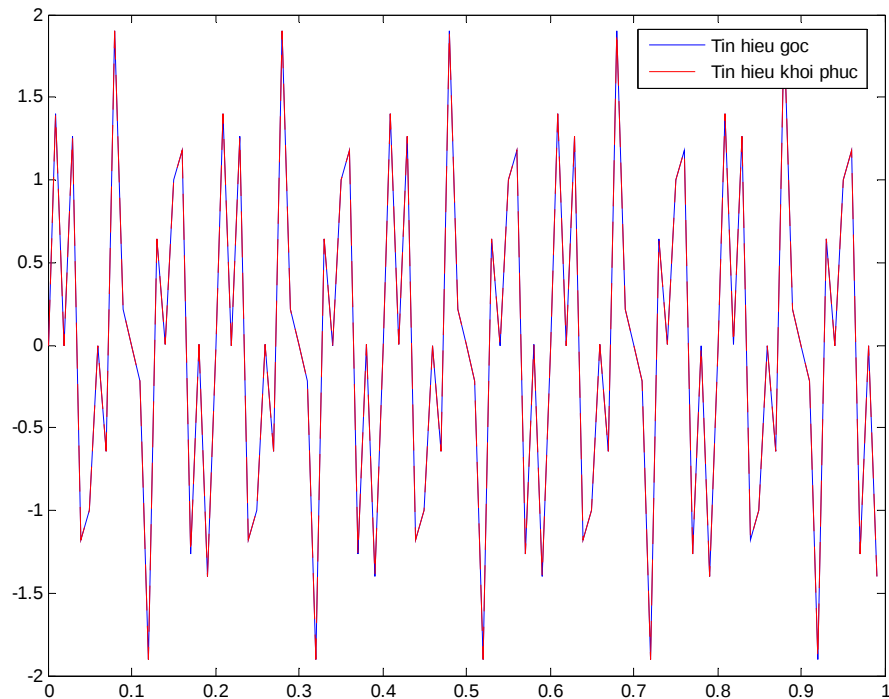
t = (0:1/100:10-1/100); % Vector thời gian
x = sin(2*pi*15*t) + sin(2*pi*40*t); % Tín hiệu x(t)
y = fft(x); % Biến đổi DFT của x
m = abs(y); p = unwrap(angle(y)); % Biên độ và pha
f = (0:length(y)-1)*99/length(y); % Vector tần số
subplot(1,2,1);
plot(f,m); title('Bien do');
set(gca,'XTick',[15 40 60 85]);
subplot(1,2,2); plot(f,p*180/pi); title('Pha');
set(gca,'XTick',[15 40 60 85]);
xr = real(ifft(y)); % Biến đổi FFT ngược
figure;
plot(t,x,'b-',t,xr,'r--');
legend('Tin hieu goc','Tin hieu khoi phuc');

```

Kết quả như sau:



Hình 9.14. Biến đổi DFT



Hình 9.15. Biến đổi DFT ngược

Ngoài hai hàm cơ bản nêu trên, trong Signal Processing Toolbox còn có một số hàm khác để tính DFT và IDFT:

- Hàm **fft2** và **ifft2** tính biến đổi DFT và IDFT 2 chiều của ma trận x .

```
>> fft2(x)
>> fft2(x,M,N); M,N: số hàng và cột của DFT
>> ifft2(X)
>> ifft2(X,M,N)
```

- Hàm **goertzel** tính biến đổi DFT theo giải thuật Goertzel.

```
>> X = goertzel(x,indvec)
```

x là tín hiệu vào còn $indvec$ là vector chỉ số (mặc định là $1:N$).

- Đôi khi ta cần sắp xếp lại chuỗi vào sao cho các điểm zero trong chuỗi ngõ ra nằm ở khoảng giữa chuỗi. Khi đó ta dùng hàm **fftshift**.

```
>> X = fftshift(x)
```

🔗 Bài tập 9-1.

Tạo và vẽ các tín hiệu sau trong MATLAB:

a. $x_1(n) = \sum_{m=0}^{10} (m+1)[\delta(n-2m) - \delta(n-2m-1)], \quad 0 \leq n \leq 25.$

b. $x_2(n) = n^2[u(n+5) - u(n-6)] + 10\delta(n) + 20 \cdot 0,5^n[u(n-4) - u(n-10)]$

c. $x_3(n) = 0,9^n \cos(0,2\pi n + \pi/3), \quad 0 \leq n \leq 20$

d. $x_4(n) = 10 \cos(0,0008\pi n^2) + w(n)$, $0 \leq n \leq 100$, với $w(n)$ là chuỗi ngẫu nhiên phân bố đều trên đoạn $[-1, 1]$

e. $x_5(n) = \{\dots, 1, 2, 3, 2, 1, 2, 3, 2, 1, \dots\}$
 $\uparrow$

✎ Bài tập 9-2.

Phép lấy tích chập có một số tính chất cơ bản như sau:

- Tính giao hoán: $x_1(n) * x_2(n) = x_2(n) * x_1(n)$
- Tính kết hợp: $[x_1(n) * x_2(n)] * x_3(n) = x_1(n) * [x_2(n) * x_3(n)]$
- Tính phân phối: $[x_1(n) * [x_2(n) + x_3(n)]] = x_1(n) * x_2(n) + x_1(n) * x_3(n)$
- Phần tử đơn vị: $x(n) * \delta(n - n_0) = x(n - n_0)$

Sử dụng hàm **conv** của MATLAB với các chuỗi $x_1(n), x_2(n), x_3(n)$ cho dưới đây để kiểm chứng lại các tính chất trên:

$$x_1(n) = n[u(n + 10) - u(n - 20)]$$

$$x_2(n) = \cos(0,1\pi n)[u(n) - u(n - 30)]$$

$$x_3(n) = 1,2^n[u(n + 5) - u(n - 10)]$$

✎ Bài tập 9-3.

Một bộ sai phân số đơn giản được định nghĩa bởi phương trình:

$$y(n) = x(n) - x(n - 1).$$

Sử dụng bộ sai phân này đối với các tín hiệu vào cho dưới đây (dùng hàm **filter**). Nhận xét phạm vi sử dụng thích hợp của bộ sai phân này.

- $x(n) = 5[u(n) - u(n - 20)]$: xung chữ nhật
- $x(n) = n[u(n) - u(n - 10)] + (20 - n)[u(n - 10) - u(n - 20)]$: xung tam giác
- $x(n) = \sin\left(\frac{\pi n}{25}\right)[u(n) - u(n - 100)]$: xung sine

✎ Bài tập 9-4.

Làm lại bài tập 9-3 nhưng dùng hàm **filtfilt**.

✎ Bài tập 9-5.

Một hệ thống tuyến tính bất biến theo thời gian được mô tả bởi phương trình sai phân:

$$y(n) - 0,5y(n - 1) + 0,25y(n - 2) = x(n) + 2x(n - 1) + x(n - 3)$$

- Xét tính ổn định của hệ thống
- Xác định và vẽ đáp ứng xung của hệ thống trong khoảng $0 \leq n \leq 100$. Xét tính ổn định dựa vào đáp ứng xung này.

✎ Bài tập 9-6.

Với mỗi hệ thống tuyến tính bất biến được định nghĩa bởi các đáp ứng xung dưới đây, hãy xác định đáp ứng tần số $H(e^{j\omega})$, đáp ứng biên độ $|H(e^{j\omega})|$, đáp ứng pha $\theta(\omega)$:

- i. $h(n) = 0,9^{|n|}$
- ii. $h(n) = \text{sinc}(0,2n)[u(n+20) - u(n-20)]$, $h(0) = 1$.
- iii. $h(n) = \text{sinc}(0,2n)[u(n) - u(n-40)]$, $h(0) = 1$
- iv. $h(n) = (0,5^n + 0,4^n)u(n)$
- v. $h(n) = 0,5^{|n|} \cos(0,1\pi n)$

Lần lượt khảo sát với các giá trị Eb/No bằng 0, 2, 4, 6, 8 dB và so sánh thông điệp nhận được với thông điệp phát. Vẽ đồ thị BER.

✎ Bài tập 9-7.

Cho tín hiệu $x(n) = 3\cos(0,5\pi n + \pi/3) + 2\sin(0,2\pi n)$ lần lượt đi vào các hệ thống trong bài tập 9-6. Vẽ tín hiệu ra $y(n)$ trong mỗi trường hợp.

✎ Bài tập 9-8.

Một bộ lọc số được mô tả bởi phương trình sai phân:

$$y(n) = x(n) + x(n-1) + 0,9y(n-1) - 0,81y(n-2)$$

- a. Sử dụng hàm **freqz**, hãy vẽ đáp ứng biên độ và đáp ứng pha của bộ lọc trên. Xác định biên độ và pha tại các tần số $\omega = \pi/3$ và $\omega = \pi$.
- b. Tính độ trễ nhóm và độ trễ pha của bộ lọc theo tần số. Vẽ đồ thị.
- c. Khởi tạo 200 mẫu tín hiệu $x(n) = \sin(\pi n/3) + 5\cos(\pi n)$. So sánh phần xác lập của ngõ ra với ngõ vào $x(n)$. Bộ lọc có ảnh hưởng gì đến biên độ và pha của tín hiệu vào?

✎ Bài tập 9-9.

Cho các hệ thống tuyến tính bất biến được định nghĩa thông qua đáp ứng xung sau đây:

- a. $h(n) = 2 \cdot 0,5^n \cdot u(n)$
- b. $h(n) = n \left(\frac{1}{3}\right)^n u(n) + \left(-\frac{1}{4}\right)^n u(n)$
- c. $h(n) = 3 \cdot 0,9^n \cos(n\pi/4 + \pi/3)u(n+1)$
- d. $h(n) = n[u(n) - u(n-10)]$
- e. $h(n) = [2 - \sin(\pi n)]u(n)$

Hãy xác định: (i) hàm truyền đạt miền z, (ii) giản đồ cực – zero, (iii) mô hình không gian trạng thái, (iv) mô hình SOS, (v) mô hình thặng dư và (vi) mô hình lattice của hệ thống.

✎ Bài tập 9-10.

Cho các hệ thống tuyến tính bất biến được định nghĩa thông qua các hàm truyền đạt sau đây:

- a. $H(z) = \frac{z+1}{z-0,5}$, hệ thống nhân quả.
- b. $H(z) = \frac{1+z^{-1}+z^{-2}}{1+0,5z^{-1}-0,25z^{-2}}$, hệ thống ổn định

c. $H(z) = \frac{z}{z - 0,25} + \frac{1 - 0,5z^{-1}}{1 + 2z^{-1}}$, hệ thống ổn định

d. $H(z) = \frac{z^2 - 1}{(z - 3)^2}$, hệ thống phản nhân quả

e. $H(z) = (1 + z^{-1} + z^{-2})^2$, hệ thống ổn định

Hãy xác định: (i) đáp ứng xung (vẽ), (ii) giản đồ cực – zero, (iii) mô hình không gian trạng thái, (iv) mô hình SOS, (v) mô hình thặng dư và (vi) mô hình lattice của hệ thống

✎ Bài tập 9-11.

Nếu các chuỗi $x(n)$ và $h(n)$ có chiều dài hữu hạn lần lượt là N_x và N_h thì phép tích chập $y(n) = x(n)*h(n)$ có thể thực hiện bằng phép nhân ma trận: giả sử $x(n)$ và $y(n)$ được sắp thành các vector cột x , y thì:

$$y = H.x$$

Trong đó các hàng của ma trận H là các vector $h(n-k)$ với $k = 0, 1, \dots, N_h - 1$. Ma trận này gọi là ma trận Toeplitz.

Xét chuỗi $x(n) = \{1, 2, 3, 4\}$ và $h(n) = \{3, 2, 1\}$.

a. Biểu diễn x dưới dạng vector cột 4×1 , y dưới dạng vector cột 6×1 . Hãy xác định ma trận H .

b. Viết một hàm MATLAB để thực hiện phép tích chập bằng phương pháp ma trận Toeplitz nói trên:

```
function [y,H] = conv_tp(h,x)
```

Nhập vào $x(n)$ và $h(n)$, hàm trả về tích chập của $x(n)$ và $h(n)$ cùng với ma trận Toeplitz H của phép tích chập này.

✎ Bài tập 9-12.

Một tín hiệu tương tự $x_a(n) = 2\sin(4\pi t) + 5\cos(8\pi t)$ được lấy mẫu ở các thời điểm $t = 0,01n$ với $n = 0, 1, \dots, N - 1$ để đạt được một chuỗi $x(n)$ gồm N điểm. Sử dụng biến đổi DFT N điểm để tìm đáp ứng biên độ gần đúng của $x_a(n)$.

Trong các giá trị N sau, hãy chọn giá trị thích hợp để có đáp ứng biên độ chính xác nhất. Vẽ phần thực và phần ảo của phổ DFT của $x_a(n)$.

i. $N = 40$

ii. $N = 50$

iii. $N = 64$

iv. $N = 70$

✎ Bài tập 9-13.

Cho tín hiệu $x(n) = \cos(\pi n/99)$, $n = 0, 1, \dots, N - 1$. Chọn $N = 4^v$ và tính biến đổi FFT N điểm của $x(n)$. Xác định thời gian thực thi với $v = 5, 6, \dots, 10$. Kiểm chứng rằng, thời gian thực thi tỷ lệ với $N \log_4 N$.

✎ Bài tập 9-14.

Làm lại bài tập 9-3 dùng hàm **fftfilt**.

Danh sách các hàm được giới thiệu trong chương 9

Các hàm tạo tín hiệu

chirp	Tạo tín hiệu tần số quét
cos	Tạo tín hiệu cos
diric	Tạo hàm Dirichlet
gauspuls	Tạo xung Gaussian tần số RF
pulstran	Tạo một chuỗi xung cùng dạng với xung gốc
sawtooth	Tạo sóng răng cưa hoặc sóng tam giác
sin	Tạo tín hiệu sin
sinc	Tạo tín hiệu sinc
square	Tạo tín hiệu sóng vuông

Các hàm vẽ tín hiệu

plot	Vẽ dạng sóng tín hiệu
-------------	-----------------------

Các hàm làm việc với các file dữ liệu

fopen	Mở một file dữ liệu
fread	Đọc một file dữ liệu
fscanf	Quét một file dữ liệu
load	Tải dữ liệu từ một file .mat vào không gian làm việc của MATLAB

Các hàm phân tích và thiết kế bộ lọc

conv	Hàm thực hiện tích chập một chiều
conv2	Hàm thực hiện tích chập hai chiều
fftfilt	Thực hiện bộ lọc trong miền tần số
filter	Hàm tính đáp ứng ra của bộ lọc
filtfilt	Thực hiện bộ lọc có khử méo pha
lactfilt	Thực hiện bộ lọc theo cấu trúc lattice
upfirdn	Thực hiện bộ lọc được lấy mẫu lại

Các hàm tính đáp ứng tần số

abs	Hàm tính biên độ
angle	Hàm xác định góc pha
freqs	Đáp ứng tần số của hệ thống liên tục
freqz	Đáp ứng tần số của hệ thống rời rạc
grpdelay	Tính độ trễ nhóm của bộ lọc
phasedelay	Tính độ trễ pha của bộ lọc
phasez	Đáp ứng pha của một hệ thống rời rạc
unwrap	Đáp ứng pha liên tục tại các vị trí $\pm 360^\circ$

Các mô hình hệ thống tuyến tính

convmtx	Hàm tính ma trận chập của một hệ thống tuyến tính
lact2tf	Chuyển từ mô hình lattice sang mô hình hàm truyền đạt
poly	Hàm tính các hệ số của đa thức từ các nghiệm của nó
residuez	Chuyển đổi qua lại giữa các mô hình thặng dư và mô hình hàm truyền đạt
roots	Tính các nghiệm của một đa thức
sos2ss	Chuyển từ mô hình các khâu bậc 2 sang mô hình không gian trạng thái
sos2tf	Chuyển từ mô hình các khâu bậc 2 sang mô hình hàm truyền đạt
sos2zp	Chuyển từ mô hình các khâu bậc 2 sang mô hình cực - zero
ss2sos	Chuyển từ mô hình không gian trạng thái sang mô hình các khâu bậc 2
ss2tf	Chuyển từ mô hình không gian trạng thái sang mô hình hàm truyền đạt
ss2zp	Chuyển từ mô hình không gian trạng thái sang mô hình cực - zero
tf2latc	Chuyển từ mô hình hàm truyền đạt sang mô hình lattice
tf2ss	Chuyển từ mô hình hàm truyền đạt sang mô hình không gian trạng thái
tf2zp	Chuyển từ mô hình hàm truyền đạt sang mô hình cực - zero
zp2sos	Chuyển từ mô hình cực - zero sang mô hình các khâu bậc 2
zp2ss	Chuyển từ mô hình cực - zero sang mô hình không gian trạng thái
zp2tf	Chuyển từ mô hình cực - zero sang mô hình hàm truyền đạt

Các hàm biến đổi Fourier

fft	Thực hiện biến đổi FFT 1 chiều
fft2	Thực hiện biến đổi FFT 2 chiều
fftshift	Thực hiện biến đổi FFT 1 chiều có sắp lại các hệ số sao cho các zero nằm ở khoảng giữa
goertzel	Thực hiện biến đổi DFT theo giải thuật Goertzel
ifft	Thực hiện biến đổi ngược FFT 1 chiều
ifft2	Thực hiện biến đổi ngược FFT 2 chiều

Chương 10

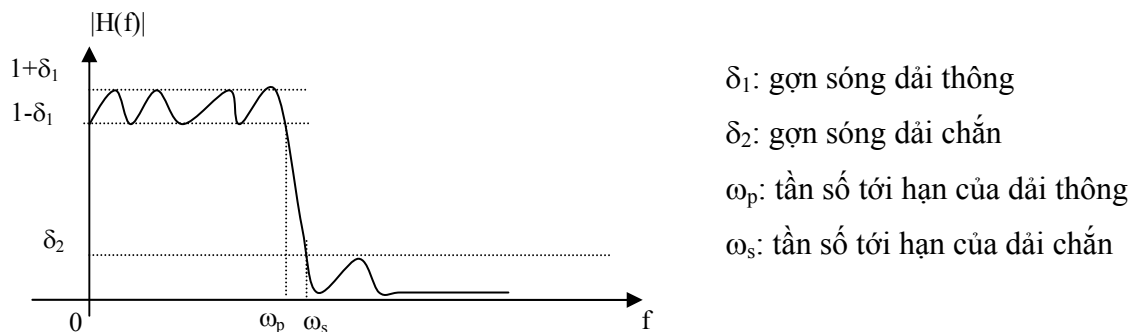
THIẾT KẾ CÁC BỘ LỌC

Thiết kế bộ lọc là một quá trình xác định các hệ số của bộ lọc sao cho thỏa mãn một số yêu cầu thiết kế. Thực hiện bộ lọc là quá trình lựa chọn và áp dụng một cấu trúc thích hợp cho bộ lọc dựa trên các hệ số đã xác định trong quá trình thiết kế. Đây là hai quá trình cơ bản cần thực hiện trước khi có thể sử dụng bộ lọc để lọc một tín hiệu nào đó.

10.1. CÁC CHỈ TIÊU THIẾT KẾ BỘ LỌC

Nói chung, một chỉ tiêu thiết kế chặt chẽ đòi hỏi bộ lọc phải thỏa mãn những yêu cầu về đáp ứng tần số bao gồm: dải thông, dải chắn, độ gợn sóng dải thông, suy hao dải chắn, độ rộng vùng chuyển tiếp. Những yêu cầu chi tiết hơn có thể là: chọn bộ lọc có bậc tối thiểu, chọn dạng đáp ứng biên độ hoặc yêu cầu sử dụng bộ lọc FIR.

Trong miền tần số, bộ lọc được đặc trưng bởi đáp ứng tần số. Dựa vào đáp ứng tần số của bộ lọc, toàn bộ dải tần số có thể được chia thành ba vùng: dải thông, dải chắn và dải chuyển tiếp. Trong thực tế, đáp ứng biên độ trong dải thông không phải là hằng số mà biến thiên trong khoảng $(1 \pm \delta_1)H_0$ với H_0 là đáp ứng biên độ trung bình. δ_1 được gọi là gợn sóng dải thông. Tương tự, đáp ứng biên độ trong dải chắn không phải luôn bằng 0 mà biến thiên trong khoảng từ 0 đến $\delta_2 H_0$ với δ_2 là độ gợn sóng dải chắn. Bộ lọc có chất lượng tốt phải có dải chuyển tiếp hẹp và các độ gợn sóng nhỏ. Hình 10.1 biểu diễn các thông số nói trên của đáp ứng tần số:



Hình 10.1. Đáp ứng biên độ của bộ lọc số

Trong MATLAB, chỉ tiêu độ gợn sóng dải chắn được thay bằng một chỉ tiêu tương tự, đó là suy hao dải chắn.

Với các chỉ tiêu đáp ứng tần số đã chọn, phương pháp thiết kế còn phụ thuộc vào chất lượng mong muốn của bộ lọc. Nếu chỉ cần chất lượng thấp, ta có thể dùng các thiết kế đơn giản; nếu đòi hỏi chất lượng cao thì phải dùng các phương pháp phức tạp hơn.

Tất cả các hàm thiết kế bộ lọc trong MATLAB đều sử dụng tần số chuẩn hoá (theo tần số đơn vị là tần số Nyquist, tức $\frac{1}{2}$ tần số lấy mẫu), do đó không cần phải cung cấp thêm thông số tần số lấy mẫu khi gọi các hàm này. Muốn chuyển từ tần số chuẩn hoá sang tần số góc, ta nhân với π ; muốn chuyển sang tần số Hertz, ta nhân với $\frac{1}{2}$ tần số lấy mẫu.

Các phương pháp thiết kế bộ lọc số được phân thành hai nhóm: thiết kế bộ lọc IIR và thiết kế bộ lọc FIR.

10.2. THIẾT KẾ BỘ LỌC IIR

Ưu điểm cơ bản của các bộ lọc IIR so với các bộ lọc FIR là: bộ lọc IIR có khả năng thoả mãn các chỉ tiêu thiết kế với bậc của bộ lọc thấp hơn so với bộ lọc FIR tương ứng. Nhược điểm của bộ lọc IIR là có tính chất pha phi tuyến. Tuy nhiên, do đặc điểm của quá trình xử lý dữ liệu trong MATLAB là mang tính chất “offline”, nghĩa là toàn bộ chuỗi dữ liệu vào đã được xác định trước khi thực hiện lọc, do đó có thể xây dựng bộ lọc không nhân quả, có pha bằng 0 (thông qua hàm **filtfilt**) để loại bỏ méo pha phi tuyến.

Các phương pháp thiết kế bộ lọc IIR được sử dụng trong MATLAB gồm có:

- Phương pháp thiết kế cổ điển dựa trên các nguyên mẫu analog, bao gồm các bộ lọc Butterworth, Chebyshev loại I, Chebyshev loại II, elliptic và Bessel.
- Phương pháp thiết kế trực tiếp (Yulewalk) tìm một bộ lọc có đáp ứng biên độ xấp xỉ một hàm mà người thiết kế mong muốn. Đây cũng là một phương pháp để xây dựng một bộ lọc thông dải nhiều băng.
- Phương pháp mô hình thông số (parametric modelling).
- Phương pháp thiết kế Butterworth tổng quát hoá

Bảng 10.1. tóm tắt các phương pháp thiết kế bộ lọc IIR cùng với các hàm được MATLAB cung cấp để thực hiện các phương pháp thiết kế này.

Bảng 10.1. Tóm tắt các phương pháp thiết kế bộ lọc IIR và các hàm MATLAB tương ứng

Phương pháp thiết kế	Mô tả	Các hàm MATLAB tương ứng
Dựa vào các bộ lọc analog	Sử dụng các cực và zero của các bộ lọc thông thấp cổ điển trong miền Laplace (liên tục), chuyển thành bộ lọc số bằng các phép biến đổi tần số và rời rạc hoá bộ lọc	Các hàm thiết kế bộ lọc hoàn chỉnh: besself, butter, cheby1, cheby2, ellip Các hàm ước lượng bậc của bộ lọc: buttord, cheblord, cheb2ord, ellipord Các hàm tạo các bộ lọc thông thấp tương tự: besselap, buttap, cheblap, cheb2ap, ellipap Các hàm thực hiện các phép biến đổi tần số: lp2bp, lp2bs, lp2hp, lp2lp Các hàm rời rạc hoá bộ lọc: bilinear,impinvar
Thiết kế trực tiếp	Thiết kế bộ lọc số trực tiếp trong miền thời gian rời rạc bằng phương pháp tiệm cận biên độ	yulewalk
Thiết kế Butterworth tổng quát	Thiết kế các bộ lọc thông thấp Butterworth có số zero nhiều hơn số cực	maxflat
Mô hình thông số	Tìm một bộ lọc số xấp xỉ một đáp ứng xấp xỉ một đáp ứng thời gian hoặc đáp ứng tần số đã định trước	Các hàm tạo mô hình trong miền thời gian: lpc, prony, stmcb Các hàm tạo mô hình trong miền tần số: invfreqs, invfreqz

10.2.1. THIẾT KẾ CÁC BỘ LỌC IIR CỔ ĐIỂN DỰA TRÊN CÁC NGUYÊN MẪU ANALOG

Nguyên tắc thiết kế bộ lọc IIR cổ điển là dựa trên sự chuyển đổi từ các bộ lọc thông thấp analog thành các bộ lọc số tương đương. Quá trình thiết kế bộ lọc IIR cổ điển bao gồm các bước sau:

- Tìm một bộ lọc thông thấp tương tự với tần số cắt bằng 1 và dùng các phép biến đổi tần số để chuyển bộ lọc nguyên mẫu này thành bộ lọc với cấu hình các dải thông và dải chắn như ta mong muốn.
- Chuyển bộ lọc nói trên sang miền số.
- Rời rạc hoá bộ lọc.

Chúng ta lần lượt khảo sát chi tiết các bước của quá trình thiết kế nói trên, đồng thời tìm hiểu các đặc tính nổi bật của mỗi loại bộ lọc IIR cổ điển.

Thiết kế các bộ lọc nguyên mẫu analog

MATLAB cung cấp các hàm để khởi tạo các bộ lọc thông thấp tương tự có tần số cắt bằng 1, đây là bước đầu tiên trong quy trình thiết kế bộ lọc IIR cổ điển. Có 5 kiểu bộ lọc thông thấp analog: bộ lọc Butterworth, bộ lọc Chebychev loại I, bộ lọc Chebychev loại II, bộ lọc elliptic và bộ lọc Bessel. Tất cả được tóm tắt trong bảng sau:

Bảng 16.2. Tóm tắt các hàm khởi tạo các bộ lọc thông thấp nguyên mẫu

Kiểu bộ lọc	Cú pháp hàm MATLAB tương ứng
Bessel	$[z, p, k] = \text{besselap}(n)$
Butterworth	$[z, p, k] = \text{buttap}(n)$
Chebychev loại I	$[z, p, k] = \text{cheb1ap}(n, R_p)$
Chebychev loại II	$[z, p, k] = \text{cheb2ap}(n, R_s)$
Elliptic	$[z, p, k] = \text{ellipap}(n, R_p, R_s)$

▪ Bộ lọc Butterworth:

Là bộ lọc toàn điểm cực, có đáp ứng tần số:

$$|H(\Omega)|^2 = \frac{1}{1 + \Omega^{2N}} \quad (10.1)$$

trong đó: N là bậc của bộ lọc

Đáp ứng tần số của bộ lọc Butterworth bằng phẳng trong các dải thông và dải chắn, là hàm đơn điệu giảm trên $[0, +\infty)$.

▪ Bộ lọc Chebychev loại I:

Bộ lọc Chebychev loại I tối thiểu hoá trị tuyệt đối của sai số giữa đáp ứng tần số thực tế và lý tưởng trên toàn bộ dải thông bằng cách đưa vào một lượng gợn sóng cân bằng trong toàn dải thông R_p (tính bằng dB). Đây là bộ lọc toàn điểm cực, có đáp ứng gợn sóng cân bằng trong dải thông và đáp ứng bằng phẳng trong dải chắn. Thời gian chuyển tiếp từ dải thông sang dải chắn nhanh hơn bộ lọc Butterworth. $|H(j\Omega)| = 10^{-R_p/20}$ tại $\Omega = 1$ với R_p là độ gợn sóng dải thông.

Đáp ứng biên độ:

$$|H(\Omega)|^2 = \frac{1}{1 + \varepsilon^2 T_N^2(\Omega)} \quad (10.2)$$

trong đó: ε là một hệ số phụ thuộc vào độ gợn sóng dải thông $\delta_1 = 10^{-Rp/20}$

$$\varepsilon^2 = \frac{1}{(1 - \delta_1)^2} - 1 \quad (10.3)$$

$T_N(x)$ là đa thức Chebychev bậc N , xác định bằng các công thức truy hồi:

$$\begin{cases} T_0(x) = 1, T_1(x) = x \\ T_{N+1}(x) = 2xT_N(x) - T_{N-1}(x) \text{ với } N = 1, 2, \dots \end{cases} \quad (10.4)$$

▪ **Bộ lọc Chebychev loại II:**

Bộ lọc Chebychev loại I tối thiểu hoá trị tuyệt đối của sai số giữa đáp ứng tần số thực tế và lý tưởng trên toàn bộ dải chắn bằng cách đưa vào một lượng gợn sóng cân bằng trong toàn dải chắn R_s (tính bằng dB). Là bộ lọc có cả cực và zero, đáp ứng bằng phẳng trong dải thông và gợn sóng trong dải chắn. Dải chắn tiến về 0 chậm hơn so với bộ lọc Chebychev loại I, thậm chí không hội tụ về 0 nếu N chẵn. Tuy nhiên, ưu điểm của nó là dải thông bằng phẳng. $|H(j\Omega)| = \delta_2 = 10^{-Rs/20}$ tại $\Omega = 1$ với R_s là độ gợn sóng dải chắn.

$$|H(\Omega)|^2 = \frac{1}{1 + \varepsilon^2 [1/T_N^2(\Omega)]} \quad (10.5)$$

Bộ lọc Chebychev đòi hỏi ít điểm cực hơn so với bộ lọc Butterworth có cùng thông số yêu cầu.

▪ **Bộ lọc elliptic:**

Là bộ lọc gợn sóng ở cả dải thông và dải chắn, đáp ứng tần số có cả cực lẫn zero và được mô tả bởi phương trình:

$$|H(\Omega)|^2 = \frac{1}{1 + \varepsilon^2 U_N(\Omega)} \quad (10.6)$$

trong đó: $U_N(x)$ là hàm elliptic Jacobian bậc N

ε là một hệ số phụ thuộc vào độ gợn sóng dải thông: $R_p = 10 \log_{10}(1 + \varepsilon^2)$ (10.7)

Do sai số được trải đều trên các dải thông và dải chắn nên bộ lọc Elliptic là bộ lọc hiệu quả nhất trên phương diện tối thiểu hóa số bậc của bộ lọc. Nếu cùng thông số bậc và các yêu cầu khác thì bộ lọc Elliptic sẽ có dải chuyển tiếp nhỏ nhất. $|H(j\Omega)| = \delta_1 = 10^{-Rp/20}$ tại $\Omega = 1$ với R_p là độ gợn sóng dải thông.

▪ **Bộ lọc Bessel:**

Là bộ lọc toàn điểm cực với hàm hệ thống:

$$H(s) = \frac{1}{B_N(s)} \quad (10.8)$$

trong đó $B_N(s)$ là đa thức Bessel bậc N , xác định bằng công thức truy hồi:

$$\begin{cases} B_0(s) = 1, B_1(s) = s + 1 \\ B_N(s) = (2N - 1)B_{N-1}(s) + s^2 B_{N-2}(s) \end{cases} \quad (10.9)$$

Bộ lọc Bessel có độ trễ nhóm phẳng tối đa tại tần số bằng 0 và gần như bằng phẳng trong toàn bộ dải thông. Do đó, bộ lọc Bessel có đáp ứng pha tuyến tính trên toàn dải thông. Đặc tính này cho phép tín hiệu sau khi lọc giữ nguyên dạng sóng trong phạm vi tần số của dải thông. Tuy nhiên đặc tính này bị mất đi khi ta chuyển sang miền số. Vì vậy MATLAB chỉ hỗ trợ cho thiết kế bộ lọc Bessel analog.

Để thỏa mãn điều kiện suy hao dải chắn, bộ lọc Bessel cần có bậc cao hơn so với các loại bộ lọc khác. Khi $\Omega = 1$ thì $|H(j\Omega)| < \frac{1}{\sqrt{2}}$ và giảm dần khi bậc của bộ lọc tăng lên.

Ví dụ 10-1. Khởi tạo một bộ lọc elliptic tương tự có bậc $n = 5$, gợn sóng dải thông $R_p = 0.5$, gợn sóng dải chắn $R_s = 20$. Vẽ đáp ứng biên độ của bộ lọc.

```
[z,p,k] = ellipap(5,0.5,20);
w = logspace(-1,1,1000);
h = freqs(k*poly(z),poly(p),w);
semilogx(w,abs(h)), grid
```

Sau khi thực thi chương trình ta có một đồ thị tương tự hình 10.2d.

Biến đổi tần số

Bước thứ hai trong quá trình thiết kế bộ lọc IIR cổ điển là dùng các phép biến đổi tần số để chuyển từ các nguyên mẫu bộ lọc thông thấp analog có tần số cắt bằng 1 thành các bộ lọc analog thông thấp, thông cao, thông dải, hoặc chắn dải có các tần số cắt như ta mong muốn. Sau đây là danh sách các hàm MATLAB để thực hiện các phép biến đổi tần số nói trên cùng với các cú pháp tương ứng của chúng.

Bảng 10.3. Các hàm thực hiện các phép biến đổi tần số

Loại biến đổi	Quy tắc biến đổi	Hàm thực hiện biến đổi
Thông thấp sang thông thấp	$s' = \frac{s}{\omega_0}$	[numt,dent] = lp2lp(num,den,Wo) [At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)
Thông thấp sang thông cao	$s' = \frac{\omega_0}{s}$	[numt,dent] = lp2hp(num,den,Wo) [At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)
Thông thấp sang thông dải	$s' = \frac{\omega_0}{BW} \frac{(s/\omega_0)^2 + 1}{(s/\omega_0)}$	[numt,dent] = lp2bp(num,den,Wo,Bw) [At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)
Thông thấp sang chắn dải	$s' = \frac{BW}{\omega_0} \frac{(s/\omega_0)}{(s/\omega_0)^2 + 1}$	[numt,dent] = lp2bs(num,den,Wo,Bw) [At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)

Từ bảng trên ta thấy rằng các phép biến đổi tần số có thể thực hiện đối với hai dạng mô hình hệ thống tuyến tính: mô hình hàm truyền đạt và mô hình không gian trạng thái. Mô hình hàm truyền đạt được biểu diễn bởi hai vector (num, den) (hoặc (numt, dent) sau khi biến đổi); còn mô hình không gian trạng thái biểu diễn bằng 4 ma trận (A, B, C, D) (hoặc (At, Bt, Ct, Dt) sau khi biến đổi).

Đối với các bộ lọc thông thấp và thông cao, ω_0 chính là tần số cắt. Với các bộ lọc thông dải và chặn dải, BW và ω_0 được xác định như sau:

$$BW = \omega_2 - \omega_1 \quad (10.10)$$

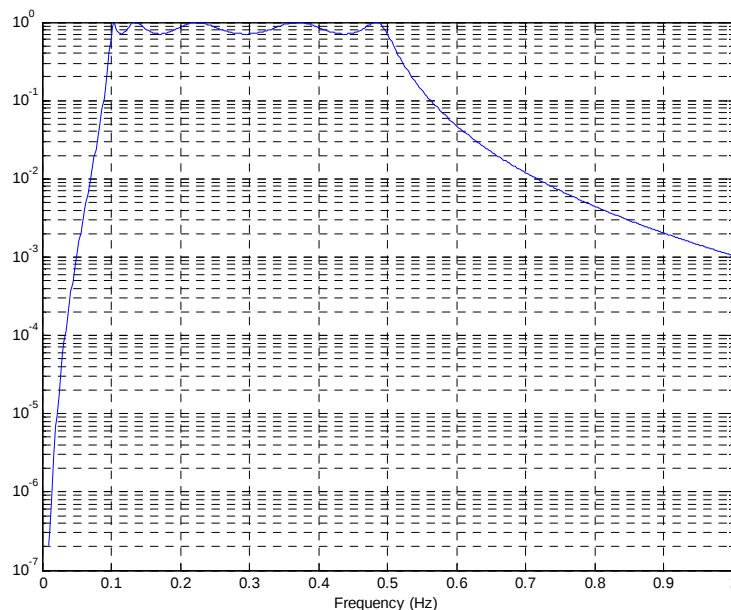
$$\omega_0 = \sqrt{\omega_1 \omega_2} \quad (10.11)$$

trong đó ω_1 và ω_2 là các tần số cắt của bộ lọc.

Trong trường hợp thông dải và chặn dải, do sử dụng phép đổi biến bậc hai nên bậc của bộ lọc mới sẽ gấp đôi bậc của bộ lọc cũ.

Ví dụ 10-2. Thiết kế một bộ lọc thông dải tương tự thuộc dạng Chebychev loại I có bậc $n = 10$, gợn sóng dải thông $R_p = 3\text{dB}$, gợn sóng dải chặn $R_s = 20$. Các tần số cắt là $\Omega_1 = \pi/5$ và $\Omega_2 = \pi$. Vẽ đáp ứng biên độ của bộ lọc.

```
[z,p,k] = cheblap(5,3); % Bộ lọc Chebychev loại I bậc 5, Rs = 3dB
[A,B,C,D] = zp2ss(z,p,k); % Chuyển sang dạng không gian trạng thái.
u1 = 0.1*2*pi; u2 = 0.5*2*pi; % Các tần số cắt (đơn vị rad/s)
Bw = u2-u1;
Wo = sqrt(u1*u2);
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw);
[b,a] = ss2tf(At,Bt,Ct,Dt); % Chuyển sang dạng hàm truyền đạt.
w = linspace(0.01,1,500)*2*pi; % Tạo vector tần số.
h = freqs(b,a,w); % Tính đáp ứng tần số.
semilogy(w/2/pi,abs(h)), grid % Vẽ đáp ứng biên độ.
xlabel('Frequency (Hz)');
```



Hình 10.3.

Rời rạc hoá bộ lọc

Bước cuối cùng trong khâu thiết kế bộ lọc IIR là chuyển bộ lọc từ miền tương tự sang miền số. Có hai phương pháp để thực hiện việc này: phương pháp bất biến xung và phương pháp biến đổi song tuyến tính. Các hàm MATLAB tương ứng với hai phương pháp này là **impinvar** và **bilinear** (xem **bảng 10.4**).

Bảng 10.4. Các hàm thực hiện rời rạc hoá bộ lọc tương tự

Biến đổi tương tự - số	Hàm thực hiện biến đổi
Bất biến xung	[numd, dend] = impinvar(num, den, fs) [numd, dend] = impinvar(num, den, fs, tol)
Biến đổi song tuyến tính	[zd, pd, kd] = bilinear(z, p, k, fs, Fp) [numd, dend] = bilinear(num, den, fs, Fp) [Ad, Bd, Cd, Dd] = bilinear(At, Bt, Ct, Dt, fs, Fp)

▪ Phương pháp bất biến xung

Nội dung của phương pháp bất biến xung là: xây dựng một bộ lọc số mà đáp ứng xung của nó là các mẫu rời rạc của đáp ứng xung của bộ lọc tương tự ban đầu. Trong MATLAB, phương pháp này được thực hiện bởi hàm **impinvar**. Hàm này chỉ chấp nhận mô hình hàm truyền đạt của bộ lọc.

Để đạt kết quả tốt nhất, bộ lọc tương tự phải không có thành phần tần số nào lớn hơn $\frac{1}{2}$ tần số lấy mẫu, vì các thành phần tần số này sẽ chồng lấn vào dải thông thấp trong quá trình lấy mẫu (tiêu chuẩn Nyquist). Phương pháp bất biến xung có thể có tác dụng đối với một số bộ lọc thông thấp và thông dải nhưng không thích hợp đối với các bộ lọc thông cao và chặn dải.

▪ Phương pháp biến đổi song tuyến tính:

Phép biến đổi song tuyến tính là một ánh xạ từ miền liên tục (miền s) sang miền số (miền z). Phép biến đổi này biến hàm truyền Laplace $H(s)$ trong miền s thành hàm truyền đạt $H(z)$ trong miền z theo quy tắc sau:

$$H(z) = H(s) \Big|_{s=k \frac{z-1}{z+1}} \quad (10.12)$$

Phép biến đổi song tuyến tính biến trục ảo $j\Omega$ trong mặt phẳng s thành vòng tròn đơn vị trong mặt phẳng z . Một điểm trên trục ảo có tung độ Ω sẽ trở thành điểm trên vòng tròn đơn vị có:

$$\omega = 2 \tan^{-1} \left(\frac{\Omega}{k} \right) \quad (10.13)$$

Phép biến đổi song tuyến tính được thực hiện bằng hàm **bilinear**. Giá trị mặc định của hệ số k là $(2 * F_s)$ (F_s là tần số lấy mẫu). Nếu cung cấp thêm thông số F_p (gọi là tần số phối hợp) thì:

$$k = \frac{2\pi f_p}{\tan(\pi f_p / f_s)} \quad (10.14)$$

Nếu có sự hiện diện của thông số F_p , phép biến đổi song tuyến tính sẽ biến đổi tần số $\Omega = 2\pi f_p$ thành cùng tần số đó trong miền z , nhưng được chuẩn hoá theo tần số lấy mẫu F_s :

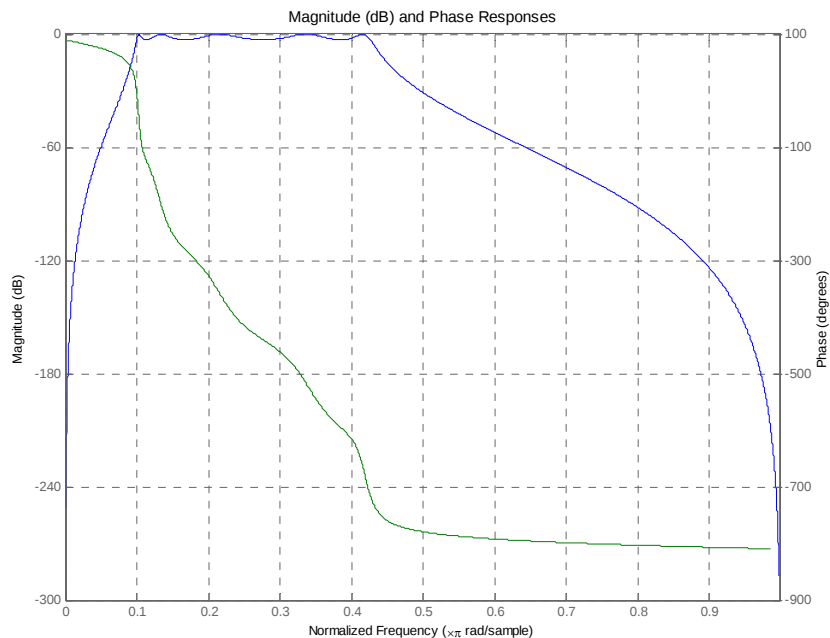
$$\omega = \frac{2\pi f_p}{f_s} \quad (10.15)$$

Hàm **bilinear** có thể thực hiện phép biến đổi tuyến tính trên ba dạng mô hình hệ thống tuyến tính: mô hình độ lợi – cực – zero, mô hình hàm truyền đạt và mô hình không gian trạng thái.

Trong ví dụ 10.2, ta có thể rời rạc hoá bộ lọc tạo được bằng cách dùng hàm **bilinear** với tần số lấy mẫu 2Hz:

```
[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,2,0.1);
[bz,az] = ss2tf(Ad,Bd,Cd,Dd); % chuyển về mô hình hàm truyền đạt.
```

Đáp ứng tần số của bộ lọc mới như sau:



Hình 10.4.

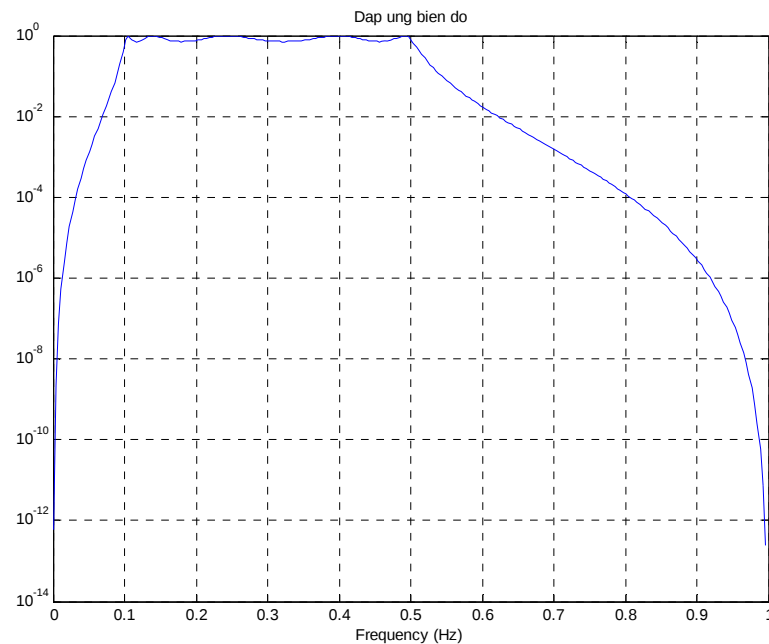
Ta thấy rằng tần số cạnh trên của dải thông nhỏ hơn 0,5Hz mặc dù, đối với bộ lọc tương tự giá trị này đúng bằng 0,5Hz. Nguyên nhân là do tính phi tuyến của bản thân phép biến đổi song tuyến tính. Để khắc phục hiện tượng này, khi xây dựng bộ lọc tương tự, ta phải tính lại các tần số ngưỡng của dải thông sao cho qua phép biến đổi song tuyến tính, nó chuyển một cách chính xác thành các tần số mong muốn như trong ví dụ sau:

Ví dụ 10-3. Hiệu chỉnh lại ví dụ 10-2 đồng thời áp dụng phép biến đổi song tuyến tính để thiết kế bộ lọc số Chebychev loại I, bậc 10, dải thông 0,1 – 0,5Hz, tần số lấy mẫu 2Hz.

```
[z,p,k] = cheblap(5,3); % Bộ lọc Chebychev loại I bậc 5, Rs = 3dB
[A,B,C,D] = zp2ss(z,p,k); % Chuyển sang dạng không gian trạng thái.
fs = 2; % Tần số lấy mẫu (hertz)
u1 = 2*fs*tan(0.1*(2*pi/fs)/2); % Tần số cắt thấp (rad/s)
u2 = 2*fs*tan(0.5*(2*pi/fs)/2); % Tần số cắt cao (rad/s)
Bw = u2 - u1; % Băng thông
Wo = sqrt(u1*u2); % Tần số trung tâm
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw);
[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,2,0.1);
[bz,az] = ss2tf(Ad,Bd,Cd,Dd); % chuyển về mô hình hàm truyền đạt.
[h,f] = freqz(bz,az,256,2); % Tính đáp ứng tần số.
semilogy(f,abs(h)), grid % Vẽ đáp ứng biên độ.
```

```
xlabel('Frequency (Hz)');
title('Dap ung bien do');
```

Kết quả như sau:



Hình 10.5.

Các hàm thiết kế bộ lọc IIR hoàn chỉnh

Để thuận tiện cho người sử dụng khi thực hiện quá trình thiết kế các bộ lọc IIR cổ điển, MATLAB cũng cung cấp các hàm thiết kế hoàn chỉnh các bộ lọc IIR thay vì phải thực hiện từng bước quy trình thiết kế như ở phần trên. Các hàm này tích hợp các quy trình đã đề cập trên đây lại, trong đó ở bước thứ ba của quá trình, tức là bước rời rạc hoá các bộ lọc tương tự, các hàm này đều sử dụng phương pháp biến đổi song tuyến tính.

▪ **Bộ lọc Butterworth:** dùng hàm **butter**:

```
>> [b,a] = butter(n,Wn,form)
>> [z,p,k] = butter(n,Wn,form)
>> [A,B,C,D] = butter(n,Wn,form)
```

trong đó:

W_n là vector gồm 2 phần tử xác định các tần số cắt (với các bộ lọc thông thấp, thông cao, W_n trở thành một vô hướng). W_n được chuẩn hoá theo $\frac{1}{2}$ tần số lấy mẫu.

$form$ là một chuỗi xác định dạng bộ lọc: 'low' (bộ lọc thông thấp), 'high' (thông cao), 'stop' (chặn dải). Nếu W_n có hai phần tử và không có thông số $form$, hàm sẽ tạo ra bộ lọc thông dải.

Để tạo các bộ lọc tương tự, ta thêm vào chuỗi 's' vào danh sách các thông số.

MATLAB cũng cung cấp hàm **buttord** để ước lượng bậc bộ lọc tối thiểu thoả mãn các yêu cầu thiết kế.

```
>> [N, Wn] = buttord(Wp, Ws, Rp, Rs)
```

Các thông số nhập của nó bao gồm các tần số ngưỡng của dải thông và dải chặn W_p và W_s , các giá trị mong muốn của gợn sóng dải thông tối đa và suy hao tối thiểu của dải chặn (R_p , R_s). Hàm trả về bậc tối thiểu của bộ lọc cùng với vector W_n để cung cấp cho hàm **butter**.

▪ **Bộ lọc Chebychev loại I:** dùng hàm **cheby1**:

```
>> [b,a] = cheby1(n,Rp,Wn,form)
>> [z,p,k] = cheby2(n,Rp,Wn,form)
>> [A,B,C,D] = cheby2(n,Rp,Wn,form)
```

Các thông số nhập tương tự như hàm **butter**, thêm vào thông số độ gợn sóng dải thông R_p (tính bằng dB). Để ước lượng bậc bộ lọc, dùng hàm **cheblord**.

▪ **Bộ lọc Chebychev loại II:** dùng hàm **cheby2**:

```
>> [b,a] = cheby2(n,Rs,Wn,form)
>> [z,p,k] = cheby2(n,Rs,Wn,form)
>> [A,B,C,D] = cheby2(n,Rs,Wn,form)
```

trong đó R_s là độ gợn sóng dải chặn (tính bằng dB). Hàm **cheb2ord** cho phép ước lượng bậc của bộ lọc.

▪ **Bộ lọc elliptic:** dùng hàm **ellip**:

```
>> [b,a] = ellip(n,Rp,Rs,Wn,form)
>> [z,p,k] = ellip(n,Rp,Rs,Wn,form)
>> [A,B,C,D] = ellip(n,Rp,Rs,Wn,form)
```

Hàm **ellipord** cho phép ước lượng bậc của bộ lọc.

▪ **Bộ lọc Bessel:** dùng hàm **besself** (chỉ thiết kế bộ lọc tương tự):

```
>> [b,a] = besself(n,Wn,form)
>> [z,p,k] = besself(n,Wn,form)
>> [A,B,C,D] = besself(n,Wn,form)
```

Không có hàm ước lượng bậc của bộ lọc.

 **Ví dụ 10-4.** Thiết kế một bộ lọc thông dải có băng thông từ 1000Hz – 2000Hz, dải chặn bắt đầu cách các tần số trên 500Hz, gợn sóng dải thông tối đa là 1dB, suy hao dải chặn tối thiểu là 60dB. Tần số lấy mẫu là 10kHz. Sử dụng bộ lọc Butterworth.

```
>> [n,Wn] = buttord([1000 2000]/5000,[500 2500]/5000,1,60); % Ước lượng bậc bộ
lọc
% và các tần số cắt
n =
    12
Wn =
    0.1951 0.4080
>> [b,a] = butter(n,Wn); % Thiết kế bộ lọc
```

10.2.2. THIẾT KẾ TRỰC TIẾP CÁC BỘ LỌC IIR TRONG MIỀN SỐ

Đây là phương pháp thiết kế trực tiếp trong miền rời rạc dựa vào một đáp ứng tần số mong muốn mà không dựa vào các bộ lọc analog. Phương pháp thiết kế này không bị bó buộc trong

các dạng cấu hình thông thấp, thông cao, thông dải hay chặn dải mà ta có thể thiết kế một bộ lọc số với đáp ứng tần số bất kỳ, có thể là gồm nhiều dải tần số.

Phương pháp này được thực hiện với hàm **yulewalk**. Giải thuật mà hàm này sử dụng là: tìm biến đổi ngược FFT của phổ công suất lý tưởng mong muốn và giải hệ phương trình Yule-Walker sử dụng các mẫu dữ liệu của hàm tự tương quan tìm được. Bộ lọc IIR được thiết kế theo phương pháp này là bộ lọc IIR đệ quy với phương pháp bình phương cực tiểu.

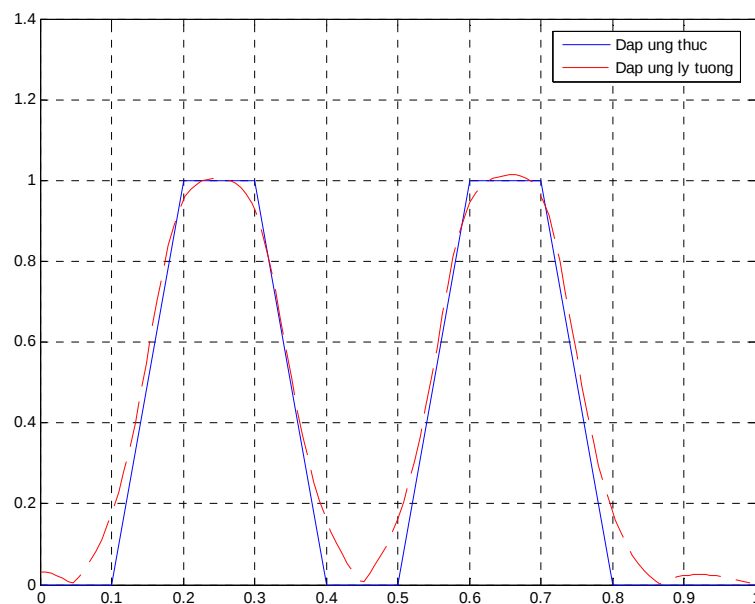
Cú pháp:

```
>> [b,a] = yulewalk(n,f,m)
```

trong đó n là bậc bộ lọc, F là vector tần số và M là vector đáp ứng biên độ tương ứng với F . Phương pháp này không quan tâm đến đáp ứng pha. (b,a) là các vector hệ số của bộ lọc được thiết kế.

 **Ví dụ 10-5.** Dùng hàm **yulewalk** thiết kế một bộ lọc có hai băng, vẽ đáp ứng tần số thực tế và đáp ứng tần số mong muốn.

```
m = [0 0 1 1 0 0 1 1 0 0];
f = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 1];
[b,a] = yulewalk(10,f,m);
[h,w] = freqz(b,a,128)
plot(f,m,w/pi,abs(h)); grid
```



Hình 10.6.

10.2.3. THIẾT KẾ BỘ LỌC BUTTERWORTH TỔNG QUÁT

Hàm **maxflat** cho phép ta thiết kế các bộ lọc Butterworth, tức là bộ lọc Butterworth với số cực và số zero khác nhau. Điều này có lợi trong một số trường hợp khi mà việc tính toán các cực phức tạp hơn so với các zero. Cú pháp hàm **maxflat** tương tự như hàm **butter**, nhưng khác ở chỗ ta có thể cung cấp hai thông số bậc của bộ lọc, một cho đa thức tử số và một cho đa thức mẫu số.

Bộ lọc này có đặc điểm là có độ phẳng tối đa, có nghĩa nó được tối ưu hoá sao cho với mọi giá trị của bậc của các đa thức tử số thì đạo hàm của đáp ứng tần số tại tần số $\Omega = 0$ và $\Omega = \pi$ đều bằng 0.

```
>> [b,a] = maxflat(nb,na,wn)
```

10.2.4. PHƯƠNG PHÁP MÔ HÌNH THÔNG SỐ

Nội dung của phương pháp mô hình thông số là dựa vào một số thông tin nào đó về hệ thống (ví dụ đáp ứng ngõ ra ứng với một ngõ vào cho trước, hoặc đáp ứng xung, đáp ứng tần số, ...) để xác định một cách gần đúng hệ thống đó. Phương pháp này có thể ứng dụng để thiết kế các bộ lọc số dựa trên các thông tin về miền thời gian hoặc miền tần số của bộ lọc.

Thiết kế trong miền thời gian

Nếu ta biết đáp ứng xung của bộ lọc giống với một chuỗi x cho trước, ta có thể dùng phương pháp mã hoá dự đoán tuyến tính (LPC - Linear Predictive Coding) để thiết kế bộ lọc toàn cực thoả mãn điều kiện trên:

```
>> [a,e] = lpc(x,n)
```

a là vector hệ số của bộ lọc toàn cực, e là variance của sai số dự đoán, n là bậc của bộ dự đoán tuyến tính.

Hoặc cũng có thể dùng phương pháp Prony để thiết kế bộ lọc IIR thoả mãn yêu cầu trên:

```
>> [b,a] = prony(h, nb, na)
```

Hàm này trả về các hệ số của bộ lọc số có bậc của các đa thức tử và mẫu lần lượt là nb , na và đáp ứng xung là vector h .

Một phương pháp khác để xác định các hệ số của bộ lọc là phương pháp lặp Steiglitz-McBride. Phương pháp này cũng có thể dùng trong trường hợp ta biết được đáp ứng ngõ ra đối với một chuỗi vào xác định.

```
>> [b,a] = stmcb(h,nb,na,n,ai)
```

Ngoài các thông số đã giới thiệu ở trên, hàm này có thêm thông số n là số lần lặp của phương pháp, ai là vector dự đoán ban đầu của a . Nếu không cung cấp thông số này thì hàm sẽ chọn mặc định ai từ $[b, ai] = \text{prony}(h, 0, na)$.

```
>> [b,a] = stmcb(y,x,nb,na,n,ai)
```

Trong trường hợp này ta nhập các chuỗi vào x và chuỗi ra y tương ứng thay vì đáp ứng xung h .

Thiết kế trong miền tần số

Có thể thiết kế bộ lọc số dựa vào đáp ứng tần số của nó bằng cách dùng hàm **invfreqz** của MATLAB.

```
>> [b,a] = invfreqz(H,w,nb,na,wt,iter,tol,'trace')
```

Hàm này trả về một bộ lọc với các hệ số thực có đáp ứng tần số phức được xác định bởi cặp vector (H,w) : w là vector tần số gồm các tần số góc chuẩn hoá trong khoảng $[0,\pi]$ (đơn vị rad/s) còn H là vector đáp ứng tần số phức tại các điểm tần số xác định bởi w . (na, nb) là bậc của các đa thức tử và mẫu của bộ lọc. Bốn thông số nói trên là bốn thông số cơ bản của hàm **invfreqz**.

Các thông số còn lại là các thông số không bắt buộc. w_t là một vector cùng chiều dài với w để đánh giá trọng số của sai số tại các điểm tần số khảo sát. Hàm **invfreqz** sẽ tìm cách tối thiểu hoá đại lượng $(B - H.A)^2 \cdot w_t$ (mặc định w_t là vector đơn vị). *iter* xác định số lần lặp tối đa của giải thuật. *tol* xác định sai số cho phép. Giải thuật sẽ dừng khi sai số nhỏ hơn giá trị *tol* (mặc định 0.01). Chuỗi 'trace' được cung cấp để yêu cầu hàm **invfreqz** tạo ra các báo cáo về từng bước lặp.

Nếu cần thiết kế bộ lọc phức, ta dùng cú pháp:

```
>> [b,a] = invfreqz(h,w,'complex',nb,na,...)
```

trong đó vector w gồm các phần tử nằm trong khoảng $[-\pi, \pi]$.

10.3. THIẾT KẾ BỘ LỌC FIR

Các bộ lọc FIR là các bộ lọc có đáp ứng xung hữu hạn và là bộ lọc toàn zero. So với bộ lọc IIR, các bộ lọc IIR có những ưu điểm nổi bật sau:

- Pha tuyến tính
- Luôn luôn ổn định
- Phương pháp thiết kế nói chung là tuyến tính
- Có thể thực hiện một cách hiệu quả bằng phần cứng
- Đáp ứng quá độ chỉ tồn tại trong thời gian hữu hạn

Tuy nhiên, nhược điểm lớn nhất của bộ lọc FIR so với bộ lọc IIR là bậc của bộ lọc FIR phải lớn hơn nhiều so với bộ lọc IIR với cùng một chỉ tiêu chất lượng. Do đó độ trễ của bộ lọc FIR cũng cao hơn.

Các phương pháp thiết kế bộ lọc FIR được sử dụng trong MATLAB được tóm tắt trong bảng dưới đây:

Bảng 10.5. Các phương pháp thiết kế bộ lọc IIR và các hàm MATLAB tương ứng

Phương pháp thiết kế	Mô tả	Các hàm thiết kế
Phương pháp cửa sổ	Sử dụng các cửa sổ để giới hạn đáp ứng xung, tức biến đổi FFT ngược của đáp ứng tần số lý tưởng	<code>fir1</code> , <code>fir2</code> , <code>kaiserord</code>
Nhiều dải tần và các dải chuyển tiếp	Gọn sóng cân bằng hoặc cực tiểu hoá bình phương sai số ở các băng con của đáp ứng tần số	<code>firls</code> , <code>firpm</code> , <code>firpmord</code>
Giới hạn bình phương cực tiểu	Tối thiểu hoá sai số bình phương tích lũy trên toàn dải tần số về giá trị tối đa cho phép	<code>fircls</code> , <code>fircls1</code>
Đáp ứng tuỳ định	Thiết kế bộ lọc tuỳ theo đáp ứng tần số, kể cả các bộ lọc phức và bộ lọc có pha phi tuyến	<code>cfirpm</code>
Raised-Cosine	Đáp ứng của bộ lọc thông thấp với vùng chuyển tiếp có dạng sine	<code>firrcos</code>

10.3.1. CÁC BỘ LỌC CÓ PHA TUYẾN TÍNH

Hầu hết các bộ lọc FIR được thiết kế trong MATLAB đều có pha tuyến tính, trừ các bộ lọc thiết kế bằng hàm **cfirpm**. Dựa vào tính chất đối xứng của các hệ số của bộ lọc FIR (còn gọi là các tap), người ta định nghĩa bốn loại bộ lọc FIR có pha tuyến tính như sau:

Bảng 10.6. Phân loại các bộ lọc FIR có pha tuyến tính

Loại bộ lọc	Bậc bộ lọc	Tính đối xứng của các hệ số bộ lọc	Đáp ứng tần số tại $f = 0$	Đáp ứng tần số tại $f = 1$
Loại I	Chẵn	Chẵn:	Không ràng buộc	Không ràng buộc
Loại II	Lẻ	$b(k) = b(n + 2 - k), k = 1, 2, \dots, n + 1$	Không ràng buộc	$H(1) = 0$
Loại III	Chẵn	Lẻ:	$H(0) = 0$	$H(1) = 0$
Loại IV	Lẻ	$b(k) = -b(n + 2 - k), k = 1, 2, \dots, n + 1$	$H(0) = 0$	Không ràng buộc

Độ trễ nhóm và độ trễ pha của bộ lọc FIR có pha tuyến tính bằng nhau và bằng một hằng số trên toàn bộ băng tần của bộ lọc. Nếu bậc của bộ lọc là n thì độ trễ nhóm sẽ bằng $n/2$, tức là tín hiệu sau khi lọc sẽ bị trễ đi $n/2$ bước so với tín hiệu vào. Đặc điểm này giúp bảo toàn dạng sóng tín hiệu trong dải thông, nghĩa là tín hiệu không bị méo pha.

Trong trường hợp mặc định, các hàm **fir1**, **fir2**, **firls**, **firpm**, **fircls**, **fircls1**, và **firrcos** đều thiết kế các bộ lọc FIR loại I và II. Do đặc điểm $H(1) = 0$, nên bộ lọc **fir1** không thể thiết kế các bộ lọc thông cao và chặn dải thuộc loại II. Thay vào đó, nếu n lẻ, hàm **fir1** sẽ tự động cộng thêm 1 vào bậc bộ lọc để trở thành loại I. Các hàm **firls** và **firpm** thiết kế các bộ lọc loại III và IV. Hàm **cfirpm** có thể thiết kế bất kỳ bộ lọc loại nào, kể cả bộ lọc có pha phi tuyến.

10.3.2. PHƯƠNG PHÁP CỬA SỔ (WINDOWING)

Xét bộ lọc thông thấp lý tưởng có tần số cắt là ω_0 (rad/s), có đáp ứng biên độ bằng 1 ở mọi tần số nhỏ hơn ω_0 và bằng 0 ở các tần số còn lại. Bộ lọc này có đáp ứng xung là:

$$h(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H(\omega) e^{j\omega n} d\omega = \frac{1}{2\pi} \int_{-\omega_0}^{\omega_0} e^{j\omega n} d\omega = \frac{\omega_0}{\pi} \text{sinc}\left(\frac{\omega_0}{\pi} n\right) \quad (10.16)$$

Bộ lọc này không thể thực hiện được trong thực tế vì nó có đáp ứng xung vô hạn và không nhân quả. Để tạo một bộ lọc đáp ứng xung hữu hạn có đáp ứng gần giống đáp ứng lý tưởng nêu trên, ta dùng một cửa sổ thời gian hữu hạn để giới hạn đáp ứng xung $h(n)$. Ta có thể tạo ra một bộ lọc có pha tuyến tính bằng cách chỉ giữ lại phần trung tâm của đáp ứng xung $h(n)$. Ví dụ, để thiết kế bộ lọc thông thấp bậc 51, có tần số cắt bằng $0,4\pi$, ta chọn các hệ số của nó như sau:

```
>> b = 0.4*sinc(0.4*(-25:25));
```

Trong trường hợp trên ta đã dùng cửa sổ chữ nhật để tác động lên $h(n)$. Trong trường hợp tổng quát, đáp ứng xung của bộ lọc FIR sẽ là:

$$h(n) = h_0(n) \cdot w(n) \quad (10.17)$$

trong đó $h_0(n)$ là đáp ứng xung lý tưởng còn $w(n)$ là cửa sổ thời gian hữu hạn. Các hàm cửa sổ khác nhau cùng với hàm MATLAB tương ứng được trình bày trong **bảng 10.6**.

Bảng 10.6. Các loại cửa sổ và hàm MATLAB tương ứng

Tên cửa sổ	Biểu thức thời gian	Hàm MATLAB tương ứng
Chữ nhật	$w = \text{ones}(n,1);$	$w = \text{rectwin}(n)$

Tam giác	Với n lẻ: $w(k) = \begin{cases} \frac{2k}{n+1} & 1 \leq k \leq \frac{n+1}{2} \\ \frac{2(n-k+1)}{n+1} & \frac{n+1}{2} < k \leq n \end{cases}$ Với n chẵn: $w(k) = \begin{cases} \frac{2k-1}{n} & 1 \leq k \leq \frac{n}{2} \\ \frac{2(n-k)+1}{n} & \frac{n}{2} < k \leq n \end{cases}$	$w = \text{triang}(n)$
Hamming	$w(k+1) = 0,54 - 0,46 \cdot \cos\left(2\pi \frac{k}{n-1}\right), k = \overline{0, n-1}$	$w = \text{hamming}(n, \text{sflag})$
Kaiser	$w(k) = \frac{I_0\left(\beta \sqrt{k(n-1-k)} / [n/2]\right)}{I_0(\beta)}, k = \overline{0, n-1}$ $\beta = \begin{cases} 0,1102(\alpha - 8,7) & \alpha \geq 50 \\ 0,5842(\alpha - 21)^{0,4} + 0,07886(\alpha - 21) & 21 < \alpha < 50 \\ 0 & \alpha \leq 21 \end{cases}$ với α là độ gọn sóng tối đa theo dB	$w = \text{kaiser}(n, \text{bta})$
Bartlett	Với n lẻ: $w(k+1) = \begin{cases} \frac{2k}{n-1} & 0 \leq k \leq \frac{n-1}{2} \\ 2 - \frac{2k}{n-1} & \frac{n-1}{2} < k \leq n-1 \end{cases}$ Với n chẵn: $w(k+1) = \begin{cases} \frac{2k}{n-1} & 0 \leq k \leq \frac{n}{2} - 1 \\ \frac{2(n-k-1)}{n-1} & \frac{n}{2} < k \leq n-1 \end{cases}$	$w = \text{bartlett}(n)$
Hann	$w(k+1) = 0,5(1 - \cos\left(2\pi \frac{k}{n-1}\right)), k = \overline{0, n-1}$	$w = \text{hann}(n, \text{sflag})$
Bartlett-Hanning	$w(k+1) = 0,62 - 0,48 \cdot \left \frac{k}{n-1} - 0,5 \right + 0,38 \cdot \cos\left(2\pi \left(\frac{k}{n-1} - 0,5 \right)\right), k = \overline{0, n-1}$	$w = \text{barthannwin}(n)$
Blackman	$w(k+1) = 0,42 - 0,5 \cdot \cos\left(\frac{2\pi k}{n-1}\right) + 0,08 \cdot \cos\left(\frac{4\pi k}{n-1}\right), k = \overline{0, n-1}$	$w = \text{blackman}(n, \text{sflag})$
Blackman - Harris	$w(k+1) = 0,35875 - 0,48829 \cdot \cos\left(\frac{2\pi k}{n-1}\right) + 0,14128 \cdot \cos\left(\frac{4\pi k}{n-1}\right) +$ $0,01168 \cdot \cos\left(\frac{6\pi k}{n-1}\right), k = \overline{0, n-1}$	$w = \text{blackmanharris}(n)$

Bohman	$w(k+1) = \left[1 - \frac{k - \frac{N}{2}}{\frac{N}{2}}\right] \cos\left[\pi \frac{k - \frac{N}{2}}{\frac{N}{2}}\right] + \frac{1}{\pi} \sin\left[\pi \frac{k - \frac{N}{2}}{\frac{N}{2}}\right]$ với $0 \leq k \leq n$	$w = \text{bohmanwin}(n)$
Chebychev		$w = \text{chebwin}(n, r)$
Gauss	$w(k+1) = \exp\left[-\frac{1}{2}\left(\alpha \frac{k - N/2}{N/2}\right)^2\right]$ với $0 \leq k \leq N, \alpha \geq 2$	$w = \text{gausswin}(n, \alpha)$
Parzen (de la Valle-Poussin)	$w(k) = \begin{cases} 1 - 6\left(\frac{k}{N/2}\right)^2 \left(1 - \frac{ k }{N/2}\right) & 0 \leq k \leq \frac{N}{4} \\ 2\left(1 - \frac{ k }{N/2}\right)^3 & \frac{N}{4} \leq k \leq \frac{N}{2} \end{cases}$	$w = \text{parzenwin}(n)$
Tukey	$w(k) = \begin{cases} \frac{1}{2} \left[1 + \cos\left(\frac{2\pi(k-1)}{r(n-1)} - \pi\right)\right] & k < \frac{r}{2}(n-1) + 1 \\ 1 & \frac{r}{2}(n-1) + 1 \leq k \leq n - \frac{r}{2}(n-1) \\ \frac{1}{2} \left[1 + \cos\left(\frac{2\pi}{r} - \frac{2\pi(k-1)}{r(n-1)} - \pi\right)\right] & n - \frac{r}{2}(n-1) < k \end{cases}$ với $k = \overline{1, n}$	$w = \text{tukeywin}(n, r)$
Flat top	$w(k) = 1 - 1.93 \cos\left(\frac{2\pi t}{T}\right) + 1.29 \cos\left(\frac{4\pi t}{T}\right) -$ $-0.388 \cos\left(\frac{6\pi t}{T}\right) + 0.0322 \cos\left(\frac{8\pi t}{T}\right) \text{ nếu } 0 \leq t \leq T,$ $w(k) = 0$ với t khác	$w = \text{flattopwin}(n, \text{sflag})$
Nuttall	$w(k+1) = 0,3635819 - 0,4891775 \cdot \cos\left(\frac{2\pi k}{n-1}\right) +$ $0,1365995 \cdot \cos\left(\frac{4\pi k}{n-1}\right) + 0,0106411 \cdot \cos\left(\frac{6\pi k}{n-1}\right), k = \overline{0, n-1}$	$w = \text{nuttallwin}(n)$

sflag: phương pháp lấy mẫu cửa sổ: ‘symmetric’ (đối xứng), ‘periodic’ (tuần hoàn)

Cửa sổ chữ nhật là loại cửa sổ đơn giản nhất. Tuy nhiên, nếu dùng cửa sổ này thì đáp ứng tần số của bộ lọc sẽ có gợn sóng trong dải thông và gợn sóng đặc biệt lớn ở vùng chuyển tiếp. Hiện tượng này gọi là hiện tượng Gibbs, nó không giảm khi tăng số bậc của bộ lọc mà nó chỉ được khắc phục khi ta dùng một số hàm cửa sổ khác không phải cửa sổ chữ nhật.

▪ Để hỗ trợ việc thiết kế các bộ lọc FIR bằng phương pháp cửa sổ, MATLAB cung cấp hai hàm **fir1** và **fir2**. Các hàm này trả về đáp ứng xung của bộ lọc FIR đã được cửa sổ hoá. Hai hàm này được mặc định sử dụng cửa sổ Hamming, nhưng cũng có thể chấp nhận các cửa sổ khác.

```
>> b = fir1(N,wn,form,win)
```

trong đó, N là bậc của bộ lọc, win là vector cửa sổ thời gian có chiều dài bằng $N + 1$. Vector wn cùng với chuỗi ký tự $form$ sẽ định nghĩa dạng bộ lọc theo cách sau:

○ Nếu wn chỉ có một phần tử thì đó chính là tần số cắt của bộ lọc. $form = 'high'$ nếu là bộ lọc thông cao, $form = 'low'$ nếu là bộ lọc thông thấp.

○ Nếu wn có hai phần tử: $wn = [w1 \ w2]$ thì hàm **fir1** tạo ra bộ lọc thông dải có các tần số cắt là $w1$ và $w2$. Nếu $form = 'stop'$ thì **fir1** tạo ra bộ lọc chắn dải có các tần số cắt $w1, w2$.

○ Nếu wn là một vector n phần tử: $wn = [w1 \ w2 \ \dots \ wn]$, hàm **fir1** sẽ tạo ra một bộ lọc nhiều dải thông và dải chắn xen kẽ nhau: $0 < w < w1, w1 < w < w2, \dots, wn < w < 1$. Trong đó, nếu $form = 'DC-1'$ thì dải đầu tiên sẽ là dải thông, còn nếu $form = 'DC-0'$ thì dải đầu tiên là dải chắn.

MATLAB còn cung cấp hàm **kaiserord** để ước lượng các thông số cần cung cấp cho hàm **fir1** (chỉ thích hợp với cửa sổ Kaiser), chỉ cần cung cấp đáp ứng tần số mong muốn.

```
>> [N,wn,bta,filtype] = kaiserord(f,a,dev,fs)
```

trong đó bta là hệ số β của cửa sổ Kaiser, $filtype$ là kiểu bộ lọc. f là vector tần số và a là vector đáp ứng biên độ tương ứng, dev là vector chứa các độ lệch cực đại cho phép trong các dải tần, fs là tần số lấy mẫu (mặc định bằng 2).

Nếu như hàm **fir1** chỉ cho phép thiết kế các bộ lọc FIR với các cấu hình các dải tần theo các dạng chuẩn như thông thấp, thông cao, thông dải,... thì hàm **fir2** cho phép thiết kế bộ lọc FIR với dạng đáp ứng tần số tùy định.

```
>> b = fir2(N,f,a,win)
```

f và a xác định đáp ứng tần số mong muốn: f là vector các điểm tần số còn a là vector đáp ứng biên độ tại các điểm đó. win là vector cửa sổ thời gian, là kết quả của các hàm cửa sổ đã đề cập ở trên. Bộ lọc được tạo ra là bộ lọc đối xứng, nghĩa là $b(k) = b(n + 2 - k)$, $k = 1, 2, \dots$. Với các bộ lọc có đáp ứng khác 0 tại tần số $F_s/2$, bậc của bộ lọc phải là số chẵn, nếu N lẻ, bậc của bộ lọc sẽ tự động tăng lên 1.

10.3.3. THIẾT KẾ BỘ LỌC FIR NHIỀU DẢI TẦN VỚI CÁC DẢI CHUYỂN TIẾP

Các hàm **firls** và **firpm** cho phép xác định bộ lọc lý tưởng mong muốn một cách tổng quát hơn so với các hàm **fir1** và **fir2**. Các hàm này cho phép thiết kế các bộ biến đổi Hilbert, các bộ vi phân và các bộ lọc khác với các hệ số có tính chất đối xứng lẻ (các bộ lọc FIR pha tuyến tính loại III và IV). Các hàm trên còn cho phép xác định các vùng tần số mà ta không quan tâm đến sai số (nghĩa là không cần tối thiểu hoá sai số so với bộ lọc lý tưởng trong vùng này), ví dụ như các dải chuyển tiếp; hoặc đưa vào các trọng số cho các dải tần khi tối thiểu hoá sai số.

Hàm **firls** là sự mở rộng của các hàm **fir1** và **fir2** trên cơ sở sự tối thiểu hoá bình phương sai số giữa đáp ứng thực và đáp ứng lý tưởng tích lũy trong toàn bộ dải tần của bộ lọc.

Hàm **firpm** thực hiện giải thuật Parks – McClellan, đó là sự kết hợp của thuật toán Remez với lý thuyết xấp xỉ Chebychev để tối thiểu hoá sai số cực đại giữa đáp ứng tần số thực và đáp

ứng tần số lý tưởng. Do đó các bộ lọc này còn gọi là bộ lọc minimax. Một tính chất của các bộ lọc này là có độ gợn sóng cân bằng, nên còn được gọi là các bộ lọc gợn sóng cân bằng (equiripple).

Cú pháp của hai hàm nói trên hoàn toàn tương tự nhau. Chúng chỉ khác nhau về giải thuật thực hiện. Chúng ta sẽ lần lượt tìm hiểu các cú pháp này trong từng trường hợp ứng dụng cụ thể.

Các cấu hình cơ bản

Chế độ hoạt động cơ bản nhất của các hàm **firls** và **firpm** là thiết kế các bộ lọc FIR pha tuyến tính loại I và II (phụ thuộc vào bậc của bộ lọc là chẵn hay lẻ) với cú pháp như sau:

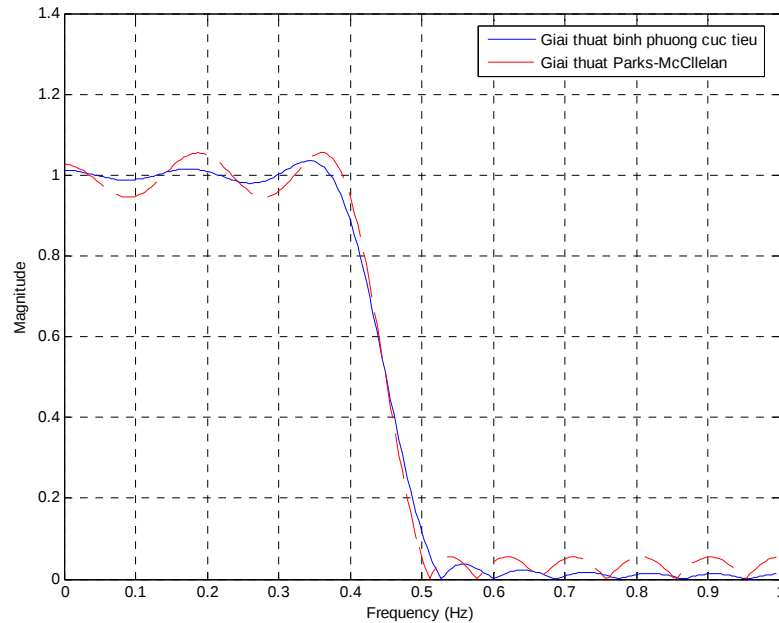
```
>> b=firpm(N,f,a)
```

```
>> b=firls(N,f,a)
```

N là bậc của bộ lọc còn (f,a) là cặp vector xác định đáp ứng tần số mong muốn của bộ lọc (giống như hàm **fir1** và **fir2**).

 **Ví dụ 10-6.** Thiết kế bộ lọc FIR thông thấp bậc 20 có đáp ứng tần số xấp xỉ bằng 1 trong khoảng tần số từ 0Hz đến 0,4Hz và xấp xỉ bằng 0 trong khoảng tần số từ 0,5Hz đến 1Hz (giả sử tần số lấy mẫu là 2Hz) bằng hai phương pháp: bình phương cực tiểu và Parks-McClellan.

```
n = 20; % Bậc bộ lọc
f = [0 0.4 0.5 1]; % Các tần số ngưỡng của các dải tần
a = [1 1 0 0]; % Biên độ mong muốn
b = firpm(n,f,a); % Thiết kế bằng giải thuật Parks-McClellan
[h1,f1] = freqz(b,1,256,2); % Tính đáp ứng tần số
bb = firls(n,f,a); % Thiết kế bằng giải thuật bình phương cực tiểu
[h2,f2] = freqz(bb,1,256,2); % Tính đáp ứng tần số
plot(f2,abs(h2),'b-',f1,abs(h1),'r--');grid % Vẽ đáp ứng biên độ
legend('Giai thuat binh phuong cuc tieu','Giai thuat Parks-McClellan');
xlabel('Frequency (Hz)');
ylabel('Magnitude')
```



Hình 10.7.

Ta thấy rằng bộ lọc thiết kế bằng **firpm** có gợn sóng cân bằng. Bộ lọc thiết kế bằng **firls** có đáp ứng tốt hơn trong các dải thông và dải chắn nhưng kém hơn so với **firpm** ở dải chuyển tiếp.

Vector trọng số

Các hàm **firls** và **firpm** đều cho phép ta nhấn mạnh mức độ tối thiểu hoá ở một số dải tần so với các dải tần khác. Để thực hiện điều này ta chỉ cần cung cấp thêm một vector trọng số **w**. Vector này có chiều dài bằng một nửa chiều dài vector **a** và **f** để bảo đảm mỗi dải tần đều có đúng một trọng số. Cú pháp trong trường hợp này là:

```
>> b=firpm(N,f,a,w)
```

```
>> b=firls(N,f,a,w)
```

Trở lại ví dụ 16.6, giả sử ta muốn độ gợn sóng trong dải chắn phải nhỏ hơn 10 lần so với trong dải thông, khi đó ta đưa thêm vào vector trọng số **w** và gọi lại các hàm **firpm** và **firls** như sau:

```
>> w = [1 10];
```

```
>> b=firpm(N,f,a,w)
```

Các bộ lọc phản đối xứng – Biến đổi Hilbert

Nếu gọi hàm **firpm** và **firls** kèm theo một thông số chuỗi ký tự **'h'** hoặc **'Hilbert'** thì kết quả trả về sẽ là đáp ứng xung của bộ lọc FIR có tính đối xứng lẻ, tức là bộ lọc có pha tuyến tính loại III hoặc IV. Bộ biến đổi Hilbert lý tưởng là bộ lọc có tính đối xứng lẻ như trên và có biên độ bằng 1 trên toàn bộ dải tần số.

Ví dụ 10-7. Thiết kế bộ biến đổi Hilbert gần đúng, sử dụng nó để tìm biến đổi Hilbert của tín hiệu $x(t) = \sin(600\pi t)$ và xác định tín hiệu giải tích tương ứng với x .

Tín hiệu giải tích của x được định nghĩa là: $x_a = x + j\hat{x}$, với $\hat{x}$ là biến đổi Hilbert của x .

Ta thiết kế bộ biến đổi Hilbert gần đúng bằng cách thiết kế một bộ lọc thông dải có dải thông chiếm gần như toàn bộ dải tần số khảo sát. Sau đó sử dụng bộ lọc này để lọc tín hiệu x , như vậy ở ngõ ra chính là biến đổi Hilbert của x , nhưng bị trễ đi một số lượng mẫu bằng một nửa bậc của bộ lọc. Do đó để tìm tín hiệu giải tích, ta phải làm trễ tín hiệu x đi một số lượng mẫu tương ứng.

```
b=firpm(20,[0.05 0.95],[1 1],'h'); % Bộ biến đổi Hilbert xấp xỉ dạng thông
dải
fs = 1000; % Tần số lấy mẫu
t = (0:1/fs:2)'; % Vector thời gian (chiều dài 2s)
x = sin(2*pi*300*t); % Tín hiệu sine tần số 300Hz
xh = filter(bb,1,x); % Biến đổi Hilbert của x
xd = [zeros(10,1); x(1:length(x)-10)]; % Làm trễ 10 mẫu
xa = xd + j*xh; % Tín hiệu giải tích
```

Phương pháp trên không khả thi đối với bộ lọc có bậc lẻ. Trong trường hợp này, có thể dùng hàm **hilbert** hoặc dùng hàm **resample** để làm trễ tín hiệu đi một số lượng mẫu không phải là số nguyên.

Các bộ vi phân

Đạo hàm một tín hiệu trong miền thời gian tương đương với nhân biến đổi Fourier của nó với hàm dốc thuần ảo. Như vậy, muốn tìm vi phân của một tín hiệu, ta đưa tín hiệu đó qua bộ lọc có đáp ứng tần số $H(\omega) = j\omega$. Các hàm **firls** và **firpm** cho phép thiết kế các bộ vi phân gần đúng (có hiện tượng delay), chỉ cần cung cấp thêm chuỗi ký tự 'd' hoặc 'differentiator', đồng thời chuyển đổi thang tần số bằng cách nhân với $\pi.f_s$. Ví dụ:

```
>> b = firpm(21,[0 1],[0 pi*fs],'d');
```

Với các bộ lọc FIR loại III thì dải vi phân phải kết thúc trước tần số Nyquist và vector biên độ cũng phải được hiệu chỉnh một cách tương ứng.

```
>> bb = firpm(20,[0 0.9],[0 0.9*pi*fs],'d');
```

Ở chế độ 'd', hàm **firpm** đánh giá trọng số của sai số bằng $1/\omega$ trong các dải tần có biên độ khác 0 để tối thiểu hoá sai số tương đối cực đại. Còn hàm **firls** đánh giá trọng số bằng $(1/\omega)^2$.

10.3.4. THIẾT KẾ BỘ LỌC FIR VỚI GIẢI THUẬT BÌNH PHƯƠNG CỰC TIỂU CÓ GIỚI HẠN (CLS – CONSTRAINED LEAST SQUARE)

Các bộ lọc FIR thiết kế theo giải thuật bình phương cực tiểu có giới hạn (CLS – Constrained Least Squares) cho phép người thiết kế không cần xác định các dải chuyển tiếp trong đáp ứng tần số một cách tường minh. Người thiết kế chỉ cần nhập vào các tần số cắt (trong trường hợp bộ lọc thông cao, thông dải, thông thấp hoặc chắn dải) hoặc các tần số giới hạn của các dải thông và dải chắn (trong trường hợp bộ lọc nhiều dải tần).

Đặc điểm quan trọng của phương pháp CLS là nó cho phép định nghĩa các giới hạn trên và dưới của độ gợn sóng tối đa cho phép trong đáp ứng tần số. Với các giới hạn này, giải thuật CLS sẽ tìm cách cực tiểu hoá sai số bình phương trên toàn bộ dải tần số, chứ không phải chỉ trên một số dải tần xác định. Giải thuật cũng cho phép giới hạn các đỉnh gây ra bởi hiện tượng Gibbs một cách tùy ý.

Phương pháp thiết kế bộ lọc FIR dùng giải thuật CLS được thực hiện bằng hai hàm **fircls** và **fircls1** của MATLAB. Hàm **fircls1** dùng để thiết kế các bộ lọc thông thấp và thông cao có pha tuyến tính còn hàm **fircls** dùng để thiết kế các bộ lọc FIR nhiều dải tần.

Thiết kế các bộ lọc CLS thông thấp và thông cao cơ bản

Đối với các bộ lọc thông thấp và thông cao, dùng hàm **fircls1**.

```
>> b = fircls1(n,wo,dp,ds,wp,ws,k,wt,'high')
```

Các thông số cơ bản cần cung cấp gồm bậc ủa bộ lọc N , tần số cắt w_0 , các độ lệch tối đa cho phép trong dải thông (dp) và dải chặn (ds) so với các giá trị lý tưởng là 1 và 0.

Nếu có thêm các thông số wp , ws , k , hàm **fircls1** sẽ đánh giá sai số bình phương trong dải thông với trọng số gấp k lần so với trong dải chặn, wp là giới hạn của dải thông và ws là giới hạn của dải chặn ($wp < w_0 < ws$ với bộ lọc thông thấp).

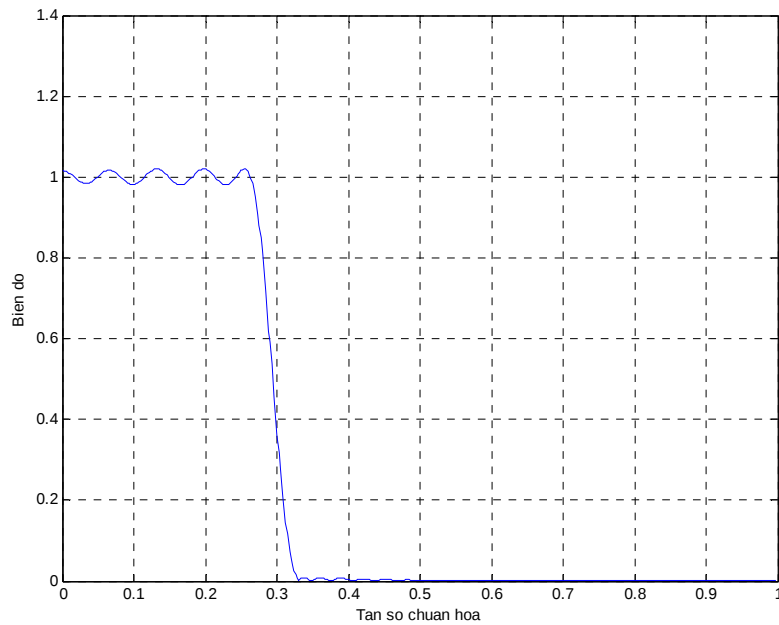
Nếu có thêm thông số wt thì bộ lọc được thiết kế phải thỏa mãn điều kiện $|e(wt)| \leq dp$ nếu wt nằm trong dải thông hoặc $|e(wt)| \leq ds$ nếu wt nằm trong dải chặn.

Trường hợp mặc định, hàm **fircls1** sẽ tạo ra bộ lọc thông thấp. Muốn thiết kế bộ lọc thông cao, thêm chuỗi 'high' vào cuối danh sách thông số.

Các giá trị tần số đều là tần số chuẩn hoá theo một nửa tần số lấy mẫu.

 **Ví dụ 10-8.** Thiết kế bộ lọc FIR thông thấp bậc 61, tần số cắt 0,3 (chuẩn hoá) với các độ lệch tối đa trong các dải thông và dải chặn lần lượt là 0,02 và 0,08. Giả sử thêm là quá trình tối thiểu hoá sai số trong dải thông được đánh giá với trọng số gấp 10 lần so với dải chặn, và khi tính trọng số thì dải thông kéo dài tới tần số 0,28 còn dải chặn bắt đầu từ 0,32.

```
n = 61; % Bậc bộ lọc
wo = 0.3; % Tần số cắt
dp = 0.02; % Độ lệch tối đa trong dải thông
ds = 0.008; % Độ lệch tối đa trong dải chặn
k = 10; % Tỷ lệ trọng số dải thông so với dải chặn
wp = 0.28; % Tần số giới hạn của dải thông
ws = 0.32; % Tần số giới hạn của dải chặn
b = fircls1(n,wo,dp,ds,wp,ws,k); % Gọi hàm fircls1
[hh,ff] = freqz(b,1,512,2); % Đáp ứng tần số của bộ lọc
plot(ff,abs(hh),'b-');grid % Vẽ đáp ứng biên độ
xlabel('Tan so chuan hoa');
ylabel('Bien do');
```



Hình 10.8.

Thiết kế các bộ lọc FIR nhiều dải tần

Trong trường hợp này, ta dùng hàm **fircls** với cú pháp như sau:

```
>> b = fircls(n,f,a,up,lo)
```

trong đó n là bậc của bộ lọc, f là một vector xác định các tần số ranh giới giữa các dải tần, a là vector xác định biên độ mong muốn trong các dải tần và up , lo lần lượt là các vector xác định các giới hạn trên và dưới của biên độ trong các dải tần nói trên. Các giá trị tần số đều là tần số chuẩn hoá.

 **Ví dụ 10-9.** Thiết kế bộ lọc FIR bậc 129 với các mô tả như sau:

Dải tần từ 0 -0,3: biên độ bằng 0, giới hạn trên 0,005, giới hạn dưới -0,005.

Dải tần từ 0,3 -0,5: biên độ bằng 0,5, giới hạn trên 0,51, giới hạn dưới 0,49.

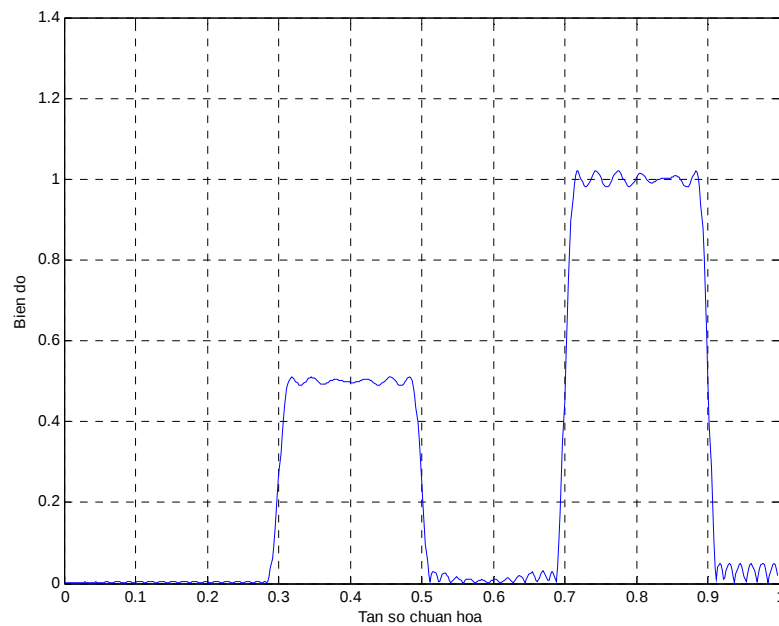
Dải tần từ 0,5 -0,7: biên độ bằng 0, giới hạn trên 0,03, giới hạn dưới -0,03.

Dải tần từ 0,7 -0,9: biên độ bằng 1, giới hạn trên 1,02, giới hạn dưới 0,98.

Dải tần từ 0,9 -1: biên độ bằng 0, giới hạn trên 0,05, giới hạn dưới -0,05.

```
n = 129; % Bậc bộ lọc
f = [0 0.3 0.5 0.7 0.9 1]; % Các tần số ngưỡng
a = [0 0.5 0 1 0]; % Biên độ mong muốn
up = [0.005 0.51 0.03 1.02 0.05]; % Giới hạn trên của biên độ
lo = [-0.005 0.49 -0.03 0.98 -0.05]; % Giới hạn dưới của biên độ
b = fircls(n,f,a,up,lo); % Các hệ số bộ lọc
[hh,ff] = freqz(b,1,512,2); % Vẽ đáp ứng tần số
plot(ff,abs(hh),'b-');grid
xlabel('Tần số chuẩn hóa');
```

```
ylabel('Biên độ');
```



Hình 10.9.

10.3.5. THIẾT KẾ BỘ LỌC FIR CÓ ĐÁP ỨNG TẦN SỐ TÙY CHỌN

Hàm **cfirpm** cho phép thiết kế các bộ lọc FIR có đáp ứng tần số phức bất kỳ do người thiết kế chọn. Điểm đặc biệt của hàm **cfirpm** so với các hàm khác là nó cho phép người thiết kế nhập vào đáp ứng tần số bằng cách cung cấp tên hàm biểu diễn đáp ứng tần số, hàm này sẽ trả về đáp ứng của bộ lọc tại các điểm tần số đã xác định. Đặc điểm này làm cho quá trình thiết kế mang tính linh hoạt và hiệu quả rất cao. Hàm **cfirpm** có thể dùng để thiết kế cả các bộ lọc FIR có pha phi tuyến, các bộ lọc FIR bất đối xứng (với các hệ số phức) và nhiều bộ lọc FIR đối xứng khác có đáp ứng tần số tùy chọn.

Giải thuật thiết kế là tối ưu hoá sai số cực đại (Chebychev) bằng cách dùng thuật giải Remez mở rộng để ước lượng ban đầu. Nếu quá trình tối ưu hoá không hội tụ thì chuyển sang giải thuật tăng – giảm (ascent – descent) để đạt được sự hội tụ về giải pháp tối ưu.

Cú pháp cơ bản của hàm này là:

```
>> b = cfirpm(n, f, {'fresp', p1, p2, ...}, w, sym)
```

trong đó n là bậc của bộ lọc, f là một vector gồm các tần số chuẩn hoá xếp theo thứ tự tăng dần từ -1 đến 1, dải tần từ $f(k)$ đến $f(k+1)$ với k lẻ sẽ là các dải thông hoặc dải chắn, dải tần từ $f(k+1)$ đến $f(k+2)$ với k chẵn là các vùng chuyển tiếp và sai số trong vùng này không được quan tâm đến khi tối thiểu hoá. 'fresp' là tên của hàm để tính đáp ứng tần số của bộ lọc tại các tần số cho bởi f , $p1, p2, \dots$ là danh sách các thông số cung cấp cho hàm **fresp**. w là một vector đánh giá trọng số của các dải tần trong quá trình tối thiểu hoá (mặc định là vector đơn vị), và **sym** là một chuỗi ký tự cho biết tính chất đối xứng của bộ lọc (bao gồm 'none', 'even', 'odd', 'real').

Các tên hàm có thể được dùng khi gọi hàm **cfirpm** là:

Bảng 10.7. Các giá trị có thể của 'fresp' khi gọi hàm **cfirpm**

Tên hàm	Mô tả
'lowpass'	Bộ lọc thông thấp
'bandpass'	Bộ lọc thông dải
'multiband'	Bộ lọc nhiều dải tần
'hilbfilt'	Bộ biến đổi Hilbert
'highpass'	Bộ lọc thông cao
'bandstop'	Bộ lọc chắn dải
'differentiator'	Bộ vi phân
'invsinc'	Hàm sinc đảo

Thiết kế bộ lọc nhiều dải tần

Trong khi gọi hàm **cfirpm** ta cung cấp tên hàm 'multiband' và các thông số đi kèm.

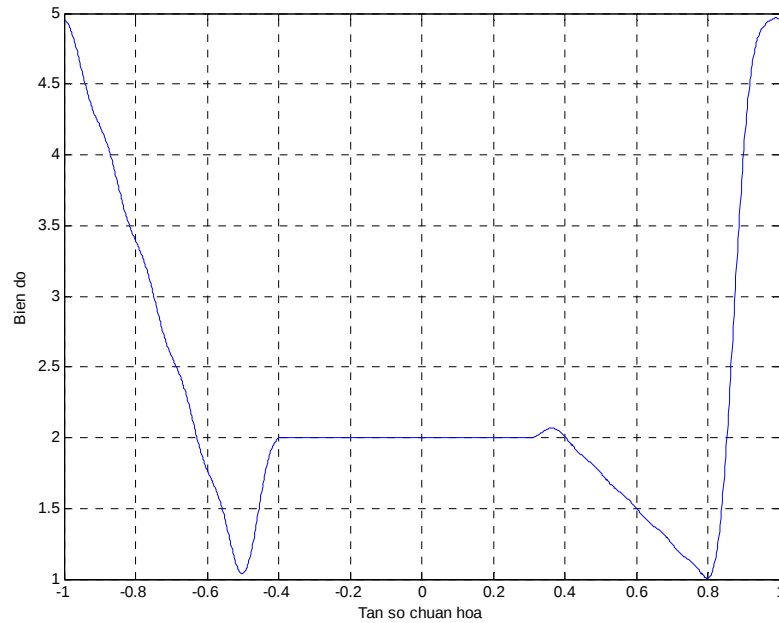
 **Ví dụ 10-10.** Thiết kế bộ lọc FIR bậc 38 với các mô tả như sau:

Dải tần từ $-1 \div -0,5$: biên độ giảm từ 5 xuống 1 ([5 1]).

Dải tần từ $-0,4 \div 0,3$: biên độ bằng 2 ([2 2]), trọng số bằng 10.

Dải tần từ $0,4 \div 0,8$: biên độ giảm từ 2 xuống 1 ([2 1]), trọng số bằng 5.

```
b = cfirpm(38, [-1 -0.5 -0.4 0.3 0.4 0.8], ...
{'multiband', [5 1 2 2 2 1]}, [1 10 5]);
[hh,ff] = freqz(b,1,512,2,'whole');
plot(ff,abs(hh),'b-');grid
xlabel('Tan so chuan hoa');
ylabel('Bien do');
```



Hình 10.10.

10.4. THỰC HIỆN BỘ LỌC

Sau quá trình thiết kế bộ lọc, ta nhận được hai vector a và b chứa các hệ số của bộ lọc. Dựa trên hai vector này, ta có thể thực hiện bộ lọc bằng cách dùng một trong hai hàm sau:

- Hàm **dfilt** cho phép định nghĩa cấu trúc của bộ lọc và khởi tạo một đối tượng bộ lọc số trong MATLAB.
- Hàm **filter** với các thông số đầu vào là các vector a và b sẽ thực hiện bộ lọc với cấu trúc chuyển vị trực tiếp dạng II và sử dụng nó để lọc tín hiệu vào. Nếu thông số đầu vào là đối tượng tạo bởi hàm **dfilt** thì hàm **filter** sẽ thực hiện cấu trúc bộ lọc định nghĩa bởi **dfilt** và dùng nó để lọc tín hiệu vào.

Việc lựa chọn cấu trúc bộ lọc để thực hiện nó phụ thuộc vào các nhiệm vụ cụ thể mà bộ lọc đó phải thực hiện. Ví dụ nếu bộ lọc hoạt động ở tốc độ cao, ta nên dùng bộ lọc FIR không đệ quy thay vì dùng bộ lọc IIR đệ quy; các bộ lọc IIR cấu trúc trực tiếp thường được thực hiện dưới cấu trúc các khâu bậc 2 (SOS) vì nó rất nhạy với nhiễu do làm tròn số khi tính toán.

Cú pháp của hàm **filter** đã được đề cập ở chương 9. Do đó ở đây chỉ trình bày cú pháp của hàm **dfilt**. Biết trước các hệ số của bộ lọc, hàm **dfilt** sẽ tạo ra một đối tượng bộ lọc số với cấu trúc do người sử dụng xác định. Với đối tượng này, ta có thể thực hiện các tác vụ khác nhau như tính đáp ứng tần số, độ trễ nhóm, ... hoặc cũng có thể dùng đối tượng này làm nhập liệu cho hàm **filter** để thực hiện lọc các tín hiệu.

Để khởi tạo đối tượng bộ lọc số, ta dùng cú pháp:

```
>> Hd = dfilt.STRUCTURE(input1,input2,...)
```

trong đó **STRUCTURE** là tên cấu trúc bộ lọc mà ta muốn thực hiện. Ví dụ, tạo đối tượng bộ lọc có cấu trúc chuyển vị trực tiếp dạng 2:

```
>> Hd = dfilt.df2t(b,a);
```

Độc giả có thể tìm danh sách các cấu trúc có thể dùng với hàm **dfilt** bằng cách gõ lệnh **help dfilt** ở cửa sổ lệnh của MATLAB. Cũng bằng cách này, các bạn có thể tìm thấy danh sách các tác vụ có thể thực hiện trên đối tượng bộ lọc số tạo bởi hàm **dfilt**.

✎ Bài tập 10-1.

Thiết kế một bộ lọc thông thấp elliptic tương tự có các đặc tính sau:

- gợn sóng dải thông tối đa 1dB
- tần số cắt của dải thông bằng 10rad/s
- suy hao dải chặn tối thiểu 40dB ở tần số lớn hơn 15rad/s.

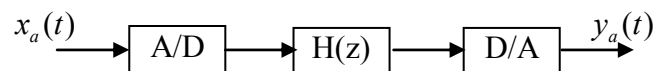
Vẽ đáp ứng biên độ, đáp ứng biên độ theo dB, đáp ứng pha và đáp ứng xung của bộ lọc.

✎ ✎ Bài tập 10-2.

Tín hiệu $x(t)$ có hai thành phần tần số: 100Hz và 130Hz. Cần phải nén thành phần tần số 130Hz tối thiểu là 50dB đồng thời cho thành phần tần số 100Hz qua với suy hao tối đa 2dB. Thiết kế bộ lọc Chebychev loại I có bậc tối thiểu để thực hiện việc này. Vẽ đáp ứng biên độ và kiểm chứng lại thiết kế.

✎ ✎ Bài tập 10-3.

Thiết kế bộ lọc số thông thấp sử dụng trong sơ đồ sau:



thoả mãn các yêu cầu:

- tốc độ lấy mẫu: 8000 mẫu/s
- tần số ngưỡng của dải thông là 1500Hz, độ gợn sóng 3dB
- tần số ngưỡng của dải chặn là 2000Hz, suy hao 40dB
- gợn sóng cân bằng trng dải thông nhưng đơn điệu trong dải chặn
- sử dụng phương pháp bất biến xung

Xác định hàm truyền đạt dạng hữu tỷ và dạng ghép các khâu bậc 2. Vẽ đáp ứng biên độ (theo dB) và đáp ứng xung của bộ lọc.

✎ Bài tập 10-4.

Xét bộ lọc thông thấp elliptic đã thiết kế ở bài tập 10-1. Dùng biến đổi song tuyến tính để rời rạc hoá bộ lọc. Vẽ đáp ứng biên độ và đáp ứng xung.

Sử dụng hàm **ellip** để thiết kế lại bộ lọc này và so sánh với kết quả ở trên.

✎ Bài tập 10-5.

Thiết kế bộ lọc thông dải dùng hàm **cheby2** với các yêu cầu như sau:

- giới hạn của dải chặn dưới: $0,3\pi$.
- giới hạn của dải chặn trên: $0,6\pi$
- suy hao các dải chặn: 50dB
- các giới hạn trên và dưới của dải thông: $0,4\pi$ và $0,5\pi$.
- gợn sóng dải thông: 0,5dB

✎ **Bài tập 10-6.**

Thiết kế lại bộ lọc thông dải trong bài tập 10-5 dùng hàm **yulewalk**.

✎ **Bài tập 10-7.**

Với đáp ứng xung của bộ lọc được thiết kế ở bài tập 10-5, hãy dùng phương pháp Prony để thiết kế lại bộ lọc này. So sánh kết quả với bài 10-5.

✎ **Bài tập 10-8.**

Thiết kế bộ lọc thông thấp FIR có tần số cắt $\omega = 0,3\pi$ sử dụng phương pháp cửa sổ chữ nhật. Lần lượt cho bậc của bộ lọc nhận các giá trị $N = 11, 41, 81$ và 121 . Vẽ đáp ứng xung và đáp ứng biên độ của bộ lọc. Nhận xét kết quả.

✎ **Bài tập 10-9.**

Làm lại bài tập trên dùng cửa sổ Hamming. So sánh cửa sổ Hamming với cửa sổ chữ nhật.

✎ **Bài tập 10-10.**

Thiết kế bộ lọc FIR số, thông thấp, pha tuyến tính sử dụng phương pháp cửa sổ Kaiser với các yêu cầu thiết kế: tần số lấy mẫu 10kHz , tần số giới hạn dải thông $1,5\text{kHz}$, tần số giới hạn dải chặn 2kHz , suy hao dải thông $0,1\text{dB}$ suy hao dải chặn 80dB . Tính bậc của bộ lọc (tất cả thực hiện trên MATLAB).

✎ **Bài tập 10-11.**

Thiết kế bộ lọc FIR chắn dải dùng cửa sổ Hanning với các yêu cầu như sau:

- giới hạn của dải thông dưới: $0,3\pi$.
- giới hạn của dải thông trên: $0,7\pi$
- gợn sóng dải thông: $0,5\text{dB}$
- các giới hạn trên và dưới của dải thông: $0,4\pi$ và $0,6\pi$.
- suy hao các dải chặn: 40dB

Vẽ đáp ứng biên độ (theo dB), đáp ứng pha và đáp ứng xung của bộ lọc.

✎ **Bài tập 10-12.**

Thiết kế bộ lọc FIR dùng cửa sổ Blackman có đáp ứng tần số thỏa mãn yêu cầu sau:

Dải tần 1: $0 \leq \omega \leq 0,3\pi$, độ lợi lý tưởng bằng 1, độ gợn sóng tối đa $0,001$

Dải tần 2: $0,4\pi \leq \omega \leq 0,7\pi$, độ lợi lý tưởng bằng $0,5$, độ gợn sóng tối đa $0,005$

Dải tần 2: $0,8\pi \leq \omega \leq \pi$, độ lợi lý tưởng bằng 0 , độ gợn sóng tối đa $0,001$

Vẽ đáp ứng biên độ và đáp ứng pha.

✎ **Bài tập 10-13.**

Làm lại bài 10-12 dùng các giải thuật bình phương cực tiểu (hàm **firls**) và giải thuật Parks-McClellan (hàm **firpm**). So sánh hai giải thuật này.

✎ **Bài tập 10-14.**

Thiết kế lại bộ lọc FIR trong bài 10-12 dùng giải thuật CLS. Nhận xét.

✎ **Bài tập 10-15.**

Thiết kế bộ lọc FIR có đáp ứng tần số được mô tả như sau:

- Dải tần từ $-1 \div -0,8$: biên độ giảm từ 5 xuống 2, trọng số bằng 1.
- Dải tần từ $-0,7 \div 0,5$: biên độ bằng 2, trọng số bằng 5.
- Dải tần từ $-0,4 \div -0,1$: biên độ giảm từ 2 xuống 1, trọng số bằng 1.
- Dải tần từ $0,1 \div 0,4$: biên độ tăng từ 1 lên 2, trọng số bằng 1.
- Dải tần từ $0,5 \div 0,7$: biên độ bằng 2, trọng số bằng 5.
- Dải tần từ $0,8 \div 1$: biên độ tăng từ 2 lên 5, trọng số bằng 1.

Vẽ đáp ứng tần số của bộ lọc này.

✎ **Bài tập 10-16.**

Thiết kế bộ vi phân gần đúng bằng cách dùng bộ lọc FIR bậc 30. Sau đó kiểm chứng lại thiết kế bằng cách đưa tín hiệu răng cưa vào và vẽ tín hiệu ra. Nhận xét.

✎ **Bài tập 10-17.**

Dùng hàm **dfilt** để tạo một đối tượng bộ lọc số với các yêu cầu như ở bài tập 10-3. Hãy cố gắng sử dụng tất cả các cấu trúc bộ lọc có thể thực hiện. Sau đó thực hiện một số tác vụ trên đối tượng này: vẽ đáp ứng biên độ, đáp ứng pha, đáp ứng xung, độ trễ pha, độ trễ nhóm,...

Danh sách các hàm được giới thiệu trong chương 10

Các hàm thiết kế bộ lọc IIR

besselap	Tạo bộ lọc thông thấp analog loại Bessel
besself	Thiết kế bộ lọc Bessel hoàn chỉnh
bilinear	Thực hiện biến đổi song tuyến tính
buttap	Tạo bộ lọc thông thấp analog Butterworth
butter	Thiết kế bộ lọc Butterworth hoàn chỉnh
buttord	Ước lượng bậc của bộ lọc Butterworth
cheb1ap	Tạo bộ lọc thông thấp analog Chebychev loại I
cheb2ap	Tạo bộ lọc thông thấp analog Chebychev loại II
cheb1ord	Ước lượng bậc của bộ lọc Chebychev loại I
cheb2ord	Ước lượng bậc của bộ lọc Chebychev loại II
cheby1	Thiết kế bộ lọc Chebychev loại I hoàn chỉnh
cheby2	Thiết kế bộ lọc Chebychev loại II hoàn chỉnh
ellip	Thiết kế bộ lọc elliptic hoàn chỉnh
ellipap	Tạo bộ lọc thông thấp analog elliptic
ellipord	Ước lượng bậc của bộ lọc elliptic
impinvar	Rời rạc hoá bộ lọc bằng phương pháp bất biến xung
invfreqs	Thiết kế bộ lọc analog trong miền tần số dựa vào đáp ứng tần số cho trước
invfreqz	Thiết kế bộ lọc số trong miền tần số dựa vào đáp ứng tần số cho trước
lp2bp	Thực hiện biến đổi tần số: thông thấp – thông dải
lp2bs	Thực hiện biến đổi tần số: thông thấp – chắn dải
lp2hp	Thực hiện biến đổi tần số: thông thấp – thông cao
lp2lp	Thực hiện biến đổi tần số: thông thấp – thông thấp
lpc	Thiết kế bộ lọc số bằng phương pháp mã hoá dự đoán tuyến tính
maxflat	Thiết kế bộ lọc thông thấp Butterworth tổng quát
prony	Thiết kế bộ lọc số bằng phương pháp Prony
stmcb	Thiết kế bộ lọc số bằng phương pháp lặp Steiglitz-McBride
yulewalk	Thiết kế bộ lọc số trực tiếp trong miền thời gian

Các hàm thiết kế bộ lọc FIR

bartlett	Hàm cửa sổ Bartlett
barthannwin	Hàm cửa sổ Bartlett - Hanning
blackman	Hàm cửa sổ Blackman
blackmanharris	Hàm cửa sổ Blackman - Harris

bohman	Hàm cửa sổ Bohman
cfirpm	Thiết kế bộ lọc FIR dựa vào đáp ứng tần số cho trước
chebwin	Hàm cửa sổ Chebychev
fir1	Thiết kế bộ lọc FIR bằng phương pháp cửa sổ
fir2	Thiết kế bộ lọc FIR bằng phương pháp cửa sổ với đáp ứng tùy chọn
fircls	Thiết kế bộ lọc FIR bằng phương pháp giới hạn bình phương cực tiểu, đáp ứng tần số có nhiều dải tần
fircls1	Thiết kế bộ lọc FIR thông thấp và thông cao bằng phương pháp giới hạn bình phương cực tiểu
firls	Thiết kế bộ lọc FIR nhiều dải tần theo phương pháp sai số bình phương cực tiểu
firpm	Thiết kế bộ lọc FIR nhiều dải tần theo phương pháp Parks - McClellan
firpmord	Ước lượng bậc của bộ lọc Parks - McClellan
firrcos	Đáp ứng của bộ lọc thông thấp với vùng chuyển tiếp có dạng sine
flattopwin	Hàm cửa sổ flattop
gausswin	Hàm cửa sổ Gauss
hamming	Hàm cửa sổ Hamming
hann	Hàm cửa sổ Hanning
hilbert	Thực hiện biến đổi Hilbert
kaiser	Hàm cửa sổ Kaiser
kaiserord	Ước lượng bậc của bộ lọc FIR theo phương pháp Kaiser
nutallwin	Hàm cửa sổ nutall
parzenwin	Hàm cửa sổ Parzen
rectwin	Hàm cửa sổ chữ nhật
resample	Hàm lấy mẫu lại
triang	Hàm cửa sổ tam giác
tukeywin	Hàm cửa sổ Tukey

Chương 11

CƠ BẢN VỀ XỬ LÝ ẢNH SỐ

Vấn đề quan trọng khi mô phỏng một hệ thống thông tin là phân tích các đáp ứng của nó trước các yếu tố gây nhiễu tồn tại trong thế giới thực, minh họa bằng các công cụ đồ họa và đánh giá xem chất lượng của nó có đáp ứng các tiêu chuẩn đã được đặt ra đối với hệ thống hay không. Vấn đề này có thể giải quyết tốt bởi các công cụ đánh giá chất lượng kênh truyền do MATLAB cung cấp.

11.1. BIỂU DIỄN ẢNH VÀ XUẤT NHẬP ẢNH

MATLAB là một công cụ tính toán được xây dựng trên cơ sở các phép xử lý ma trận. Đây lại là đối tượng rất thích hợp cho việc biểu diễn các hình ảnh, trong đó, mỗi phần tử của ma trận biểu diễn dữ liệu màu hoặc mức xám của 1 điểm ảnh. Trong MATLAB, mỗi hình ảnh thường được biểu diễn bằng một ma trận hai chiều. Mỗi một phần tử của ma trận tương ứng với 1 pixel (một phần tử ảnh – picture element – biểu diễn bằng 1 điểm trên màn hình). Điểm ảnh ở góc trên bên trái là ứng với hàng 1, cột 1. Một số hình ảnh được biểu diễn bằng một ma trận ba chiều, ví dụ ảnh RGB, trong đó chiều thứ ba có kích thước bằng 3, nghĩa là ma trận này có thể chia thành 3 ma trận hai chiều, ma trận thứ nhất ứng với độ sáng màu đỏ (R – Red) của điểm ảnh, ma trận thứ hai ứng với độ sáng màu xanh dương (B – Blue) và ma trận thứ ba ứng với độ sáng màu lục (G – Green).

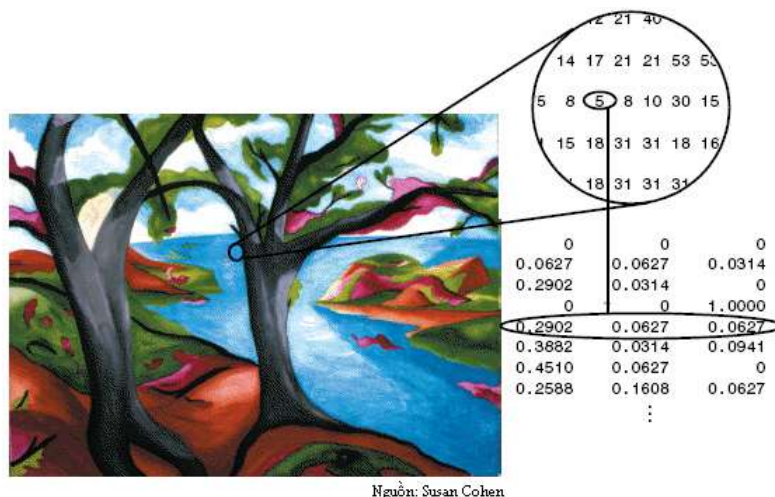
MATLAB lưu giữ các hình ảnh dưới một trong các kiểu dữ liệu sau: `logical`, `uint8`, `uint16`, `double`. Người sử dụng có thể sử dụng các phép toán và các thao tác cơ bản trên ma trận, chẳng hạn như truy xuất các phần tử, thay đổi kích thước, sắp xếp, cắt bỏ một phần ma trận ... để tác động lên các dữ liệu hình ảnh với điều kiện các thao tác này chấp nhận các kiểu dữ liệu ảnh nói trên. Riêng với các kiểu dữ liệu `uint8` và `uint16`, ta không thể sử dụng các phép toán số học như cộng, trừ, nhân, chia. Trong trường hợp này, MATLAB cung cấp thêm các hàm thực hiện các phép toán này mà chúng tôi sẽ đề cập ở cuối phần này.

11.1.1. CÁC KIỂU HÌNH ẢNH TRONG MATLAB

Image Processing Toolbox của MATLAB hỗ trợ bốn kiểu biểu diễn hình ảnh cơ bản, gồm: ảnh chỉ số (indexed images), ảnh độ sáng (intensity images), ảnh nhị phân (binary images), ảnh RGB (RGB images).

Ảnh chỉ số

Với cách biểu diễn ảnh này, mỗi ảnh sẽ được biểu diễn bởi hai ma trận, một ma trận dữ liệu ảnh X và một ma trận màu (còn gọi là bản đồ màu) `map`. Ma trận dữ liệu có thể thuộc kiểu `uint8`, `uint16`, hoặc `double`. Ma trận màu là một ma trận kích thước $m \times 3$ gồm các phần tử kiểu `double` có giá trị trong khoảng $[0,1]$. Mỗi hàng của ma trận xác định các thành phần `red`, `green`, `blue` của một màu trong tổng số m màu được sử dụng trong ảnh. Giá trị của một phần tử trong ma trận dữ liệu cho biết màu của điểm ảnh đó là màu nằm ở hàng nào trong ma trận màu. Nếu ma trận dữ liệu thuộc kiểu `double`, giá trị 1 sẽ tương ứng với hàng thứ 1 trong bảng màu, giá trị thứ hai tương ứng với màu ở hàng thứ hai, ... Nếu ma trận dữ liệu thuộc kiểu `uint8` hoặc `uint16`, giá trị 0 ứng với hàng 1, giá trị 1 ứng với hàng 2, ... Riêng với kiểu `uint16`, MATLAB không hỗ trợ đủ các phép toán so với kiểu `uint8` nên khi cần xử lý ta cần chuyển sang kiểu dữ liệu `uint8` hoặc `double` bằng các hàm **`imapprox`** hoặc **`im2double`**. Hình 11.1 minh họa cách biểu diễn ảnh theo chỉ số.

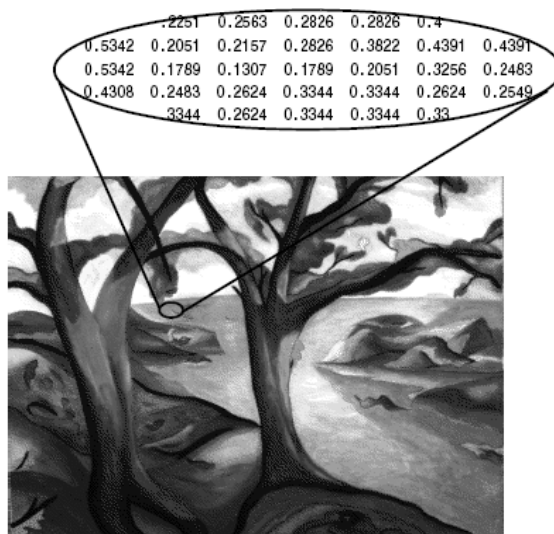


Nguồn: Susan Cohen

Hình 11.1. Biểu diễn ảnh bằng phương pháp chỉ số

Ảnh biểu diễn theo độ sáng

Mỗi ảnh được biểu diễn bởi một ma trận hai chiều, trong đó giá trị của mỗi phần tử cho biết độ sáng (hay mức xám) của điểm ảnh đó. Ma trận này có thể thuộc một trong các kiểu `uint8`, `uint16`, hoặc `double`. Trong đó, giá trị nhỏ nhất (0) ứng với màu đen còn giá trị lớn nhất (255 hoặc 65535 hoặc 1 tùy kiểu dữ liệu là `uint8`, `uint16`, hay `double`) ứng với màu trắng. Như vậy, ảnh biểu diễn theo kiểu này còn gọi là ảnh “trắng đen” hoặc ảnh gray scale (xem hình 11.2).



Hình 11.2. Biểu diễn ảnh theo độ sáng

Ảnh nhị phân

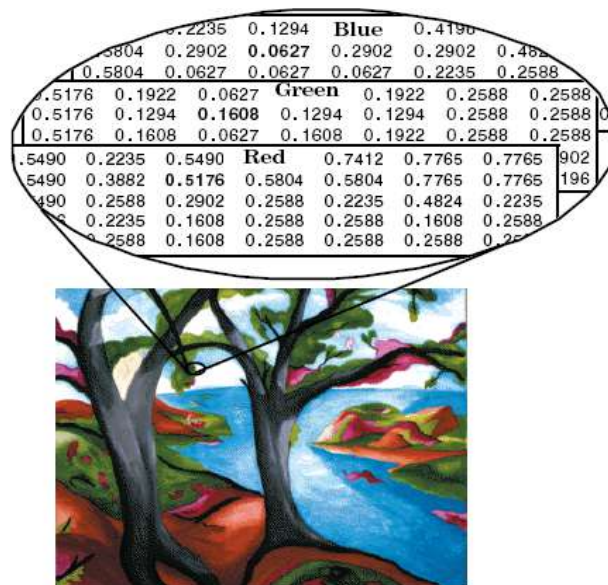
Ảnh nhị phân cũng được biểu diễn bởi ma trận hai chiều nhưng thuộc kiểu `logical`, có nghĩa là mỗi điểm ảnh chỉ có thể nhận một trong hai giá trị 0 (đen) hoặc 1 (trắng) (hình 11.3).



Hình 11.3. Ảnh nhị phân

Ảnh RGB

Ảnh RGB còn gọi là ảnh “truecolor” do tính trung thực của nó. Ảnh này được biểu diễn bởi một ma trận ba chiều kích thước $m \times n \times 3$, với $m \times n$ là kích thước ảnh theo pixels. Ma trận này định nghĩa các thành phần màu red, green, blue cho mỗi điểm ảnh, các phần tử của nó có thể thuộc kiểu `uint8`, `uint16`, hoặc `double`. Ví dụ, điểm ảnh ở vị trí (10,5) sẽ có ba thành phần màu được xác định bởi các giá trị (10,5,1), (10,5,2) và (10,5,3). Các file ảnh hiện nay thường sử dụng 8 bit cho một thành phần màu, nghĩa là 24 bit cho mỗi điểm ảnh (khoảng 16 triệu màu). Cách biểu diễn ảnh RGB được minh họa ở hình 11.4.



Hình 11.4. Biểu diễn ảnh RGB

Dãy ảnh đa khung (multiframe)

Trong một số ứng dụng, người sử dụng có thể có nhu cầu lưu một chuỗi các ảnh ghi nhận tại những thời điểm liên tiếp nhau để tiện cho việc hiển thị, ví dụ ảnh từ các đoạn video hoặc cắt lát cắt từ máy chụp ảnh cộng hưởng từ, ... Để phục vụ cho các ứng dụng này, MATLAB cho phép ta lưu nhiều ảnh thành một dãy, mỗi ảnh gọi là một frame, bằng hàm `cat`.

```
>> muti = cat(4, A1, A2, A3, ... ) % A1, A2, A3, ... là các ảnh cần lưu
```

Ta cũng có thể truy xuất đến một frame trong dãy ảnh đa frame này theo cách như sau:

```
>> frm3 = multi(:, :, :, 3) % truy xuất frame thứ 3
```

Chuyển đổi giữa các kiểu ảnh

Có một số thao tác xử lý ảnh trong MATLAB chỉ thực hiện được trên kiểu ảnh này mà không thực hiện được trên một kiểu ảnh khác, ví dụ muốn lọc ảnh màu, thì ảnh cần lọc phải ở dạng RGB. Vì vậy, trong MATLAB có một số hàm cho phép người sử dụng chuyển đổi qua lại giữa các kiểu ảnh cho tiện xử lý. **Bảng 11.1** dưới đây liệt kê danh sách các hàm như vậy.

Bảng 11.1. Các hàm chuyển đổi giữa các kiểu ảnh

Tên hàm	Cú pháp	Mô tả
dither	dither(RGB,map) dither(I)	Tạo ảnh nhị phân từ ảnh trắng đen I hoặc tạo ảnh indexed từ ảnh RGB và ma trận màu map bằng phương pháp dithering
gray2ind	[X,MAP] = gray2ind(I,N) [X,MAP] = gray2ind(BW,N)	Chuyển đổi ảnh nhị phân BW hoặc ảnh intensity thành ảnh index thông qua ma trận màu tuyến tính gray(N)
grayslice	X=grayslice(I,N) X=grayslice(I,V)	Chuyển đổi ảnh trắng đen thành ảnh index bằng cách lấy ngưỡng. Các giá trị ngưỡng được xác định bởi vector V hoặc bởi N (khi đó các giá trị ngưỡng là 1/n, 2/n, ... (n-1)/n)
im2bw	bw = im2bw(i,level) bw = im2bw(x,map,level) bw = im2bw(rgb,level)	Chuyển đổi các loại ảnh trắng đen, ảnh index, ảnh RGB thành nhị phân bằng cách lấy ngưỡng bởi level
ind2gray	i = ind2gray(x,map)	Chuyển đổi ảnh index I với ma trận màu map thành ảnh trắng đen
ind2rgb	rgb = ind2rgb(x,map)	Chuyển đổi ảnh index I với ma trận màu map thành ảnh RGB
mat2gray	i = mat2gray(a,[amin amax])	Tạo ảnh trắng đen từ ma trận A, giá trị Amin tương ứng với mức 0 (đen), giá trị Amax ứng với mức 1 (trắng), mặc định: Amin, Amax sẽ là giá trị lớn nhất và nhỏ nhất của A
rgb2gray	i = rgb2gray(rgb)	Chuyển đổi ảnh RGB thành ảnh trắng đen
rgb2ind	[x,map] = rgb2ind(rgb,n) x = rgb2ind(rgb,map) [x,map] = rgb2ind(rgb,tol)	Chuyển ảnh RGB rgb thành ảnh index [x,map]: TH1: $N \leq 65535$, số màu tối đa là N TH2: mỗi pixel được xấp xỉ đến màu gần giống nhất trong bản màu map TH3: $\text{tol} \in [0,1]$, số màu tối đa: $(\text{floor}(1/\text{tol})+1)^3$

11.1.2. ĐỌC VÀ GHI CÁC DỮ LIỆU ẢNH

▪ Hàm **imread** đọc các file ảnh với bất kỳ các định dạng ảnh đã biết hiện nay và lưu lại dưới dạng một ma trận biểu diễn ảnh trong MATLAB. Hầu hết các định dạng ảnh hiện nay dùng 8 bit cho mỗi pixel (ứng với một thành phần màu), do đó sau khi đọc MATLAB sẽ lưu lại dưới dạng ma trận thuộc kiểu `uint8`. Với các định dạng ảnh 16 bit như PNG và TIFF, MATLAB sẽ dùng kiểu `uint16`. Với ảnh index, ma trận màu sẽ được lưu với kiểu `double`. Cú pháp của hàm **imread**:

```
>> A = imread(filename,fmt)
```

```
>> [X,map] = imread(filename,fmt)
```

Trong đó: `filename` là chuỗi xác định tên file cần đọc cùng với đường dẫn (nếu file này không nằm trong thư mục hiện hành).

`fmt` là chuỗi cho biết định dạng của ảnh, thí dụ `'bmp'`, `'gif'`, `'jpg'`, ...

Ngoài ra, hàm **imread** còn có thêm một số thông số khác tùy vào từng định dạng ảnh cụ thể. Gõ lệnh `help` ở cửa sổ lệnh của MATLAB để biết về các thông số này.

MATLAB hỗ trợ cả một số định dạng ảnh có thể chứa nhiều ảnh như HDF, TIFF hay GIF. Trong trường hợp mặc định, hàm `imread` chỉ đọc ảnh đầu tiên trong các ảnh này. Tuy nhiên, ta có thể cung cấp thêm một thông số là chỉ số của ảnh cần đọc trong dãy như trong ví dụ sau đây.

 **Ví dụ 11-1.** Đọc một chuỗi 27 ảnh liên tiếp trong một file TIFF và lưu vào một dãy 4 chiều:

```
mri = uint8(zeros(128,128,1,27)); % khởi tạo một dãy 4 chiều
for frame=1:27
    [mri(:,:,,frame),map] = imread('mri.tif',frame);
end
```

▪ Hàm **imwrite** cho phép lưu một ảnh biểu diễn bằng một ma trận trong MATLAB thành một file ảnh dưới một trong các định dạng ảnh đã biết. Cú pháp cơ bản của hàm này như sau:

```
>> imwrite (A, filename, fmt)
>> imwrite (X, map, filename, fmt)
>> imwrite (... , param1, val1, param2, val2, ...)
```

Có thể bỏ qua thông số `fmt` nếu trong chuỗi `filename` có cả phần mở rộng (sau dấu chấm). Tùy thuộc vào định dạng ảnh cần lưu, ta cung cấp thêm tên các thông số `param1`, `param2`, ... cùng với giá trị tương ứng của chúng. Ví dụ, lệnh sau đây thực hiện ghi vào file `mypicture.jpg` với chất lượng nén là 100:

```
>> imwrite(A, 'mypicture.jpg', 'Quality', 100)
```

Để kiểm chứng xem các thông số mà chúng ta đã xác định trong hàm **imwrite** có được thực hiện đúng hay không, hoặc để xem các thông số của một file ảnh nào đó, ta có thể dùng hàm **imfinfo**:

```
>> info = imfinfo(filename,fmt)
```

Ví dụ:

```
>> info = imfinfo('test.tif');
>> info.BitDepth
ans =
1
```

Các thông tin mà hàm này cung cấp được liệt kê trong **bảng 11.2**.

11.1.3.CHUYỂN ĐỔI GIỮA CÁC KIỂU DỮ LIỆU

Chúng ta có thể chuyển đổi giữa các kiểu dữ liệu `uint8`, `uint16` và `double` nhờ sử dụng các hàm chuyển kiểu của MATLAB mà ta đã biết như hàm **double**, **uint8**, **uint16**, nhưng việc chuyển kiểu như vậy có thể không phù hợp với cách biểu diễn ảnh trong MATLAB, do đó sau khi chuyển kiểu phải dịch thang độ hoặc offset thì mới trở thành dữ liệu ảnh hợp lệ. Nhằm

mục đích giúp đỡ người sử dụng trong việc chuyển kiểu nói trên, MATLAB cung cấp sẵn các hàm thực hiện chuyển kiểu cho các ma trận biểu diễn ảnh, bao gồm: **im2double**, **im2uint8**, và **im2uint16**. Cú pháp của các hàm này rất đơn giản, chỉ cần nhập vào ma trận ảnh cần chuyển kiểu, riêng với ảnh indexed cần thêm vào chuỗi 'indexed'.

Tuy nhiên, khi thực hiện chuyển đổi giữa các kiểu dữ liệu cần lưu ý một số vấn đề như sau:

- Khi chuyển từ một kiểu dữ liệu dùng nhiều bit sang một kiểu dữ liệu dùng ít bit hơn, ví dụ từ uint16 sang uint8, một số thông tin chi tiết về bức ảnh ban đầu sẽ bị mất đi. Nói chung sự mất mát này vẫn chấp nhận được vì 256 vẫn lớn hơn số mức xám tối đa mà mắt người có thể phân biệt được. Tuy nhiên nếu cần những ảnh có chất lượng cao thì cũng nên lưu ý sự mất mát này.
- Không phải lúc nào cũng có thể chuyển đổi kiểu dữ liệu đối với kiểu ảnh indexed, vì các giá trị của ma trận ảnh xác định một địa chỉ trong bản đồ màu chứ không phải là giá trị màu, do đó không thể lượng tử hoá được. Ví dụ, ta không thể chuyển một ảnh indexed kiểu double với ma trận màu có 300 màu thành kiểu uint8 vì kiểu dữ liệu này chỉ có 256 giá trị. Muốn chuyển được, đầu tiên ta phải dùng hàm **imapprox** để giảm số màu cần để biểu diễn ảnh xuống (bằng cách ghép các màu gần giống nhau lại thành một màu) rồi mới chuyển.

Bảng 11.2. Các thông tin có được khi gọi hàm **imfinfo**

Tên thuộc tính	Mô tả
Filename	Chuỗi chứa tên file
FileModDate	Chuỗi cho biết ngày hiệu chỉnh file gần đây nhất
FileSize	Một số nguyên chỉ kích thước file (đơn vị byte)
Format	Chuỗi cho biết định dạng của ảnh
FormatVersion	Chuỗi hoặc số cho biết tên phiên bản của định dạng
Width	Số nguyên chỉ chiều rộng ảnh (pixels)
Height	Số nguyên chỉ chiều cao ảnh (pixels)
BitDepth	Số nguyên cho biết số bits trên một pixel
ColorType	Chuỗi cho biết kiểu ảnh: 'truecolor', 'grayscale' hoặc 'indexed'

11.1.4. CÁC PHÉP TOÁN SỐ HỌC CƠ BẢN ĐỐI VỚI DỮ LIỆU ẢNH

Các phép toán số học cơ bản trên các dữ liệu ảnh bao gồm các phép cộng, trừ, nhân và chia. Đây là những thao tác xử lý ảnh cơ bản trước khi thực hiện các biến đổi phức tạp khác. Người sử dụng có thể dùng các phép toán số học mà MATLAB cung cấp để tác động lên dữ liệu ảnh. Tuy nhiên, do MATLAB chỉ hỗ trợ các phép toán này trên kiểu double nên cần thực hiện chuyển đổi kiểu trước khi thực hiện. Để làm giảm bớt thao tác này, trong MATLAB Image Processing Toolbox có cung cấp các hàm thực hiện các phép toán số học trên ảnh mà có thể chấp nhận bất kỳ kiểu dữ liệu ảnh nào và trả về kết quả thuộc cùng kiểu với các toán hạng. Các hàm này cũng xử lý các dữ liệu tràn một cách tự động. Dưới đây là danh sách các hàm thực hiện các phép toán số học cơ bản trên ảnh cùng với cú pháp tương ứng:

Bảng 11.3. Các phép toán số học trên ảnh

Tên hàm	Cú pháp	Mô tả
imabsdiff	<code>z = imabsdiff(x, y)</code>	Trừ mỗi phần tử của Y từ phần tử tương ứng của X, sau đó trả về trị tuyệt đối của hiệu
imadd	<code>z = imadd(x, y, out_class)</code>	Cộng hai ảnh hoặc cộng một ảnh với một hằng số, <code>output_class</code> là chuỗi xác định kiểu dữ liệu của tổng
imcomplement	<code>im2 = imcomplement(im)</code>	Lấy bù của ảnh <code>im</code>
imdivide	<code>z = imdivide(x, y)</code>	Chia các phần tử của ảnh <code>x</code> cho phần tử tương ứng của <code>y</code> , các giá trị phân số được làm tròn
imlincomb	<code>z = imlincomb(k1, a1, k2, a2, ..., kn, an, k, out_class)</code>	Lấy tổ hợp tuyến tính của các ảnh: $z = k1.*a1 + k2.*a2 + \dots + kn.*an + k$
immultiply	<code>z = immultiply(x, y)</code>	Nhân hai ảnh hoặc nhân một ảnh với một hằng số, nếu kết quả bị tràn thì sẽ được giới hạn lại trong tầm cho phép
imsubtract	<code>z = imsubtract(x, y)</code>	Trừ hai ảnh hoặc trừ một ảnh cho một hằng số, nếu kết quả bị tràn thì sẽ được giới hạn lại trong tầm cho phép

Phép cộng hai ảnh thường dùng để xếp chồng một ảnh lên trên một ảnh khác. Phép cộng một ảnh với một hằng số làm tăng độ sáng của ảnh.

 **Ví dụ 11-2.** Chồng lẫn hai ảnh trên một nền chung:

```
I = imread('rice.png');           % Đọc ảnh thứ nhất
J = imread('cameraman.tif');       % Đọc ảnh thứ hai
K = imadd(I, J);                  % Cộng hai ảnh
imshow(I)                          % Hiển thị ảnh thứ nhất
imshow(J)                          % Hiển thị ảnh thứ hai
imshow(K)                          % Hiển thị ảnh tổng
```

Sau khi thực hiện chương trình ta được ảnh kết quả như sau (**hình 11.5**):



Hình 11.5.

 **Ví dụ 11-3.** Làm tăng độ sáng ảnh bằng phép cộng với hằng số:

```
RGB = imread('peppers.png');  
RGB2 = imadd(RGB, 50);  
subplot(1,2,1); imshow(RGB);  
subplot(1,2,2); imshow(RGB2);
```



Hình 11.6. Tăng độ sáng bằng phép cộng

Phép trừ hai ảnh thường dùng để phát hiện những sự khác nhau giữa các ảnh trong một chuỗi các ảnh của cùng một cảnh. Phép trừ một ảnh cho một hằng số làm giảm độ sáng của ảnh.

Phép nhân một ảnh với một hằng số, còn gọi là scaling, là một phép toán thường gặp trong xử lý ảnh. Nó làm tăng hoặc giảm độ sáng của ảnh tùy theo hệ số nhân là lớn hơn hay nhỏ hơn 1, nhưng khác với phép cộng ảnh với hằng số, phép nhân cho phép bảo toàn độ tương phản của ảnh, do đó ảnh có vẻ thực hơn.

 **Ví dụ 11-4. Làm tăng độ sáng ảnh bằng phép nhân với hằng số:**

```
RGB = imread('peppers.png');  
RGB2 = immultiply(RGB, 1.5);  
subplot(1,2,1); imshow(RGB);  
subplot(1,2,2); imshow(RGB2);
```



Hình 11.7. Tăng độ sáng bằng phép nhân

Phép chia hai ảnh cũng dùng để phát hiện những thay đổi giữa các ảnh liên tiếp của cùng một đối tượng nhưng dưới dạng những thay đổi tỷ lệ.

11.1.5. CÁC HÀM HIỂN THỊ HÌNH ẢNH TRONG MATLAB

Để phục vụ chức năng hiển thị hình ảnh, MATLAB cung cấp hai hàm cơ bản là **image** và **imagesc**. Ngoài ra, trong Image Processing Toolbox cũng có hai hàm hiển thị ảnh khác, đó là **imview** và **imshow**.

- Hàm **image**(X, Y, C) hiển thị hình ảnh biểu diễn bởi ma trận C kích thước MxN lên trục tọa độ hiện hành. X, Y là các vector xác định vị trí của các pixel C(1,1) và C(M,N) trong hệ trục hiện hành. Tọa độ của pixel C(1,1) là (X(1),Y(1)) còn tọa độ của pixel C(M,N) là (X(end),Y(end)). Nếu không cung cấp X, Y, thì MATLAB sẽ mặc định là tọa độ của C(1,1) là (1,1), của C(M,N) là (M,N). Nếu ma trận C chỉ có hai chiều thì MATLAB hiểu rằng đây là ma trận ảnh dạng index với ma trận màu là ma trận màu hiện hành trong hệ thống. Ngoài ra có thể cung cấp thêm các cặp thông số (tên thuộc tính/ giá trị thuộc tính) cho hàm **image**. Ví dụ, hàm **image(..., 'parent', ax)** xác định hệ trục tọa độ mà ảnh sẽ hiển thị trên đó là hệ trục tọa độ ax.
- Hàm **imagesc** có chức năng tương tự như hàm **image**, ngoại trừ việc dữ liệu ảnh sẽ được co giãn (scale) để sử dụng toàn bộ bản đồ màu hiện hành.
- Hàm **imview** cho phép hiển thị hình ảnh trên một cửa sổ riêng, nền Java, gọi là Image Viewer. Image Viewer cung cấp các công cụ cho phép dò tìm và xác định giá trị các pixel một cách linh hoạt. Sử dụng hàm này khi ta cần khảo sát bức ảnh và cần các thông tin về các pixel.



Hình 11.8. Hiển thị ảnh bằng hàm **imview**

- Giống như các hàm **image** và **imagesc**, hàm **imshow** cũng tạo một đối tượng đồ họa thuộc loại **image** và hiển thị ảnh trên một **figure**. Hàm **imshow** sẽ tự động thiết lập các giá trị của các đối tượng **image**, **axes** và **figure** để thể hiện hình ảnh. Sử dụng hàm này trong các trường hợp ta cần lợi dụng các công cụ chú giải và các hỗ trợ in ấn có sẵn trong **figure**.



Hình 11.9. Hiển thị ảnh bằng hàm **imshow**

11.2. CÁC PHÉP BIẾN ĐỔI HÌNH HỌC

Các phép biến đổi hình học là những phép toán biến các điểm ảnh từ vị trí này thành các điểm ảnh ở vị trí khác trong ảnh mới. Đó là những thao tác xử lý ảnh cơ bản như quay, thay đổi kích thước, cắt một phần ảnh, ...

11.2.1. PHÉP NỘI SUY ẢNH

Nội suy là quá trình ước lượng giá trị của ảnh tại một điểm nằm giữa hai pixel có giá trị đã biết. Chẳng hạn, nếu ta thay đổi kích thước ảnh sao cho nó chứa nhiều pixel hơn ảnh gốc, thì giá trị của các pixel thêm vào sẽ được xác định bằng phép nội suy. Phép nội suy cũng là cơ sở để thực hiện các biến đổi hình học khác, ví dụ biến đổi kích thước hoặc quay ảnh, ...

Image Processing Toolbox cung cấp ba phương pháp nội suy ảnh, bao gồm: nội suy theo các lân cận gần nhất, nội suy song tuyến tính và nội suy bicubic. Cả ba phương pháp đều thực hiện theo một nguyên tắc chung: để xác định giá trị của một pixel ảnh nội suy, ta tìm một điểm trong ảnh ban đầu tương ứng với pixel đó, sau đó giá trị của pixel ở ảnh mới sẽ được tính bằng trung bình có trọng số của một tập các pixel nào đó ở lân cận của điểm vừa xác định, trong đó trọng số của các pixel phụ thuộc vào khoảng cách tới điểm này.

Với phương pháp lân cận gần nhất, pixel mới sẽ được gán giá trị của pixel chứa điểm tương ứng của nó (pixel mới) trong ảnh ban đầu. Với phương pháp song tuyến tính, pixel mới sẽ được gán là trung bình có trọng số của các pixel trong một lân cận kích thước 2x2. Với phương pháp bicubic, pixel mới sẽ được gán là trung bình có trọng số của các pixel trong một lân cận kích thước 4x4.

Phương pháp đầu tiên là phương pháp đơn giản và nhanh nhất, nhưng chất lượng không tốt bằng hai phương pháp còn lại. Số pixel được đưa vào để tính trọng số càng nhiều thì chất lượng càng tốt nhưng thời gian càng lâu. Ngoài ra, chỉ có phương pháp đầu tiên là có thể áp dụng cho mọi kiểu ảnh và kiểu dữ liệu vì nó không làm thay đổi tập giá trị của các pixel. Các phương pháp còn lại không thích hợp cho ảnh indexed, nhưng với ảnh RGB thì nên dùng các phương pháp này để bảo đảm chất lượng ảnh.

Với ảnh RGB, phép nội suy được thực hiện một cách riêng biệt trên ba mặt phẳng màu đỏ, lam và lục.

Với ảnh nhị phân dùng nội suy song tuyến tính hoặc bicubic, cần lưu ý đến kiểu dữ liệu, vì giá trị của pixel mới có thể nhận giá trị khác 0 và 1. Nếu ảnh gốc thuộc kiểu `double` thì ảnh mới sẽ là ảnh trắng đen thuộc kiểu `double`, nếu ảnh gốc thuộc kiểu `uint8` thì ảnh mới sẽ là ảnh nhị phân kiểu `uint8`, trong đó các giá trị khác 0 và 1 sẽ được làm tròn về 0 hoặc 1.

11.2.2. THAY ĐỔI KÍCH THƯỚC ẢNH

Hàm `imresize` cho phép người sử dụng thay đổi kích thước của ảnh. Ngoài kích thước ảnh mới, người sử dụng còn có thể xác định phương pháp nội suy sẽ dùng và loại bộ lọc dùng để chống aliasing.

```
>> b = imresize(a,m,Method)
```

Dòng lệnh trên tạo ảnh mới `b` có kích thước gấp `m` lần ảnh gốc `a`. `Method` là một chuỗi xác định phương pháp nội suy sẽ dùng: `'nearest'` (lân cận gần nhất), `'bilinear'` (song tuyến tính) hoặc `'bicubic'`. Phương pháp mặc định là `'nearest'`. Thay vì xác định tỷ số `m`, ta có thể xác định trực tiếp kích thước ảnh mới theo đơn vị pixel bằng cách dùng cú pháp:

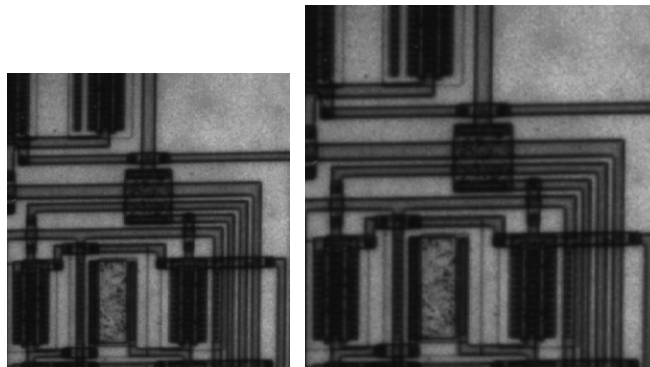
```
>> b = imresize(a,[mrows mcols],method)
```

Trong đó `mrows` và `mcols` là số cột và số hàng của ảnh mới. Hoặc ta cũng có thể xác định cụ thể bậc của bộ lọc chống aliasing (kích thước mặc định là 11x11) hoặc cung cấp cụ thể đáp ứng xung h của bộ lọc theo các cú pháp dưới đây:

```
>> b = imresize(...,method,N) % Dùng bộ lọc kích thước NxN
>> b = imresize(...,method,h) % Dùng bộ lọc có đáp ứng xung h
```

 **Ví dụ 11-5.** Tăng kích thước ảnh lên 1,25 lần dùng phương pháp bicubic với bộ lọc chống aliasing bậc 5:

```
I = imread('circuit.tif');
J = imresize(I,1.25,'bicubic',5);imshow(I)
figure, imshow(J)
```



Hình 11.10. Tăng kích thước ảnh

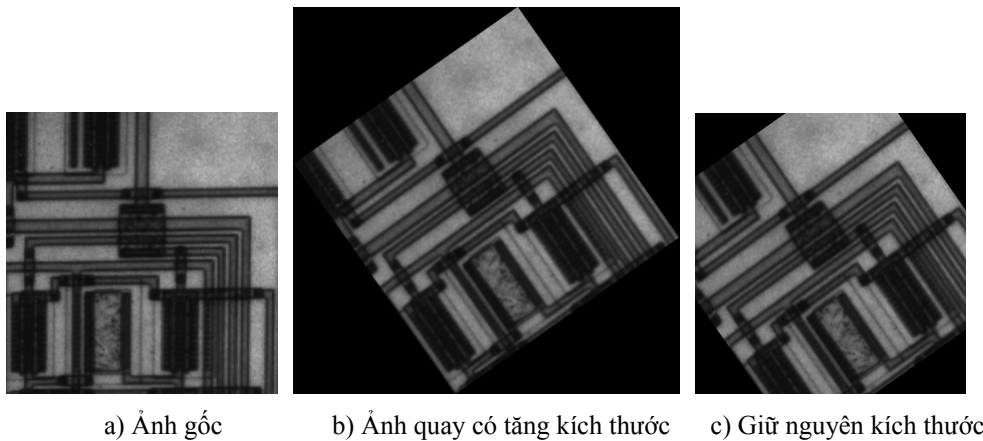
11.2.3. PHÉP QUAY ẢNH

Để thực hiện các phép quay ảnh, ta có thể sử dụng hàm **imrotate**. Ngoài hai thông số cơ bản là ảnh gốc và góc quay, người sử dụng cũng có thể xác định phương pháp nội suy sẽ dùng, và kích thước của ảnh mới (đủ lớn để chứa ảnh mới hay chỉ bằng kích thước ảnh cũ). Thông số mặc định là 'nearest' (lân cận gần nhất) và 'loose' (tăng kích thước nếu cần). Trong trường hợp tăng kích thước, các điểm ảnh ở ngoài phần ảnh gốc sẽ được set về 0 (màu đen). Dưới đây là cú pháp của hàm này, với Bbox là chuỗi xác định kích thước ảnh mới.

```
>> b = imrotate(a,angle,Method,Bbox)
```

 **Ví dụ 11-6.** Quay ảnh đi một góc 35°, dùng phương pháp nội suy song tuyến tính:

```
I = imread('circuit.tif');
J = imrotate(I,35,'bilinear');
K = imrotate(I,35,'bilinear','crop');
imshow(I)
figure, imshow(J)
figure, imshow(K)
```



a) Ảnh gốc

b) Ảnh quay có tăng kích thước

c) Giữ nguyên kích thước

Hình 11.11. Minh họa phép quay ảnh**11.2.4. TRÍCH XUẤT ẢNH**

Khi cần trích xuất một phần của ảnh gốc, ta dùng hàm **imcrop**. Khi sử dụng hàm này, người sử dụng có thể có hai lựa chọn: xác định cụ thể vị trí của phần ảnh cần trích (dưới dạng hình chữ nhật) bằng cách cung cấp các thông số vị trí khi gọi hàm hoặc sử dụng mouse để chọn phần ảnh cần trích xuất.

Nếu chọn cách thứ nhất, ta dùng cú pháp như sau:

```
>> x2 = imcrop(x,map,rect)    % Ảnh indexed
>> a2 = imcrop(a,rect)       % Ảnh grayscale hoặc RGB
```

trong đó `rect = [Xmin Ymin width height]`, với `(Xmin, Ymin)` là tọa độ góc trên bên trái của phần ảnh cần trích, `width` và `height` là chiều rộng và chiều cao của phần ảnh cần trích.

Nếu dùng cách thứ hai, ta không cần cung cấp thông số `rect`, khi thực hiện hàm này, con trỏ sẽ chuyển sang dạng chữ thập, người dùng sẽ drag chuột để chọn phần ảnh cần trích sao đó thả chuột. Hàm **imcrop** sẽ trả về phần ảnh nằm trong phạm vi xác định bởi mouse.

Nếu không cung cấp thông số ảnh gốc, hàm **imcrop** sẽ mặc định chọn ảnh trên hệ trục tọa độ hiện hành. Ngoài ra, trong trường hợp xác định bằng mouse, người sử dụng có thể truy xuất các thông tin về vị trí và kích thước của phần ảnh đã chọn bằng cách yêu cầu thêm các output của hàm này:

```
>> [A2,rect] = imcrop(A)
>> [X2,rect] = imcrop(X,map)
```

11.2.5. THỰC HIỆN PHÉP BIẾN ĐỔI HÌNH HỌC TỔNG QUÁT

Ngoài các phép biến đổi hình học cụ thể trên đây, MATLAB còn cho phép thực hiện các phép biến đổi hình học khác do người sử dụng tùy định bằng cách cung cấp một hàm thực hiện biến đổi hình học tổng quát, đó là hàm **imtransform**. Để thực hiện một phép biến đổi hình học nào đó, người sử dụng cần cung cấp ảnh cần biến đổi `A` và cấu trúc của phép biến đổi hình học, gọi là `TFORM`.

```
>> B = imtransform(A,TFORM,interp)
>> [B,XData,YData] = imtransform(...,param1,val1,param2,val2,...)
```

trong đó, `interp` là chuỗi xác định phương pháp nội suy sẽ dùng. `(Xdata, Ydata)` xác định vị trí của ảnh `B` trong hệ trục `X-Y`. Ngoài ra có thể cung cấp thêm các cặp thông số (tên thông

số / giá trị thông số) để xác định các thông số cụ thể của phép biến đổi. Bạn đọc có thể tìm hiểu các thông số này bằng cách gõ lệnh `help imtransform` từ cửa sổ lệnh của MATLAB.

Như vậy, vấn đề quan trọng nhất khi gọi hàm này là phải xác định cấu trúc của phép biến đổi. Việc này được thực hiện bằng cách sử dụng các hàm xây dựng cấu trúc biến đổi, trong đó thông dụng nhất là hai hàm **maketform** và **cp2tform**.

Bảng 11.4. Các dạng biến đổi hình học có thể dùng với hàm **maketform**

TFORM_type	Mô tả	Cú pháp
'affine'	Các phép biến đổi affine, bao gồm phép dịch, phép quay, co giãn, và xén. Đường thẳng biến thành đường thẳng, bảo toàn tính song song nhưng không bảo toàn góc (hình chữ nhật có thể trở thành hình bình hành)	$T = \text{maketform}('affine', A)$ $T = \text{maketform}('affine', U, X)$ A là ma trận $(N+1) \times (N+1)$ hoặc $(N+1) \times N$ sao cho vector $U1$ ($1 \times N$) sẽ biến thành vector $X1 = U1 * A(1:N,1:N) + A(N+1,1:N)$ U và X là các ma trận sao cho mỗi hàng của U biến thành hàng tương ứng của X sau phép biến đổi. U và X đều có kích thước 3×2 , gồm tọa độ của 3 đỉnh của 1 tam giác.
'projective'	Các phép biến đổi trong đó các đường thẳng vẫn biến thành đường thẳng nhưng các đường song song trở nên hội tụ tại 1 điểm nào đó trong hoặc ngoài ảnh (có thể là vô cực)	$T = \text{maketform}('projective', A)$ $T = \text{maketform}('projective', U, X)$ A là ma trận $(N+1) \times (N+1)$ sao cho vector $U1$ ($1 \times N$) sẽ biến thành vector $X1 = W(1:N)/W(N+1)$ với $W = [U1] * A$. U và X là các ma trận sao cho mỗi hàng của U biến thành hàng tương ứng của X sau phép biến đổi. U và X đều có kích thước 4×2 , gồm tọa độ của 4 đỉnh của 1 tứ giác.
'box'	Trường hợp đặc biệt của phép biến đổi affine, trong đó mỗi chiều được dịch và co giãn một cách độc lập với nhau	$T = \text{maketform}('box', Tsize, low, high)$ $T = \text{maketform}('box', inbounds, outbounds)$ Phép biến đổi sẽ ánh xạ vùng giới hạn bởi các đỉnh có tọa độ xác định bởi $inbounds$ (hoặc $\text{ones}(1, N)$ và $Tsize$) thành vùng giới hạn bởi các đỉnh có tọa độ xác định bởi $outbounds$ (hoặc low và $high$) $inbounds$ và $outbounds$: $2 \times N$ $Tsize, low, high$: $1 \times N$
'custom'	Các phép biến đổi do người dùng tự định nghĩa	$T = \text{maketform}('custom', Ndims_in, Ndims_out, forward_fcn, inverse_fcn, TData)$ $Ndims_in, Ndims_out$: số chiều của ảnh vào và ra. $forward_fcn, inverse_fcn$: các hàm thực hiện biến đổi thuận và nghịch $TData$: các thông số khác kèm theo
'composite'	Kết hợp hai hay nhiều phép biến đổi	$T = \text{maketform}('composite', T1, T2, \dots, TL)$ $T1, T2, \dots, TL$: các phép biến đổi đã định nghĩa trước

Cú pháp chung của hàm **maketform**:

```
>> T = maketform(TFORM_type, ...)
```

trong đó `TFORM_type` là một chuỗi xác định dạng cấu trúc biến đổi hình học, và sau đó là các thông số đi kèm tùy thuộc vào từng dạng cấu trúc cụ thể. Các dạng cấu trúc này được trình bày trong bảng 11.4.

Hàm `cp2tform` trả về cấu trúc của phép biến đổi bằng cách suy từ các cặp điểm điều khiển trong ảnh gốc và sau khi biến đổi:

```
>> TFORM = cp2tform(input_points,base_points,TFORM_type,order)
```

`input_points` và `base_points` là các ma trận $M \times 2$ xác định tọa độ (X,Y) của M điểm điều khiển trong ảnh biến đổi và trong ảnh gốc. `TFORM_type` có thể là một trong những chuỗi sau: 'linear conformal', 'affine', 'projective', 'polynomial', 'piecewise linear', 'lwm'. Nếu là 'polynomial' thì cần cung cấp thêm thông số order cho biết bậc của đa thức (mặc định bằng 3).

 **Ví dụ 11-7.** Thực hiện phép xạ ảnh để tạo hiệu ứng 3D cho ảnh “bàn cờ”:

```
I = checkerboard(20,1,1); % Tạo ảnh bàn cờ
figure; imshow(I)

T = maketform('projective',[1 1; 41 1; 41 41; 1 41],...
[5 5; 40 5; 35 30; -10 30]); % Định nghĩa phép biến đổi xạ ảnh

R = makesampler('cubic','circular'); % Định nghĩa cấu trúc resampler

K = imtransform(I,T,R,'Size',[100 100],'XYScale',1); % Thực hiện phép xạ ảnh

figure, imshow(K)
```



Hình 11.12.

11.3. CÁC PHÉP BIẾN ĐỔI ẢNH

Phép biến đổi ảnh là một dạng biểu diễn toán học của ảnh ngoài cách biểu diễn bằng ma trận thông thường. Các phép biến đổi ảnh có nhiều ứng dụng như nâng cao chất lượng ảnh, trích xuất các đặc tính, hoặc nén ảnh.

11.3.1. BIẾN ĐỔI FOURIER

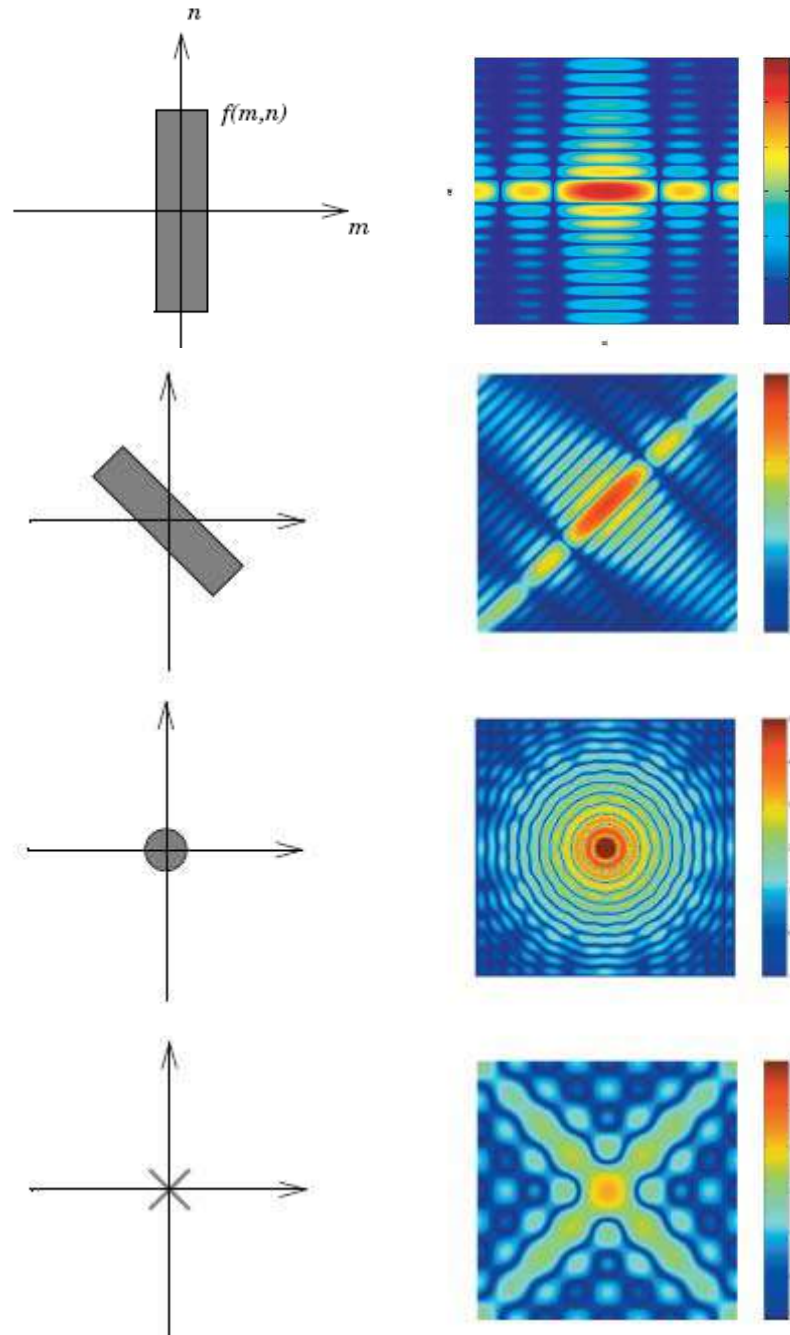
Phép biến đổi Fourier biểu diễn ảnh dưới dạng tổng của các lũy thừa phức của các thành phần, biên độ, tần số và pha khác nhau của ảnh. Phép biến đổi Fourier có vai trò rất quan trọng trong các ứng dụng rộng rãi của xử lý ảnh số, bao gồm nâng cao chất lượng ảnh, phân tích, khôi phục và nén ảnh.

Nếu $f(m,n)$ là một hàm của hai biến không gian rời rạc m và n , thì biến đổi Fourier hai chiều của $f(m,n)$ được định nghĩa như sau:

$$F(\omega_1, \omega_2) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} f(m,n) e^{-jm\omega_1} e^{-jn\omega_2} \quad (11.1)$$

ω_1, ω_2 là các biến tần số với đơn vị rad/mẫu. $F(\omega_1, \omega_2)$ gọi là biểu diễn trong miền tần số của $f(m,n)$. $F(\omega_1, \omega_2)$ là hàm tuần hoàn chu kỳ 2π đối với các biến ω_1, ω_2 , do đó chỉ cần xét

$-\pi \leq \omega_1, \omega_2 \leq \pi$. $F(0,0)$ chính là tổng các giá trị của $f(m,n)$ và được gọi là thành phần hằng số hoặc thành phần DC của biến đổi Fourier. Nếu $f(m,n)$ biểu diễn độ sáng của ảnh X ở vị trí pixel (m,n) thì $F(\omega_1, \omega_2)$ chính là biến đổi Fourier của ảnh X . Các hình vẽ dưới đây minh họa phép biến đổi Fourier cho một vài dạng ảnh $f(m,n)$, trong đó logarith của biến đổi Fourier, $\log|F(\omega_1, \omega_2)|$ được thể hiện dưới dạng ảnh.



Hình 11.13. Biến đổi Fourier của một số dạng ảnh đơn giản

Do các dữ liệu trên máy tính được lưu trữ dưới dạng rời rạc, cụ thể là dữ liệu ảnh được tổ chức theo đơn vị pixel nên phép biến đổi Fourier cũng được rời rạc hoá thành biến đổi Fourier rời rạc (DFT – Discrete Fourier Transform). Giả sử rằng hàm $f(m,n)$ chỉ khác 0 trong miền $(0 \leq m \leq M-1, 0 \leq n \leq N-1)$. Các phép biến đổi DFT thuận và nghịch kích thước $M \times N$ được định nghĩa như sau:

$$F(p, q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) e^{-j(2\pi/M)pm} e^{-j(2\pi/N)qn} \quad (0 \leq p \leq M-1, 0 \leq q \leq N-1) \quad (11.2)$$

$$f(m, n) = \frac{1}{MN} \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) e^{j(2\pi/M)pm} e^{j(2\pi/N)qn} \quad (0 \leq m \leq M-1, 0 \leq n \leq N-1) \quad (11.3)$$

$F(p, q)$ được gọi là các hệ số của biến đổi DFT. Trong MATLAB, các mảng và ma trận có chỉ số bắt đầu từ 1, do đó phần tử $F[1,1]$ sẽ ứng với hệ số $F(0,0)$, phần tử $F[2,2]$ ứng với $F(1,1)$, ...

Các hàm MATLAB **fft**, **fft2**, **fftn** sẽ thực hiện các phép biến đổi Fourier rời rạc 1 chiều, 2 chiều và n chiều. Các hàm **ifft**, **ifft2**, **ifftn** thực hiện các phép biến đổi DFT ngược. Với các ứng dụng xử lý ảnh, ta chỉ cần quan tâm đến các hàm **fft2** và **ifft2**.

```
>> F = fft2(X, Mrows, Ncols)
```

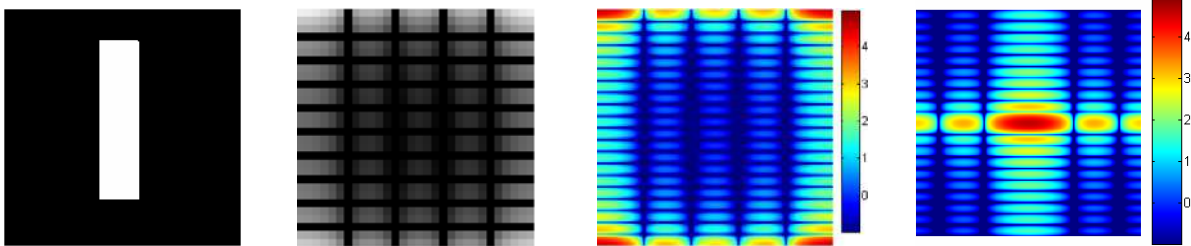
```
>> X = ifft2(F, Mrows, Ncols)
```

trong đó, $Mrows \times Ncols$ là kích thước của biến đổi DFT. Nếu ảnh ban đầu có kích thước nhỏ hơn thì MATLAB sẽ tự động thêm vào các zero pixel trước khi biến đổi.

Sau khi thực hiện biến đổi DFT bằng hàm **fft2**, thành phần DC của biến đổi sẽ nằm ở góc trên bên trái của ảnh. Có thể dịch chuyển thành phần này về trung tâm bằng cách dùng hàm **fftshift**.

 **Ví dụ 11-8.** Thực hiện phép biến đổi Fourier rời rạc cho một ảnh đơn giản:

```
f = zeros(30,30); % Tạo một ảnh kích thước 30x30
f(5:24,13:17) = 1;
imshow(f, 'notruesize') % Hiển thị ảnh ban đầu
F = fft2(f); % Biến đổi DFT với kích thước 30x30
F2 = log(abs(F));
imshow(F2, [-1 5], 'notruesize');
colormap(jet); colorbar % Hiển thị ảnh biến đổi
F = fft2(f, 256, 256); % Biến đổi DFT với kích thước 256x256
imshow(log(abs(F)), [-1 5]); colormap(jet); colorbar % Hiển thị ảnh
F2 = fftshift(F); % Chuyển thành phần DC về trung tâm
imshow(log(abs(F2)), [-1 5]); colormap(jet); colorbar % Hiển thị ảnh
```



a) Ảnh gốc b) Biến đổi DFT 30x30 c) Biến đổi DFT 256x256 d) Sử dụng **fftshift**

Hình 11.14. Minh họa biến đổi DFT

Phép biến đổi Fourier có nhiều ứng dụng, chẳng hạn tìm đáp ứng tần số của các bộ lọc tuyến tính, tính tích chập, tìm ảnh tương quan, ... Trong các chương sau, đọc giả có thể tìm thấy nhiều ví dụ ứng dụng biến đổi này trong xử lý ảnh.

11.3.2. BIẾN ĐỔI COSINE RỜI RẠC

Biến đổi cosine rời rạc (DCT – Discrete cosine Transform) biểu diễn ảnh dưới dạng tổng của các cosine của các thành phần biên độ và tần số khác nhau của ảnh. Đặc điểm nổi bật của biến đổi này là: hầu hết các thông tin về ảnh chỉ tập trung trong một vài hệ số của biến đổi DCT, trong khi các hệ số còn lại chỉ chứa rất ít thông tin. Do đó, phép biến đổi này là cơ sở cho các kỹ thuật nén ảnh. Ví dụ, giải thuật nén có tổn hao theo chuẩn quốc tế JPEG là giải thuật được xây dựng trên cơ sở biến đổi DCT.

Theo định nghĩa, biến đổi DCT 2 chiều của một ma trận A kích thước $M \times N$ là:

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N} \quad \text{với} \quad \begin{matrix} 0 \leq p \leq M-1 \\ 0 \leq q \leq N-1 \end{matrix} \quad (11.4)$$

trong đó:
$$\alpha_p = \begin{cases} 1/\sqrt{M} & p = 0 \\ \sqrt{(2/M)} & 1 \leq p \leq M-1 \end{cases} \quad \text{và} \quad \alpha_q = \begin{cases} 1/\sqrt{N} & q = 0 \\ \sqrt{(2/N)} & 1 \leq q \leq N-1 \end{cases} \quad (11.5)$$

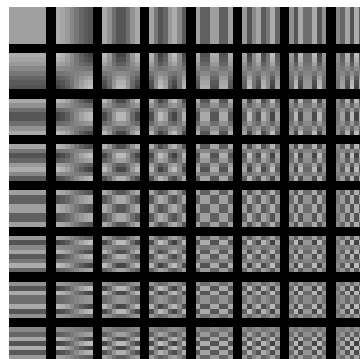
Các giá trị B_{pq} được gọi là các hệ số của biến đổi DCT. Lưu ý rằng trong MATLAB, hệ số B_{00} sẽ được biểu diễn là $B[1,1]$, B_{12} sẽ được biểu diễn bởi $B[2,3]$, ...

Biến đổi ngược DCT được xác định bởi phương trình sau:

$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N} \quad \text{với} \quad \begin{matrix} 0 \leq m \leq M-1 \\ 0 \leq n \leq N-1 \end{matrix} \quad (11.6)$$

Từ (11.6) ta thấy rằng mỗi ma trận A kích thước $M \times N$ có thể biểu diễn dưới dạng tổng của MN hàm số dạng $A \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}$ với trọng số tương ứng là B_{pq} .

Các hàm này gọi là các hàm cơ sở của biến đổi DCT. Hàm cơ bản ứng với $p = q = 0$ gọi là hàm cơ sở DC và B_{00} được gọi là hệ số DC của biến đổi DCT. Hình vẽ 11.15 minh họa 64 hàm cơ sở của biến đổi DCT của ma trận 8×8 , trong đó hàm cơ sở DC nằm ở góc trên bên trái.



Hình 11.15.

Phép biến đổi DCT thuận và nghịch được thực hiện bằng các hàm **dct2** và **idct2**. Các hàm này sử dụng giải thuật dựa theo FFT để tăng tốc độ tính toán. Hàm này thích hợp với các ảnh có kích thước lớn.

```
>> B = dct2(A,M,N) % hoặc dct2(A,[M N]) hoặc dct2(A)
```

```
>> A = idct2(B,M,N) % hoặc idct2(B,[M N]) hoặc idct2(B)
```

Ngoài ra, còn một cách khác để tính DCT bằng MATLAB, đó là sử dụng ma trận biến đổi T. Cách này chỉ áp dụng cho các ma trận vuông. Ma trận biến đổi T có kích thước MxM được định nghĩa như sau:

$$T_{pq} = \begin{cases} 1/\sqrt{M} & p=0, 0 \leq q \leq M-1 \\ \sqrt{\frac{2}{M}} \cos \frac{\pi(2q+1)p}{2M} & \text{nơi khác} \end{cases} \quad (11.7)$$

Với A là ma trận kích thước MxM thì T*A sẽ là ma trận kích thước MxM trong đó mỗi cột của nó là biến đổi DCT 1 chiều của các cột tương ứng của A. Biến đổi DCT 2 chiều của A sẽ là B = T*A*T'. T là một ma trận thực, trực chuẩn, nghĩa là $T^{-1} = T'$, do đó biến đổi DCT ngược của B chính là A = T'*B*T. Trong MATLAB, ma trận T là kết quả trả về khi ta gọi hàm **dctmtx**.

```
>> T = dctmtx(M) % MxM là kích thước của T
```

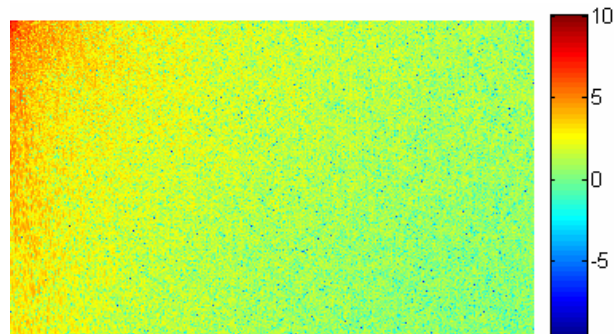
Sau đây là một ví dụ minh họa cách thực hiện biến đổi DCT trong MATLAB. Ta thực hiện DCT trên ảnh “autumn.tif”. Sau đó các hệ số có biên độ <10 sẽ được cho bằng 0, rồi thực hiện biến đổi DCT ngược. Vì hầu hết các thông tin của ảnh chỉ tập trung trong một vài hệ số DCT nên ảnh thu được sẽ không khác biệt nhiều so với ảnh gốc.



a) Ảnh gốc



b) Ảnh sau khi nén các hệ số DCT <10 về 0



c) Biến đổi DCT của ảnh gốc

Hình 11.16.

Ví dụ 11-9. Thực hiện biến đổi DCT và DCT ngược:

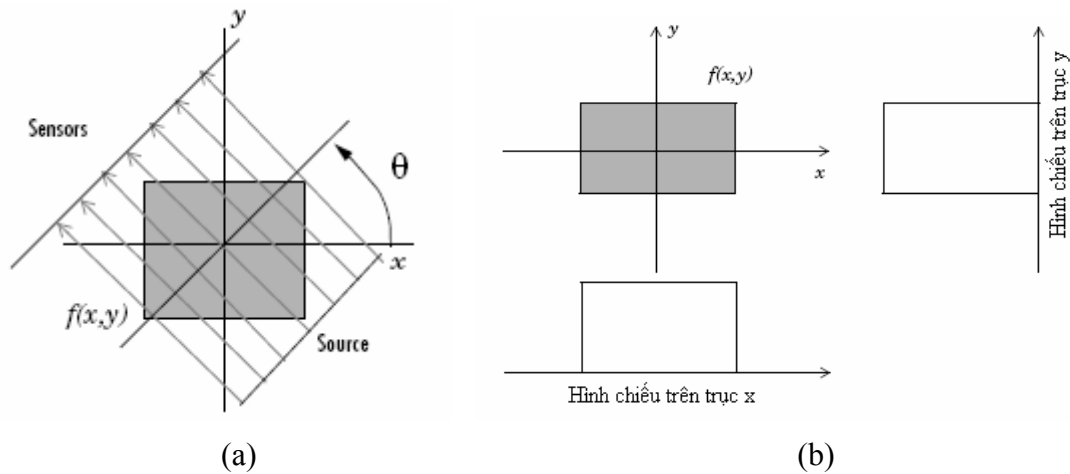
```
RGB = imread('autumn.tif'); % Đọc ảnh
I = rgb2gray(RGB); % Chuyển sang dạng grayscale
J = dct2(I); % Biến đổi DCT
imshow(log(abs(J)),[]), colormap(jet(64)), colorbar % Hiển thị biến đổi DCT
J(abs(J) < 10) = 0; % Nén các hệ số <10 về 0
```

```
K = idct2(J); % Biến đổi ngược DCT
imview(I) % Hiển thị ảnh gốc
imview(K,[0 255]) % Hiển thị ảnh sau khi nén
```

Kết quả được trình bày ở hình 11.16.

11.3.3. BIẾN ĐỔI RADON

Phép biến đổi Radon, được thực hiện bởi hàm `radon` trong MATLAB, biểu diễn ảnh dưới dạng các hình chiếu của nó dọc theo các hướng xác định. Hình chiếu của một hàm hai biến $f(x,y)$ là một tập hợp các tích phân đường. Hàm radon tính các tích phân đường từ nhiều điểm nguồn dọc theo các đường dẫn song song, gọi là các tia chiếu (beam), theo một hướng xác định nào đó. Các tia chiếu này nằm cách nhau 1 pixel. Để biểu diễn toàn bộ ảnh, hàm radon sẽ lấy nhiều hình chiếu song song của ảnh từ các góc quay khác nhau bằng cách xoay các điểm nguồn quanh tâm của ảnh. Quá trình này được minh họa ở **hình 11.17a**.



Hình 11.17.

Ví dụ, tích phân đường của $f(x,y)$ theo hướng thẳng đứng chính là hình chiếu của $f(x,y)$ trên trục x, còn tích phân đường của $f(x,y)$ theo hướng nằm ngang chính là hình chiếu của $f(x,y)$ trên trục y (**hình 11.17b**).

Tổng quát, biến đổi Radon của $f(x,y)$ ứng với góc quay θ là tích phân đường của f dọc theo trục y' :

$$R_{\theta}(x') = \int_{-\infty}^{+\infty} f(x' \cos \theta - y' \sin \theta, x' \sin \theta + y' \cos \theta) dy' \quad (11.8)$$

Trong đó (x', y') là hệ trục tọa độ có được bằng cách xoay hệ trục (x, y) đi một góc bằng θ :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (11.9)$$

Hình 11.8 mô tả cách tính biến đổi Radon với góc quay θ . Trong MATLAB, biến đổi Radon được tính bằng hàm **radon** với cú pháp như sau:

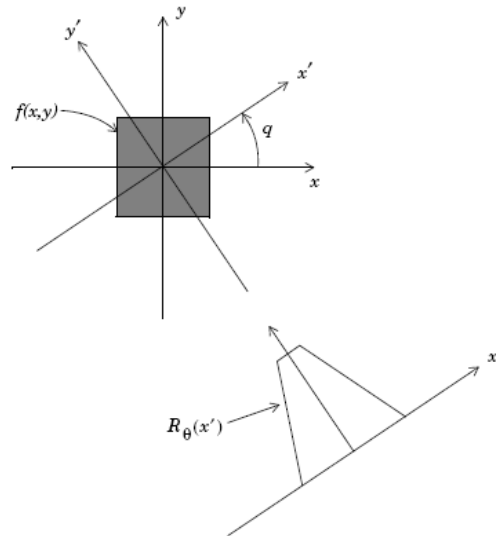
```
>> [R,Xp] = radon(I,theta)
```

Trong đó:

θ là một vector gồm các góc quay cần tính biến đổi Radon

R là một ma trận mà mỗi cột của nó là biến đổi Radon $R(x')$ ứng với một góc quay θ .

x_p là vector chứa các tọa độ x' tương ứng (x_p là như nhau đối với mọi góc quay θ).



Hình 11.18.

Ví dụ 11-10. Khảo sát biến đổi Radon của ảnh một đối tượng hình vuông:

a. Vẽ biến đổi Radon như một hàm của x' trong hai trường hợp $\theta = 0^\circ$ và $\theta = 45^\circ$

b. Hiển thị biến đổi Radon dưới dạng một ảnh

```
I = zeros(100,100);           % Tạo ảnh gốc
I(25:75, 25:75) = 1;          % Ảnh hình vuông
imshow(I)                      % Hiển thị ảnh gốc

[R,xp] = radon(I,[0 45]);       % Biến đổi Radon ứng với các góc 0° và 45°
figure; plot(xp,R(:,1)); title('R_{0^o} (x\prime)') % Vẽ biến đổi Radon ...
figure; plot(xp,R(:,2)); title('R_{45^o} (x\prime)') % ... vừa tính được
theta = 0:180;                  % Cho góc quay thay đổi từ 0° đến 180°
[R,xp] = radon(I,theta); % Tìm biến đổi Radon
imagesc(theta,xp,R);           % Hiển thị biến đổi Radon
title('R_{\theta} (X\prime)');
xlabel('\theta (degrees)');
ylabel('X\prime');
set(gca,'XTick',0:20:180);
colormap(hot);
colorbar
```

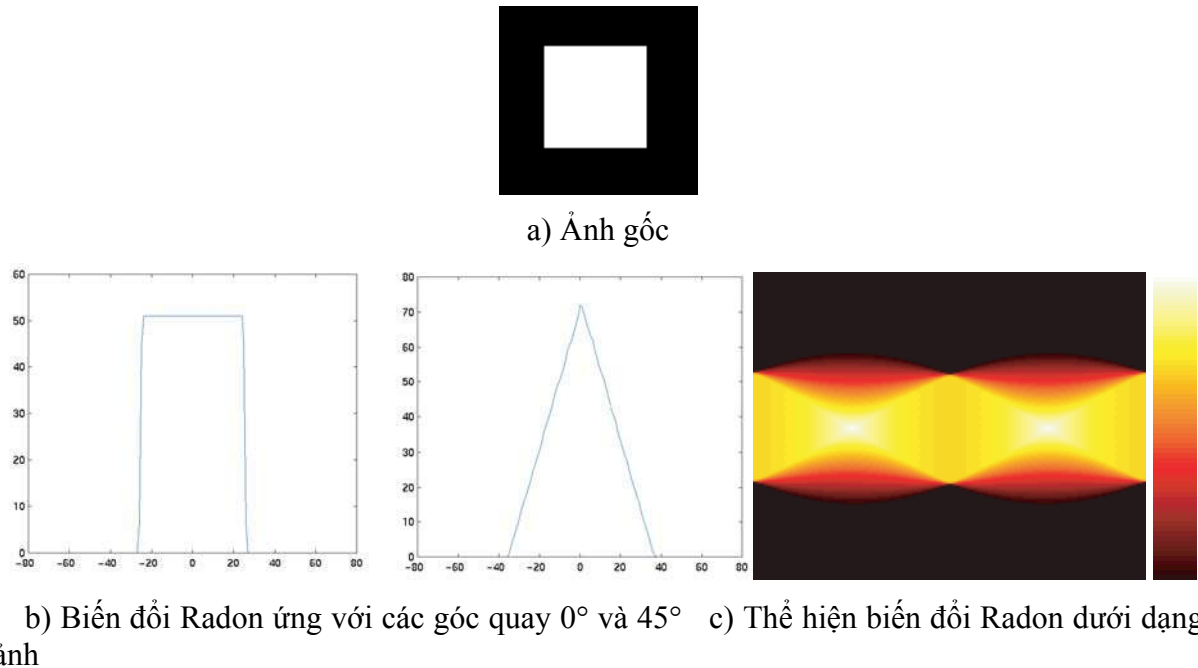
Kết quả thực thi đoạn chương trình trên được thể hiện trong **hình 11.19**.

Từ các dữ liệu hình chiếu của ảnh ứng với các góc θ khác nhau, có thể tái tạo lại ảnh gốc bằng phép biến đổi radon ngược. Hàm **iradon** sẽ thực hiện công việc này:

```
>> I = iradon(R,theta)
```

Trong ví dụ 11.10 ở trên, các dữ liệu hình chiếu được lấy từ ảnh gốc qua phép biến đổi Radon thuận. Tuy nhiên, trong hầu hết các ứng dụng thực tế, các dữ liệu này có được nhờ các thiết bị chuyên dụng, còn ảnh gốc là đối tượng mà ta chưa biết. Chẳng hạn, trong kỹ thuật X quang,

dữ liệu hình chiếu thu được bằng cách đo mức độ suy hao phóng xạ khi chiếu tia phóng xạ qua vật mẫu dưới các góc khác nhau. Ảnh gốc ở đây là ảnh mặt cắt ngang của vật mẫu, trong đó các giá trị độ sáng chính là mật độ vật chất của vật mẫu tại vị trí đó. Từ các dữ liệu hình chiếu thu thập từ thiết bị X quang, người ta sẽ khôi phục lại ảnh gốc bằng biến đổi Radon ngược. Như vậy, ta thấy rằng một ứng dụng quan trọng của biến đổi Radon là để khôi phục ảnh trong kỹ thuật chiếu xạ. Ví dụ 11-11 dưới đây làm rõ nhận xét này.



Hình 11.9.

Ví dụ 11-11. Ứng dụng phép biến đổi Radon: khôi phục ảnh từ các dữ liệu hình chiếu

Trong ví dụ này, chúng ta tạo các dữ liệu hình chiếu giả (thay vì thu thập các dữ liệu này bằng máy) bằng cách lấy biến đổi Radon của một ảnh tương tự như các ảnh X quang thực. Sau đó tái tạo ảnh gốc bằng biến đổi Radon ngược. Chất lượng ảnh tái tạo sẽ phụ thuộc vào số lượng hình chiếu có được.

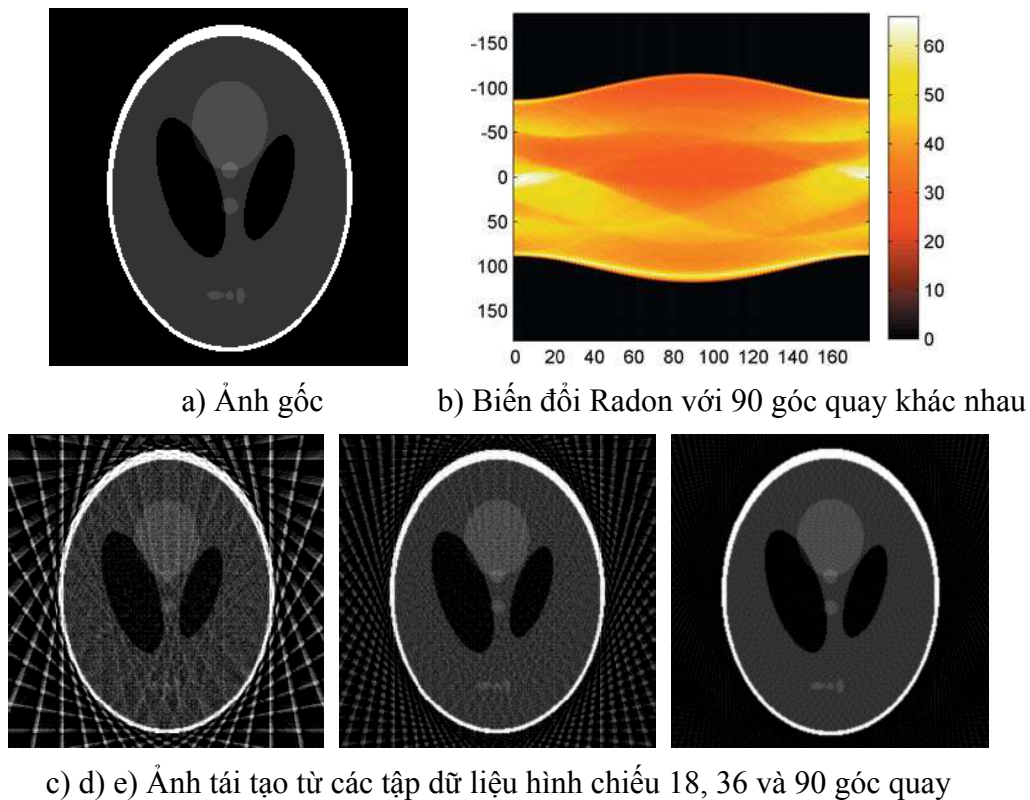
```
P = phantom(256); % Tạo ảnh phantom
imshow(P)          % Hiển thị ảnh gốc

theta1 = 0:10:170; [R1,xp] = radon(P,theta1); % Tạo các tập dữ liệu hình chiếu
theta2 = 0:5:175; [R2,xp] = radon(P,theta2); % ... ở ba mức độ khác nhau
theta3 = 0:2:178; [R3,xp] = radon(P,theta3);

figure, imagesc(theta3,xp,R3); colormap(hot); colorbar % Hiển thị tập dữ liệu
xlabel('\theta'); ylabel('x\prime'); % ... thứ ba

I1 = iradon(R1,10); % Khôi phục ảnh bằng biến đổi Radon ngược
I2 = iradon(R2,5);  % ... sử dụng ba tập dữ liệu hình chiếu
I3 = iradon(R3,2);

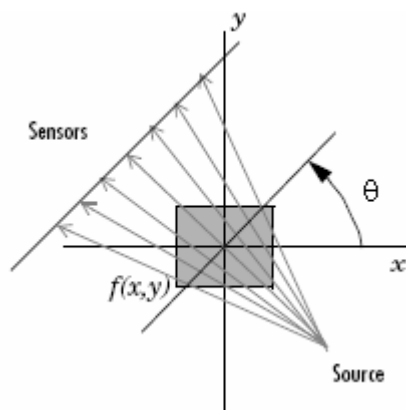
imshow(I1)          % Hiển thị các ảnh tái tạo
figure, imshow(I2)
figure, imshow(I3)
```



Hình 11.20.

11.3.4. PHÉP BIẾN ĐỔI FAN-BEAM

Phép biến đổi fan-beam cũng biểu diễn ảnh dưới dạng một tập các hình chiếu của ảnh. Một hình chiếu của một hàm hai biến $f(x,y)$ được định nghĩa là tích phân đường của $f(x,y)$ dọc theo các đường dẫn đồng quy tại một điểm nguồn (thay vì song song với nhau như ở biến đổi Radon). Để biểu diễn một ảnh, ta tính toán các hình chiếu tại các góc quay khác nhau khi quay điểm nguồn quanh tâm của ảnh (xem hình 11.21).



Hình 11.21.

Các hàm thực hiện biến đổi fan-beam thuận và nghịch là **fanbeam** và **ifanbeam**.

```
>> F = fanbeam(I,D,param1,val1,param2,val2,...)
```

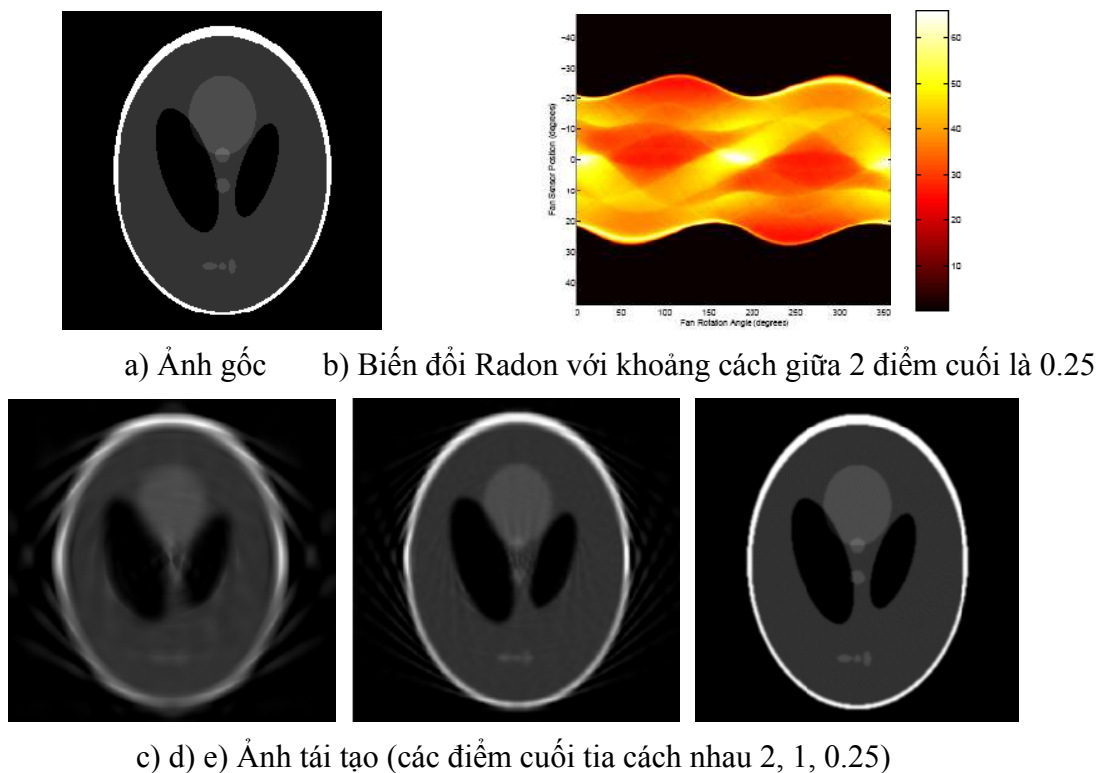
```
>> I = ifanbeam(F,D,param1,val1,param2,val2,...)
```

với D là khoảng cách từ điểm nguồn đến tâm của ảnh. Các góc quay được thay đổi liên tục từ 0° đến 360° , mỗi lần tăng 1° . Có thể thay đổi giá số 1° này bằng cách cung cấp thêm thông số

'FanRotationIncrement' và giá trị gia số mới. Ngoài ra có thể thay đổi các thông số khác bằng cách cung cấp các cặp (param1/val1) cho hàm **fanbeam** và **ifanbeam**.

 **Ví dụ 11-12.** Ứng dụng phép biến đổi Fan-beam: khôi phục ảnh từ các dữ liệu hình chiếu:

```
P = phantom(256); % Tạo ảnh phantom
imshow(P)          % Hiển thị ảnh gốc
D = 250;           % Khoảng cách từ nguồn đến tâm của ảnh
dsensor1 = 2;      % Xây dựng tập dữ liệu hình chiếu trong 3 trường hợp
F1 = fanbeam(P,D,'FanSensorSpacing',dsensor1); % ... tùy khoảng cách giữa
dsensor2 = 1;      % ... hai điểm cuối tia liên tiếp là 2,1 hoặc 0.25
F2 = fanbeam(P,D,'FanSensorSpacing',dsensor2);
dsensor3 = 0.25
[F3, sensor_pos3, fan_rot_angles3] = fanbeam(P,D,...
'FanSensorSpacing',dsensor3);
figure, imagesc(fan_rot_angles3, sensor_pos3, F3)% Hiển thị biến đổi fan-
beam
colormap(hot); colorbar
xlabel('Fan Rotation Angle (degrees)')
ylabel('Fan Sensor Position (degrees)')
output_size = max(size(P));
Ifan1 = ifanbeam(F1,D,          % Khôi phục ảnh từ các tập dữ liệu hình chiếu
'FanSensorSpacing',dsensor1,'OutputSize',output_size);
figure, imshow(Ifan1)
Ifan2 = ifanbeam(F2,D,
'FanSensorSpacing',dsensor2,'OutputSize',output_size);
figure, imshow(Ifan2)
Ifan3 = ifanbeam(F3,D,
'FanSensorSpacing',dsensor3,'OutputSize',output_size);
figure, imshow(Ifan3)
```



Hình 11.22

✎ Bài tập 11-1.

Chọn một số file ảnh có trong máy tính của bạn và thực hiện các thao tác sau:

- i. Đọc ảnh bằng hàm **imread**
- ii. Dùng hàm **imfinfo** để xem các thông tin của ảnh
- iii. Chuyển đổi qua lại giữa các kiểu ảnh indexed, gray scale, nhị phân và RGB
- iv. Chuyển đổi kiểu dữ liệu của ảnh: `logical`, `double`, `uint8` hoặc `uint16`.
- v. Hiển thị ảnh sau khi chuyển
- vi. Lưu ảnh mới vào một file khác cùng format với file ảnh ban đầu.

✎ Bài tập 11-2.

- a. Load ảnh **cameraman.tif** vào một ma trận các mức xám (dùng **imread**) và chuyển ma trận này sang kiểu `double`.
- b. Tìm median của các giá trị mức xám (dùng hàm **median**) của ảnh.
- c. Tạo một bản copy của ma trận nói trên, sau đó thay các giá trị <127 trở thành 0. Dùng lệnh `for` để thực hiện việc này.
- d. Tạo một bản copy của ma trận nói trên, sau đó thay các giá trị >127 trở thành 255. Cố gắng không dùng các vòng lặp.
- e. Tạo ảnh âm bản bằng cách thay các giá trị mức xám v bằng $255 - v$. Không được dùng vòng lặp.

f. Chuyển các ma trận ảnh ở câu a, c, d, e về kiểu `uint8`. Hiển thị 4 ảnh trên một figure 2x2.

🔗 **Bài tập 11-3.**

Chọn một vài ảnh nào đó trong máy tính và thực hiện các phép biến đổi hình học cơ bản: phép quay, phép trích xuất một phần ảnh.

🔗 **Bài tập 11-4.**

Thực hiện các phép biến đổi hình học tự định nghĩa để tạo các hiệu ứng mỹ thuật cho ảnh `peppers.png`.

🔗 **Bài tập 11-5.**

Viết một hàm MATLAB trả về giá trị mức xám tại một điểm có tọa độ không nguyên dùng phép nội suy song tuyến tính: `p = getpixel(x, y)` (xét ảnh `uint8` hoặc `uint16`).

🔗 **Bài tập 11-6.**

Viết hàm MATLAB thực hiện phép biến đổi hình học bằng ma trận 2x2 định nghĩa như sau:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \text{ hay } u = A.x$$

Trong đó $u = [u \ v]^T$ và $x = [x \ y]^T$ là các tọa độ trước và sau khi biến đổi, $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ là một ma trận kích thước 2x2 cho trước gọi là ma trận biến đổi thuận.

Định nghĩa hàm này dưới dạng `ImageOut = map2x2(ImageIn, A)`, với A là ma trận biến đổi thuận. Sử dụng phương pháp nội suy lân cận gần nhất để thực hiện biến đổi.

🔗 **Bài tập 11-7.**

Sử dụng chương trình của bài tập 11-6 để thực hiện biến đổi hình học cho ảnh checkerboard với các ma trận biến đổi thuận như sau:

$$\begin{array}{llll} \text{a. } A = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{2} \\ -\frac{1}{2} & \frac{1}{\sqrt{2}} \end{bmatrix} & \text{b. } A = \begin{bmatrix} 1 & \frac{1}{10} \\ 0 & 1 \end{bmatrix} & \text{c. } A = \begin{bmatrix} 1 & 0 \\ \frac{1}{10} & 1 \end{bmatrix} & \text{d. } A = \begin{bmatrix} 1 & \frac{1}{10} \\ \frac{1}{10} & 1 \end{bmatrix} \\ \text{e. } A = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix} & \text{f. } A = \begin{bmatrix} \frac{1}{4} & 0 \\ 0 & \frac{1}{4} \end{bmatrix} & \text{g. } A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & \end{array}$$

🔗 **Bài tập 11-8.**

Viết hàm MATLAB thực hiện phép dịch được định nghĩa như sau:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \text{ hay } u = Ix + x_0$$

với $u = [u \ v]^T$, $x = [x \ y]^T$ là các tọa độ trước và sau khi biến đổi, $x_0 = [x_0 \ y_0]^T$ là vector hằng, I là ma trận đơn vị kích thước 2x2.

Định nghĩa hàm này dưới dạng `ImageOut = translate(ImageIn, x0)`, với x_0 là vector hằng. Sử dụng phương pháp nội suy lân cận gần nhất để thực hiện biến đổi.

✎ **Bài tập 11-9.**

Viết hàm MATLAB thực hiện phép biến đổi được định nghĩa như sau:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x + x_0 \\ y + y_0 \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \text{ hay } u = A(x + x_0) + x_1$$

với $u = [u \ v]^T$, $x = [x \ y]^T$ là các tọa độ trước và sau khi biến đổi, $x_0 = [x_0 \ y_0]^T$, $x_1 = [x_1 \ y_1]^T$ là các vector hằng, A là ma trận biến đổi kích thước 2x2.

Định nghĩa hàm này dưới dạng `ImageOut = transform(ImageIn, A, x0, x1)`, với x_0, x_1 là các vector hằng và A là ma trận biến đổi thuận. Sử dụng phương pháp nội suy lân cận gần nhất để thực hiện biến đổi.

✎ **Bài tập 11-10.**

Sử dụng hàm đã viết ở bài tập 11-9 để biến đổi ảnh **cameraman.tif** với các thông số biến đổi như sau:

$$\begin{aligned} \text{a. } A &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, x_0 = 0 \text{ và } x_1 = \begin{bmatrix} N-1 \\ 0 \end{bmatrix} & \text{b. } A &= \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, x_0 = 0 \text{ và } x_1 = \begin{bmatrix} 0 \\ N-1 \end{bmatrix} \\ \text{c. } A &= \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}, x_0 = 0 \text{ và } x_1 = \begin{bmatrix} N-1 \\ N-1 \end{bmatrix} & \text{d. } A &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}, x_0 = \begin{bmatrix} -\frac{1}{2}N \\ -\frac{1}{2}N \end{bmatrix} \text{ và} \\ & & & x_1 = \begin{bmatrix} \frac{1}{2}N \\ \frac{1}{2}N \end{bmatrix} \end{aligned}$$

(NxN là kích thước của ảnh)

✎ **Bài tập 11-11.**

Chọn 2 ảnh trong máy tính và thực hiện các bước sau:

- i. Thực hiện biến đổi DCT
- ii. Hiển thị phổ DCT của ảnh
- iii. Dời phần phổ DCT về trung tâm. Hiển thị phổ mới.
- iv. Chỉ giữ lại 10% số hệ số DCT, còn lại cho bằng 0. Thực hiện biến đổi DCT ngược. Hiển thị ảnh thu được.

Sắp xếp tất cả các kết quả hiển thị trong một figure.

✎ **Bài tập 11-12.**

Ảnh **mri.tif** là một ảnh giả lập kết quả chụp từ thiết bị X quang. Hãy thực hiện biến đổi Radon đối với ảnh này để tạo tập dữ liệu hình chiếu, sau đó khôi phục ảnh từ tập dữ liệu này. Xét các trường hợp mật độ của các tia chiếu như sau:

- a. Các tia cách nhau 2 pixels
- b. Các tia cách nhau 1 pixels
- c. Các tia cách nhau 0.5 pixels

✂ ***Bài tập 11-13.***

Làm lại bài tập 11-12 đối với ảnh **spine.tif** và dùng phép biến đổi fan-beam.

Danh sách các hàm được giới thiệu trong chương 11

Các hàm chuyển đổi loại ảnh và kiểu dữ liệu ảnh

dither	Tạo ảnh nhị phân từ ảnh trắng đen hoặc ảnh RGB
gray2ind	Chuyển ảnh trắng đen (gray scale) thành ảnh indexed
grayslice	Chuyển ảnh trắng đen (gray scale) thành ảnh indexed bằng cách lấy ngưỡng
im2bw	Chuyển các loại ảnh thành ảnh nhị phân
im2double	Chuyển kiểu dữ liệu ảnh thành double
im2uint16	Chuyển kiểu dữ liệu ảnh thành uint16
im2uint8	Chuyển kiểu dữ liệu ảnh thành uint8
imapprox	Xấp xỉ ảnh indexed bằng cách giảm số màu
ind2gray	Chuyển ảnh indexed thành ảnh gray scale
ind2rgb	Chuyển ảnh indexed thành ảnh RGB
mat2gray	Tạo ảnh gray scale từ ma trận
rgb2gray	Chuyển ảnh RGB thành ảnh gray scale
rgb2ind	Chuyển ảnh RGB thành ảnh indexed

Các hàm truy xuất dữ liệu ảnh

imfinfo	Truy xuất các thông tin về ảnh
imread	Đọc ảnh từ file và xuất ra ma trận ảnh
imwrite	Lưu ma trận ảnh thành file ảnh

Các hàm thực hiện biến đổi số học đối với ảnh

imabsdiff	Lấy giá trị tuyệt đối của hiệu hai ảnh
imadd	Cộng hai ảnh hoặc cộng ảnh với hằng số
imcomplement	Lấy phần bù của ảnh
imdivide	Chia hai ảnh hoặc chia một ảnh cho một hằng số
imlincomb	Lấy tổ hợp tuyến tính của các ảnh
immultiply	Nhân hai ảnh hoặc nhân ảnh với một hằng số
imsubtract	Trừ hai ảnh hoặc trừ ảnh đi một hằng số

Các hàm hiển thị ảnh

image	Hiển thị ảnh từ một ma trận
imagesc	Giống hàm image nhưng có thể co giãn dữ liệu ảnh
imshow	Hiển thị ảnh trên một figure
imview	Hiển thị ảnh trên một cửa sổ nền Java

Các hàm biến đổi hình học

cp2tform	Định nghĩa phép biến đổi hình học từ các cặp điểm tương ứng
-----------------	---

imcrop	Trích xuất một phần ảnh
imresize	Thay đổi kích thước ảnh
imrotate	Thực hiện phép quay ảnh
imtransform	Thực hiện phép biến đổi hình học tổng quát
maketform	Định nghĩa phép biến đổi hình học tổng quát

Các hàm thực hiện các phép biến đổi ảnh

dct2	Biến đổi DCT thuận
dctmtx	Ma trận biến đổi DCT
fanbeam	Biến đổi Fan-beam
fft2	Biến đổi FFT thuận
fftshift	Biến đổi FFT với thành phần DC dịch về trung tâm
idct2	Biến đổi DCT nghịch
ifanbeam	Biến đổi Fan-beam nghịch
ifft2	Biến đổi FFT nghịch
iradon	Biến đổi Radon nghịch

Chương 12

NÂNG CAO CHẤT LƯỢNG ẢNH

Một ứng dụng quan trọng và mang tính thực tiễn cao của kỹ thuật xử lý ảnh số là nâng cao chất lượng ảnh. Cụ thể là các kỹ thuật khôi phục lại ảnh gốc từ ảnh bị nhiễu khi truyền trong hệ thống, làm rõ nét những ảnh chụp trong những điều kiện khó khăn, chẳng hạn ảnh chụp các hành tinh khác hoặc các ảnh chụp từ vệ tinh, hoặc phục vụ cho công tác điều tra của cảnh sát, ... Nói chung, các kỹ thuật nâng cao chất lượng ảnh là các kỹ thuật cải tiến chất lượng ảnh thể hiện thông qua các thông số đánh giá chất lượng khách quan (ví dụ tỷ số tín hiệu trên nhiễu SNR, sai số bình phương trung bình MSE, ...) và đôi khi là cả các đánh giá chủ quan (ví dụ làm rõ nét một số đặc tính của ảnh, ...). Các phương pháp chính được dùng để cải thiện chất lượng ảnh bao gồm: phương pháp biến đổi mức xám (hay độ sáng) của ảnh, phương pháp cân bằng histogram và các phương pháp lọc nhiễu ảnh.

12.1. PHƯƠNG PHÁP BIẾN ĐỔI MỨC XÁM

Phương pháp biến đổi mức xám là một kỹ thuật nâng cao chất lượng ảnh trong đó tập các giá trị mức xám của ảnh được ánh xạ vào một miền giá trị mới. Quá trình này được định nghĩa bởi biểu thức:

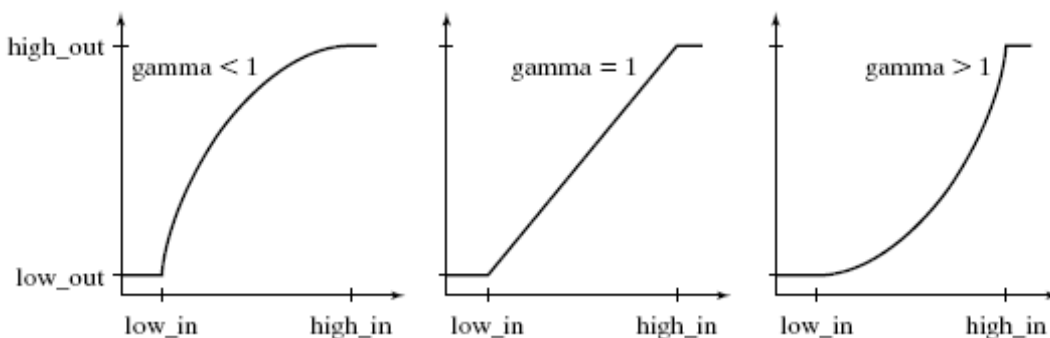
$$g(x,y) = T[f(x,y)] \quad (12.1)$$

trong đó $f(x,y)$ là ảnh ban đầu, $g(x,y)$ là ảnh sau khi biến đổi còn T là một toán tử trên f , được định nghĩa bởi một hàm của các giá trị độ sáng ở các điểm lân cận của điểm (x,y) đang xét. Các điểm lân cận là các điểm nằm trong 1 hình vuông có tâm tại điểm (x,y) . Trong trường hợp đơn giản nhất, hình vuông này có kích thước 1×1 .

Hàm **imadjust** là một hàm cơ bản trong Image Processing Toolbox của MATLAB dùng để biến đổi mức xám của ảnh với cú pháp như sau:

```
>> J=imadjust(I,[low_in; high_in],[low_out; high_out],gamma)% gray-scale
>> newmap=imadjust(map,[low_in; high_in],[low_out; high_out],gamma) %
indexed
>> RGB2=imadjust(RGB,[low_in; high_in],[low_out; high_out],gamma)% RGB
```

Hàm **imadjust** biến đổi các giá trị mức xám nằm trong khoảng $[low_in, high_in]$ thành các giá trị nằm trong khoảng $[low_out, high_out]$ theo một quy luật được định nghĩa tùy theo giá trị $gamma$ như minh họa ở hình 12.1.



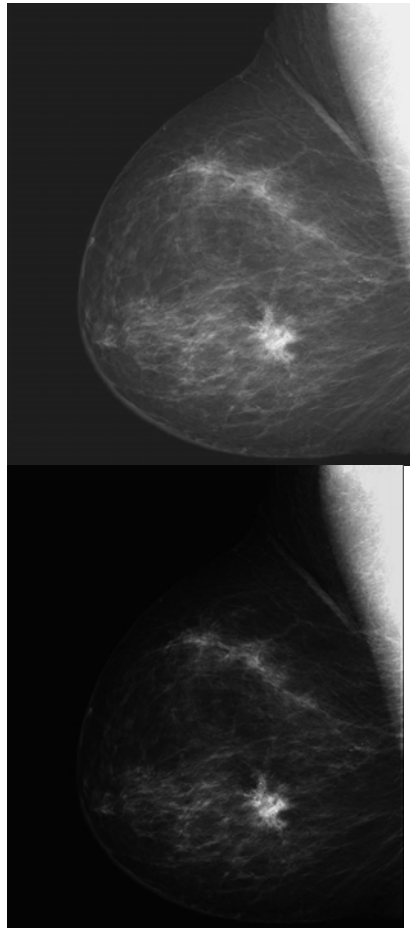
Hình 12.1. Biến đổi mức xám bằng hàm imadjust

Ảnh ban đầu và ảnh biến đổi thuộc cùng kiểu dữ liệu (`uint8`, `uint16`, hoặc `double`). Các thông số giới hạn được nhập cho hàm này đều nằm trong khoảng $[0,1]$, MATLAB sẽ tự chuyển đổi thành giá trị thích hợp tùy theo kiểu dữ liệu của ảnh (ví dụ với kiểu `uint8`, $[0\ 1]$ sẽ trở thành $[0\ 255]$, với `uint16`, $[0\ 1]$ trở thành $[0\ 65535]$). Đối với ảnh indexed, hàm **imadjust** thực hiện biến đổi ma trận màu của ảnh thay vì biến đổi trực tiếp trên ảnh.

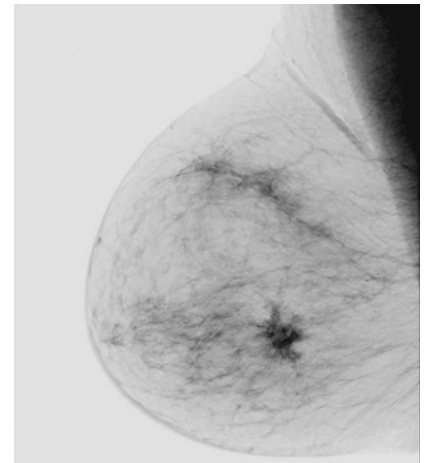
 **Ví dụ 12-1.** Tạo ảnh âm bản và thay đổi độ tương phản bằng hàm **imadjust**:

Để làm rõ những bức ảnh có nhiều vùng tối, người ta thường chuyển sang dạng âm bản. Ngoài ra, đôi khi cần phải làm nổi bật những vùng có độ sáng nằm trong một khoảng xác định nào đó. Tất cả các trường hợp trên đều có thể thực hiện bằng hàm **imadjust**. Trong ví dụ này, chúng ta sẽ xem xét quá trình này đối với ảnh chụp một mẫu mô ngực.

```
f = imread('tissue.bmp');
imshow(f)
g = imadjust(f, [0 1] , [1 0]);
figure, imshow(g)
g1 = imadjust(f, [0.5 0.75], [0 1]);
figure, imshow(g1)
```



a) Ảnh gốc
xám



b) Ảnh âm bản

c) Ảnh sau khi biến đổi mức

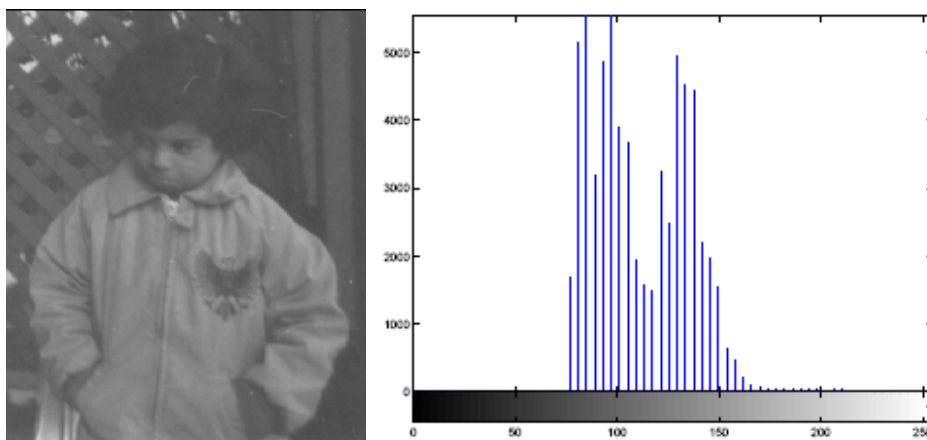
Hình 12.2.

Thông thường, trước khi thực hiện biến đổi mức xám ta cần thực hiện hai bước:

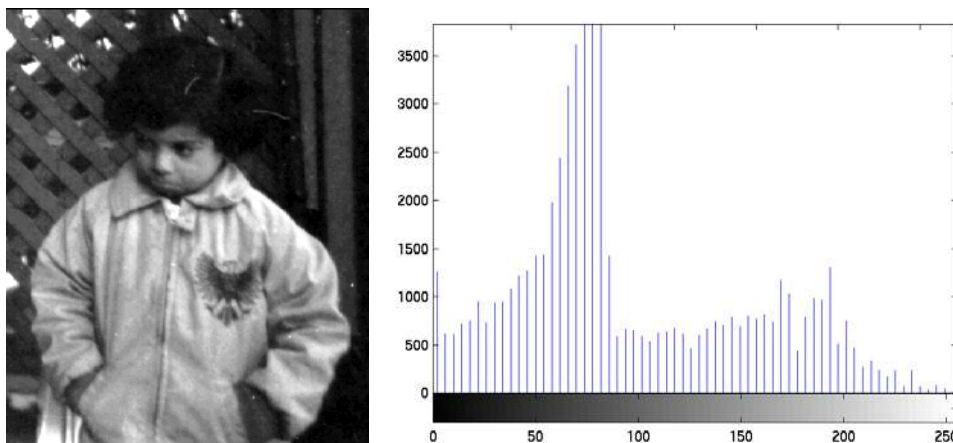
- Xem biểu đồ histogram của ảnh để biết được các giới hạn mức xám của ảnh. Histogram là một biểu đồ cột biểu thị tần số xuất hiện của các mức xám khác nhau có trong ảnh. Trong MATLAB, hàm `imhist` cho phép hiển thị biểu đồ histogram của các dạng ảnh gray-scale, index và RGB.
- Xác định các giới hạn mức xám của ảnh và chuyển sang dạng phân số trong khoảng $[0 \ 1]$ để nhập cho hàm `imadjust`.

 **Ví dụ 12-2.** Sử dụng hàm `imhist` và hàm `imadjust` để thay đổi độ tương phản của ảnh:

```
I = imread('pout.tif'); % Đọc ảnh gốc
imshow(I)                % Hiển thị ảnh gốc
figure, imhist(I,64)      % Xem biểu đồ histogram
J = imadjust(I,[0.3 0.65],[0 1],1); % Biến đổi mức xám
imshow(J)                % Hiển thị ảnh sau khi biến đổi
figure, imhist(J,64)      % Xem biểu đồ histogram
```



a) Ảnh gốc và biểu đồ histogram



b) Ảnh sau biến đổi và biểu đồ histogram

Hình 12.3.

Tuy nhiên, để thuận tiện cho người sử dụng, MATLAB cung cấp hàm `stretchlim`. Hàm này tính toán histogram của ảnh và xác định các giá trị giới hạn của mức xám một cách tự động. Nó trả về một vector mà ta có thể dùng làm cặp thông số `[low_in;high_in]` cung cấp cho hàm `imadjust` (trong trường hợp mặc định, hàm `imadjust` sử dụng hàm `stretchlim` để tạo ra cặp `[low_in;high_in]` nếu người sử dụng không cung cấp). Bình thường hàm `stretchlim`

sẽ lấy các giá trị mức xám nằm ở mức 1% và 99% trong vùng biến thiên mức xám của ảnh. Tuy nhiên, ta cũng có thể thay đổi mở rộng hoặc thu hẹp các giới hạn này bằng cách cung cấp thêm thông số *tol* cho hàm **stretchlim**. Đó là một vector gồm hai phần tử thuộc $[0,1]$ cho biết ta sẽ chọn các giới hạn ở mức nào trong vùng biến thiên mức xám (mặc định là $[0.01\ 0.99]$).

```
>> [low,high] = stretchlim(I,tol)
```

Ngoài phương pháp sử dụng hàm **imadjust** nêu trên, người sử dụng có thể tự định nghĩa các toán tử biến đổi mức xám và áp dụng vào các ảnh cụ thể. Dưới đây chúng tôi trình bày hai phép biến đổi mức xám không dùng hàm **imadjust**, đó là phép biến đổi logarithm và phép mở rộng độ tương phản.

Phép biến đổi logarithm được thực hiện dựa trên biểu thức sau:

```
>> g = c*log(1 + double(f))
```

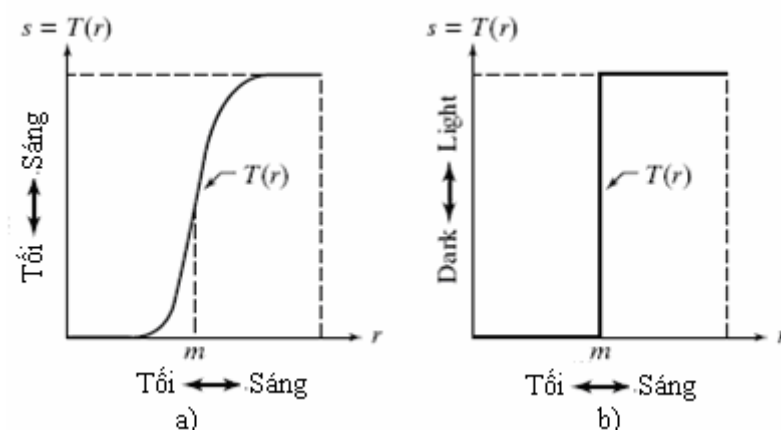
trong đó *c* là một hằng số. Dạng biến đổi này gần giống với biến đổi trong hình 12.1a với các giá trị giới hạn trên cả hai trục đều là 0 và 1. Tuy nhiên, nếu ở hình 12.1a đồ thị thay đổi tùy theo *gamma* thì ở đây hàm log là cố định. Một ứng dụng rất quan trọng của biến đổi logarithm là nén dải động. Ví dụ, trong chương trước ta thấy rằng các giá trị của phổ Fourier của ảnh thường thay đổi rất rộng (khoảng $[0\ 10^6]$). Bằng phép biến đổi log, dải động 10^6 của phổ này sẽ giảm xuống còn khoảng 14, giúp cho việc quan sát dễ dàng hơn. Sau khi thực hiện biến đổi, ta chuyển kết quả về kiểu dữ liệu tương ứng với ảnh gốc, thí dụ với ảnh `uint8`, ta có thể dùng dòng lệnh sau:

```
>> gs = im2uint8(mat2gray(g));
```

Phép mở rộng độ tương phản được định nghĩa bằng biểu thức sau và được minh họa trên hình 12.4a:

$$s = T(r) = \frac{1}{1 + (m/r)^E} \quad (12.2)$$

Trong đó *r* là giá trị mức xám ban đầu, *s* là giá trị mức xám sau khi biến đổi, *m* là mức ngưỡng và *E* là một hằng số quy định độ dốc của hàm. Hàm này nén các giá trị mức xám dưới *m* vào thành một dải hẹp trong vùng tối và nén các giá trị trên *m* thành một dải hẹp trong vùng sáng, do đó làm tăng độ tương phản của ảnh.



Hình 12.4.

Biến đổi này được thực hiện bằng dòng lệnh MATLAB sau:

```
>> g = 1./(1 + (m./(double(f) + eps)).^E)
```

Hằng số *eps* được thêm vào để tránh trường hợp *f* có phần tử bằng 0.

 **Ví dụ 12-3.** Làm lại ví dụ 12-2 không sử dụng hàm *imadjust*:

```
I = imread('pout.tif'); % Đọc ảnh gốc
imshow(I) % Hiển thị ảnh gốc
J = 1./(1 + (0.5./(im2double(I) + eps)).^5); % Tăng độ tương phản
figure, imshow(J) % Hiển thị ảnh sau biến đổi
```



a) Ảnh gốc



b) Ảnh sau khi biến đổi

Hình 12.5.

12.2. CÂN BẰNG HISTOGRAM

12.2.1. TẠO VÀ VẼ BIỂU ĐỒ HISTOGRAM

Histogram của một ảnh số nhận L giá trị mức xám khác nhau trong khoảng $[0, G]$ là một hàm rời rạc được định nghĩa bởi:

$$h(r_k) = n_k \quad (12.3)$$

với r_k là giá trị mức xám thứ k còn n_k là số pixel của ảnh có mức xám bằng r_k . Giá trị của G phụ thuộc vào kiểu dữ liệu của ảnh: $G = 1$ đối với kiểu double, $G = 255$ với kiểu uint8, $G = 65535$ với kiểu uint16. Thông thường, người ta cũng định nghĩa histogram dưới dạng chuẩn hoá, đó là tỷ số của n_k so với tổng số pixel của ảnh:

$$p(r_k) = \frac{h(r_k)}{n} = \frac{n_k}{n} \quad (12.4)$$

với $k = 1, 2, \dots, L$.

Hàm cơ bản nhất dùng để tính histogram của ảnh trong MATLAB là hàm `imhist` với cú pháp như sau:

```
>> [h, r] = imhist(I, N)
```

trong đó I là ảnh đầu vào. Các giá trị mức xám trong khoảng $[0, G]$ được chia thành N khoảng (gọi là bin), vị trí của các bin được xác định bởi vector r và h là vector histogram, nghĩa là số các pixel có giá trị mức xám nằm trong từng bin. Nếu gọi hàm **imhist** mà không cần trả về các thông số h, r thì MATLAB sẽ tạo một figure và vẽ biểu đồ histogram của ảnh.

Ngoài cách vẽ trực tiếp bằng cách dùng hàm `imhist` không cần thông số trả về như trên, ta có thể tính vector histogram của ảnh theo cú pháp trên, sau đó dùng các hàm **bar**, **stem**, hoặc **plot** để vẽ biểu đồ histogram.

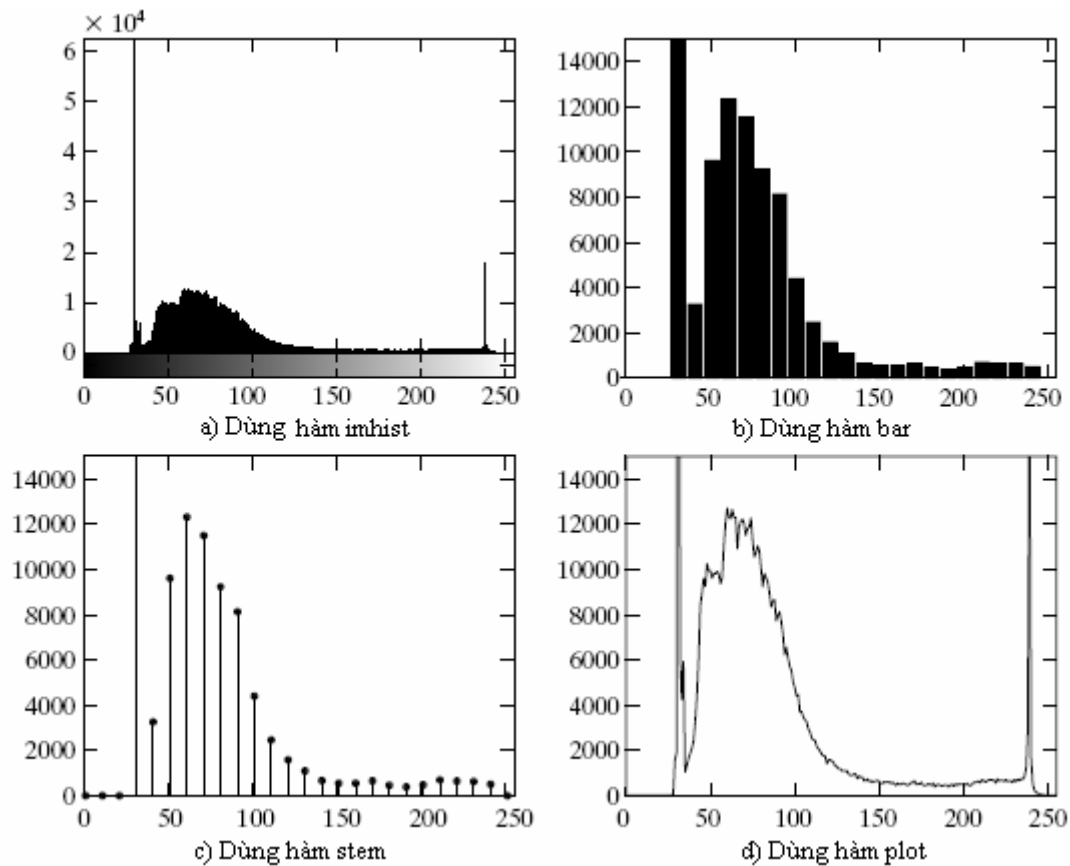
Ví dụ 12-4. Vẽ biểu đồ histogram bằng các hàm *imhist*, *bar*, *stem* và *plot*:

Ví dụ này minh họa các biểu đồ histogram của ảnh *f* thu được khi vẽ bằng các cách khác nhau. Các kết quả được trình bày ở hình 12.6.

```
>> [h,r] = imhist(f);
>> h1 = h(1:10:256);
>> imhist(f) % Cách 1: dùng imhist
>> horz = 1:10:256;
>> bar(horz, h1) % Cách 2: dùng bar
>> stem(r,h) % Cách 3: dùng stem
>> plot(r,h) % Cách 4: dùng plot
```

Các dòng lệnh sau dùng để điều chỉnh các giới hạn của các trục tọa độ được hiển thị:

```
>> axis([0 255 0 15000])
>> set(gca, 'xtick', 0:50:255)
>> set(gca, 'ytick', 0:2000:15000)
```



Hình 12.6.

12.2.2.CÂN BẰNG HISTOGRAM

Giả sử rằng các giá trị mức xám của ảnh là đại lượng liên tục trong khoảng $[0,1]$, và gọi $p_r(r)$ là hàm mật độ xác suất của các giá trị mức xám của một ảnh cho trước. Giả sử ta thực hiện phép biến đổi mức xám sau đây để được các giá trị mức xám mới *s*:

$$s = T(r) = \int_0^r p_r(w)dw \quad (12.5)$$

Khi đó, có thể chứng minh rằng phân bố xác suất của ảnh mới sẽ là phân bố đều trên $[0,1]$:

$$p_s(s) = \begin{cases} 1 & 0 \leq s \leq 1 \\ 0 & \text{nơi khác} \end{cases} \quad (12.6)$$

Như vậy, phép biến đổi này làm cho các giá trị mức xám của ảnh trở nên có xác suất xuất hiện phân bố đều trên toàn khoảng $[0,1]$, nghĩa là dải động của ảnh sẽ lớn hơn và độ tương phản cao hơn. Quá trình biến đổi này gọi là quá trình cân bằng ảnh. Khi làm việc với các giá trị mức xám rời rạc như trong các ảnh số thì hàm mật độ xác suất sẽ trở thành histogram của ảnh và quá trình biến đổi nói trên được gọi là quá trình cân bằng histogram. Gọi $p_r(r_j)$ với $j = 1, 2, \dots, L$ là các giá trị histogram chuẩn hoá của ảnh thì quá trình cân bằng histogram được biểu diễn bởi phương trình sau:

$$s_k = T(r_k) = \sum_{j=1}^k p_r(r_j) = \sum_{j=1}^k \frac{n_j}{n} \quad (12.7)$$

trong đó $k = 1, 2, \dots, L$ và s_k là các giá trị mức xám của ảnh.

Quá trình cân bằng histogram được thực hiện bởi hàm **histeq** với các cú pháp như sau:

```
>> g = histeq(f, nlev)
```

Trong đó, **nlev** xác định số giá trị mức xám rời rạc trong ảnh mới. Nếu **nlev** = **L** thì hàm **histeq** thực hiện quá trình cân bằng $T(r_k)$ một cách trực tiếp. Nếu **nlev** < **L** thì hàm này tìm cách phân bố các mức xám để tạo một histogram phẳng. Giá trị mặc định của **nlev** là 64. Thông thường ta chọn **nlev** bằng số mức xám tối đa của ảnh.

 **Ví dụ 12-5.** Dùng hàm **histeq** để cân bằng histogram cho ảnh:

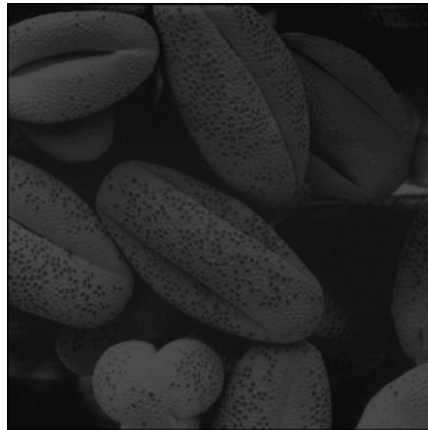
Ví dụ sau đây minh hoạ cách sử dụng hàm **histeq** để cân bằng histogram cho một ảnh gray scale được biểu diễn bởi ma trận **f**. Các hình 12.7a và 12.7b là ảnh gốc cùng với istogram của nó, hình 12.7c và **d** là ảnh và histogram sau khi cân bằng. Chất lượng ảnh được cải thiện rõ rệt.

```
>> imshow(f)
>> figure, imhist(f)
>> ylim('auto')
>> g = histeq(f, 256);
>> figure, imshow(g)
>> figure, imhist(g)
>> ylim('auto')
```

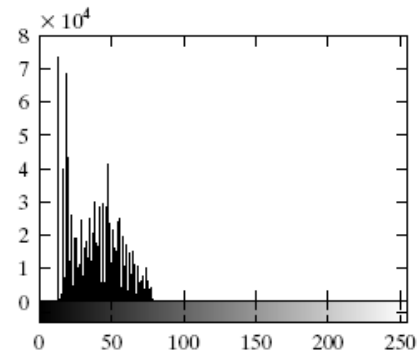
Hàm truyền đạt của quá trình cân bằng chính là hàm xác suất tích lũy của các mức xám của ảnh. Bạn đọc có thể vẽ lại ảnh này bằng cách dùng các lệnh dưới đây:

```
>> hnrm = imhist(f)./numel(f); % Histogram chuẩn hoá
>> cdf = cumsum(hnrm); % Hàm truyền đạt (tích lũy)
>> x = linspace(0, 1, 256);
>> plot(x, cdf) % Vẽ cdf theo x.
```

```
>> axis([0 1 0 1])           % Đặt thông số cho các trục tọa độ
>> set(gca, 'xtick', 0:.2:1)
>> set(gca, 'ytick', 0:.2:1)
>> xlabel('Input intensity values', 'fontsize', 9)
>> ylabel('Output intensity values', 'fontsize', 9)
>> text(0.18, 0.5, 'Transformation function', 'fontsize', 9)
```



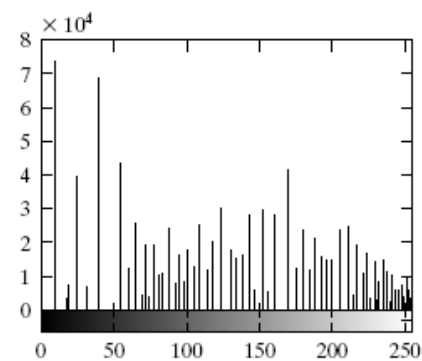
a)



b)

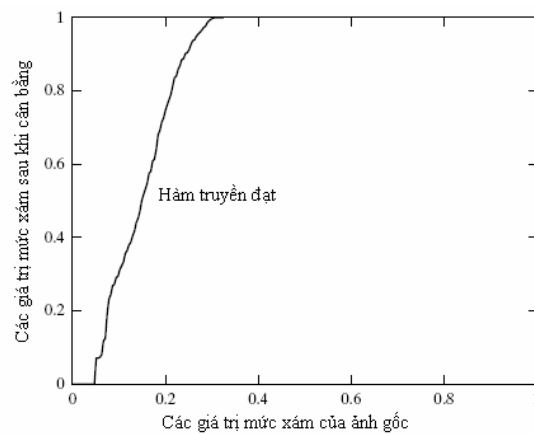


c)



d)

Hình 12.7.



Hình 12.8.

Hàm truyền đạt được vẽ ở hình 12.8.

▪ Hàm truyền đạt của quá trình cân bằng histogram là một hàm được tạo ra tùy theo histogram của ảnh. Nói cách khác, đây là một hàm có tính thích nghi đối với ảnh đầu vào. Tuy nhiên, với một bức ảnh cụ thể thì hàm truyền đạt này không thể thay đổi. Mặt khác, việc mở rộng phạm vi biến thiên của các giá trị mức xám không phải lúc nào cũng mang lại chất lượng cao hơn. Trong một số trường hợp, ta cần biến đổi các mức xám của ảnh sao cho histogram của ảnh mới có một dạng cụ thể định trước. Quá trình này gọi là quá trình phối hợp (hay xác định) histogram.

Nguyên tắc của quá trình này khá đơn giản. Giả sử ảnh có độ xám là đại lượng liên tục trên $[0,1]$. Gọi r, z lần lượt là giá trị độ xám của ảnh gốc và ảnh mới, $p_r(r)$ và $p_z(z)$ là các hàm mật độ xác suất tương ứng của chúng. Ở phần trên, ta thấy phép biến đổi

$s = T(r) = \int_0^r p_r(w)dw$ sẽ trả về mức xám s có mật độ xác suất phân bố đều. Ta định nghĩa một biến z có tính chất như sau:

$$H(z) = \int_0^z p_z(w)dw = s \quad (12.8)$$

z chính là độ sáng của ảnh có hàm mật độ xác suất là $p_z(z)$. Từ hai phương trình trên suy ra:

$$z = H^{-1}(s) = H^{-1}(T(r)) \quad (12.9)$$

Vậy để xác định hàm truyền đạt của quá trình phối hợp, cần xác định H^{-1} . Với các dữ liệu ảnh rời rạc, việc này luôn thực hiện được nếu $p_z(z_k)$ đúng là histogram của 1 ảnh nào đó (nghĩa là có diện tích bằng 1 và các giá trị mức xám đều không âm).

Trong MATLAB, quá trình phối hợp histogram cũng được thực hiện bằng hàm `histeq` nhưng với cú pháp như sau:

```
>> g = histeq(f,hgram)
>> newmap = histeq(X,map,hgram)
```

Dải các giá trị mức xám $[0,G]$ được chia thành `length(hgram)` khoảng đều nhau, `hgram` là một vector gồm các số nguyên đếm số giá trị mức xám trong mỗi khoảng chia. Giá trị mặc định của `hgram` là `hgram = ones(1,n)*prod(size(A))/n`. Cú pháp thứ hai dùng cho ảnh index, hàm `histeq` thực hiện cân bằng trên ma trận màu của ảnh.

 **Ví dụ 12-6.** Dùng hàm *histeq* để phối hợp histogram cho ảnh:

Ví dụ dưới đây cho thấy với các ảnh có nhiều phần tối (hình 12.9a và b), quá trình cân bằng histogram không mang lại kết quả rõ rệt (hình 12.9c và d) nhưng nếu dùng quá trình phối hợp histogram sao cho histogram của ảnh mới có dạng bimodal Gaussian thì hiệu quả sẽ tốt hơn (hình 12.9e và f). Dạng histogram mong muốn được vẽ ở hình 12.9g.

```
imshow(f) % Ảnh gốc
figure, imhist(f) % và histogram tương ứng
f1 = histeq(f,256)
figure, imshow(f1) % Ảnh sau khi cân bằng histogram
figure, imhist(f1) % và histogram tương ứng
%% Thiết lập dạng histogram mong muốn (bimodal Gaussian)
```

```
M1 = 0.15, SIG1 = 0.05, M2 = 0.75, SIG2 = 0.05, A1 = 1, A2 = 0.07, K = 0.002
```

```
c1 = A1 * (1 / ((2 * pi) ^ 0.5) * sig1);
```

```
k1 = 2 * (sig1 ^ 2);
```

```
c2 = A2 * (1 / ((2 * pi) ^ 0.5) * sig2);
```

```
k2 = 2 * (sig2 ^ 2);
```

```
z = linspace(0, 1, 256);
```

```
p = k + c1 * exp(-((z - m1) .^ 2) ./ k1) + ...
```

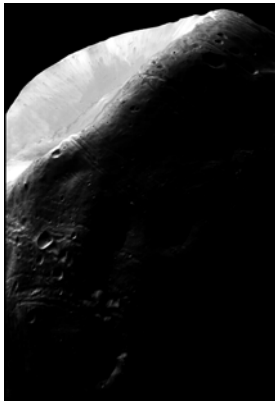
```
c2 * exp(-((z - m2) .^ 2) ./ k2);
```

```
p = p ./ sum(p(:)); % p: dạng histogram mong muốn
```

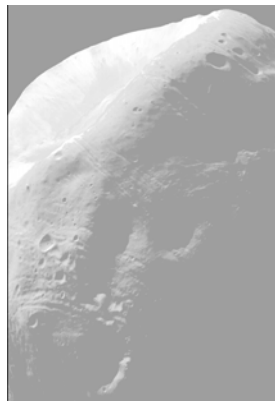
```
g = histeq(f, p)
```

```
figure, imshow(g) % Ảnh sau khi phối hợp histogram
```

```
figure, imhist(g) % và histogram tương ứng
```



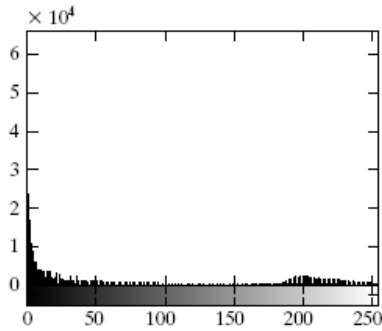
a)



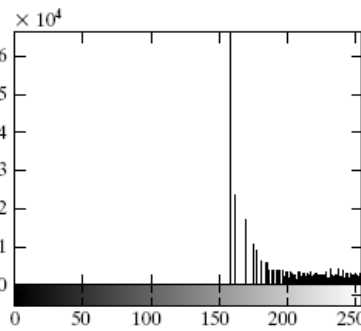
c)



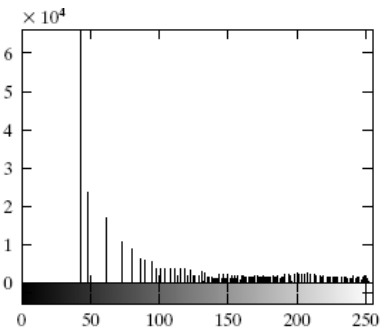
e)



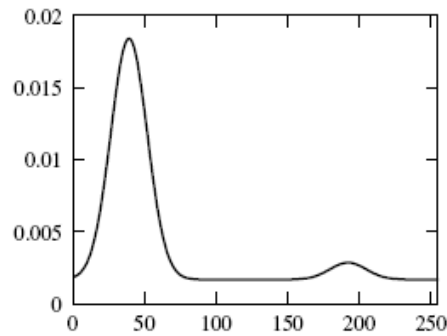
b)



d)



f)



Hình 12.9.

Ngoài hàm **histeq**, ta cũng có thể thực hiện cân bằng histogram bằng hàm **adapthisteq**. Quá trình này gọi là cân bằng histogram thích nghi với độ tương phản bị giới hạn (CLAHE – Contrast-Limited Adaptive Histogram Equalization). Hàm **histeq** tác động lên toàn bộ ảnh bởi một hàm biến đổi mức xám, còn hàm **adapthisteq** sẽ phân chia ảnh thành các vùng nhỏ (gọi là tile) và tác động riêng trên mỗi vùng, sau đó sẽ kết hợp giữa các tile lân cận theo phương pháp nội suy song tuyến tính để loại bỏ các chi tiết lạ xuất hiện ở biên giới các tile.

```
>> J = adapthisteq(I,param1, val1,param2, val2...)
```

Nhập liệu cho hàm **adapthisteq** bao gồm ảnh I và giá trị của các thông số liên quan (nếu cần). Các thông số này bao gồm: 'NumTiles' (mặc định 8x8, là số tile trên hàng ngang và trên một cột dọc), 'ClipLimit' (giới hạn độ tương phản, mặc định 0.01), 'Nbins' (số bins, mặc định 256), 'Range' (chuỗi xác định các giới hạn mức xám của ảnh, mặc định: 'full'), 'Distribution' (chuỗi xác định kiểu phân bố, gồm 'uniform', 'rayleigh',.. mặc định 'uniform'), 'Alpha' (hằng số cho phân bố rayleigh hoặc phân bố hàm mũ, mặc định 0.4).

12.3. LỌC ẢNH

Nhiều là một vấn đề thường gặp đối với các ảnh số. Nhiều có thể hình thành do nhiều nguyên nhân khác nhau. Ví dụ nếu ảnh được scan từ một ảnh chụp sử dụng film thì nhiễu có thể phát sinh bởi các hạt nhỏ trên film hoặc từ thiết bị scan, nếu ảnh được chụp từ các máy chụp ảnh số thì cơ chế thu thập dữ liệu (chẳng hạn bộ phát hiện CCD) sẽ tạo ra nhiễu. Ngoài ra khi truyền ảnh trong các hệ thống viễn thông, ảnh cũng có thể bị nhiễu do kênh truyền gây ra. Để khắc phục ảnh hưởng của nhiễu, ta phải sử dụng các phương pháp lọc ảnh khác nhau. Mỗi phương pháp thích hợp với một số loại nhiễu nhất định.

Để mô phỏng ảnh hưởng của nhiễu lên ảnh, MATLAB cung cấp hàm **imnoise** cho phép tạo ra các loại nhiễu khác nhau và cộng vào ảnh gốc. Các cú pháp của hàm **imnoise** được giới thiệu ở bảng 12-1.

Bảng 12-1. Các cú pháp của hàm **imnoise**

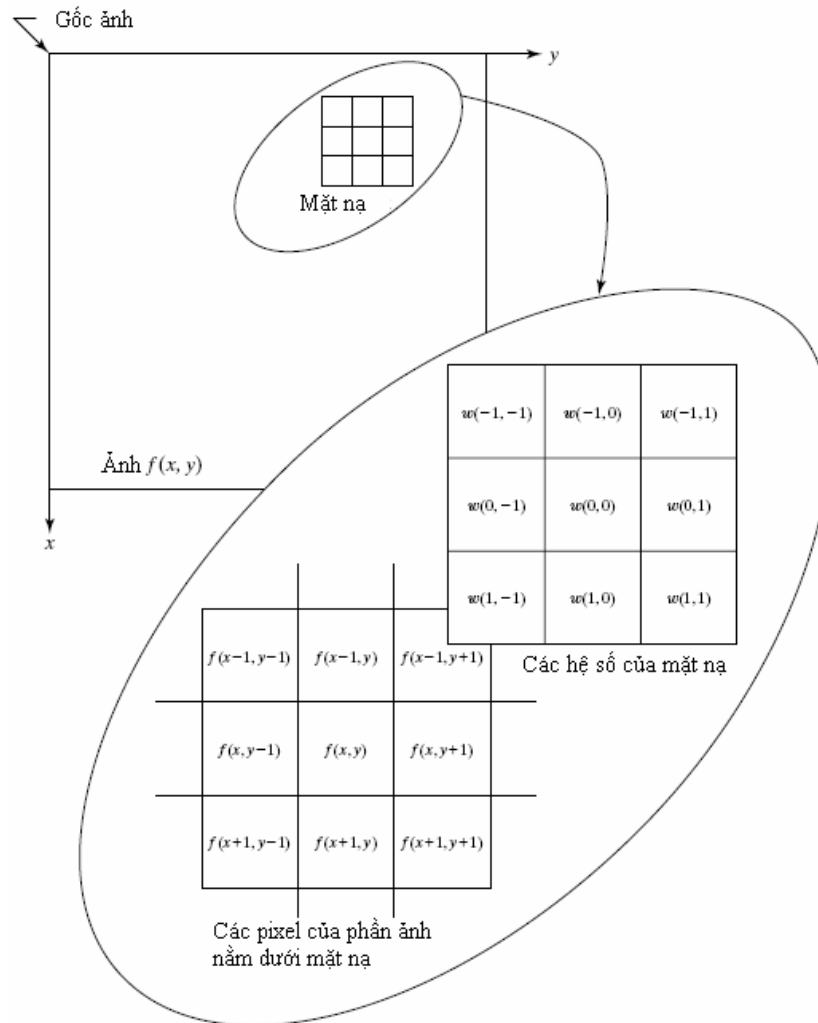
Cú pháp	Mô tả
<code>J = imnoise(I, 'gaussian', m, v)</code>	Tạo nhiễu trắng Gauss có trung bình bằng m và variance
<code>J = imnoise(I, 'localvar', v)</code>	Tạo nhiễu trắng Gauss có trung bình bằng 0 và các variance khác nhau cho từng pixel. v là vector variance có kích thước bằng số pixel của I
<code>J = imnoise(I, 'localvar', level, v)</code>	Như trên nhưng với vector v được xác định theo một hàm của các mức xám. level là vector gồm các giá trị mức xám thuộc [0,1] và v là vector variance tương ứng.
<code>J = imnoise(I, 'poisson')</code>	Tạo nhiễu Poisson
<code>J = imnoise(I, 'salt & pepper', D)</code>	Tạo nhiễu “salt and pepper” với mật độ là D (mặc định 0.05)
<code>J = imnoise(I, 'speckle', V)</code>	Tạo nhiễu nhân: $J = I + n * I$ với n là nhiễu phân bố đều với trung bình bằng 0 và variance bằng v (mặc định 0.04)

Phương pháp lọc ảnh bao gồm các bước như sau: (1) xác định một điểm trung tâm (x,y); (2) thực hiện các phép toán chỉ liên quan đến một số điểm lân cận với (x,y); (3) kết quả trả về chính là đáp ứng của quá trình lọc tại điểm (x,y); (4) lặp lại quá trình lọc với các điểm (x,y) khác.

Nếu phép toán thực hiện trên các điểm lân cận là tuyến tính thì ta có quá trình lọc tuyến tính (còn gọi là phép chập không gian), ngược lại, ta có quá trình lọc phi tuyến.

12.3.1. LỌC TUYẾN TÍNH

Lọc tuyến tính là phương pháp lọc trong đó mỗi pixel của ảnh mới là tổ hợp tuyến tính của các mức xám của các pixel lân cận với pixel (x,y) của ảnh gốc, nghĩa là mỗi pixel lân cận sẽ được nhân với một hệ số tương ứng nào đó rồi cộng tất cả lại. Nếu vùng lân cận của (x,y) có kích thước $m \times n$, thì sẽ có $m \times n$ hệ số lọc và chúng cũng được sắp thành một ma trận kích thước $m \times n$. Ma trận này có nhiều tên gọi khác nhau: ma trận lọc, mặt nạ, nhân lọc, khuôn lọc hoặc cửa sổ lọc. Cơ chế của quá trình lọc tuyến tính được mô tả ở hình 12.10. Lần lượt di chuyển tâm của mặt nạ w qua các điểm khác nhau của ảnh f . Tại mỗi điểm (x,y) , đáp ứng của bộ lọc là tổng các tích của các hệ số lọc với mức xám của điểm tương ứng của ảnh. Kích thước mặt nạ là các số lẻ.



Hình 12.10.

Lưu ý rằng có thể có hai loại phép toán liên quan đến lọc tuyến tính: phép tính tương quan và phép chập. Với phép tính tương quan, ta chỉ cần di chuyển trực tiếp mặt nạ qua các điểm ảnh như hình 12.10, còn với phép chập, trước khi di chuyển qua các điểm ảnh, ta phải xoay mặt nạ đi 180° .

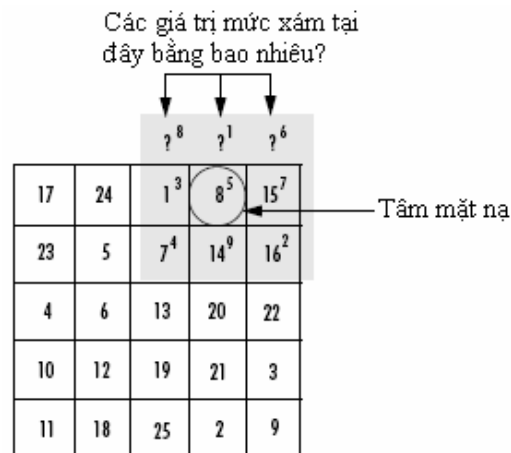
Trong Image Processing Toolbox, phép lọc tuyến tính được thực hiện bằng hàm **imfilter** với cú pháp tổng quát như sau:

```
>> g = imfilter(f, w, filtering_mode, boundary_options, size_options)
```

với f là ảnh ban đầu, w là mặt nạ lọc và g là ảnh sau khi lọc. Các thông số còn lại là các thông số tùy chọn:

'filtering_mode' có thể là 'corr' (thực hiện phép tính tương quan) hoặc 'conv' (phép chập), giá trị mặc định là 'corr'.

Khi tính toán đáp ứng lọc tại các pixel ở biên của ảnh, một phần mặt nạ sẽ nằm ở ngoài biên giới của ảnh, do đó khi tính đáp ứng lọc, ta không biết phải gán giá trị mức xám tại những điểm này bằng bao nhiêu (xem hình 12.11). 'boundary_options' xác định quy tắc chèn thêm các mức xám ở ngoài biên của ảnh trước khi lọc.

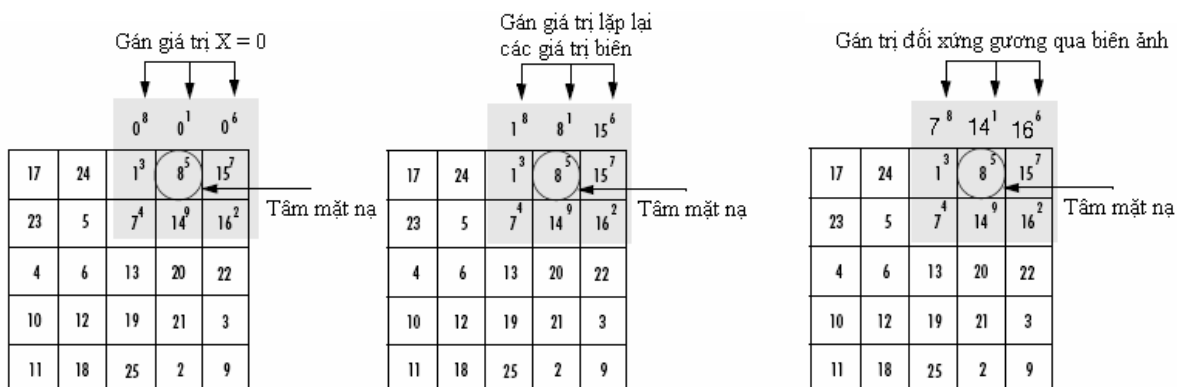


Hình 12.11.

Có bốn quy tắc chèn thêm mức xám ngoài biên, được xác định bởi 4 cách chọn giá trị của 'boundary_options':

- X : X là giá trị mức xám cụ thể được gán cho các điểm ngoài, mặc định là 0.
- 'symmetric' : các mức xám bên ngoài là ảnh đối xứng gương của các điểm ảnh bên trong qua biên của ảnh.
- 'replicate' : các mức xám bên ngoài lặp lại các giá trị mức xám trên biên của ảnh
- 'circular' : các mức xám được tính toán trên cơ sở giả thiết ảnh vào có tính tuần hoàn

Hình vẽ dưới đây (12.12) minh họa các cách chèn thêm biên ảnh:



Hình 12.12.

'size_options' là một chuỗi xác định kích thước ảnh sau khi lọc: giá trị 'full' có nghĩa là toàn bộ kết quả tính toán (kể cả các vị trí được chèn thêm) được đưa vào ảnh mới, như vậy

kích thước sẽ lớn hơn ảnh ban đầu; 'same' có nghĩa là MATLAB chỉ giữ lại phần ảnh có kích thước bằng ảnh ban đầu.

Hàm **imfilter** xử lý các kiểu dữ liệu khác nhau theo cách tương tự với các hàm thực hiện các phép toán số học. Ảnh sau khi lọc có cùng kiểu với ảnh gốc, tuy nhiên khi tính toán hàm **imfilter** tự chuyển sang kiểu số thực dấu chấm động với độ chính xác kép. Nếu kết quả vượt ra ngoài tầm giá trị của kiểu dữ liệu, hàm này sẽ tự động điều chỉnh giá trị cho phù hợp.

 Ví dụ 12-7.

```
>> A=magic(5)
A =
    17     24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> h=[-1 0 1]
h =
    -1     0     1

>> imfilter(A,h)
ans =
    24    -16    -16    14    -8
     5    -16     9     9   -14
     6     9    14     9   -20
    12     9     9   -16   -21
    18    14   -16   -16    -2
```

Nếu trước khi lọc, ta chuyển ảnh sang kiểu `uint8` thì sau khi lọc, các giá trị âm sẽ tự động chuyển về 0.

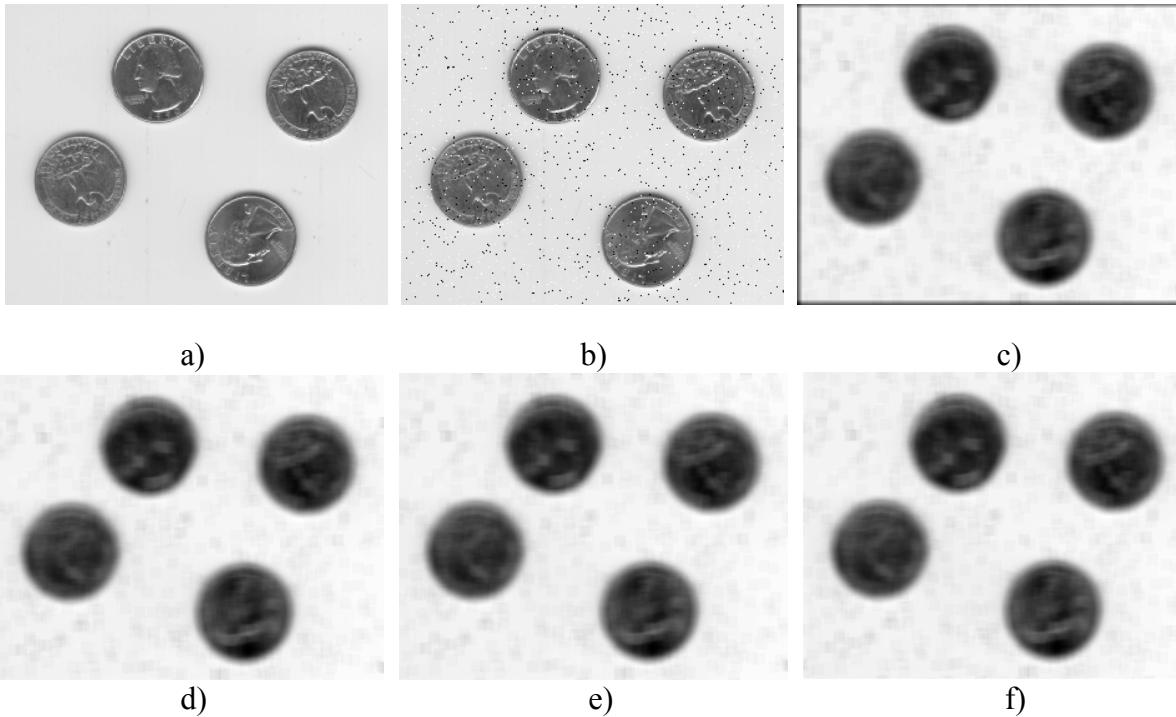
```
>> A = uint8(magic(5));
imfilter(A,h)
ans =
    24     0     0    14     0
     5     0     9     9     0
     6     9    14     9     0
    12     9     9     0     0
    18    14     0     0     0
```

 Ví dụ 12-8. Sử dụng hàm **imfilter** với các cách chèn các điểm ngoài biên khác nhau

Hình 12.13a là một ảnh gốc kích thước 242x308. Ta cộng nhiễu cho ảnh và khảo sát cửa sổ lọc kích thước 9x9.

```
I = imread('eight.tif'); % Đọc ảnh gốc
imshow(I)
J = imnoise(I,'salt & pepper',0.02); % Cộng nhiễu salt&pepper với mật độ 0.02
```

```
figure, imshow(J)
w = ones(9)/81; % Cửa sổ lọc 9x9
gd = imfilter(f, w); % Lọc với thông số mặc định
figure, imshow(gd, [ ])
gr = imfilter(f, w, 'replicate'); % Gán các trị lặp lại biên ảnh
figure, imshow(gr, [ ])
gs = imfilter(f, w, 'symmetric'); % Gán trị đối xứng gương
figure, imshow(gs, [ ])
gc = imfilter(f, w, 'circular'); % Gán theo kiểu 'circular'
figure, imshow(gc, [ ])
```



Hình 12.13.

a) Ảnh gốc b) Ảnh nhiễu c) “zero padding” d) ‘replicate’ e) ‘symmetric’ f) ‘circular’

MATLAB đã định nghĩa trước một số bộ lọc tuyến tính để người sử dụng chọn lựa. Bằng cách gọi hàm **fspecial**, ta nhận được một mặt nạ lọc có thể dùng để cung cấp cho hàm **imfilter**.

Bảng 12-2. Các cú pháp của hàm **fspecial**

Cú pháp	Mô tả
$H = \text{fspecial}('average', N)$	Bộ lọc trung bình với kích thước $N \times N$ (mặc định là 3×3)
$H = \text{fspecial}('disk', r)$	Bộ lọc trung bình dạng tròn (pill box) với bán kính r (nằm trong ma trận vuông kích thước $2*r-1$). Mặc định $r = 5$
$H = \text{fspecial}('gaussian', N, \text{sigma})$	Bộ lọc Gaussian với kích thước $N \times N$, độ lệch chuẩn sigma . Giá trị mặc định là $N = 3, \text{sigma} = 0.5$

<code>H = fspecial('laplacian',alpha)</code>	Bộ lọc Laplacian kích thước 3x3. alpha là thông số nằm trong khoảng [0 1], mặc định là 0.2
<code>H = fspecial('log',N,sigma)</code>	Đối xứng quay của bộ lọc Gaussian. Giá trị mặc định: N = 5, sigma = 0.5
<code>H = fspecial('motion',len,theta)</code>	Bộ lọc xấp xỉ chuyển động tuyến tính của camera khi di chuyển len pixels theo một góc theta. Giá trị mặc định là len=9 và theta = 0
<code>H = fspecial('prewitt')</code>	Bộ lọc Prewitt: $H = \begin{bmatrix} 1 & 1 & 1; 0 & 0 & 0; -1 & -1 & -1 \end{bmatrix}$
<code>H = fspecial('sobel')</code>	Bộ lọc Sobel: $H = \begin{bmatrix} 1 & 2 & 1; 0 & 0 & 0; -1 & -2 & -1 \end{bmatrix}$
<code>H = fspecial('unsharp',alpha)</code>	Bộ lọc tăng độ tương phản. alpha là thông số tương tự như bộ lọc Laplacian

 **Ví dụ 12-9.** Sử dụng hàm **imfilter** với cửa sổ lọc định nghĩa trước bởi hàm **fspecial**:

```
originalRGB = imread('peppers.png');
h = fspecial('motion',30,45);
filteredRGB = imfilter(originalRGB,h);
imshow(originalRGB)
figure,imshow(filteredRGB)
```



a) Ảnh gốc



b) Ảnh lọc xấp xỉ chuyển động của camera

Hình 12.14.

Lưu ý: Hàm **imfilter** có thể sử dụng với các ảnh nhiều chiều, chẳng hạn ảnh RGB.

12.3.2. LỌC PHI TUYẾN

Phương pháp lọc phi tuyến cũng dựa trên nguyên tắc cửa sổ lọc giống như phương pháp lọc tuyến tính, tuy nhiên, thay vì lấy tổng có trọng số của các pixel lân cận thì phương pháp lọc phi tuyến sẽ thực hiện một phép toán phi tuyến trên các điểm lân cận này, ví dụ phép lấy cực đại của các mức xám lân cận là một phép toán phi tuyến.

Quá trình lọc phi tuyến tổng quát được thực hiện bởi hàm **nlfilter** và **colfilt**. Hàm **nlfilter** thực hiện một cách trực tiếp trên cả hai chiều, trong khi hàm **colfilt** thực hiện lọc theo từng cột. Hàm **colfilt** đòi hỏi nhiều bộ nhớ hơn nhưng tốc độ thực thi nhanh hơn hàm **nlfilter**.

```
>> g = nlfilter(f,[m n],@fun,parameters)
>> g = colfilt(f,[m n],'sliding',@fun,parameters)
```

Trong đó, f là ảnh gốc, cửa sổ lọc có kích thước $m \times n$, fun là một hàm phi tuyến đã được định nghĩa trước và `parameters` là các thông số tùy chọn khác (dùng lệnh `help colfilt` hoặc `help nlfilt` để biết thêm về các thông số này).

Cũng như phương pháp lọc tuyến tính, trước khi lọc phi tuyến, ta cần chèn thêm các điểm phụ ở ngoài biên của ảnh bằng cách sử dụng hàm **padarray**:

```
>> fp = padarray(f, [r c], method, direction)
```

Trong đó `[r c]` là số cột và số hàng thêm vào, `method` có thể là `'replicate'`, `'symmetric'` hoặc `'circular'` với ý nghĩa tương tự như hàm **imfilter**, `direction` có thể là `'pre'`, `'post'` hoặc `'both'` cho phép lựa chọn giữa các phương án thêm vào trước phần tử đầu tiên của mỗi chiều, hoặc sau phần tử cuối của mỗi chiều hoặc cả hai (mặc định).

Ví dụ, nếu $f = [1 \ 2; 3 \ 4]$ thì lệnh dưới đây:

```
>> fp = padarray(f, [3 2], 'replicate', 'post')
```

sẽ tạo ra ma trận fp như sau:

```
fp =
     1     2     2     2
     3     4     4     4
     3     4     4     4
     3     4     4     4
     3     4     4     4
```

Ví dụ 12-10. Sử dụng hàm **colfilt**:

Sau đây là một ví dụ minh họa cách sử dụng hàm **colfilt**. Đầu tiên ta định nghĩa hàm phi tuyến **gmean**, sau đó thêm các điểm phụ bằng cách dùng hàm **padarray**. Cuối cùng gọi hàm **colfilt** với các thông số cần thiết.

```
%% Định nghĩa hàm phi tuyến
function v = gmean(A)
mn = size(A, 1);
v = prod(A, 1).^(1/mn);
%% Kết thúc định nghĩa
>> f = padarray(f, [m n], 'replicate');           % Chèn thêm các pixel
>> g = colfilt(f, [m n], 'sliding', @gmean);       % Lọc phi tuyến
```

Ngoài các hàm thực hiện bộ lọc phi tuyến tổng quát nói trên, trong MATLAB có sẵn một hàm lọc phi tuyến cụ thể, đó là bộ lọc hạng (rank filter hoặc order-statistic filter), thực hiện bởi hàm **ordfilt2**. Đồng thời, một trường hợp đặc biệt của bộ lọc này, đó là bộ lọc trung vị (median filter), cũng có thể thực hiện bằng hàm **medfilt2**.

```
>> g = ordfilt2(f, order, domain, padopt)
>> g = medfilt2(f, [m n], padopt)
```

Hàm **ordfilt** sẽ sắp thứ tự từ nhỏ đến lớn các điểm nằm trong một miền xác định bởi `domain` (`domain` là một ma trận kích thước $m \times n$ (kích thước của sổ lọc) gồm các giá trị 0 hoặc 1, trong đó các điểm có giá trị 0 sẽ không thuộc miền khảo sát), sau đó thay thế pixel của ảnh gốc bằng giá trị mức xám sắp hạng thứ `order`. Ví dụ, muốn lấy phần tử nhỏ nhất trong lân cận $m \times n$ của một pixel, ta dùng lệnh:

```
>> g = ordfilt2(f, 1, ones(m, n))
```

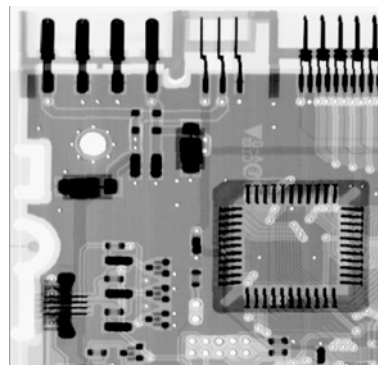
Padopt cho biết phương pháp chèn các điểm phụ là 'zero' hay 'symmetric'.

Hàm **medfilt2** là trường hợp đặc biệt của hàm **ordfilt2**, trong đó phần tử được chọn là phần tử xếp hạng chính giữa và domain là một cửa sổ kích thước $m \times n$. Với hàm này, padopt cũng có thể là 'indexed', khi đó các pixel thêm vào sẽ là 1 nếu ảnh thuộc kiểu double và là 0 trong các trường hợp khác.

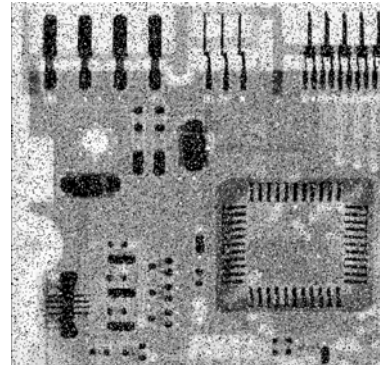
Hàm **medfilt2** thường được sử dụng trong những trường hợp có những giá trị pixel lớn hơn hoặc nhỏ hơn hẳn các pixel khác (chẳng hạn trong trường hợp có nhiễu "salt and peper"). Khi đó nếu dùng bộ lọc trung bình thì đáp ứng của bộ lọc có thể bị kéo lên (hoặc giảm xuống) theo giá trị nhiễu nói trên.

 **Ví dụ 12-11.** Sử dụng bộ lọc median để loại nhiễu "salt and peper":

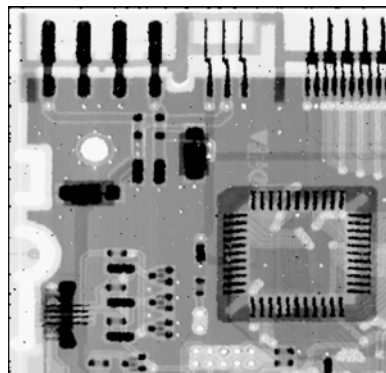
```
f=imread('circuit1.tif');           % Ảnh gốc
imshow(f)
fn = imnoise(f, 'salt & pepper', 0.2); % Ảnh nhiễu salt and peper
figure,imshow(fn)
gm = medfilt2(fn);                  % Lọc median với zero padding
figure,imshow(gm)
gms = medfilt2(fn, 'symmetric');
figure,imshow(gms)                  % Lọc median với 'symmetric' padding
```



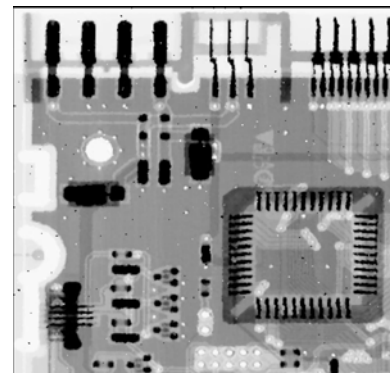
a) Ảnh gốc



b) Ảnh nhiễu



c) Lọc median với zero padding



d) Lọc median với 'symmetric' padding

Hình 12.15.

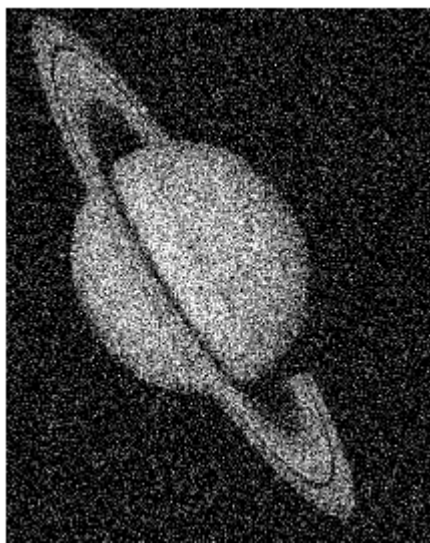
Ta thấy chất lượng ảnh lọc rất tốt so với ảnh nhiễu. Kết quả ở hình c (chèn thêm các giá trị 0) với hình d (chèn bằng phương pháp 'symmetric') là gần giống nhau, chỉ khác ở chỗ hình d không bị xuất hiện các chấm đen ở biên ảnh.

12.3.3. LỌC THÍCH NGHI

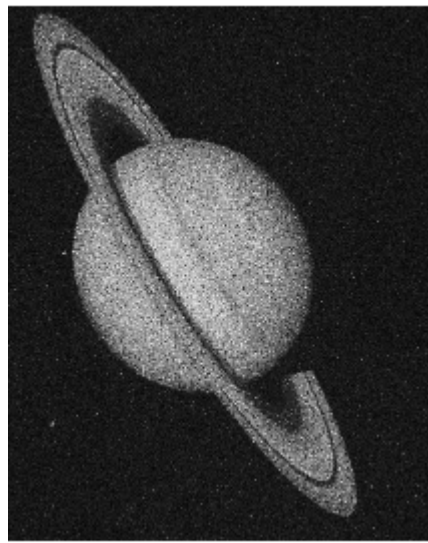
Hàm **wiener2** cho phép thực hiện bộ lọc Wiener đối với các ảnh số. Đây là một loại bộ lọc tuyến tính nhưng có tính thích nghi tùy theo các giá trị variance của từng pixel ảnh. Đối với các pixel có variance lớn, hàm này chỉ làm phẳng ảnh ở mức độ thấp nhưng với các pixel có variance nhỏ, nó sẽ có xu hướng trải rộng các giá trị mức xám hơn. Chất lượng của bộ lọc này tốt hơn các bộ lọc tuyến tính thông thường, vì nó vẫn bảo tồn được những chi tiết tần số cao của ảnh, vẫn giữ được những đường nét biên của ảnh. Tuy nhiên khối lượng tính toán đòi hỏi cao hơn do đó thời gian thực thi hàm này sẽ dài hơn. Bộ lọc Wiener thích hợp với các loại nhiễu có công suất băng phẳng, chẳng hạn nhiễu Gauss.

 **Ví dụ 12-12.** Sử dụng bộ lọc Wiener để loại nhiễu Gauss:

```
RGB = imread('saturn.png');
I = rgb2gray(RGB);
J = imnoise(I, 'gaussian', 0, 0.05);
K = wiener2(J, [5 5]);
imshow(J)
figure, imshow(K)
```



a) Ảnh nhiễu Gauss



b) Ảnh sau khi lọc Wiener

Hình 12.16.

Bài tập 12-1.

Dùng hàm **imadjust** để điều chỉnh độ tương phản cho các ảnh có trong máy tính của bạn. Có thể sử dụng hàm **imhist** để xem histogram của ảnh.

Bài tập 12-2.

Viết hàm MATLAB thực hiện các phép biến đổi mức xám sau đây:

a. $s = T(r) = A \cdot r^{1/2}$. Chọn A để $v \in [0, L-1]$.

$$b. s = T(r) = \begin{cases} L-1 & r \geq L/2 \\ 0 & r < L/2 \end{cases}$$

✎ **Bài tập 12-3.**

Định nghĩa phép biến đổi $s = T(r)$ ánh xạ tuyến tính từ $[L_{\min}, L_{\max}]$ vào $[0, L-1]$. Giả sử $L_{\min} \geq 0$ và $L_{\max} \leq L-1$.

Viết chương trình MATLAB sử dụng phép biến đổi nói trên để cân bằng ảnh **pout.tif**.

✎ **Bài tập 12-4.**

Load 6 ảnh tùy chọn trong máy, sau đó thay đổi độ tương phản bằng cách mở rộng histogram (dùng **imadjust**) và bằng cách cân bằng histogram (dùng **histeq**). Hiện thị tất cả 18 ảnh cùng với histogram tương ứng của chúng. Sắp xếp các ảnh này trong 6 figure, mỗi figure gồm ảnh gốc, hai ảnh sau khi biến đổi cùng với các histogram của chúng.

✎ **Bài tập 12-5.**

Làm lại bài tập 12-4 nhưng sử dụng hai phương pháp: cân bằng histogram và phối hợp histogram theo dạng bimodal Gaussian.

✎ **Bài tập 12-6.**

Cân bằng histogram cho ảnh **pout.tif** sử dụng phương pháp cân bằng thích nghi (tự chọn các thông số thích hợp để đạt kết quả tối ưu).

✎ **Bài tập 12-7.**

Xét ảnh **pic1_degraded.bmp** là ảnh kém chất lượng do bị nhiễu so với ảnh gốc là **pic1_original.bmp**. Nhiễu được hình thành theo cách sau: màu ở mỗi pixel sẽ bị thay đổi thành 0 hoặc 255 với xác suất bằng p , và giữ nguyên với xác suất bằng $1 - p$ (nhiễu salt and pepper). Nhiễu ở mỗi pixel đều độc lập với nhau.

a. Hãy ước lượng giá trị của p .

b. Lọc ảnh bằng bộ lọc median kích thước 3×3 . Các điểm biên được xử lý theo cách mặc định của MATLAB. Tính MSE của ảnh nhiễu so với ảnh gốc và ảnh sau khi lọc so với ảnh gốc:

$$MSE = \frac{1}{3MN} \sum_{c=R,G,B} \sum_{x=1}^M \sum_{y=1}^N [Image_1(x, y, c) - Image_2(x, y, c)]^2$$

c.* Phương pháp này chưa tối ưu vì quá trình lọc được áp dụng trên tất cả các điểm ảnh trong khi nhiễu chỉ xảy ra ở một vài pixel. Hãy tìm một phương pháp tốt hơn để lọc ảnh. Viết hàm MATLAB để thực hiện ý tưởng đó.



pic1_degraded.bmp



pic1_original.bmp

✎ **Bài tập 12-8.**

Sử dụng các bộ lọc tuyến tính để lọc nhiễu cho ảnh trong bài 12-7:

$$\text{a. } W = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{b. } W = \frac{1}{9} \begin{bmatrix} 1 & 3 & 1 \\ 3 & 1 & 3 \\ 1 & 3 & 1 \end{bmatrix}$$

✎ **Bài tập 12-9.**

Lần lượt cộng các loại nhiễu khác nhau vào ảnh **autumn.tif**, sau đó dùng hàm **colfilt** để lọc ảnh nhiễu, hàm lọc phi tuyến được sử dụng là $s = \sqrt{\prod r_k}$ với r_k là độ xám của các pixel nằm trong lân cận kích thước 3x3 của pixel đang xét.

- Nhiễu Gaussian với trung bình bằng 0 và variance bằng 0.05
- Nhiễu salt and pepper với mật độ 0.05
- Nhiễu Poisson
- Nhiễu Speckle với mật độ 0.05

✎ **Bài tập 12-10.**

Làm lại bài tập trên sử dụng hàm **nlfilter**.

✎ **Bài tập 12-11.**

Dùng bộ lọc Wiener để lọc các ảnh nhiễu trong bài 12-9. Từ đó nhận xét trong trường hợp nào bộ lọc Wiener đạt hiệu quả cao nhất.

Danh sách các hàm được giới thiệu trong chương 12

Phương pháp biến đổi mức xám

imadjust	Biến đổi mức xám của ảnh
stretchlim	Xác định các giới hạn biến đổi cho hàm imadjust

Phương pháp cân bằng histogram

adapthisteq	Cân bằng histogram theo phương pháp thích nghi
bar	Vẽ biểu đồ thanh
histeq	Cân bằng histogram
imhist	Xác định và vẽ biểu đồ histogram của ảnh
plot	Vẽ đồ thị của một hàm cho trước
stem	Vẽ biểu đồ xung

Các hàm thực hiện lọc ảnh

colfilt	Hàm thực hiện lọc phi tuyến, xử lý theo cột
fspecial	Tạo các bộ lọc tuyến tính theo các dạng có sẵn
imfilter	Thực hiện lọc tuyến tính
imnoise	Tạo nhiễu và cộng vào ảnh
medfilt2	Thực hiện bộ lọc trung vị (median)
nlfilt	Thực hiện lọc phi tuyến, xử lý trực tiếp trên cả hai chiều
padarray	Chèn các pixel ở biên trước khi thực hiện lọc
ordfilt2	Thực hiện bộ lọc thứ tự tổng quát
wiener2	Thực hiện bộ lọc thích nghi Wiener

Chương 13

NÉN ẢNH SỐ

Mục đích của việc nén ảnh số là mã hoá các dữ liệu ảnh về một dạng thu gọn, tối thiểu hoá cả số bits dùng để biểu diễn ảnh lẫn các sai khác do quá trình nén gây ra. Tầm quan trọng của vấn đề nén ảnh có thể thấy rõ qua các số liệu cụ thể: với một bức ảnh trắng đen kích thước 512x512 pixels, mỗi pixel được biểu diễn bởi 8 bits (biểu diễn một trong 256 giá trị mức xám), cần khoảng 256 Kbytes dữ liệu. Với ảnh màu cần gấp ba lần con số này. Với các dữ liệu video, cần 25 frames trên một giây, như vậy 1 đoạn video chỉ 30s phải cần đến 540MB dữ liệu, một con số quá lớn. Do đó vấn đề nén ảnh là hết sức cần thiết.

Nói chung, các phương pháp nén ảnh chủ yếu được phân thành 2 nhóm: nhóm không tổn hao và nhóm có tổn hao. Các phương pháp nén ảnh không tổn hao cho phép biểu diễn ảnh với chất lượng hoàn toàn ngang bằng với ảnh gốc. Các phương pháp này dựa trên các giải thuật nén được áp dụng cho tất cả các đối tượng dữ liệu nói chung chứ không chỉ riêng dữ liệu ảnh, ví dụ mã Huffman, mã số học, mã Golomb, ... Tuy nhiên, các phương pháp này không lợi dụng được những đặc tính riêng của dữ liệu ảnh và tỷ lệ nén rất thấp. Do đó, trong thực tế, các phương pháp nén có tổn hao là các phương pháp được sử dụng chủ yếu. Với các phương pháp này, luôn có sự đánh đổi giữa dung lượng ảnh với chất lượng ảnh.

13.1. PHƯƠNG PHÁP MÃ HOÁ XỬ LÝ KHỐI BTC (BLOCK TRUNCATING CODING)

Ý tưởng cơ bản của phương pháp này là chia ảnh ra thành nhiều khối 4x4 và lượng tử hoá các pixel trong khối về hai giá trị a và b. Trong mỗi khối, cần tính giá trị trung bình $\bar{x}$ và độ lệch chuẩn σ để thực hiện mã hoá. Quá trình lượng tử hoá hai mức được thực hiện đối với các pixel trong mỗi khối sao cho bit 0 được dùng biểu diễn cho các giá trị nhỏ hơn giá trị trung bình, và bit 1 biểu diễn các mức xám của các pixel còn lại. Ảnh sẽ được tái tạo từ các giá trị $\bar{x}$ và σ và từ ma trận các bit dữ liệu mã hoá (còn gọi là mặt phẳng bit) bằng cách gán các giá trị mức xám a và b cho các bit 0 và 1 tương ứng:

$$a = \bar{x} - \sigma \sqrt{\frac{q}{m-q}} \quad (13.1)$$

$$b = \bar{x} + \sigma \sqrt{\frac{m-q}{q}} \quad (13.2)$$

trong đó m (=16) là tổng số pixel trong mỗi khối, q là số bit 1 trong mặt phẳng bit. Các giá trị lượng tử được chọn như trên để giá trị trung bình và variance của ảnh vẫn được bảo toàn sau khi giải nén, do đó phương pháp này còn gọi là phương pháp mã hoá xử lý khối bảo toàn moment (MPBTC - moment preserving BTC). Một dạng khác của phương pháp BTC là phương pháp BTC moment tuyệt đối (AMBTC - absolute moment BTC) chọn các giá trị lượng tử a và b là các giá trị trung bình của các pixels trong hai nhóm (nhóm bit 1 và nhóm bit 0):

$$a = \frac{1}{m-q} \cdot \sum_{x_i < \bar{x}} x_i \quad (13.3)$$

$$b = \frac{1}{q} \cdot \sum_{x_i > \bar{x}} x_i \quad (13.4)$$

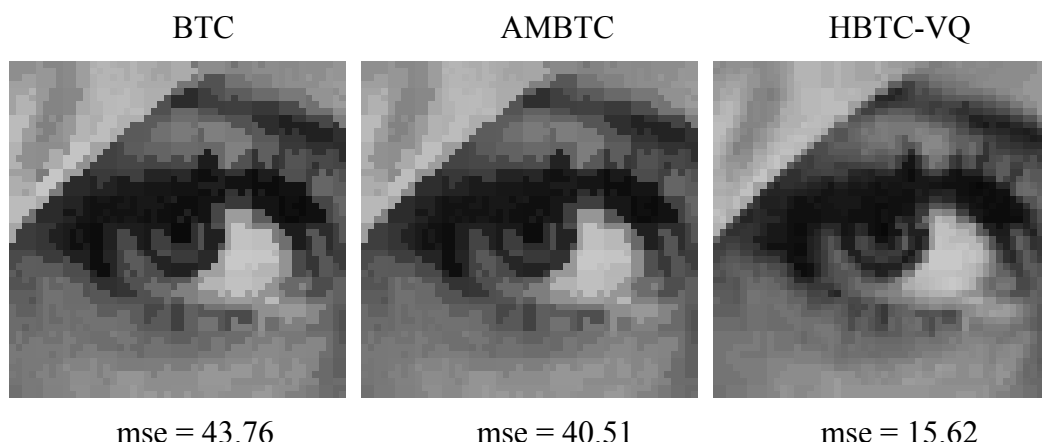
Cách chọn các giá trị a và b như trên không bảo đảm bảo toàn moment bậc hai (tức là variance) của ảnh. Tuy nhiên, các kết quả phân tích cho thấy giá trị biểu diễn cho các pixel để đạt MSE tối ưu chính là giá trị trung bình của các pixel chứ không phải variance. Hình 13.1 là một thí dụ về phương pháp BTC bảo toàn moment.

Ảnh gốc	Mặt phẳng bit	Ảnh tái tạo																																																
<table><tr><td>2</td><td>9</td><td>12</td><td>15</td></tr><tr><td>2</td><td>11</td><td>11</td><td>9</td></tr><tr><td>2</td><td>3</td><td>12</td><td>15</td></tr><tr><td>3</td><td>3</td><td>4</td><td>14</td></tr></table>	2	9	12	15	2	11	11	9	2	3	12	15	3	3	4	14	<table><tr><td>0</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	1	1	0	1	1	1	0	0	1	1	0	0	0	1	<table><tr><td>2</td><td>12</td><td>12</td><td>12</td></tr><tr><td>2</td><td>12</td><td>12</td><td>12</td></tr><tr><td>2</td><td>2</td><td>12</td><td>12</td></tr><tr><td>2</td><td>2</td><td>2</td><td>12</td></tr></table>	2	12	12	12	2	12	12	12	2	2	12	12	2	2	2	12
2	9	12	15																																															
2	11	11	9																																															
2	3	12	15																																															
3	3	4	14																																															
0	1	1	1																																															
0	1	1	1																																															
0	0	1	1																																															
0	0	0	1																																															
2	12	12	12																																															
2	12	12	12																																															
2	2	12	12																																															
2	2	2	12																																															
$\bar{x} = 7.94$	$q = 9$	$a = 2.3$																																																
$\sigma = 4.91$		$b = 12.3$																																																

Hình 13.1

Đối với phương pháp BTC bảo toàn moment, dữ liệu lượng tử được biểu diễn bởi cặp $(\bar{x}, \sigma)$. Một nhược điểm của phương pháp này là các mức lượng tử khi giải mã lại được tính từ các giá trị $(\bar{x}, \sigma)$ đã được lượng tử hoá, nghĩa là các giá trị $(\bar{x}, \sigma)$ có sai số do quá trình làm tròn. Như vậy, chất lượng ảnh sẽ bị giảm do nguyên nhân phát sinh ngay từ giai đoạn mã hoá. Một cách giải quyết khác là tính các giá trị lượng tử a và b ngay ở giai đoạn mã hoá và truyền cặp (a, b) thay vì cặp $(\bar{x}, \sigma)$. Với cách này, ta vừa có thể giảm được sai số lượng tử vừa giảm được quá trình tính toán khi giải mã.

Tuy nhiên, nhược điểm chính của phương pháp BTC là chất lượng của nó rất kém đối với các khối có độ tương phản cao bởi vì trong trường hợp này chỉ hai giá trị lượng tử không thể mô tả đủ toàn bộ khối. Để giải quyết khó khăn này người ta sử dụng phương pháp thay đổi các kích thước của các khối. Với các khối kích thước lớn, độ tương phản thấp, ta giảm được tốc độ bit yêu cầu, ngược lại các khối kích thước nhỏ thích hợp với những vùng có độ tương phản cao để bảo đảm chất lượng ảnh. Một giải pháp khác cho vấn đề trên là phân cấp theo cấu trúc cây 4 nhánh: ảnh được chia thành các khối kích thước $m_1 \times m_1$, nếu σ nhỏ hơn một giá trị ngưỡng định trước σ_{th} (nghĩa là độ tương phản thấp) thì vẫn áp dụng giải thuật BTC, nếu ngược lại, chia khối này thành 4 khối con và quá trình trên lặp lại cho đến khi đạt tới mức ngưỡng hoặc khi tới giới hạn nhỏ nhất của kích thước khối $m_2 \times m_2$.



Hình 13.2. Các ảnh nén bằng các giải thuật BTC khác nhau.

Hình vẽ 13.2 minh hoạ kết quả nén ảnh bằng các biến thể khác nhau của giải thuật BTC, trong đó hình thứ ba là kết quả áp dụng kết hợp các giải pháp trình bày ở trên – phương pháp HBTC – VQ (Hierarchical BTC with Vector Quantization – BTC phân cấp với lượng tử hoá vector).

MATLAB (Version 7.0) không cung cấp các hàm nén ảnh cụ thể. Tuy nhiên, chúng ta có thể viết các hàm thực hiện các giải thuật nén ảnh thông dụng từ các hàm có sẵn trong MATLAB nói chung và trong Image Processing Toolbox nói riêng. Ví dụ, sau đây là một hàm thực hiện giải thuật nén BTC.

Trong phần phụ lục cuối quyển sách này, bạn đọc có thể tìm thấy một hàm thực hiện phép nén ảnh theo giải thuật BTC: hàm **btcode**. Cú pháp của hàm này như sau:

```
>> out = btcode (infile,bx,by,outfile)
```

Trong đó *infile* là tên file ảnh gốc cần nén, *bx*, *by* là các kích thước của mỗi khối, *outfile* là tên file ảnh sau khi nén.

 **Ví dụ 13-1.** Dùng hàm **btcode** để nén ảnh theo giải thuật BTC:

Sau đây là chương trình kết quả thực hiện nén ảnh **peppers.tif** bằng cách dùng hàm **btcode**:

```
in = imread('peppers.tif')
out = btcode('peppers.tif',4,4,'peppers_comp.tif')
imshow(in)
figure, imshow(out)
```



Ảnh gốc

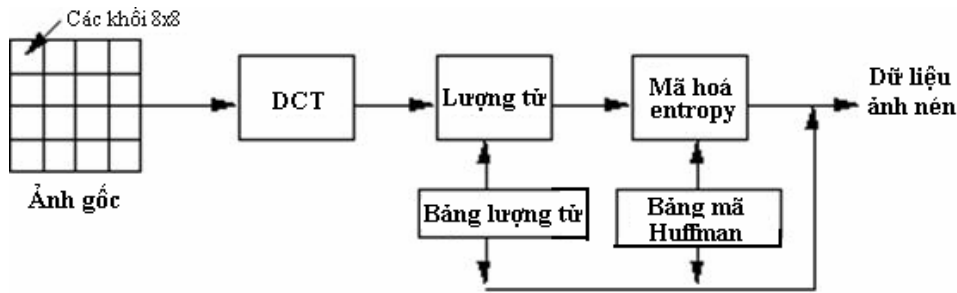
Ảnh nén

Hình 13.3.

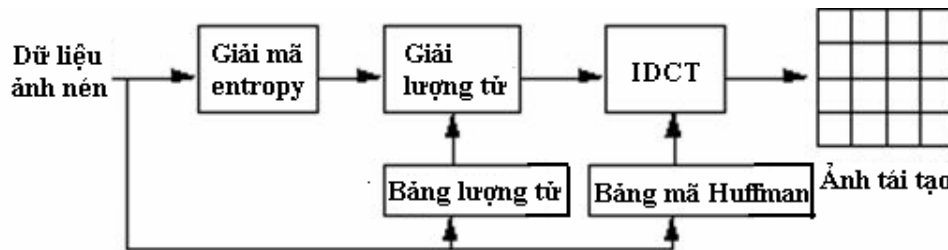
Dung lượng file ảnh gốc là 280KB còn của ảnh nén là 214KB.

13.2. NÉN TỔN HAO DỰA VÀO DCT

Giải thuật nén có tổn hao được biết đến nhiều nhất có lẽ là giải thuật nén dùng DCT. Đây là giải thuật được chuẩn hoá với tên gọi JPEG (lấy từ tên gọi của tổ chức đã định ra tiêu chuẩn nén này: Joint Photographic Experts Group (Nhóm liên kết các chuyên gia xử lý ảnh)). Hình vẽ 13.4 và 13.5 mô tả sơ đồ khối của giải thuật nén và giải nén ảnh JPEG.



Hình 13.4.



Hình 13.5.

Để nén ảnh theo giải thuật JPEG, ta chia ảnh thành các khối 8x8 (hoặc 16x16). Mỗi khối 8x8 này sẽ được xử lý riêng biệt qua các bước của quá trình nén ảnh. Đầu tiên, ta thực hiện biến đổi DCT thuận đối với mỗi khối. Như chúng ta đã biết, do các điểm ảnh kề cận nhau thường có tính tương quan rất cao, phép biến đổi DCT thuận có xu hướng tập trung hầu hết năng lượng của bức ảnh vào trong một vài hệ số DCT tần số thấp. Đây là cơ sở để ta có thể thực hiện nén ảnh. Với một khối ảnh kích thước 8x8 trích từ ảnh nguồn, hầu hết các hệ số của nó đều bằng 0 hoặc gần bằng 0. Như vậy, ta có thể không cần biểu diễn các hệ số này khi truyền dữ liệu ảnh đi. Lưu ý rằng bản thân biến đổi DCT không làm mất mát thông tin của ảnh gốc, nó chỉ chuyển các thông tin này về một dạng khác mà ta có thể mã hoá một cách hiệu quả hơn.

Sau khi qua bộ biến đổi DCT, mỗi hệ số trong số 64 hệ số biến đổi DCT được lượng tử hoá dựa vào một bảng giá trị lượng tử được thiết kế kỹ lưỡng. Một phương pháp lượng tử đơn giản có thể dùng là chỉ giữ lại một vài hệ số DCT tần số thấp (các hệ số có giá trị lớn) còn tất cả các hệ số còn lại gán bằng 0. Trong chuẩn nén JPEG, mỗi hệ số DCT sẽ được chia cho một trọng số ở vị trí tương ứng trong một ma trận lượng tử 8x8, sau đó làm tròn về số nguyên gần nhất:

$$C_{ij} = \text{round} \left(\frac{D_{ij}}{Q_{ij}} \right) \quad (13.5)$$

với D là ma trận các hệ số DCT, Q là ma trận lượng tử (kích thước 8x8).

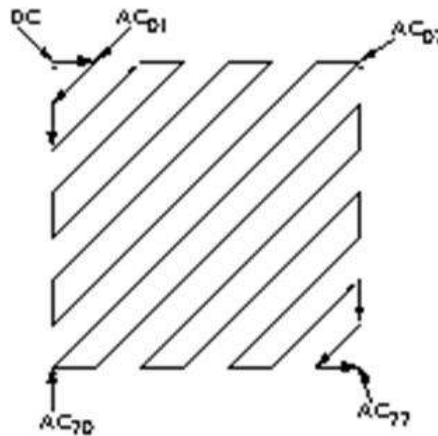
Thông thường mắt người rất khó phân biệt các thành phần tần số cao của ảnh, do đó các thành phần này sẽ được chia cho các trọng số lớn hơn. JPEG định nghĩa các ma trận lượng tử tùy theo cấp chất lượng từ 1 đến 100 (cấp 100 có chất lượng tốt nhất). Với cấp chất lượng 50, ma trận lượng tử được định nghĩa là:

$$Q_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Với các cấp chất lượng k với $k \neq 50$, ma trận lượng tử được định nghĩa bởi phương trình sau:

$$Q_k = \begin{cases} Q_{50} \cdot \left(\frac{50}{k} \right) & k < 50 \\ Q_{50} \cdot \left(\frac{100-k}{50} \right) & k > 50 \end{cases} \quad (13.6)$$

Sau khi lượng tử, các hệ số biến đổi DCT sẽ được sắp xếp theo một chuỗi zig-zag bắt đầu từ thành phần DC, rồi đến thành phần AC thứ 1, thành phần AC thứ 2, ... như hình vẽ 13.6.



Hình 13.6.

Cuối cùng các hệ số này được chuyển thành chuỗi bit nhị phân bằng các kỹ thuật mã hoá entropy. Kỹ thuật mã hoá thường dùng là RLC (Run Length Coding) trong đó thay vì dùng chuỗi các bit 0 liên tiếp thì ta chỉ cần dùng một vài bit để biểu diễn chiều dài của chuỗi. Phương pháp này thích hợp với các hệ số DCT nói trên vì hầu hết các hệ số DCT tần số cao đều bằng 0.

Ở công đoạn giải mã, bộ giải mã (bộ đọc file JPEG) sẽ thực hiện quá trình ngược lại: giải mã entropy, sau đó nhân các hệ số thu được với phần tử tương ứng của ma trận lượng tử, rồi biến đổi ngược DCT để tái tạo lại ảnh ban đầu.

MATLAB không cung cấp hàm thực hiện nén ảnh bằng DCT. Để thực hiện quá trình nén dựa trên DCT, ta phải viết đoạn chương trình dùng các hàm xử lý ảnh cơ bản để thực hiện các bước nén ảnh như trên. Trong phần phụ lục cuối sách, chúng tôi có giới thiệu hàm **detcompr** thực hiện giải thuật nén DCT với cú pháp như sau:

```
>> im = dctcompr (infile,coeff,outfile)
```

`infile`, `outfile` là tên các file ảnh gốc và ảnh sau khi nén, `coeff` là số các hệ số DCT được giữ lại trong toàn bộ ảnh.

Ví dụ dưới đây giới thiệu một đoạn chương trình ngắn khác để thực hiện nén ảnh theo DCT. Trong ví dụ này, chúng ta sử dụng hàm **dctmtx** để tạo ra ma trận biến đổi DCT kích thước $N \times N$ (cụ thể $N = 8$):

```
>> D = dctmtx(N)
```

Từ đó có thể tính các hệ số DCT của ảnh A (với A là ma trận vuông) bằng công thức $D \cdot A \cdot D'$. Cách tính này cho kết quả nhanh hơn so với hàm DCT, đặc biệt là khi ta chia ảnh thành nhiều khối nhỏ vì khi đó chỉ cần tính ma trận D một lần. Để chia ảnh thành các khối con và xử lý từng khối, ta có thể dùng hàm **blkproc**:

```
>> B = blkproc(A,[M N],fun)
```

Hàm này xử lý ảnh đầu vào A bằng cách chia A thành các khối $M \times N$ và áp dụng hàm `fun` cho mỗi khối.

 **Ví dụ 13-2.** Thực hiện giải thuật nén ảnh dựa trên DCT bằng cách chỉ giữ lại 10 hệ số DCT đầu tiên trong tổng số 64 hệ số DCT:

```
I = imread('cameraman.tif'); % Đọc ảnh cameraman.tif
I = im2double(I); % Chuyển sang kiểu double
T = dctmtx(8); % Tính ma trận biến đổi DCT 8x8
B = blkproc(I,[8 8],'P1*x*P2',T,T'); % Thực hiện phép nhân T*X*T' với mỗi khối

mask = [1 1 1 1 0 0 0 0 % Mặt nạ, dùng để giữ lại chỉ 10 hệ số DCT
        1 1 1 0 0 0 0 0
        1 1 0 0 0 0 0 0
        1 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0];

B2 = blkproc(B,[8 8],'P1.*x',mask); % Lượng tử hoá các hệ số DCT
I2 = blkproc(B2,[8 8],'P1*x*P2',T,T'); % Giải mã
imshow(I), figure, imshow(I2) % Hiển thị các ảnh
```



Ảnh gốc



Ảnh nén bằng giải thuật DCT

Hình 13.7.

Có thể nhận thấy rằng mặc dù khoảng 85% hệ số DCT đã bị loại bỏ nhưng chất lượng ảnh vẫn có thể chấp nhận được.

 **Ví dụ 13-3.** Thực hiện giải thuật nén DCT dùng hàm *dctcompr*:

```
in = imread('autumn.tif')
out=dctcompr('autumn.tif',8000,'autumn_comp.tif');
imshow(in)
figure, imshow(out)
```



Ảnh gốc



Ảnh nén dùng giải thuật DCT

Hình 13.8.

13.3. NÉN ẢNH BẰNG GIẢI THUẬT PHÂN TÍCH TRỊ RIÊNG (SVD)

13.3.1. GIỚI THIỆU PHƯƠNG PHÁP SVD

Phương pháp phân tích trị riêng (SVD – Singular Value Decomposition) là một đề tài rất được quan tâm của Đại số tuyến tính. Phương pháp này có nhiều ứng dụng thực tế, một trong số đó là ứng dụng trong kỹ thuật nén ảnh. Đặc điểm quan trọng của phương pháp này là nó có thể áp dụng cho bất kỳ ma trận thực $m \times n$ nào. Nội dung của nó là: phân tích một ma trận A cho trước thành 3 ma trận U , S , V , sao cho:

$$A = USV^T \quad (13.7)$$

trong đó U và V là các ma trận trực giao và S là ma trận đường chéo. Ma trận U là ma trận gồm các vector riêng trái của A , ma trận V là ma trận gồm các vector riêng phải của A và ma trận S là ma trận đường chéo, mỗi phần tử đường chéo là một trị riêng của A . Các trị riêng được sắp trên đường chéo chính theo thứ tự sau:

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0 \quad (13.8)$$

với r là hạng của ma trận A và $p = \min\{m, n\}$.

Để tìm các ma trận S , U , V , ta thực hiện các bước như sau:

Bước 1: tìm ma trận V . Nhân A^T vào hai vế của (13.7), ta có:

$$A^T.A = (USV^T)^T.USV^T = V.S^T.U^T.U.S.V^T = V.S^T.S.V^T = V.S^2.V^T \quad (13.9)$$

(do $U^T.U = I$). Như vậy, để tìm ma trận S và V , ta chỉ cần tìm các trị riêng và các vector riêng của $A^T.A$ vì từ (13.9) ta thấy các trị riêng của $A^T.A$ chính là bình phương các phần tử của S còn các vector riêng của $A^T.A$ chính là các cột của V .

Bước 2: tìm ma trận U . Nhân hai vế của (13.7) với A^T và sử dụng: $V^T.V = I$, ta cũng có:

$$A.A^T = U.S^2.U^T \quad (13.10)$$

Vậy các cột của U chính là các vector riêng của $A.A^T$.

Cuối cùng, ta phân tích ma trận A dưới dạng như sau:

$$A = \begin{pmatrix} u_1 & \dots & u_r & \dots & u_m \end{pmatrix} \begin{pmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & \ddots & \\ & & & & 0 \end{pmatrix} \begin{pmatrix} v_1^T \\ \vdots \\ v_r^T \\ \vdots \\ v_n^T \end{pmatrix} \quad (13.11)$$

A và S là các ma trận kích thước $m \times n$, U là ma trận $m \times m$ và V là ma trận $n \times n$.

13.3.2. ỨNG DỤNG SVD ĐỂ NÉN ẢNH SỐ

Bằng cách phân tích ma trận ảnh A dưới dạng $A = USV^T$, ta có thể biểu diễn xấp xỉ ma trận A bằng ít phần tử hơn. Nếu hạng của ma trận A là $r < m$, hoặc $r < n$, ta có thể giảm đi những thông tin thừa:

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_r u_r v_r^T + 0 u_{r+1} v_{r+1}^T + \dots \quad (13.12)$$

Hơn nữa, các trị riêng được sắp xếp theo thứ tự giảm dần, do đó những số hạng phía sau sẽ có ít ảnh hưởng đến ảnh và có thể bỏ đi các số hạng này.

MATLAB có hàm **svd** để thực hiện nhân tử hoá ma trận A thành USV^T :

```
>> [U, S, V] = svd(A)
```

Chúng ta có thể sử dụng hàm này kết hợp với phương trình (13.12) để viết một hàm nén ảnh dùng giải thuật SVD. Bạn đọc có thể tham khảo hàm **svdcompr** ở phần phụ lục. Cú pháp của nó là:

```
>> im = svdcompr (infile, singvals, outfile)
```

trong đó *infile* và *outfile* là tên các file ảnh gốc và ảnh nén, *singvals* là số các trị riêng lớn nhất được giữ lại.

 **Ví dụ 13-4.** Thực hiện giải thuật nén SVD dùng hàm *svdcompr*:

Trong ví dụ này ta chỉ giữ lại 30 số hạng đầu tiên trong phương trình (13.12).

```
in = imread('2.bmp')
out=svdcompr('2.bmp',30,'svd.bmp');
imshow(in),figure, imshow(out)
```



Ảnh gốc



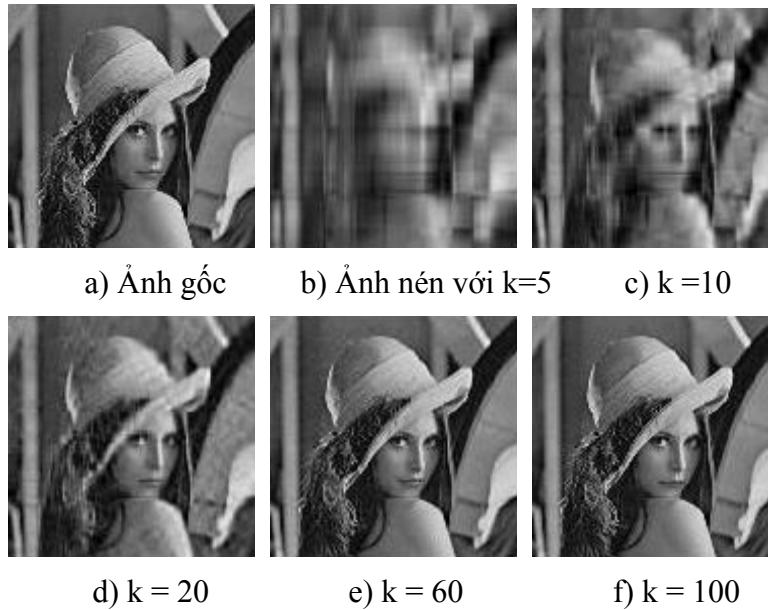
Ảnh nén bằng giải thuật SVD

Hình 13.9.

 **Ví dụ 13-5.** Một đoạn chương trình khác thực hiện nén ảnh theo giải thuật SVD:

```
close all
[A,map]=imread('lena.gif');
B=im2double(A,'indexed');
imshow(B,map)
[u,s,v]=svd(B);
C=zeros(size(B));
for j=1:k
C=C+s(j,j)*u(:,j)*v(:,j).';
end
C=floor(C);
imshow(C,map)
k=find(C<1);
C(k)=1;
set(gcf,'Unit','inches','Paperposition',[0,0,2,1])
print -djpeg 'lenak.jpg'
```

Dưới đây là kết quả của đoạn chương trình trên với các giá trị k (số vòng lặp, cũng là số trị riêng được giữ lại) khác nhau:

**Hình 13.10.**

Trong hình vẽ 13.9, hình b ứng với số trị riêng giữ lại là 5, ảnh này tiết kiệm được 99,9% dung lượng so với ảnh gốc nhưng chất lượng kém, chỉ thấy được dạng ảnh mà không thấy rõ các chi tiết. Nếu giữ lại 60 trị riêng lớn nhất, chất lượng ảnh đã đạt gần với ảnh gốc (hình e) và ta tiết kiệm được khoảng 84% dung lượng. Khi tăng số lần lặp lên 100, chất lượng ảnh rất tốt (hình f) mà vẫn tiết kiệm được 55% so với ảnh gốc.

🔗 Bài tập 13-1.

Viết lại hàm **btcode** trong phần 13.1 để thực hiện các giải thuật BTC khác, bao gồm BTC với kích thước khối biến đổi và BTC-VQ.

🔗 Bài tập 13-2.

Sử dụng hàm **btcode** ở phần phụ lục và hàm vừa viết trong bài tập 13-2 để nén ảnh **autumn.tif**. Từ đó so sánh các giải thuật BTC thông qua tỷ lệ nén và MSE của ảnh.

🔗 Bài tập 13-3.

Kiểm tra xem hàm **dctcompr** có thực hiện đúng theo giải thuật nén JPEG chuẩn không? Nếu chưa hãy sửa lại hàm này cho đúng chuẩn JPEG.

🔗 Bài tập 13-4.

Dùng hàm **dctcompr** và hàm đã thực hiện trong bài tập 13-3 để nén ảnh **peppers.png**. So sánh kết quả thông qua tỷ lệ nén và MSE của ảnh.

🔗 Bài tập 13-5.

Thực hiện lại ví dụ 13-2 đối với 3 ảnh tùy chọn. Với mỗi ảnh xét 3 trường hợp:

- Chỉ giữ lại 5 hệ số DCT đầu tiên
- Giữ lại 20 hệ số DCT đầu tiên
- Giữ lại 40 hệ số DCT đầu tiên

🔗 Bài tập 13-6.

Thực hiện nén các ảnh trong bài tập 13-5 bằng giải thuật SVD. Lần lượt xét các trường hợp:

- Giữ lại 10 trị riêng lớn nhất

- b. Giữ lại 50 trị riêng lớn nhất
- c. Giữ lại 100 trị riêng lớn nhất

✎ **Bài tập 13-7.**

Viết các hàm thực hiện các giải thuật nén ảnh khác mà bạn biết. Có thể tham khảo tại web site <http://www.mathworks.com/matlabcentral/fileexchange>.

Danh sách các hàm được giới thiệu trong chương 13

blkproc	Chia ảnh thành nhiều khối và xử lý từng khối
btcode	Nén ảnh bằng phương pháp mã hoá xử lý khối
dct2	Biến đổi DCT
dctcompr	Nén ảnh có tổn hao bằng phương pháp DCT
dctmtx	Tính ma trận biến đổi DCT
svdcompr	Nén ảnh bằng phương pháp phân tích trị riêng



PHẦN III

ỨNG DỤNG MATLAB TRONG HỆ THỐNG VIỄN THÔNG

Chương 14

MÃ HÓA NGUỒN

Các nguồn tin tức trong đời sống vô cùng đa dạng và phong phú. Để xử lý các tin tức này bằng các hệ thống thông tin thì trước hết các tin tức này phải được biến đổi để trở thành các tín hiệu mà hệ thống có thể nhận biết và xử lý được. Quá trình này được gọi là quá trình *mã hoá nguồn*, hoặc còn có các tên gọi khác như quá trình *lượng tử hoá* hay *định dạng tín hiệu*. Mã hoá nguồn là quá trình xử lý các dữ liệu tin tức để làm giảm những phần dư thừa và để chuẩn bị cho các thao tác xử lý cần thiết sau đó. Hai vấn đề chính khi đề cập đến mã hoá nguồn là: chuyển đổi tương tự – số và nén dữ liệu.

14.1. TẠO MỘT NGUỒN TÍN HIỆU

Trong MATLAB, cách phổ biến để biểu diễn một tín hiệu là biểu diễn dưới dạng vector hoặc ma trận. MATLAB cung cấp hai dạng tín hiệu ngẫu nhiên dùng làm nguồn phát: nguồn ký hiệu ngẫu nhiên và nguồn số ngẫu nhiên. Ngoài ra còn có hai dạng nguồn tín hiệu khác phục vụ cho việc mô phỏng kênh truyền, đó là nguồn nhiễu trắng phân bố Gauss và nguồn tạo bit lỗi để mô phỏng các bộ mã sửa sai (xem chương 16 và chương 17).

▪ Hàm **randsrc** cho phép tạo ra một ma trận mà mỗi phần tử của nó được lấy từ một tập ký hiệu cho trước với xác suất xuất hiện được quy định bởi người sử dụng. Dòng lệnh sau đây cho phép tạo một ma trận 5x4, trong đó mỗi phần tử của nó được lấy từ một trong ba số 0,1,2 với xác suất xuất hiện tương ứng là 0.5, 0.25 và 0.25:

```
>> a=randsrc(5,4,[0,1,2;0.5,0.25,0.25])
```

a =

1	0	0	0
2	2	0	0
2	2	2	1
1	0	0	0
0	2	0	0

Nếu không chỉ rõ xác suất xuất hiện của mỗi ký hiệu thì các ký hiệu sẽ có xác suất phân bố đều.

```
>> a=randsrc(2,5,[0,1])
```

a =

1	1	1	1	0
0	0	1	0	0

▪ Để tạo một ma trận gồm các số nguyên ngẫu nhiên trong một khoảng cho trước, ta sử dụng hàm **randint**:

```
>> c=randint(5,4,[2,5])
```

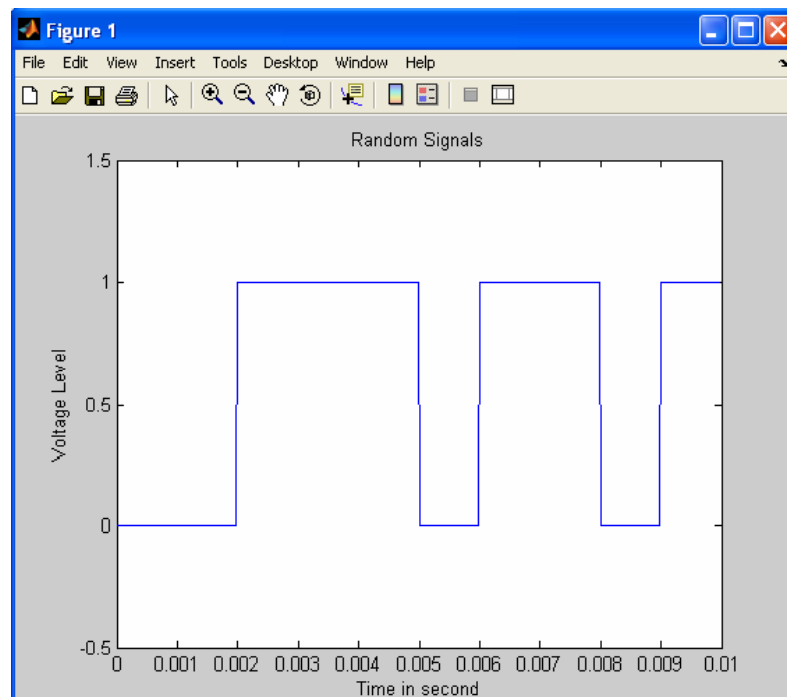
c =

2	4	3	4
4	3	5	3
3	5	5	3

Giả sử chuỗi bit trên được chuyển thành tín hiệu đơn cực với tốc độ bit là fb . Để vẽ tín hiệu này ta tạo ra nhiều mẫu tín hiệu trong một chu kỳ bit. Ví dụ đoạn mã sau sẽ thực hiện vẽ tín hiệu ứng với 10 bit đầu tiên của chuỗi bit với tần số lấy mẫu $fs = 50fb$:

```
fb = 1000; % Tốc độ bit
fs = 50*fb; % Tần số lấy mẫu
nbit = 10; % Vẽ 10 chu kỳ bit đầu tiên
len = fs/fb*nbit; % Tổng số mẫu trong 10 chu kỳ bit
time = 0:(1/fs):(len-1)/fs; % Vector thời gian
x = randint(n,1); % Chuỗi dữ liệu nhị phân ngẫu nhiên
pattern=[]; % khởi tạo vector tín hiệu
for k=1:nbit % Xét từng chu kỳ bit
    if x(k,1)==0
        sig=zeros(1,50); % Tạo 50 mẫu giá trị 0
    else
        sig=ones(1,50); % Tạo 50 mẫu giá trị 1
    end
    pattern=[pattern sig];
end
plot(time,pattern);
axis([0 len/fs -0.5 1.5]);
title('Random Signals');
xlabel('Time in second'); ylabel('Voltage Level');
```

Kết quả thu được khi thực thi chương trình:



Hình 14.2.

Lưu ý: Các bạn có thể nhận được các kết quả khác với hình vẽ trên vì chuỗi bit nhận được từ hàm **randint** sẽ thay đổi ngẫu nhiên sau mỗi lần thực thi chương trình.

14.2. LƯỢNG TỬ HÓA TÍN HIỆU

Lượng tử hoá là quá trình rời rạc hoá tín hiệu về mặt biên độ, cụ thể là thay thế tất cả các giá trị của tín hiệu nằm trong một khoảng xác định nào đó thành một giá trị duy nhất. Miền giá trị của tín hiệu được chia thành một số hữu hạn các khoảng chia. Như vậy, độ lớn của tín hiệu sau khi lượng tử chỉ có thể nhận một trong số hữu hạn các giá trị cho trước.

Tập hợp các khoảng chia gọi là sự phân hoạch của tín hiệu (partition). Tập các giá trị thay thế cho mỗi khoảng chia gọi là bộ mã (codebook).

MATLAB biểu diễn phân hoạch của tín hiệu bằng một vector mà các phần tử của nó là các điểm ranh giới giữa hai khoảng chia liên tiếp. Ví dụ, nếu tín hiệu có miền xác định là R , được phân hoạch thành các khoảng $(-\infty, 0]$, $(0, 2]$, $(2, 4]$ và $(4, +\infty)$ thì có thể biểu diễn sự phân hoạch này bằng vector:

```
>> partition = [0,2,4];
```

Tương ứng với vector phân hoạch tín hiệu là vector biểu diễn bộ mã tín hiệu. Các phần tử của nó là các giá trị thay thế trong mỗi khoảng chia tương ứng của phân hoạch. Nếu ta thay thế các giá trị trong khoảng $(-\infty, 0]$ bằng -1, các giá trị trong khoảng $(0, 2]$ bằng 1, các giá trị trong khoảng $(2, 4]$ bằng 3 và các giá trị trong khoảng $(4, +\infty]$ bằng 5 thì vector biểu diễn bộ mã sẽ là:

```
>> codebook = [-1,1,3,5];
```

Để thực hiện quá trình lượng tử hoá, MATLAB cung cấp hàm **quantiz**:

```
>> [indx, quant, distor] = quantiz(sig, partition, codebook)
```

trong đó **sig** là tín hiệu trước khi lượng tử và **partition**, **codebook** lần lượt là phân hoạch và bộ mã lượng tử.

Hàm **quantiz** trả về:

- o tín hiệu sau khi lượng tử hoá quant
- o vector **indx** có chiều dài bằng chiều dài tín hiệu, mỗi phần tử của nó là chỉ số của giá trị tương ứng của tín hiệu trong bộ mã lượng tử. Nói cách khác, vector **indx** cho biết mỗi phần tử của tín hiệu vào nằm trong khoảng chia nào. Giữa các vector **indx**, **quant** và **codebook** có mối liên hệ:

```
quant = codebook(indx+1);
```

- o nhiễu lượng tử do phép lượng tử hoá này gây ra. Nhiễu lượng tử là sai số bình phương trung bình giữa tín hiệu gốc và tín hiệu sau khi lượng tử **distor**

Ví dụ:

```
>> parti=[0,2,4];
```

```
>> codebook=[-1,1,3,5];
```

```
>> samp=[-3,-1,1,1,5,3,5,4,2,1];
```

```
>> [index,quantized]=quantiz(samp,parti,codebook);
```

```
>> index'
```

cho biet thu tu muc luong tu

```
ans =
```

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

```
>> quantized
quantized =
    -1    -1     1     1     5     3     5     3     1     1
```

Trong các hệ thống thông tin số, người ta thường phân hoạch miền giá trị của tín hiệu thành q khoảng bằng nhau với $q = 2^v$. Các giá trị thay thế chính là điểm giữa của mỗi khoảng chia. Mỗi giá trị lượng tử này lại được biểu diễn bằng một từ mã nhị phân có chiều dài v bit.

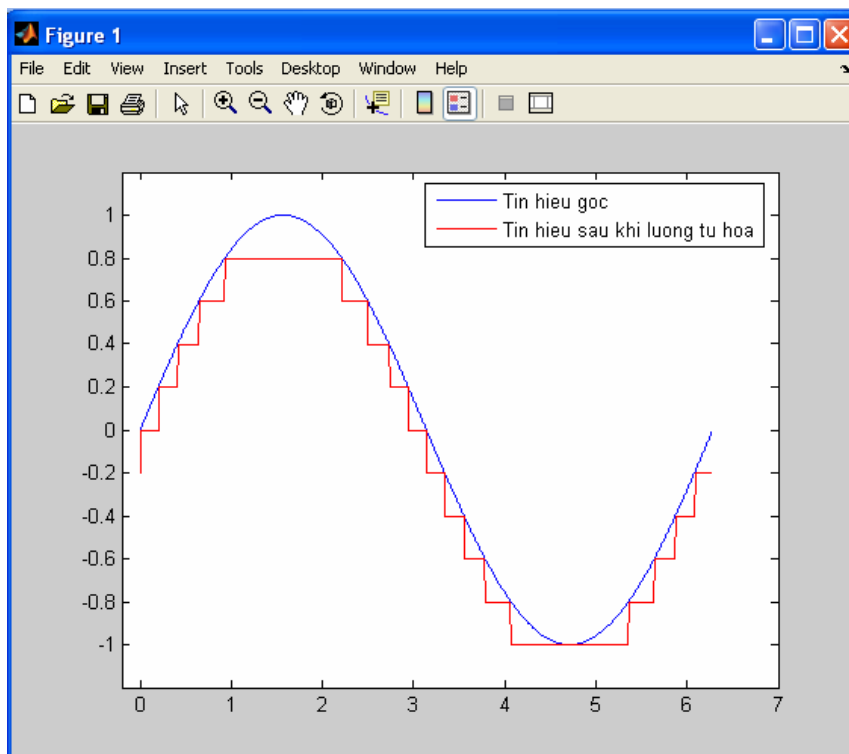
Ví dụ sau đây sẽ minh họa rõ hơn quá trình lượng tử hoá một tín hiệu liên tục.

Ví dụ 14-2. Thực hiện lượng tử hoá tín hiệu $x(t) = \sin t$ với số mức lượng tử là 12, độ dài các khoảng chia bằng nhau. Vẽ tín hiệu trước và sau khi lượng tử.

Tín hiệu $x(t) = \sin t$ có miền giá trị là $[-1,1]$, do đó ta chia miền giá trị này thành các khoảng chia có độ dài 0.2.

```
t = [0:.1:2*pi]; % Các thời điểm lấy mẫu tín hiệu sine
sig = sin(t); % Tín hiệu sine chưa lượng tử
partition = [-1:.2:1]; % Phân hoạch thành 12 khoảng chia
codebook = [-1.2:.2:1]; % Bộ mã lượng tử gồm 12 mức
[index,quants] = quantiz(sig,partition,codebook); % Lượng tử hoá.
plot(t,sig,'x',t,quants,'.')
legend('Tín hiệu gốc','Tín hiệu sau khi lượng tử hoá');
axis([-2 7 -1.2 1.2])
```

Kết quả được minh họa trong hình dưới:



Hình 14.3.

14.3. TỐI ƯU HÓA CÁC THÔNG SỐ CỦA QUÁ TRÌNH LƯỢNG TỬ

Quá trình lượng tử hoá làm phát sinh sai số giữa tín hiệu lượng tử hoá với tín hiệu thực. Sai số này gọi là nhiễu lượng tử. Ta có thể đánh giá nhiễu lượng tử bằng hàm **quantiz** đã đề cập ở phần trước. Vấn đề đặt ra ở đây là làm cách nào để tối thiểu hoá loại nhiễu này. Nhiễu lượng tử phụ thuộc vào cách phân hoạch và tập các giá trị lượng tử được chọn. Nếu số khoảng chia càng nhiều thì nhiễu lượng tử càng nhỏ. Tuy nhiên, giải pháp này không phải lúc nào cũng thực hiện được do những hạn chế khách quan của hệ thống. Một phương pháp đơn giản hơn để tối thiểu hoá nhiễu lượng tử ở mức chấp nhận được mà không cần phải phân hoạch tín hiệu thành quá nhiều khoảng, đó là phương pháp tối ưu hoá theo giải thuật Lloyd. Giải thuật này dựa trên nguyên tắc huấn luyện, nghĩa là các thông số của quá trình lượng tử sẽ được chọn và cập nhật một cách thích nghi với từng loại tín hiệu đưa vào lượng tử.

Quá trình tối ưu hoá nói trên sẽ được thực hiện bằng hàm **lloyds** của MATLAB Communications Toolbox.

```
>> [partition, codebook] = lloyds(training_set, ini_codebook, tol)
```

hoặc:

```
>> [partition, codebook, distortion, rel_distortion] = lloyds(...)
```

`training_set` là tập dữ liệu dùng để huấn luyện. Đó là một tín hiệu tiêu biểu của nguồn tin tức mà ta cần lượng tử hoá; `ini_codebook` là bộ mã khởi đầu do ta chọn; `tol` là sai số cho phép của quá trình huấn luyện (giá trị mặc định nếu không nhập thông số này là 10^{-7}).

Các kết quả xuất ra gồm phân hoạch và bộ mã đã tối ưu hoá (`partition` và `codebook`), nhiễu lượng tử tuyệt đối (`distortion`) và tương đối (`rel-distortion`) của quá trình lượng tử.

Ví dụ sau đây minh hoạ sự tối ưu hoá các thông số của quá trình lượng tử bằng cách dùng hàm **lloyds** trong MATLAB. Nhiễu lượng tử phát sinh từ hai quá trình lượng tử hoá chưa tối ưu và đã tối ưu sẽ được so sánh với nhau.

 **Ví dụ 14-3.** Thực hiện lại quá trình lượng tử hoá tín hiệu $x(t) = \sin t$ với các thông số lượng tử đã tối ưu hoá. So sánh nhiễu lượng tử trước và sau khi tối ưu hoá.

```
% Bắt đầu với các thông số lượng tử như ở ví dụ 14-2
t = [0:.1:2*pi];
sig = sin(t);
partition = [-1:.2:1];
codebook = [-1.2:.2:1];
% Thực hiện tối ưu hoá với bộ mã khởi đầu là codebook.
[partition2,codebook2] = lloyds(sig,codebook);
[index,quants,distor] = quantiz(sig,partition,codebook);
[index2,quant2,distor2] = quantiz(sig,partition2,codebook2);
% So sánh nhiễu lượng tử trước và sau khi tối ưu hoá
[distor, distor2]
```

Kết quả xuất ra của sổ lệnh của MATLAB:

```
ans =
    0.0148    0.0024
```

Sau khi tối ưu hoá, nhiễu lượng tử đã giảm đi: $0.0148/0.0024 = 6.2$ lần.

14.4. ĐIỀU CHẾ MÃ XUNG VI SAI DPCM (DIFFERENTIAL PULSE CODE MODULATION)

Như trên đã đề cập, thông thường người ta sẽ phân hoạch miền xác định của tín hiệu thành 2^v khoảng, sau đó mỗi khoảng tín hiệu sẽ được lượng tử hoá, sau đó mã hoá bằng một từ mã nhị phân có chiều dài v bit. Phương pháp lượng tử hoá này được gọi là phương pháp điều mã xung (*Pulse Code Modulation*). Phương pháp này không cần đòi hỏi bất kỳ thông tin nào về tín hiệu ở các thời điểm trước đó. Trong thực tế, vì tín hiệu thường thay đổi chậm từ thời điểm lấy mẫu này sang thời điểm lấy mẫu kế tiếp nên nếu ta thực hiện lượng tử và mã hoá các giá trị sai biệt giữa thời điểm hiện tại với thời điểm trước đó thì sẽ tốn ít giá trị hơn so với mã hoá đầy đủ độ lớn của tín hiệu. Trên cơ sở này, ta có một phương pháp lượng tử hoá mới, gọi là *lượng tử hoá tiên đoán*, trong đó giá trị của tín hiệu ở thời điểm hiện tại sẽ được tính thông qua một số các giá trị của tín hiệu ở các thời điểm quá khứ. Tiêu biểu cho loại lượng tử hoá này là kỹ thuật điều mã xung vi sai (DPCM – Differential Pulse Code Modulation).

Để thực hiện mã hoá DPCM, ta không những phải xác định sự phân hoạch và bộ mã lượng tử mà còn phải xác định *hàm dự đoán*, để dự đoán giá trị của tín hiệu ở thời điểm hiện tại. Thông thường, người ta sử dụng hàm dự đoán tuyến tính:

$$y(k) = p(1)x(k-1) + p(2)x(k-2) + \dots + p(m-1)x(k-m+1) + p(m)x(k-m) \quad (14.1)$$

trong đó x là tín hiệu gốc còn $y(k)$ là giá trị dự đoán của $x(k)$; p là một vector gồm các hằng số thực.

Thay vì lượng tử hoá tín hiệu x , ta sẽ thực hiện lượng tử hoá tín hiệu *sai số dự đoán* $y - x$.

m được gọi là bậc dự đoán. Trường hợp đặc biệt $m = 1$ được gọi là *điều chế delta*.

Trong MATLAB Communications Toolbox, hàm dự đoán được sử dụng là hàm dự đoán tuyến tính như trên và được biểu diễn bằng vector:

```
>> predictor = [0, p(1), p(2), p(3), ..., p(m-1), p(m)]
```

Các hàm **dpcmenco** và **dpcmdeco** sẽ lần lượt thực hiện quá trình mã hoá và giải mã DPCM.

```
>> [indx, quant] = dpcmenco(sig, partition, codebook, predictor)
```

$quant$ là vector chứa các giá trị lượng tử còn $indx$ là vector chứa các chỉ số tương ứng trong bộ mã.

```
>> [sig, quant] = dpcmdeco(indx, codebook, predictor)
```

sig là tín hiệu tin tức được khôi phục còn $quant$ là tín hiệu sai số dự đoán.

 **Ví dụ 14-4.** Thực hiện mã hoá DPCM đối với tín hiệu sóng răng cưa, sử dụng hàm dự đoán $y(k) = x(k-1)$ (điều chế delta). Giải mã và phục hồi tín hiệu ban đầu. Đánh giá sai số bình phương trung bình giữa tín hiệu gốc và tín hiệu khôi phục.

```
predictor = [0 1]; % Hàm dự đoán y(k)=x(k-1)
partition = [-1:.1:.9]; % Sai số chỉ nằm trong khoảng [-1,1]
codebook = [-1:.1:1];
t = [0:pi/50:2*pi];
x = sawtooth(3*t); % Tín hiệu gốc
% Lượng tử hoá tín hiệu x theo phương pháp DPCM.
encodedx = dpcmenco(x, codebook, partition, predictor);
```

```
% Khôi phục tín hiệu x từ tín hiệu đã điều chế.
decodedx = dpcmdeco(encodedx,codebook,predictor);
plot(t,x,t,decodedx,'--')

legend('Tín hiệu gốc','Tín hiệu sau khi giải ma','Location','NorthOutside');

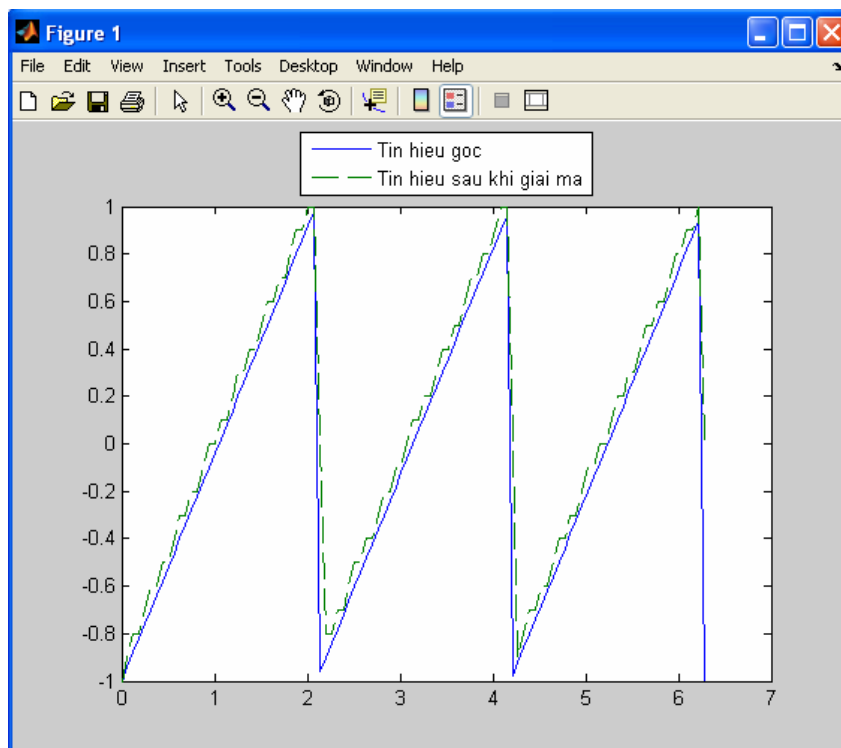
distor = sum((x-decodedx).^2)/length(x) % Sai số bình phương trung bình
```

Kết quả thực hiện:

- Trên cửa sổ lệnh của MATLAB:

```
>> distor =
    0.0327
```

- Đồ thị dạng sóng tín hiệu:



Hình 14.4.

14.5. TỐI ƯU HOÁ CÁC THÔNG SỐ CỦA QUÁ TRÌNH MÃ HÓA DPCM

Cũng giống như với phương pháp lượng tử hoá PCM, ta có thể tối ưu hoá các thông số mã hoá DPCM bằng giải thuật huấn luyện, chỉ cần cung cấp dữ liệu huấn luyện. Quá trình tối ưu hoá này được thực hiện bằng hàm **dpcmopt** của MATLAB với một trong các dạng sau:

```
>> predictor = dpcmopt(training_set,ord)
```

Kết quả trả về là vector biểu diễn hàm dự đoán tối ưu từ tập dữ liệu huấn luyện và bậc dự đoán cho trước.

```
>> [predictor,codebook,partition] = dpcmopt(training_set,ord,length)
```

```
>> [predictor,codebook,partition] = dpcmopt(training_set,ord,ini_codebook)
```

Nếu cung cấp thêm chiều dài **length** của bộ mã **codebook** hoặc bộ mã khởi đầu **ini_codebook** thì hàm sẽ trả về phân hoạch và bộ mã tối ưu.

 **Ví dụ 14-5.** Thực hiện lại quá trình mã hoá như ví dụ 14-4 nhưng sử dụng các thông số mã hoá tối ưu, biết bậc dự đoán bằng 1 và bộ mã khởi đầu $[-1:1:1]$

Ta sử dụng hàm **dpcmopt** để tìm các thông số tối ưu trước khi mã hoá. Lưu ý rằng nhiều lượng tử sinh ra trong ví dụ này nhỏ hơn nhiều so với ví dụ trước.

```
t = [0:pi/50:2*pi];
x = sawtooth(3*t); % Tín hiệu gốc
initcodebook = [-1:1:1]; % Bộ mã khởi đầu
% Tối ưu hoá các thông số DPCM từ bộ mã khởi đầu và bậc dự đoán bằng 1.
[predictor, codebook, partition] = dpcmopt(x, 1, initcodebook);
% Lượng tử hoá tín hiệu x theo phương pháp DPCM.
encodedx = dpcmenco(x, codebook, partition, predictor);
% Phục hồi tín hiệu x từ tín hiệu đã điều chế.
decodedx = dpcmdeco(encodedx, codebook, predictor);
distor = sum((x-decodedx).^2)/length(x) % Sai số bình phương trung bình
```

Sai số lượng tử ở ví dụ này là:

```
distor =
    0.0063
```

14.6. NÉN VÀ GIÃN TÍN HIỆU

Trong các ứng dụng xử lý tín hiệu thoại (speech processing), trước khi lượng tử hoá, người ta thường thực hiện nén (compress) tín hiệu theo hàm logarithm, mục đích là để tín hiệu ở mức biên độ nhỏ sẽ thay đổi nhiều mức hơn so với ở các giá trị biên độ lớn, do đó sai số lượng tử tương đối ở các mức biên độ nhỏ và lớn sẽ không chênh lệch nhau nhiều như đối với trường hợp không nén.

Để khôi phục lại đúng tín hiệu ban đầu thì sau khi giải mã, ta phải đưa qua một bộ giãn tín hiệu (expander) có đặc tuyến truyền đạt là nghịch đảo của đặc tuyến của bộ nén (compressor). Sự kết hợp của bộ nén và bộ giãn tín hiệu gọi chung là bộ nén giãn tín hiệu (compander).

Hai luật nén giãn thường được sử dụng trong xử lý tín hiệu thoại là luật μ dùng ở Bắc Mỹ và luật A dùng ở châu Âu:

○ Luật μ :
$$y = y_{\max} \frac{\ln[1 + \mu(|x|/x_{\max})]}{\ln(1 + \mu)} \operatorname{sgn} x \quad (14.2)$$

Theo các chuẩn ở Bắc Mỹ, giá trị của μ là 255. $x_{\max}$ và $y_{\max}$ lần lượt là các giá trị dương lớn nhất của x và y .

○ Luật A:
$$y = \begin{cases} y_{\max} \frac{A(|x|/x_{\max})}{1 + \ln A} \operatorname{sgn} x & 0 < \frac{|x|}{x_{\max}} \leq \frac{1}{A} \\ y_{\max} \frac{1 + \ln[A(|x|/x_{\max})]}{1 + \ln A} \operatorname{sgn} x & \frac{1}{A} < \frac{|x|}{x_{\max}} \leq 1 \end{cases} \quad \begin{matrix} A \text{ là hằng số} \\ (14.3) \end{matrix}$$

Giá trị chuẩn của A là 87.6.

MATLAB cung cấp hàm **compand** để thực hiện nén giãn tín hiệu. Hàm này hỗ trợ hai luật nén giãn A và μ nói trên.

```
>> out = compand(in, param, v, method)
```

in là tín hiệu vào còn out là tín hiệu ra

V là biên độ đỉnh của tín hiệu vào

param là thông số của luật nén giãn (hằng số A hoặc μ)

method có thể nhận một trong các giá trị sau:

method	Chức năng
'mu/compressor'	Nén theo luật μ
'mu/expander'	Giãn theo luật μ
'A/compressor'	Nén theo luật $A\mu$
'A/expander'	Giãn theo luật A

Ví dụ sau đây minh hoạ quá trình nén giãn theo luật μ .

 **Ví dụ 14-6.** Thực hiện quá trình lượng tử hoá tín hiệu hàm mũ theo hai phương pháp: lượng tử hoá đều và lượng tử hoá có nén giãn theo luật μ . So sánh sai số bình phương trung bình của hai phương pháp.

Với phương pháp đầu tiên ta sử dụng trực tiếp hàm **quantiz** đối với tín hiệu lũy thừa. Tín hiệu được phân hoạch thành các khoảng đều nhau có chiều dài bằng 1.

Với phương pháp thứ hai, đầu tiên ta sử dụng hàm **compand** để nén tín hiệu theo luật μ , sau đó dùng hàm **quantiz** để lượng tử hoá tín hiệu thu được và cuối cùng lại dùng hàm **compand** để phục hồi tín hiệu ban đầu.

Kết quả xuất ra gồm sai số bình phương trung bình của hai phương pháp và đồ thị biểu diễn tín hiệu gốc và tín hiệu đã được nén. Có thể nhận thấy rằng sai số lượng tử ở phương pháp thứ hai nhỏ hơn so với phương pháp thứ nhất.

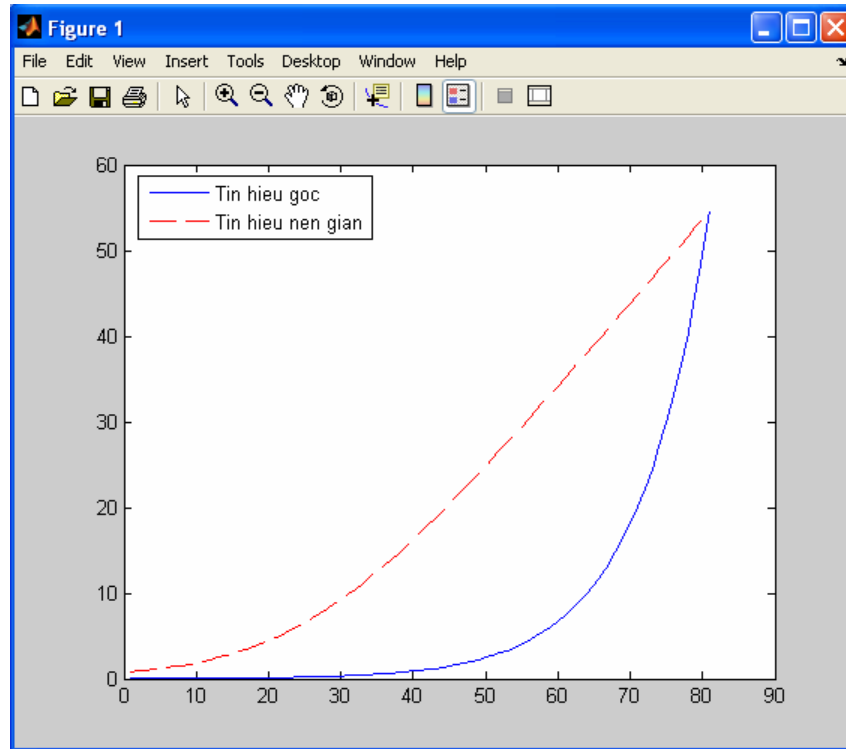
```
Mu = 255; % Thông số cho bộ nén giãn theo luật  $\mu$ 
sig = -4:.1:4;
sig = exp(sig); % Tín hiệu hàm mũ cần lượng tử
V = max(sig);
% 1. Lượng tử hoá đều (không nén giãn).
[index,quants,distor] = quantiz(sig,0:floor(V),0:ceil(V));
% 2. Sử dụng cùng phân hoạch và bộ mã lượng tử nhưng có nén trước khi lượng tử
% hoá và giải nén sau đó
compsig = compand(sig,Mu,V,'mu/compressor');
[index,quants] = quantiz(compsig,0:floor(V),0:ceil(V));
newsig = compand(quants,Mu,max(quants),'mu/expander');
distor2 = sum((newsig-sig).^2)/length(sig);
[distor, distor2] % Hiển thị sai số lượng tử của hai phương pháp.
plot(sig); % Vẽ tín hiệu gốc.
hold on;
plot(compsig,'r--'); % Vẽ tín hiệu đã nén giãn.
```

```
legend('Tin hieu goc','Tin hieu nen gian','Location','NorthWest')
```

Sai số lượng tử:

```
ans =  
0.5348    0.0397
```

Đồ thị dạng tín hiệu:



Hình 14.5.

14.7. MÃ HÓA HUFFMAN

Phương pháp mã hoá Huffman là một kỹ thuật dùng để nén dữ liệu. Một nguồn dữ liệu thường bao gồm nhiều ký hiệu, mỗi ký hiệu có xác suất xuất hiện khác nhau. Nếu chúng ta mã hoá các ký hiệu này bằng các từ mã có độ dài bằng nhau thì sẽ không hiệu quả bằng cách mã hoá sau cho các ký hiệu có xác suất xuất hiện thấp được biểu diễn bởi từ mã có chiều dài lớn hơn so với các từ mã biểu diễn ký hiệu có xác suất xuất hiện cao hơn. Đó là nguyên tắc của phương pháp mã hoá Huffman. Như vậy chiều dài trung bình của các từ mã Huffman sẽ phụ thuộc vào xác suất xuất hiện (hay còn gọi là tần số thống kê) của các ký hiệu trong tập dữ liệu nguồn. Điều kiện quan trọng đối với các từ mã trong bộ mã Huffman là không có một từ mã nào của bộ mã là phần đầu của một từ mã khác trong bộ mã (ví dụ: không thể tồn tại đồng thời hai từ mã 01 và 0110 trong cùng một bộ mã Huffman).

Lưu ý: Trong trường hợp một chuỗi dữ liệu dài cấu tạo bởi tập ký hiệu có ít phần tử và có sự phân bố lệch nhau nhiều thì phương pháp mã hoá số học sẽ có hiệu quả hơn phương pháp mã hoá Huffman.

- Để thành lập một bộ mã Huffman, MATLAB có hàm **huffmandict**, ta chỉ cần cung cấp các thông tin về tính chất thống kê của tập dữ liệu nguồn:

```
>> [dict,avlen] = huffmandict(sym,prob,N,var)
```

Hàm này trả về bộ mã Huffman **dict** cùng với chiều dài trung bình **avlen** của các từ mã của bộ mã. Các thông số nhập vào tối thiểu gồm có tập các ký hiệu của nguồn dữ liệu (**sym**)

và xác suất xuất hiện của từng ký hiệu trong tập ký hiệu này (cho bởi vector `prob`). Ngoài ra còn có thể cung cấp thêm hai thông số nhập khác:

- Nếu dùng các từ mã N-ary thay vì mã nhị phân thì ta phải nhập thêm giá trị N. Lưu ý là N là một số nguyên có giá trị từ 2 đến 10 và không được vượt quá số ký hiệu của tập nguồn.
- Một cách mặc định, MATLAB sẽ dùng giải thuật variance lớn nhất. Nếu chọn giải thuật variance cực tiểu, ta phải cung cấp thêm thông số `var` với giá trị là `'min'`.

Sau đây là một ví dụ đơn giản để minh họa cách sử dụng hàm **huffmandict**:

```
symbols = [1:5] % Tập ký hiệu
prob = [.3 .3 .2 .1 .1] % Xác suất xuất hiện tương ứng
[dict,avglen] = huffmandict(symbols,prob);
% Hiện thị bảng mã.
temp = dict;
for i = 1:length(temp)
temp{i,2} = num2str(temp{i,2});
end
avglen
temp
```

Kết quả hiển thị như sau:

```
avglen =
    2.2000
temp =
    [1]      '0  1'
    [2]      '0  0'
    [3]      '1  0'
    [4]      '1  1  1'
    [5]      '1  1  0'
```

▪ Để thực hiện mã hoá và giải mã theo phương pháp Huffman, ta có thể sử dụng các hàm **huffmanenco** và **huffmandeco**:

```
>> sig_encode = huffmanenco(sig,dict)
```

Hàm này thực hiện mã hoá Huffman cho chuỗi dữ liệu `sig` với bộ mã Huffman tương ứng là `dict`. Bộ mã này được tạo từ hàm **huffmandict**.

```
>> sig = huffmandeco(comp,dict)
```

Hàm này thực hiện quá trình giải mã với `comp` là dữ liệu đã được mã hoá.

 **Ví dụ 14-7.** Thực hiện mã hoá và giải mã theo phương pháp Huffman cho một chuỗi dữ liệu cho trước.

Trong ví dụ này, giả sử ta đã có một chuỗi dữ liệu cần mã hoá. Ta sẽ thực hiện đếm số ký hiệu trong tập nguồn và xác suất xuất hiện tương ứng. Từ đó chọn bộ mã Huffman bằng cách dùng hàm **huffmandict** và thực hiện mã hoá và giải mã dùng các hàm **huffmanenco**, **huffmandeco**.

Để đơn giản hoá các vấn đề không cần thiết, chúng ta sẽ tạo ra một chuỗi dữ liệu để mã hoá thay vì lấy chuỗi dữ liệu ngẫu nhiên. Chuỗi dữ liệu này gồm có 3 ký hiệu. Ta sẽ dùng hàm **repmat** để tăng chiều dài chuỗi dữ liệu nhưng vẫn giữ nguyên xác suất xuất hiện các ký hiệu (hàm **repmat** sẽ tạo một ma trận mới bằng cách lặp lại nhiều lần một ma trận có kích thước nhỏ hơn).

```
% Dữ liệu mã hoá (lặp lại 50 lần đoạn [2 1 4 2 1 1 5 4 3 1])
sig = repmat([2 1 4 2 1 1 5 4 3 1],1,50);
symbols = [1 2 3 4 5]; % Tập các ký hiệu có trong dữ liệu cần mã hoá
p = [0.4 0.2 0.1 0.2 0.1]; % Xác suất ký hiệu tương ứng của các ký hiệu
dict = huffmandict(symbols,p); % Thành lập bộ mã.
hcode = huffmanenco(sig,dict); % Mã hoá dữ liệu.
dhsig = huffmandeco(hcode,dict); % Giải mã.
dhsig(1:10) % Hiển thị 10 ký hiệu đầu tiên của chuỗi dữ liệu đã được
            % giải mã
```

14.8. MÃ HÓA SỐ HỌC (ARITHMETIC CODING)

Đây là một phương pháp khác để nén dữ liệu. Phương pháp này thích hợp cho các nguồn dữ liệu có số ký hiệu ít và xác suất xuất hiện các ký hiệu chênh lệch lớn. Mã hoá số học đòi hỏi phải cung cấp tính chất thống kê của tập dữ liệu nguồn. Cụ thể, đối với MATLAB, ta phải cung cấp số lần xuất hiện của từng ký hiệu trong tập dữ liệu nguồn. Ví dụ, để mã hoá một nguồn dữ liệu gồm 10 ký hiệu x, 10 ký hiệu y và 80 ký hiệu z, ta phải định nghĩa vector counts chỉ ra số lần xuất hiện của 3 ký hiệu x, y, z như sau:

```
>> counts = [10,10,80];
```

Khi đã xác định rõ số lần xuất hiện của các phần tử trong vector counts nói trên, ta có thể thực hiện mã hoá và giải mã bằng các hàm sau:

```
>> code = arithenco(seq, counts)
>> dseq = arithdeco(code, counts, len)
```

code là chuỗi dữ liệu đã được mã hoá còn dseq là chuỗi dữ liệu được khôi phục với số ký hiệu khôi phục được xác định bởi len.

 **Ví dụ 14-8.** Thực hiện mã hoá và giải mã theo phương pháp số học cho một chuỗi dữ liệu gồm 10 ký hiệu x, 10 ký hiệu y và 80 ký hiệu z.

```
seq = repmat([3 3 1 3 3 3 3 3 2 3],1,50); % x: 1, y: 2, z: 3
counts = [10 10 80];
code = arithenco(seq,counts);
dseq = arithdeco(code,counts,length(seq));
```

Bài tập 14-1.

Tạo một chuỗi dữ liệu ngẫu nhiên lấy từ tập $M = 8$ ký hiệu. Vẽ tín hiệu biểu diễn chuỗi dữ liệu trên dùng:

- Tín hiệu nhị phân (2 mức 0 và 1)
- Tín hiệu 8 mức

Bài tập 14-2.

Thực hiện điều chế PCM cho tín hiệu $x(t) = \sin t$. Vẽ chuỗi xung phát đi. Sau đó tiến hành giải mã khôi phục lại tín hiệu ban đầu. Vẽ tín hiệu gốc và tín hiệu khôi phục. Đánh giá sai số bình phương trung bình.

✎ **Bài tập 14-3.**

Làm lại bài tập trên nhưng sử dụng nén giãn theo luật A và luật μ . Đánh giá sai số bình phương trung bình trong ba trường hợp: không nén giãn, nén giãn theo luật A và theo luật μ

✎ **Bài tập 14-4.**

Thực hiện mã hoá và giải mã DPCM cho tín hiệu $x(t) = \sin t + 3\cos t$ với bậc dự đoán bằng 2 với các thông số lượng tử được tối ưu hoá. Đánh giá sai số lượng tử.

✎ **Bài tập 14-5.**

Thực hiện mã hoá thông điệp sau đây “Welcome to Vietnam” bằng ba phương pháp:

- Mã ASCII
- Mã Huffman
- Mã số học

So sánh chiều dài trung bình của các từ mã trong ba phương pháp trên.

✎ **Bài tập 14-6.**

Thực hiện một hệ thống thông tin đơn giản: truyền một thông điệp “Hello world” từ máy phát đến máy thu bao gồm các bước sau:

- Mã hoá thông điệp bằng mã Huffman (từ mã nhị phân)
- Điều chế bằng phương pháp BPSK
- Phát đi trên kênh truyền có nhiễu AWGN
- Giải điều chế BPSK
- Giải mã Huffman và thu lại thông điệp ban đầu

Lần lượt khảo sát với các giá trị E_b/N_0 bằng 0, 2, 4, 6, 8 dB và so sánh thông điệp nhận được với thông điệp phát. Vẽ đồ thị BER.

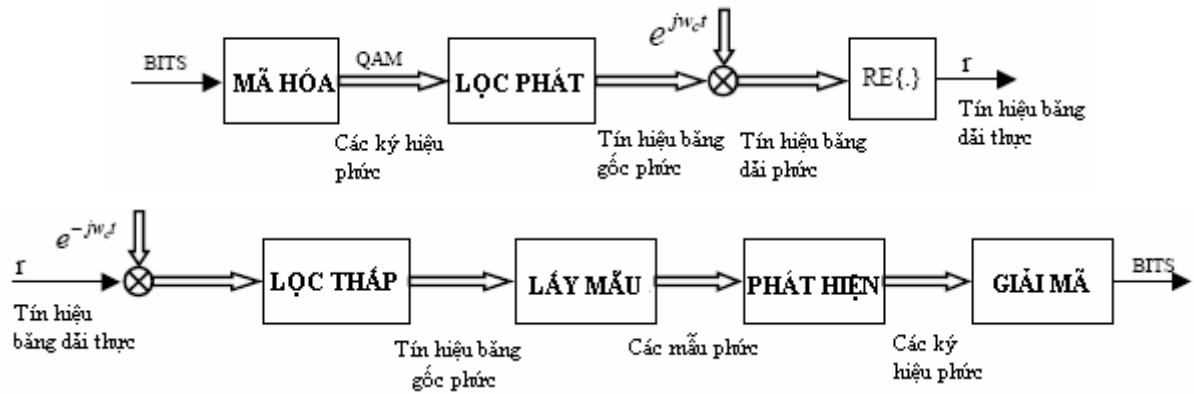
✎ **Bài tập 14-7.**

Thực hiện yêu cầu tương tự như bài tập 14-6 với tín hiệu là tín hiệu $x = \cos(2\pi \cdot 250 \cdot t)$ (t tính bằng s):

- Điều mã xung PCM 4 bit
- Điều chế QPSK
- Truyền trên kênh truyền có nhiễu AWGN với $E_b/N_0 = 4\text{dB}$
- Giải điều chế QPSK
- Giải điều chế PCM để phục hồi lại tín hiệu ban đầu

Vẽ tín hiệu tại mỗi bước. Đánh giá sai số bình phương trung bình.

✎ **Bài tập 14-8.**



Hình 14.6.

Cho hệ thống 16-QAM như hình 14.6. Thông điệp vào được nhập từ một file **input.txt**. Dùng mã hóa Huffman và PCM. Bộ lọc sửa dạng xung là bộ lọc raised-cosine chuẩn (dùng hàm **rcosine**). Kênh truyền có nhiễu fading Rayleigh với SNR = 15dB. Sau khi giải mã thông điệp ở máy thu, lưu vào file **output.txt**. Viết chương trình mô phỏng hệ thống trên.

Danh sách các hàm được giới thiệu trong chương 14**Các hàm tạo nguồn tín hiệu**

randint	Tạo ma trận các số nguyên ngẫu nhiên
randsrc	Tạo ma trận các giá trị ngẫu nhiên lấy từ một tập nguồn với xác suất cho trước

Lượng tử hoá tín hiệu

quantiz	Lượng tử hoá tín hiệu
lloyd	Tối ưu hoá các thông số của quá trình lượng tử

Mã hoá nguồn

dpcmdeco	Giải mã DPCM (điều chế mã xung vi sai)
dpcmenco	Mã hoá DPCM (điều chế mã xung vi sai)
dpcmopt	Tối ưu hoá các thông số của quá trình mã hoá DPCM

Nén và giãn tín hiệu

arithenco	Mã hoá số học
arithdeco	Giải mã hoá số học
compand	Nén và giãn tín hiệu theo luật A và luật μ
huffmandeco	Giải mã Huffman
huffmandict	Thành lập bộ mã Huffman
huffmanenco	Mã hoá Huffman
repmat	Tạo ma trận mới bằng cách lặp lại nhiều lần một ma trận

Chương 15

TRUYỀN DẪN BASEBAND VÀ PASSBAND

Một vấn đề cơ bản trong các hệ thống thông tin là vấn đề điều chế. MATLAB cung cấp đầy đủ các hàm để thực hiện các phương pháp điều chế khác nhau (kể cả điều chế số lẫn điều chế tương tự) và một số nguồn tín hiệu ngẫu nhiên dùng để mô phỏng quá trình này. Trong chương này chúng ta sẽ tìm hiểu cách điều chế và giải điều chế tín hiệu trong MATLAB.

Vấn đề: Xử lý một chuỗi bit dữ liệu bằng một hệ thống thông tin cơ bản gồm nguồn tín hiệu, bộ điều chế, kênh truyền và bộ giải điều chế.

15.1. ĐIỀU CHẾ TƯƠNG TỰ

■ Để biểu diễn một tín hiệu tương tự, MATLAB dùng các vector hoặc ma trận. Mỗi tín hiệu tương tự được biểu diễn bằng một vector, trong đó mỗi phần tử của vector là giá trị của tín hiệu tại các thời điểm lấy mẫu xác định bởi vector thời gian t . Giả sử tần số lấy mẫu là f_s thì vector thời gian sẽ gồm các phần tử cách đều nhau một khoảng bằng $1/f_s$.

Ví dụ để biểu diễn tín hiệu $x = \sin(20\pi t)$ trong khoảng $[0, 0.1s]$ với tần số lấy mẫu 8kHz, ta viết đoạn mã sau:

```
Fs = 8000; % Tần số lấy mẫu 8000Hz
t = [0:.1*Fs]'/Fs; % Các thời điểm lấy mẫu trong 0.1s
x = sin(20*pi*t); % Biểu diễn tín hiệu
plot(t,x); % Vẽ tín hiệu x.
```

Để biểu diễn các tín hiệu đa kênh, ta sử dụng một ma trận mà mỗi cột (hay mỗi hàng) của nó ứng với một kênh tín hiệu, chẳng hạn trong ví dụ trên nếu biểu diễn tín hiệu 2 kênh, một kênh có pha ban đầu bằng 0, một kênh có pha ban đầu là $\pi/8$, ta viết như sau:

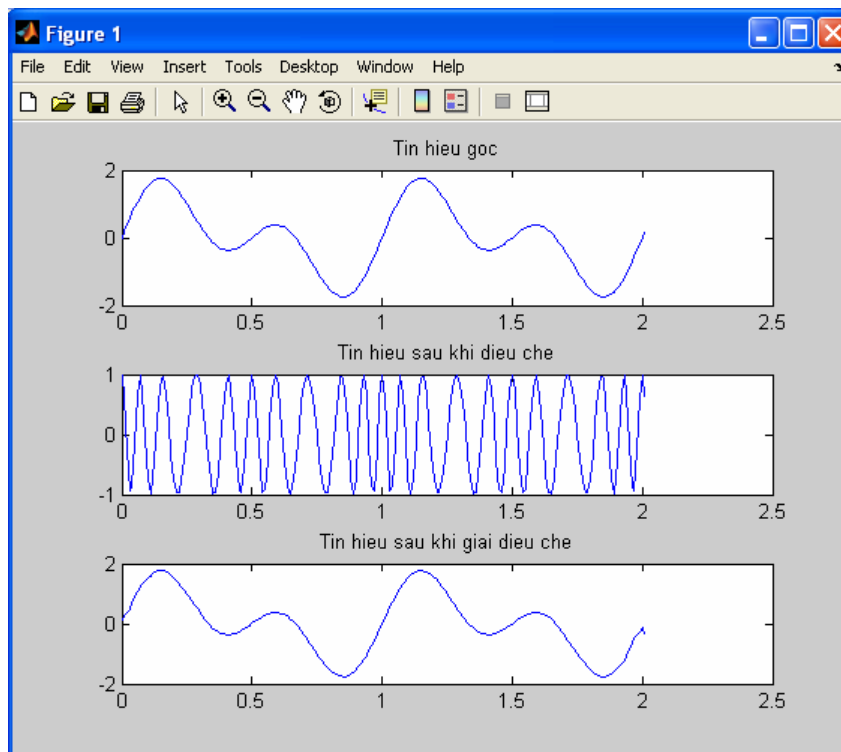
```
Fs = 8000;
t = [0:.1*Fs]'/Fs;
x = [sin(20*pi*t), sin(20*pi*t+pi/8)];
```

■ MATLAB cung cấp sẵn các hàm để thực hiện quá trình điều chế tương tự, chỉ cần cung cấp tín hiệu điều chế, tần số sóng mang và tần số lấy mẫu.

 **Ví dụ 15-1.** Điều chế và giải điều chế tín hiệu $x(t) = \sin(2\pi t) + \sin(4\pi t)$ dùng phương pháp điều chế pha.

```
% Thiết lập các thông số.
Fs = 100; % Tần số lấy mẫu
t = [0:2*Fs+1]'/Fs; % Các thời điểm lấy mẫu
% Khởi tạo một tín hiệu tương tự là tổng của hai tín hiệu sin.
x = sin(2*pi*t) + sin(4*pi*t);
Fc = 10; % Tần số sóng mang
phasedev = pi/2; % Độ di pha
y = pmmmod(x,Fc,Fs,phasedev); % Điều chế.
z = pmdemod(y,Fc,Fs,phasedev); % Giải điều chế.
```

```
% Vẽ các tín hiệu  
figure;  
subplot(3,1,1); plot(t,x); % Vẽ tín hiệu ban đầu.  
title('Tín hiệu gốc');  
subplot(3,1,2); plot(t,y); % Vẽ tín hiệu sau khi điều chế.  
title('Tín hiệu sau khi điều chế');  
subplot(3,1,3); plot(t,z); % Vẽ tín hiệu sau khi giải điều chế.  
title('Tín hiệu sau khi giải điều chế');
```

**Hình 15.1.**

▪ Mặc dù ví dụ trên chỉ minh họa phương pháp điều chế pha, nhưng các phương pháp điều chế khác đều có thể thực hiện theo cách hoàn toàn giống như trên. Bảng sau đây liệt kê các hàm dùng để thực hiện các phương pháp điều chế tương tự:

Bảng 15.1. Các hàm điều chế tương tự

Tên hàm và cú pháp	Chức năng	Giải thích
<code>ammod(x, fc, fs, phase, amp)</code>	Điều chế AM	phase, amp: pha và biên độ sóng mang
<code>amdemod(y, fc, fs, phase, amp, num, den)</code>	Giải điều chế AM	num, den: bộ lọc dùng cho giải điều chế
<code>ssbmod(x, fc, fs, phase, 'upper')</code>	Điều chế SSB	'upper': chọn băng trên (mặc định: băng dưới)
<code>ssbdemod(y, fc, fs, phase, num, den)</code>	Giải điều chế SSB	num, den: bộ lọc giải điều chế (mặc định: <code>butter(5, fc*2/fs)</code>)
<code>fmmmod(x, fc, fs, fdev, phase)</code>	Điều chế FM	fdev: độ di tần của tín hiệu sau điều chế
<code>fmdemod(y, fc, fs, fdev, phase)</code>	Giải điều chế FM	
<code>pmmmod(x, fc, fs, pdev, phase)</code>	Điều chế PM	pdev: độ di pha của tín hiệu sau điều chế
<code>pmdemod(y, fc, fs, pdev, phase)</code>	Giải điều chế PM	

15.2. ĐIỀU CHẾ SỐ

▪ Trước khi thực hiện điều chế số, ta phải biểu diễn nguồn dữ liệu (một tập hợp M ký hiệu rời rạc) thành một tín hiệu mà các giá trị của nó là các số nguyên từ 0 đến M-1. Chẳng hạn, nếu thực hiện điều chế dựa trên một tập gồm 8 ký hiệu, thì nguồn ký hiệu đầu vào phải được biểu diễn dưới dạng một vector cột gồm các số nguyên từ 0 đến 7:

```
>> s = [2 3 4 1 4 7 0 4 5 2]';
```

Hoặc dưới dạng ma trận nếu là tín hiệu đa kênh:

```
>> s = [2 1;
        3 5;
        4 0;
        2 1];
```

▪ Ví dụ sau đây sẽ minh họa các bước thực hiện điều chế số cho một tập ký hiệu cho trước.

 **Ví dụ 15-2.** Thực hiện điều chế 16-QAM với nguồn dữ liệu là chuỗi bit đã tạo trong ví dụ 15-1. Sau đó thực hiện giải điều chế 16-QAM

Để giải quyết bài toán trên, đầu tiên ta phải biểu diễn nguồn dữ liệu vào dưới dạng một vector gồm các số nguyên từ 0 đến 15. Do đó ta phải tách chuỗi bit nhị phân đã tạo ở ví dụ 15-1 thành các nhóm 4 bit (gọi là 4-tuple) rồi chuyển mỗi nhóm này thành một số nguyên tương ứng trong khoảng từ 0 đến 15. Hàm **reshape** cho phép chuyển chuỗi bit thành một ma trận 4 cột, như vậy mỗi hàng của ma trận ứng với 1 nhóm 4 bit. Hàm **bi2de** cho phép chuyển các số nhị phân thành thập phân. Đoạn chương trình sau sẽ thực hiện quá trình xử lý nguồn dữ liệu nêu trên:

```
%% Chuyển đổi bit - ký hiệu
```

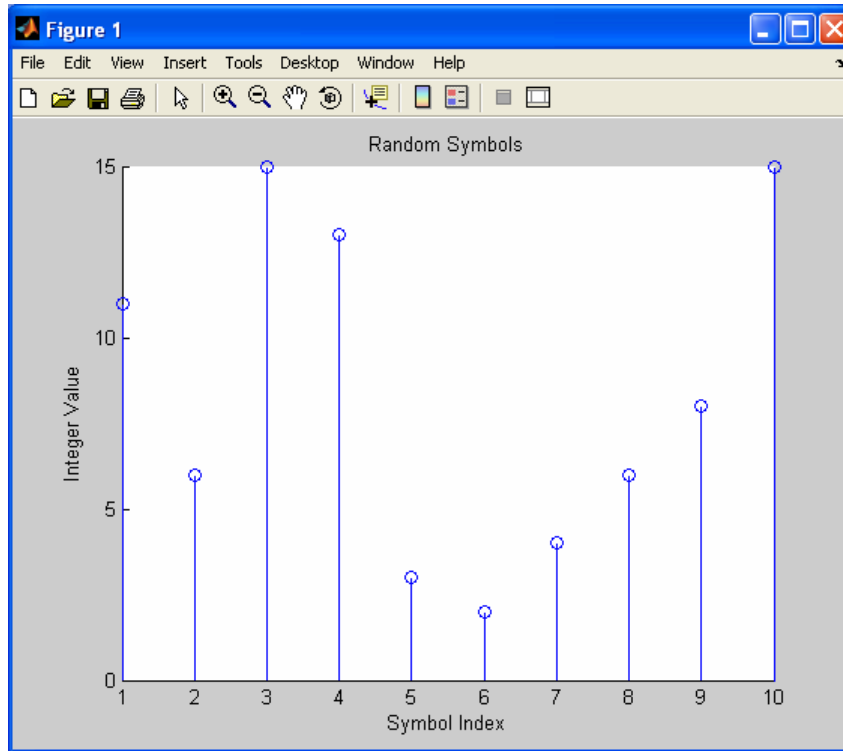
```
% Chuyển chuỗi bit trong vector x thành các ký hiệu k-bit (k=4)
```

```
k = 4; % Chuyển thành các 4-tuple
```

```

xsym = bi2de(reshape(x,k,length(x)/k).','left-msb');
%% Giản đồ xung
% Vẽ các ký hiệu trên giản đồ xung.
figure; % Khởi tạo một cửa sổ đồ họa mới.
stem(xsym(1:10)); % Vẽ giản đồ xung
title('Random Symbols');
xlabel('Symbol Index'); ylabel('Integer Value');

```



Hình 15.2.

Lúc này ta có thể thực hiện điều chế 16-QAM:

```

%% Điều chế số 16-QAM
M = 16;
y = qammod(xsym,M);

```

Sau khi điều chế ta được một vector phức y mà mỗi phần tử của nó là một trong 16 điểm trong mặt phẳng phức. 16 điểm này hình thành giản đồ sao (constellation plot) của phương pháp điều chế QAM. Trong ví dụ 15-4 chúng ta sẽ khảo sát cách vẽ giản đồ này.

Cuối cùng là quá trình giải điều chế. Sau khi giải điều chế ta phải chuyển chuỗi ký hiệu nhận được thành chuỗi bit nhị phân bằng cách dùng hàm **de2bi** và **reshape**.

```

%% Giải điều chế sử dụng phương pháp 16-QAM.
zsym = qamdemod(y,M);
%% Chuyển đổi ngược từ các ký hiệu thành chuỗi bit nhị phân
z = de2bi(zsym,'left-msb'); % Chuyển các số nguyên thành các số nhị phân 4 bit.
% Đổi ma trận z thành một vector
z = reshape(z.',prod(size(z)),1);

```

▪ Trong thí dụ trên, chúng ta thấy rằng hàm **qammod** cũng như các hàm điều chế số khác trong MATLAB đều trả về vector phức, do đó ta chưa thể vẽ được dạng sóng của tín hiệu sau khi điều chế. Sở dĩ như vậy là vì MATLAB sử dụng phương pháp biểu diễn tín hiệu điều chế bằng tín hiệu băng gốc tương đương. Nếu tín hiệu điều chế có dạng:

$$y = Y_1(t) \cos(2\pi f_c t + \theta) - Y_2(t) \sin(2\pi f_c t + \theta) \quad (15.1)$$

trong đó f_c là tần số và θ là pha ban đầu của sóng mang, thì biểu diễn băng gốc tương đương của nó sẽ là:

$$[Y_1(t) + jY_2(t)]e^{j\theta} \quad (15.2)$$

Cách biểu diễn này làm giảm bớt số lượng các phép tính cần thực hiện khi mô phỏng bởi vì nếu biểu diễn bằng tín hiệu thực thì phải thực hiện lấy mẫu ở tốc độ cao hơn tần số sóng mang.

Muốn vẽ dạng sóng tín hiệu sau khi điều chế, ta phải chuyển đổi dạng tín hiệu băng gốc thu được sau khi dùng các hàm điều chế số thành tín hiệu băng dải thực. Các thông số cần có để thực hiện điều này là tốc độ bit và tần số lấy mẫu. Ví dụ sau đây minh họa quá trình này.

 **Ví dụ 15-3.** Vẽ dạng sóng tín hiệu sau khi điều chế ở ví dụ 15-2, giả sử tốc độ ký hiệu bằng 1Kbps, tần số sóng mang bằng 10KHz, tần số lấy mẫu bằng 100KHz.

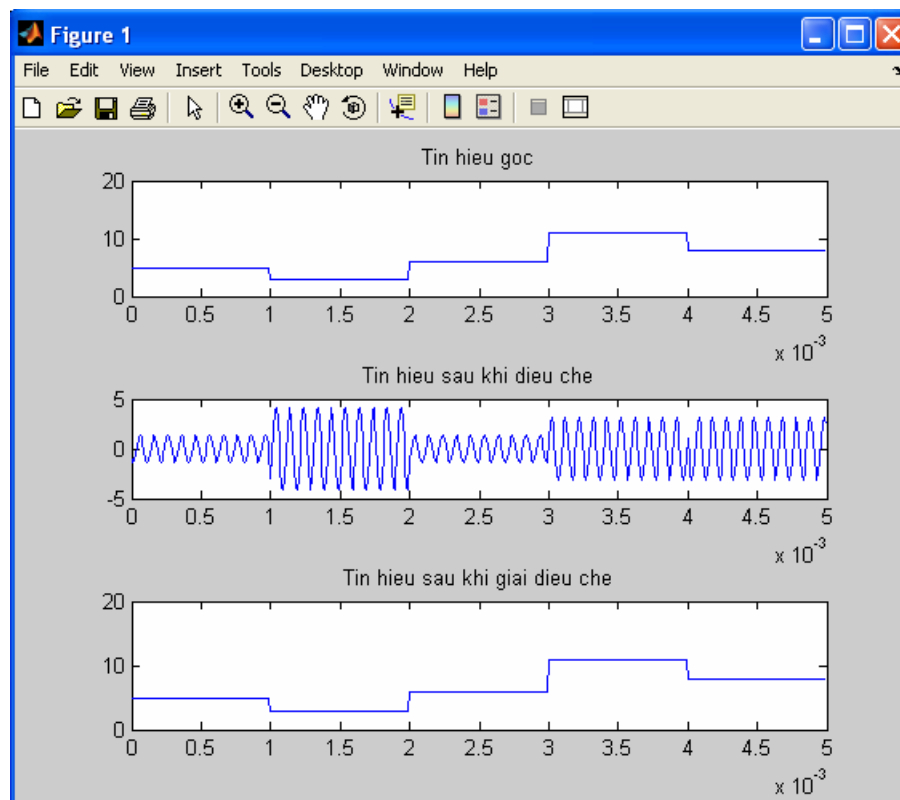
```
% Thiết lập các thông số
fd = 1000; % Tốc độ ký hiệu
fc = 10*fd; % Tần số sóng mang
fs = 10*fc; % Tần số lấy mẫu
len = length(y)*fs/fd; % Số mẫu của tín hiệu y
Yreal = real(y); % Tín hiệu Y1(t) là phần thực của tín hiệu điều chế băng gốc y
Yimag = imag(y); % Tín hiệu Y2(t) là phần ảo của tín hiệu điều chế băng gốc y
% Khởi tạo vector tín hiệu
y1 = [];
y2 = [];
xtmp = [];
ztmp = [];
for k=1:length(y) % Xét từng chu kỳ ký hiệu
    sigx=xsym(k,1)*ones(1,100); % Tạo 100 mẫu giá trị bằng x
    sig1=Yreal(k,1)*ones(1,100); % Tạo 100 mẫu giá trị bằng Yreal
    sig2=Yimag(k,1)*ones(1,100); % Tạo 100 mẫu giá trị bằng Yimag
    sigz=zsym(k,1)*ones(1,100); % Tạo 100 mẫu giá trị bằng z
    y1=[y1 sig1];
    y2=[y2 sig2];
    xtmp = [xtmp sigx];
    ztmp = [ztmp sigz];
end
time = 0:(1/fs):(len-1)/fs; % Vector thời gian
```

```

ymod = y1.*cos(2*pi*fc*time)-y2.*sin(2*pi*fc*time); % Tín hiệu sau điều chế
figure;
subplot(3,1,1); plot(time,xtmp); % Vẽ tín hiệu trước khi điều chế.
title('Tín hiệu gốc');
subplot(3,1,2); plot(time,ymod); % Vẽ tín hiệu sau khi điều chế.
title('Tín hiệu sau khi điều chế');
subplot(3,1,3); plot(time,ztmp); % Vẽ tín hiệu sau khi giải điều chế.
title('Tín hiệu sau khi giải điều chế');

```

Kết quả thực thi chương trình:



Hình 15.3.

- Ngoài phương pháp điều chế QAM được trình bày ở thí dụ trên, MATLAB có sẵn các hàm để thực hiện các phương pháp điều chế số khác như trình bày trong bảng dưới đây:

Bảng 15.2. Các hàm điều chế số của MATLAB

Tên hàm và cú pháp	Chức năng	Giải thích
<code>dpskmod(x,M,phaserot)</code>	Điều chế DPSK	M: số trạng thái của ký hiệu; phaserot: độ xoay pha của phép điều chế
<code>dpskdemod(x,M,phaserot)</code>	Giải điều chế DPSK	
<code>fskmod(x,M,fre_sep,N,Fs,cont)</code>	Điều chế FSK	fre_sep: khoảng cách giữa các tần số liên tiếp N: số mẫu trên một ký hiệu Fs: tần số lấy mẫu, mặc định 1 cont: tính liên tục về pha giữa hai ký hiệu ('cont': liên tục (mặc định), 'discont': gián đoạn)
<code>fskdemod(x,M,fre_sep,N,Fs)</code>	Giải điều chế FSK	
<code>genqammod(x,const)</code>	Điều chế QAM theo mã Gray	const: vector chuyển đổi X: chuỗi các số nguyên từ 0 đến length(const)-1
<code>genqamdemod(y,const)</code>	Giải điều chế QAM theo mã Gray	
<code>mkskmod(x,N,data_enc,phase)</code>	Điều chế MSK	data_enc: phương pháp mã hoá dữ liệu ('diff': mã hoá vi sai (mặc định), 'nondiff': mã hoá không vi sai)
<code>mkskdemod(x,N,data_enc,phase)</code>	Giải điều chế MSK	
<code>oqpskmod(x,phase)</code>	Điều chế OQPSK	phase: offset pha của tín hiệu sau điều chế
<code>oqpskdemod(x,phase)</code>	Giải điều chế OQPSK	
<code>pammod(x,M,phase)</code>	Điều chế PAM	phase: pha ban đầu của tín hiệu sau điều chế
<code>pamdemod(y,M,phase)</code>	Giải điều chế PAM	
<code>qammod(x,M,phase)</code>	Điều chế QAM	phase: offset pha của tín hiệu sau điều chế
<code>qamdemod(y,M,phase)</code>	Giải điều chế QAM	
<code>pskmod(x,M,phase)</code>	Điều chế PSK	
<code>pskdemod(y,M,phase)</code>	Giải điều chế PSK	
<code>modnorm(const,powtype,powval)</code>	Tính hệ số tỷ lệ để chuẩn hoá tín hiệu sau điều chế	const: vector constellation powtype: 'avpow' hoặc 'peakpow': chuẩn hoá theo công suất trung bình hay công suất đỉnh powval: giá trị chuẩn hoá (tính bằng W)

- Mỗi quá trình điều chế số được mô tả bằng một đồ thị hình sao (constellation plot). Đây là một đồ thị được vẽ trên mặt phẳng phức, gồm tập hợp các điểm biểu diễn pha và biên độ sóng

mang ứng với các ký hiệu trong tập ký hiệu nguồn. Để vẽ đồ thị hình sao này, chúng ta có thể sử dụng hàm **scatterplot** của MATLAB. Các bước thực hiện như sau:

- ✓ Tạo vector tín hiệu $x = [0:M-1]$, trong đó M là số ký hiệu có thể có của tập nguồn.
- ✓ Thực hiện phép điều chế số với dữ liệu vào là x .
- ✓ Áp dụng hàm **scatterplot** đối với tín hiệu sau khi điều chế.

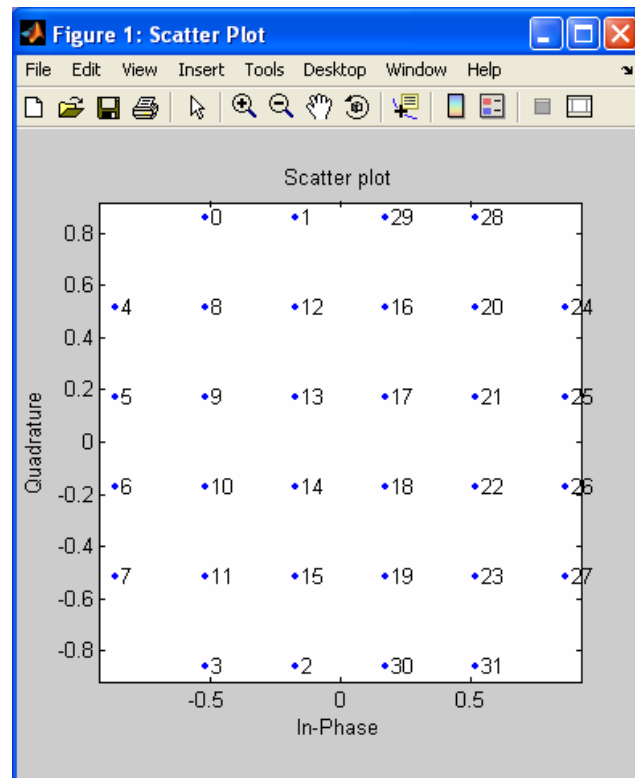
Ví dụ 15-4. Vẽ đồ thị phân bố (constellation plot) cho quá trình điều chế 32-QAM.

Dưới đây là đoạn mã thực hiện yêu cầu trên:

```
M = 32; % Số ký hiệu điều chế
x = [0:M-1]; % Tạo vector tín hiệu [0:M-1]
y = qammod(x,M); % Điều chế 32-QAM
scale = modnorm(y,'peakpow',1); % Chuẩn hoá công suất đỉnh bằng 1W
y = scale*y; % Thay đổi tỷ lệ vẽ.
scatterplot(y); % Vẽ đồ thị
```

Để chú thích mỗi điểm trên đồ thị ứng với ký hiệu tương ứng của nó, ta thêm vào các dòng sau:

```
% Chú thích bằng cách đánh số các điểm.
hold on; % Chú thích trên cùng đồ thị vừa vẽ.
for jj=1:length(y)
text(real(y(jj)),imag(y(jj)),[' ' num2str(jj-1)]);
end
hold off;
```



Hình 15.4.

▪ Nếu xem kỹ các ví dụ về điều chế QAM ở trên ta sẽ thấy rằng cách bố trí các điểm biểu diễn các ký hiệu trong tập nguồn như trên là không tuân theo mã Gray (tức số nhị phân ứng với hai điểm kề nhau trên đồ thị có thể khác nhau nhiều hơn một bit). Muốn thực hiện điều chế QAM đúng quy luật mã Gray, ta phải thực hiện chuyển đổi (mapping) trước khi điều chế và sau khi giải điều chế như trong ví dụ sau:

📖 **Ví dụ 15-5.** Thực hiện lại quá trình điều chế ở ví dụ 15-2 nhưng sử dụng mã Gray.

Để giải quyết bài toán trên, ta thay đổi đoạn chương trình chuyển đổi bit thành các ký hiệu trước khi điều chế và đoạn chương trình chuyển đổi ngược từ các ký hiệu thành chuỗi bit sau khi điều chế như sau:

```
%% Chuyển đổi bit thành ký hiệu
% Đổi các bit nhị phân của x thành các ký hiệu k bit sử dụng mã hóa Gray
% Bước 1. Định nghĩa vector chuyển đổi theo mã Gray. Vector này phụ thuộc
% cách sắp xếp 16 điểm trên đồ thị 16-QAM.
mapping = [0 1 3 2 4 5 7 6 12 13 15 14 8 9 11 10].';
% Bước 2. Thực hiện chuyển đổi như bình thường.
xsym = bi2de(reshape(x,k,length(x)/k),'left-msb');
% Bước 3. Dùng vector đã định nghĩa ở bước 1 để chuyển sang mã Gray.
xsym = mapping(xsym+1);
%% Chuyển đổi các ký hiệu thành các bit
% Thực hiện quá trình ngược với quá trình trên.
% Bước 1. Định nghĩa vector chuyển đổi ngược.
[dummy demapping] = sort(mapping);
% Đầu tiên, demapping có giá trị chạy từ 1 đến M.
% Trừ đi 1 để các giá trị nằm giữa 0 và M-1.
demapping = demapping - 1;
% Bước 2. Chuyển đổi giữa mã Gray và mã nhị phân thông thường.
zsym = demapping(zsym+1);
% Bước 3. Thực hiện chuyển đổi thập phân - nhị phân như bình thường.
z = de2bi(zsym,'left-msb');
% Đổi ma trận z thành một vector.
z = reshape(z.',prod(size(z)),1);
```

Lưu ý: trong ví dụ trên các vector mapping và demapping là giống nhau. Tuy nhiên, một cách tổng quát, đối với các trường hợp khác thì hai vector này sẽ khác nhau.

🔗 Bài tập 15-1.

Tạo một chuỗi xung vuông đơn cực có chu kỳ là 1ms. Sau đó dùng tín hiệu này để điều chế sóng mang có tần số 50kHz, biên độ 1V, dùng các phương pháp sau:

- AM
- AM-DSC
- AM-SSB
- FM

Bài tập 15-2.

Thực hiện phép điều chế QPSK cho một chuỗi bit nhị phân ngẫu nhiên chiều dài 1000 bit, tốc độ 1000bps, tần số sóng mang 40kHz, tần số lấy mẫu 400kHz, bằng hai phương pháp:

- Dùng hàm có sẵn của MATLAB
- Trực tiếp từ định nghĩa

HD: biểu diễn chuỗi bit bằng 2 tín hiệu nhị phân Q và I , sau đó dùng công thức:

$$y(t) = b_I \cos(2\pi f_c t) - b_Q \sin(2\pi f_c t)$$

Bài tập 15-3.

Vẽ constellation plot cho các quá trình điều chế số sau:

- 16-PSK
- 8-QAM
- 8-QAM theo mã Gray

Bài tập 15-4.

Thực hiện phép điều chế BPSK (dùng định nghĩa) cho một chuỗi bit nhị phân ngẫu nhiên chiều dài 1000 bit, tốc độ 64Kbps, tần số sóng mang 1,92 MHz, tần số lấy mẫu 19,2MHz. Sau đó thực hiện giải điều chế. Vẽ dạng sóng các tín hiệu.

Bài tập 15-5.

Khảo sát hiệu ứng offset pha ở máy thu: thực hiện lại bài tập 15-4 trong hai trường hợp:

- sóng mang ở máy thu đồng bộ với máy phát và
- có offset pha sóng mang $\pi/18$ ở máy thu.
- có offset pha sóng mang $\pi/6$ ở máy thu.

Vẽ các tín hiệu và nhận xét.

Danh sách các hàm được giới thiệu trong chương 15

Các hàm điều chế và giải điều chế tương tự

ammod	Điều chế biên độ (AM)
amdemod	Giải điều chế biên độ (AM)
fmmod	Điều chế tần số (FM)
fmdemod	Giải điều chế tần số (FM)
pmmod	Điều chế pha (PM)
pmdemod	Giải điều chế pha (PM)
ssbmod	Điều chế AM đơn biên (SSB)
ssbdemod	Giải điều chế AM đơn biên (SSB)

Các hàm biến đổi cơ số

bi2de	Đổi từ hệ nhị phân sang hệ thập phân
de2bi	Đổi từ hệ thập phân sang hệ nhị phân

Các hàm điều chế và giải điều chế số

dpskmod	Điều chế DPSK
dpskdemod	Giải điều chế DPSK
fskmod	Điều chế FSK
fskdemod	Giải điều chế FSK
genqammod	Điều chế QAM theo mã Gray
genqamdemod	Giải điều chế QAM theo mã Gray
mskmod	Điều chế MSK
mskdemod	Giải điều chế MSK
modnorm	Tính hệ số tỷ lệ để chuẩn hoá biên độ tín hiệu điều chế
oqpskmod	Điều chế OQPSK
oqpskdemod	Giải điều chế OQPSK
pammod	Điều chế biên độ xung PAM
pamdemod	Giải điều chế biên độ xung PAM
qammod	Điều chế QAM
qamdemod	Giải điều chế QAM
pskmod	Điều chế PSK
pskdemod	Giải điều chế PSK
scatterplot	Vẽ đồ thị phân bố của các phương pháp điều chế số

Các phép toán cơ bản khác

reshape	Sắp xếp lại ma trận bằng cách thay đổi số dòng và cột
----------------	---

real	Trả về phần thực của một số phức
imag	Trả về phần ảo của một số phức

Chương 16

KÊNH TRUYỀN VÀ ĐÁNH GIÁ CHẤT LƯỢNG KÊNH TRUYỀN

Với một kênh thông tin cơ bản, tín hiệu tin tức sau khi điều chế sẽ được gửi đi trên kênh truyền. Trong thực tế tín hiệu khi truyền trên kênh truyền sẽ chịu tác động bởi các yếu tố của kênh truyền làm cho tín hiệu thu được không còn giống hoàn toàn tín hiệu phát. Tùy theo các dạng môi trường truyền khác nhau và các hệ thống thông tin khác nhau, sự tác động nói trên sẽ có những đặc trưng khác nhau. MATLAB cho phép người sử dụng mô phỏng ba loại kênh truyền cơ bản, đó là: kênh truyền với nhiễu AWGN, kênh truyền fading và kênh truyền đối xứng nhị phân. Ngoài ra, với các công cụ toán học vô cùng phong phú của MATLAB, người sử dụng có thể tự tạo ra những kênh truyền có những đặc trưng riêng theo ý mình hoặc kết hợp các dạng kênh truyền cơ bản nói trên.

Một vấn đề quan trọng khi mô phỏng một hệ thống thông tin là phân tích các đáp ứng của nó trước các yếu tố gây nhiễu tồn tại trong thế giới thực, minh họa bằng các công cụ đồ họa và đánh giá xem chất lượng của nó có đáp ứng các tiêu chuẩn đã được đặt ra đối với hệ thống hay không. Vấn đề này có thể giải quyết tốt bởi các công cụ đánh giá chất lượng kênh truyền do MATLAB cung cấp.

16.1. KÊNH TRUYỀN AWGN (ADDITIVE WHITE GAUSSIAN NOISE)

Kênh truyền AWGN là dạng kênh truyền có nhiễu cộng, trắng và phân bố theo hàm Gauss. Như vậy, một tín hiệu khi truyền qua kênh truyền này sẽ phải thêm vào một tín hiệu ngẫu nhiên không mong muốn phân bố theo hàm Gauss:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (16.1)$$

▪ Kênh truyền AWGN trong MATLAB được mô phỏng bằng hàm **awgn**. Hàm này sẽ cộng nhiễu Gauss vào một tín hiệu cho trước (có thể là tín hiệu thực hay phức), mức công suất nhiễu do người sử dụng quy định thông qua hai thông số nhập:

- Mức công suất tín hiệu phát (đơn vị mặc định là dBW, hoặc có thể dùng đơn vị W)
- Tỷ số SNR (đơn vị mặc định là dB)

Cú pháp của hàm **awgn** như sau:

```
>> awgn(X, SNR, SigPower, State, PowerType)
```

Trong đó: X là tín hiệu phát

SNR là tỷ số công suất tín hiệu trên nhiễu (tính bằng dB)

SigPower cho biết công suất tín hiệu vào (dBW), hoặc nếu SigPower = 'measured' thì MATLAB sẽ đo công suất tín hiệu phát trước khi cộng nhiễu

State cho biết trạng thái của bộ phát tín hiệu ngẫu nhiên

PowerType chỉ ra đơn vị của SNR và SigPower là đơn vị decibel ('dB') hay đơn vị tuyến tính ('linear')

Ba thông số cuối không nhất thiết phải đưa vào, trong trường hợp không có các thông số này thì xem như công suất tín hiệu phát bằng 0dBW và đơn vị của SNR là dB.

▪ **Lưu ý:** Ngoài thông số SNR, để đánh giá mức độ nhiễu của kênh truyền, người ta cũng thường sử dụng các thông số $\frac{E_b}{N_0}$ (năng lượng bit trên mật độ công suất nhiễu) hoặc $\frac{E_s}{N_0}$ (năng lượng ký hiệu trên mật độ công suất nhiễu). Đọc giả cần lưu ý công thức chuyển đổi giữa các thông số này.

Quan hệ giữa $\frac{E_b}{N_0}$ và $\frac{E_s}{N_0}$:

$$E_s / N_0 (\text{dB}) = E_b / N_0 (\text{dB}) + 10 \lg k \quad (16.2)$$

với k là số bit thông tin chứa trong một ký hiệu.

Quan hệ giữa $\frac{E_s}{N_0}$ và SNR:

$$E_s / N_0 (\text{dB}) = \text{SNR} (\text{dB}) + 10 \lg (T_{\text{sym}} / T_{\text{samp}}) \text{ nếu tín hiệu phát là tín hiệu phức}$$

$$E_s / N_0 (\text{dB}) = \text{SNR} (\text{dB}) + 10 \lg (2T_{\text{sym}} / T_{\text{samp}}) \text{ nếu tín hiệu phát là tín hiệu thực}$$

trong đó T_{sym} và T_{samp} lần lượt là chu kỳ ký hiệu và chu kỳ lấy mẫu.
(16.3)

📖 **Ví dụ 16-1.** Làm lại ví dụ 15-3 với kênh truyền có nhiễu AWGN với $E_b / N_0 = 10 \text{ dB}$.

Trong thí dụ 15-3 ở chương 15, ta đã tạo ra tín hiệu điều chế bằng gốc là y. Đây là tín hiệu phát, tín hiệu này sẽ được cộng thêm nhiễu AWGN bằng cách dùng hàm **awgn** với các thông số nhập là y, SNR và chọn SigPower là 'measured' (chỉ cộng nhiễu mà không thay đổi công suất tín hiệu phát).

%% Tín hiệu phát

ytx = y;

%% Kênh truyền

% Truyền tín hiệu trên kênh truyền AWGN.

EbNo = 10; % Đơn vị dB

snr = EbNo + 10*log10(k) - 10*log10(nsamp); % nsamp = Tsym/Tsamp

ynoisys = awgn(ytx,snr,'measured');

%% Tín hiệu thu

yrx = ynoisys;

Nếu muốn thay đổi công suất phát tín hiệu ta cung cấp giá trị công suất cho thông số SigPower. Ví dụ, để công suất phát bằng 4W, ta viết như sau:

snr = 10^(snr/10); % Chuyển sang đơn vị tuyến tính

ynoisys = awgn(ytx,snr,4,'linear');

▪ Như đã đề cập ở chương 15, MATLAB biểu diễn tín hiệu điều chế số dưới dạng tín hiệu băng gốc tương đương và nói chung đây là tín hiệu phức. Để vẽ tín hiệu điều chế trong miền thời gian ta phải thực hiện chuyển đổi từ tín hiệu băng gốc phức sang tín hiệu băng dải thực. Tuy nhiên MATLAB cũng cung cấp các hàm tạo nhiễu ngẫu nhiên cho phép ta viết lại các hàm điều chế để tạo các tín hiệu điều chế thực (tín hiệu băng dải) và cộng nhiễu để mô phỏng

kênh truyền theo phương pháp thông thường (phương pháp băng dải). Ví dụ sau đây sẽ minh họa điều này.

 **Ví dụ 16-2.** *Viết chương trình điều chế 1 chuỗi bit nhị phân tốc độ 8Kbps bằng phương pháp BPSK với sóng mang 100KHz và phát đi trên kênh truyền có nhiễu AWGN với $E_b/N_0=10\text{dB}$, sau đó giải điều chế. Vẽ các tín hiệu phát và thu.*

Chọn tần số lấy mẫu f_s bằng 1MHz.

Phương pháp thực hiện tương tự như ví dụ 15-3.

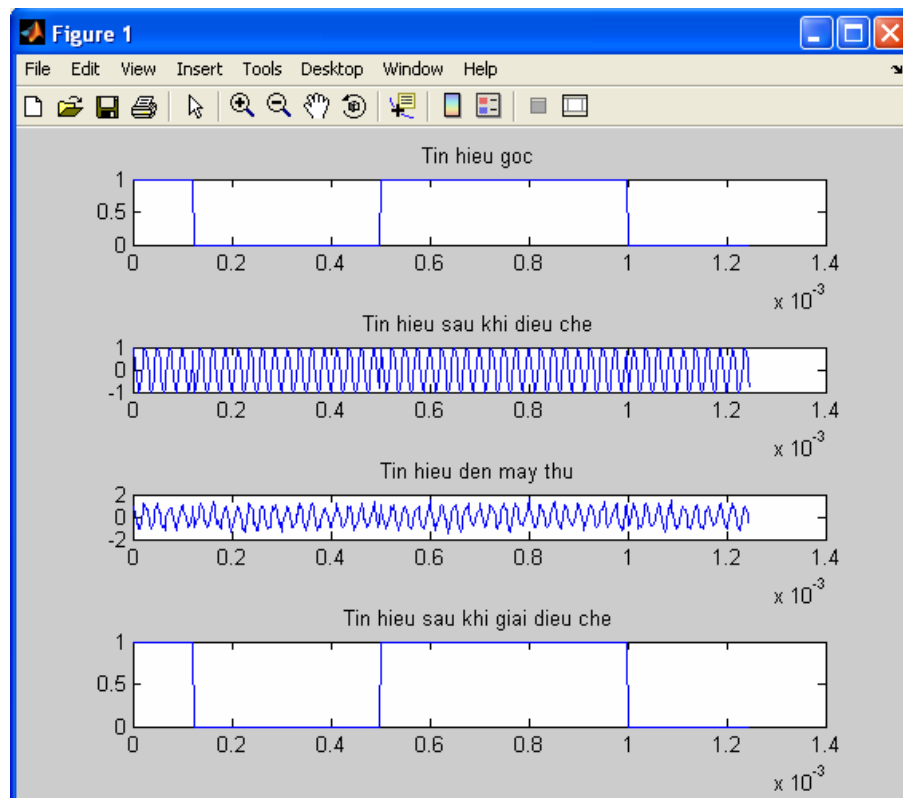
```
% Thiết lập các thông số
N = 10; % Số bit
x = randint(N,1); % Chuỗi bit ngẫu nhiên N bit
M = 2; % Số mức của tín hiệu
k = 1; %
Fb = 8000; % Tốc độ bit
Fc = 40000; % Tần số sóng mang
Fs = 400000; % Tần số lấy mẫu
EbNo = 10; % Đơn vị dB
Nsamp = floor(Fs/Fb*N); % Số mẫu
Time = [0:1/Fs:(Nsamp-1)/Fs]; % Vector thời gian
% Xây dựng các tín hiệu
% Tín hiệu tin tức
xmsg = zeros(1,Nsamp);
for i=1:Nsamp
    xmsg(i) = x(floor((i-1)*Fb/Fs)+1);
end
% Sóng mang
xcar = cos(2*pi*Fc*Time);
% Tín hiệu điều chế BPSK
ytx = (2*xmsg-1).*xcar;
% Cộng nhiễu
snr = EbNo + 10*log10(k) - 10*log10(Nsamp);
ynoisys = awgn(ytx,snr,'measured');
% Giải điều chế
ytmp = ynoisys.*xcar; % Nhân với sóng mang
[num,den]=butter(2,Fc/(Fs/2)); % Bộ lọc thấp Butterworth bậc 2, tần số cắt Fc
ztmp = filter(num,den,ytmp); % Lọc bỏ tần số cao
for i = 1:length(x) % Lấy mẫu và quyết định
    if ztmp(floor((i-1)/Fb+1/2/Fb)*Fs) > 0 % Lấy mẫu ở giữa bit
        z(i) = 1; % Ngưỡng quyết định là 0
    else
```

```

        z(i) = 0;
    end
end
xdemod = zeros(1,Nsamp);
for i=1:Nsamp
    xdemod(i) = z(floor((i-1)*Fb/Fs)+1);
end
figure;
subplot(4,1,1); plot(Time,xmsg); % Vẽ tín hiệu trước khi điều chế.
title('Tín hiệu gốc');
subplot(4,1,2); plot(Time,ytx); % Vẽ tín hiệu sau khi điều chế.
title('Tín hiệu sau khi điều chế');
subplot(4,1,3); plot(Time,ynois); % Vẽ tín hiệu thu được.
title('Tín hiệu đến máy thu');
subplot(4,1,4); plot(Time,xdemod); % Vẽ tín hiệu sau khi giải điều chế.
title('Tín hiệu sau khi giải điều chế');

```

Kết quả thực thi chương trình:



Hình 16.1.

Lưu ý: ngoài hàm *awgn* ra, MATLAB còn cung cấp các hàm để tạo tín hiệu nhiễu Gauss ở cấp thấp hơn cho phép người sử dụng tự tạo tín hiệu nhiễu theo cách của mình:

- Hàm **wgn**(M,N,pow,imp,sigtype,powtype) tạo ra ma trận nhiễu Gauss kích thước MxN, công suất là pow trên tải có trở kháng imp. Sigtype cho biết tín hiệu là thực ('real')

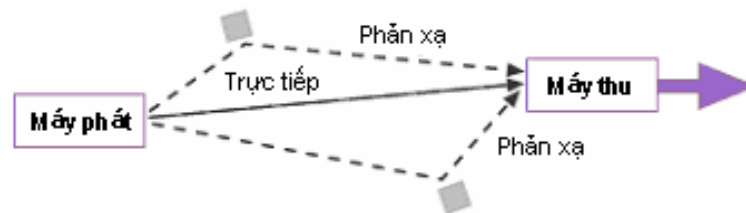
hay phức ('complex'), powtype cho biết đơn vị là dBW ('dBW'), dBm ('dBm') hay W ('linear'). Giá trị mặc định là `imp = 1`, `sigtype = 'real'`, `powtype = 'dBW'`.

Hàm **randn**(M, N) tạo ra một ma trận ngẫu nhiên kích thước MxN có phân bố xác suất theo hàm Gauss với trung bình bằng 0 và phương sai bằng 1.

16.2. KÊNH TRUYỀN FADING

Kênh truyền fading Rayleigh hoặc Rician là mô hình hữu dụng để mô tả các hiện tượng thực trong thông tin không dây (wireless communications). Các hiện tượng này bao gồm hiệu ứng tán xạ đa đường, sự phân tán thời gian, và hiệu ứng Doppler gây ra bởi sự dịch chuyển tương đối giữa máy phát và máy thu.

Hình vẽ dưới đây minh họa kênh truyền fading.



Hình 16.2.

Trên đường dẫn chính từ máy phát đến máy thu, các phiên bản khác nhau của tín hiệu (do sự trì hoãn thời gian) sẽ tập hợp ở máy thu. Các tín hiệu phát đi phản xạ trên các vật chướng ngại khác nhau trên đường truyền sẽ đến máy thu theo các đường khác nhau. Tất cả các tín hiệu nói trên được kết hợp lại tại máy thu gây ra hiệu ứng đa đường. Thông thường, quá trình fading được đặc trưng bởi phân bố Rayleigh nếu đường truyền không thẳng (non-line-of-sight) và được đặc trưng bằng phân bố Rician nếu là kênh truyền tầm nhìn thẳng (line-of-sight).

Sự di chuyển tương đối giữa máy phát và máy thu gây ra hiệu ứng Doppler. Sự tán xạ trên các vật chướng ngại làm cho độ dịch chuyển Doppler thay đổi trên một khoảng nào đó, gọi là phổ Doppler. Độ dịch chuyển Doppler cực đại xảy ra khi đối tượng khảo sát dịch chuyển theo hướng ngược với hướng di chuyển của mobile.

- Trong Communication Toolbox của MATLAB, kênh truyền fading được mô hình hoá dưới dạng một bộ lọc. Tín hiệu truyền qua kênh truyền fading có nghĩa là tín hiệu được lọc bởi bộ lọc đặc biệt này. Quá trình mô phỏng kênh truyền fading gồm 3 bước:

- Bước 1: Tạo một đối tượng kênh truyền (channel object) mô tả các tính chất của kênh truyền mà ta muốn mô phỏng. Đối tượng kênh truyền là một dạng biến trong MATLAB. Để tạo một đối tượng kênh truyền fading ta có thể sử dụng một trong hai hàm **rayleighchan** hoặc **ricianchan**. Sử dụng **ricianchan** trong trường hợp kênh truyền có một đường truyền thẳng kết hợp với một hoặc nhiều đường phản xạ; còn trong trường hợp chỉ có một hoặc vài đường phản xạ (non line-of-sight), ta dùng **rayleighchan**.

Ví dụ, để tạo một kênh truyền fading Rayleigh tác động lên một tín hiệu được lấy mẫu ở tần số 1000000Hz, với độ dịch chuyển Doppler tối đa là 130Hz, ta viết dòng lệnh sau:

```
>> c1 = rayleighchan(1/100000,130);
```

Một cách khác để tạo ra một đối tượng kênh truyền là copy lại một đối tượng có sẵn, sau đó thay đổi các thuộc tính của nó như mong muốn.

```
>> c2 = copy(c1);
```

```
>> c2 = c1;
```

Trong hai cách nêu trên, cách thứ nhất sẽ tạo ra một đối tượng có các thuộc tính độc lập với đối tượng c1, trong khi đó các thuộc tính của đối tượng tạo ra theo cách thứ 2 sẽ phụ thuộc vào đối tượng c1.

○ **Bước 2:** Hiệu chỉnh các thông số của kênh truyền theo nhu cầu mô phỏng.

Mỗi một đối tượng kênh truyền có một số các thuộc tính riêng, chẳng hạn: loại kênh truyền, tần số lấy mẫu, độ dịch chuyển Doppler cực đại, ... Để xem các thuộc tính của một đối tượng kênh truyền, ta chỉ cần gõ tên của đối tượng trên cửa sổ lệnh của MATLAB. Để truy xuất đến một thuộc tính cụ thể ta viết tên của đối tượng, sau đó là dấu chấm, rồi đến tên của thuộc tính cần truy xuất. Như vậy, để hiệu chỉnh đối tượng ta chỉ cần gán cho các thuộc tính của nó các giá trị mà ta mong muốn. Ví dụ:

```
>> c1 = rayleighchan(1/100000,130); % Khởi tạo một đối tượng kênh truyền
% fading
```

```
>> c1 % Xem các thuộc tính của c1
```

```
c1 =
```

```
ChannelType: 'Rayleigh'
InputSamplePeriod: 1.0000e-005
MaxDopplerShift: 130
PathDelays: 0
AvgPathGaindB: 0
NormalizePathGains: 1
PathGains: 0.2104- 0.6197i
ChannelFilterDelay: 0
ResetBeforeFiltering: 1
NumSamplesProcessed: 0
```

```
>> g = c1.PathGains % Lấy giá trị thuộc tính PathGains của c1.
```

```
g =
```

```
0.2104 - 0.6197i
```

Lưu ý: - Một số thuộc tính có tính chất “Read Only”, nghĩa là ta không thể thay đổi giá trị của nó. Ví dụ thuộc tính NumSamplesProcessed cho biết số mẫu đã xử lý từ lần reset cuối cùng.

- Một số thuộc tính có tính liên kết với nhau, khi thuộc tính này thay đổi thì thuộc tính kia cũng thay đổi theo.

- Ngoài các thuộc tính trên, đối tượng kênh truyền fading Rician có thêm một thuộc tính là Kfactor.

▪ Sau đây là ý nghĩa của một số thuộc tính quan trọng:

PathDelays: Các giá trị trì hoãn của các đường dẫn.

- Giá trị đầu tiên thường chọn là 0 ứng với đường tín hiệu đến máy thu đầu tiên.

- Với môi trường trong nhà (indoor), các trì hoãn tiếp theo sẽ có giá trị từ 1 đến 100ns.

- Với môi trường bên ngoài (outdoor), các trì hoãn tiếp theo sẽ có giá trị từ 100ns đến 10μs.

AveragePathGains: Chỉ thị độ lợi công suất trung bình cho mỗi đường fading.

- Trong mô phỏng ta thường chọn giá trị độ lợi trung bình trong khoảng -20dB đến 0dB. Giá trị độ lợi theo dB sẽ giảm gần như tuyến tính theo thời gian trì hoãn, tuy nhiên đường cong cụ thể sẽ phụ thuộc vào môi trường truyền.

- Để bảo đảm công suất tổng cộng của các đường bằng 1, ta phải chuẩn hoá các độ lợi bằng thuộc tính `NormalizePathGains`.

MaxDopplerShift: Độ dịch chuyển Doppler cực đại.

- Trong một số ứng dụng wireless, chẳng hạn trong hệ thống thông tin di động, người ta thường biểu diễn thông số này dưới dạng tốc độ di chuyển của mobile. Nếu mobile di chuyển với tốc độ v (m/s), tần số sóng mang là f (Hz), c là tốc độ ánh sáng (m/s) thì độ dịch chuyển Doppler cực đại sẽ là:

$$f_d = \frac{vf}{c} \quad (16.4)$$

- $f_d = 0$ ứng với kênh truyền tĩnh

KFactor: Hệ số K của kênh truyền fading Rician là tỷ số công suất phản xạ và khuếch tán trên đường truyền trực tiếp (line-of-sight), biểu diễn dạng tuyến tính, không theo dB.

- Giá trị tiêu biểu đối với kênh Rician: $K = 1 \div 10$.

- Giá trị $K = 0$ ứng với kênh truyền Rayleigh.

ResetBeforeFiltering: Là một biến Boolean, nếu bằng 1: đối tượng sẽ được reset trước khi thực hiện lọc một tín hiệu. Trong trường hợp tín hiệu cần xử lý là một chuỗi các vector thì ta sẽ phải thực hiện lọc nhiều lần. Muốn bảo đảm tính liên tục qua các lần thực hiện, nghĩa là giữ lại các thông tin trạng thái của đối tượng, ta phải set thuộc tính này bằng 0. Ngược lại nếu không muốn lưu các thông tin trạng thái cho lần lọc kế tiếp, ta set thuộc tính này bằng 1, hoặc dùng lệnh **reset**.

○ **Bước 3:** Đưa tín hiệu qua kênh truyền bằng cách dùng hàm **filter**.

▪ Ta thực hiện hàm **filter** với các thông số nhập là tên đối tượng kênh truyền và tín hiệu phát đi.

Để minh hoạ phương pháp mô phỏng kênh truyền fading trong MATLAB, chúng ta khảo sát một số ví dụ cụ thể sau:

 **Ví dụ 16-3.** Vẽ công suất tín hiệu bị nhiễu fading khi truyền qua kênh truyền fading Rayleigh có độ dịch chuyển Doppler cực đại bằng 100Hz, tần số lấy mẫu tín hiệu là 100KHz.

```
c = rayleighchan(1/10000,100); % Khởi tạo đối tượng kênh truyền
sig = j*ones(2000,1); % Tín hiệu phát
y = filter(c,sig); % Đưa tín hiệu qua kênh truyền.
C % Xem các thuộc tính của kênh truyền.
% Vẽ công suất tín hiệu nhiễu theo số mẫu.
plot(20*log10(abs(y)))
```

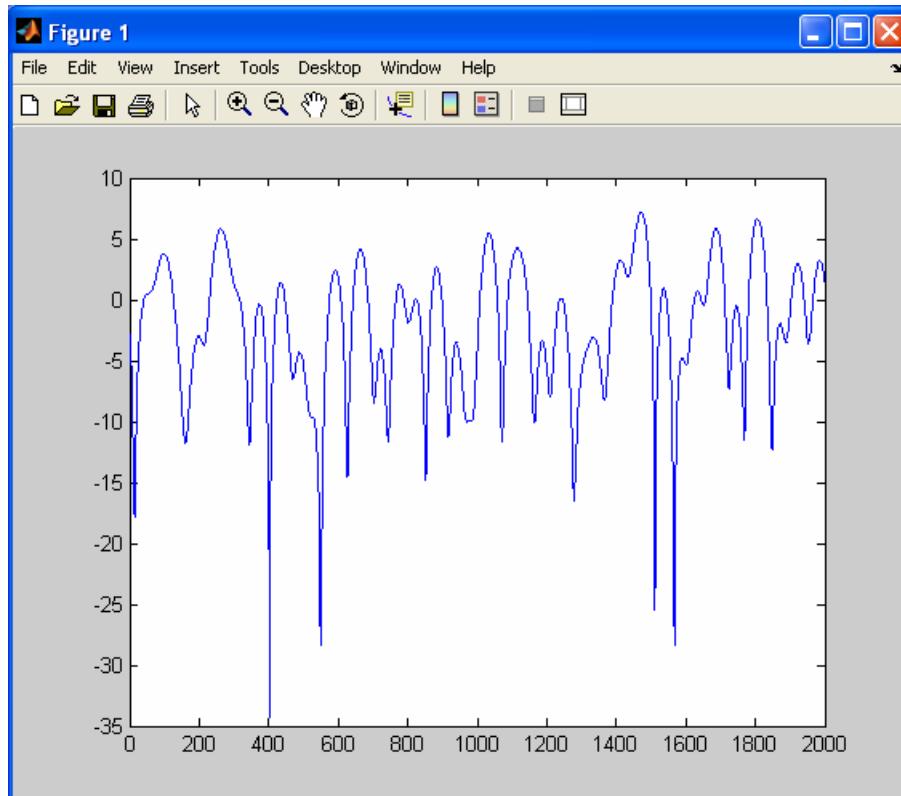
Kết quả thực thi chương trình:

```
c =
    ChannelType: 'Rayleigh'
```

```

InputSamplePeriod: 1.0000e-004
MaxDopplerShift: 100
PathDelays: 0
AvgPathGaindB: 0
NormalizePathGains: 1
PathGains: -1.1700+ 0.1288i
ChannelFilterDelay: 0
ResetBeforeFiltering: 1
NumSamplesProcessed: 2000

```



Hình 16.3.

Ví dụ 16-4. Mô phỏng quá trình truyền tín hiệu điều chế DPSK qua kênh truyền fading Rayleigh như ở ví dụ 16-3 kết hợp với nhiễu AWGN. Tính và vẽ tỷ lệ bit lỗi ứng với các giá trị SNR khác nhau.

Điểm cần lưu ý với ví dụ này là ta phải thực hiện hàm filter trước khi thực hiện hàm **awgn** khi mô phỏng ảnh hưởng kết hợp của nhiễu Gauss và fading. Sau đây là chương trình thực hiện:

```

% Tạo một đối tượng mô tả kênh truyền Rayleigh .
chan = rayleighchan(1/10000,100);
% Tạo chuỗi dữ liệu phát
M = 2; % Số mức của điều chế DPSK
tx = randint(50000,1,M); % Chuỗi bit ngẫu nhiên
dpskSig = dpskmod(tx,M); % Tín hiệu điều chế DPSK
fadedSig = filter(chan,dpskSig); % Truyền qua kênh truyền fading
% Tính BER ứng với các giá trị khác nhau của SNR.

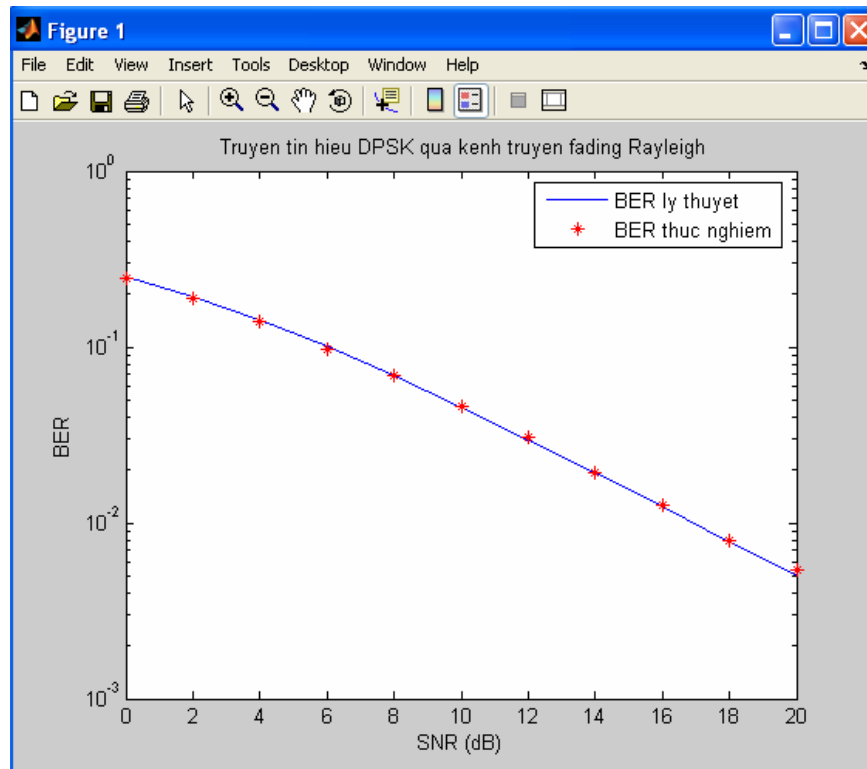
```

```

SNR = 0:2:20; % Khoảng giá trị SNR tính theo dB.
for n = 1:length(SNR)
    rxSig = awgn(fadedSig,SNR(n)); % Cộng nhiễu AWGN.
    rx = dpskdemod(rxSig,M); % Giải điều chế.
    % Tính BER. Bỏ qua mẫu đầu tiên ứng với điều kiện đầu của DPSK
    [nErrors, BER(n)] = biterr(tx(2:end),rx(2:end));
end
% Tính BER theo lý thuyết.
BERtheory = berfading(SNR,'dpsk',M,1);
% Vẽ đồ thị kết quả.
semilogy(SNR,BERtheory,'b-',SNR,BER,'r*');
legend('BER lý thuyết','BER thực nghiệm');
xlabel('SNR (dB)'); ylabel('BER');
title('Truyền tín hiệu DPSK qua kênh truyền fading Rayleigh');

```

Kết quả mô phỏng như sau:



Hình 16.4.

16.3. KÊNH TRUYỀN ĐẢO BIT NHỊ PHÂN

Mô hình kênh truyền đảo bit nhị phân là một mô hình dùng để mô phỏng các phương pháp mã hoá sửa sai. Cách tác động của kênh truyền này đối với một tín hiệu nhị phân truyền qua nó là: thực hiện đảo bit tín hiệu theo một xác suất cho trước.

MATLAB thực hiện mô phỏng loại kênh truyền này bằng cách dùng hàm **bsc** với các thông số nhập là tín hiệu nhị phân truyền qua và xác suất đảo bit p .

Nếu muốn mô phỏng kênh truyền đảo bit nhị phân với các tính chất thống kê liên quan đến số bit lỗi trong một từ mã, ta có thể dùng hàm `randerr`. Hàm này tạo ra một ma trận gồm các phần tử 0 hoặc 1. Nếu ta xem mỗi hàng của ma trận ứng với một từ mã thì số phần tử 1 trong mỗi hàng sẽ biểu diễn số bit lỗi trong mỗi từ mã. Ta có thể quy định số bit lỗi trong một từ mã với một xác suất cho trước. Ví dụ để tạo ma trận nhị phân 5×4 (5 từ mã 4 bit) sao cho số lượng bit 1 trong mỗi hàng sẽ bằng 1 với xác suất bằng 0.3 và bằng 2 với xác suất bằng 0.7:

```
>> f=randerr(3,4,[1,2;0.3,0.7])
f =
f =
    0     0     0     1
    0     1     1     0
    0     1     0     1
```

 **Ví dụ 16-5.** Khảo sát mã chập bằng cách sử dụng mô hình kênh truyền đảo bit nhị phân với xác suất đảo bit là 0.01.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]); % Mã hoá Trellis
msg = ones(10000,1); % Dữ liệu mã hóa
code = convenc(ones(10000,1),t); % Mã hóa bằng mã chập.
[ncode,err] = bsc(code,.01); % Tạo các lỗi bit.
numchanerrs = sum(sum(err)) % Tổng số lỗi của kênh truyền
dcode = vitdec(ncode,t,2,'trunc','hard'); % Giải mã.
[numsyserrs,ber] = biterr(dcode,msg) % Số lỗi sau khi giải mã
```

Kết quả:

```
numchanerrs =
    144
numsyserrs =
    28
ber =
    0.0028
```

16.4. ĐÁNH GIÁ CHẤT LƯỢNG THÔNG QUA MÔ PHỎNG (PHƯƠNG PHÁP MONTE CARLO)

Phương pháp cơ bản để tính tỷ lệ lỗi bit hoặc tỷ lệ lỗi ký hiệu trong một hệ thống thông tin số là mô phỏng việc truyền một số lượng lớn các bit hoặc ký hiệu qua kênh truyền, sau đó so sánh toàn bộ dữ liệu nhận được với toàn bộ dữ liệu trước khi truyền. Trong lý thuyết xác suất thống kê, phương pháp tính xác suất dựa vào các phép thử như trên được gọi là phương pháp Monte Carlo.

Các hàm **biterr** và **symerr** trong MATLAB sẽ so sánh hai tập dữ liệu cho trước và trả về số bit lỗi hoặc số ký hiệu lỗi. Một sự khác nhau giữa hai phần tử tương ứng của hai tập dữ liệu (có thể là bit hoặc ký hiệu) được tính là một lỗi. Thông thường, hai tập dữ liệu khảo sát là tập dữ liệu trước khi mã hoá ở máy phát và tập dữ liệu ở máy thu sau khi giải mã.

Kết quả mô phỏng sẽ càng chính xác nếu số lượng dữ liệu mô phỏng càng nhiều. Cụ thể, ta nên chọn tập dữ liệu đủ lớn sao cho có ít nhất 100 lỗi xảy ra.

 **Ví dụ 16-6.** Xử lý một chuỗi dữ liệu nhị phân: điều chế bằng phương pháp 16-QAM, sau đó truyền qua kênh truyền có nhiễu AWGN với tỷ số $E_b/N_0 = 10\text{dB}$. Thực hiện giải điều chế và tính tỷ số bit lỗi của chuỗi bit thu được.

Ví dụ này là sự mở rộng của ví dụ 15-2, trong đó ta thêm vào hàm tạo nhiễu Gauss **awgn** và hàm tính tỷ lệ lỗi bit **biterr**.

```
%% Định nghĩa các thông số.
M = 16;
k = log2(M);
n = 3e4;
nsamp = 1;
%% Nguồn tín hiệu
% Tạo chuỗi dữ liệu nhị phân ngẫu nhiên dưới dạng vector cột.
x = randint(n,1); % Chuỗi bit nhị phân ngẫu nhiên
%% Chuyển đổi các bit thành các ký hiệu k bit.
xsym = bi2de(reshape(x,k,length(x)/k).','left-msb');
%% Điều chế 16-QAM
y = qammod(xsym,M);
%% Tín hiệu phát
ytx = y;
%% Kênh truyền
% Truyền tín hiệu qua kênh truyền AWGN.
EbNo = 10; % Đơn vị dB
snr = EbNo + 10*log10(k) - 10*log10(nsamp);
ynoisys = awgn(ytx,snr,'measured');
%% Tín hiệu thu
yrx = ynoisys;
%% Giải điều chế
zsym = qamdemod(yrx,M);
%% Chuyển đổi ngược từ các ký hiệu thành chuỗi bit nhị phân.
z = de2bi(zsym,'left-msb');
z = reshape(z.',prod(size(z)),1);
%% Tính BER
% So sánh y và z để đếm số bit lỗi và tính tỷ lệ lỗi bit
[number_of_errors,bit_error_rate] = biterr(x,z)
```

Kết quả thực hiện đoạn chương trình trên:

```
number_of_errors =
    80
bit_error_rate =
    0.0027
```

Trong ví dụ trên hàm **biterr** trả về số bit lỗi và tỷ lệ lỗi bit khi so sánh hai chuỗi dữ liệu x và z. Một cách tổng quát, hàm này có thể so sánh hai ma trận số nguyên bất kỳ có cùng kích thước. Ngoài hai thông số chính là hai ma trận dữ liệu, còn có thêm hai thông số tùy chọn:

- `[num, ratio] = biterr(..., K)` so sánh hai ma trận với K là số bit để biểu diễn một phần tử trong ma trận. K phải lớn hơn hoặc bằng số bit tối thiểu để biểu diễn phần tử lớn nhất của ma trận.

- `[num, ratio] = biterr(..., flag)` dùng chuỗi ký tự flag để xác định phương thức so sánh. Nếu `flag = 'column-wise'` thì hàm **biterr** sẽ thực hiện so sánh theo từng cột và xuất kết quả dưới dạng vector hàng. Nếu `flag = 'row-wise'` thì hàm này sẽ thực hiện so sánh theo từng hàng và xuất kết quả dưới dạng vector cột. Nếu `flag = 'overall'`, hàm sẽ thực hiện so sánh trên toàn bộ ma trận và như vậy kết quả là một vô hướng.

- `[num, ratio, individual] = biterr(...)` cho phép trả về ma trận individual, trong đó mỗi phần tử của nó là số bit lỗi giữa hai phần tử tương ứng của hai ma trận so sánh.

Ví dụ:

```
>> A = [1 2 3; 1 2 2];
>> B = [1 2 0; 3 2 2];
>> [Num, Rat]=biterr(A,B,3)
Num =
    3
Rat =
    0.1667
>> [Num,Rat,Ind] = biterr(A,B,3,'column-wise')
Num =
    1    0    2
Rat =
    0.1667    0    0.3333
Ind =
    0    0    2
    1    0    0
```

- Hàm **symerr** tương tự như hàm **biterr** chỉ khác ở chỗ so sánh các ký hiệu thay vì các bit. Ví dụ sau đây minh họa sự khác nhau giữa tỷ lệ bit lỗi và tỷ lệ ký hiệu lỗi.

```
>> a = [1 2 3]'; b = [1 4 4]';
>> format rat % Hiển thị dạng tỷ số.
>> [snum,srate] = symerr(a,b)
snum =
    2
srate =
    2/3
>> [bnum,brate] = biterr(a,b)
bnum =
    5
```

brate =

5/9

Số bit tối thiểu để biểu diễn phần tử lớn nhất trong hai ma trận là 3 bit. Các ma trận đều có 3 phần tử, do đó có $3 \times 3 = 9$ bit. Số bit lỗi là 5, gồm hai bit lỗi ở phần tử thứ hai và 3 bit lỗi ở phần tử thứ ba.

16.5. TÍNH XÁC SUẤT LỖI TRÊN LÝ THUYẾT

Trong quá trình mô phỏng một hệ thống viễn thông, ta thường phải đối chiếu kết quả mô phỏng với các công thức lý thuyết. Trong một số hệ thống tiêu biểu, nhiều công trình lý thuyết đã cung cấp các công thức tính tỷ lệ bit lỗi dưới dạng tường minh. Các công thức này được xây dựng thành các hàm MATLAB cho phép người sử dụng MATLAB có thể kiểm tra lại một cách nhanh chóng độ chính xác của quá trình mô phỏng.

Bảng 16.1. Các hàm tính xác suất lỗi theo lý thuyết

Hàm	Chức năng
berawgn	Xác suất lỗi của kênh truyền AWGN không mã hoá
bercoding	Xác suất lỗi của kênh truyền AWGN có mã hoá
berfading	Xác suất lỗi của kênh truyền fading Rayleigh không mã hoá
bersync	Xác suất lỗi của kênh truyền AWGN không mã hoá và đồng bộ không chính xác

Độc giả có thể tìm hiểu cách sử dụng các hàm này bằng cách gõ lệnh **help** + tên hàm ở cửa sổ lệnh của MATLAB. Các thông số chính cần nhập vào là tỷ số Eb/No, phương pháp điều chế cùng với các thông số tương ứng với nó. Ví dụ, để tính xác suất lỗi trên kênh truyền có Eb/No = 10dB, phương pháp điều chế là FSK với M = 2, giải điều chế đồng bộ, ta gõ các dòng lệnh sau:

```
>> ebno = 10;
>> M = 4;
>> berawgn(ebno, 'fsk', M, 'coherent')
ans =
    7.6892e-006
```

Sau đây là một ví dụ minh họa cách sử dụng các hàm nói trên để so sánh kết quả mô phỏng với lý thuyết.

 **Ví dụ 16-7.** Thực hiện mô phỏng kênh truyền AWGN không mã hoá để đánh giá tỷ lệ bit lỗi cho trường hợp điều chế 8-PAM tùy theo các giá trị của Eb/No. Vẽ đồ thị BER và so sánh với đồ thị BER lý thuyết.

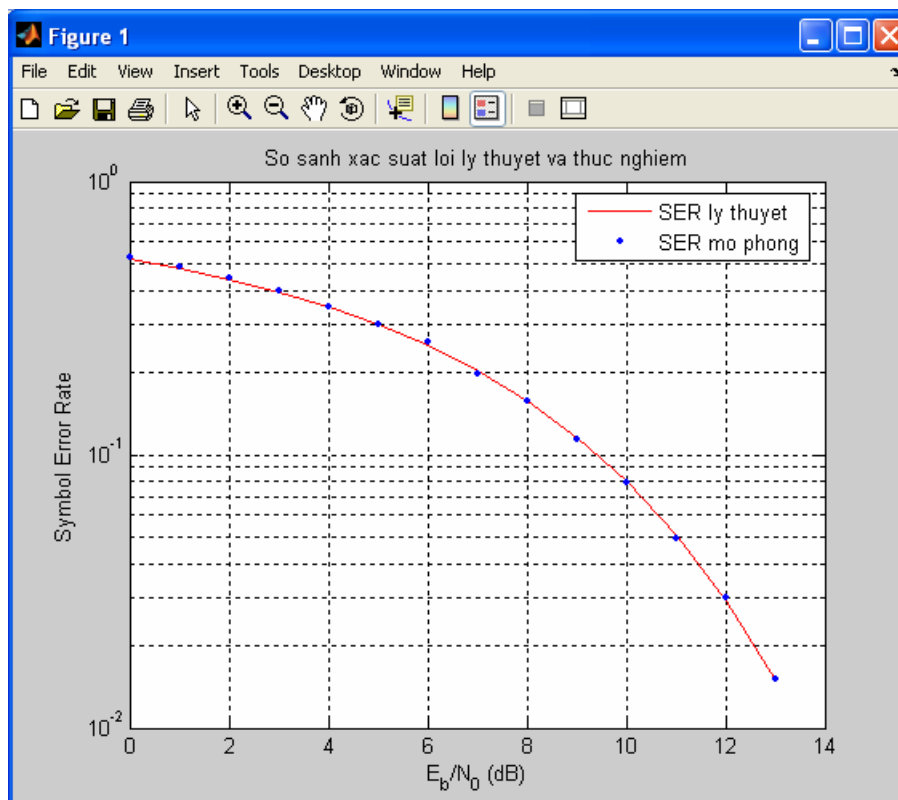
```
% 1. Tính xác suất lỗi theo lý thuyết.
M = 8; EbNo = [0:13];
ser = berawgn(EbNo, 'pam', M) .* log2(M);
% Vẽ đồ thị SER theo lý thuyết.
figure; semilogy(EbNo, ser, 'r');
xlabel('E_b/N_0 (dB)'); ylabel('Symbol Error Rate');
grid on; drawnow;
% 2. Tính xác suất lỗi bằng cách mô phỏng.
```

```

n = 10000; % Tổng số ký hiệu được xử lý
k = log2(M); % Số bit trên một ký hiệu
snr = EbNo+3+10*log10(k); % Đổi từ Eb/No sang SNR
ynoisyy=zeros(n,length(snr));
x = randint(n,1,M); % Tạo tín hiệu tin tức
y = pammod(x,M); % Điều chế
% Truyền tín hiệu qua kênh truyền AWGN. Lặp lại với các giá trị khác nhau
% của SNR
for jj = 1:length(snr)
ynoisyy(:,jj) = awgn(real(y),snr(jj),'measured');
end
z = pamdemod(ynoisyy,M); % Giải điều chế
% Tính xác suất lỗi ký hiệu.
[num,rt] = symerr(x,z);
% 3. Vẽ đồ thị SER mô phỏng trên cùng một đồ thị.
hold on; semilogy(EbNo,rt,'b. ');
legend('SER lý thuyết','SER mô phỏng');
title('So sánh xác suất lỗi lý thuyết và thực nghiệm');
hold off;

```

Kết quả mô phỏng như sau:



Hình 16.5.

16.6. MỘT SỐ CÔNG CỤ HỖ TRỢ ĐỂ VẼ ĐỒ THỊ BER

Đồ thị BER là một công cụ trực quan để đánh giá chất lượng của một hệ thống thông tin, nó biểu diễn sự thay đổi của xác suất lỗi bit hoặc xác suất lỗi ký hiệu theo các thông số SNR hoặc E_b/N_0 của hệ thống. Thông thường, các đồ thị BER được biểu diễn trên hệ trục logarithm, trong đó trục hoành biểu diễn giá trị E_b/N_0 (hoặc SNR) theo đơn vị decibel, còn trục tung biểu diễn xác suất lỗi theo thang logarithm (cơ số 10). Trong Communication Toolbox của MATLAB, ngoài các công cụ tính xác suất lỗi để tạo ra nguồn dữ liệu cho đồ thị BER, còn có các công cụ hỗ trợ để hiệu chỉnh đồ thị BER để tăng tính trực quan của đồ thị. Đó là các công cụ **berfit** và **berconfint**.

▪ Công cụ curve-fitting: **berfit**

Đây là công cụ dùng để hiệu chỉnh đồ thị, được sử dụng khi ta có một tập hợp các số liệu thực nghiệm chưa hoàn hảo nhưng cần phải vẽ một đồ thị BER tương đối trơn. Hàm **berfit** cho phép ta:

- Lựa chọn tất cả các thông số của quá trình làm trơn đồ thị, ví dụ chọn trước biểu thức nội suy dùng để hiệu chỉnh đồ thị
- Vẽ các điểm dữ liệu thực nghiệm cùng với đồ thị đã hiệu chỉnh
- Nội suy thêm các điểm trên đồ thị nằm giữa các điểm dữ liệu thực nghiệm để làm đường cong trơn hơn
- Lấy những thông tin cần thiết của quá trình hiệu chỉnh, ví dụ giá trị số của các điểm trên đường cong hoặc các hệ số của đa thức nội suy biểu diễn đường cong

Lưu ý: hàm **berfit** thực hiện quá trình nội suy, chứ không phải là ngoại suy. Phép ngoại suy để tạo các dữ liệu BER nằm ngoài khoảng giá trị thực nghiệm là không đáng tin cậy.

▪ Công cụ xác định khoảng tin cậy của dữ liệu: **berconfint**

```
>> [ber,intv] = berconfint(Nerr,Ntrial,level)
```

Hàm **berconfint** thực hiện tính toán xác suất lỗi và khoảng tin cậy của dữ liệu khi thực hiện phương pháp Monte Carlo với **Ntrial** phép thử và có **Nerr** lỗi. Độ tin cậy được xác định bởi thông số **level**. Nếu không nhập thông số này thì giá trị mặc định là 0.95 (95%).

Trong ví dụ dưới đây, ta sẽ thấy được ứng dụng của các hàm **berfit** và **berconfint** để vẽ đồ thị BER. Cũng trong ví dụ này, quá trình mô phỏng sẽ dừng khi số bit lỗi đạt đến một giá trị cho trước thay vì dừng khi truyền hết **N** bit như ở các thí dụ trước.

 **Ví dụ 16-8.** Thực hiện mô phỏng hệ thống thông tin DBPSK. Ứng dụng công cụ làm trơn đồ thị trong khi vẽ đồ thị BER.

Đầu tiên, ta khởi tạo các thông số và các tín hiệu mô phỏng cần thiết. Hai thông số chính là khoảng giá trị khảo sát của E_b/N_0 và số bit lỗi tối thiểu phải có trước khi tính giá trị BER đối với E_b/N_0 đang xét.

```
siglen = 1000; % Số bit cho mỗi lần thử
M = 2; % DBPSK
EbNomin = 0; EbNomax = 10; % Khoảng giá trị Eb/No theo dB
numerrmin = 5; % Tính BER khi số bit lỗi > 5
EbNovec = EbNomin:1:EbNomax;
numEbNos = length(EbNovec); % Số giá trị Eb/No
% Cấp phát trước các vector để chứa các dữ liệu.
```

```
ber = zeros(1,numEbNos); % Các giá trị BER
intv = cell(1,numEbNos); % Dãy các khoảng tin cậy
```

Tiếp theo, ta sử dụng vòng lặp để thực hiện nhiều lần quá trình mô phỏng cho các giá trị khác nhau của Eb/No. Trong mỗi vòng lặp lại có một vòng lặp **while** để bảo đảm quá trình mô phỏng vẫn tiếp tục cho đến khi số bit lỗi vượt quá giá trị định trước. Sau mỗi vòng lặp các vector ber và intv sẽ được cập nhật.

```
% Lặp lại với các giá trị khác nhau của Eb/No.
for jj = 1:numEbNos
EbNo = EbNovec(jj);
snr = EbNo; % Điều chế BPSK
ntrials = 0; % Số lần thực hiện vòng lặp while bên dưới
numerr = 0; % Số lỗi ứng với giá trị Eb/No đang xét
% Lặp lại mô phỏng cho đến khi số bit lỗi > numerrmin
while (numerr < numerrmin)
msg = randint(siglen, 1, M); % Tạo chuỗi tin tức.
txsig = dpskmod(msg,M); % Điều chế
rxsig = awgn(txsig, snr, 'measured'); % Cộng nhiễu.
decodmsg = dpskdemod(rxsig,M); % Giải điều chế.
newerrs = biterr(msg,decodmsg); % Số lỗi trong lần thử này
numerr = numerr + newerrs; % Tổng số lỗi ứng với giá trị Eb/No đang xét
ntrials = ntrials + 1; % Cập nhật số lần thử.
end
% Tỷ lệ lỗi và khoảng tin cậy 98% cho giá trị Eb/No hiện tại
[ber(jj), intv1] = berconfint(numerr, (ntrials * siglen), .98);
intv{jj} = intv1; % Lưu lại khoảng tin cậy
disp(['EbNo = ' num2str(EbNo) ' dB, ' num2str(numerr) ...
' errors, BER = ' num2str(ber(jj))])
end
```

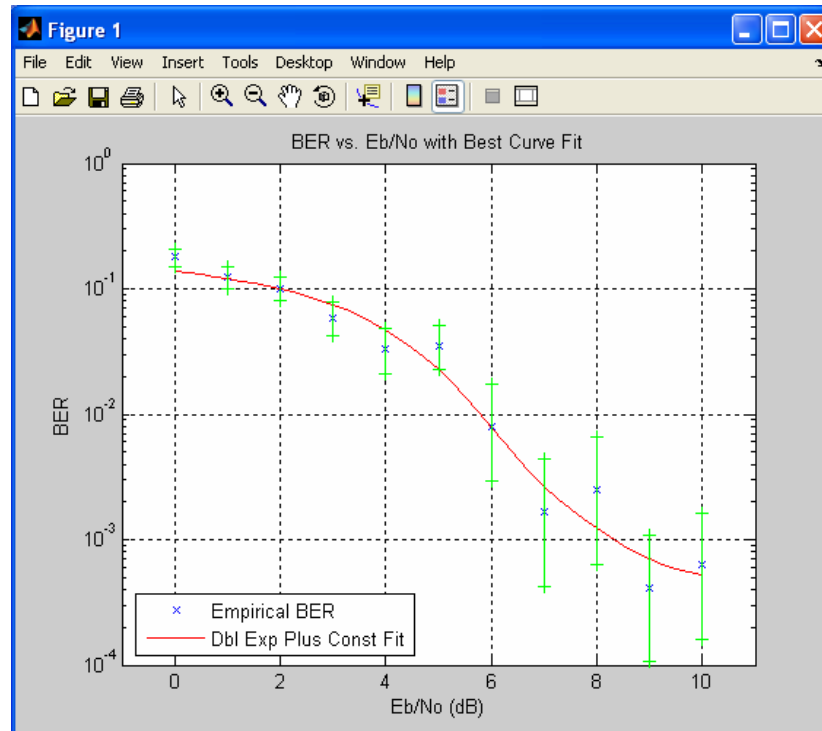
Sau mỗi vòng lặp, trên màn hình sẽ xuất hiện dòng kết quả tương tự như sau:

```
EbNo = 7 dB, 9 errors, BER = 0.0045
```

Cuối cùng là quá trình vẽ đồ thị BER, trong đó có sử dụng hàm **berfit** để làm trơn đồ thị.

```
% Dùng hàm berfit để làm trơn và vẽ đồ thị BER.
fitEbNo = EbNomin:0.25:EbNomax; % Các giá trị nội suy
berfit(EbNovec,ber,fitEbNo);
% Vẽ khoảng tin cậy.
hold on;
for jj=1:numEbNos
semilogy([EbNovec(jj) EbNovec(jj)],intv{jj},'g-+');
end
hold off;
```

Kết quả mô phỏng như sau:



Hình 16.6.

Trong hình vẽ trên, hàm **berfit** tự tạo ra các chú thích gồm: 'Empirical BER' (Giá trị BER thực nghiệm) và 'Dbl Exp Plus Const Fit' (cho biết loại hàm nội suy được sử dụng).

16.7. GIẢN ĐỒ MẮT (EYE DIAGRAM)

Giản đồ mắt là một công cụ rất đơn giản và thuận tiện để khảo sát các ảnh hưởng của nhiễu giao thoa liên ký tự (ISI-Intersymbol Interference) và một số yếu tố khác làm giảm chất lượng kênh truyền. Giản đồ mắt được xây dựng bằng cách vẽ tín hiệu thu theo thời gian trong khoảng thời gian cố định (thường là một chu kỳ ký hiệu), sau khi hết khoảng thời gian này, trục thời gian lại quay về điểm khởi đầu. Như vậy sẽ có nhiều đường cong chồng lấn lên nhau trong khoảng thời gian định sẵn nói trên.

Giản đồ mắt mang lại nhiều thông tin hữu ích về tính chất của kênh truyền. Vị trí mà giản đồ mắt mở rộng nhất sẽ ứng với thời điểm thực hiện lấy mẫu và quyết định để khôi phục lại tín hiệu tin tức. Một giản đồ mắt mở rộng có nghĩa là kênh truyền ít bị ảnh hưởng bởi nhiễu, nghĩa là BER thấp. Ngược lại, nếu giản đồ mắt khép lại thì kênh truyền bị can nhiễu lớn và tỷ số BER có giá trị lớn.

Để vẽ giản đồ mắt trong MATLAB, ta dùng hàm **eyediagram**.

```
>> eyediagram(X,N,Period,Offset,Plotstring,H);
```

Trong đó:

X là tín hiệu cần vẽ giản đồ mắt, có thể có một trong các dạng format như trong bảng 16.2.

N : số mẫu tín hiệu trong khoảng thời gian vẽ giản đồ mắt

Period: khoảng thời gian vẽ giản đồ mắt là từ $-Period/2$ đến $+Period/2$. Giá trị mặc định là 1.

Offset: số thứ tự của mẫu nằm ở trung tâm của khoảng thời gian vẽ ($0 < \text{Offset} < N$).
Giá trị mặc định bằng 0.

Plotstring: chuỗi xác định ký hiệu và màu sắc dùng để vẽ. Giá trị mặc định là 'b- '.

H: xác định đối tượng figure mà ta muốn vẽ giản đồ mắt (mặc định là []).

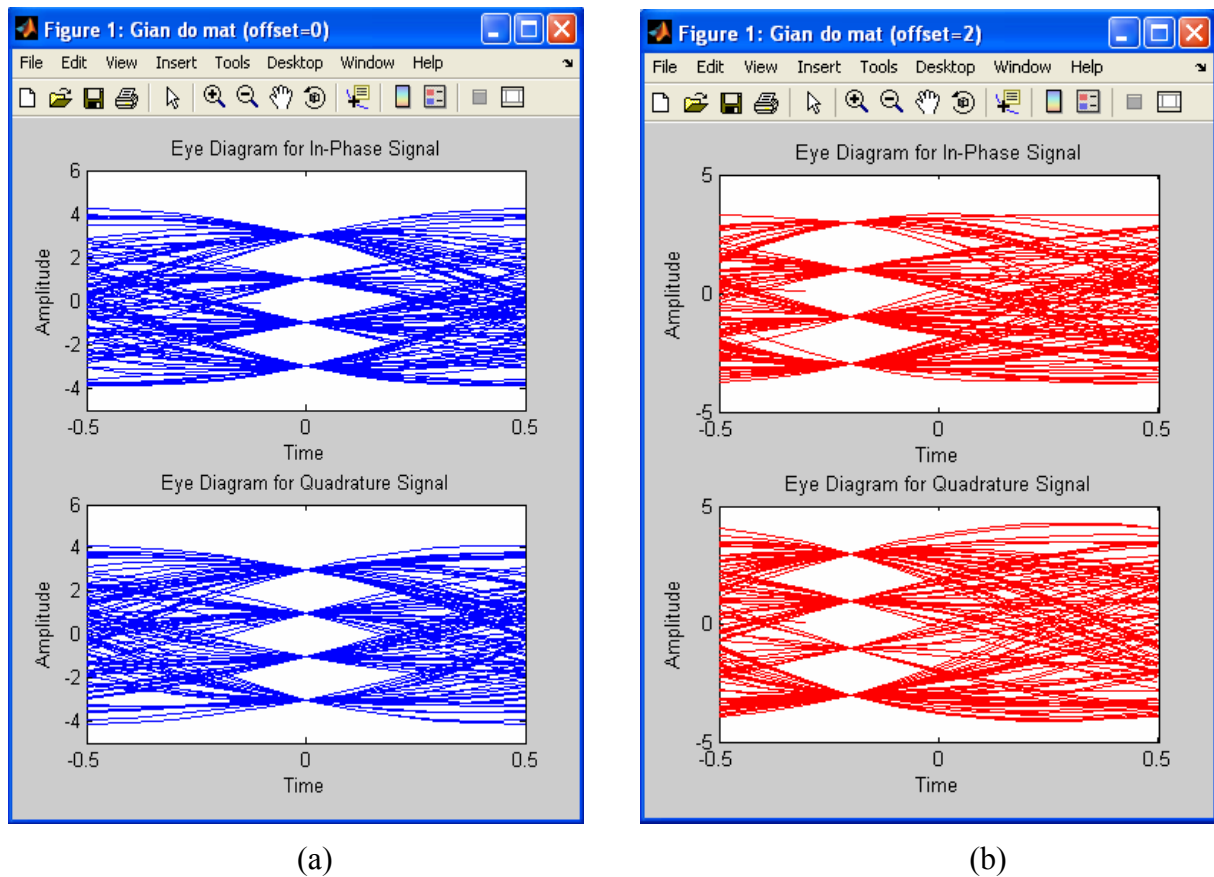
Bảng 16.2. Các dạng format của tín hiệu cần vẽ giản đồ mắt

Dạng tín hiệu	Thành phần In-phase	Thành phần Quadrature
Ma trận thực có hai cột	Cột thứ nhất	Cột thứ hai
Vector phức	Phần thực	Phần ảo
Vector thực	Nội dung của chính vector tín hiệu	Luôn luôn bằng 0

 **Ví dụ 16-9.** Vẽ giản đồ mắt cho tín hiệu thu sau khi truyền qua kênh truyền có đặc tính *raised-cosine*, phương pháp điều chế là 16-QAM.

```
% Định nghĩa các thông số
M = 16; Fd = 1; Fs = 10;
Pd = 100; % Số điểm tính toán
msg_d = randint(Pd,1,M); % Chuỗi dữ liệu ngẫu nhiên M mức
msg_a = qammod(msg_d,M); % Điều chế 16-QAM.
% Giả sử kênh truyền tương đương với một bộ lọc raised cosine.
delay = 3; % Thời gian trễ
rcv = rcosflt(msg_a,Fd,Fs,'fir/normal',.5,delay);
% Cắt bỏ phần đuôi của tín hiệu ra.
N = Fs/Fd;
propdelay = delay .* N + 1; % Trì hoãn truyền của bộ lọc
rcv1 = rcv(propdelay:end-(propdelay-1),:);
% Vẽ giản đồ mắt của tín hiệu thu với offset bằng 0
offset1 = 0;
h1 = eyediagram(rcv1,N,1/Fd,offset1);
set(h1,'Name','Gian do mat (offset=0)');
```

Giản đồ mắt thu được như sau (hình a):



Hình 16.7.

Nếu thay đổi giá trị offset bằng 2, ta được đồ thị như hình b.

16.8. ĐỒ THỊ PHÂN BỐ (SCATTER PLOT)

Đồ thị phân bố (scatter plot) của một tín hiệu biểu diễn sự phân bố các giá trị của tín hiệu đo tại các thời điểm quyết định. Thời điểm tối ưu là thời điểm mà giản đồ mắt mở rộng nhất. Đồ thị phân bố cũng cho biết các thông tin về kênh truyền. Nếu kênh truyền không nhiễu, các giá trị của tín hiệu sẽ tập trung tại các điểm ứng với các trạng thái của phương pháp điều chế. Kênh truyền có nhiễu càng lớn thì sự phân bố của các điểm tín hiệu thu càng rộng.

Để vẽ đồ thị phân bố, ta dùng hàm **scatterplot** với cú pháp như sau:

```
>> scatterplot(X,N,offset,plotstring,H)
```

Ý nghĩa các thông số tương tự như hàm **eyediagram**.

Ví dụ 16-10. Vẽ đồ thị phân bố cho tín hiệu thu tương tự ở ví dụ 16-9.

Phương pháp thực hiện tương tự như ví dụ 16-9, chỉ cần thay hàm **eyediagram** bằng hàm **scatterplot**.

```
% Định nghĩa các thông số.
```

```
M = 16; Fd = 1; Fs = 10; N = Fs/Fd;
```

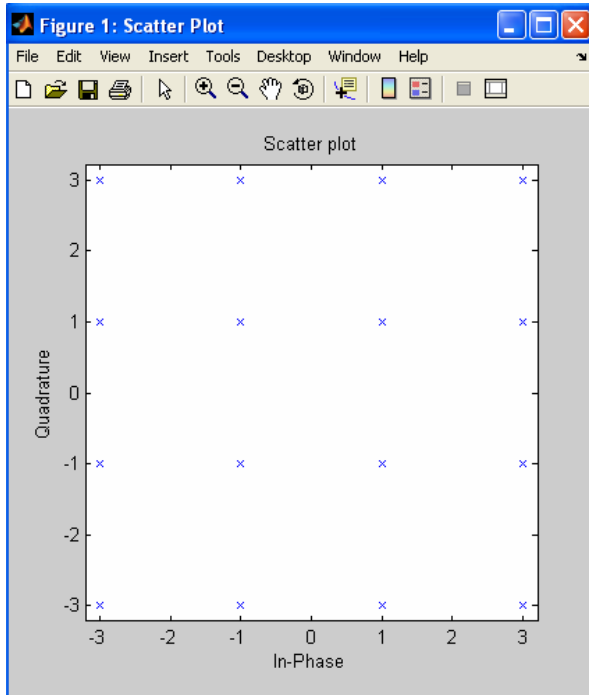
```
Pd = 200; % Số điểm tính toán
```

```
msg_d = randint(Pd,1,M); % Chuỗi dữ liệu ngẫu nhiên
```

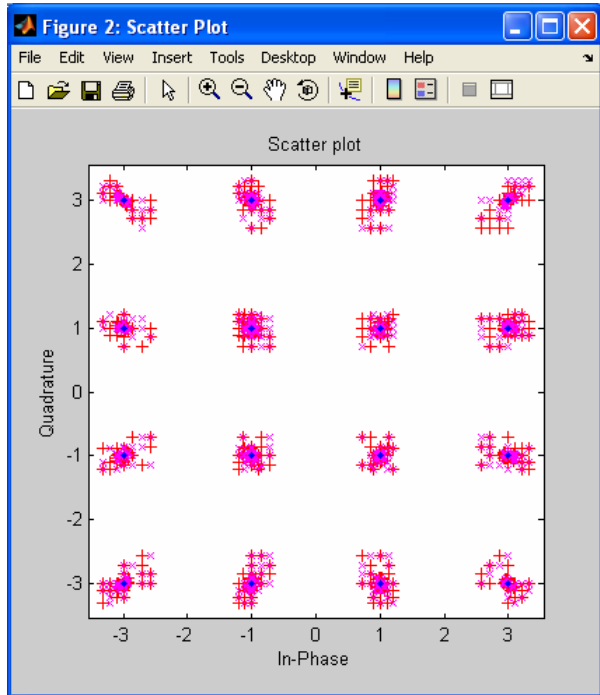
```
msg_a = qammod(msg_d,M); % Điều chế 16-QAM.
```

```
msg_a = rectpulse(msg_a,N); % Lấy mẫu lên đối với tín hiệu điều chế
```

```
% Giả sử đặc tính kênh truyền tương đương với bộ lọc raised-cosine
rcv = rcosflt(msg_a,Fd,Fs);
% Vẽ đồ thị phân bố cho tín hiệu thu, bỏ qua 3 ký hiệu đầu và 4 ký hiệu cuối.
rcv_a = rcv(3*N+1:end-4*N,:);
h = scatterplot(rcv_a,N,0,'bx');
```



(a)



(b)

Hình 16.8.

Kết quả được trình bày ở hình 16.8a ở trên. Trong trường hợp offset được chọn không tối ưu, ví dụ trong ba dòng lệnh sau đây, hai dòng 1 và 2 lần lượt minh họa các trường hợp offset được chọn quá trễ hoặc quá sớm. Dòng lệnh thứ ba minh họa trường hợp tối ưu. Đồ thị phân bố được vẽ lại như hình 16.8b.

```
hold on;
scatterplot(rcv_a,N,N+1,'r+',h); % Plot +'s
scatterplot(rcv_a,N,N-1,'mx',h); % Plot x's
scatterplot(rcv_a,N,0,'b.',h); % Plot dots
```

16.9. ĐÁNH GIÁ CHẤT LƯỢNG DÙNG KỸ THUẬT SEMIANALYTIC (BÁN PHÂN TÍCH)

Phương pháp đánh giá chất lượng kênh truyền thông qua mô phỏng là phương pháp thông dụng để phân tích hầu hết các hệ thống viễn thông. Tuy nhiên, nếu kênh truyền có tỷ lệ lỗi bit thấp thì để đánh giá chính xác cần phải mô phỏng trên một tập dữ liệu có kích thước khá lớn. Điều này đòi hỏi nhiều thời gian để thực thi toàn bộ chương trình mô phỏng. Trong những trường hợp này ta có thể sử dụng kỹ thuật bán phân tích (semianalytic) thay vì chỉ dùng mô phỏng. Kỹ thuật này cho phép rút ra kết quả nhanh hơn nhiều so với phương pháp sử dụng mô phỏng hoàn toàn.

Kỹ thuật bán phân tích là một sự kết hợp giữa mô phỏng và phân tích để đánh giá tỷ lệ bit lỗi của một hệ thống thông tin. Trong MATLAB, hàm **semianalytic** sẽ giúp người sử dụng thực hiện kỹ thuật này.

Tuy nhiên, kỹ thuật bán phân tích chỉ thích hợp với các hệ thống có đủ các tính chất sau:

- Tất cả các hiệu ứng đa đường, nhiễu lượng tử hoặc tính phi tuyến của mạch phải được khảo sát trước khi đánh giá tác động của nhiễu.
- Máy thu phải đồng bộ hoàn toàn với sóng mang từ máy phát. Các dạng nhiễu pha và timing jitter không được xét đến.
- Với kênh truyền không nhiễu thì tín hiệu thu không bị lỗi.
- Nhiễu trong hệ thống là nhiễu Gauss.

Trình tự thực hiện phương pháp này như sau:

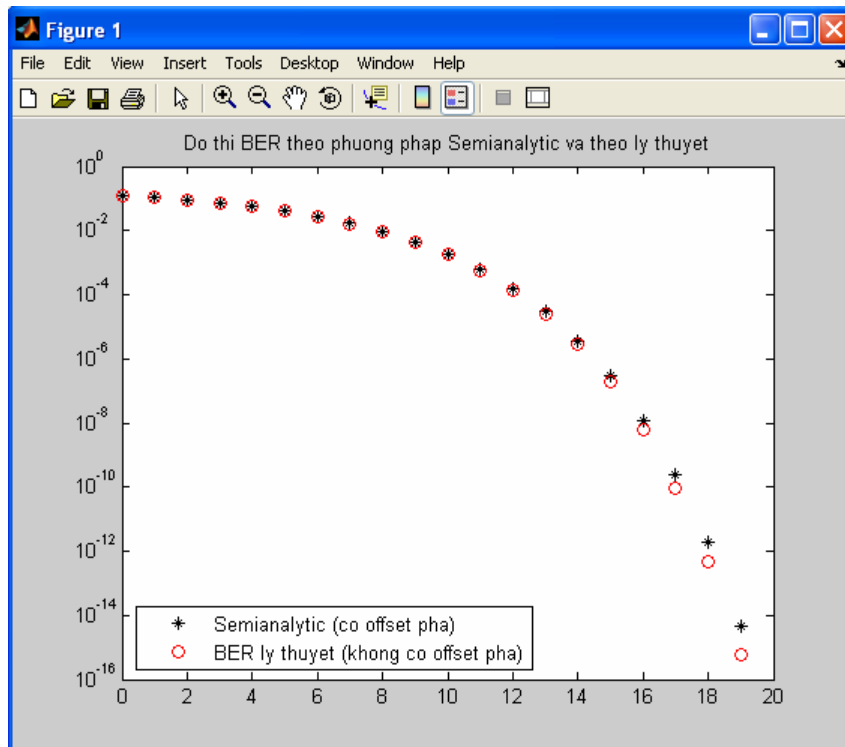
- Tạo một tín hiệu tin tức có ít nhất M^L ký hiệu, trong đó M là số trạng thái có thể có của tín hiệu điều chế và L là chiều dài đáp ứng xung của kênh truyền. Thông thường ta sử dụng một chuỗi giả ngẫu nhiên có chiều dài $(\log_2 M)M^L$, trong đó xác suất xuất hiện bit 0 và 1 là như nhau.
- Dùng tín hiệu này điều chế một sóng mang cao tần, sử dụng các phương pháp điều chế băng gốc (xem chương 14)
- Lọc tín hiệu phát bằng các bộ lọc raised cosine hoặc Butterworth, Chebychev,... Lưu lại tín hiệu phát `txsig`.
- Đưa tín hiệu qua kênh truyền không nhiễu. Kênh truyền có thể có các yếu tố như hiệu ứng đa đường, dịch pha, mạch phi tuyến, ... nhưng không có nhiễu. Lưu lại kết quả: `rxsig`.
- Dùng hàm **semianalytic** để thực hiện việc đánh giá kênh truyền. Hàm này sẽ thực hiện lọc tín hiệu thu và áp dụng phân bố Gauss trên mỗi điểm tín hiệu thu để tìm xác suất lỗi ký hiệu, sau đó chuyển thành xác suất lỗi bit.

 **Ví dụ 16-11.** Dùng kỹ thuật *semianalytic* để thực hiện mô phỏng một kênh thông tin số sử dụng điều chế 16-QAM. Vẽ đồ thị BER và so sánh với lý thuyết.

Trong thí dụ này, ta sẽ thấy được trình tự mô phỏng dùng phương pháp bán phân tích. Kết quả tính BER sẽ được so sánh với giá trị BER tính theo công thức lý thuyết.

```
% Bước 1. Tạo tín hiệu tin tức có chiều dài >= M^L.
M = 16; % Số trạng thái của tín hiệu điều chế
L = 1; % Chiều dài đáp ứng xung của kênh truyền
msg = [0:M-1 0]; % Thông điệp M-ary chiều dài > M^L
% Bước 2. Điều chế bằng phương pháp 16-QAM (điều chế băng gốc).
modsig = qammod(msg,M);
Nsamp = 16;
modsig = rectpulse(modsig,Nsamp); % Định dạng xung vuông.
% Bước 3. Sử dụng bộ lọc phát.
txsig = modsig; % Không dùng bộ lọc trong ví dụ này
% Bước 4. Đưa tín hiệu phát qua kênh truyền
rxsig = txsig*exp(j*pi/180); % Offset pha tính
```

```
% Bước 5. Sử dụng hàm semianalytic.
% Chọn bộ lọc thu bằng cách nhập các hệ số của hàm truyền s.
% Trong ví dụ này, sử dụng bộ tích phân lý tưởng
num = ones(Nsamp,1)/Nsamp;
den = 1;
EbNo = [0:20]; % Khoảng giá trị khảo sát của Eb/No
ber = semianalytic(txsig,rxsig,'qam',M,Nsamp,num,den,EbNo);
% Tính BER theo lý thuyết.
bertheory = berawgn(EbNo,'qam',M);
% Vẽ đồ thị BER theo phương pháp bán phân tích và theo lý thuyết.
figure; semilogy(EbNo,ber,'k*');
hold on; semilogy(EbNo,bertheory,'ro');
title('Đồ thị BER theo phương pháp Semianalytic và theo lý thuyết');
legend('Semianalytic','BER lý thuyết','Location','SouthWest');
hold off;
```



Hình 16.9.

✎ Bài tập 16-1.

Vẽ đồ thị BER ứng với các giá trị SNR khác nhau khi truyền tín hiệu qua kênh truyền AWGN bằng hai phương pháp điều chế QPSK và BPSK. Chuỗi bit dữ liệu có chiều dài bằng 1000.

✎ Bài tập 16-2.

Làm lại bài tập trên nhưng vẽ thêm dạng sóng các tín hiệu ở các điểm khác nhau trong hệ thống (chỉ vẽ cho 10 bit đầu tiên)

✎ Bài tập 16-3.

Làm lại ví dụ 16-4 nhưng có xét đến trì hoãn kênh truyền.

✎ **Bài tập 16-4.**

Xét ví dụ 16-4. Với SNR bằng 5dB. Hãy thực hiện mô phỏng và vẽ dạng sóng các tín hiệu trên kênh truyền fading cho trong ví dụ 16-4. Phương pháp điều chế là:

a. BPSK

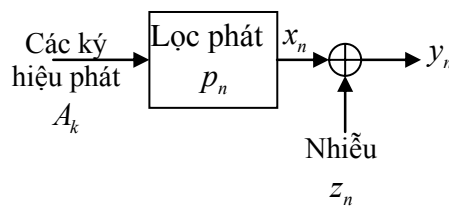
b. FSK

✎ **Bài tập 16-5.**

Khảo sát phương pháp mã hoá BCH (7,4) bằng cách sử dụng mô hình kênh truyền đảo bit nhị phân với số bit lỗi trong mỗi từ mã bằng 1 với xác suất 0.6 và bằng 2 với xác suất 0.3, bằng 3 với xác suất 0.1.

✎ **Bài tập 16-6.**

Khảo sát một hệ thống thông tin như sau:



Chuỗi ký hiệu vào là chuỗi 4-PAM với tập nguồn là $\{-3 -1 1 3\}$, chiều dài 100 ký hiệu.

Bộ lọc sửa dạng xung là bộ lọc Raised-cosine có đáp ứng xung là:

$$p(t) = \left[\frac{\sin(\pi t / T)}{(\pi t / T)} \right] \cdot \frac{\cos(\alpha \pi t / T)}{1 - (2\alpha t / T)^2}$$

Với T là chu kỳ ký hiệu, bằng 8000 ký hiệu/s, tốc độ lấy mẫu 24000Hz, hệ số roll-off $\alpha = 0.5$.

Tìm tỷ lệ lỗi bit nếu nhiễu là nhiễu Gauss với variance lần lượt là:

a. 0.01

b. 0.1

c. 1

✎ **Bài tập 16-7.**

Làm lại bài tập 15-5 nhưng có khảo sát ảnh hưởng của nhiễu. Xét hai trường hợp:

a. Chỉ có nhiễu AWGN

b. Nhiễu AWGN và fading Rayleigh (tùy chọn thông số cụ thể).

✎ **Bài tập 16-8.**

Vẽ và so sánh đồ thị BER của kênh truyền AWGN theo ba phương pháp: lý thuyết, mô phỏng và bán phân tích. Các giá trị SNR lần lượt là 2, 4, 6, 8, 10 dB. Chuỗi bit dữ liệu có chiều dài bằng 100000. Các phương pháp điều chế được sử dụng là:

a. BPSK

b. QPSK

c. MSK

✎ **Bài tập 16-9.**

Mô phỏng kênh truyền AWGN với số bit mô phỏng là 10^6 để đánh giá tỷ lệ bit lỗi của các phương pháp điều chế BPSK, QPSK, FSK, MSK, và 16QAM. Công cụ để đánh giá là đồ thị BER.

✎ **Bài tập 16-10.**

Làm lại ví dụ 16-4 nhưng có xét đến trì hoãn kênh truyền

✎ **Bài tập 16-11.**

Mô phỏng kênh truyền có nhiễu AWGN với các giá trị Eb/No bằng 0, 2, 4 dB và vẽ giản đồ mắt cho mỗi trường hợp. Phương pháp điều chế là 16QAM.

✎ **Bài tập 16-12.**

Vẽ giản đồ mắt cho các trường hợp nhiễu khác nhau với hệ thống cho trong bài tập 16-6.

✎ **Bài tập 16-13.**

Với Eb/No bằng 2 dB, hãy vẽ đồ thị phân bố cho các phương pháp điều chế 8PSK và 8QAM. Quá trình mô phỏng chỉ kết thúc khi số bit lỗi lớn hơn 100.

Danh sách các hàm được giới thiệu trong chương 16

Các hàm mô phỏng kênh truyền

awgn	Tạo nhiễu AWGN và cộng vào tín hiệu phát
bsc	Mô phỏng kênh truyền đảo bit nhị phân
copy	Sao chép một đối tượng kênh truyền fading
randn	Tạo ma trận ngẫu nhiên có trung bình bằng 0 và phương sai bằng 1
rayleighchan	Tạo một đối tượng mô tả kênh truyền fading Rayleigh
reset	Reset các thuộc tính của một đối tượng kênh truyền fading
ricianchan	Tạo một đối tượng mô tả kênh truyền fading Rician
wgn	Tạo ma trận nhiễu ngẫu nhiên

Các hàm đánh giá chất lượng kênh truyền

berawgn	Xác suất lỗi của kênh truyền AWGN không mã hoá (theo lý thuyết)
bercoding	Xác suất lỗi của kênh truyền AWGN có mã hoá (theo lý thuyết)
berconfint	Tính xác suất lỗi với độ tin cậy cho trước
berfading	Xác suất lỗi của kênh truyền fading không mã hoá (theo lý thuyết)
berfit	Vẽ và làm trơn đồ thị BER từ các giá trị thực nghiệm thô
bersync	Xác suất lỗi của kênh truyền AWGN không mã hoá, đồng bộ không chính xác
biterr	Tính xác suất lỗi bit
eyediagram	Vẽ giản đồ mắt
scatterplot	Vẽ đồ thị phân bố
semianalytic	Đánh giá kênh truyền theo phương pháp semianalytic và trả về xác suất lỗi
symerr	Tính xác suất lỗi ký hiệu

Chương 17

MÃ HÓA KÊNH TRUYỀN

Mã hoá sửa sai là kỹ thuật phát hiện các lỗi xuất hiện khi dữ liệu được truyền từ máy phát đến máy thu, đồng thời có thể sửa các lỗi này để thông tin không bị sai lệch. Nội dung cơ bản của các kỹ thuật mã hoá là: bên cạnh các bit hay ký hiệu mang thông điệp cần truyền đi, bộ mã hoá sẽ phát thêm một hoặc nhiều ký hiệu dư thừa có quan hệ với các ký hiệu mang tin. Bộ giải mã sẽ dựa vào các ký hiệu dư thừa này để phát hiện các lỗi trong chuỗi ký hiệu nhận được và có thể sửa các lỗi này.

Cho đến nay các nhà nghiên cứu đã tìm ra rất nhiều kỹ thuật mã hoá sửa sai khác nhau, nhưng nhìn chung có hai nhóm chính là mã khối và mã chập.

17.1. MÃ KHỐI

Mã khối là một trường hợp đặc biệt của mã hoá sửa sai. Kỹ thuật mã khối thực hiện phép ánh xạ từ một số lượng cố định các ký hiệu thông tin thành một số cố định các ký hiệu của từ mã đã được mã hoá. Bộ mã hoá sẽ xử lý mỗi khối dữ liệu một cách độc lập. Nó là một thiết bị không nhớ.

Các kỹ thuật mã khối tuyến tính được phân chia thành các loại theo sơ đồ dưới đây:



Hình 17.1. Phân loại mã khối

Dưới đây là danh sách các hàm được MATLAB cung cấp để thực hiện các kỹ thuật mã khối tuyến tính nêu trên:

Bảng 17.1. Các hàm MATLAB thực hiện mã khối tuyến tính

Kỹ thuật mã hoá	Danh sách các hàm MATLAB
Mã khối tuyến tính	encode, decode, gen2par, syndtable
Mã vòng	encode, decode, cyclpoly, cyclgen, gen2par, syndtable
Mã BCH	bchenc, bchdec, bchgenpoly
Mã Hamming	encode, decode, hammgen, gen2par, syndtable
Mã Reed-Solomon	rsenc, rsdec, rsgenpoly, rsencof, rsdecof

Các hàm này thực hiện các tác vụ sau:

- Mã hoá và giải mã một thông điệp dùng một trong các kỹ thuật mã hoá nêu trên.

○ Xác định các đặc trưng của kỹ thuật mã hoá tương ứng, ví dụ khả năng sửa lỗi hoặc chiều dài hợp lệ của thông điệp, ...

○ Tính toán các yếu tố của từng kỹ thuật mã hoá, bao gồm bảng giải mã, đa thức sinh, ma trận kiểm tra, chuyển đổi giữa đa thức sinh và ma trận kiểm tra,...

Trong kỹ thuật mã khối tuyến tính, mỗi thông điệp sẽ được chia thành các khối nhỏ gồm k ký hiệu, mỗi khối này sẽ được chèn thêm vào các ký hiệu để được một từ mã chiều dài n ký hiệu. Ta gọi k là chiều dài phần thông điệp, n là chiều dài từ mã, và phép mã hoá này được ký hiệu là mã $[n,k]$.

17.1.1. BIỂU DIỄN MỘT PHẦN TỬ TRONG TRƯỜNG GALOIS

▪ Vì lý thuyết trường Galois là cơ sở của các kỹ thuật mã hoá sửa sai nên trước tiên cần giới thiệu cách biểu diễn một phần tử trong trường Galois khi sử dụng MATLAB.

▪ Nhắc lại một số thuật ngữ cơ bản có liên quan:

Một *phần tử nguyên* (*primitive element*) của trường $GF(2^m)$ là một phần tử sinh của một nhóm cyclic các phần tử khác không trong $GF(2^m)$. Nói cách khác, mọi phần tử khác không của trường $GF(2^m)$ đều có thể biểu diễn dưới dạng một lũy thừa cơ số nguyên của phần tử nguyên nói trên.

Một *đa thức nguyên* (*primitive polynomial*) của trường $GF(2^m)$ là một đa thức tối tiểu của một phần tử nguyên nào đó trong $GF(2^m)$. Đó là một đa thức với các hệ số nhị phân có bậc nhỏ nhất (khác 0) nhận phần tử nguyên làm nghiệm trong $GF(2^m)$. Một hệ quả rút ra từ định nghĩa này, đó là: đa thức nguyên sẽ có bậc bằng m và là đa thức tối giản.

▪ Trong MATLAB, để biểu diễn một dãy Galois, ta có thể dùng hàm **gf**. Hàm này tạo ra một biến mà MATLAB xem đó như là một dãy Galois, thay vì là một dãy các số nguyên bình thường. Do đó các phép toán mà MATLAB thực hiện trên dãy này là các phép toán trong trường Galois. Hàm **gf** cần được cung cấp các thông số nhập như sau:

○ Dữ liệu của trường Galois, x , là một dãy các số nguyên nằm trong khoảng 0 đến $2^m - 1$.

○ Một số nguyên m cho biết x thuộc trường $GF(2^m)$. Giá trị hợp lệ của m là từ 1 đến 16. Thông số này không bắt buộc phải có, trong trường hợp không có thì xem như nó nhận giá trị bằng 1 (trường $GF(2)$).

○ Một số nguyên dương cho biết ta sử dụng đa thức nguyên nào để biểu diễn x . Thông số này cũng không nhất thiết phải có.

📖 **Ví dụ 17-1.** Tạo một dãy Galois trong trường $GF(16)$ bằng 2 cách:

1. Dùng đa thức nguyên mặc định của MATLAB

2. Dùng đa thức nguyên $D^4 + D^3 + 1$

```
>> x=[0:15];
```

```
>> a=gf(x,4) % Tạo dãy Galois 16 phần tử trong trường GF(16)
```

```
% Đa thức nguyên mặc định
```

```
a = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)
```

```
Array elements =
```

```
Columns 1 through 13
```

```
0    1    2    3    4    5    6    7    8    9   10   11   12
```

```

Columns 14 through 16
13      14      15
>> b=gf(15,4); % Tạo một dãy Galois 15 phần tử trong GF(16)
>> c=gf(x,4,25) % Tạo dãy Galois 16 phần tử trong GF(16)
           % ... với đa thức nguyên D^4 + D^3 + 1 (25 = 11001)
c = GF(2^4) array. Primitive polynomial = D^4+D^3+1 (25 decimal)
Array elements =
Columns 1 through 13
0      1      2      3      4      5      6      7      8      9      10     11     12
Columns 14 through 16
13      14      15

```

17.1.2. MÃ REED-SOLOMON

▪ Mã Reed-Solomon sử dụng các ký hiệu m bit thay vì dùng các bit. Một thông điệp cần mã hoá bởi mã Reed-Solomon $[n,k]$ sẽ được biểu diễn bởi một dãy Galois k cột trong trường Galois $GF(2^m)$. Mỗi phần tử của dãy là một số nguyên trong khoảng từ 0 đến $2^m - 1$. Như vậy, từ mã tương ứng với thông điệp này sẽ được biểu diễn bằng một dãy Galois n cột trong trường Galois $GF(2^m)$. Chiều dài n của từ mã nằm giữa 3 và $2^m - 1$.

Ví dụ:

```

>> n=7;k=3;
>> msg = gf([1 6 4; 0 4 3],3); % Thông điệp là 1 dãy Galois trong trường GF(8).

```

▪ Các thông số của mã Reed-Solomon:

Giá trị cho phép của các thông số cho mã Reed-Solomon đối với MATLAB được mô tả trong bảng dưới đây:

Bảng 17.2. Giá trị cho phép của các thông số mã Reed-Solomon trong MATLAB

Ký hiệu	Ý nghĩa	Giá trị hoặc phạm vi biến thiên
m	Số bit trên một ký hiệu	Số nguyên từ 3 đến 16
n	Số ký hiệu của một từ mã	Số nguyên từ 3 đến $2^m - 1$
k	Số ký hiệu của một mẫu tin	Số nguyên dương nhỏ hơn n , sao cho $n - k$ là số chẵn
t	Khả năng sửa lỗi của bộ mã	$(n - k)/2$

▪ Đa thức sinh:

Để mã hoá một thông điệp, ta phải dựa trên một đa thức sinh cho trước. Đối với mã Reed-Solomon, MATLAB cung cấp sẵn các đa thức sinh mặc định tùy theo giá trị $[n,k]$. Tuy nhiên, người sử dụng cũng có thể tạo một đa thức sinh theo ý mình, hoặc thực hiện các thao tác trên các đa thức sinh bằng cách dùng hàm **rsгенpoly**. Hàm này trả về một vector hàng liệt kê các hệ số của đa thức sinh theo thứ tự số mũ giảm dần. Đa thức sinh này có dạng $(X - A)(X - A^2) \dots (X - A^{N-K})$, trong đó A là một nghiệm của một đa thức nguyên mặc định của trường $GF(N+1)$ ($N = 2^m - 1$). Ngoài ra, hàm này còn cho biết khả năng sửa lỗi T của bộ mã. Cú pháp như sau:

```
>> [genpoly,T] = rsgenpoly(N,K,prim_poly,B)
```

Ngoài hai thông số cơ bản của bộ mã là N và K , người sử dụng có thể cung cấp thêm hai thông số (không bắt buộc):

- o **prim_poly**: xác định đa thức nguyên trong trường $GF(N+1)$ nhận A là nghiệm thay vì dùng đa thức nguyên mặc định của MATLAB. **prim_poly** là một số nguyên mà biểu diễn nhị phân của nó chính là các hệ số của đa thức nguyên nói trên theo thứ tự số mũ giảm dần (ví dụ: **prim_poly** = 13 = 1101_2 sẽ biểu diễn đa thức nguyên $D^3 + D^2 + 1$)

- o **B**: nếu cung cấp thêm thông số này thì hàm **rsgenpoly** sẽ trả về đa thức $(X - A^B)(X - A^{B+1})...(X - A^{B+N-K-1})$ (B là một số nguyên).

Ví dụ:

```
>> r = rsgenpoly(15,13)
r = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)
Array elements =
      1      6      8
```

trả về đa thức $X^2 + (A^2 + A)X + A^3$ với A là nghiệm của đa thức nguyên trong $GF(16)$.

▪ Mã hoá và giải mã Reed-Solomon:

Để mã hoá một thông điệp **msg** dùng bộ mã Reed-Solomon $[n,k]$, ta sử dụng hàm **rsenc** với cú pháp sau:

```
>> code = rsenc(msg,n,k,genpoly,paritypos)
```

genpoly và **paritypos** là hai thông số không bắt buộc; **genpoly** là một vector hàng gồm các hệ số của đa thức sinh của phép mã hoá, **paritypos** cho biết vị trí của các ký hiệu kiểm tra thêm vào thông điệp là ở đầu ('beginning') hay ở cuối ('end'). Giá trị mặc định là 'end'.

Thông điệp mã hoá sẽ được giải mã bằng hàm **rsdec**:

```
>> [decoded,cnumerr,ccode] = rsdec(code,n,k,genpoly,paritypos)
```

decoded là một dãy Galois biểu diễn thông điệp đã giải mã. **cnumerr** là một vector cột, mỗi phần tử của nó là số lỗi đã được sửa khi giải mã hàng tương ứng trong thông điệp mã hoá. Nếu số lỗi lớn hơn $(n-k)/2$, bộ mã không có khả năng sửa lỗi. Khi đó, số lỗi trả về là -1 và **decoded** sẽ lấy hàng tương ứng của **code** và bỏ đi $n-k$ ký hiệu cuối. **ccode** có cùng format như **code**, nó biểu diễn thông điệp mã hoá chính xác (tức là dãy **code** sau khi đã sửa những chỗ sai). Nếu bộ mã không sửa lỗi được ở hàng nào thì trong dãy **ccode**, hàng đó sẽ không thay đổi so với dãy **code**.

Sau đây là một số ví dụ minh hoạ vấn đề mã hoá và giải mã Reed-Solomon:

 **Ví dụ 17-2.** Mã hoá một thông điệp gồm 4 từ bằng mã Reed-Solomon $[7,3]$. Tạo lỗi ngẫu nhiên và cộng vào thông điệp mã hoá. Giải mã và so sánh với thông điệp ban đầu.

Trong thí dụ này ta tạo các lỗi ngẫu nhiên nhưng thoả mãn điều kiện tổng số lỗi trong mỗi hàng đúng bằng $t = (n - k)/2 = 2$ lỗi. Như vậy bộ mã có khả năng sửa tất cả các lỗi sai.

```
m = 3; % Số bit trên một ký hiệu
n = 2^m-1; k = 3; % Chiều dài từ mã và chiều dài thông điệp
t = (n-k)/2; % Khả năng sửa lỗi của bộ mã
```

```

nw = 4; % Tổng số từ của thông điệp
msgw = gf(randint(nw,k,2^m),m); % Tạo thông điệp ngẫu nhiên
genpoly = rsgenpoly(n,k); % Tạo đa thức sinh
c = rsenc(msgw,n,k,genpoly); % Mã hoá dữ liệu.
noise = (1+randint(nw,n,2^m-1)).*randerr(nw,n,t); % t lỗi trên mỗi hàng
cnoisy = c + noise; % Công nhiễu vào mã.
[dc,nerrs,corrcode] = rsdec(cnoisy,n,k,genpoly); % Giải mã.
% Kiểm tra xem bộ giải mã có hoạt động tốt không.
isequal(dc,msgw) & isequal(corrcode,c)
nerrs % Số lỗi đã được sửa.
msgw % Thông điệp ban đầu
cnoisy % Thông điệp mã hoá đã bị lỗi
corrcode % Thông điệp mã hoá đúng
dc % Thông điệp giải mã

```

Kết quả xuất hiện trên cửa sổ lệnh của MATLAB như sau:

```

ans =
    1
nerrs =
    2
    2
    2
    2
msgw = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    7     7     6
    1     6     3
    4     3     4
    3     0     6
cnoisy = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    0     7     6     4     4     5     4
    1     1     0     6     4     3     1
    4     3     4     5     2     6     2
    3     0     6     5     5     5     3
corrcode = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    7     7     6     4     6     5     4
    1     6     3     6     4     3     1
    4     3     4     5     3     2     2
    3     0     6     0     5     6     3

```

```
dc = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    7     7     6
    1     6     3
    4     3     4
    3     0     6
```

Ví dụ sau minh họa trường hợp bộ mã không có khả năng sửa lỗi:

 **Ví dụ 17-3.** Mã hoá một thông điệp gồm 3 từ bằng mã Reed-Solomon [7,3]. Tạo lỗi sao cho bộ mã không có khả năng sửa lỗi. Giải mã và so sánh với thông điệp ban đầu.

```
n=7; k=3; % Chiều dài từ mã và thông điệp
m=3; % Số bit trên một ký hiệu
msg = gf([7 4 3;6 2 2;3 0 5],m) % Thông điệp gồm 3 từ
code = rsenc(msg,n,k); % Mã hoá
% Tạo 1 lỗi ở từ thứ 1, 2 lỗi ở từ thứ 2, 3 lỗi ở từ thứ 1
errors = gf([3 0 0 0 0 0 0;4 5 0 0 0 0 0;6 7 7 0 0 0 0],m);
codeNoi = code + errors
[dec,cnumerr,ccode] = rsdec(codeNoi,n,k) % Giải mã sai: cnumerr(3)= -1
```

Kết quả thực hiện chương trình:

```
msg = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    7     4     3
    6     2     2
    3     0     5

codeNoi = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    4     4     3     7     0     0     4
    2     7     2     7     6     7     3
    5     7     2     5     6     0     6

dec = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    7     4     3
    6     2     2
    5     7     2

cnumerr =
    1
    2
   -1

ccode = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
    7     4     3     7     0     0     4
```

6	2	2	7	6	7	3
5	7	2	5	6	0	6

Hàng thứ 3 bị sai: hàng thứ 3 của dec chính là 3 phần tử đầu của hàng thứ 3 của code, hàng thứ 3 của ccode cũng là hàng thứ 3 của code.

17.1.3. MÃ BCH

- Biểu diễn thông điệp và từ mã đối với mã BCH:

Muốn mã hoá một thông điệp bằng mã BCH thì trước hết thông điệp phải được biểu diễn bằng một dãy Galois nhị phân (trường GF(2)) gồm k cột. Tương ứng với thông điệp này là từ mã được biểu diễn bằng một dãy Galois nhị phân n cột. Mỗi một dòng của dãy Galois này ứng với một từ của thông điệp.

Đối với mã BCH, n phải là một số nguyên có dạng $2^m - 1$, với m là một số nguyên lớn hơn 2; k là một số nguyên nhỏ hơn n, và với mỗi n, k chỉ có thể nhận một vài giá trị. Các giá trị cho phép của k tùy theo n được trình bày trong bảng sau với các giá trị $n < 1000$.

Bảng 17.3. Các giá trị của n và k đối với mã BCH

Giá trị của n	Các giá trị cho phép của k
7	4
15	5, 7, 11
31	6, 11, 16, 21, 26
63	7, 10, 16, 18, 24, 30, 36, 39, 45, 51, 57
127	8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99
255	9, 13, 21, 29, 37, 45, 47, 55, 63, 71, 79, 87, 91, 99, 107, 115, 123, 131, 139, 147, 155, 163, 171, 179, 187
511	10, 19, 28, 31, 40, 49, 58, 67, 76, 85, 94, 103, 112, 121, 130, 139, 148, 157, 166, 175, 184, 193, 202, 211, 220, 229, 238, 241, 250, 259, 268, 277, 286, 295, 304, 313, 322, 331, 340, 349, 358, 367, 376, 385, 394, 403, 412, 421, 430, 439, 448, 457, 466, 475, 484, 493, 502

- Mã hoá và giải mã BCH:

Để mã hoá và giải mã thông điệp dùng mã BCH ta sử dụng hai hàm **bchenc** và **bchdec** với cú pháp tương tự như các hàm mã hoá và giải mã Reed-Solomon, chỉ khác ở chỗ các thông điệp và từ mã là các dãy Galois trong trường GF(2) thay vì GF(2^m).

```
>> code = bchenc(msg,n,k)
```

```
>> [decoded,cnumerr,ccode] = bchdec(code,n,k,paritypos)
```

Hàm **bchgenpoly** tạo ra đa thức sinh cho mã BCH [n,k] đồng thời trả về khả năng sửa lỗi t của bộ mã.

```
>> [genpoly,t] = bchgenpoly (n,k,prim_poly)
```

 **Ví dụ 17-4.** Làm lại ví dụ 17-2 với phương pháp mã hoá được sử dụng là BCH [15,5].

```
n = 15; k = 5; % Chiều dài từ mã và thông điệp
```

```
[gp,t] = bchgenpoly(n,k); % t là khả năng sửa lỗi.
```

```
nw = 4; % Tổng số từ của thông điệp
```

```
msgw = gf(randint(nw,k)) % Tạo thông điệp ngẫu nhiên
```

```

c = bchenc(msgw,n,k); % Mã hoá dữ liệu.
noise = randerr(nw,n,t); % Mỗi hàng có t lỗi, nhiều là ma trận nhị phân
cnoisy = c + noise % Cộng nhiễu vào các từ mã.
[dc,nerrs,corrcode] = bchdec(cnoisy,n,k) % Giải mã thông điệp.
% Kiểm tra hoạt động của bộ giải mã.
chk2 = isequal(dc,msgw) & isequal(corrcode,c)
nerrs % Số lỗi đã được sửa ứng với mỗi từ của thông điệp.

```

Kết quả thực hiện chương trình như sau (msgw: thông điệp gốc, cnoisy: thông điệp mã hoá bị lỗi, dc: thông điệp sau khi giải mã):

```
msgw = GF(2) array.
```

```
Array elements =
```

1	1	1	0	1
1	0	1	1	1
0	1	1	0	0
1	1	1	0	0

```
cnoisy = GF(2) array.
```

```
Array elements =
```

```
Columns 1 through 13
```

1	0	1	0	0	0	1	1	0	0	1	0	1
1	0	1	0	1	1	0	0	0	1	1	1	0
0	1	1	0	0	1	0	0	0	1	0	1	1
1	1	1	0	0	1	0	1	0	1	0	0	0

```
Columns 14 through 15
```

0	1
0	1
1	0
1	1

```
dc = GF(2) array.
```

```
Array elements =
```

1	1	1	0	1
1	0	1	1	1
0	1	1	0	0
1	1	1	0	0

```
nerrs =
```

```
3 3 3 3
```

```
corrcode = GF(2) array.
```

```
Array elements =
```

```
Columns 1 through 13
```

1	1	1	0	1	0	1	1	0	0	1	0	0
1	0	1	1	1	0	0	0	0	1	0	1	0

0	1	1	0	0	1	0	0	0	1	1	1	1
1	1	1	0	0	0	0	1	0	1	0	0	1

Columns 14 through 15

0	1
0	1
0	1
1	0

chk2 =

1

nerres =

3 3 3 3

17.1.4. MÃ KHỐI TUYẾN TÍNH

Ngoài mã Reed-Solomon và mã BCH, các loại mã khối tuyến tính còn lại, bao gồm mã vòng, mã Hamming có cách biểu diễn gần giống nhau trong MATLAB. Do đó trong phần này chúng ta sẽ khảo sát chung các loại mã này.

▪ Biểu diễn mã khối tuyến tính: các từ mã và thông điệp có thể được biểu diễn bởi một trong ba dạng sau đây:

○ *Dạng vector nhị phân*: một thông điệp có thể được biểu diễn bằng một vector mà các phần tử của nó là các bit 0 hoặc 1. Với phép mã hoá $[n,k]$, thông điệp sẽ được chia thành các vector k phần tử, mỗi vector này được mã hoá để tạo thành một vector từ mã gồm n phần tử. Cuối cùng, thông điệp mã hoá là vector nhị phân được ghép bởi các vector từ mã này.

○ *Dạng ma trận nhị phân*: thông điệp gốc và thông điệp mã hoá được tổ chức dưới dạng ma trận các số nhị phân. Mỗi hàng của ma trận ứng với một từ của thông điệp hoặc từ mã tương ứng của nó. Như vậy, với phép mã hoá $[n,k]$, thông điệp gốc sẽ được biểu diễn bằng ma trận nhị phân gồm k cột còn thông điệp mã hoá được biểu diễn bằng ma trận nhị phân n cột. Các bit parity nằm ở các vị trí đầu của mỗi hàng.

○ *Dạng vector thập phân*: các thông điệp và từ mã cũng có thể biểu diễn dưới dạng một vector mà mỗi phần tử của nó là biểu diễn thập phân của các bits trong một từ của thông điệp hoặc các bits của một từ mã.

Lưu ý: nếu 2^n hay 2^k quá lớn, ta nên dùng dạng vector nhị phân thay vì dạng vector thập phân, vì về bản chất, các hàm mã hoá được thực hiện trên các bit nhị phân, và ta có thể gặp phải sai số khi chuyển đổi từ một số nhị phân có nhiều bit thành số thập phân.

Ngoài ra, cần lưu ý là nếu dùng dạng vector thập phân thì hàm mã hoá **encode** sẽ coi bit tận cùng bên trái là bit có trọng số thấp nhất.

▪ Các thông số của mã khối tuyến tính: bao gồm ma trận sinh, ma trận kiểm tra, bảng giải mã và đa thức sinh.

○ *Ma trận sinh*: quá trình mã hoá một thông điệp bằng mã khối tuyến tính $[n,k]$ được thực hiện nhờ một ma trận sinh G có kích thước $k \times n$. Mỗi từ mã là một vector kích thước $1 \times n$, kết quả của phép nhân vector kích thước $1 \times k$ biểu diễn thông điệp với ma trận G . Nếu G có dạng $[I_k \ P]$ hoặc $[P \ I_k]$, trong đó P là một ma trận kích thước $k \times (n-k)$ còn I_k là ma trận đơn vị cấp k thì G được gọi là ma trận sinh dạng chính tắc.

○ *Ma trận kiểm tra*: để giải mã một thông điệp được mã hoá bằng mã khối tuyến tính $[n, k]$, ta dựa vào một ma trận kích thước $(n-k) \times n$, gọi là ma trận kiểm tra H . Ma trận này thoả mãn phương trình $G \cdot H^T = 0 \pmod{2}$, với G là đa thức sinh còn H^T là ma trận chuyển vị của H . Nếu $G = [I_k \ P]$ thì $H = [-P^T \ I_{n-k}]$; tương tự, nếu $G = [P \ I_k]$ thì $H = [I_{n-k} \ -P^T]$. Đối với mã nhị phân thì dấu trừ trước P^T không có ý nghĩa vì $-1 = 1 \pmod{2}$.

○ *Đa thức sinh (đối với mã vòng)*: mã vòng có một tính chất đại số đặc biệt, đó là: quá trình mã hoá có thể được xác định hoàn toàn bởi một đa thức trong trường $GF(2)$, gọi là đa thức sinh. Đó là một đa thức bậc $n - k$ và là ước của đa thức $X^n - 1$.

Trong MATLAB, hàm **cyclpoly** được dùng để tạo ra các đa thức sinh cho mã vòng $[n, k]$:

```
>> pol = cyclpoly(n, k, opt)
```

Hàm này trả về một vector nhị phân gồm các hệ số của đa thức sinh liệt kê theo thứ tự tăng dần. `opt` là một thông số phụ, có thể nhận các giá trị sau

Nếu `opt = 'min'` : tạo đa thức sinh có trọng số (tức số các hệ số bằng 1) nhỏ nhất.

Nếu `opt = 'max'` : tạo đa thức sinh có trọng số lớn nhất.

Nếu `opt = 'all'` : tạo tất cả các đa thức sinh có thể, mỗi đa thức được biểu diễn trên một hàng.

Nếu `opt = L` : tạo tất cả đa thức sinh có trọng số bằng L .

Nếu không có đa thức nào thoả điều kiện ràng buộc thì `pol = []`.

Ví dụ:

```
>> genpoly=cyclpoly(7,3,'all')
```

```
genpoly =
```

```
1      0      1      1      1
1      1      1      0      1
```

Lệnh trên tạo ra hai đa thức sinh: $1 + X^2 + X^3 + X^4$ và $1 + X + X^2 + X^4$. Ta có cùng kết quả nếu dùng **cyclpoly** (7, 3, 4).

Ma trận sinh H và ma trận kiểm tra G được tạo bằng các hàm **hammgen** (mã Hamming) và **cyclgen** (mã vòng):

```
>> [H, G, k] = cyclgen (n, p, opt)
```

với p là đa thức sinh của mã vòng, `opt` là một thông số không bắt buộc cho biết ma trận kiểm tra H có tính hệ thống (tức là có thể viết dưới dạng $[P^T \ I_{n-k}]$ hoặc $[I_{n-k} \ P^T]$) hay không.

Nếu `opt = 'system'` (đây là trường hợp mặc định) thì H có tính hệ thống, ngược lại nếu `opt = 'nonsys'` thì H không có tính hệ thống.

```
>> [H, G, n, k] = hammgen (m)
```

```
>> [H, G, n, k] = hammgen (m, p)
```

trong đó m là số nguyên ≥ 3 và $n = 2^m - 1$, $k = 2^m - m - 1$; p là một đa thức nguyên trong $GF(2)$.

Ví dụ:

```
>> [parmat, genmat] = hammgen(3)
```

```
parmat =
```

```

1      0      0      1      0      1      1
0      1      0      1      1      1      0
0      0      1      0      1      1      1
genmat =
1      1      0      1      0      0      0
0      1      1      0      1      0      0
1      1      1      0      0      1      0
1      0      1      0      0      0      1
>> genpoly = cyclpoly(7,3);
>> [parmat,genmat] = cyclgen(7,genpoly)
parmat =
1      0      0      0      1      1      0
0      1      0      0      0      1      1
0      0      1      0      1      1      1
0      0      0      1      1      0      1
genmat =
1      0      1      1      1      0      0
1      1      1      0      0      1      0
0      1      1      1      0      0      1

```

Ta cũng có thể chuyển đổi giữa ma trận sinh và ma trận kiểm tra bằng cách dùng hàm **gen2par**:

```

>> H = gen2par(G)
Ví dụ:
>> genmat = [1 0 0 1 0; 0 1 0 1 1; 0 0 1 0 1];
>> parmat = gen2par(genmat)
parmat =
1      1      0      1      0
0      1      1      0      1
>> genmat1=gen2par(parmat)
genmat1 =
1      0      0      1      0
0      1      0      1      1
0      0      1      0      1

```

○ *Bảng mã sửa sai (syndrome)*: bảng mã sửa sai được bộ giải mã dùng để sửa các lỗi có thể gặp phải trong quá trình truyền. Mỗi bảng mã là một ma trận n cột và 2^{n-k} hàng, mỗi hàng là một vector sai ứng với một từ mã nhận được. Với mã Hamming, số hàng của bảng mã là $n + 1$, cho phép sửa tất cả các lỗi đơn trong mọi từ mã.

Với một ma trận kiểm tra cho trước, có thể tạo ra bảng mã sửa sai tương ứng bằng cách dùng hàm **syndtable**.

```
>> t = syndtable(h)
```

 **Ví dụ 17-5.** Sử dụng bảng syndrome để kiểm tra và sửa lỗi khi giải mã thông điệp mã hoá bằng mã Hamming [7,4].

Phương pháp sửa lỗi như sau: giả sử vector biểu diễn thông điệp nhận được là r . Ta tính vector syndrome tương ứng bằng cách nhân vector r với chuyển vị của ma trận kiểm tra: $s = r \cdot H^T$. Đổi vector syndrome thành số thập phân. Nó sẽ cho biết vị trí của vector sai tương ứng trong bảng syndrome. Cuối cùng, lấy vector sai cộng với vector thu, đó chính là thông điệp đã được sửa lỗi.

```
% Sử dụng mã Hamming [7,4].
m = 3; n = 2^m-1; k = n-m;
parmat = hamngen(m); % Tạo ma trận kiểm tra.
trt = syndtable(parmat); % Tạo bảng giải mã syndrome.
recd = [1 0 0 1 1 1 1] % Giả sử đây là vector nhận được.
syndrome = rem(recd * parmat',2); % Tính syndrome
syndrome_de = bi2de(syndrome,'left-msb'); % Đổi sang số thập phân.
disp(['Syndrome = ',num2str(syndrome_de),...
' (decimal), ',num2str(syndrome),' (binary)'])
corrvect = trt(1+syndrome_de,:) % Vector sai
% Từ mã đúng:
correctedcode = rem(corrvect+recd,2)
```

Kết quả:

```
recd =
      1      0      0      1      1      1      1
Syndrome = 3 (decimal), 0 1 1 (binary)
corrvect =
      0      0      0      0      1      0      0
correctedcode =
      1      0      0      1      0      1      1
```

▪ **Mã hoá và giải mã với các loại mã khối tuyến tính:**

Hàm **encode** thực hiện mã hoá các loại mã khối tuyến tính, mã vòng và mã Hamming. Với mã khối tuyến tính, cần cung cấp ma trận sinh `genmat`; với mã Hamming có thể (không nhất thiết) cung cấp đa thức nguyên `p_poly` tương ứng; với mã vòng, cần cung cấp đa thức sinh `cyc_poly` cho bộ mã.

```
>> [code, added] = encode(msg, n, k, 'linear', genmat)
>> [code, added] = encode(msg, n, k, 'linear/decimal', genmat)
>> [code, added] = encode(msg, n, k, 'hamming', p_poly)
>> [code, added] = encode(msg, n, k, 'hamming/decimal', p_poly)
>> [code, added] = encode(msg, n, k, 'cyclic', cyc_poly)
>> [code, added] = encode(msg, n, k, 'cyclic/decimal', cyc_poly)
```

`added` là số cột phải thêm vào ma trận `msg` để `msg` có kích thước vừa đủ để tiến hành mã hoá.

Hàm **decode** thực hiện giải mã thông điệp mã hoá bằng các mã khối tuyến tính nói trên. Ngoài các thông số giống như đối với hàm **encode**, có thể đưa thêm bảng giải mã syndrome để sửa lỗi cho thông điệp. Nếu không có bảng giải mã, hàm **decode** không sửa lỗi nhưng có khả năng đánh dấu các vị trí lỗi phát hiện được. Ví dụ:

```
>> [msg, err, ccode, cerr] = decode(code, n, k, 'cyclic', cyc_poly,
dec_table)

>> [msg, err, ccode, cerr] = decode(code, n, k, 'linear', genmat,
dec_table)
```

trong đó, **err** là tổng số lỗi phát hiện được, **ccode** là từ mã đã được sửa sai, **cerr** là ma trận xác định các vị trí lỗi. Nếu số lỗi vượt quá khả năng sửa lỗi của bộ mã thì **err** sẽ có giá trị âm.

 **Ví dụ 17-6.** Sử dụng mã khối tuyến tính $[4,2]$ để mã hoá và giải mã một thông điệp cụ thể.

Trong thí dụ này, ta tạo một thông điệp gồm 3 từ **msg** = [0 1; 0 0; 1 0], mã hoá, cộng nhiễu ngẫu nhiên và giải mã. Kết quả dưới đây cho thấy thông điệp bị sai một lỗi ở từ thứ 3 và bộ giải mã đã sửa được lỗi này.

```
n = 4; k = 2;

genmat = [[1 1; 1 0], eye(2)]; % Ma trận sinh
msg = [0 1; 0 0; 1 0]; % Thông điệp gồm 3 từ, mỗi từ 2 bit
% Tạo ba từ mã, mỗi từ 4 bit.
code = encode(msg,n,k,'linear',genmat);
noisycode = rem(code + randerr(3,4,[0 1;.7 .3]),2); % Cộng nhiễu.
trt = syndtable(gen2par(genmat)); % Tạo bảng giải mã syndrome
% Giải mã, sửa sai và đánh dấu các vị trí bị sai.
[newmsg,err,ccode,cerr] = decode(noisycode,n,k,'linear',genmat,trt)
err_words = find(err~=0) % Xác định từ đã bị lỗi.
```

Kết quả:

```
newmsg =
    0    1
    0    0
    1    0

err =
    0
    0
    1

ccode =
    1    0    0    1
    0    0    0    0
    1    1    1    0

cerr =
    0
    0
```

```

1
err_words =
3

```

 **Ví dụ 17-7.** Sử dụng mã vòng $[4,2]$ để mã hoá và giải mã một thông điệp cụ thể. Giả sử bảng giải mã là ma trận 0.

```

n = 4; k = 2;
genpoly = [1 0 1]; % Đa thức sinh  $1 + x^2$ 
msg = [0 1; 0 0; 1 0];
code = encode(msg,n,k,'cyclic',genpoly);
noisycode = rem(code + randerr(3,4,[0 1;.7 .3]),2);
trt = zeros(2^(n-k),n); % Không sửa lỗi
[newmsg,err,ccode,cerr] = decode(noisycode,n,k,'cyclic',genpoly,trt)
err_words = find(err~=0) % Xác định từ đã bị lỗi.

```

Kết quả như sau:

```

newmsg =
    0    1
    0    0
    0    0
err =
    0
    0
   -1
ccode =
    0    1    0    1
    0    0    0    0
    0    0    1    0
cerr =
    0
    0
   -1
err_words =
    3

```

Trong trường hợp này, từ thứ 3 có 2 lỗi, vượt quá khả năng sửa lỗi của bộ mã.

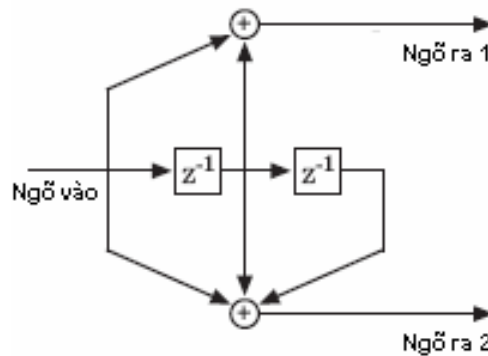
17.2. MÃ CHẬP

Mã chập là một phương pháp mã hoá sửa sai quan trọng. Khác với mã khối, mã chập là loại mã có nhớ. Mặc dù, bộ mã hoá chập cũng nhận các thông điệp có số lượng ký hiệu xác định và tạo ra một từ mã cũng có số ký hiệu xác định, nhưng từ mã tạo ra ở mỗi thời điểm không chỉ phụ thuộc vào các ký hiệu của thông điệp hiện tại mà còn phụ thuộc vào các ký hiệu của các thông điệp đã được mã hoá trước đó.

MATLAB cung cấp đầy đủ các công cụ để thực hiện mã chập kể cả thuận lẫn hồi tiếp. Mã chập có thể được mô tả dưới dạng cấu trúc lưới (trellis) hoặc tập các đa thức sinh. Giải thuật mà MATLAB sử dụng là giải thuật Viterbi, có thể giải mã bằng quyết định cứng (hard-decision) hay quyết định mềm (thích nghi, soft-decision).

17.2.1. DẠNG ĐA THỨC CỦA BỘ MÃ HOÁ CHẬP

Dạng đa thức của bộ mã hoá chập mô tả các kết nối giữa các thanh ghi dịch và các bộ cộng modulo 2. Ví dụ, hình vẽ sau mô tả một bộ mã hoá chập thuận có một ngõ vào, 2 ngõ ra và 2 thanh ghi dịch.



Hình 17.2. Sơ đồ mô tả bộ mã hoá dạng đa thức

Nói chung, mô tả bộ mã hoá chập dưới dạng đa thức có nghĩa là xác định các thành phần của nó, bao gồm: các chiều dài giới hạn, các đa thức sinh, và đa thức hồi tiếp (chỉ có ở bộ mã hoá chập hồi tiếp).

- Các chiều dài giới hạn: biểu diễn bằng một vector có chiều dài bằng số ngõ vào, mỗi phần tử của nó cho biết số bit được chứa trong mỗi thanh ghi dịch, kể cả các bit ngõ vào hiện tại. Chẳng hạn trong hình vẽ trên, chiều dài giới hạn bằng 3 (một vô hướng) vì chỉ có một ngõ vào và tổng số bit chứa trong thanh ghi dịch bằng $2 + 1 \text{ bit vào} = 3 \text{ bit}$.
- Các đa thức sinh: nếu sơ đồ mã hoá gồm có k ngõ vào và n ngõ ra thì ma trận sinh của bộ mã sẽ có kích thước $k \times n$. Phần tử nằm ở hàng thứ i và cột thứ j của ma trận cho biết tác dụng của ngõ vào thứ i đối với ngõ ra thứ j .

Với các bit hệ thống của bộ mã hoá hồi tiếp có tính hệ thống, các phần tử tương ứng của ma trận sinh được lấy từ các phần tử tương ứng của vector hồi tiếp.

Trong các trường hợp khác, phần tử ở hàng i cột j được xác định như sau:

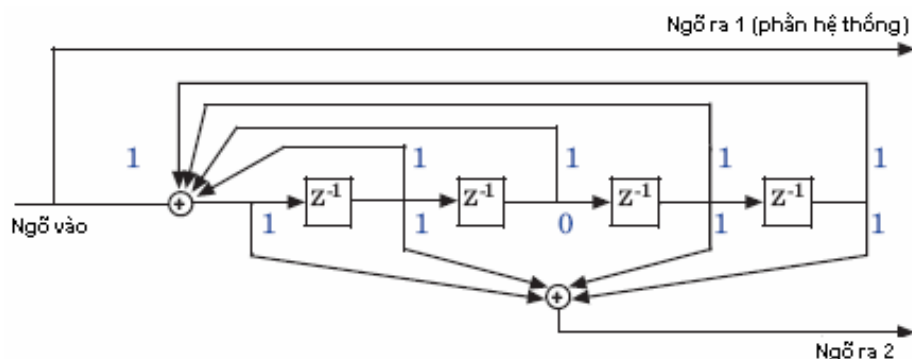
- Xây dựng một biểu diễn nhị phân bằng cách điền 1 vào mỗi điểm trên thanh ghi dịch mà từ đó có một đường kết nối đến bộ cộng, điền 0 vào những điểm còn lại trên thanh ghi dịch. Như vậy với mỗi bộ cộng có một số nhị phân với bit tận cùng bên phải là bit trọng số thấp nhất.
- Chuyển số nhị phân này thành số bát phân.

Ví dụ, trong sơ đồ trên, các số nhị phân tương ứng với các bộ cộng ở trên và dưới lần lượt là 110 và 111. Như vậy, ma trận sinh sẽ là $[6 \ 7]$.

- Đa thức hồi tiếp: khi biểu diễn bộ mã hoá hồi tiếp, cần phải xác định vector hồi tiếp. Chiều dài của vector này bằng với số ngõ vào. Các phần tử của vector này biểu diễn các kết nối hồi tiếp đối với mỗi ngõ vào, được xác định bằng cách xây dựng số nhị phân và chuyển sang hệ bát phân như cách xác định ma trận sinh được đề cập ở phần trên. Nếu bộ mã hoá có tính hệ

thông thì các phần tử của ma trận sinh và các phần tử của vector hồi tiếp ứng với các bit hệ thống phải có giá trị bằng nhau.

Ví dụ: xét sơ đồ mã hoá như sau:



Hình 17.3. Ví dụ về bộ mã hoá hồi tiếp

Bộ mã hoá có chiều dài giới hạn là 5, ma trận sinh là $[37 \ 33]$ và vector hồi tiếp là 37. Bởi vì ngõ ra thứ 1 là ngõ ra hệ thống nên phần tử tương ứng của ma trận sinh cũng bằng phần tử của vector hồi tiếp.

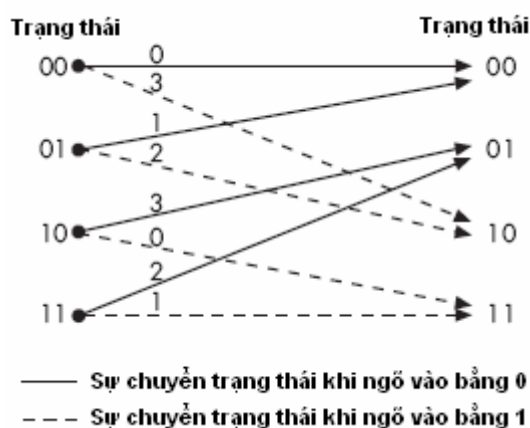
MATLAB hỗ trợ dạng mô tả đa thức bằng cách cung cấp hàm **poly2trellis** cho phép chuyển đổi dạng đa thức sang dạng cấu trúc trellis để có thể sử dụng các hàm mã hoá và giải mã.

```
>> trellis = poly2trellis(conslen, codegen, feedback)
```

trong đó *conslen*, *codegen*, *feedback* lần lượt là các chiều dài giới hạn, ma trận sinh và vector hồi tiếp. Tham số thứ ba chỉ xuất hiện trong trường hợp bộ mã hoá là hồi tiếp.

17.2.2. DẠNG CẤU TRÚC TRELLIS CỦA BỘ MÃ HÓA CHẬP

Dạng mô tả bằng cấu trúc lưới (trellis) của bộ mã hoá chập cho biết mỗi ngõ vào có ảnh hưởng như thế nào đến ngõ ra và sự chuyển trạng thái của bộ mã. Ví dụ, hình vẽ dưới đây là mô tả bằng cấu trúc trellis cho bộ mã được giới thiệu ở phần trước.



Hình 17.4. Mô tả bộ mã hoá bằng cấu trúc trellis

Bộ mã hoá có bốn trạng thái biểu diễn bằng 2 bit nhị phân (từ 00 đến 11), có một ngõ vào và hai ngõ ra. Mỗi mũi tên liền nét biểu thị sự chuyển trạng thái khi ngõ vào bằng 0, mỗi mũi tên đứt nét biểu thị sự chuyển trạng thái khi ngõ vào bằng 1. Trên mỗi mũi tên là một số bát phân biểu diễn ngõ ra hiện tại của bộ mã. Bộ mã này gọi là bộ mã tỷ lệ $\frac{1}{2}$ (1 ngõ vào, 2 ngõ ra).

- Trong MATLAB Communication Toolbox, dạng mô tả bằng cấu trúc trellis được biểu diễn bằng một cấu trúc gồm có 5 trường:

Bảng 17.4. Các trường của một cấu trúc trellis

Tên trường trong cấu trúc	Số chiều	Ý nghĩa
numInputSymbols	Vô hướng	Số ký hiệu ngõ vào của bộ mã hoá
numOutputSymbols	Vô hướng	Số ký hiệu ngõ ra của bộ mã hoá 2^n
numStates	Vô hướng	Số trạng thái trong bộ mã hoá
nextStates	Ma trận kích thước numStates x 2^k	Các trạng thái kế ứng với các tổ hợp của trạng thái hiện tại và ngõ vào hiện tại
outputs	Ma trận kích thước numStates x 2^k	Các ngõ ra ứng với các tổ hợp của trạng thái hiện tại và ngõ vào hiện tại

Tên của cấu trúc do người sử dụng đặt, nhưng các trường phải có các tên như trong bảng trên.

Trong ma trận nextStates, mỗi phần tử là một số nguyên từ 0 đến numStates – 1. Phần tử nằm ở hàng thứ i và cột thứ j chỉ thị trạng thái kế khi trạng thái hiện tại là i – 1 và các bits ngõ vào có biểu diễn thập phân tương ứng là j – 1 (bit ngõ vào đầu tiên là bit trọng số lớn nhất (MSB)).

Trong ma trận ngõ ra, phần tử ở hàng i và cột j là biểu diễn thập phân của các bit ngõ ra khi trạng thái hiện tại là i – 1 và các bits ngõ vào có biểu diễn thập phân là j – 1. Bit ngõ ra đầu tiên là bit có trọng số lớn nhất.

Để định nghĩa một cấu trúc trellis trong MATLAB, có thể dùng một trong các cách sau:

- Định nghĩa lần lượt từng trường trong số 5 trường của cấu trúc trellis, sử dụng cú pháp tencautruc.tentruong. Ví dụ, nếu s là tên cấu trúc, có thể viết:

```
>> s.numInputSymbols = 2;
```

- Gán giá trị cho cả 5 trường của cấu trúc trellis bằng cách dùng lệnh **struct**. Ví dụ:

```
s = struct('numInputSymbols',2,'numOutputSymbols',2,...
'numStates',2,'nextStates',[0 1;0 1],'outputs',[0 0;1 1]);
```

- Xuất phát từ mô tả dạng đa thức của bộ mã và sử dụng hàm **poly2trellis** để chuyển thành dạng mô tả bằng cấu trúc trellis (xem lại phần mô tả dạng đa thức của bộ mã hoá chập).

Người sử dụng có thể kiểm tra một cấu trúc nào đó có phải là cấu trúc trellis hay không bằng cách dùng hàm **istrellis**:

```
>> [isok, status] = istrellis(s)
```

Hàm này trả về một biến boolean isok, biến này bằng 1 nếu s là một cấu trúc trellis. Nếu s không phải cấu trúc trellis thì isok bằng 0, đồng thời hàm **istrellis** trả về chuỗi status chỉ ra lý do tại sao s không phải là cấu trúc trellis.

Để minh họa cách biểu diễn một cấu trúc trellis, ta trở lại với bộ mã hoá được giới thiệu ở đầu phần “Mã chập”. Dạng cấu trúc trellis của nó được thể hiện ở hình 17.4. Bộ mã hoá này được mô tả trong MATLAB bằng cấu trúc sau:

```
trellis = struct('numInputSymbols',2,'numOutputSymbols',4,...
'numStates',4,'nextStates',[0 2;0 2;1 3;1 3],...
```

```
'outputs',[0 3;1 2;3 0;2 1]);
```

Số ký hiệu ngõ vào của bộ mã bằng 2 vì bộ mã hoá có một ngõ vào nhận 1 trong 2 giá trị 0 hoặc 1. Tương tự, số ký hiệu ngõ ra là $2^2 = 4$ vì bộ mã có 2 ngõ ra. Số trạng thái của bộ mã bằng 4 vì thanh ghi dịch của bộ mã gồm có hai khối trữ. Để xác định các ma trận `nextStates` và `outputs`, ta dựa vào sơ đồ mô tả ở hình 13.4. Các ma trận này có bốn cột ứng với bốn trạng thái của bộ mã và hai cột ứng với hai ký hiệu ngõ vào là 0 hoặc 1. Các phần tử của ma trận `nextStates` là trạng thái mà mũi tên tương ứng chỉ đến, còn các phần tử của ma trận `outputs` là giá trị ghi trên mũi tên tương ứng.

17.2.3. MÃ HÓA VÀ GIẢI MÃ MÃ CHẬP

▪ Mã hoá mã chập được thực hiện bằng lệnh **convenc**. Hàm này trả về vector ngõ ra `code` ứng với thông điệp ngõ vào `msg`, với `trellis` là cấu trúc mô tả bộ mã hoá.

```
>> [code final_state] = convenc (msg,trellis,init_state)
```

`init_state` là một thông số phụ, xác định trạng thái khởi đầu của thanh ghi dịch, đó là một số nguyên trong khoảng từ 0 đến `trellis.numStates - 1`. Giá trị mặc định là 0.

Hàm này cũng trả về trạng thái sau cùng `final_state` của bộ mã hoá sau khi xử lý xong thông điệp.

▪ Giải mã mã chập: MATLAB sử dụng giải thuật Viterbi để giải mã mã chập. Có hai phương pháp giải mã: phương pháp quyết định cứng và phương pháp quyết định mềm.

○ Phương pháp quyết định cứng: thông điệp mã hoá ngõ vào `code` có dạng nhị phân và thông điệp giải mã ở ngõ ra `decoded` cũng là một vector nhị phân, không cần thực hiện thêm các xử lý khác.

```
>> decoded = vitdec(code,trellis,tblen,opmode,dectype)
```

`opmode` cho biết chế độ hoạt động của bộ giải mã. Có ba khả năng lựa chọn:

‘trunc’: bộ mã hoá được xem như khởi đầu từ trạng thái all-zero (toàn bit 0). Bộ giải mã sẽ thực hiện dò từ trạng thái ứng với độ đo tốt nhất.

‘term’: bộ mã hoá được xem như khởi đầu và kết thúc ở trạng thái all-zero. Bộ giải mã tiến hành dò từ trạng thái all-zero.

‘cont’: bộ mã hoá được xem như khởi đầu từ trạng thái all-zero. Bộ giải mã sẽ thực hiện dò từ trạng thái ứng với độ đo tốt nhất. Trường hợp này có xét tới độ trễ bằng `tblen` ký hiệu.

`dectype` cho biết phương pháp giải mã:

‘unquant’: bộ giải mã nhận vào tín hiệu thực, có dấu. Giá trị +1 biểu diễn mức logic 0 và giá trị -1 biểu diễn mức logic 1.

‘hard’: phương pháp quyết định cứng, dữ liệu vào có dạng nhị phân.

‘soft’: phương pháp quyết định mềm, sẽ được trình bày sau.

`tblen` là độ sâu dò tìm của bộ giải mã.

○ Phương pháp quyết định mềm: cần xác định số bit quyết định mềm `nsdec`, đồng thời dữ liệu ngõ vào cũng bao gồm các số nguyên từ 0 đến $2^{nsdec}-1$.

```
>> decoded = vitdec(code,trellis,tblen,opmode,'soft',nsdec).
```

Giá trị ngõ vào bằng 0 biểu thị mức 0 với độ tin cậy cao nhất, còn giá trị $2^{nsdec}-1$ biểu thị mức 1 với độ tin cậy cao nhất. Các giá trị khác biểu thị mức 0 hoặc mức 1 với độ tin cậy kém hơn. Ví dụ đối với trường hợp số bit quyết định mềm bằng 3, ta có bảng mô tả như sau:

Bảng 17.5. Ý nghĩa của các giá trị ngõ vào

Giá trị ngõ vào	Biểu thị
0	Mức 0 với độ tin cậy cao nhất (cấp 1)
1	Mức 0 với độ tin cậy cấp 2
2	Mức 0 với độ tin cậy cấp 3
3	Mức 0 với độ tin cậy thấp nhất
4	Mức 1 với độ tin cậy thấp nhất
5	Mức 1 với độ tin cậy cấp 3
6	Mức 1 với độ tin cậy cấp 2
7	Mức 1 với độ tin cậy cao nhất

o Ngoài các thông số cơ bản nêu trên, người sử dụng có thể cung cấp thêm các thông tin về các điều kiện đầu của bộ giải mã như các độ đo trạng thái đầu, các trạng thái khởi đầu và các ngõ vào ban đầu. Đồng thời, hàm **vitdec** cũng trả về các độ đo trạng thái cuối cùng, các trạng thái cuối và các giá trị ngõ vào sau cùng.

```
>> decoded = vitdec(..., 'cont', ..., init_metric, init_states, init_inputs)
>> [decoded final_metric final_states final_inputs] = vitdec(..., 'cont', ...)
```

Độc giả có thể tìm thêm các thông tin về các thông số này bằng cách gõ lệnh **help vitdec** ở cửa sổ lệnh của MATLAB.

 **Ví dụ 17-8.** Minh họa phương pháp giải mã mã chập bằng quyết định mềm.

Đoạn chương trình dưới đây minh họa phương pháp giải mã quyết định mềm với số bit quyết định mềm bằng 3. Đầu tiên ta sử dụng hàm **convenc** để tạo một thông điệp mã hoá bằng mã chập. Thông điệp này sau đó được cộng với nhiễu Gauss tạo bằng hàm **awgn** để mô phỏng thông điệp bị nhiễu. Trước khi giải mã, ta tiến hành lượng tử tín hiệu thu với số mức lượng tử bằng 8 để chuyển dữ liệu nhiễu thành các số nguyên từ 0 đến 7. Dữ liệu gần bằng 0 được gán giá trị 0, dữ liệu ở gần 1 được gán giá trị 7, các dữ liệu cách xa 0 và 1 hơn sẽ được gán các giá trị từ 2 đến 6 tùy theo việc chúng gần 0 và 1 như thế nào. Cuối cùng, tiến hành giải mã bằng hàm **vitdec** và tính tỷ lệ lỗi bit. Do sử dụng chế độ hoạt động liên tục ('cont') nên bộ giải mã sẽ tạo ra một khoảng thời gian trễ bằng với **tblen**, và **msg(1)** sẽ tương ứng với **decode(tblen+1)**.

```
msg = randint(4000,1,2,139); % Dữ liệu ngẫu nhiên
t = poly2trellis(7,[171 133]); % Định nghĩa cấu trúc trellis.
code = convenc(msg,t); % Mã hoá dữ liệu.
ncode = awgn(code,6,'measured',244); % Cộng nhiễu.
% Lượng tử hoá để chuẩn bị giải mã bằng phương pháp lấy ngưỡng mềm
qcode = quantiz(ncode,[0.001,.1,.3,.5,.7,.9,.999]);
tblen = 48; delay = tblen; % Độ sâu dò tìm
```

```

decoded = vitdec(qcode,t,tblen,'cont','soft',3); % Giải mã.
% Tính số lỗi và tỷ lệ bit lỗi.
[number,ratio] = biterr(decoded(delay+1:end),msg(1:end-delay))

```

Dưới đây là kết quả của một lần thực hiện chương trình:

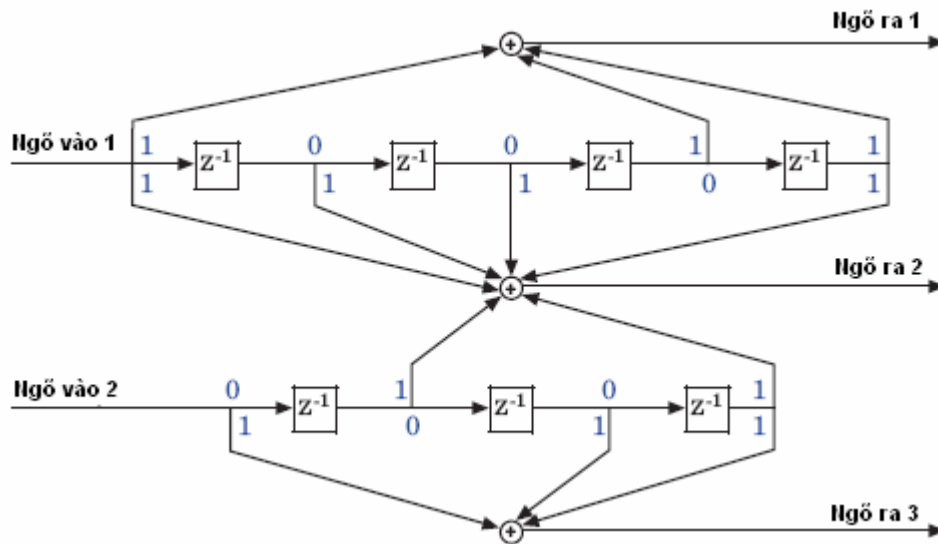
```

number =
    5
ratio =
    0.0013

```

Cuối cùng chúng ta kết thúc phần này với một ví dụ mang tính tổng hợp về kỹ thuật mã chập.

Ví dụ 17-9. Sử dụng bộ mã hoá mã chập thuận tỷ lệ 2/3 được mô tả trong hình vẽ 17.5 để mã hoá một chuỗi dữ liệu ngẫu nhiên. Tạo nhiễu Gauss và cộng vào dữ liệu. Sau đó thực hiện giải mã bằng hai phương pháp quyết định cứng (chỉ có hai mức lượng tử là 0 và 1) và quyết định mềm với số bit quyết định bằng 3. Tính tỷ lệ lỗi bit và so sánh kết quả.



Hình 17.5. Bộ mã hoá mã chập thuận tỷ lệ 2/3

Trước hết cần xác định các thông số cho bộ mã hoá này. Giới hạn chiều dài của bộ mã phải là một vector có chiều dài bằng 2 vì bộ mã hoá có hai ngõ vào. Thanh ghi dịch ứng với ngõ vào 1 có 4 ô nhớ (4 bộ delay), cộng với bit ngõ vào hiện tại, vậy chiều dài giới hạn ứng với ngõ vào 1 là 5. Tương tự, chiều dài giới hạn ứng với ngõ vào 2 là 4. Vậy vector chiều dài giới hạn là [5 4].

Ma trận sinh của bộ mã hoá là ma trận 2 x 3 (2 ngõ vào, 3 ngõ ra). Phần tử ở hàng i cột j cho biết ảnh hưởng của ngõ vào thứ i lên ngõ ra thứ j . Ví dụ, để xác định phần tử ở hàng thứ 2 và cột thứ 3, ta nhận xét thấy phần tử tận cùng bên trái và hai phần tử tận cùng bên phải của thanh ghi dịch thứ 2 được kết nối tới bộ cộng ở ngõ ra 3. Vậy, phần tử ở hàng thứ 2 cột 3 của ma trận sinh có biểu diễn nhị phân là 1011, tức là bằng 13 trong hệ bát phân. Thực hiện tương tự đến các vị trí khác, ta có ma trận sinh như sau: [23 35 0; 0 5 13].

Để thực hiện quá trình mã hoá và giải mã, dùng hàm **poly2trellis** để chuyển các thông số trên thành một cấu trúc trellis.

Dưới đây là đoạn chương trình cho ví dụ 17-9.

```

len = 1000;

```

```

msg = randint(2*len,1); % Thông điệp nhị phân ngẫu nhiên: 2 bit mỗi ký hiệu
trell = poly2trellis([5 4],[23 35 0;0 5 13]); % Định nghĩa cấu trúc trellis
code = convenc(msg,trell) % Mã hoá thông điệp.
ncode = awgn(code,7,'measured',244); % Cộng nhiễu.
% Giải mã bằng phương pháp lấy ngưỡng cứng
hcode = (1+sign(ncode-0.5))/2; % Mức ngưỡng bằng 0.5
hdecoded = vitdec(hcode,trell,34,'cont','hard'); % Giải mã.
disp('Phuong phap lay nguong cung')
[hnumber,hratio] = biterr(hdecoded(68+1:end),msg(1:end-68))
% Giải mã bằng phương pháp lấy ngưỡng mềm
% Lượng tử hoá trước khi giải mã.
qcode = quantiz(ncode,[0.001,.1,.3,.5,.7,.9,.999]);
sdecoded = vitdec(qcode,trell,34,'cont','soft',3); % Giải mã.
disp('Phuong phap lay nguong mem')
[snumber,sratio] = biterr(sdecoded(68+1:end),msg(1:end-68))

```

Kết quả thực hiện chương trình:

```

Phuong phap lay nguong cung
hnumber =
    168
hratio =
    0.0870
Phuong phap lay nguong mem
snumber =
    43
sratio =
    0.0223

```

✎ Bài tập 17-1.

Tạo một chuỗi dữ liệu ngẫu nhiên chiều dài $N = 1000$ bit. Mã hoá chuỗi dữ liệu bằng các phương pháp liệt kê ở dưới. Dùng hàm **randerr** để tạo lỗi ngẫu nhiên và tác động vào chuỗi dữ liệu trên. Giải mã chuỗi dữ liệu này và so với dữ liệu gốc. Tính số lỗi và tỷ lệ lỗi.

- Mã BCH [15,11]
- Mã vòng [15,11] với đa thức sinh $1 + X + X^4$
- Mã Hamming [15,11]
- Mã khối tuyến tính [15,11]
- Mã Reeds-Solomon [15,11]

✎ Bài tập 17-2.

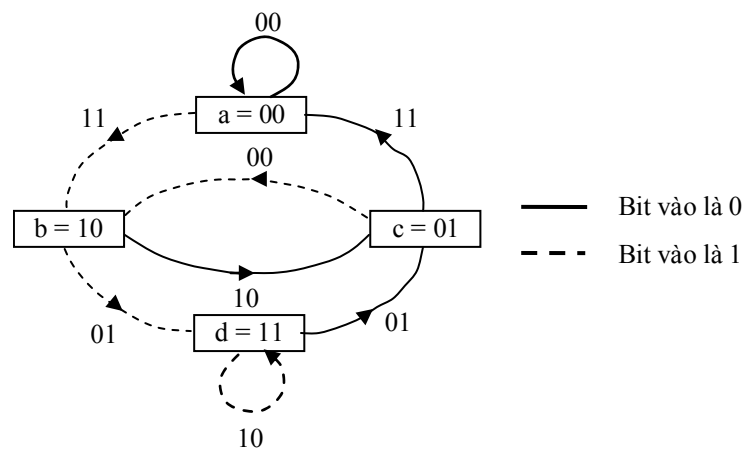
Khảo sát mã khối tuyến tính [7,4] với ma trận sinh sau đây:

$$G = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

- Tìm ma trận kiểm tra H bằng các hàm của MATLAB
- Dùng bộ mã hoá này để mã hoá thông điệp “Hello! Welcome to Ho Chi Minh City” (đã được mã hoá bằng mã ASCII).
- Giải mã mã khối tuyến tính, tính tỷ lệ bit lỗi. Khôi phục lại thông điệp ban đầu

✎ **Bài tập 17-3.**

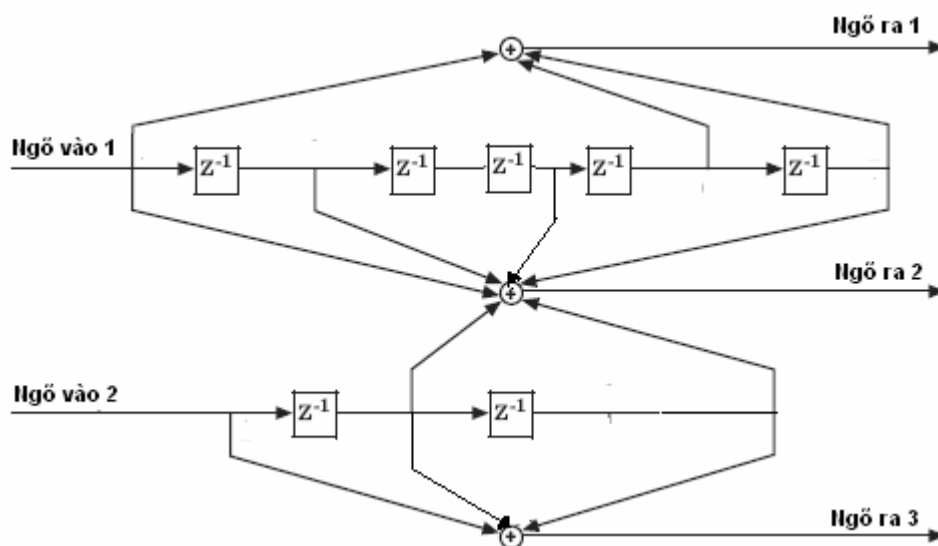
Xây dựng cấu trúc trellis trong MATLAB để mô tả bộ mã hoá mã chập có sơ đồ trạng thái như sau:



Hình 17.6.

✎ **Bài tập 17-4.**

Cho bộ mã hoá mã chập có sơ đồ khối như sau:



Hình 17.7.

Sử dụng bộ mã hoá này để mã hoá một thông điệp ngẫu nhiên chiều dài $N = 1000$. Thông điệp sau đó được truyền qua kênh truyền có nhiễu Gauss có $\text{SNR} = 5\text{dB}$. Giải mã thông điệp và so sánh với thông điệp gốc. Tính tỷ lệ bit lỗi. Dùng hai phương pháp:

- Quyết định mềm với 3 bit lượng tử
- Quyết định cứng

✎ Bài tập 17-5.

Thực hiện một hệ thống thông tin đơn giản: truyền một thông điệp ngẫu nhiên gồm 1000 ký hiệu từ máy phát đến máy thu bao gồm các bước sau:

- Mã hoá thông điệp bằng mã Huffman (từ mã nhị phân)
- Mã hoá sửa sai dùng mã Hamming [7,4]
- Điều chế bằng phương pháp BPSK
- Phát đi trên kênh truyền có nhiễu AWGN
- Giải điều chế BPSK
- Giải mã Hamming
- Giải mã Huffman và thu lại thông điệp ban đầu

Lần lượt khảo sát với các giá trị E_b/N_0 bằng 0, 2, 4, 6, 8 dB và so sánh thông điệp nhận được với thông điệp phát. Vẽ đồ thị BER.

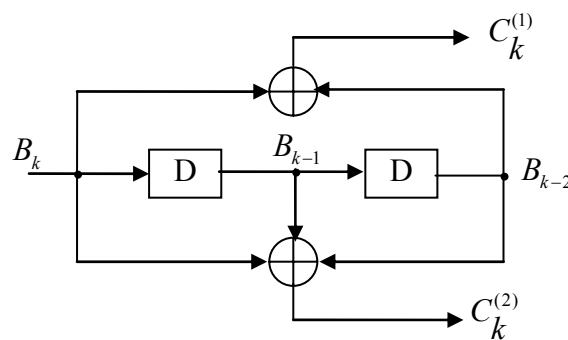
✎ Bài tập 17-6.

Làm lại bài tập trên với E_b/N_0 bằng 2dB nhưng lần lượt sử dụng các phương pháp mã hoá sau:

- Mã Reed-Solomon [7,3]
- Mã BCH [7,4]
- Mã khối tuyến tính [4,2]
- Mã vòng [4,2]
- Mã trellis với cấu trúc như trong ví dụ 17-9.

✎ Bài tập 17-7.

Khảo sát bộ mã hoá mã chập ở hình 17.8. Tạo một chuỗi bit, kích thước 1x10. Các chuỗi bit ngõ ra của bộ mã hoá là gì, nếu đa thức sinh là $G(X) = [1+X^2 \ 1+X+X^2]$.

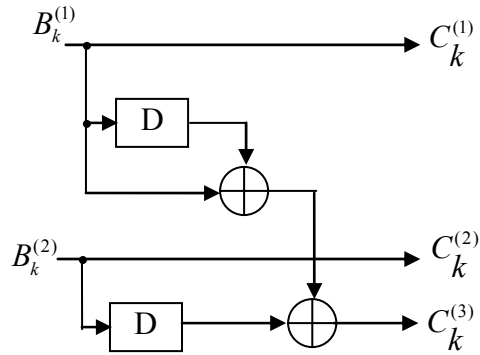


Hình 17.8.

Một bộ mã gọi là có tính hệ thống nếu các bit vào lại xuất hiện ở phần đầu của chuỗi bit đã mã hoá. Hỏi bộ mã trên có tính hệ thống không?

✎ **Bài tập 17-8.**

Khảo sát bộ mã hoá mã chập ở hình 17.9. Tạo hai chuỗi bit b_1 và b_2 , kích thước mỗi chuỗi là 1×10 . Các chuỗi bit ngõ ra của bộ mã hoá là gì, nếu đa thức sinh là $G(X) = [1 \ 0 \ 1+X; 0 \ 1 \ X]$. Bộ mã có tính hệ thống không?



Hình 17.9.

Danh sách các hàm được giới thiệu trong chương 17**Các hàm thực hiện các loại mã khối tuyến tính**

bchdec	Giải mã BCH
bchenc	Mã hoá BCH
bchgenpoly	Tạo đa thức sinh cho BCH
cyclgen	Tạo ma trận kiểm tra cho mã vòng
cyclpoly	Tạo đa thức sinh cho mã vòng
decode	Giải mã mã khối tuyến tính tổng quát
encode	Mã hoá mã khối tuyến tính tổng quát
gen2par	Chuyển đổi giữa ma trận kiểm tra và ma trận sinh
gf	Tạo một biến dãy Galois
hammgen	Tạo ma trận kiểm tra và ma trận sinh cho mã Hamming
rsdec	Giải mã Reed-Solomon
rsdecof	Giải mã file ASCII dùng mã hoá Reed-Solomon
rsenc	Giải mã Reed-Solomon
rsencof	Mã hoá file ASCII dùng mã hoá Reed-Solomon
rsgenpoly	Tạo đa thức sinh cho mã Reed-Solomon

Các hàm thực hiện các loại mã chập

convenc	Thực hiện mã hoá mã chập
istrellis	Kiểm tra xem một cấu trúc nào đó có phải là cấu trúc trellis không
poly2trellis	Chuyển mã chập từ dạng mô tả bằng đa thức sang dạng cấu trúc trellis
struct	Mô tả các thuộc tính của cấu trúc trellis
vitdec	Giải mã mã chập bằng giải thuật Viterbi

Chương 18

CÁC BỘ CÂN BẰNG

Nhiều giao thoa liên ký tự là một loại nhiễu phổ biến trong các hệ thống viễn thông. Nhiễu này xuất hiện ở các kênh truyền phân tán theo thời gian. Chẳng hạn trong một môi trường tán xạ đa đường, một ký hiệu có thể được truyền theo các đường khác nhau, đến máy thu ở các thời điểm khác nhau, do đó có thể giao thoa với các ký hiệu khác. Để khắc phục hiện tượng nhiễu ISI và cải thiện chất lượng hệ thống, có nhiều phương pháp khác nhau nhưng phương pháp được đề cập nhiều nhất là sử dụng bộ cân bằng để bù lại đặc tính tán xạ thời gian của kênh truyền.

Các bộ cân bằng có thể được phân loại thành ba lớp:

- Các bộ cân bằng tuyến tính
- Các bộ cân bằng hồi tiếp quyết định
- Các bộ cân bằng MLSE (Maximum – Likelihood Sequence Estimation)

MATLAB Communication toolbox cung cấp các hàm hỗ trợ cả ba loại bộ cân bằng này. Các bộ cân bằng MLSE sử dụng giải thuật Viterbi, còn các bộ cân bằng tuyến tính và cân bằng hồi tiếp quyết định là các bộ cân bằng thích nghi. Cơ sở hoạt động của nó dựa trên các giải thuật thích nghi, bao gồm:

- Giải thuật bình phương trung bình cực tiểu (LMS – Least Mean Square)
- Giải thuật LMS có dấu (Signed LMS)
- Giải thuật LMS chuẩn hoá (Normalized LMS)
- Giải thuật LMS có kích thước bước nhảy thay đổi (Variable – step – size LMS)
- Giải thuật bình phương cực tiểu đệ quy (RLS – Recursive Least Square)
- Giải thuật module không đổi (CMA – Constant Modulus Algorithm)

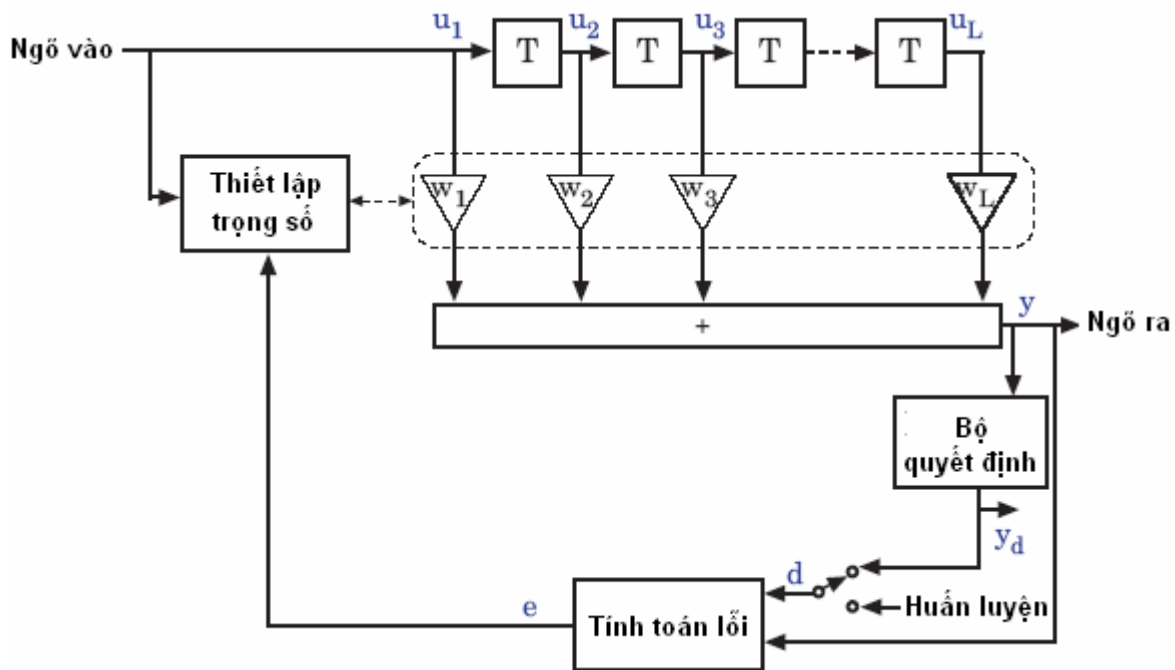
18.1. CÁC BỘ CÂN BẰNG THÍCH NGHI

Nói chung, các bộ cân bằng thích nghi bao gồm ba loại chính như sau: bộ cân bằng với khoảng cách một ký hiệu, bộ cân bằng khoảng cách tỷ lệ và bộ cân bằng hồi tiếp quyết định.

18.1.1. BỘ CÂN BẰNG KHOẢNG CÁCH KÝ HIỆU

Bộ cân bằng khoảng cách ký hiệu là bộ cân bằng tuyến tính cấu tạo bởi một dãy các khối tạo trễ để lưu trữ các mẫu tín hiệu vào. Sau mỗi một chu kỳ ký hiệu, bộ cân bằng sẽ tính tổng (có trọng số) của các mẫu dữ liệu lưu trong các khối trễ và xuất kết quả ra ngõ ra, đồng thời cập nhật các trọng số theo một trong các giải thuật thích nghi đã đề cập ở phần đầu của chương này để chuẩn bị cho chu kỳ ký hiệu kế tiếp. Bộ cân bằng này được gọi là “định khoảng theo ký hiệu” vì tốc độ của các mẫu ở ngõ ra và ngõ vào bằng nhau và dựa trên cơ sở một chu kỳ ký hiệu.

Hình vẽ 19.1 minh họa sơ đồ nguyên lý của một bộ cân bằng định khoảng theo ký hiệu gồm có L trọng số và chu kỳ ký hiệu là T.



Hình 18.1. Bộ cân bằng định khoảng theo ký hiệu

- Các trọng số của bộ cân bằng được cập nhật bằng các giải thuật thích nghi. Giá trị của bộ trọng số cập nhật sẽ phụ thuộc vào giá trị của bộ trọng số hiện tại, giá trị ngõ vào, giá trị ngõ ra. Ngoài ra, ngoại trừ các bộ cân bằng dùng giải thuật CMA, giá trị của bộ trọng số mới còn phụ thuộc vào một tín hiệu tham chiếu có tính chất phụ thuộc vào chế độ hoạt động của bộ cân bằng.
- Mỗi liên hệ giữa tín hiệu tham chiếu và chế độ hoạt động của bộ cân bằng được mô tả trong bảng 18.1.

Bảng 18.1. Tín hiệu tham chiếu và các chế độ hoạt động của bộ cân bằng

Chế độ hoạt động	Tín hiệu tham chiếu
Chế độ huấn luyện	Chuỗi phát đã xác định trước
Chế độ quyết định	Tín hiệu ra khỏi bộ quyết định, ký hiệu y_d trong hình 18.1

Trong các ứng dụng thông thường, bộ cân bằng sẽ khởi đầu ở chế độ huấn luyện để thu thập các thông tin về tính chất của kênh truyền, sau một thời gian mới chuyển sang chế độ quyết định.

- Quá trình tính toán lỗi sẽ tạo ra một tín hiệu sai số e xác định bằng biểu thức sau:

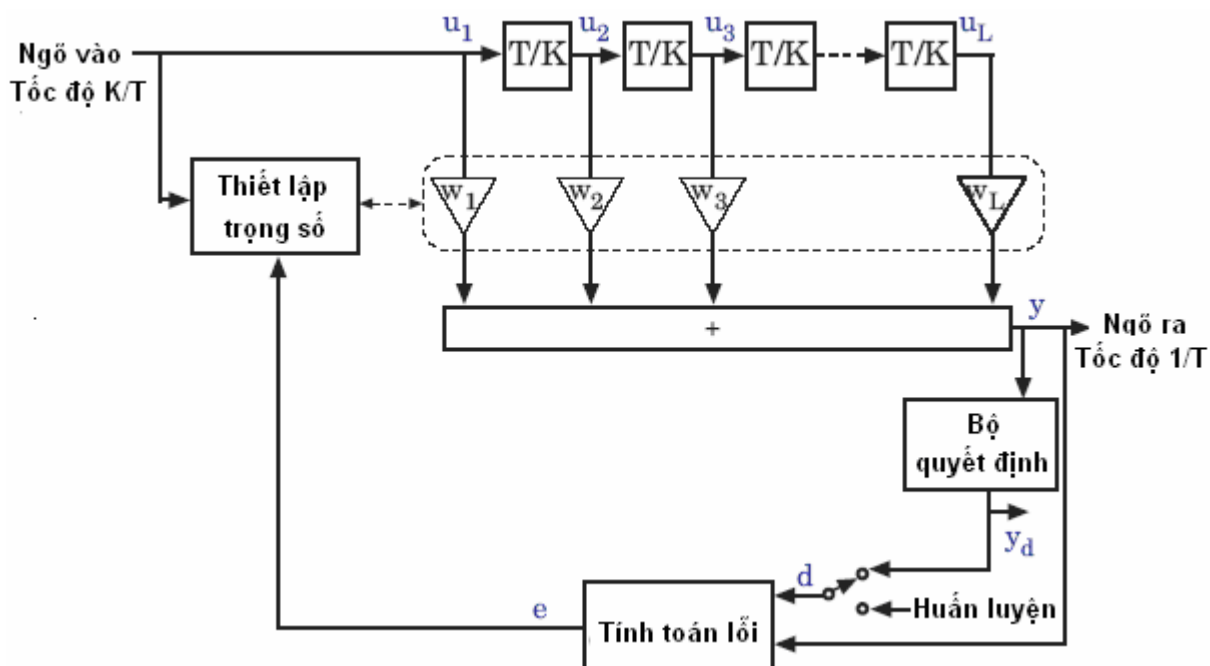
$$e = \begin{cases} d - y & \text{với các giải thuật khác CMA} \\ y(R - |y|^2) & \text{với giải thuật CMA} \end{cases} \quad (18.1)$$

trong đó R là một hằng số chỉ phụ thuộc vào dạng tín hiệu.

18.1.2. BỘ CÂN BẰNG ĐỊNH KHOẢNG TỶ LỆ

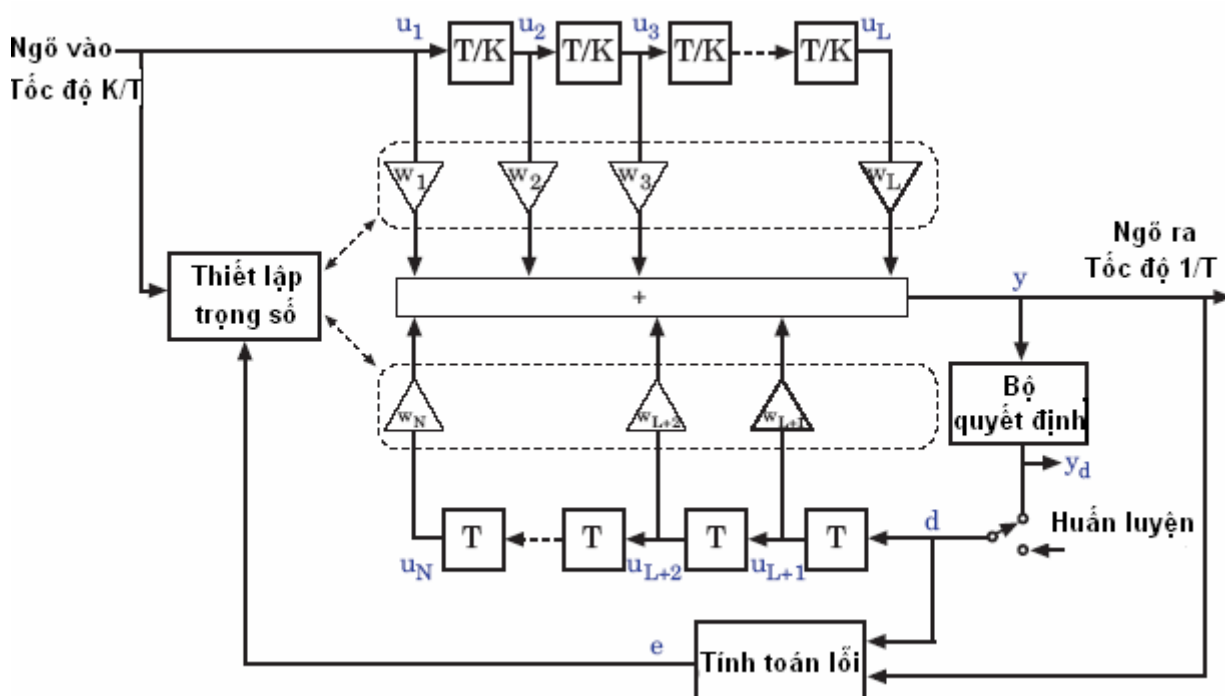
Bộ cân bằng định khoảng tỷ lệ cũng là bộ cân bằng tuyến tính với cấu tạo tương tự như bộ cân bằng định khoảng theo ký hiệu đã được giới thiệu ở trên. Tuy nhiên, bộ cân bằng định khoảng

tỷ lệ chỉ xuất dữ liệu ra và cập nhật trọng số sau khi đã nhận được K mẫu dữ liệu vào với K là một số nguyên (thường lớn hơn 1). Như vậy tốc độ dữ liệu ra và tốc độ cập nhật trọng số sẽ chậm hơn K lần so với tốc độ dữ liệu vào. Thường chọn $K = 2$. Sơ đồ nguyên lý của bộ cân bằng định khoảng tỷ lệ được minh họa ở hình vẽ 18.2.



Hình 18.2. Bộ cân bằng định khoảng tỷ lệ

18.1.3. BỘ CÂN BẰNG HỒI TIẾP QUYẾT ĐỊNH



Hình 18.3. Bộ cân bằng hồi tiếp quyết định - định khoảng tỷ lệ

Bộ cân bằng hồi tiếp quyết định là bộ cân bằng phi tuyến, cấu tạo bởi một bộ lọc thuận và một bộ lọc hồi tiếp. Bộ lọc thuận có cấu tạo tương tự như bộ cân bằng tuyến tính, còn bộ lọc hồi tiếp cấu tạo bởi một dãy các khối trễ mà ngõ vào của nó là các quyết định thực hiện trên tín

hiệu ngõ ra bộ cân bằng. Mục đích của bộ cân bằng hồi tiếp quyết định (DFE) là vừa triệt nhiễu giao thoa liên ký tự ISI (mục đích chính của các bộ cân bằng) đồng thời tối thiểu hoá sự tăng cường nhiễu của bộ cân bằng. Đối với các bộ cân bằng tuyến tính thì vấn đề tăng cường nhiễu là một nhược điểm lớn.

Sơ đồ hình 19.3 biểu diễn một bộ cân bằng DFE định khoảng tỷ lệ có L trọng số ở nhánh thuận và $N - L$ là số trọng số của bộ lọc hồi tiếp. Bộ lọc thuận nằm ở phía trên, còn phía dưới là bộ lọc hồi tiếp. Nếu $K = 1$ thì bộ cân bằng này trở thành bộ cân bằng DFE định khoảng theo ký hiệu.

Sau một chu kỳ ký hiệu (của ngõ ra), bộ cân bằng sẽ nhận vào K mẫu dữ liệu vào ở bộ lọc thuận, và một mẫu quyết định (hoặc một mẫu dữ liệu huấn luyện) ở bộ lọc hồi tiếp. Bộ cân bằng sẽ tính tổng có trọng số của tất cả các giá trị lưu trong bộ lọc thuận và bộ lọc hồi tiếp, đồng thời cập nhật các trọng số để chuẩn bị cho chu kỳ ký hiệu kế tiếp. Giải thuật cập nhật trọng số phải tối ưu hoá đồng thời các trọng số thuận và các trọng số hồi tiếp. Điều này đặc biệt quan trọng đối với giải thuật bình phương cực tiểu hồi quy RLS.

18.2. CÁC GIẢI THUẬT CÂN BẰNG THÍCH NGHI

Khâu cập nhật trọng số là khâu quan trọng nhất của bộ cân bằng. Nó quyết định hiệu quả của bộ cân bằng. Các bộ cân bằng thích nghi sử dụng các giải thuật cập nhật trọng số thích nghi, nghĩa là trọng số mới sẽ phụ thuộc vào trọng số cũ và các giá trị ngõ vào, ngõ ra và tín hiệu sai số giữa ngõ vào và ngõ ra. Sau đây là các giải thuật thích nghi được sử dụng trong MATLAB.

18.2.1. GIẢI THUẬT BÌNH PHƯƠNG TRUNG BÌNH CỰC TIỂU (LMS – LEAST MEAN SQUARE)

Giải thuật LMS tìm cách cập nhật trọng số sao cho giá trị trung bình của bình phương sai số giữa giá trị ngõ ra của bộ cân bằng với giá trị ngõ ra mà ta mong muốn đạt được. Giả sử I_k ($k = 1, 2, \dots, N$) là các giá trị ngõ ra mong muốn, $\hat{I}_k$ ($k = 1, 2, \dots, N$) là các giá trị ngõ ra thực tế, thì mục đích của giải thuật LMS là tối thiểu hoá đại lượng:

$$J = \frac{1}{N} \sum_{k=1}^N (I_k - \hat{I}_k)^2 \quad (18.2)$$

Người ta tìm được quy tắc cập nhật trọng số của giải thuật này như sau:

$$W(k+1) = \lambda W(k) + \mu U^* e \quad (18.3)$$

trong đó:

$W(k+1)$ và $W(k)$ là vector các trọng số của bộ cân bằng ở bước thứ $k + 1$ và bước thứ k :

$$W = [w_1, w_2, \dots, w_L] \quad (18.4)$$

U là vector dữ liệu ngõ vào (dấu $*$ biểu thị liên hiệp phức):

$$U = [u_1, u_2, \dots, u_L] \quad (18.5)$$

e là tín hiệu sai số.

λ là một hằng số thực trong khoảng $[0, 1]$ gọi là hệ số rò. Đối với giải thuật cập nhật trọng số thông thường thì $\lambda = 1$, còn giải thuật cập nhật trọng số không có nhớ thì $\lambda = 0$.

μ là một hằng số gọi là kích cỡ bước (step size).

18.2.2. GIẢI THUẬT LMS CÓ DẤU (SIGN LMS)

Đây là một dạng biến thể của giải thuật LMS, bao gồm ba loại với quy tắc cập nhật trọng số tương ứng cho mỗi loại được xác định bởi các phương trình dưới đây:

- Sign LMS: $W(k+1) = \lambda W(k) + \mu U^* \text{sign}(\text{Re}(e))$ (18.6)

- Signed regressor LMS: $W(k+1) = \lambda W(k) + \mu \text{sign}(\text{Re}(U)) \cdot \text{Re}(e)$ (18.7)

- Sign-sign LMS: $W(k+1) = \lambda W(k) + \mu \cdot \text{sign}(\text{Re}(U)) \cdot \text{sign}(\text{Re}(e))$ (18.8)

18.2.3. GIẢI THUẬT LMS CHUẨN HÓA (NORMALIZED LMS)

Giải thuật này sử dụng quy tắc cập nhật trọng số sau đây:

$$W(k+1) = \lambda W(k) + \mu \frac{U^* e}{U^H U + b} \quad (18.9)$$

trong đó b là một tham số gọi là tham số phân cực của giải thuật, b có giá trị nằm giữa 0 và 1. Tham số này được đưa vào để khắc phục nhược điểm của giải thuật LMS khi tín hiệu vào có giá trị nhỏ. Ký hiệu U^H biểu diễn ma trận chuyển vị Hermit của ma trận U .

18.2.4. GIẢI THUẬT LMS CÓ BƯỚC NHẢY THAY ĐỔI (VARIABLE-STEP-SIZE LMS)

Giải thuật này sử dụng các giá trị kích cỡ bước khác nhau trong các lần cập nhật trọng số. Cho trước giá số $\Delta\mu$ và các giới hạn $\mu_{\min}$ và $\mu_{\max}$, thì giá trị của μ sẽ được cập nhật theo quy luật như sau:

- Đầu tiên, tính giá trị $\mu_0 = \mu + \Delta\mu \cdot \text{Re}(G \cdot G_{\text{prev}})$, trong đó μ là giá trị hiện tại của kích cỡ bước, $G = U \cdot e^*$ và G_{prev} là giá trị trước đó của G .
- Nếu $\mu_0 > \mu_{\max}$ thì giá trị mới của μ sẽ là $\mu_{\max}$, nếu $\mu_0 < \mu_{\min}$ thì giá trị mới của μ sẽ là $\mu_{\min}$. Trong các trường hợp còn lại, giá trị mới sẽ là μ_0 .

Tập trọng số mới được xác định bằng công thức:

$$W(k+1) = \lambda W(k) + 2\mu G^* \quad (18.10)$$

18.2.5. GIẢI THUẬT BÌNH PHƯƠNG CỰC TIỂU HỒI QUY (RLS – RECURSIVE LEAST SQUARE)

Giải thuật này được thực hiện trên cơ sở một ma trận P gọi là ma trận tương quan nghịch đảo, và một hằng số thực nằm trong khoảng $[0, 1]$ gọi là hệ số “quên” (forget factor), ký hiệu f . Ở trạng thái khởi đầu, $P = c_0 \cdot I_N$ với c_0 là một hằng số nào đó còn I_N là ma trận đơn vị cấp N (N là số trọng số của bộ cân bằng).

Ở mỗi bước cập nhật trọng số, đầu tiên giải thuật này sẽ tiến hành xác định một vector K gọi là vector độ lợi Kalman. Nếu gọi P là ma trận tương quan nghịch đảo hiện tại, và u là vector ngõ vào hiện tại, thì:

$$K = \frac{P \cdot U}{f + U^H \cdot P \cdot U} \quad (18.11)$$

Sau đó, ma trận P sẽ được cập nhật theo quy luật dưới đây:

$$P_{\text{new}} = f^{-1} (P - K \cdot U^H \cdot P) \quad (18.12)$$

P_{new} là ma trận tương quan nghịch đảo mới.

Cuối cùng, tập trọng số mới sẽ được xác định bởi:

$$W(k+1) = W(k) + K^* e \quad (18.13)$$

18.2.6. GIẢI THUẬT MODULUS HẲNG SỐ (CONSTANT MODULUS ALGORITHM)

Giải thuật này sử dụng cùng một quy tắc cập nhật trọng số như giải thuật LMS: $W(k+1) = \lambda W(k) + \mu U^* e$, chỉ khác ở chỗ tín hiệu sai số được xác định theo phương trình:

$$e = y(R - |y|^2) \quad (18.14)$$

trong đó R là một hằng số chỉ phụ thuộc vào dạng tín hiệu.

Giải thuật này phụ thuộc rất lớn vào giá trị khởi đầu của tập trọng số. Nên chọn giá trị khởi đầu là một vector khác 0. Thông thường, người ta chọn, trọng số trung tâm bằng 1, còn các trọng số còn lại bằng 0.

18.3. SỬ DỤNG CÁC BỘ CÂN BẰNG THÍCH NGHI TRONG MATLAB

▪ Về cơ bản, để sử dụng một bộ cân bằng thích nghi trong MATLAB, cần tiến hành theo các bước sau:

- *Bước 1:* Khởi tạo một đối tượng mô tả bộ cân bằng, đối tượng này sẽ xác định loại bộ cân bằng và giải thuật thích nghi nào được sử dụng cho bộ cân bằng đó. Đối tượng này là một biến trong MATLAB chứa các thông tin liên quan đến bộ cân bằng, ví dụ như loại, giải thuật thích nghi, các giá trị trọng số, ...
- *Bước 2:* Thay đổi các đặc tính của đối tượng mô tả bộ cân bằng tùy theo nhu cầu sử dụng. Chẳng hạn, ta có thể thay đổi số trọng số của bộ cân bằng hoặc thay đổi giá trị các trọng số, ...
- *Bước 3:* Sử dụng đối tượng cân bằng này để tác động vào tín hiệu mà ta cần cân bằng. MATLAB cung cấp hàm **equalize** để thực hiện tác vụ này.

Trong các phần sau đây, chúng ta sẽ lần lượt tìm hiểu các quá trình xây dựng bộ cân bằng, chọn giải thuật cân bằng và sử dụng bộ cân bằng khi lập trình với MATLAB.

18.3.1. XÁC ĐỊNH GIẢI THUẬT THÍCH NGHI

Việc lựa chọn giải thuật thích nghi nào là phụ thuộc vào từng tình huống cụ thể, sau đây là một số quy tắc chung khi lựa chọn giải thuật:

- Giải thuật LMS thực thi nhanh nhưng tốc độ hội tụ chậm và độ phức tạp tính toán tăng tuyến tính theo số lượng trọng số.
- Giải thuật RLS có tốc độ hội tụ nhanh nhưng độ phức tạp tính toán tỷ lệ với bình phương của số trọng số. Giải thuật này có thể trở nên bất ổn khi số trọng số lớn.
- Các biến thể của giải thuật LMS có dấu sẽ làm đơn giản hóa vấn đề thực hiện phần cứng.
- Giải thuật LMS chuẩn hoá và giải thuật LMS có kích cỡ bước thay đổi rất hiệu quả đối với các biến đổi của các đại lượng thống kê của tín hiệu vào (thí dụ công suất).
- Giải thuật modulus hằng số hữu dụng trong các trường hợp ta không thể có chuỗi dữ liệu huấn luyện, và có hiệu quả tốt nhất đối với các phương pháp điều chế module không đổi (ví dụ như PSK). Tuy nhiên nếu không được cung cấp một số thông tin phụ, giải thuật này có thể dẫn đến sự sai pha, ví dụ giải thuật CMA có thể tính toán các trọng số theo đúng mô hình

QPSK nhưng các trạng thái pha có thể là 90° , 180° , 270° thay vì 45° , 135° , 225° , 315° . Để khắc phục nhược điểm này, có thể dùng các phương pháp điều chế vi sai.

Sau khi chọn xong giải thuật thích hợp, ta sẽ khởi tạo một đối tượng để mô tả bộ cân bằng với giải thuật đã chọn. Bảng dưới đây là mô tả các hàm MATLAB để khởi tạo các đối tượng mô tả giải thuật thích nghi cho bộ cân bằng.

Bảng 18.2. Các hàm khởi tạo đối tượng mô tả bộ cân bằng

Hàm khởi tạo	Giải thuật thích nghi	Các đặc tính của đối tượng được khởi tạo
<code>lms(stepsize, leakagefactor)</code>	LMS	AlgType: 'LMS' StepSize: Kích cỡ bước (stepsize) LeakageFactor: Hệ số rò (leakagefactor) (mặc định 1)
<code>signlms(stepsize, algtype)</code>	LMS có dấu	AlgType: 'Sign LMS', 'Signed Regressor LMS' hoặc 'Sign Sign LMS' StepSize: Kích cỡ bước (stepsize) LeakageFactor: Hệ số rò (leakagefactor) (mặc định 1)
<code>normlms(stepsize, bias)</code>	LMS chuẩn hoá	AlgType: 'LMS' StepSize: Kích cỡ bước (stepsize) LeakageFactor: Hệ số rò (leakagefactor) (mặc định 1) Bias: Thông số phân cực (bias) (mặc định 0)
<code>varlms(initstep, incstep, minstep, maxstep)</code>	LMS có kích cỡ bước thay đổi	AlgType: 'Variable Step Size LMS' LeakageFactor: Hệ số rò (mặc định 1) InitStep: trị khởi đầu của kích cỡ bước IncStep: gia số của kích cỡ bước MinStep: trị cực tiểu của kích cỡ bước MaxStep: trị cực đại của kích cỡ bước
<code>rls(forgetfactor, invcorrinit)</code>	RLS	AlgType: 'RLS' ForgetFactor: Hệ số “quên” (forgetfactor) InvCorrInit: Trị khởi đầu của ma trận tương quan nghịch đảo
<code>cma(stepsize, leakagefactor)</code>	CMA	AlgType: 'Constant Modulus' StepSize: Kích cỡ bước (stepsize) LeakageFactor: Hệ số rò (leakagefactor) (mặc định 1)

Để truy xuất một đặc tính của đối tượng mô tả giải thuật thích nghi có tên ALG (ví dụ hiệu chỉnh giá trị của đặc tính), ta dùng cú pháp `ALG.Prop` trong đó 'Prop' là tên của đặc tính mà ta muốn truy xuất. Ví dụ: `ALG.LeakageFactor = 1;`

18.3.2. XÂY DỰNG ĐỐI TƯỢNG MÔ TẢ BỘ CÂN BẰNG THÍCH NGHI

Trong MATLAB, các đối tượng mô tả bộ cân bằng thích nghi được khởi tạo bằng một trong hai hàm **lineareq** (bộ cân bằng thích nghi) và **dfe** (bộ cân bằng hồi tiếp quyết định).

```
>> eqobj = lineareq(nweights, alg, sigconst)
```

Hàm này khởi tạo một đối tượng mô tả bộ cân bằng định khoảng theo ký hiệu có tên là eqobj, số trọng số là nweights và giải thuật thích nghi được mô tả bởi đối tượng alg. Sigconst là một vector mô tả dạng tín hiệu. Giá trị mặc định là [-1 1], tương ứng với tín hiệu BPSK.

```
>> eqobj = lineareq(nweights, alg, sigconst, nsamp)
```

Hàm này khởi tạo một đối tượng mô tả bộ cân bằng định khoảng tỷ lệ, có các trọng số được định khoảng bằng $T/nsamp$, trong đó nsamp là một số nguyên dương còn T là chu kỳ ký hiệu. Nếu nsamp = 1 ta có bộ cân bằng định khoảng theo ký hiệu.

Ví dụ: khởi tạo một đối tượng mô tả bộ cân bằng định khoảng ký hiệu có 10 trọng số cập nhật bằng giải thuật RLS với kích cỡ bước bằng 3, tốc độ dữ liệu ra bằng $\frac{1}{2}$ tốc độ dữ liệu vào:

```
>> eqfrac = lineareq(10, rls(0.3), [-1 1], 2);
```

Sau đây là các thuộc tính của đối tượng tạo bởi hàm **lineareq**:

EqType: 'Linear Equalizer'

nWeights: Số trọng số

nSampPerSym: Số mẫu dữ liệu vào trên một ký tự (bằng nsamp)

RefTap: Chỉ số tham chiếu tới các tap của đường dây trễ (nằm giữa 1 và nWeights)

SigConst: Vector mô tả dạng tín hiệu

Weights: Vector trọng số phức

WeightInputs: Vector các dữ liệu vào các khối trọng số

ResetBeforeFiltering: Reset trạng thái bộ cân bằng sau mỗi lần gọi (nhận giá trị 0 hoặc 1)

NumSamplesProcessed: Số mẫu đã được xử lý

Ngoài ra các đặc tính của đối tượng mô tả giải thuật thích nghi cũng là các đặc tính của đối tượng mô tả bộ cân bằng.

```
>> eqobj = dfe(nfwdweights, nfbkweights, alg, sigconst, nsamp)
```

Hàm này khởi tạo một đối tượng mô tả bộ cân bằng hồi tiếp định khoảng theo ký hiệu có tên là eqobj, số trọng số của bộ lọc thuận là nfwdweights, của bộ lọc hồi tiếp là nfbkweights và giải thuật thích nghi được mô tả bởi đối tượng alg. Sigconst là một vector mô tả dạng tín hiệu. Giá trị mặc định là [-1 1], tương ứng với tín hiệu BPSK. Các trọng số bộ lọc thuận được định khoảng bằng $T/nsamp$ với T là chu kỳ ký hiệu. Nếu nsamp = 1 (hoặc không được đưa vào trong danh sách tham số của hàm) thì bộ cân bằng được tạo ra là bộ cân bằng DFE định khoảng theo ký hiệu.

Các đặc tính của đối tượng mô tả bộ cân bằng DFE nói chung cũng giống như của bộ cân bằng tuyến tính, chỉ khác ở hai đặc tính:

EqType: 'Decision Feedback Equalizer'

`nWeights`: Số các trọng số của bộ lọc thuận và bộ lọc hồi tiếp, được biểu diễn bởi một vector hàng gồm 2 phần tử [`n fwdweights nfbkweights`].

Ví dụ: khởi tạo đối tượng mô tả bộ cân bằng DFE định khoảng theo ký hiệu, bộ lọc thuận có 3 tap, bộ lọc hồi tiếp có 2 tap, cập nhật trọng số bằng giải thuật RLS với kích cỡ bước bằng 0.3:

```
>> eqdfe = dfe(3,2,rls(0.3));
```

Có thể tạo một đối tượng mô tả bộ cân bằng mới giống hệt với một bộ cân bằng đã có bằng cách dùng lệnh gán hoặc lệnh **copy**:

```
>> c2 = copy(c1) ; c1 và c2 là các đối tượng mô tả bộ cân bằng
```

```
>> c2 = c1;
```

Trong trường hợp thứ nhất, `c1` và `c2` độc lập với nhau, còn trong trường hợp 2, hai bộ cân bằng luôn có đặc tính giống nhau.

18.3.3. TRUY XUẤT VÀ HIỆU CHỈNH CÁC ĐẶC TÍNH CỦA BỘ CÂN BẰNG THÍCH NGHI

Trong quá trình sử dụng bộ cân bằng, ta có thể phải tham khảo hoặc phải thay đổi một số đặc tính của nó. Giả sử bộ cân bằng được mô tả bởi đối tượng có tên là `eqobj`, muốn truy xuất một đặc tính 'Prop' nào đó của bộ cân bằng, ta dùng cú pháp `eqobj.Prop`. Các đặc tính thường được truy xuất gồm có:

- Các đặc tính về cấu trúc bộ cân bằng (ví dụ số lượng trọng số)
- Các thông tin liên quan đến giải thuật cập nhật trọng số thích nghi được sử dụng. Ví dụ, kích cỡ bước,...
- Thông tin về trạng thái hiện tại của bộ cân bằng (ví dụ giá trị các trọng số)
- Những chỉ dẫn về phương cách tác động lên tín hiệu (ví dụ có reset hay không trước khi thực hiện cân bằng)

Lưu ý: khi ta thay đổi một thuộc tính của bộ cân bằng thì có thể có những thuộc tính liên quan cũng sẽ thay đổi theo (ví dụ: thay đổi số trọng số thì vector giá trị các trọng số cũng bị thay đổi).

18.3.4. SỬ DỤNG BỘ CÂN BẰNG THÍCH NGHI

Phần này sẽ trình bày cách sử dụng bộ cân bằng để tác động lên một tín hiệu bị nhiễu trên kênh truyền. Để thực hiện việc này, đầu tiên bộ cân bằng phải được khởi động với một chuỗi dữ liệu đã xác định trước gọi là chuỗi dữ liệu huấn luyện. Chế độ này gọi là chế độ huấn luyện, đây là quá trình mà bộ cân bằng sẽ thu thập các thông tin về đặc tính của kênh truyền để cập nhật các trọng số một cách phù hợp. Sau bước này bộ cân bằng sẽ tiếp tục hoạt động với tín hiệu vào là tín hiệu cần phải cân bằng, chế độ này gọi là chế độ hướng quyết định. Trong chế độ này, các dữ liệu ngõ ra bộ cân bằng chính là tín hiệu đã được khắc phục nhiễu.

Hàm **equalize** sẽ thực hiện nhiệm vụ cân bằng một tín hiệu cho trước. Nếu chúng ta cung cấp vector chuỗi huấn luyện như một thông số nhập của hàm **equalize**, bộ cân bằng đầu tiên sẽ hoạt động ở chế độ huấn luyện, sau khi đã xử lý hết chuỗi huấn luyện, nó sẽ tự động chuyển sang chế độ hướng quyết định. Nếu không đưa vào chuỗi huấn luyện, bộ cân bằng sẽ hoạt động ở chế độ hướng quyết định ngay lập tức. Đặc biệt, các bộ cân bằng sử dụng giải thuật CMA không cần chuỗi huấn luyện nên ta không đưa vào vector này.

```
>> [y, yd, e] = equalize(eqobj, x, trainsig)
```

trong đó `eqobj` là biến đối tượng mô tả bộ cân bằng, `x` là tín hiệu cần xử lý, `trainsig` là vector biểu diễn chuỗi huấn luyện, `y` là tín hiệu ngõ ra sau khi cân bằng, `yd` là chuỗi ký hiệu

mà bộ cân bằng đã phát hiện được, còn e là vector sai số giữa y và tín hiệu tham chiếu (chuỗi huấn luyện hoặc chuỗi dữ liệu mà bộ cân bằng phát hiện).

 **Ví dụ 18-1.** Sử dụng bộ cân bằng tuyến tính định khoảng ký hiệu, có 8 trọng số, cập nhật theo giải thuật LMS với kích cỡ bước 0.01 để cân bằng một tín hiệu QPSK.

Trong ví dụ này, ta chọn chuỗi dữ liệu huấn luyện chính là 500 mẫu đầu tiên của tín hiệu phát. Kết quả cân bằng được minh họa bằng đồ thị phân bố của các tín hiệu bị nhiễu, tín hiệu sau khi cân bằng và tín hiệu QPSK lý tưởng.

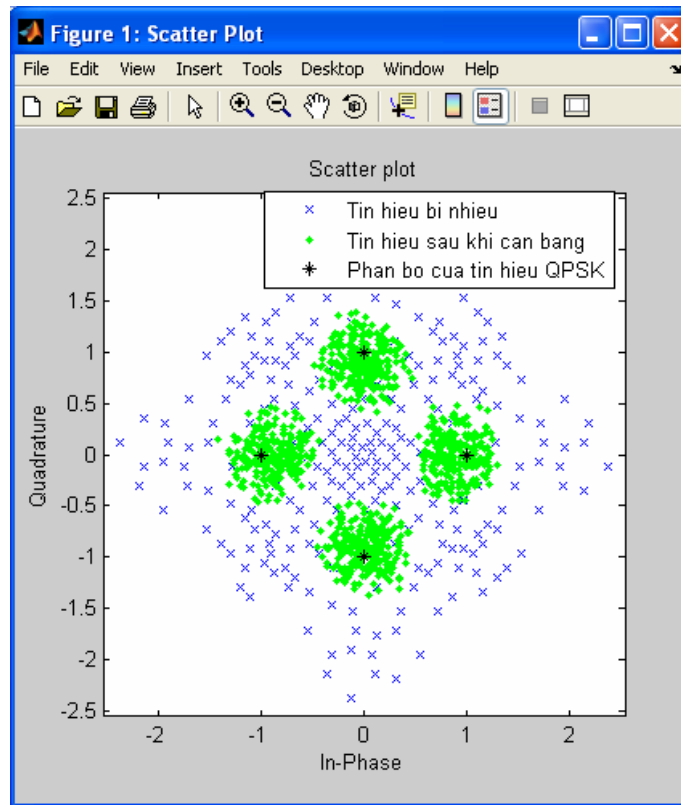
```
% Thiết lập các thông số và tín hiệu.
M = 4; % Kích thước tập ký hiệu điều chế
msg = randint(1500,1,M); % Thông điệp ngẫu nhiên
modmsg = pskmod(msg,M); % Điều chế QPSK
trainlen = 500; % Chiều dài chuỗi huấn luyện
chan = [.986; .845; .237; .123+.31i]; % Các hệ số của kênh truyền
filtmsg = filter(chan,1,modmsg); % Mô phỏng méo kênh truyền.
% Cân bằng tín hiệu thu.
eq1 = lineareq(8, lms(0.01)); % Khởi tạo đối tượng mô tả bộ cân bằng.
eq1.SigConst = pskmod([0:M-1],M); % Thiết lập vector mô tả dạng tín hiệu.
[symbolest,yd] = equalize(eq1,filtmsg,modmsg(1:trainlen)); % Cân bằng.
% Vẽ đồ thị phân bố.
h = scatterplot(filtmsg,1,trainlen,'bx'); hold on;
scatterplot(symbolest,1,trainlen,'g.',h);
scatterplot(eq1.SigConst,1,0,'k*',h);
legend('Tín hiệu bị méo','Tín hiệu sau khi cân bằng',...
'Phân bố tín hiệu lý tưởng');
hold off;
% Tính tỷ lệ lỗi khi có và không có bộ cân bằng.
demodmsg_noeq = pskdemod(filtmsg,M); % Giải điều chế tín hiệu chưa cân
bằng.
demodmsg = pskdemod(yd,M); % Giải điều chế tín hiệu phát hiện bởi bộ cân
bằng.
[nnoeq,rnoeq] = symerr(demodmsg_noeq(trainlen+1:end),...
msg(trainlen+1:end));
[neq,req] = symerr(demodmsg(trainlen+1:end),...
msg(trainlen+1:end));
disp('Tỷ lệ lỗi ký hiệu khi có và không có bộ cân bằng:')
disp([req rnoeq])
```

Kết quả thực thi chương trình:

Tỷ lệ lỗi ký hiệu khi có và không có bộ cân bằng:

```
0      0.3310
```

Đồ thị phân bố:



Hình 18.4.

Sau đây là một ví dụ khác minh họa bộ cân bằng thích nghi hoạt động ở chế độ hướng quyết định. Nếu đặc tính kênh truyền không thay đổi, ta có thể sử dụng lại bộ cân bằng đã được huấn luyện trước đó để cân bằng các tín hiệu tiếp theo đi vào bộ cân bằng. Chỉ cần thiết lập thuộc tính `ResetBeforeFiltering = 0` (không reset sau mỗi lần thực hiện cân bằng). Ví dụ:

```
% Cân bằng tín hiệu thu theo từng đoạn.
eqlms.ResetBeforeFiltering = 0; % Không reset trước khi cân bằng
% 1. Xử lý chuỗi dữ liệu huấn luyện.
s1 = equalize(eqlms,filtmsg(1:trainlen),modmsg(1:trainlen));
% 2. Xử lý phần dữ liệu tiếp theo ở chế độ quyết định.
s2 = equalize(eqlms,filtmsg(trainlen+1:800));
% 3. Xử lý phần dữ liệu còn lại ở chế độ quyết định.
s3 = equalize(eqlms,filtmsg(801:end));
s = [s1; s2; s3]; % Toàn bộ ngõ ra của bộ cân bằng
```

▪ Thời gian trễ của bộ cân bằng: khi sử dụng các bộ cân bằng thích nghi ngoại trừ các bộ cân bằng dùng giải thuật CMA, ta cần thiết lập chỉ số tham chiếu `RefTap` sao cho nó vượt quá độ trễ tính theo số ký hiệu giữa ngõ ra bộ điều chế ở máy phát với ngõ vào của bộ cân bằng. Khi điều kiện này được thỏa mãn thì tổng thời gian trễ giữa ngõ ra bộ điều chế với ngõ ra bộ cân bằng là: $(\text{RefTap} - 1) / n_{\text{SampPerSym}}$ ký hiệu. Trong thực tế, do ta không biết chính xác độ trễ của kênh truyền nên thường chọn chỉ số tham chiếu ứng với tap trung tâm của bộ cân bằng tuyến tính, hoặc tap trung tâm của bộ lọc thuận trong bộ cân bằng DFE.

Đối với các bộ cân bằng sử dụng giải thuật CMA, biểu thức trên không có ý nghĩa vì bộ cân bằng không có chỉ số tham chiếu. Muốn xác định độ trễ của kênh truyền, ta dựa vào dữ liệu

thực nghiệm sau khi các trọng số của bộ cân bằng đã hội tụ. Có thể sử dụng hàm **xcorr** để xác định tương quan chéo giữa ngõ ra bộ điều chế và ngõ ra bộ cân bằng (đánh lệnh **help xcorr** ở cửa sổ lệnh để xem chi tiết về hàm này).

Có hai cách để đưa độ trễ D vào quá trình khảo sát: chèn thêm dữ liệu hoặc cắt bớt dữ liệu.

- Thêm vào tập dữ liệu gốc D ký hiệu ở đoạn cuối. Trước khi so sánh dữ liệu gốc và dữ liệu thu, bỏ đi D ký hiệu ở phần đầu của dữ liệu thu. Theo cách này, toàn bộ dữ liệu gốc (không tính phần thêm vào) đều được so sánh với dữ liệu thu.
- Trước khi so sánh, bỏ đi D ký hiệu cuối của dữ liệu gốc và D ký hiệu đầu tiên của dữ liệu thu. Theo cách này, một số ký hiệu gốc sẽ không được so sánh với dữ liệu thu.

 **Ví dụ 18-2.** Sử dụng bộ cân bằng tuyến tính định khoảng ký hiệu, có 3 trọng số, cập nhật theo giải thuật LMS chuẩn hoá để cân bằng một tín hiệu BPSK, có xét đến độ trễ của kênh truyền, dùng phương pháp cắt bớt dữ liệu.

```
M = 2; % Sử dụng phương pháp điều chế BPSK.
msg = randint(1000,1,M); % Dữ liệu ngẫu nhiên
modmsg = pskmod(msg,M); % Điều chế.
trainlen = 100; % Chiều dài của chuỗi huấn luyện
trainsig = modmsg(1:trainlen); % Chuỗi huấn luyện
% Định nghĩa bộ cân bằng.
eqclin = lineareq(3,normlms(.0005,.0001),pskmod(0:M-1,M));
eqclin.RefTap = 2; % Thiết lập chỉ số tham chiếu là tap trung tâm.
[eqsig,detsym] = equalize(eqclin,modmsg,trainsig); % Cân bằng.
detmsg = pskdemod(detsym,M); % Giải điều chế tín hiệu đã phát hiện.
% Bù lại độ trễ do thiết lập chỉ số tham chiếu.
D = (eqclin.RefTap - 1)/eqclin.nSampPerSym;
trunc_detmsg = detmsg(D+1:end); % Loại bỏ D ký hiệu đầu của dữ liệu đã cân bằng.
trunc_msg = msg(1:end-D); % Bỏ đi D ký hiệu cuối của thông điệp.
% Tính tỷ lệ lỗi, không xét chuỗi huấn luyện.
[numerrs,ber] = biterr(trunc_msg(trainlen+1:end),...
trunc_detmsg(trainlen+1:end))
```

Kết quả xuất ra như sau:

```
numerrs =
    0
ber =
    0
```

▪ Trong trường hợp dữ liệu vào được chia thành nhiều vector liên tiếp nhau (ví dụ như trong các quá trình lặp vòng), ta phải sử dụng hàm **equalize** nhiều lần, lưu lại trạng thái nội của bộ cân bằng để chuẩn bị cho lần thực hiện kế tiếp. Nói cách khác, các giá trị cuối của các thuộc tính **WeightInputs** và **Weights** sau một vòng lặp cũng chính là giá trị khởi đầu trong vòng lặp kế tiếp. Có hai khả năng có thể xảy ra: sử dụng cùng một bộ cân bằng trong các vòng lặp khác nhau, hoặc thay đổi bộ cân bằng sau mỗi vòng lặp. Sau đây là quy trình sử dụng bộ cân bằng trong các vòng lặp ứng với hai khả năng nêu trên:

- Sử dụng cùng một bộ cân bằng trong các vòng lặp:
 - 1) Trước khi bắt đầu các vòng lặp, khởi tạo bộ cân bằng mà ta cần sử dụng.
 - 2) Thiết lập thuộc tính `ResetBeforeFiltering = 0` để bảo đảm tính liên tục giữa các lần lặp.
 - 3) Trong mỗi vòng lặp, sử dụng hàm **equalize** để thực hiện cân bằng.
- Thay đổi bộ cân bằng trong các vòng lặp:
 - 1) Trước khi bắt đầu các vòng lặp, khởi tạo các đối tượng mô tả các bộ cân bằng khác nhau mà ta sẽ sử dụng trong các vòng lặp. Ví dụ: tạo đối tượng `eqcma` mô tả bộ cân bằng dùng giải thuật CMA và một đối tượng `eqlms` mô tả bộ cân bằng dùng giải thuật LMS.
 - 2) Với mỗi đối tượng được tạo ra ở bước 1, thiết lập thuộc tính `ResetBeforeFiltering` bằng 0.
 - 3) Tạo một đối tượng mô tả bộ cân bằng `eq_current` để chỉ bộ cân bằng được sử dụng trong các vòng lặp và gán cho nó giá trị là đối tượng mô tả bộ cân bằng cần sử dụng trong vòng lặp đang xét. Ví dụ: trong vòng lặp đầu tiên, gán `eq_current = eqcma`, trong vòng lặp thứ hai, gán `eq_current = eqlms, ...`
 - 4) Trong mỗi vòng lặp, thực hiện các bước sau:
 - a) Sử dụng hàm **equalize** với đối tượng `eq_current` để thực hiện quá trình cân bằng
 - b) Copy giá trị của các thuộc tính `WeightInputs` và `Weights` từ đối tượng `eq_current` sang đối tượng mà chúng ta sẽ sử dụng trong vòng lặp kế. Ví dụ:


```
eqlms.WeightInputs = eq_current.WeightInputs;
eqlms.Weights = eq_current.Weights;
```
 - c) Định nghĩa lại đối tượng `eq_current` để chỉ đến đối tượng mô tả bộ cân bằng sẽ sử dụng trong vòng lặp kế tiếp. Như vậy, `eq_current` sẽ mô tả một bộ cân bằng mới nhưng vẫn duy trì các trạng thái cũ và các trọng số cũ.

 **Ví dụ 18-3.** Sử dụng bộ cân bằng tuyến tính để cân bằng một tín hiệu 16QAM, tín hiệu vào được tách thành ba chuỗi dữ liệu nối tiếp nhau. Đầu tiên, sử dụng giải thuật RLS, sau lần lặp thứ hai, chuyển sang giải thuật LMS.

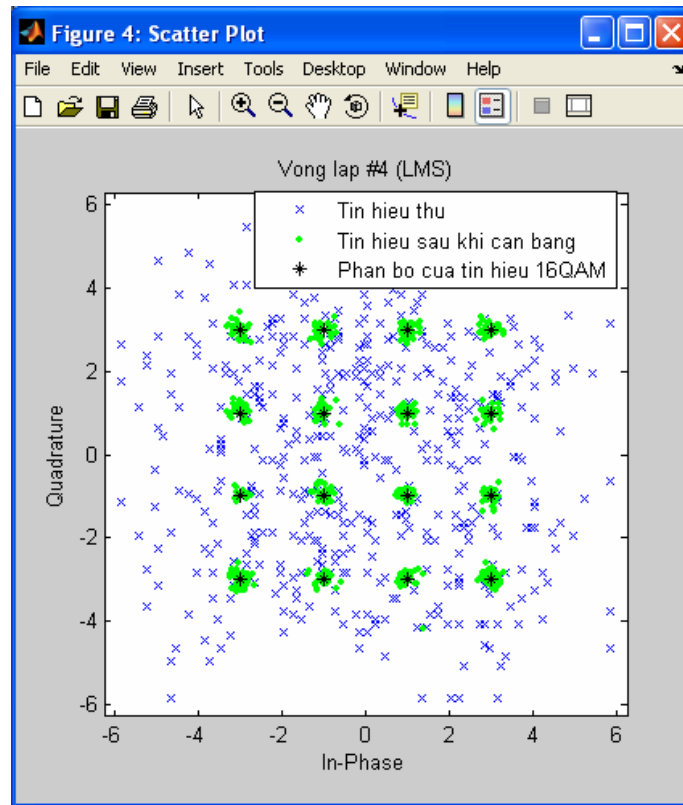
```
% Thiết lập các thông số.
M = 16; % Tập ký hiệu của pp điều chế
sigconst = gammod(0:M-1,M); % Dạng phân bố của tín hiệu 16-QAM
chan = [1 0.45 0.3+0.2i]; % Các hệ số của kênh truyền
% Thiết lập các bộ cân bằng.
eqrls = lineareq(6, rls(0.99,0.1)); % Khởi tạo một đối tượng mô tả bộ cân bằng RLS.
eqrls.SigConst = sigconst; % Mô tả tín hiệu.
eqrls.ResetBeforeFiltering = 0; % Duy trì sự liên tục giữa các lần lặp.
eqlms = lineareq(6, lms(0.003)); % Khởi tạo một đối tượng mô tả bộ cân bằng LMS.
eqlms.SigConst = sigconst; % Mô tả tín hiệu.
eqlms.ResetBeforeFiltering = 0; % Duy trì sự liên tục giữa các lần lặp.
```

```

eq_current = eqlrs; % Sử dụng bộ cân bằng RLS trong lần lặp đầu tiên.
% Vòng lặp chính
for jj = 1:4
    msg = randint(500,1,M); % Thông điệp ngẫu nhiên
    modmsg = qammod(msg,M); % Điều chế 16-QAM.
    % Thiết lập chuỗi huấn luyện cho lần lặp đầu tiên.
    if jj == 1
        ltr = 200; trainsig = modmsg(1:ltr);
    else
        % Sử dụng chế độ quyết định sau lần lặp đầu tiên.
        ltr = 0; trainsig = [];
    end
    % Mô phỏng méo kênh truyền
    filtmsg = filter(chan,1,modmsg);
    % Cân bằng tín hiệu thu.
    s = equalize(eq_current,filtmsg,trainsig);
    % Vẽ các tín hiệu.
    h = scatterplot(filtmsg(ltr+1:end),1,0,'bx'); hold on;
    scatterplot(s(ltr+1:end),1,0,'g.',h);
    scatterplot(sigconst,1,0,'k*',h);
    legend('Tin hieu thu','Tin hieu sau khi can bang','Dang phan bo tin hieu');
    title(['Iteration #' num2str(jj) ' (' eq_current.AlgType ')']);
    hold off;
    % Chuyển từ RLS sang LMS sau lần lặp thứ hai.
    if jj == 2
        eqlms.WeightInputs = eq_current.WeightInputs; % Copy các ngõ vào cuối vòng lặp.
        eqlms.Weights = eq_current.Weights; % Copy các trọng số ở cuối vòng lặp.
        eq_current = eqlms; % Sử dụng bộ cân bằng LMS.
    end
end
end

```

Ở mỗi vòng lặp, chương trình sẽ thực hiện vẽ đồ thị phân bố của tín hiệu thu, tín hiệu sau khi cân bằng và phân bố của tín hiệu 16QAM lý tưởng. Ví dụ, hình vẽ dưới mô tả kết quả sau khi thực hiện vòng lặp thứ tư:



Hình 18.5.

18.4. CÁC BỘ CÂN BẰNG MLSE

Các bộ cân bằng MLSE sử dụng giải thuật Viterbi để cân bằng các tín hiệu truyền qua các kênh truyền phân tán thời gian. Để sử dụng bộ cân bằng MLSE, dùng hàm **mlseeq** trong MATLAB Communication Toolbox. Hàm này nhận vào tín hiệu băng gốc đã được điều chế tuyến tính và xuất ra ước lượng chuỗi xác suất cực đại của tín hiệu, dựa vào ước lượng kênh truyền được mô tả dưới dạng một bộ lọc đáp ứng xung hữu hạn (FIR).

Giải thuật giải mã tín hiệu thu mà hàm **mlseeq** sử dụng bao gồm ba bước:

- Dùng bộ lọc FIR, tương ứng với ước lượng của kênh truyền để tác động lên các ký hiệu ngõ vào.
- Sử dụng giải thuật Viterbi để tính toán các đường dẫn lùi và độ đo trạng thái, chính là các số được gán cho các ký hiệu ở mỗi bước của giải thuật Viterbi. Các độ đo dựa trên không gian Euclide.
- Xuất ra ước lượng chuỗi xác suất lớn nhất, đó là một chuỗi các số phức tương ứng với các điểm phân bố (constellation) của tín hiệu sau khi điều chế.

Theo lý thuyết, bộ cân bằng MLSE cho chất lượng tốt nhất, nhưng quá trình tính toán phức tạp.

Cú pháp tổng quát của hàm **mlseeq** như sau:

```
>> y = mlseeq(x, chcffs, const, tblen, opmode, nsamp)
```

trong đó:

x là vector tín hiệu băng gốc cần cân bằng

chcffs là vector biểu diễn ước lượng của kênh truyền

`const` là vector định nghĩa dạng phân bố của tín hiệu lý tưởng (signal constellation)

`tblen` là một số dương xác định mức độ dò (traceback depth). Bộ cân bằng thực hiện dò từ trạng thái có độ đo tốt nhất.

`nsamp` là số mẫu trên một ký hiệu ngõ vào, còn gọi là hệ số lấy mẫu quá. Nếu `nsamp > 1` thì `chcffs` biểu diễn một kênh truyền được lấy mẫu quá.

`opmode` là một chuỗi cho biết chế độ hoạt động của bộ cân bằng, có thể nhận một trong các giá trị sau:

`'rst'`: không có delay:

```
>> y = mlseeq(...,'rst',nsamp,preamble,postamble)
```

Trong chế độ này, có thể thêm vào các vector với các phần tử từ 0 đến $M-1$ (M : bậc của phương pháp điều chế) vào trước và sau chuỗi dữ liệu. Các vector này được xác định tương ứng bởi các thông số `preamble` và `postamble`.

`'cont'`: tín hiệu ra bị delay một khoảng bằng `tblen` ký hiệu.

```
>> [y final_metric final_states final_inputs] = mlseeq(...,'cont',
,nsamp,init_metric,init_states,init_inputs)
```

Ở chế độ này, người sử dụng có thể cung cấp thêm các thông tin về các độ đo trạng thái khởi đầu và trạng thái đầu của bộ nhớ của bộ cân bằng,... đồng thời nhận về các thông tin tương ứng sau khi kết thúc cân bằng. Bảng 18.3 chỉ ra kích thước và giá trị có thể của các thông số này.

Bảng 18.3. Các thông số phụ liên quan đến hàm **mlseeq**

Thông số	Mô tả	Kích thước	Phạm vi giá trị
<code>init_metric</code>	Các độ đo trạng thái	$1 \times \text{numStates}$	Số thực
<code>init_states</code>	Các trạng thái traceback	$\text{numStates} \times \text{tblen}$	0 đến $\text{numStates} - 1$
<code>init_inputs</code>	Các ngõ vào traceback	$\text{numStates} \times \text{tblen}$	0 đến $M - 1$
$\text{numStates} = M^{(L-1)}$ với L là chiều dài đáp ứng xung của kênh truyền			

▪ Sau đây là một ví dụ sử dụng cú pháp cơ bản của hàm **mlseeq** để thực hiện cân bằng. Kết quả càng tốt nếu mức độ dò càng lớn, tuy nhiên thời gian tính toán lâu hơn.

Ví dụ 18-4. Sử dụng bộ cân bằng MLSE với traceback depth bằng 10 để cân bằng một tín hiệu điều chế QPSK. Tính thời gian thực thi và tỷ lệ lỗi ký hiệu. So sánh với trường hợp traceback depth bằng 2.

```
M = 4; const = pskmod([0:M-1],M); % Dạng điều chế 4-PSK
orgmsg = [1 2 2 0 3 1 3 3 2 1 0 2 3 0 1]';
msg = pskmod([1 2 2 0 3 1 3 3 2 1 0 2 3 0 1]',M); % Thông điệp sau điều chế
chcoeffs = [.986; .845; .237; .12345+.31i]; % Các hệ số kênh truyền
filtmsg = filter(chcoeffs,1,msg); % Mô phỏng méo kênh truyền.
tblen = 10; % Traceback depth
chanest = chcoeffs; % Giả sử đặc tính kênh truyền đã biết trước.
tic
msgEq = mlseeq(filtmsg,chanest,const,tblen,'rst'); % Cân bằng.
```

```
rcvmsg = pskdemod(msgEq,M); % Giải điều chế
disp('Truong hop 1:')
[numerr1,SER1] = symerr(orgmsg,rcvmsg) % Số ký hiệu lỗi và tỷ lệ ký hiệu lỗi
toc
tblen = 2; % Traceback depth
tic
msgEq = mlseeq(filtmsg,chanest,const,tblen,'rst'); % Cân bằng.
rcvmsg = pskdemod(msgEq,M); % Giải điều chế
disp('Truong hop 2:')
[numerr2,SER2] = symerr(orgmsg,rcvmsg) % Số ký hiệu lỗi và tỷ lệ ký hiệu lỗi
toc
```

Kết quả xuất ra màn hình:

```
Truong hop 1:
numerr1 =
    0
SER1 =
    0
Elapsed time is 0.010000 seconds.
Truong hop 2:
numerr2 =
    0
SER2 =
    0
Elapsed time is 0.050000 seconds.
```

▪ Chế độ hoạt động liên tục (continuos mode):

Trong trường hợp dữ liệu vào được phân hoạch thành nhiều vector nối tiếp nhau thì cách thích hợp nhất là sử dụng bộ cân bằng MLSE ở chế độ liên tục. Trong chế độ này, hàm **mlseeq** sẽ lưu lại các trạng thái trong của nó sau mỗi lần thực hiện và sẽ dùng các giá trị lưu trữ này làm các giá trị khởi đầu trong lần thực hiện tiếp theo. Để hoạt động ở chế độ liên tục, ta chọn thông số `opmode = 'cont'`. Quá trình thực hiện cân bằng trong chế độ liên tục gồm 2 bước như sau:

- Trước khi bắt đầu vòng lặp, khởi tạo ba ma trận rỗng `sm`, `ts`, `ti` để lưu các độ đo trạng thái, các trạng thái traceback và các ngõ vào traceback.
- Trong mỗi vòng lặp, dùng hàm **mlseeq** như sau:

```
[y,sm,ts,ti] = mlseeq(x,chcoeffs,const,tblen,'cont',nsamp,sm,ts,ti);
```

Dùng các thông số `sm`, `ts`, `ti` làm các thông số khởi đầu của lần lặp hiện tại, đồng thời sau khi thực hiện xong, hàm **mlseeq** lại cập nhật các giá trị ở cuối lần lặp vào trong các ma trận `sm`, `ts`, `ti` chuẩn bị cho lần lặp kế.

Trong chế độ liên tục, nếu ta có mức độ dò (traceback depth) bằng `tblen` thì ngõ ra sẽ bị trễ đi `tblen` ký hiệu. Điều này có nghĩa là `tblen` ký hiệu ngõ ra đầu tiên sẽ không liên quan đến tín hiệu vào, trong khi đó `tblen` ký hiệu cuối cùng của ngõ vào cũng không liên quan đến tín hiệu ngõ ra. Trong ví dụ sau, hàm **mlseeq** sử dụng mức traceback bằng 3 và 3 ký hiệu đầu tiên của ngõ ra không liên quan đến các ký hiệu ngõ vào (là vector `ones(1,10)`).

```
>> y = mlseeq(ones(1,10),1,[-7:2:7],3,'cont')
```

```
y =
```

```
-7 -7 -7 1 1 1 1 1 1 1
```

Vấn đề delay có ý nghĩa quan trọng khi ta cần so sánh chuỗi dữ liệu ngõ ra và ngõ vào, chẳng hạn để tính tỷ lệ lỗi ký hiệu, v.v....

 **Ví dụ 18-5.** Sử dụng bộ cân bằng MLSE với traceback depth bằng 10 hoạt động ở chế độ liên tục để cân bằng một tín hiệu điều chế 4-PSK được phân hoạch thành nhiều chuỗi dữ liệu nối tiếp. Tính toán tỷ lệ lỗi ký hiệu và vẽ đồ thị phân bố cho các tín hiệu trước và sau khi cân bằng.

Đầu tiên, cần tạo ba ma trận `sm`, `ts` và `ti` để lưu các thông số khởi đầu cho hoạt động của bộ cân bằng trong mỗi vòng lặp, các thông số này sẽ thay đổi sau mỗi lần lặp. Trong mỗi vòng lặp, tiên hành tạo thông điệp ngẫu nhiên, điều chế, lọc và cân bằng. Đồng thời, cuối mỗi vòng lặp, phải cập nhật các giá trị của `sm`, `ts` và `ti`.

Cuối cùng, tiên hành tính tỷ lệ lỗi ký hiệu tích lũy qua toàn bộ các vòng lặp. Khi tính toán, ta lưu ý rằng do có hiện tượng delay ở ngõ ra, nên ta phải bỏ đi `tblen` ký hiệu đầu tiên của chuỗi thu được và `tblen` ký hiệu cuối cùng của chuỗi phát trước khi so sánh. Điều này bảo đảm các ký hiệu được so sánh là các ký hiệu thật sự có nghĩa và có tính tương ứng với nhau.

Dưới đây là toàn bộ chương trình:

```
n = 200; % Số ký hiệu trong mỗi lần lặp
numiter = 25; % Số lần lặp
M = 4; % Điều chế 4-PSK.
const = pskmod(0:M-1,M); % Dạng tín hiệu 4-PSK
chcoeffs = [1 ; 0.25]; % Các hệ số của kênh truyền
chanest = chcoeffs; % Ước lượng kênh truyền
tblen = 10; % Mức độ traceback của bộ cân bằng
nsamp = 1; % Số mẫu dữ liệu vào trên một ký hiệu
sm = []; ts = []; ti = []; % Khởi tạo dữ liệu cho bộ cân bằng.
% Khởi tạo các dữ liệu có tính tích lũy.
fullmodmsg = []; fullfiltmsg = []; fullrx = [];
for jj = 1:numiter
    msg = randint(n,1,M); % Vector tín hiệu ngẫu nhiên
    modmsg = pskmod(msg,M); % Tín hiệu PSK
    filtmsg = filter(chcoeffs,1,modmsg); % Tín hiệu sau khi qua kênh truyền
    % Thực hiện cân bằng, sử dụng các giá trị khởi đầu là các giá trị có được
    % khi
    % kết thúc lần lặp trước, và lưu lại các giá trị cuối lần lặp này.
    [rx sm ts ti] = mlseeq(filtmsg,chanest,const,tblen,...
```

```

'cont',nsamp,sm,ts,ti);
% Cập nhật các vector bằng cách tích lũy thêm kết quả mới.
fullmodmsg = [fullmodmsg; modmsg];
fullfiltmsg = [fullfiltmsg; filtmsg];
fullrx = [fullrx; rx];
end
% Tính tổng số ký hiệu lỗi. Chú ý tới vấn đề delay.
numsymerrs = symerr(fullrx(tblen+1:end),fullmodmsg(1:end-tblen))
% Vẽ các tín hiệu trước và sau khi cân bằng.
h = scatterplot(fullfiltmsg); hold on;
scatterplot(fullrx,1,0,'r*',h);
legend('Filtered signal before equalization','Equalized signal',...
'Location','NorthOutside');
hold off;

```

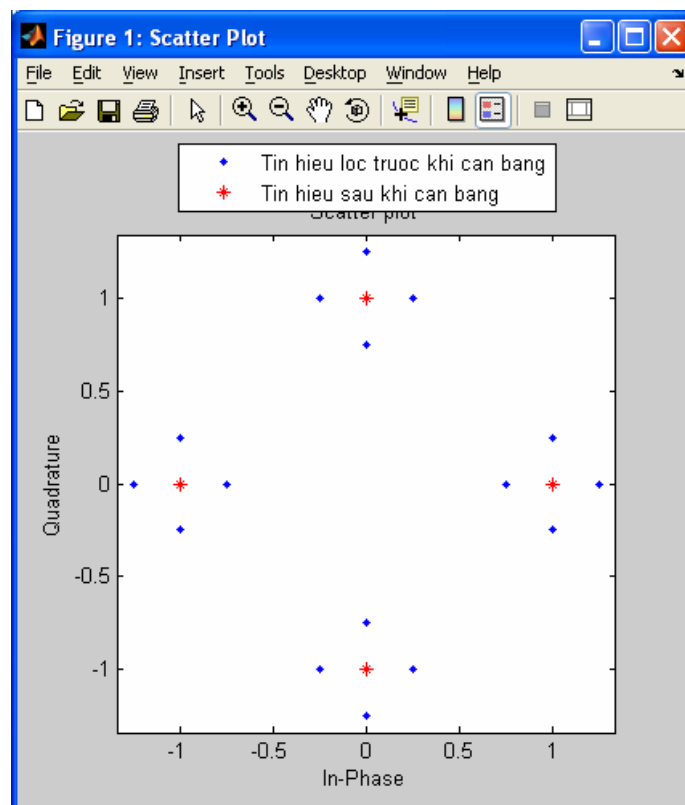
Kết quả thực thi chương trình như sau:

```

numsymerrs =
    0

```

Đồ thị phân bố:



Hình 18.6.

- Một số hệ thống thông tin sử dụng các chuỗi ký hiệu đã biết trước để chèn thêm vào đầu hoặc cuối chuỗi dữ liệu phát. Các chuỗi thêm vào này được gọi lần lượt là preamble và postamble. Hàm **mlseq** cũng cho phép ta thêm vào các vector này, với điều kiện bộ cân bằng

hoạt động trong chế độ không có delay (`opmode = 'rst'`). Khi gọi hàm `mlseeq`, ta cung cấp cho nó các vector mà các phần tử của nó là một chỉ số tham chiếu tới vector phân bố của tín hiệu điều chế lý tưởng. Ví dụ, với phương pháp điều chế 4-PSK, vector phân bố của tín hiệu là $[1 \ j \ -1 \ -j]$, vector preamble bằng $[1 \ 2 \ 4]$ có nghĩa là tín hiệu điều chế sẽ bắt đầu với $[1 \ j \ -1 \ -j]$. Nếu hệ thống không sử dụng preamble (hoặc postamble), ta cung cấp vector tương ứng bằng $[]$.

 **Ví dụ 18-6.** Mô phỏng quá trình cân bằng một tín hiệu điều chế 4-PSK dùng bộ cân bằng MLSE, hệ thống có sử dụng preamble và không sử dụng postamble.

```
M = 4; % Use 4-PSK modulation.
const = pskmod(0:3,4); % PSK constellation
tblen = 16; % Traceback depth for equalizer
preamble = [3; 1]; % Expected preamble, as integers
msgIdx = randint(98,1,M); % Random symbols
msgIdx = [preamble; msgIdx]; % Include preamble at the beginning.
msg = pskmod(msgIdx,M); % Modulated message
chcoeffs = [.623; .489+.234i; .398i; .21]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
filtmsg = filter(chcoeffs,1,msg); % Introduce channel distortion.
d = mlseeq(filtmsg,chanest,const,tblen,'rst',1,preamble,[]);
[nsymerrs ser] = symerr(msg,d) % Symbol error rate
```

Kết quả xuất ra của sổ lệnh của MATLAB:

```
nsymerrs =
    0
ser =
    0
```

Bài tập 18-1.

Thực hiện lại ví dụ 18-1 nhưng thay đổi giải thuật cập nhật trọng số. Cụ thể là dùng các giải thuật sau:

- Giải thuật LMS có dấu, kích cỡ bước 0,01.
- Giải thuật LMS chuẩn hoá với kích cỡ bước 0,01.
- Giải thuật LMS có kích cỡ bước thay đổi trong khoảng 0,005 đến 0,015, gia số 0,001.
- Giải thuật RLS, kích cỡ bước 0,01, forget factor = 0,99
- Giải thuật CMA, kích cỡ bước 0,01

Bài tập 18-2.

Làm lại bài 18-1 nhưng sử dụng bộ cân bằng định khoảng tỷ lệ với $K = 2$.

Bài tập 18-3.

Sử dụng bộ cân bằng tuyến tính để cân bằng một tín hiệu 16QAM, tín hiệu vào được tách thành ba chuỗi dữ liệu nối tiếp nhau. Đầu tiên, sử dụng giải thuật RLS, sau lần lặp thứ hai, chuyển sang giải thuật CMA. Đặc tính kênh truyền là $chan = [1 \ 0.45 \ 0.3+0.2i]$.

Bài tập 18-4.

Xây dựng một bộ cân bằng DFE định khoảng tỷ lệ với $K = 2$, có 5 trọng số ở nhánh thuận và 3 trọng số ở nhánh hồi tiếp, sử dụng giải thuật RLS với kích cỡ bước bằng 0,2. Dùng bộ cân bằng này để cân bằng một tín hiệu ngẫu nhiên điều chế bằng phương pháp QPSK, truyền qua kênh truyền có hệ số $[.623; .489+.234i; .398i; .21]$. Xác định tỷ lệ lỗi bit và thời gian thực thi.

Lần lượt thay đổi giá trị kích cỡ bước: 0,01; 0,05; 0,1; 0,5. Xác định lại tỷ lệ lỗi bit và thời gian thực thi. Nhận xét kết quả.

✎ **Bài tập 18-5.**

Thay bộ cân bằng DFE trong bài tập 18-4 bằng một bộ cân bằng MLSE có traceback bằng 2, hoạt động ở chế độ liên tục. So sánh thời gian thực thi và tỷ lệ bit lỗi so với bài tập 18-4. Sau đó tăng traceback lên các giá trị 5, 10, 20 và nhận xét kết quả.

✎ **Bài tập 18-6.**

Mô phỏng quá trình cân bằng một tín hiệu điều chế 16-QAM dùng bộ cân bằng MLSE, hệ thống có sử dụng preamble và postamble: preamble = [1; 3; 5; 6], postamble = [2; 1; 3; 4], traceback = 10. Kênh truyền có hệ số là $[.986; .845; .237; .123+.31i]$. Xác định tỷ lệ bit lỗi. Vẽ tín hiệu trước và sau khi cân bằng.

✎ ***Bài tập 18-7.**

Không dùng các hàm cân bằng có sẵn của MATLAB, hãy viết chương trình mô phỏng quá trình cân bằng tín hiệu điều chế 16-QAM như ở trong bài tập 18-6, sử dụng phương pháp nghịch đảo ma trận tự tương quan. So sánh kết quả với bài tập 18-6.

Danh sách các hàm được giới thiệu trong chương 18

cma	Khởi tạo đối tượng cân bằng theo giải thuật module không đổi
dfe	Tạo đối tượng mô tả bộ cân bằng hồi tiếp quyết định
equalize	Thực hiện quá trình cân bằng một tín hiệu
lms	Khởi tạo đối tượng cân bằng theo giải thuật LMS
lineareq	Tạo đối tượng mô tả bộ cân bằng thích nghi
mlseq	Tạo đối tượng mô tả bộ cân bằng MLSE
normlms	Khởi tạo đối tượng cân bằng theo giải thuật LMS chuẩn hoá
rls	Khởi tạo đối tượng cân bằng theo giải thuật RLS
signlms	Khởi tạo đối tượng cân bằng theo giải thuật LMS có dấu
varlms	Khởi tạo đối tượng cân bằng theo giải thuật LMS có kích cỡ bước thay đổi
xcorr	Hàm tương quan chéo giữa hai tín hiệu

PHỤ LỤC

A. CÁC HÀM SỬ DỤNG TRONG SÁCH KHÔNG CÓ SẴN TRONG MATLAB

Sau đây là các hàm được đề cập trong sách nhưng không được MATLAB cung cấp sẵn (hàm tự viết):

1. **btcode** (Phần 13.1)

```
function [out] = btcode (infile,bx,by,outfile)
% BTCODE (infile,singvals,outfile)
% Image Compression Using Block Truncation Coding.
% Written by Luigi Rosa - L'Aquila - Italia
% infile is input file name present in the current directory
% bx is a positive integer (x block size)
% by is a positive integer (y block size)
% outfile is output file name which will be created
%*****
if (exist(infile)==2)
    a = imread(infile);
    figure('Name','Input image');
    imshow(a);
else
    warndlg('The file does not exist.',' Warning ');
    out=[];
    return
end
%-----
if isgray(a)
    dvalue = double(a);
    dx=size(dvalue,1);
    dy=size(dvalue,2);
    % if input image size is not a multiple of block size image is resized
    modx=mod(dx,bx);
    mody=mod(dy,by);
    dvalue=dvalue(1:dx-modx,1:dy-mody);
    % the new input image dimensions (pixels)
    dx=dx-modx;
    dy=dy-mody;
    % number of non overlapping blocks required to cover
    % the entire input image
```

```

    nbx=size(dvalue,1)/bx;
    nby=size(dvalue,2)/by;
% the output compressed image
    matrice=zeros(bx,by);
% the compressed data
    m_u=zeros(nbx,nby);
    m_l=zeros(nbx,nby);
    mat_log=logical(zeros(bx,by));
    posbx=1;
    for ii=1:bx:dx
        posby=1;
        for jj=1:by:dy
            % the current block
            blocco=dvalue(ii:ii+bx-1,jj:jj+by-1);
            % the average gray level of the current block
            m=mean(mean(blocco));
            % the logical matrix corresponding to the current block
            blocco_binario=(blocco>=m);
            % the number of pixel (of the current block) whose gray level
            % is greater than the average gray level of the current block
            K=sum(sum(double(blocco_binario)));
            % the average gray level of pixels whose level is GREATER than
            % the block average gray level
            mu=sum(sum(double(blocco_binario).*blocco))/K;
            % the average gray level of pixels whose level is SMALLER than
            % the block average gray level
            if K==bx*by
                ml=0;
            else
                ml=sum(sum(double(~blocco_binario).*blocco))/(bx*by-K);
            end
            % the COMPRESSED DATA which correspond to the input image
            m_u(posbx,posby)=mu; %---> the m_u matrix
            m_l(posbx,posby)=ml; %---> the m_l matrix
            mat_log(ii:ii+bx-1,jj:jj+by-1)=blocco_binario;
                                                                    %--->logical matrix

            % the compressed image
            matrice(ii:ii+bx-1,jj:jj+by-
1)=(double(blocco_binario).*mu)+(double(~blocco_binario).*ml);
            posby=posby+1;

```

```

        end
        posbx=posbx+1;
    end
    % display the logical matrix
    figure('Name','Logical matrix');
    imshow(mat_log);
    if isa(a,'uint8')
        out=uint8(matrice);
        figure('Name','Compressed image');
        imshow(out);
        imwrite(out, outfile);
        return
    end
    if isa(a,'uint16')
        out=uint16(matrice);
        figure('Name','Compressed image');
        imshow(out);
        imwrite(out, outfile);
        return
    end
    if isa(a,'double')
        out=(matrice);
        figure('Name','Compressed image');
        imshow(out);
        imwrite(out, outfile);
        return
    end
end
end
%-----
if isrgb(a)
    double_a=double(a);
    ax=size(a,1)-mod(size(a,1),bx);
    ay=size(a,2)-mod(size(a,2),by);
    out_rgb=zeros(ax,ay,3);
    %-----
    %-----
    % ----- RED component -----
    dvalue=double_a(:,:,1);
    dx=size(dvalue,1);
    dy=size(dvalue,2);

```

```

% if input image size is not a multiple of block size image is resized
modx=mod(dx,bx);
mody=mod(dy,by);
dvalue=dvalue(1:dx-modx,1:dy-mody);
% the new input image dimensions (pixels)
dx=dx-modx;
dy=dy-mody;
% number of non overlapping blocks required to cover
% the entire input image
nbx=size(dvalue,1)/bx;
nby=size(dvalue,2)/by;
% the output compressed image
matrice=zeros(bx,by);
% the compressed data
m_u=zeros(nbx,nby);
m_l=zeros(nbx,nby);
mat_log=logical(zeros(bx,by));
posbx=1;
for ii=1:bx:dx
    posby=1;
    for jj=1:by:dy
        % the current block
        blocco=dvalue(ii:ii+bx-1,jj:jj+by-1);
        % the average gray level of the current block
        m=mean(mean(blocco));
        % the logical matrix corresponding to the current block
        blocco_binario=(blocco>=m);
        % the number of pixel (of the current block) whose gray level
        % is greater than the average gray level of the current block
        K=sum(sum(double(blocco_binario)));
        % the average gray level of pixels whose level is GREATER than
        % the block average gray level
        mu=sum(sum(double(blocco_binario).*blocco))/K;
        % the average gray level of pixels whose level is SMALLER than
        % the block average gray level
        if K==bx*by
            ml=0;
        else
            ml=sum(sum(double(~blocco_binario).*blocco))/(bx*by-K);
        end
    end
end

```

```

        % the COMPRESSED DATA which correspond to the input image
        m_u(posbx,posby)=mu;          %---> the m_u matrix
        m_l(posbx,posby)=ml;          %---> the m_l matrix
        mat_log(ii:ii+bx-1,jj:jj+by-1)=blocco_binario;
                                          %---> the logical matrix

        % the compressed image
        matrice(ii:ii+bx-1,jj:jj+by- ...
1)=(double(blocco_binario).*mu)+(double(~blocco_binario).*ml);
        posby=posby+1;
    end
    posbx=posbx+1;
end
out_rgb(:,:,1)=matrice;
% ----- GREEN component -----
dvalue=double_a(:,:,2);
dx=size(dvalue,1);
dy=size(dvalue,2);
% if input image size is not a multiple of block size image is resized
modx=mod(dx,bx);
mody=mod(dy,by);
dvalue=dvalue(1:dx-modx,1:dy-mody);
% the new input image dimensions (pixels)
dx=dx-modx;
dy=dy-mody;
% number of non overlapping blocks required to cover
% the entire input image
nbx=size(dvalue,1)/bx;
nby=size(dvalue,2)/by;
% the output compressed image
matrice=zeros(bx,by);
% the compressed data
m_u=zeros(nbx,nby);
m_l=zeros(nbx,nby);
mat_log=logical(zeros(bx,by));
posbx=1;
for ii=1:bx:dx
    posby=1;
    for jj=1:by:dy
        % the current block
        blocco=dvalue(ii:ii+bx-1,jj:jj+by-1);

```

```

% the average gray level of the current block
m=mean(mean(blocco));

% the logical matrix corresponding to the current block
blocco_binario=(blocco>=m);

% the number of pixel (of the current block) whose gray level
% is greater than the average gray level of the current block
K=sum(sum(double(blocco_binario)));

% the average gray level of pixels whose level is GREATER than
% the block average gray level
mu=sum(sum(double(blocco_binario).*blocco))/K;

% the average gray level of pixels whose level is SMALLER than
% the block average gray level
if K==bx*by
    ml=0;
else
    ml=sum(sum(double(~blocco_binario).*blocco))/(bx*by-K);
end

% the COMPRESSED DATA which correspond to the input image
m_u(posbx,posby)=mu;           %---> the m_u matrix
m_l(posbx,posby)=ml;           %---> the m_l matrix
mat_log(ii:ii+bx-1,jj:jj+by-1)=blocco_binario;
                                %---> the logical matrix

% the compressed image
matrice(ii:ii+bx-1,jj:jj+by- ...
1)=(double(blocco_binario).*mu)+(double(~blocco_binario).*ml);
    posby=posby+1;

end
posbx=posbx+1;
end
out_rgb(:,:,2)=matrice;

% ----- BLUE component -----
dvalue=double_a(:,:,3);
dx=size(dvalue,1);
dy=size(dvalue,2);

% if input image size is not a multiple of block size image is resized
modx=mod(dx,bx);
mody=mod(dy,by);
dvalue=dvalue(1:dx-modx,1:dy-mody);

% the new input image dimensions (pixels)
dx=dx-modx;

```

```

dy=dy-mody;
% number of non overlapping blocks required to cover
% the entire input image
nbx=size(dvalue,1)/bx;
nby=size(dvalue,2)/by;
% the output compressed image
matrice=zeros(bx,by);
% the compressed data
m_u=zeros(nbx,nby);
m_l=zeros(nbx,nby);
mat_log=logical(zeros(bx,by));
posbx=1;
for ii=1:bx:dx
    posby=1;
    for jj=1:by:dy
        % the current block
        blocco=dvalue(ii:ii+bx-1,jj:jj+by-1);
        % the average gray level of the current block
        m=mean(mean(blocco));
        % the logical matrix corresponding to the current block
        blocco_binario=(blocco>=m);
        % the number of pixel (of the current block) whose gray level
        % is greater than the average gray level of the current block
        K=sum(sum(double(blocco_binario)));
        % the average gray level of pixels whose level is GREATER than
        % the block average gray level
        mu=sum(sum(double(blocco_binario).*blocco))/K;
        % the average gray level of pixels whose level is SMALLER than
        % the block average gray level
        if K==bx*by
            ml=0;
        else
            ml=sum(sum(double(~blocco_binario).*blocco))/(bx*by-K);
        end
        % the COMPRESSED DATA which correspond to the input image
        m_u(posbx,posby)=mu; %---> the m_u matrix
        m_l(posbx,posby)=ml; %---> the m_l matrix
        mat_log(ii:ii+bx-1,jj:jj+by-1)=blocco_binario;
                                %--->logical matrix
    % the compressed image

```

```

        matrice(ii:ii+bx-1,jj:jj+by-...
1)=(double(blocco_binario).*mu)+(double(~blocco_binario).*ml);
        posby=posby+1;
    end
    posbx=posbx+1;
end
out_rgb(:, :, 3)=matrice;
%-----
if isa(a, 'uint8')
    out=uint8(out_rgb);
    figure('Name','Compressed image');
    imshow(out);
    imwrite(out, outfile);
    return
end
if isa(a, 'uint16')
    out=uint16(out_rgb);
    figure('Name','Compressed image');
    imshow(out);
    imwrite(out, outfile);
    return
end
if isa(a, 'double')
    out=(out_rgb);
    figure('Name','Compressed image');
    imshow(out);
    imwrite(out, outfile);
    return
end
end
%-----

```

2. dctcompr (Mục 13.2 chương 13)

```

function [im] = dctcompr (infile,coeff,outfile)
% DCTCOMPR (infile,coeff,outfile)
% Image compression based on Discrete Cosine Transform.
% Written by Luigi Rosa - L'Aquila - Italia
% infile is input file name present in the current directory
% coeff is the number of coefficients with the most energy
% outfile is output file name which will be created
%*****

```

```

if (exist(infile)==2)
    a = imread(infile);
    figure('Name','Input image');
    imshow(a);
else
    warndlg('The file does not exist.',' Warning ');
    im=[];
    return
end
if isrgb(a)
    if isa(a(:,:,1),'uint8')
        red = double(a(:,:,1));
        green = double(a(:,:,2));
        blue = double(a(:,:,3));
        red_dct=dct2(red);
        green_dct=dct2(green);
        blue_dct=dct2(blue);
        red_pow = red_dct.^2;
        green_pow = green_dct.^2;
        blue_pow = blue_dct.^2;
        red_pow=red_pow(:);
        green_pow=green_pow(:);
        blue_pow=blue_pow(:);
        [B_r,index_r]=sort(red_pow);
        [B_g,index_g]=sort(green_pow);
        [B_b,index_b]=sort(blue_pow);
        index_r=flipud(index_r);
        index_g=flipud(index_g);
        index_b=flipud(index_b);
        im_dct_r=zeros(size(red));
        im_dct_g=zeros(size(green));
        im_dct_b=zeros(size(blue));
        for ii=1:coeff
            im_dct_r(index_r(ii))=red_dct(index_r(ii));
            im_dct_g(index_g(ii))=green_dct(index_g(ii));
            im_dct_b(index_b(ii))=blue_dct(index_b(ii));
        end
        im_r=idct2(im_dct_r);
        im_g=idct2(im_dct_g);
        im_b=idct2(im_dct_b);
    end
end

```

```

        im=zeros(size(red,1),size(red,2),3);
        im(:,:,1)=im_r;
        im(:,:,2)=im_g;
        im(:,:,3)=im_b;
        im=uint8(im);
        imwrite(im, outfile);
        figure('Name','Output image');
        imshow(im);
        return;
end
if isa(a(:,:,1),'uint16')
    red = double(a(:,:,1));
    green = double(a(:,:,2));
    blue = double(a(:,:,3));
    red_dct=dct2(red);
    green_dct=dct2(green);
    blue_dct=dct2(blue);
    red_pow = red_dct.^2;
    green_pow = green_dct.^2;
    blue_pow = blue_dct.^2;
    red_pow=red_pow(:);
    green_pow=green_pow(:);
    blue_pow=blue_pow(:);
    [B_r,index_r]=sort(red_pow);
    [B_g,index_g]=sort(green_pow);
    [B_b,index_b]=sort(blue_pow);
    index_r=flipud(index_r);
    index_g=flipud(index_g);
    index_b=flipud(index_b);
    im_dct_r=zeros(size(red));
    im_dct_g=zeros(size(green));
    im_dct_b=zeros(size(blue));
    for ii=1:coeff
        im_dct_r(index_r(ii))=red_dct(index_r(ii));
        im_dct_g(index_g(ii))=green_dct(index_g(ii));
        im_dct_b(index_b(ii))=blue_dct(index_b(ii));
    end
    im_r=idct2(im_dct_r);
    im_g=idct2(im_dct_g);
    im_b=idct2(im_dct_b);

```

```

    im=zeros(size(red,1),size(red,2),3);
    im(:,:,1)=im_r;
    im(:,:,2)=im_g;
    im(:,:,3)=im_b;
    im=uint16(im);
    imwrite(im, outfile);
    figure('Name','Output image');
    imshow(im);
    return;
end
if isa(a(:,:,1),'double')
    red = double(a(:,:,1));
    green = double(a(:,:,2));
    blue = double(a(:,:,3));
    red_dct=dct2(red);
    green_dct=dct2(green);
    blue_dct=dct2(blue);
    red_pow = red_dct.^2;
    green_pow = green_dct.^2;
    blue_pow = blue_dct.^2;
    red_pow=red_pow(:);
    green_pow=green_pow(:);
    blue_pow=blue_pow(:);
    [B_r,index_r]=sort(red_pow);
    [B_g,index_g]=sort(green_pow);
    [B_b,index_b]=sort(blue_pow);
    index_r=flipud(index_r);
    index_g=flipud(index_g);
    index_b=flipud(index_b);

    im_dct_r=zeros(size(red));
    im_dct_g=zeros(size(green));
    im_dct_b=zeros(size(blue));
    for ii=1:coeff
        im_dct_r(index_r(ii))=red_dct(index_r(ii));
        im_dct_g(index_g(ii))=green_dct(index_g(ii));
        im_dct_b(index_b(ii))=blue_dct(index_b(ii));
    end
    im_r=idct2(im_dct_r);
    im_g=idct2(im_dct_g);

```

```

        im_b=idct2(im_dct_b);
        im=zeros(size(red,1),size(red,2),3);
        im(:,:,1)=im_r;
        im(:,:,2)=im_g;
        im(:,:,3)=im_b;
        imwrite(im, outfile);
        figure('Name','Output image');
        imshow(im);
        return;
    end
end
if isgray(a)
    dvalue=double(a);
    if isa(a,'uint8')
        img_dct=dct2(dvalue);
        img_pow=(img_dct).^2;
        img_pow=img_pow(:);
        [B,index]=sort(img_pow);
        B=flipud(B);
        index=flipud(index);
        compressed_dct=zeros(size(dvalue));
        for ii=1:coeff
            compressed_dct(index(ii))=img_dct(index(ii));
        end
        im=idct2(compressed_dct);
        im=uint8(im);
    end
    if isa(a,'uint16')
        img_dct=dct2(dvalue);
        img_pow=(img_dct).^2;
        img_pow=img_pow(:);
        [B,index]=sort(img_pow);
        B=flipud(B);
        index=flipud(index);
        compressed_dct=zeros(size(dvalue));
        for ii=1:coeff
            compressed_dct(index(ii))=img_dct(index(ii));
        end
        im=idct2(compressed_dct);
        im=uint16(im);
    end
end

```

```

end
if isa(a, 'double')
    img_dct=dct2(dvalue);
    img_pow=(img_dct).^2;
    img_pow=img_pow(:);
    [B,index]=sort(img_pow);
    B=flipud(B);
    index=flipud(index);
    compressed_dct=zeros(size(dvalue));
    for ii=1:coeff
        compressed_dct(index(ii))=img_dct(index(ii));
    end
    im=idct2(compressed_dct);
end
imwrite(im, outfile);
figure('Name','Output image');
imshow(im);
return;
end

```

3. svdcompr (Mục 13.3 chương 13)

```

function [im] = svdcompr (infile,singvals,outfile)
% IMCOMPR (infile,singvals,outfile)
% Image compression based on Singular Value Decomposition.
% Written by Luigi Rosa - L'Aquila - Italia
% infile is input file name present in the current directory
% singvals is the number of largest singular values (positive integer)
% outfile is output file name which will be created
% Compression ratio is equal to  $k(n+m+k) / n*m$ 
% where k is the number of singular values (singvals)
% and [n,m]=size(input_image)
%*****
if (exist(infile)==2)
    a = imread(infile);
    figure('Name','Input image');
    imshow(a);
else
    warndlg('The file does not exist.',' Warning ');
    im=[];
    return
end

```

```
if isrgb(a)
    if isa(a(:,:,1),'uint8')
        red = double(a(:,:,1));
        green = double(a(:,:,2));
        blue = double(a(:,:,3));
        [u,s,v] = svds(red, singvals);
        imred = uint8(u * s * transpose(v));
        [u,s,v] = svds(green, singvals);
        imgreen = uint8(u * s * transpose(v));
        [u,s,v] = svds(blue, singvals);
        imblue = uint8(u * s * transpose(v));
        im(:,:,1) = imred;
        im(:,:,2) = imgreen;
        im(:,:,3) = imblue;
        imwrite(im, outfile);
        figure('Name','Output image');
        imshow(im);
        return;
    end
    if isa(a(:,:,1),'uint16')
        red = double(a(:,:,1));
        green = double(a(:,:,2));
        blue = double(a(:,:,3));
        [u,s,v] = svds(red, singvals);
        imred = uint16(u * s * transpose(v));
        [u,s,v] = svds(green, singvals);
        imgreen = uint16(u * s * transpose(v));
        [u,s,v] = svds(blue, singvals);
        imblue = uint16(u * s * transpose(v));
        im(:,:,1) = imred;
        im(:,:,2) = imgreen;
        im(:,:,3) = imblue;
        imwrite(im, outfile);
        figure('Name','Output image');
        imshow(im);
        return;
    end
    if isa(a(:,:,1),'double')
        red = double(a(:,:,1));
        green = double(a(:,:,2));
```

```
        blue = double(a(:,:,3));
        [u,s,v] = svds(red, singvals);
        imred = (u * s * transpose(v));
        [u,s,v] = svds(green, singvals);
        imgreen = (u * s * transpose(v));
        [u,s,v] = svds(blue, singvals);
        imblue = (u * s * transpose(v));
        im(:,:,1) = imred;
        im(:,:,2) = imgreen;
        im(:,:,3) = imblue;
        imwrite(im, outfile);
        figure('Name','Output image');
        imshow(im);
        return;
    end
end
if isgray(a)
    dvalue=double(a);
    [u,s,v] = svds(dvalue, singvals);
    if isa(a,'uint8')
        im = uint8(u * s * transpose(v));
    end
    if isa(a,'uint16')
        im = uint16(u * s * transpose(v));
    end
    if isa(a,'double')
        im = (u * s * transpose(v));
    end
    imwrite(im, outfile);
    figure('Name','Output image');
    imshow(im);
    return;
end
```

B. MỤC LỤC TRA CỨU CÁC HÀM MATLAB SỬ DỤNG TRONG SÁCH

abs	3,9,69,101	box	38	dctmtx	164, 203
adapthisteq	186	bsc	239	de2bi	222
addpath	6	btcode	201,207, 208, 303	decode	256, 268
all	49,50	buttap	118-119	det	21, 62
amdemod	221	butter	118,125,126,127, 221	dfe	288, 289
ammod	221	buttord	118, 125	diag	21
angle	101,102	ceil	9	diric	93
any	49,50	cfirpm	129,130,139,140	disp	53, 71
arithdeco	215	char	69	dither	150
arithenco	215	cheb1ap	118, 119	double	10, 89, 151
awgn	231, 232, 238, 241, 274	cheb1ord	118, 126	dpcmdeco	209
axes	77	cheb2ap	118, 119	dpcmenco	209
axis	35, 36	cheb2ord	118, 126	dpcmopt	210, 211
bar	37, 180	chebwin	132	dpskdemod	225
barh	37	cheby1	118, 126	dpskmod	225
barthannwin	131	cheby2	118, 126, 142	ellip	118, 126, 142
bartlett	131	chirp	92	ellipap	118, 119
bchdec	256, 262	clear	6,7,11,25,46	ellipord	118, 126
bchenc	256, 262	clf	38	encode	258,264,267,268
bchgenpoly	256,262	close	44	equalize	286,289,292,293
berawgn	243	cma	287	error	62
bercoding	243	colfilt	191,192,196	errorbar	37
berconfint	245	compand	211, 212	exit	3
berfading	239,243	contour	42	eye	21
berfit	245-247	conv	94, 112	eyediagram	247, 249
bersync	243	conv2	95	fanbeam	168, 169
besselap	118, 119	convenc	273, 274	feval	65
besself	118, 126	convmtx	107, 109	fft	109, 162
bi2de	221	copy	235, 289	fft2	111, 162
bilinear	118, 123, 124	cos	4, 5, 66	fftfilt	97, 99, 114
biterr	240-242	cp2tform	159, 160	fftshift	111, 162
blackman	131	cyclgen	256, 265	figure	32, 37, 85
blackmanharris	131	cyclpoly	256, 265	fill	37, 60
blkproc	203	dct2	163	filter	96-100,112,141,237
bohman	132	dctcompr	203,204,208,310	filtfilt	97-99,112,118
find	51, 52	grid	36, 38	imdivide	153

findstr	70, 71, 74	grpdelay	102	imfilter	187, 189-192
fir1	129,130,133,134	gtext	38	imhist	178,180,181,194
fir2	129,130,133,134	guide	79, 80	imfinfo	151, 152, 170
fircls	129,130,137,138	hammggen	265	imlincomb	153
fircls1	129,130,137	hamming	265	immultiply	153
firls	129,130,133-136,143	hann	265	imnoise	186
firpm	129,130,133-136,143	help	5	impinvar	123
firpmord	129	helpwin	5	imread	150, 151, 170
firrcos	129, 130	hilbert	136	imresize	156
flattopwin	132	hist	37	imrotate	157
floor	9	histeq	182,184,186,195	imshow	155
fmdemod	221	hold off	34	imsubtract	153
fmmod	221	hold on	34	imtransform	158
fopen	94	huffmandeco	214	imview	155
for ... end	46,56,58,170	huffmandict	213, 214	imwrite	151
format	5, 10, 72	huffmanenco	214	ind2gray	150
fprintf	71-74	idct2	163	ind2rgb	150
fread	94	if ... else ... end	53	input	155,158,171,173
freqs	101	if ... elseif ... else end	53, 55	inv	5, 30
freqz	100-103, 113	if ... end	53	invfreqs	118
fscanf	94	ifanbeam	168, 169	invfreqz	118, 128, 129
fskdemod	225	ifft	109, 162	iradon	166
fskmod	225	ifft2	111, 162	isempty	58
fspecial	190-191	im2bw	150	istrellis	272
gauspuls	92	im2double	147, 152	kaiser	131
gausswin	132	im2uint16	152	kaiserord	129, 131
gen2par	256, 266	im2uint8	152	lact2tf	108
genqamdmod	225	imabsdiff	153	lactfilt	97
genqammod	225	imadd	153	legend	36-38
get	85	imadjust	176-180, 194	legend off	38
getframe	46	imag	230	length	14
gf	257	image	155	lineareq	288
global	64	imagesc	155	linspace	21
goertzel	111	imapprox	147, 152	lloyds	208
gray2ind	150	imcomplement	153	lms	287
grayslice	150	imcrop	158	load	7, 94
loglog	32	pie	37	roots	104, 109

logspace	21	plot	32,33,36,39,91,180,181	rotate3D	40
lookfor	5, 6	plot3	39	round	9
lp2bp	118, 121	pmdemod	221	rsdec	256, 259
lp2bs	118, 121	pmmmod	221	rsdecof	256
lp2hp	118, 121	polar	37	rsenc	256, 259
lp2lp	118, 121	poly	104, 109	rsencof	256
lpc	118	poly2trellis	271, 272, 275	rsgenpoly	256, 258
maketform	159	prony	118	save	7
mat2gray	150	pskdemod	225	sawtooth	92
max	21,27,62,95,104	pskmod	225	scatterplot	226, 249
maxflat	118, 127	pulstran	92	semianalytic	251
medfilt2	192, 193	pwd	6	semilogx	32
mesh	41, 42	qamdemod	225	semilogy	32
meshgrid	40	qammod	222, 223, 225	set	84, 85
min	11,19,27,62,205	quantiz	206, 208, 212	shading	42
mlseeq	295-300	quit	3	signlms	287
modnorm	225	rand	21, 58	sin	3,32,91,96,109
movie	46	randint	203, 206	sinc	93, 113
moviein	46	randn	21, 50, 235	single	10
mskmod	225	randsrc	203	size	21
nlfilt	191	rank	21	sos2ss	109
norm	21	rayleighchan	235	sos2tf	109
normlms	287	real	110, 223, 230	sos2zp	109
nuttallwin	132	rectwin	130	sprintf	74
ones	21,30,130,159,184	rem	55	square	92
oqpskdemod	225	repmat	215	ss2sos	109
oqpskmod	225	reset	237	ss2tf	109
ordfilt2	192, 193	reshape	27, 221, 222	ss2zp	109
padarray	192	residuez	105, 109	ssbdemod	221
pamdemod	225	resample	136	ssbmod	221
pammod	225	return	61, 64	stem	37, 180, 181
parzenwin	132	rgb2gray	150	stmcb	118
path	6	rgb2ind	150	strcat	46
pcolor	42	ricianchan	235	stretchlim	178, 179
peaks	42	rls	287	struct	272
phasez	101	rmpath	6	subplot	36, 38
sum	17,19,21,26,27,58	tril	21	while ... end	56-58,246

surf	42	triu	21	who	6, 7, 11
svdcompr	206, 315	tukeywin	132	whos	6
switch ... case end	55, 56	uicontextmenu	77	wiener2	194
symerr	240, 242	uicontrol	77,85-89,91,93,95	xcorr	292
text	38, 87	uimenu	77	xlabel	37, 38
tf2latc	107, 109	unwrap	101, 102	xor	49, 50
tf2ss	109	upfirdn	97, 98	ylabel	37, 38
tf2zp	104, 109	varlms	287	yulewalk	118,127,143
title	36-38	vitdec	274	zeros	21
trace	21	waterfall	41, 48	zp2sos	109
triang	131	Wgn	234	zp2ss	109
				zp2tf	104, 109

C. CÁC THUẬT NGỮ VIẾT TẮT

2D	Two Dimensions
3D	Three Dimensions
AM	Amplitude Modulation
AMBTC	Absolute Moment Block Truncating Coding
AM-DSC	Amplitude Modulation – Double Sideband Suppressed Carrier
AM-SSB	Amplitude Modulation – Single Sideband
ARMA	Autoregressive Moving-average
ASCII	American Standard Code for Information Interchange
BCH	Bose-Chaudhuri-Hocquenghem (Coding)
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
BTC	Block Truncating Coding
CCD	Charge – coupled Device
CLAHE	Contrast – Limited Adaptive Histogram Equalization
CLS	Constrained Least Square
CMA	Constant Modulus Algorithm
DBPSK	Differential Binary Phase Shift Keying
DCT	Discrete Cosine Transform
DFE	Decision Feedback Equalization
DFT	Discrete Fourier Transform
DPCM	Differential Pulse Code Modulation
DPSK	Differential Phase Shift Keying
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FM	Frequency Modulation
FSK	Frequency Shift Keying
GF	Galois Field
GIF	Graphics Interchange Format
GUI	Graphical User Interface
HBTC-VQ	Hierarchical Block Truncating Coding – Vector Quantization
HDF	Hierarchical Data Format
IDFT	Inverse Discrete Fourier Transform
IIR	Infinite Impulse Response
Inf	Infinite

ISI	Intersymbol Interference
JPEG	Joint Photographic Experts Group
LPC	Linear Predictive Coding
LMS	Least Mean Square
MLSE	Maximum – Likelihood Sequence Estimation
MPBTC	Moment Preserving Block Truncating Coding
MSB	Most Significant Bit
MSE	Mean Square Error
MSK	Minimum Shift Keying
NaN	Not a Number
OQPSK	Offset Quadrature Phase Shift Keying
PAM	Pulse Amplitude Modulation
PCM	Pulse Code Modulation
PM	Phase Modulation
PSK	Phase Shift Keying
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RGB	Red – Green – Blue
RF	Radio Frequency
RLC	Run Length Coding
RLS	Recursive Least Square
SNR	Signal to Noise Ratio
SOS	Second Order Sections
SSB	Single Sideband
SVD	Singular Value Decomposition
TIFF	Tagged Image File Format

TÀI LIỆU THAM KHẢO

- [1] Theodore Rappaport, “*Wireless Communications – Principles and Practice*”, 2nd Edition, Prentice Hall PTR, 2002.
- [2] R. C. Gonzalez, R.E. Woods, S.L. Eddins, “*Digital Signal Processing Using MATLAB*”, Prentice Hall PTR, ISBN 0-13-008519-7, 2004.
- [3] V.K. Ingle, J.G. Proakis, “*Digital Signal Processing Using MATLAB v4.0*”, International Thomson Publishing, ISBN 0-534-93805-1, 1997
- [4] The Mathworks, “*Communication Toolbox for use with MATLAB*”, Version 3.0.1, 2004.
- [5] The Mathworks, “*Signal Processing Toolbox for use with MATLAB*”, Version 6, 2004.
- [6] The Mathworks, “*Image Processing Toolbox for use with MATLAB*”, Version 4, 2003.
- [7] Pasi Franti, “*Image Compression*”, Lecture Notes, University of Joensuu, 2002.
- [8] Website: <http://www.mathworks.com/>