

A gentle introduction to Matlab

The "Mat" in Matlab does not stand for "mathematics", but for "matrix"..

⇒ all objects in matlab are matrices of some sort! Keep this in mind when using it.

Matlab is a high level *interpreted* programming language:

- ▶ a matlab program is typically a set of instructions that are evaluated iteratively;
- ▶ most of the work can be done directly from the command line.

“;” hides the result on the command line

Remark that

```
>> f(0.2,3.2)
```

```
ans =
```

```
0.5120
```

but

```
>> f(0.2,3.2);
```

produces no output.

Defining a function

We want to plot the iterates of some function f . First, we define the function.

```
>> f=inline('r.*x.*(1-x)', 'x', 'r')
```

```
f =
```

```
Inline function:
```

```
f(x,r) = r.*x.*(1-x)
```

This defines a function (here, with two arguments, x and r), that can then be used:

```
>> f(0.2,3.2)
```

```
ans =
```

```
0.5120
```

p. 1 Computing iterates

p. 2

Creating a vector

To create a vector, use the command

$x =$ first entry : step : last entry,

or, if entries are a subset of the integers,

$x =$ first entry : last entry.

For example, we want to plot the iterates of the logistic map, so

```
x=0:0.01:1;
```

Note the “;”: otherwise, we get the full 101 elements vector displayed.

What is the size of .. ?

As mentioned, in matlab everything is a matrix. For matrix operations, size is important, and it is frequent to make mistakes. To check, whos and size. whos gives a lot of information.

```
>> whos x
  Name      Size      Bytes  Class

  x         1x101      808    double array
Grand total is 101 elements using 808 bytes
```

Various variables can be listed on the line after whos:

```
>> whos x k
  Name      Size      Bytes  Class

  k         1x1        8    double array
  x         1x101      808    double array
Grand total is 102 elements using 816 bytes
```

size

size, on the other hand, is "attributable". It can be used like this

```
>> size(x)
```

```
ans =
     1    101
```

but also like this, since the result is a vector

```
>> [r,c]=size(x)
```

```
r =
     1
```

```
c =
    101
```

in which case, r and c take the values of the numbers of rows and columns, respectively.

Vectorized functions versus nonvectorized functions

Recall that we wrote

```
>> f=inline('r.*x.*(1-x)','x','r')
```

that is, every multiplication sign took the form `.*` instead of `*`. Here, this is needed: we want to use the *vectorized* form of the function, and be able to pass to f a vector instead of a single value. The `.*` form means that the operation is applied to every entry in the vector/matrix. Same exists for `/` and `^`. Can also use the function `vectorize`.

The result of using this vectorized form is that f will be applied to every entry of x , and will produce a vector.

Vectorized operations have been optimized in matlab, and are extremely fast. When possible, they should be used instead of loops.

Vectorized vs nonvectorized

Define

```
>> f=inline('r.*x.*(1-x)','x','r')
>> g=inline('r*x*(1-x)','x','r')
```

and for simplicity, consider the vector

```
>> x=[1,2];
```

Then

```
>> f(x,3.5)
g(x,3.5)
ans =
     0    -7
```

```
??? Error using ==> inlineeval
Error in inline expression ==> r*x*(1-x)
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Plotting

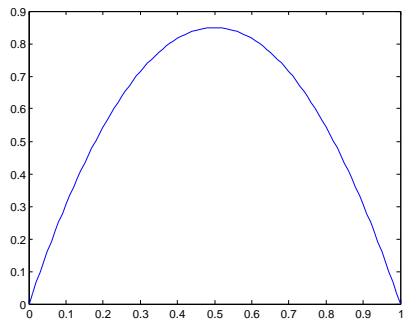
Basic plotting is very easy. The format is

```
plot(x_axis,y_value)
```

so, for example (with f as defined above),

```
plot(x,f(x,3.4))
```

(here, ";" or not does not matter, as the figure appears in a new window and all that ";" changes is the output in the command window).



Making things a bit more fancy

This is a very basic plot.

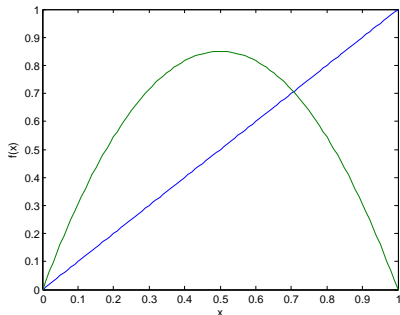
- ▶ We could want to plot more than one object (for example, the line $y = x$ would be nice)..

```
plot(x,x,x,f(x,3.4));
```

Ordering is by pairs: $x_1, f_1(x_1), x_2, f_2(x_2)$. Two elements in a pair **must have** the same number of columns. Different pairs **can have** different numbers of columns. Each element in a given pair can be a point, a vector, a matrix.

- ▶ We could want to label the axes..

```
xlabel('x');  
ylabel('f(x)');
```



For the moment, we only have $f(x)$. We want $f^n(x)$, for a given n . Several ways.

- ▶ Taking for example $r = 3.5$, use
`f(f(x,3.5),3.5)`
- ▶ The downside to this method is that matlab does not allow to formally define f^n , so tricks have to be used for larger values of n , for example, produce a string containing the command
`f(f(f(f(x,3.5),3.5),3.5),3.5),3.5)`
 and evaluate it. Complicated..
- ▶ Another method consists in using the result found at the previous step to evaluate the next. We do that..

We are going to use a very nice feature of matlab: adding elements to a vector, or rows/columns to a matrix, is automatic. Suppose for example that we had defined x as

```
x=0:0.01:0.5;
```

Then

```
x=[x,0.51:0.01:1];
```

would produce the vector x as we had earlier.

Be careful! Note that the command was

```
x=[x,0.51:0.01:1];
```

that is, the old and new entries were separated by a “,”. This is *horizontal concatenation*. The command with a “;” tries to add a new row. In our case, we get

```
>> z=[z;0.51:0.01:1]
```

```
??? Error using ==> vertcat
```

All rows in the bracketed expression must have the same number of columns.

because we are trying to add a row of 50 elements to a row of 51 elements. But

```
>> z=[z;0.51:0.01:1.01]
```

works, and gives a 2×51 matrix.

Here, we are going to use the latter form of the command, and add each successive iterate to a solution matrix M .

First, define an empty matrix,

```
M=[];
```

Then we need to loop from 1 to n , where n is the iterate that we want.

Loops

The command uses the same type of syntax as the creation of a vector: to loop from 4 to 12 by steps of 1,

```
for i=4:12,
    command(s) to be repeated, maybe using the value i
end;
```

whereas to loop by non-unit or non-integer steps, say from 4 to 12 by steps of 1.35,

```
for i=4:1.35:12,
    command(s) to be repeated, maybe using the value i
end;
```

Note that in that case, the last i is equal to 10.75, not 12, since $10.75 + 1.35 = 12.1 > 12$. The same is true when using non-unit steps to create vectors.

Accessing matrix elements

Suppose that M is an $m \times n$ -matrix. Then

- ▶ $M(i, j)$ is the element on the i th row and j th column.
- ▶ $M(i, :)$ is the i th row.
- ▶ $M(:, j)$ is the j th column.
- ▶ $M(\text{end}, :)$ is the last row of M (`end` is a reserved word which always points to the last valid index in a given matrix dimension).
- ▶ $M(:, \text{end})$ is the last column of M .
- ▶ $M(\text{end}, 1:10)$ are the first 10 entries in the last row of M .
- ▶ $M(1:2, 3:5)$ is the submatrix of M consisting of rows 1 and 2 and columns 3 to 5 of M .

Back to the iterates

After some thought, we realize that we will need to go back one iterate. So instead of starting with empty matrix M , fill the first row of M with first iterate, and start at iterate 2.

```
n=10;
r=3.5;

M=f(x, r);

for i=2:n,
    M=[M; f(M(end, :), r)];
end;

plot(x, M);
```

This plots all the iterates to n . A bit crowded..

