Chương 3: Lập trình Matlab

Viện Toán ứng dụng và Tin học, ĐHBK Hà Nội

Hà Nội, tháng 8 năm 2015

Nội dung

Mở đầu

Tiếp cận lập trình Matlab

Một thủ tục chuẩn của việc sử dụng lập trình MatLab cho việc giải quyết một bài toán kỹ thuật bao gồm các bước:

1. Phân tích bài toán và xác định thuật giải (trên giấy)
2. Phác thảo các công thức tính toán (trên giấy)
3. Viết chương trình MatLab (M-file) sử dụng MATLAB Editor/Debugger
4. Kiểm nghiệm và sửa lỗi
5. Giải bài toán

Mở đầu

Tiếp cận lập trình Matlab

Một thủ tục chuẩn của việc sử dụng lập trình MatLab cho việc giải quyết một bài toán kỹ thuật bao gồm các bước:

1. Phân tích bài toán và xác định thuật giải (trên giấy)
2. Phác thảo các công thức tính toán (trên giấy)
3. Viết chương trình MatLab (M-file) sử dụng MATLAB Editor/Debugger
4. Kiểm nghiệm và sửa lỗi
5. Giải bài toán

Mở đầu

Tiếp cận lập trình Matlab

Một thủ tục chuẩn của việc sử dụng lập trình MatLab cho việc giải quyết một bài toán kỹ thuật bao gồm các bước:

1. Phân tích bài toán và xác định thuật giải (trên giấy)
2. Phác thảo các công thức tính toán (trên giấy)
3. Viết chương trình MatLab (M-file) sử dụng MATLAB Editor/Debugger
4. Kiểm nghiệm và sửa lỗi
5. Giải bài toán

Mở đầu

Tiếp cận lập trình Matlab

Một thủ tục chuẩn của việc sử dụng lập trình MatLab cho việc giải quyết một bài toán kỹ thuật bao gồm các bước:

1. Phân tích bài toán và xác định thuật giải (trên giấy)
2. Phác thảo các công thức tính toán (trên giấy)
3. Viết chương trình MatLab (M-file) sử dụng MATLAB Editor/Debugger
4. Kiểm nghiệm và sửa lỗi
5. Giải bài toán

Mở đầu

Tiếp cận lập trình Matlab

Một thủ tục chuẩn của việc sử dụng lập trình MatLab cho việc giải quyết một bài toán kỹ thuật bao gồm các bước:

1. Phân tích bài toán và xác định thuật giải (trên giấy)
2. Phác thảo các công thức tính toán (trên giấy)
3. Viết chương trình MatLab (M-file) sử dụng MATLAB Editor/Debugger
4. Kiểm nghiệm và sửa lỗi
5. Giải bài toán

Mở đầu

- Các chương trình MatLab được chứa trong các "m-files"
  - Là các file văn bản thông thường, không phải file nhị phân
  - Các file phải có đuôi ".m"
- Các "m-file' phải được đặt trong đường dẫn hiện thời trong cửa sổ Command Window
  - MatLab quản lý đường dẫn trong của nó
  - Đường dẫn là một danh sách các thư mục mà MatLab sẽ tìm kiếm một "m-file" để thực thi
  - Một chương trình có thể tồn tại và không có lỗi nhưng có thể vẫn không chạy nếu MatLab không tìm thấy nó
  - Có thể thay đổi đường dẫn bằng cách dùng các lệnh `path`, `addpath` và `rmpath`.

Nội dung

Các thủ tục (Script Files)

- Không thực sự là các chương trình
  - ➢ Không có các dữ liệu input/output
  - ➢ Các biến thủ tục là một phần của không gian làm việc
- Hữu ích cho các công việc cố định
- Hữu ích như là một công cụ khi tạo các tài liệu cho các bài tập ở nhà

Lời khuyên

Các hàm (`functions`) có rất nhiều tiện ích so với các thủ tục (`scripts`)
$\Longrightarrow$ Luôn luôn sử dụng hàm thay cho thủ tục.

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Tất cả các biến được tạo ra trong thủ tục sẽ được thêm vào không gian làm việc. Điều này sẽ có ảnh hưởng đáng kể bởi vì

- Các biến đã tồn tại trong không gian làm việc có thể bị viết chồng lên
- Sự thực thi của các thủ tục có thể bị ảnh hưởng bởi trạng thái của các biến trong không gian làm việc.

Ví dụ 1

Thủ tục easyplot

```
% Load
D=load('xy.dat');          % D is a matrix with two columns
x=D(:,1);    y=D(:,2);     % x is the first column, y is second one
plot(x,y)                  % Generate the plot and label it
xlabel('x axis')
ylabel('y axis')
title('Plot of generic x-y data set')
```

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Thủ tục `easyplot` tác động lên không gian làm việc bằng cách tạo ra ba biến:

```
>> clear
>> who
                        (không có biến nào)
>> easyplot
>> who
Your variables are:
D   x   y
```

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Các thủ tục (Script Files)

Tác dụng phụ của các thủ tục

Nói chung, các tác dụng phụ:

- Diễn ra khi một chương trình thay đổi các biến ngoại trừ input/output
- Có thể gây các lỗi mà rất khó để phát hiện ra
- Không phải lúc nào cũng tránh được

Các tác dụng phụ của thủ tục

- Tạo ra và thay đổi các biến trong không gian làm việc
- Không đưa ra khuyến cáo rằng các biến trong không gian làm việc đã bị thay đổi.

Bởi vì các thủ tục có các tác dụng phụ, tốt hơn là gói gọn tất cả trong hàm "m-file".

Nội dung

Các hàm m-file

- Trong MatLab thì tên hàm phải trùng với tên của file có đuôi .m
- Hàm là các chương trình con:
  - ➢ Các hàm sử dụng các tham số đầu vào/ra để kết hợp chúng với các hàm khác và các lệnh window
  - ➢ Các hàm sử dụng các biến địa phương (local variables) mà chỉ tồn tại khi hàm đang thực thi. Các biến địa phương được phân biệt với các biến trùng tên trong không gian làm việc hoặc của các hàm khác.
- Các dữ liệu đầu vào cho phép cùng một thủ tục tính toán (cùng thuật toán) áp dụng với các dữ liệu khác nhau. Do đó, các hàm m-file có thể dùng lại nhiều lần.
- Các hàm có thể gọi các hàm khác
- Các thủ tục riêng có thể gói vào trong một hàm. Các tiếp cận này cho phép phát triển lời giải cấu trúc của các bài toán phức tạp.

Các hàm m-file

- Trong MatLab thì tên hàm phải trùng với tên của file có đuôi .m
- Hàm là các chương trình con:
  - ➢ Các hàm sử dụng các tham số đầu vào/ra để kết hợp chúng với các hàm khác và các lệnh window
  - ➢ Các hàm sử dụng các biến địa phương (local variables) mà chỉ tồn tại khi hàm đang thực thi. Các biến địa phương được phân biệt với các biến trùng tên trong không gian làm việc hoặc của các hàm khác.
- Các dữ liệu đầu vào cho phép cùng một thủ tục tính toán (cùng thuật toán) áp dụng với các dữ liệu khác nhau. Do đó, các hàm m-file có thể dùng lại nhiều lần.
- Các hàm có thể gọi các hàm khác
- Các thủ tục riêng có thể gói vào trong một hàm. Các tiếp cận này cho phép phát triển lời giải cấu trúc của các bài toán phức tạp.

Các hàm m-file

- Trong MatLab thì tên hàm phải trùng với tên của file có đuôi .m
- Hàm là các chương trình con:
  - ➢ Các hàm sử dụng các tham số đầu vào/ra để kết hợp chúng với các hàm khác và các lệnh window
  - ➢ Các hàm sử dụng các biến địa phương (local variables) mà chỉ tồn tại khi hàm đang thực thi. Các biến địa phương được phân biệt với các biến trùng tên trong không gian làm việc hoặc của các hàm khác.
- Các dữ liệu đầu vào cho phép cùng một thủ tục tính toán (cùng thuật toán) áp dụng với các dữ liệu khác nhau. Do đó, các hàm m-file có thể dùng lại nhiều lần.
- Các hàm có thể gọi các hàm khác
- Các thủ tục riêng có thể gói vào trong một hàm. Các tiếp cận này cho phép phát triển lời giải cấu trúc của các bài toán phức tạp.

Các hàm m-file

- Trong MatLab thì tên hàm phải trùng với tên của file có đuôi .m
- Hàm là các chương trình con:
  - ➢ Các hàm sử dụng các tham số đầu vào/ra để kết hợp chúng với các hàm khác và các lệnh window
  - ➢ Các hàm sử dụng các biến địa phương (local variables) mà chỉ tồn tại khi hàm đang thực thi. Các biến địa phương được phân biệt với các biến trùng tên trong không gian làm việc hoặc của các hàm khác.
- Các dữ liệu đầu vào cho phép cùng một thủ tục tính toán (cùng thuật toán) áp dụng với các dữ liệu khác nhau. Do đó, các hàm m-file có thể dùng lại nhiều lần.
- Các hàm có thể gọi các hàm khác
- Các thủ tục riêng có thể gói vào trong một hàm. Các tiếp cận này cho phép phát triển lời giải cấu trúc của các bài toán phức tạp.

Các hàm m-file

- Trong MatLab thì tên hàm phải trùng với tên của file có đuôi .m
- Hàm là các chương trình con:
  - ➢ Các hàm sử dụng các tham số đầu vào/ra để kết hợp chúng với các hàm khác và các lệnh window
  - ➢ Các hàm sử dụng các biến địa phương (local variables) mà chỉ tồn tại khi hàm đang thực thi. Các biến địa phương được phân biệt với các biến trùng tên trong không gian làm việc hoặc của các hàm khác.
- Các dữ liệu đầu vào cho phép cùng một thủ tục tính toán (cùng thuật toán) áp dụng với các dữ liệu khác nhau. Do đó, các hàm m-file có thể dùng lại nhiều lần.
- Các hàm có thể gọi các hàm khác
- Các thủ tục riêng có thể gói vào trong một hàm. Các tiếp cận này cho phép phát triển lời giải cấu trúc của các bài toán phức tạp.

Các hàm m-file

- Trong MatLab thì tên hàm phải trùng với tên của file có đuôi .m
- Hàm là các chương trình con:
  - ➢ Các hàm sử dụng các tham số đầu vào/ra để kết hợp chúng với các hàm khác và các lệnh window
  - ➢ Các hàm sử dụng các biến địa phương (local variables) mà chỉ tồn tại khi hàm đang thực thi. Các biến địa phương được phân biệt với các biến trùng tên trong không gian làm việc hoặc của các hàm khác.
- Các dữ liệu đầu vào cho phép cùng một thủ tục tính toán (cùng thuật toán) áp dụng với các dữ liệu khác nhau. Do đó, các hàm m-file có thể dùng lại nhiều lần.
- Các hàm có thể gọi các hàm khác
- Các thủ tục riêng có thể gói vào trong một hàm. Các tiếp cận này cho phép phát triển lời giải cấu trúc của các bài toán phức tạp.

Các hàm m-file

- Trong MatLab thì tên hàm phải trùng với tên của file có đuôi .m
- Hàm là các chương trình con:
  - ➢ Các hàm sử dụng các tham số đầu vào/ra để kết hợp chúng với các hàm khác và các lệnh window
  - ➢ Các hàm sử dụng các biến địa phương (local variables) mà chỉ tồn tại khi hàm đang thực thi. Các biến địa phương được phân biệt với các biến trùng tên trong không gian làm việc hoặc của các hàm khác.
- Các dữ liệu đầu vào cho phép cùng một thủ tục tính toán (cùng thuật toán) áp dụng với các dữ liệu khác nhau. Do đó, các hàm m-file có thể dùng lại nhiều lần.
- Các hàm có thể gọi các hàm khác
- Các thủ tục riêng có thể gói vào trong một hàm. Các tiếp cận này cho phép phát triển lời giải cấu trúc của các bài toán phức tạp.

Các hàm m-file

Cú pháp

Dòng đầu tiên của hàm "m-file" có dạng

```
function [outArgs]=funName(inArgs)
```

trong đó `outArgs` là danh sách các biến đầu ra, được đặt trong `[ ]`

- Các biến trong `outArgs` được cách nhau bởi dấu ","
- `[ ]` là tùy chọn nếu chỉ có 1 tham số đầu ra
- Hàm mà không có `outArgs` vẫn là hợp lệ

và danh sách các biến đầu vào `inArgs` được đặt trong `( )`

- Các biến trong `inArgs` được cách nhau bởi dấu ","
- Hàm mà không có `inArgs` vẫn là hợp lệ
- Có thể kiểm tra tính hợp lệ của tên hàm bằng cách dùng lệnh

```
>> isvarname funName
```

Các hàm m-file

Input và Output

twosum.m: two inputs, no output

```
function twosum(x,y)
% twosum  Add two matrices  and print the result
% two inputs, no output
x+y
```

threesum.m: three inputs, one output

```
function s=threesum(x,y,z)
% threesum  Add three matrices and return the result
% three inputs, one output
s=x+y+z;
```

addmult.m: two inputs, two outputs

```
function [s,p]=addmult(x,y)
% addmult   Compute sum and product of two matrices
% two inputs, two outputs
s=x+y;
```

Các hàm m-file

Input và Output

Ví dụ 2

Xét hàm `twosum`

```
>> twosum(2,2)
ans =
     4
>> x=[1 2]; y=[3 4];
>> twosum(x,y)
ans =
      4     6
>> A = [1 2; 3 4]; B = [5 6; 7 8];
>> twosum(A,B);
ans =
      6     8
     10    12
```

Các hàm m-file

Input và Output

Ví dụ 3

```
>> clear
>> x = 4; y = -2;
>> twosum(1,2)
ans =
     3
>> x+y
ans =
     2
>> disp([x y])
   4    -2
>> who
Your variables are:
ans  x  y
```

Trong ví dụ các biến `x` và `y` được định nghĩa trong không gian làm việc là khác với các biến `x, y` được xác định trong hàm `twosum`. Các biến `x, y` trong `twosum` là các biến địa phương trong hàm này.

Các hàm m-file

Tóm tắt về các tham số Input và Output

- Các giá trị được kết hợp thông qua các dữ liệu input và output
- Các biến được định nghĩa trong một hàm là biến địa phương. Các hàm khác và môi trường cửa sổ lệnh sẽ không "nhìn" được chúng.
- Số lượng các biến trả về nên trùng với số lượng các biến output trong hàm.

Các hàm m-file

Tóm tắt về các tham số Input và Output

- Các giá trị được kết hợp thông qua các dữ liệu input và output
- Các biến được định nghĩa trong một hàm là biến địa phương. Các hàm khác và môi trường cửa sổ lệnh sẽ không "nhìn" được chúng.
- Số lượng các biến trả về nên trùng với số lượng các biến output trong hàm.

Các hàm m-file

Tóm tắt về các tham số Input và Output

- Các giá trị được kết hợp thông qua các dữ liệu input và output
- Các biến được định nghĩa trong một hàm là biến địa phương. Các hàm khác và môi trường cửa sổ lệnh sẽ không "nhìn" được chúng.
- Số lượng các biến trả về nên trùng với số lượng các biến output trong hàm.

Nội dung

Nhập, xuất dữ liệu

Các hàm nhập dữ liệu

- Hàm `input` có thể được sử dụng để nhập dữ liệu từ bàn phím.
- Các tham số đầu vào của các hàm được ưa dùng hơn.

Các hàm xuất dữ liệu

- Hàm `disp` có thể được sử dụng cho các kết quả đơn giản
- Dùng hàm `fprintf` cho các dữ liệu định dạng trước.

Nhập, xuất dữ liệu

Xuất dữ liệu với `disp` và `fprintf`

Xuất dữ liệu trong cửa sổ lệnh được thực hiện với hàm `disp` hoặc `fprintf`. Nếu muốn ghi dữ liệu vào file bắt buộc phải dùng hàm `fprintf`.

disp

Sử dụng rất đơn giản. Tuy nhiên việc điều khiển định dạng của các output là rất hạn chế.

fprintf

Tương đối phức tạp hơn `disp`. Cung cấp toàn bộ các cách điều khiển định dạng của các output.

Nhập, xuất dữ liệu

Hàm `disp`

Cú pháp

```
disp(outMatrix)
```

trong đó `outMatrix` có thể là ma trận số hoặc xâu.

Ví dụ 4

```
>> disp(5)
     5
>> x = 1:3; disp(x)
     1     2     3
>> y = 3-x; disp([x; y])
     1     2     3
     2     1     0
>> disp([x y])
     1     2     3     2     1     0
>> disp([x' y])
??? Error using ==> horzcat
CAT arguments dimensions are not consistent.
```

Nhập, xuất dữ liệu

Hàm disp

Ví dụ 5

```
>> disp('Hello World!')
Hello World!
>> s='Have a nice day'; disp(s)
Have a nice day

>> t='You are using Matlab 7.10.0';
>> disp([s;t])
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.

>> disp(char(s,t))
Have a nice day
You are using Matlab 7.10.0
```

Nhập, xuất dữ liệu

Hàm `disp`

Chú ý 4.1

Lệnh `disp([s;t])` xuất hiện lỗi bởi vì `s` có ít ký tự hơn `t`. Hàm `char` tạo một ma trận xâu bằng cách đặt mỗi input trên một dòng riêng và chèn thêm các khoảng trắng nếu cần.

```
>> S=char(s,t);
>> length(s), length(t), length(S(1,:))
ans =
    15
ans =
    27
ans =
    27
```

Nhập, xuất dữ liệu

Hàm num2str

Hàm num2str thường được dùng với hàm disp để tạo ra dữ liệu đầu ra được gán nhãn của một giá trị số

Cú pháp

stringValue=num2str(numericValue)
chuyển numericValue thành một xâu biểu diễn giá trị số đó.

Ví dụ 6

```
>> num2str(pi)
ans =
3.1416
```

Nhập, xuất dữ liệu

Hàm num2str

```
>> A=eye(3)
A =
     1     0     0
     0     1     0
     0     0     1
>> S=num2str(A)
S =
1  0  0
0  1  0
0  0  1
```

Mặc dù A và S có vẻ chứa cùng các giá trị, chúng không tương đương. A là một ma trận số còn S là ma trận xâu.

```
>> A-S
??? Error using ==> minus
Matrix dimensions must agree.
```

Nhập, xuất dữ liệu

Sử dụng num2str với disp

```
>> x=sqrt(2);
>> outString=['x=',num2str(x)];
>> disp(outString)
x=1.4142
```

hoặc

```
>> disp(['x=',num2str(x)])
x=1.4142
```

Nhập, xuất dữ liệu

Sử dụng num2str với disp

Chú ý

Cấu trúc

```
disp(['x=',num2str(x)])
```

chỉ làm việc khi x là một ma trận hàng còn với ma trận cột thì không

```
>> y=1:4;
>> z=y';
>> disp(['z=',num2str(z)])
??? Error using ==> horzcat
CAT arguments dimensions are not consistent.
```

Nhập, xuất dữ liệu

Sử dụng num2str với disp

Thay vào đó, sử dụng hai lệnh disp để hiển thị cột của các vector hay ma trận

```
>> disp('z='); disp(z)
z=
     1
     2
     3
     4
```

hoặc đơn giản là nhập vào tên của biến mà không có dấu ";" cuối dòng

```
>> z
z =
     1
     2
     3
     4
```

Nhập, xuất dữ liệu

Hàm format

Hàm format điều chỉnh độ chính xác của dữ liệu in ra.

```
>> format short
>> disp(pi)
    3.1416
>> format long
>> disp(pi)
   3.141592653589793
```

Ngoài ra, thông số thứ hai của hàm num2str cũng có thể dùng với mục đích trên

```
>> disp(['pi=',num2str(pi,2)])
pi=3.1
>> disp(['pi=',num2str(pi,4)])
pi=3.142
>> disp(['pi=',num2str(pi,8)])
pi=3.1415927
```

Nhập, xuất dữ liệu

Hàm `fprintf`

Cú pháp

```
fprintf(outFormat, outVariables)
fprintf(filehandle, outFormat, outVariables)
```

sử dụng *outFormat* để chuyển *outVariables* thành các xâu được in ra. Trong dạng đầu tiên, kết quả sẽ hiển thị trong cửa sổ lệnh. Trong dạng thứ hai, kết quả sẽ được lưu vào file được tham chiếu bởi *fileHandle*.

Ví dụ 7

```
>> x=3;
>> fprintf('Square root of %g is %8.6f\n',x,sqrt(x))
Square root of 3 is 1.732051
```

Nhập, xuất dữ liệu

Hàm `fprintf`

Thành phần *outFormat* định rõ cách các *outVariables* được chuyển thành và hiển thị. Xâu *outFormat* có thể chứa bất kỳ một ký tự nào. Nó cũng phải chứa một mã chuyển đổi cho mỗi *outVariables*. Các mã chuyển đổi cơ bản được cho dưới bảng sau:

Mã	Dạng
`%s`	dạng xâu
`%d`	dạng số nguyên
`%f`	dạng dấu chấm động
`%e`	dạng dấu chấm động trong ký hiệu khoa học
`%g`	dạng gọn nhất của `%f` hoặc `%e`
`\n`	chèn một dòng mới sau xâu kết quả
`\t`	chèn một tab sau xâu kết quả

Nhập, xuất dữ liệu

Hàm `fprintf`

Ta có thể chỉ định thêm độ rộng và độ chính xác của kết quả bằng các cú pháp:

```
% wd
% w.pf
% w.pe
```

trong đó `w` là số ký tự trong độ rộng của kết quả cuối cùng và `p` là số chữ số sau dấu phẩy sẽ được hiển thị. Một số ví dụ

Giá trị	%8.4f	%12.3e	%10g	%8d
2	2.0000	2.000e+00	2	2
sqrt(2)	1.4142	1.414e+00	1.41421	1.414214e+00
sqrt(2e-11)	0.0000	4.472e-06	4.47214e-06	4.472136e-06
sqrt(2e11)	447213.5955	4.472e+05	447214	4.472136e+05

Nhập, xuất dữ liệu

Hàm `fprintf`

Có thể dùng `fprintf` để in vector hoặc ma trận dưới dạng ngắn gọn. Điều này có thể dẫn tới các kết quả không như mong muốn. Ví dụ

```
>> x=1:4; y=sqrt(x);
>> fprintf('%9.4f\n',y)
   1.0000
   1.4142
   1.7321
   2.0000
```

Ở đây, định dạng `%9.4f` được sử dụng lại cho mỗi thành phần của `y`. Điều này có thể sẽ không cho kết quả như mong muốn:

```
>> fprintf('y=%9.4f\n',y)
y=   1.0000
y=   1.4142
y=   1.7321
y=   2.0000
```

Nhập, xuất dữ liệu

Hàm `fprintf`

Hàm `fprintf` duyệt các *outVariables* theo các cột. Điều này cũng có thể dẫn đến các kết quả không như mong muốn

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

A =

     1     2     3
     4     5     6
     7     8     9

>> fprintf('%8.2f %8.2f % 8.2f \n', A )
    1.00     4.00     7.00
    2.00     5.00     8.00
    3.00     6.00     9.00
```

Nhập, xuất dữ liệu

Hàm `fprintf`

Xuất dữ liệu ra file

Để ghi dữ liệu ra file cần phải tạo ra một *fileHandle* với lệnh `fopen`. Tất cả tác dụng của các định dạng cũng như vector hóa đều có thể áp dụng.

Ví dụ 8

Lưu các thành phần của một vector vào một file

```
x=1:10;
fout=fopen('out.dat','wt');
fprintf(fout,'   k   x(k)\n');
for k=1:length(x)
    fprintf(fout,'%4d    % 5.2f\n',k,x(k));
end
fclose(fout)
```

Nội dung

Điều khiển luồng

Để có thể thực thi một thuật toán, một ngôn ngữ lập trình cần có các cấu trúc điều khiển

- Các cấu trúc lặp (Looping or Iteration)
- Các cấu trúc điều kiện: rẽ nhánh (Branching)
- So sánh (Comparison)

So sánh

Sự so sánh được thể hiện qua các toán tử quan hệ (Relational Operators). Các toán tử này được dùng để kiểm tra hai giá trị bằng nhau, nhỏ hơn, lớn hơn.

Toán tử	Ý nghĩa
<	$<$
<=	$\leq$
>	$>$
>=	$\geq$
==	$=$
~=	$\neq$

Điều khiển luồng

So sánh (tiếp)

- Khi áp dụng các toán tử quan hệ thì kết quả sẽ là một giá trị logic, tức là *True* hoặc *False*.
- Trong MatLab , các giá trị khác 0, bao gồm cả một xâu khác rỗng là tương đương với *True*. Chỉ có giá trị 0 là tương đương với *False*.

Chú ý 5.1

Trong các toán tử quan hệ <=, >= và ~= thì ký hiệu "=" phải đứng sau. Điều này có nghĩa =<,=> và =~ là không hợp lệ.

Điều khiển luồng

Các toán tử quan hệ

Ví dụ 9

Kết quả của một phép toán quan hệ là *True* (1) hoặc *False* (0)

```
>> a=3; b=5;
>> aIsSmaller=a<b
aIsSmaller =
     1
>> bisSmaller=b<a
bisSmaller =
     0
>> x=1:5; y=5:-1:1;
>> z=x>y
z =
     0     0     0     1     1
```

Điều khiển luồng

Các toán tử logic (Logical Operators)

Các toán tử logic được sử dụng để kết hợp các biểu thức logic (với "and" và "or") hoặc thay đổi giá trị logic với "not".

Toán tử	Ý nghĩa
&&	and
||	or
~	not

Điều khiển luồng

Các toán tử logic (Logical Operators)

Ví dụ 10

```
>> a=3; b=5;
>> aIsSmaller=a<b; bIsSmaller=b<a;
>> bothTrue=aIsSmaller && bIsSmaller
bothTrue =
     0
>> eitherTrue=aIsSmaller || bIsSmaller
eitherTrue =
     1
>> ~eitherTrue
ans =
     0
```

Điều khiển luồng

Các toán tử logic và quan hệ

Tóm tắt

- Các toán tử quan hệ liên quan đến các phép so sánh của hai giá trị.
- Kết quả của một phép toán quan hệ là một giá trị logic (True (1)/ False (0)).
- Các toán tử logic kết hợp (hoặc phủ định) các giá trị logic tạo ra các giá trị logic mới.
- Luôn có nhiều hơn một cách thể hiện cùng một phép so sánh.

Lời khuyên

Để bắt đầu, tập trung vào các so sánh đơn giản. Đừng sợ biểu thức logic quá dài (nhiều phép so sánh).

Điều khiển luồng

Cấu trúc điều kiện hoặc rẽ nhánh

- Dựa vào kết quả của một phép so sánh, hoặc của phép kiểm tra logic, các khối mã chương trình đã chọn sẽ được thực thi hoặc bỏ qua.
- Các cấu trúc điều kiện bao gồm: `if`, `if...else` và `if...elseif`, hoặc cấu trúc `switch`.

Có 3 dạng của cấu trúc `if`

1. `if`
2. `if...else`
3. `if...elseif`

Điều khiển luồng

Cấu trúc `if`

Cú pháp

```
if expression
    block of statements
end
```

Khối *block of statements* chỉ được thực thi nếu *expression* nhận giá trị *True*.

Ví dụ 11

```
if a<0
    disp('a is negative');
end
```

Nếu viết trên một dòng thì sau `if expression` cần có dấu ",":

```
if a<0,  disp('a is negative'); end
```

Điều khiển luồng

Cấu trúc `if...else` và `if...elseif`

```
if x<0
   error('x is negative; sqrt(x) is imaginary');
else
   r=sqrt(x);
end
```

```
if x>0
   disp('x is positive');
elseif x<0
   disp('x is negative');
else
   disp('x is exactly zero');
end
```

Điều khiển luồng

Cấu trúc switch

Câu lệnh switch rất hữu dụng khi tập giá trị của các biến kiểm tra là rời rạc (có thể là số nguyên hay xâu ký tự)

Cú pháp

```
switch    expression
     case value1
        block of statements
     case value2
        block of statements
        ...
     otherwise
        block of statements
end
```

Điều khiển luồng

Cấu trúc switch

Ví dụ 12

```
color=input('Enter your favorite color: ','s'); % color is a string
switch color
    case 'red'
        disp('Your color is red');
    case 'blue'
        disp('Your color is blue');
    case 'green'
        disp('Your color is green');
    otherwise
        disp('Your color is not red, blue or green');
end
```

Điều khiển luồng

Cấu trúc lặp `for`

Cú pháp

```
for  index=expression
     block of statements
end
```

Ví dụ 13

Tính tổng các thành phần của một vector

```
x=1:5; % create a row vector
sumx=0; % initialize the sum
for k=1:length(x)
   sumx=sumx+x(k);
end
```

Điều khiển luồng

Cấu trúc lặp `for`

Ví dụ 14

Vòng lặp `for` *với chỉ số tăng theo mức 2 đơn vị*

```
for k=1:2:n
   block of statements
end
```

Ví dụ 15

Vòng lặp `for` *với chỉ số giảm dần*

```
for k=n:-1:1
   block of statements
end
```

Điều khiển luồng

Cấu trúc lặp `for`

Ví dụ 16

Vòng lặp `for` *với chỉ số không phải là số nguyên*

```
for x=0:pi/15:pi
   fprintf('%8.2f   %8.5f\n',x,sin(x));
end
```

Chú ý 5.2

Trong ví dụ trên, `x` là một đại lượng vô hướng trong vòng lặp. Mỗi lần lặp, `x` được gán với 1 trong các cột của `0:pi/15:pi`.

Điều khiển luồng

Cấu trúc lặp `while`

Cú pháp

```
while expression
    block of statements
end
```

- Khối lệnh *block of statements* được thực thi nếu điều kiện *expression* vẫn là *True*.
- Để tránh tình trạng lặp vô hạn, nên đặt giới hạn trên cho số lần lặp.

Điều khiển luồng

Cấu trúc lặp `while`

Ví dụ 17

Giải phương trình $f(x) = 0$ trên khoảng phân ly nghiệm $[a, b]$ bằng phương pháp chia đôi

```
n=0;
while abs(b-a)>=err && n<maxit
    c=(a+b)/2;
    fc=feval(fun,c);
    if (fa*fc<0)
        b=c;
    else a=c;
    end
    n=n+1;
end
```

Điều khiển luồng

Cấu trúc lặp `while`

- Các câu lệnh `break` và `return` là các cách khác nhau để thoát khỏi một cấu trúc lặp. Cả hai lệnh này đều có thể dùng cho cấu trúc `for` và `while`.
- `break` được sử dụng để thoát khỏi phạm vi của vòng lặp hiện thời `for` hoặc `while`, chương trình sẽ tiếp tục sau đó.
- `return` được dùng để thoát khỏi một hàm hiện thời. Điều này sẽ ảnh hưởng đến việc thoát khỏi một vòng lặp. Bất kỳ một câu lệnh nào tiếp theo vòng lặp trong hàm đều bị bỏ qua.

Nội dung

Vector hóa

Vector hóa là việc sử dụng các phép toán vector để xử lý toàn bộ các phần tử của một vector hay ma trận. Thật ra các biểu thức vector hóa là tương đương với phép lặp trên các phần tử của ma trận hay vector. Biểu thức vector hóa sẽ ngắn gọn và thực thi nhanh hơn các biểu thức lặp thông thường.

- Sử dụng các phép toán vector thay cho vòng lặp khi có thể
- Tiền cấp phát bộ nhớ cho các vector hay ma trận
- Sử dụng việc đánh chỉ mục vector hóa và các hàm logic
- Mã không sử dụng vector hóa gọi là mã vô hướng (scalar code) bởi vì các phép toán được thực hiện trên các phần tử vô hướng của vector hay ma trận thay vì toàn bộ.

Lời khuyên

Chương trình tuy chậm mà chính xác còn hơn chương trình nhanh mà không chính xác.
$\Longrightarrow$ **Bắt đầu với các mã vô hướng, sau đó vector hóa nếu cần**

Vector hóa

Vector hóa là việc sử dụng các phép toán vector để xử lý toàn bộ các phần tử của một vector hay ma trận. Thật ra các biểu thức vector hóa là tương đương với phép lặp trên các phần tử của ma trận hay vector. Biểu thức vector hóa sẽ ngắn gọn và thực thi nhanh hơn các biểu thức lặp thông thường.

- Sử dụng các phép toán vector thay cho vòng lặp khi có thể
- Tiền cấp phát bộ nhớ cho các vector hay ma trận
- Sử dụng việc đánh chỉ mục vector hóa và các hàm logic
- Mã không sử dụng vector hóa gọi là mã vô hướng (scalar code) bởi vì các phép toán được thực hiện trên các phần tử vô hướng của vector hay ma trận thay vì toàn bộ.

Lời khuyên

Chương trình tuy chậm mà chính xác còn hơn chương trình nhanh mà không chính xác.
$\Longrightarrow$ **Bắt đầu với các mã vô hướng, sau đó vector hóa nếu cần**

Vector hóa

Vector hóa là việc sử dụng các phép toán vector để xử lý toàn bộ các phần tử của một vector hay ma trận. Thật ra các biểu thức vector hóa là tương đương với phép lặp trên các phần tử của ma trận hay vector. Biểu thức vector hóa sẽ ngắn gọn và thực thi nhanh hơn các biểu thức lặp thông thường.

- Sử dụng các phép toán vector thay cho vòng lặp khi có thể
- Tiền cấp phát bộ nhớ cho các vector hay ma trận
- Sử dụng việc đánh chỉ mục vector hóa và các hàm logic
- Mã không sử dụng vector hóa gọi là mã vô hướng (scalar code) bởi vì các phép toán được thực hiện trên các phần tử vô hướng của vector hay ma trận thay vì toàn bộ.

Lời khuyên

Chương trình tuy chậm mà chính xác còn hơn chương trình nhanh mà không chính xác.
$\Longrightarrow$ **Bắt đầu với các mã vô hướng, sau đó vector hóa nếu cần**

Vector hóa

Vector hóa là việc sử dụng các phép toán vector để xử lý toàn bộ các phần tử của một vector hay ma trận. Thật ra các biểu thức vector hóa là tương đương với phép lặp trên các phần tử của ma trận hay vector. Biểu thức vector hóa sẽ ngắn gọn và thực thi nhanh hơn các biểu thức lặp thông thường.

- Sử dụng các phép toán vector thay cho vòng lặp khi có thể
- Tiền cấp phát bộ nhớ cho các vector hay ma trận
- Sử dụng việc đánh chỉ mục vector hóa và các hàm logic
- Mã không sử dụng vector hóa gọi là mã vô hướng (scalar code) bởi vì các phép toán được thực hiện trên các phần tử vô hướng của vector hay ma trận thay vì toàn bộ.

Lời khuyên

Chương trình tuy chậm mà chính xác còn hơn chương trình nhanh mà không chính xác.
$\Longrightarrow$ **Bắt đầu với các mã vô hướng, sau đó vector hóa nếu cần**

Vector hóa

Vector hóa là việc sử dụng các phép toán vector để xử lý toàn bộ các phần tử của một vector hay ma trận. Thật ra các biểu thức vector hóa là tương đương với phép lặp trên các phần tử của ma trận hay vector. Biểu thức vector hóa sẽ ngắn gọn và thực thi nhanh hơn các biểu thức lặp thông thường.

- Sử dụng các phép toán vector thay cho vòng lặp khi có thể
- Tiền cấp phát bộ nhớ cho các vector hay ma trận
- Sử dụng việc đánh chỉ mục vector hóa và các hàm logic
- Mã không sử dụng vector hóa gọi là mã vô hướng (scalar code) bởi vì các phép toán được thực hiện trên các phần tử vô hướng của vector hay ma trận thay vì toàn bộ.

Lời khuyên

Chương trình tuy chậm mà chính xác còn hơn chương trình nhanh mà không chính xác.
$\Longrightarrow$ **Bắt đầu với các mã vô hướng, sau đó vector hóa nếu cần**

Vector hóa

Vector hóa là việc sử dụng các phép toán vector để xử lý toàn bộ các phần tử của một vector hay ma trận. Thật ra các biểu thức vector hóa là tương đương với phép lặp trên các phần tử của ma trận hay vector. Biểu thức vector hóa sẽ ngắn gọn và thực thi nhanh hơn các biểu thức lặp thông thường.

- Sử dụng các phép toán vector thay cho vòng lặp khi có thể
- Tiền cấp phát bộ nhớ cho các vector hay ma trận
- Sử dụng việc đánh chỉ mục vector hóa và các hàm logic
- Mã không sử dụng vector hóa gọi là mã vô hướng (scalar code) bởi vì các phép toán được thực hiện trên các phần tử vô hướng của vector hay ma trận thay vì toàn bộ.

Lời khuyên

Chương trình tuy chậm mà chính xác còn hơn chương trình nhanh mà không chính xác.
$\Longrightarrow$ **Bắt đầu với các mã vô hướng, sau đó vector hóa nếu cần**

Vector hóa

Vector hóa là việc sử dụng các phép toán vector để xử lý toàn bộ các phần tử của một vector hay ma trận. Thật ra các biểu thức vector hóa là tương đương với phép lặp trên các phần tử của ma trận hay vector. Biểu thức vector hóa sẽ ngắn gọn và thực thi nhanh hơn các biểu thức lặp thông thường.

- Sử dụng các phép toán vector thay cho vòng lặp khi có thể
- Tiền cấp phát bộ nhớ cho các vector hay ma trận
- Sử dụng việc đánh chỉ mục vector hóa và các hàm logic
- Mã không sử dụng vector hóa gọi là mã vô hướng (scalar code) bởi vì các phép toán được thực hiện trên các phần tử vô hướng của vector hay ma trận thay vì toàn bộ.

Lời khuyên

Chương trình tuy chậm mà chính xác còn hơn chương trình nhanh mà không chính xác.
$\Longrightarrow$ **Bắt đầu với các mã vô hướng, sau đó vector hóa nếu cần**

Vector hóa

Thay thế vòng lặp bởi các phép toán vector

Mã vô hướng

```
x=...
for k=1:length(x)
   y(k)=sin(x(k));
end
```

Mã vector hóa tương đương

```
x=...
y=sin(x);
```

Vector hóa

Tiền cấp phát bộ nhớ

Vòng lặp sau sẽ tăng chiều của s sau mỗi lần lặp

```
y=[4 -1 9 0];
for j=1:length(y)
    if y(j)>0
        s(j)=sqrt(y(j));
    else
        s(j)=0;
    end
end
```

Tiền cấp phát cho s trước khi gán các giá trị cho các thành phần

```
y=[4 -1 9 0];
s=zeros(size(y));
for j=1:length(y)
    if y(j)>0
        s(j)=sqrt(y(j));
    end
end
```

Vector hóa

Đánh chỉ mục vector hóa và các hàm logic

Việc vector hóa mã hoàn toàn đòi hỏi việc sử dụng việc đánh *chỉ số mảng* (array indexing) và đánh *chỉ số logic* (logical indexing).

Đánh chỉ số mảng

```
>> x=sqrt(0:4:20)
x =
     0    2.0000    2.8284    3.4641    4.0000    4.4721
>> i=[1 2 5];
>> y=x(i)
y =
     0     2     4
```

Biểu thức `y=x(i)` tương đương với đoạn mã

```
k=0;
for i=[1 2 5],  k=k+1;  y(k)=x(i);  end
```

Vector hóa

Đánh chỉ mục vector hóa và các hàm logic

Đánh chỉ số logic

```
>> x=sqrt(0:4:20)
x =
     0    2.0000    2.8284    3.4641    4.0000    4.4721
>> j=find(rem(x,2)==0)
j =
     1     2     5
>> z=x(j)
z =
     0     2     4
```

Vector hóa

Đánh chỉ mục vector hóa và các hàm logic

Ví dụ 18

Vector hóa mã vô hướng
Xét đoạn mã

```
y=. . .
s=zeros(size(y));
for j=1:length(y)
  if y(j)>0
    s(j)=sqrt(y(j));
  end
end
```

Thực ra, có thể thay thế toàn bộ vòng lặp bằng cách sử dụng đánh chỉ số logic hoặc đánh chỉ số mảng.

Vector hóa

Đánh chỉ mục vector hóa và các hàm logic

Ví dụ (tiếp)

```
y=. . .
s=zeros(size(y));
i=find(y>0);
s(y>0)=sqrt(y(y>0));
```

hoặc gọn hơn

```
y=. . .
s=zeros(size(y));
s(y>0)=sqrt(y(y>0));
```

Vector hóa

Vector hóa các phép sao chép

Sao chép toàn bộ các cột (hàng)

Mã vô hướng

```
[m,n]=size(A); % Giả sử rằng A và B có cùng số hàng (cột)
for i=1:m
  B(i,1)=A(i,1);
end
```

Mã vector hóa

```
B(:,1)=A(:,1);
```

Vector hóa

Vector hóa các phép sao chép

Sao chép và chuyển vị các ma trận con

Mã vô hướng

```
for j=2:3
  B(1,j)=A(j,3);
end
```

Mã vector hóa

```
B(1,2:3)=A(2:3,3);
```

Vector hóa

Một số ví dụ khác

Xóa các thành phần của một mảng

Để xóa các thành phần không phải là số (NaN) hoặc giá trị vô cùng (inf) của một mảng x ta có thể dùng đoạn mã sử dụng đánh chỉ số mảng

```
i=find(isnan(x) | isinf(x));  % Find bad elements
x(i)=[];                      % and delete them
```

hay một cách khác

```
i=find(~isnan(x) & ~isinf(x)); % Find elements that are are not NaN and not
x=x(i);                                % Keep those elements
```

Ta có thể thay đổi các đoạn mã trên bằng cách sử dụng chỉ số logic

```
x(isnan(x) | isinf(x))=[]; % Delete bad elements
```

hoặc

```
x=x(~isnan(x) & ~isinf(x)); % Keep good elements
```

Vector hóa

Một số ví dụ khác

Hàm từng khúc (Piecewise functions)

Hàm sinc được định nghĩa bởi $\mathrm{sinc}(x) = \begin{cases} \sin(x)/x, & x \neq 0 \\ 1, & x = 0. \end{cases}$

So sánh đoạn mã sử dụng lệnh `find`

```
function y=sinc(x)
y=ones(size(x));          % Set y to all ones, sinc(0)=1;
i=find(x~=0);             % Find nonzero x values
y(i)=sin(x(i))./x(i);     % Compute sinc when x ~=0
end
```

và một cách viết thú vị khác:

```
y=(sin(x)+(x==0))./(x+(x==0));
```

Vector hóa

Một số ví dụ khác

Nội suy đa thức

Cho n mốc nội suy $x_1, x_2, \ldots, x_n$ và các giá trị hàm tương ứng $y_1, y_2, \ldots, y_n$. Khi đó, các hệ số $c_0, c_1, \ldots, c_{n-1}$ của đa thức nội suy bậc $n-1$ có thể được tính bằng cách giải hệ

$$\begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \cdots & x_1^2 & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \cdots & x_2^2 & x_2 & 1 \\ \vdots & & & & \vdots & \\ x_n^{n-1} & x_n^{n-2} & \cdots & x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} c_{n-1} \\ c_{n-2} \\ \vdots \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

hay $Ac = y$.

Hệ trên có định thức Vandermond $|A| = \prod_{1 \le i < j}^{n} (x_j - x_i) \neq 0$ nên có nghiệm duy nhất.

Vector hóa

Một số ví dụ khác

Nội suy đa thức

```
function c=polyinterp(x,y)
x = x(:); y=y(:);          % Make sure that x and y are both column vectors
n = length(x);             % n= Number of points

%%% Compute the left-hand side matrix %%%
xMatrix = repmat(x,1,n);     % Make an nxn matrix with x on every column
powMatrix = repmat(n-1:-1:0,n,1);  % Make another nxn matrix of exponents
A = xMatrix .^ powMatrix;    % Compute the powers
c=A\y;                      % Solve the matrix equation for coefficients
end
```

Vector hóa

Một số ví dụ khác

Nội suy đa thức

Trong chương trình trên, để xây dựng ma trận vế trái A, trước hết tạo ra 2 ma trận $n \times n$ của cơ sở và lũy thừa, sau đó sử dụng toán tử lũy thừa từng từ `.^`. Hàm `repmat` ("replicate matrix") được sử dụng để tạo ma trận cơ sở `xMatrix` và ma trận lũy thừa `powMatrix`:

$$\texttt{xMatrix} = \begin{bmatrix} x(1) & x(1) & \cdots & x(1) \\ x(2) & x(2) & \cdots & x(2) \\ \vdots & & & \vdots \\ x(n) & x(n) & \cdots & x(n) \end{bmatrix}; \quad \texttt{powMatrix} = \begin{bmatrix} n-1 & n-2 & \cdots & 0 \\ n-1 & n-2 & \cdots & 0 \\ \vdots & & & \vdots \\ n-1 & n-2 & \cdots & 0 \end{bmatrix}$$

- Ma trận `xMatrix` được tạo bằng cách lặp lại vector cột `x` n lần.
- Ma trận `powMatrix` được tạo bởi một vector hàng với các thành phần $n-1, n-2, \ldots, 0$ lặp lại n lần.
- Đây chỉ là một ví dụ, có thể sử dụng hàm chuẩn của MatLab `polyfit` cho một loạt các đa thức nội suy (đọc thêm help để tìm hiểu thêm).

Nội dung

Quản lý các biến Input, Output

- Mỗi hàm có các biến nội tại bao gồm `nargin` (number of input arguments) và `nargout` (number of output arguments).
- Sử dụng giá trị `nargin` trong phần đầu của hàm để xác định có bao nhiêu biến đầu vào sẽ được sử dụng.
- Sử dụng giá trị `nargout` trong phần cuối của hàm để xác định số biến đầu ra mong muốn.

Lợi ích

- Cho phép một chương trình đơn có thể thực hiện nhiều công việc liên quan.
- Cho phép các hàm giả thiết các giá trị mặc định của một số biến đầu vào, do đó làm đơn giản việc sử dụng hàm trong một số trường hợp.

Quản lý các biến Input, Output

Ví dụ 19

Xét hàm `plot`

	`nargin`	`nargout`
`plot(x,y)`	2	0
`plot(x,y,'s')`	3	0
`plot(x,y,'s--')`	3	0
`plot(x1,y1,'s',x2,y2,'o')`	6	0
`h=plot(x,y)`	2	1

Các giá trị của `nargin` và `nargout` được xác định khi hàm `plot` được gọi ra.

Nội dung

Tính giá trị hàm một cách gián tiếp

Sử dụng hàm `feval`

Lợi ích

- Cho phép các thủ tục đã được viết xử lý một hàm $f(x)$ bất kỳ.
- Chia nhỏ một thuật toán phức tạp bằng cách sử dụng các đoạn mã riêng.

Ví dụ 20

```
function s=fsum(fun,a,b,n)
   x=linspace(a,b,n);
   y=feval(fun,x);
   s=sum(y);
end
```

```
>> fsum('sin',0,pi,5)
ans =
    2.4142
>> fsum('cos',0,pi,5)
ans =
     0
```

Các hàm `inline`

MatLab giới thiệu các mở rộng của lập trình hướng đối tượng (object-oriented programming - OOP). Hàm `inline` rất đơn giản và giúp chương trình linh hoạt hơn. Cụ thể, ta không cần viết các hàm m-files để tính giá trị một số hàm có công thức đơn giản và vẫn dùng được hàm `feval`.
Thay vì

```
function y=myFun(x)
y=x.^2-log(x);
```

ta dùng

```
myFun=inline('x.^2-log(x)');
```

Cả hai dạng khai báo trên của `myFun` cho phép các biểu thức dạng

```
z=myFun(3);
s=linspace(1,5);
t=myFun(s);
```

Nội dung

Chú thích

- Cú pháp : `% Matlab comment line`
- Các chú thích đặc biệt
  - Các khối comment liền nhau trong `m-file` chính là phần `help` của `m-file` đó:

    `>> help filename`

    $\Longrightarrow$ Khi viết một hàm `m-file`, cố gắng thêm các chú thích: mô tả mục đích của hàm, yêu cầu về các biến input và định dạng của các biến output.
  - Mã "cells" được phân định bởi `%% Cell title`
    - Trình soạn thảo Matlab Editor có những khả năng đặc biệt để làm việc với các "cells"
    - Sử dụng `publish('file.m')` để thực thi file.m và tạo ra các ouput dễ nhìn.

```
% publish all m-files in current directory
files=dir('*.m');
cellfun(@(x) publish(x,struct('evalCode',false)),...
    {files.name},'UniformOutput',false);
```

Nội dung

Gỡ lỗi

- MatLab hỗ trợ một trình gỡ lỗi tương tác
- Các lệnh `type` và `dbtype` hiển thị toàn bộ nội dung của một m-file
- Lệnh `error` hiển thị một lời nhắn trên màn hình và dừng hẳn chương trình.
- Hàm `warning` hiển thị một lời nhắn lên màn hình tuy nhiên không dừng chương trình
- Các lệnh `pause` hoặc `keyboard` có thể dùng để tạm dừng chương trình. Để thoát khỏi chế độ gỡ lỗi (debug-mode) và tiếp tục chương trình dùng một trong các lệnh `return, dbcont, dbquit`.

Gỡ lỗi

Sử dụng lệnh keyboard

```
function r = quadroot(a,b,c)
% quadroot Roots of quadratic equation and demo of keyboard command
%
% Synopsis: r = quadroot(a,b,c)
%
% Input: a,b,c = coefficients of a*x^2 + b*x + c = 0
%
% Output: r = column vector containing the real or complex roots
d = b^2 - 4*a*c;
if d<0
  fprintf('Warning in function QUADROOT:\n');
  fprintf('\tNegative discriminant\n\tType "return" to continue\n');
  keyboard;
end
q = -0.5*( b + sign(b)*sqrt(b^2 - 4*a*c) );
r = [q/a; c/q]; % store roots in a column vector
```

Nội dung

(The Profiler)

MatLab phiên bản 5.0 hoặc mới hơn cung cấp một công cụ gọi là "profiler" hỗ trợ việc xác định các đoạn tắc nghẽn (bottlenecks) trong chương trình. Xét chương trình

```
function result=example1(Count)
for k=1:Count
    result(k)=sin(k/50);
    if result(k) < -0.9
        result(k)=gammaln(k);
    end
end
end
```

Để phân tích chương trình, trước hết dùng các lệnh sau để khởi động "profiler" và xóa tất cả các dữ liệu cũ

```
>> profile on
>> profile clear
```

(The Profiler)

Bây giờ, chạy thử chương trình

```
>> example1(50000);
```

Sau đó, nhập vào lệnh

```
>> profile report
```

Profiler tạo một thông báo dạng HTML về chương trình và khởi tạo một cửa sổ trình duyệt. Tùy theo từng hệ thống máy tính mà các kết quả có thể hiển thị ở các dạng khác nhau.

Tiền cấp phát bộ nhớ cho mảng

Các biến ma trận trong MatLab có khả năng điều chỉnh số hàng và số cột một cách linh động. Ví dụ

```
>> a=2
a =
     2
>> a(4,4)=1
a =
     2     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     1
```

MatLab tự động điều chỉnh kích cỡ của ma trận. Do đó, bộ nhớ dành cho dữ liệu ma trận cần phải được tiền cấp phát với cỡ lớn.

Tiền cấp phát bộ nhớ cho mảng

Ví dụ 21

Xét đoạn mã

```
a(1)=1;
b(1)=0;
for k=2:8000
    a(k)=0.99803 * a(k-1) -  0.06279 * b(k-1);
    b(k)=0.06729 * a(k-1) + 0.99803 * b(k-1);
end
```

Thời gian thực thi đoạn mã trên là 0.147 giây.

Tiền cấp phát bộ nhớ cho mảng

Sau khi vòng lặp `for` kết thúc kích thước của hai mảng `a,b` đều là 10000. Do đó, để tiền cấp phát bộ nhớ, tạo ra hai vector hàng `a,b` với 10000 phần tử 0:

```
a = zeros(1,10000);
b = zeros(1,10000);
a(1) = 1;
b(1) = 0;
for k = 2:10000
    a(k) = 0.99803 * a(k-1) -  0.06279 * b(k-1);
    b(k) = 0.06729 * a(k-1) + 0.99803 * b(k-1);
end
```

Với sự thay đổi này, thời gian thực thi chỉ còn là 0.005 giây (nhanh hơn gần 3 lần).

Giới hạn một giá trị mà không dùng cấu trúc `if`

Để giới hạn một giá trị trong một khoảng cho trước, một cách trực tiếp để lập trình là

```
if x < lowerBound
x = lowerBound;
elseif x > upperBound
x = upperBound;
end
```

Tuy nhiên, cách này thực thi rất chậm. Một phương pháp nhanh hơn đó là dùng các hàm `min` và `max`

```
x = max(x,lowerBound); % Clip elements from below, x >= lowerBound
x = min(x,upperBound); % Clip elements from above, x <= upperBound
```

Hơn nữa đoạn mã này tác động lên từng từ trong trường hợp x là ma trận bất kỳ.

Chuyển một mảng bất kỳ thành vector cột

Trong nhiều trường hợp ta sẽ phải chuyển một mảng bất kỳ thành một ma trận cột, ví dụ khi yêu cầu đối với dữ liệu đầu vào của một hàm phải là một vector cột. Câu lệnh sau sẽ chuyển một mảng bất kỳ bao gồm một vector hàng, một ma trận hay một vector cột thành một vector cột

```
x = x(:); % convert x to a column vector
```

Bằng cách dùng lệnh trên cùng với phép chuyển vị `.'`, ta có thể chuyển một mảng bất kỳ về một vector hàng.

Chuẩn hóa vector

Để chuẩn hóa một vector `v`, ta có thể sử dụng lệnh

```
v = v/norm(v)
```

Tuy nhiên, để chuẩn hóa một tập các vector `v(:,1)`, `v(:,2)`,... đòi hỏi phải tính `v(:,k)/norm(v(:,k))` trong một vòng lặp `for` hoặc đoạn mã vector hóa

```
vMag = sqrt(sum(v.^2));
v = v./vMag(ones(1,size(v,1)),:);
```

Tốc độ thực hiện của đoạn mã vector hóa nhanh hơn đáng kể so với việc dùng vòng lặp `for`. Ví dụ, với vài ngàn vector có độ dài 3, cách tiếp cận vector hóa nhanh hơn khoảng 10 lần.