Matlab Programming

K. Cooper

2012

Repetitive Operations

We can use `sum()` to sum vector elements.

Repetitive Operations

We can use `sum()` to sum vector elements.

`v=1:5;`
`sum(v)` gives 15

Repetitive Operations

We can use `sum()` to sum vector elements.

`v=1:5;`
`sum(v)` gives 15

Of course, we could also do
`u=ones(5,1); v*u` to get the 15.

Repetitive Operations

We can use `sum()` to sum vector elements.

`v=1:5;`
`sum(v)` gives 15

Of course, we could also do
`u=ones(5,1); v*u` to get the 15.

There is also a `prod()` function to evaluate the product of the elements of the columns of a matrix.

For loops

```
for var=[range vector] expressions; end;
```

For loops

```
for var=[range vector] expressions; end;
```

We could do the sum from the previous slide as

```
total=0; for i=1:5 total = total+i; end;
```

For loops

```
for var=[range vector] expressions; end;
```

We could do the sum from the previous slide as

```
total=0; for i=1:5 total = total+i; end;
```

or again

```
total=0; for i=[1,2,3,4,5] total = total+i; end;
```

For loops

```
for var=[range vector] expressions; end;
```

We could do the sum from the previous slide as

```
total=0; for i=1:5 total = total+i; end;
```

or again

```
total=0; for i=[1,2,3,4,5] total = total+i; end;
```

While loops

```
while condition expressions; end;
```

While loops

```
while condition expressions; end;
```

We could do the sum from two slides ago as

```
total=0; i=1; while i<=5 total=total+i; i=i+1; end;
```

While loops

```
while condition expressions; end;
```

We could do the sum from two slides ago as

```
total=0; i=1; while i<=5 total=total+i; i=i+1; end;
```

Note the extra work as compared with all previous approaches.

While loops

- Do not use while loops when you know beforehand how many iterations you need.
- Do not forget to change the variable you are testing – while loops are the easiest way to put the computer in an infinite repetition.

Types of programs

- One can type sequences of commands on the Matlab command line by using <Shift>+<Enter> to make new lines.
- For more comprehensive programming, create a file of Matlab commands. This file must have a ".m" extension.
- A simple collection of Matlab commands is called a *script*. It shares the variables of a Matlab session.
- A self-contained sequence of commands that have specific input and output is called a *function*.

Scripts

- Scripts are composed of ordinary Matlab commands saved in a file with a ".m" extension.
- Scripts have access to all variables from the Matlab session, and can modify them.
- The Matlab session has access to all variables created by the script.
- A script is simply a way to save a sequence of commands so it can be reused.
- A script is called by typing the name of the file it is stored in, without the ".m" extension.

Functions

- A function must start with a special line
  `function output=name(arguments)`
- `output` is some matrix containing variables the Matlab session will have access to. All other variables in the function are private.
- `name` is the name of the function, and the function must be stored in a file named `name.m`
- `arguments` is some list of variables that are input from the calling line in the Matlab session: e.g. `x,y`
- This function is called as `name(x,y)`

Where do I save it?

- Save the ".m" file in the directory where you will run Matlab.
- The `pwd` command lets you see the path of the directory you are currently in
- The `ls` command allows you to see the contents of your current directory
- Advanced users can change the `MATLABPATH` variable to specify where they want to save ".m" files using the `path` command:
  `path(path,'newdirectory')`
- You can look at the current MATLABPATH using the `matlabpath` command

Arguments

- Ordinarily there will be a one-to-one correspondence between the arguments in the calling statement and the arguments on the `function` line.
- If there are more arguments on the `function` line than in the calling statement, that is acceptable. Just test the arguments in the function.
- It is an error to have more arguments in the calling statement than on the `function` line.
- Every function gets the variable `nargin` that tells how many arguments there are.

Example

```
function [ output_args ] = mytest( x1,x2,x3)

if nargin>0, x1
end
if nargin > 1, x2
end
if nargin > 2, x3
end

end
```

In this case the call `mytest('One argument')` works. Note that this function is saved in the file `mytest.m`.

Cell Arrays

- In general the output argument will be a matrix; possibly 1×1 or $1 \times n$.
- If you need to send out multiple items of different types (e.g. numeric and character) you can use a *cell array*.
- Cell arrays are indexed by integers using braces $\{ \}$
- Cell arrays can contain data of any type or dimension.
- E.g. `ca{1} = 1; ca{2}=[1,2;3,4]; ca{3}='Hello';`
- The argument list of a function can take a variable **`varargin`** that is a cell array containing all the rest of the arguments.

Example

This simple function takes any set of plot options you specify and applies them to a plot of the sine function from 0 to π.

Example

This simple function takes any set of plot options you specify and applies them to a plot of the sine function from 0 to π.

```
function out = sineplot(varargin)
    x = 0:.1:pi;
    out{1}=[x;sin(x)];
    out{2}=plot(x,out{1}(2,:),varargin{:});
end
```

It can be called as e.g.

```
sineplot('gd','LineWidth',3,'MarkerFaceColor',
'red','MarkerSize',15)
```

Example

This simple function takes any set of plot options you specify and applies them to a plot of the sine function from 0 to π.

```
function out = sineplot(varargin)
    x = 0:.1:pi;
    out{1}=[x;sin(x)];
    out{2}=plot(x,out{1}(2,:),varargin{:});
end
```

It can be called as e.g.

```
sineplot('gd','LineWidth',3,'MarkerFaceColor',
'red','MarkerSize',15)
```

It returns handles to the data, and to the plot data series. One could use those to modify the plot further.