Vectors
Matrices
Arrays.

Matrix,vector,scalar

Originally MATLAB's one and only data structure was an ordinary 2-dimensional matrix consisting of complex numbers.

Special cases:

- Column vector: `(m,1)`-matrix
- Row vector: `(1,n)`-matrix
- Scalar: `(1,1)`-matrix
- Empty: `(m,0)` or `(0,n)`-matrix

```
>> A=[1 2 3 4 ;5 6 7 8; 9 10 11 12]
>> [m,n]=size(A) % Matrix knows its size
>> v=-[1 2 3 4 ]
>> length(v), size(v)
>> 1:10      % [1,2,3,...,10]
>> size(ans) % ans:previous non-assigned result
>> who, whos % workspace variables
```

Transpose

Continue previous session:

```
>> A                      % Look into A.
>> A'                     % Transpose A
>> C=A+i*ones(size(A)) % Complex matrix
>> C'         % In addition complex conjugation
>> C.'                    % Without conjugation
>> rowvect=[1 2 3 4]
>>      % blank or (,) means: 'put beside me'
>> colvect=[1;2;3;4]   % (;) means: 'put under me'
>> colvect' % Another way for column vector(if real)
```

Note: If `A` has complex entries, `A’` contains their complex conjugates. If you want to avoid that, use `A.’`

Functions for building vectors colon(:),linspace,logspace

- `v=a:b, w=a:h:b;` default: h=1
- `v=linspace(a,b,N);` `default: N=100`
- `v=logspace(a,b,N);` $10^a, \ldots, 10^b$, `N points`
  `default: N=50`

```
>> 0:10; 0:.1:1;
>> 10:-2:0
  ans =
    10     8     6     4     2     0
>> logspace(0,1,4)
   ans =
    1.0000    2.1544    4.6416   10.0000
>> 10.^linspace(0,1,4)
   ans =
    1.0000    2.1544    4.6416   10.0000
```

Calculus with vectors

```
>> u = 0:0.1:10;  % Remember semicolon (;)
>> length(u)
ans =
   101
>> w = 5*sin(u); %
>> [u(1:10)' w(1:10)']
```

Alternatively, by 'misuse' of Matlab:

```
>> for k=1:length(u)
     z(k)=5*sin(u(k));
   end;
```

Vector excercise

Make the following variables:

- aVec $= \begin{bmatrix}3.14 & 15 & 9 & 26\end{bmatrix}$
- bVec $= \begin{bmatrix}2.71 \\ 8 \\ 28 \\ 182\end{bmatrix}$
- cVec$= [5, 4.8, \ldots, -4.8, -5]$ (all the numbers from 5 to -5 with increments of -0.2)
- dVec $= [10^0\, 10^{0.01}\, \ldots 10^{0.99}\, 10^1]$ logarithmically spaced numbers between 1 and 10. (first without `logspace`, then `help logspace`)
- eVec = `'Hello there'` (String is a vector of characters) `['Hello ' 'there']` Concatenate, need brackets. `n=4;figure;title(['Test nr.  '  num2str(n)])`

“Scalar functions” support vectorization

The previous example leads us to the following general idea:

- Functions which applied to a scalar produce a scalar result are called *scalar functions*. When such functions are applied to a vector, they **operate on every element of the vector**.
- Mathematical functions among others are of this type. (`help elfun`, `help specfun`)

Scalar functions and pointwise arithmetic support vectorization

Assume we want to compute values of

$$y = e^{-x} \sin x$$

at a vector x. We need the vector
$y = (e^{-x(1)} \sin(x(1)), e^{-x(2)} \sin(x(2)), \ldots, e^{-x(n)} \sin(x(n)))$
Here we need the **pointwise product** `(.*)` of two vectors:

```
>> x=-pi:.1:pi;
>> y=exp(-x).*sin(x);
```

This is just the data we need for plotting. `>> plot(x,y)`

Creating matrices

- Square brackets `[...]` to define matrices
- Spaces (and/or commas) to separate columns (elements of row vector).
- Semi-colons (;) to separate rows (elements of column vector)
- `>> [ 3 4 5 ; 6 7 8 ]` is a 2-by-3 matrix
- If A and B are matrices with the same number of rows, then `>> C = [ A B ]` is the array formed by stacking A and B side by side

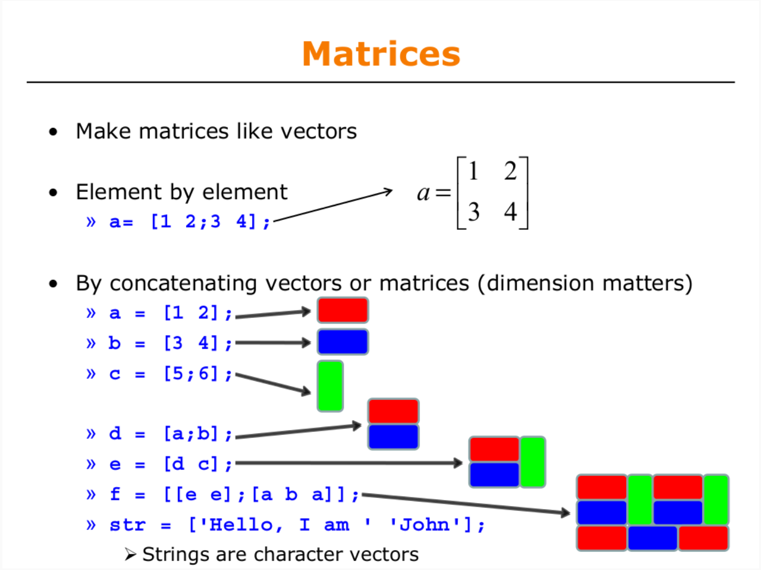
```
 >> A=ones(2,2);B=2*ones(2,3);[A B]
ans =
     1     1     2     2     2
     1     1     2     2     2
```

Creating matrices, continued

- If A and B are matrices with the same number of columns, then `[ A ; B ]` is the matrix formed by stacking A on top of B.
- So, `[[ 3 ; 6 ] [ 4 5 ; 7 8 ] ]` is equal to `[ 3 4 5;6 7 8 ]`

Next slide from: MIT's Matlab-course
illustrates the building of arrays out of smaller pieces.

Matrices, visual summary [MIT: open courseware]



Matrices

- Make matrices like vectors
- Element by element

```
» a= [1 2;3 4];
```

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- By concatenating vectors or matrices (dimension matters)

```
» a = [1 2];
» b = [3 4];
» c = [5;6];

» d = [a;b];
» e = [d c];
» f = [[e e];[a b a]];
» str = ['Hello, I am ' 'John'];
```

  - ➢ Strings are character vectors

Some functions for building matrices

`eye,vander,hilb,zeros,ones,diag,rand,reshape,magic`

Complete list: `help elmat`

```
>> A = zeros(2,5)
>> B = ones(3)    % or ones(3 ,3)
>> R = rand(3,2)
>> N = randn(3,2)
>> I = eye(3)      % or eye(3,3)
>> D = diag(-2:2)
```

Compare `rand` and `randn` Try repeatedly

`>> R = rand(3,2)` Use (↑) in command window

Repeat : `>> rng('default'); R = rand(3,2)` or

`>>s=rng; R = rand(3,2); rng(s);rand(3,2)`

Matrices: reshape

reshape

- Forms a matrix of given size for given data.
- Data will be placed in “frame” of given size in **column order**. (Matlab is column oriented.)
- Nr. of datapoints (`numel(data)`) has to match product of dimensions.

```
>> A=reshape(1:6,2,3)  % 2x3 matrix from data 1:6 ...
    in column order
>> B=reshape(1:6,3,2)' % Row-order
>> C=reshape(A,1,6)    % Back to vector 1:6
```

Visualization of matrices

Have fun with some commands of type:

```
>> mesh(ones(30));hold on;mesh(zeros(30));
>> mesh(eye(30));shg; hold off
>> imagesc(diag(-5:5)),colorbar;shg
>> surf(magic(10));colorbar;shg
>> surfc(vander(-1:.1:1));colorbar;shg
>> imagesc(reshape(0:24,5,5)),colorbar
```

Modify some parameters, and try to see what kind of matrices the visualizations reveal to you.
In the figure-window you can click the *"rotate-arrow"* and rotate your figure with the mouse.
Task: Explain:
`V=vander(-1:.1:1);plot(V(:,end-1),V);help vander`

"Vector functions" (also support vectorization)

- Let vector function f mean that `f(vector)=scalar`. Such function operate on a matrix columnwise returnig a vector whose length is the length of rows (= nr. of columns). Examples are many that are classified under *datafunction*, `help datafun`.
- Example: `max,min,mean,sum,....` Also many functions of type `f(vector)=vector` behave similarly, except they applied to a matrix return a matrix "columnwise". Examples: `sort,cumsum,cumprod, ...`
- These functions can be called to operate rowwise: for example `min(A,2)` forms a column vector of row-minimums. Similarly for higher-dimensional arrays as well.

Selected “datafuns”, summary

- `min,max,sum,prod,mean,std,diff,del2`
  These functions **return scalar** result **for vectors**, and operate **columnwise** or **row-wise** for **matrices, returnig a vector**.
- `cumsum,cumprod,sort`
  are examples that return a matrix of the same size as input.
- More: `help datafun`

Exercise on min,max,sum

1. How do you compute the min and max of all elements of a matrix (two ways)?
2. l_1 - norm and l_∞ - norms of a matrix A are:

$$||A||_1 = \max_{1\le j\le n} \sum_{i=1}^{n} |a_{ij}|, \quad ||A||_\infty = \max_{1\le i\le n} \sum_{j=1}^{n} |a_{ij}|$$

   Form MATLAB-expressions for computing these.
3. Test with suitable matrices like: `A=randi(5,5)`, `A=magic(6,6)`, `A=reshape(1:16,4,4)`, `....` Check using Matlab's `norm`-function.

Matrix- and array algebra

Often one needs to evaluate the “pointwise” product of two vectors, as seen already in some examples.

```
u=1:5, v=[1 -1 1 -1 1], w=u.*v
x=linspace(0,2*pi,20); y=x.*sin(x);
plot(x,y), shg
```

More generally, matrices in MATLAB have two products:
- The ordinary matrix product
- Pointwise product

```
 A=reshape(1:9,3,3), B=2*ones(size(A)), C=A.*B, D=A*B
 [A B], [C D]
```

The next slide shows the general outline.

Matrix- and array algebra

`A, B` matrices, matching size, `c` scalar.

Matrix algebra

- `A + B, A+c`
- `A*B` matrix product
- `A'` (conjugate) transpose
- `A.'` transpose without conjugation
- `A^p (A*A*...A)` Matrix power (A square matrix.)
- `A\b`
  $Ax = b \iff x = A\backslash b$ (if A is invertible)

Array algebra

- `A + B, A+c`
- `A.*B` Pointwise product
- `A.^p, A.^B` Pointwise power, p scalar, A and B of same size.
- `A./B, c./A` Pointwise divide. Subtle `1.0/A, 1.0./A,1./A`
- **Note:** `c/A` usually leads to an error.

Indexing: Accessing single element of a vector

If A is a vector, then

- A(1) is its first element
- A(2) is its second element
- . . .
- A(end) is its last element

For matrices either *columnwise linear indexing*, or

- A(1,1) is the element on the first row of the first column
- A(2,1) is the element on the second row of the first column
- A(3,4) is the element on the third row and fourth column
- A(4,end) is the last element of the fourth row

Example

```
>> A = [3 4.2 -7 10.1 0.4 -3.5 ];
>> A(3)
>> A(4) = log(8);
>> A
>> A(end)
>> A(end+1)       % Error
>> A(end+1)=-1    % Matlab extends A
```

Indexing: Accessing multiple elements of a matrix

Index need not be a single number – you can index with a vector.

```
A = [ 3  4.2  -7  10.1  0.4  -3.5 ]
A([1 4 6]) % 1-by-3, 1st, 4th, 6th entry
Index = [3 2 3 5];
A(Index) % 1-by-4
A([4 3 2 1])  % Reverse order
A([1 2 2 3 3 3 4 4 4 4])  % Index vector can have ...
    repetitions.
```

Index should contain integers. Shape of the index will define the shape of the output matrix.

Exercise

Using MATLAB indexing, compute the perimeter sum of the matrix "`magic(8)`".

Perimeter sum adds together the elements that are in the first and last rows and columns of the matrix. Try to make your code independent of the matrix dimensions using `end`.

You can do it at least in 2 ways of thought:
1. Pick the rows and parts of the columns that exclude the corner points, and use `sum` suitably.

2. Embed a zero-matrix of size $n-2 \times n-2$ inside magic(n) (n=8) to produce, say Abd. Then do the summation either sum(sum(...)) or sum(Abd(:))

Answer: 910

Exercise (Perhaps skip for now)

- 1. Use `reshape` to form the matrix A:

```
1     2     3     4
5     6     7     8
9    10    11    12
```

Hint: start with a 4×3-matrix and transpose it.

- 2. Find the size `[m,n]` of A (pretend, you just forgot it.)
- 3. Embed A into a zero-matrix AA of size $(m+2) \times (n+2)$
- 4. Replace the first row of AA by $[1, 2, \dots m+2]$ and the last row with $[m+2, m+1, \dots 1]$.**Hint:** step -1, or `fliplr`
- 5. replace the m “inner entries (zeros)” `(2,...,end-1)` of the first column of AA by -1's

Linear Systems

Linear systems of equations

Given the system of equations:

$$\begin{cases} 6x + 12y + 4z = 70 \\ 7x - 2y + 3z = 5 \\ 2x + 8y - 9z = 64 \end{cases}$$

Solve it!

```
>> A=[6 12 4;7 -2 3;2 8 -9]
>> b=[70;5;64];
>> x=A\b; x'
ans =
      3      5     -2
```

Linear systems of equations, continued

```
>> [A*x b] % Check by multiplication:
ans =
    70    70
     5     5
    64    64

>> b=[70;5;64];
>> x=A\b; x'
ans =
     3     5    -2
>> x=inv(A)*b % Alternatively multiply by inverse
```

- Backslash \ is recommended for efficiency and accuracy.
- Linear systems don't always have a unique solution.
- `det(A)==0` is not a numerically reliable way of testing "almost singularity". See `help cond, rcond`.

Excercises

Solve the system of equations

$$\begin{cases} 2x + y = 3 \\ x - 2y = -1 \end{cases}$$

using the “backslash” operator, and check the result.

Using the same technique, solve below system, and check result.

$$\begin{cases} 35x_1 + 0x_2 + 14x_3 + 16x_4 + 2x_5 = 67 \\ 27x_1 + 7x_2 + 14x_3 + 4x_4 + -7x_5 = 45 \\ -13x_1 - 2x_2 + 6x_3 + 10x_4 + 8x_5 = 9 \\ 30x_1 - 1x_2 - 12x_3 + 7x_4 - 11x_5 = 13 \\ 7x_1 + 14x_2 + 7x_3 - 3x_4 - 10x_5 = 15 \end{cases}$$

Basics of Graphics

Basic 2d-graphics, plot

- *"Matlab has excellent support for data visualization and graphics with over 70 types of plots currently available. We won't be able to go into all of them here, nor will we need to, as they all operate in very similar ways. In fact, by understanding how Matlab plotting works in general, we'll be able to see most plot types as simple variations of each other. Fundamentally, they all use the same basic constructs."*
- Links:
  - https://se.mathworks.com/help/matlab/ref/plot.html
  - http://ubcmatlabguide.github.io/html/plotting.html

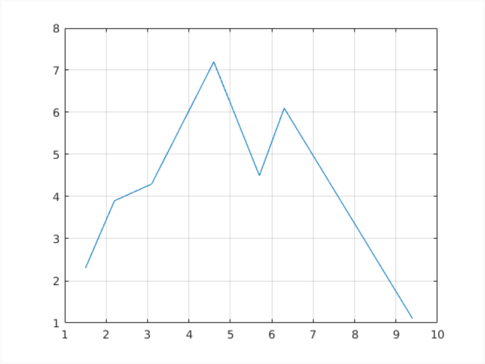
Basic 2d-graphics

- If `x` is a 1-by-N (or N-by-1) vector, and `y` is a 1-by-N (or N-by-1) vector, then
  ```
  >> plot(x,y)
  ```
  creates a figure window, and plots the data points with joining line segments in the axes. The points are:
  `(x(1),y(1)), (x(2),y(2)),..., (x(N),y(N))`
- The axes are automatically chosen so that all data just fits into the figure window. This can be changed by the `axis, xlim, ylim`-commands.

Basic 2d-graphics, plot

Function *plot* can be used for simple "join-the-dots" xy-plots.

```
>> x=[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y=[2.3 3.9 4.3 7.2 4.5 6.1 1.1];
>> plot(x,y);grid on
```



Basic 2d-graphics, general form

Continue keeping the previous plot:

```
>> hold on          % Keep the previous lines.
>> plot(x,y,'or')   % Mark datapoints with ...
    'o'-marker, r='red'
>> shg              % show graphics
```

- General form:

  `plot(x1,y1,'string1',x2,y2,'string2', ...)`

  The `'string'`-parts may be missing.
- `plot(x,y,’r*--’)`

  Use red *-markers, join with red dashed line segments.

help plot -> table of markers

Various line types, plot symbols and colors: plot(X,Y,S)

S is a character string made from one element from any or all of the following 3 columns:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p,h	pentagram, hexagram		

Plotting graphs of functions

Just take enough points to get smoothness.

```
>> x=linspace(0,3*pi);  % Default: 100 points
>> y=sqrt(x).*sin(x);   % Note again: (.*)
>> plot(x,y)
>> figure  % Open a new graphics window.
>> x1=linspace(0,pi,1000);  % More points.
>> y1=cos(4*x1).*sin(x1);
>> m=mean(y1);
>> plot(x1,y1,[0 pi],[m m],'r--') % "red" dashed
>> legend('Function','mean'); grid on
```

Refrences in Finnish:

http://math.aalto.fi/∼apiola/matlab/opas/mini/vektgraf.html

http://math.aalto.fi/∼apiola/matlab/opas/lyhyt/grafiikka.html

Excercise

Let's do some plotting. Do the following:

a) Graph the function $f(x) = sin(x)$ on the interval $x \in [0, 1]$. Try changing the plot colour, and observe your discretization by using different plotting styles.

b) Graph the function $f(x) = \frac{1}{4}x\sin(x)$ on the interval $x \in [0, 40]$ in the same plot with $y_1 = \frac{1}{4}x$ and $y_2 = -\frac{1}{4}x$. Plot the lines with red dashes, and change the line width of f to 3.

c) Plot a curve with x coordinate of $\cos(t)$ and y coordinate of $\sin(t)$ when $t \in [0, 2\pi]$.

d) Plot a parametric curve

$$\begin{cases} x = \cos(t)\left(e^{\cos(t)} - 2\cos(4t) - \sin\left(\frac{t}{12}\right)\right) \\ y = \sin(t)\left(e^{\cos(t)} - 2\cos(4t) - \sin\left(\frac{t}{12}\right)\right) \end{cases}$$

Some plots will probably not look like you expect: try using `axis equal` or `axis square`.